API Data Formats

jim stafford

Fall 2024 v2022-10-02: Built: 2024-11-19 21:32 EST

Table of Contents

Chapter 1. Introduction

Web content is shared using many standardized MIME Types. We will be addressing two of them here

- XML
- JSON

I will show manual approaches to marshaling/unmarshalling first. However, content is automatically marshalled/unmarshalled by the web client container once everything is set up properly. Manual marshaling/unmarshalling approaches are mainly useful in determining provider settings and annotations — as well as to perform low-level development debug outside the server on the shape and content of the payloads.

1.1. Goals

The student will learn to:

- identify common/standard information exchange content types for web API communications
- manually marshal and unmarshal Java types to and from a data stream of bytes for multiple content types
- negotiate content type when communicating using web API
- pass complex Data Transfer Objects to/from a web API using different content types
- resolve data mapping issues

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. design a set of Data Transfer Objects (DTOs) to render information from and to the service
2. define Java class content type mappings to customize marshalling/unmarshalling
3. specify content types consumed and produced by a controller
4. specify content types supplied and accepted by a client

Chapter 2. Pattern Data Transfer Object

There can be multiple views of the same conceptual data managed by a service. They can be the same physical implementation — but they serve different purposes that must be addressed. We will be focusing on the external client view (Data Transfer Object (DTO)) during this and other web tier lectures. I will specifically contrast the DTO with the internal implementation view (Business Object (BO)) right now to help us see the difference in the two roles.

2.1. DTO Pattern Problem Space

Context

Business Objects (data used directly by the service tier and potentially mapped directly to the database) represent too much information or behavior to transfer to remote client

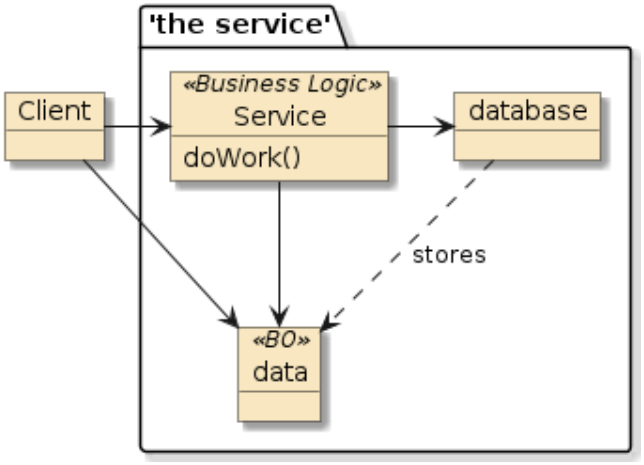

Figure 1. Clients and Service Sharing Implementation Data

Problem

Issues can arise when service implementations are complex.

- client may get data they do not need
- client may get data they cannot handle
- client may get data they are not authorized to use
- client may get too much data to be useful (e.g., entire database serialized to client)

Forces

The following issues are assumed to be true:

- some clients are local and can share object references with business logic
- handling specifics of remote clients is outside core scope of business logic

2.2. DTO Pattern Solution Space

Solution

- define a set of data that is appropriate for transferring requests and responses between client and service
- define a Remote (Web) Facade over Business Logic to handle remote communications with the client
- remote Facade constructs Data Transfer Objects (DTOs) from Business Objects that are appropriate for remote client view
- remote Facade uses DTOs to construct or locate Business Objects to communicate with Business Logic

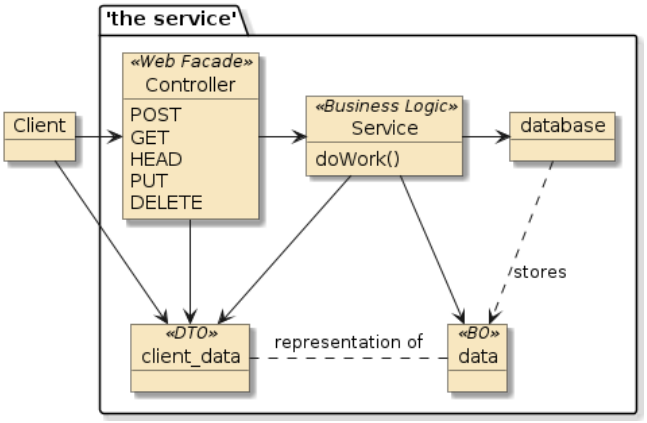

Figure 2. DTO Represents Client View of Data

DTO/BO Mapping Location is a Design Choice

The design decision of which layer translates between DTOs of the API and BOs of the service is not always fixed. Since the DTO is an interface pattern and the Web API is one of many possible interface facades and clients of the service — the job of DTO/BO mapping may be done in the service tier instead.

2.3. DTO Pattern Players

Data Transfer Object

- represents a subset of the state of the application at a point in time
- not dependent on Business Objects or server-side technologies
  - doing so would require sending Business Objects to client
- XML and JSON provide the “ultimate isolation” in DTO implementation/isolation

Remote (Web) Facade

- uses Business Logic and DTOs to perform core business logic
- manages interface details with client

Business Logic

- performs core implementation duties that may include interaction with backend services and databases

Business Object (Entity)

- representation of data required to implement service
- may have more server-side-specific logic when DTOs are present in the design

DTOs and BOs can be same class(es) in simple or short-lived services

DTOs and BOs can be the same class in small services. However, supporting multiple versions of clients over longer service lifetimes may cause even small services to split the two data models into separate implementations.

Chapter 3. Sample DTO Class

The following is an example DTO class we will look to use to represent client view of data in a simple "Quote Service". The `QuoteDTO` class can start off as a simple POJO and — depending on the binding (e.g., JSON or XML) and binding library (e.g., Jackson, JSON-B, or JAXB) - we may have to add external configuration and annotations to properly shape our information exchange.

The class is a vanilla POJO with a default constructor, public getters and setters, and other convenience methods — mostly implemented by Lombok. The quote contains three different types of fields (int, String, and LocalDate). The `date` field is represented using `java.time.LocalDate`.

Example Starting POJO for DTO

```
package info.ejava.examples.svc.content.quotes.dto;

import lombok.*;
import java.time.LocalDate;

@NoArgsConstructor ①
@AllArgsConstructor
@Data ②
@Builder
@With
public class QuoteDTO {
    private int id;
    private String author;
    private String text;
    private LocalDate date; ③
    private String ignored; ④
}
```

① default constructor

② public setters and getters

③ using Java 8, java.time.LocalDate to represent generic day of year without timezone

④ example attribute we will configure to be ignored

Lombok @Builder and @With

@Builder will create a new instance of the class using incrementally defined properties. @With creates a copy of the object with a new value for one of its properties. `@Builder` can be configured to create a copy constructor (i.e., a copy builder with no property value change).

Lombok @Builder and Constructors

@Builder requires an all-args-ctor and will define a package-friendly one unless there is already a ctor defined. Unmarshallers require a no-args-ctor and can be provided using `@NoArgsConstructor`. The presence of the no-args-ctor turns off the required all-args-ctor for `@Builder` and can be re-enabled with `@AllArgsConstructor`.

Chapter 4. Time/Date Detour

While we are on the topic of exchanging data — we might as well address time-related data that can cause numerous mapping issues. Our issues are on multiple fronts.

- what does our time-related property represent?
  - e.g., a point in time, a point in time in a specific timezone, a birthdate, a daily wake-up time
- what type do we use to represent our expression of time?
  - do we use legacy Date-based types that have a lot of support despite ambiguity issues?
  - do we use the newer `java.time` types that are more explicit in meaning but have not fully caught on everywhere?
- how should we express time within the marshalled DTO?
- how can we properly unmarshal the time expression into what we need?
- how can we handle the alternative time wire expressions with minimal pain?

4.1. Pre Java 8 Time

During pre-Java8, we primarily had the following time-related `java.util` classes

Date	represents a point in time without timezone or calendar information. The point is a Java long value that represents the number of milliseconds before or after 1970 UTC. This allows us to identify a millisecond between 292,269,055 BC and 292,278,994 AD when applied to the Gregorian calendar.
Calendar	interprets a Date according to an assigned calendar (e.g., Gregorian Calendar) into years, months, hours, etc. Calendar can be associated with a specific timezone offset from UTC and assumes the Date is relative to that value.

During the pre-Java 8 time period, there was also a time-based library called Joda that became popular at providing time expressions that more precisely identified what was being conveyed.

4.2. java.time

The ambiguity issues with `java.util.Date` and the expression and popularity of Joda caused it to be adopted into Java 8 (JSR 310). The following are a few of the key `java.time` constructs added in Java 8.

Instant	represents a point in time at 00:00 offset from UTC. The point is a nanosecond and improves on `java.util.Date` by assigning a specific UTC timezone. The `toString()` method on `Instant` will always print a UTC-relative value (`1970-01-01T00:00:00.000000001Z`).
OffsetDateTime	adds `Calendar`-like view to an Instant with a fixed timezone offset (`1970-01-01T00:00:00-04:00`).

ZonedDateTime	adds timezone identity to OffsetDateTime — which can be used to determine the appropriate timezone offset (i.e., daylight savings time) (`1969-12-31T23:00:00.000000001-05:00[America/New_York]`, `1769-12-31T23:03:58.000000001-04:56:02[America/New_York]`). This class is useful in presenting current time relative to where and when the time is represented. For example, during early testing I made a typo in my 1776 date and used 1976 for year. I also used ZoneId.systemDefault() ("America/New_York"). The ZoneId had a -04:00 hour difference from UTC in 1976 and a peculiar -04:56:02 hour difference from UTC in 1776. ZoneId has the ability to derive a different timezone offset based on rules for that zone.
LocalDate	a generic date, independent of timezone and time (`1970-01-01`). A common example of this is a birthday or anniversary.
LocalTime	a generic time of day, independent of timezone or specific date. This allows us to express "I set my alarm for 6am" - without specifying the actual dates that is performed (`00:00:00.000000001`).
LocalDateTime	a date and time but lacking a specific timezone offset from UTC (`1970-01-01T00:00:00.000000001`). This allows a precise date/time to be stored that is assumed to be at a specific timezone offset (usually UTC) — without having to continually store the timezone offset to each instance.
Duration	a time-based amount of time (e.g., 30.5 seconds). Vocabulary supports from milliseconds to days.
Period	a date based amount of time (e.g., 2 years and 1 day). Vocabulary supports from days to years.

4.3. Date/Time Formatting

There are two primary format frameworks for formatting and parsing time-related fields in text fields like XML or JSON:

java.text.DateFormat	This legacy `java.text` framework's primary job is to parse a String of text into a Java Date instance or format a Java Date instance into a String of text. Subclasses of `DateFormat` take care of the details and java.text.SimpleDateFormat accepts a String format specification. An example format `yyyy-MM-dd'T'HH:mm:ss.SSSX` assigned to UTC and given a Date for the 4th of July would produce `1776-07-04T00:00:00.000Z`.
java.time.format.DateTimeFormatter	This newer `java.time` formatter performs a similar role to `DateFormat` and `SimpleDateFormat` combined. It can parse a String into `java.time` constructs as well as format instances to a String of text. It does not work directly with Dates, but the `java.time` constructs it does produce can be easily converted to/from `java.util.Date` thru the `Instance` type. The coolest thing about `DateTimeFormatter` is that not only can it be configured using a parsable string — it can also be defined using Java objects. The following is an example of the formatter. It is built using the ISO_LOCAL_DATE and ISO_LOCAL_TIME formats.

Example DateTimeFormatter.ISO_LOCAL_DATE_TIME

```
public static final DateTimeFormatter ISO_LOCAL_DATE_TIME;
static {
    ISO_LOCAL_DATE_TIME = new DateTimeFormatterBuilder()
            .parseCaseInsensitive()
            .append(ISO_LOCAL_DATE)
            .appendLiteral('T')
            .append(ISO_LOCAL_TIME)
            .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
}
```

This, wrapped with some optional and default value constructs to handle missing information makes for a pretty powerful time parsing and formatting tool.

4.4. Date/Time Exchange

There are a few time standards supported by Java date/time formatting frameworks:

ISO 8601	This standard is cited in many places but hard to track down an official example of each and every format — especially when it comes to 0 values and timezone offsets. However, an example representing a ZonedDateTime and EST may look like the following: `1776-07-04T02:30:00.123456789-05:00` and `1776-07-04T07:30:00.123456789Z`. The nanoseconds field is 9 digits long but can be expressed to a level of supported granularity — commonly 3 decimal places for `java.util.Date` milliseconds.
RFC 822/ RFC 1123	These are lesser followed standards for APIs and includes constructs like an English word abbreviation for day of week and month. The DateTimeFormatter example for this group is `Tue, 3 Jun 2008 11:05:30 GMT` [1]

My examples will work exclusively with the ISO 8601 formats. The following example leverages the Java expression of time formatting to allow for multiple offset expressions (`Z`, `+00`, `+0000`, and `+00:00`) on top of a standard LOCAL_DATE_TIME expression.

Example Lenient ISO Date/Time Parser

```
public static final DateTimeFormatter UNMARSHALLER
    = new DateTimeFormatterBuilder()
            .parseCaseInsensitive()
            .append(DateTimeFormatter.ISO_LOCAL_DATE)
            .appendLiteral('T')
            .append(DateTimeFormatter.ISO_LOCAL_TIME)
            .parseLenient()
            .optionalStart().appendOffset("+HH", "Z").optionalEnd()
            .optionalStart().appendOffset("+HH:mm", "Z").optionalEnd()
            .optionalStart().appendOffset("+HHmm", "Z").optionalEnd()
            .optionalStart().appendLiteral('[').parseCaseSensitive()
                            .appendZoneRegionId()
                            .appendLiteral(']').optionalEnd()
            .parseDefaulting(ChronoField.OFFSET_SECONDS,0)
```

```
            .parseStrict()
            .toFormatter();
```

Use ISO_LOCAL_DATE_TIME Formatter by Default

Going through the details of `DateTimeFormatterBuilder` is out of scope for what we are here to cover. Using the `ISO_LOCAL_DATE_TIME` formatter should be good enough in most cases.

[1] *"DateTimeFormatter RFC_1123_DATE_TIME Javadoc"*, DateTimeFormatter Javadoc, Oracle

Chapter 5. Java Marshallers

I will be using four different data marshalling providers during this lecture:

- **Jackson JSON** the default JSON provider included within Spring and Spring Boot. It implements its own proprietary interface for mapping Java POJOs to JSON text.
- **JSON Binding (JSON-B)** a relatively new Jakarta EE standard for JSON marshalling. The reference implementation is Yasson from the open source Glassfish project. It will be used to verify and demonstrate portability between the built-in Jackson JSON and other providers.
- **Jackson XML** a tightly integrated sibling of Jackson JSON. This requires a few extra module dependencies but offers a very similar setup and annotation set as the JSON alternative. I will use Jackson XML as my primary XML provider during examples.
- **Java Architecture for XML Binding (JAXB)** a well-seasoned XML marshalling framework that was the foundational requirement for early JavaEE servlet containers. I will use JAXB to verify and demonstrate portability between Jackson XML and other providers.

Spring Boot comes with a Jackson JSON pre-wired with the web dependencies. It seamlessly gets called from RestTemplate, RestClient, WebClient and the RestController when `application/json` or nothing has been selected. Jackson XML requires additional dependencies — but integrates just as seamlessly with the client and server-side frameworks for `application/xml`. For those reasons — Jackson JSON and Jackson XML will be used as our core marshalling frameworks. JSON-B and JAXB will just be used for portability testing.

Chapter 6. JSON Content

JSON is the content type most preferred by Javascript UI frameworks and NoSQL databases. It has quickly overtaken XML as a preferred data exchange format.

Example JSON Document

```
{
  "id" : 0,
  "author" : "Hotblack Desiato",
  "text" : "Parts of the inside of her head screamed at other parts of the inside of
her head.",
  "date" : "1981-05-15"
}
```

Much of the mapping can be accomplished using Java reflection. Provider-specific annotations can be added to address individual issues. Let's take a look at how both Jackson JSON and JSON-B can be used to map our `QuoteDTO` POJO to the above JSON content. The following is a trimmed down copy of the DTO class I showed you earlier. What kind of things do we need to make that mapping?

Review: Example DTO

```
@Data
public class QuoteDTO {
    private int id;
    private String author;
    private String text;
    private LocalDate date; ①
    private String ignored; ②
}
```

① may need some LocalDate formatting

② may need to mark as excluded

6.1. Jackson JSON

For the simple cases, our DTO classes can be mapped to JSON with minimal effort using Jackson JSON. However, we potentially need to shape our document and can use Jackson annotations to customize. The following example shows using an annotation to eliminate a property from the JSON document.

Example Pre-Tweaked JSON Payload

```
{
  "id" : 0,
  "author" : "Hotblack Desiato",
  "text" : "Parts of the inside of her
head screamed at other parts of the
inside of her head.",
  "date" : [ 1981, 5, 15], ①
  "ignored" : "ignored" ②
}
```

① LocalDate in a non-ISO array format

② unwanted field included

Example QuoteDTO with Jackson Annotation(s)

```
import
com.fasterxml.jackson.annotation.JsonIgn
ore;
...
public class QuoteDTO {
    private int id;
    private String author;
    private String text;
    private LocalDate date;
    @JsonIgnore ①
    private String ignored;
}
```

① Jackson `@JsonIgnore` causes the Java property to be ignored when converting to/from JSON

Date/Time Formatting Handled at ObjectMapper/Marshaller Level

The example annotation above only addressed the `ignore` property. We will address date/time formatting at the ObjectMapper/marshaller level below.

6.1.1. Jackson JSON Initialization

Jackson JSON uses an `ObjectMapper` class to go to/from POJO and JSON. We can configure the mapper with options or configure a reusable builder to create mappers with prototype options. Choosing the latter approach will be useful once we move inside the server.

Jackson JSON Imports

```
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
```

We have the ability to simply create a default ObjectMapper directly.

Simple Jackson JSON Initialization

```
ObjectMapper mapper = new ObjectMapper();
```

However, when using Spring it is useful to use the Spring `Jackson2ObjectMapperBuilder` class to set many of the data marshalling types for us.

Jackson JSON Initialization using Builder

```
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
...
ObjectMapper mapper = new Jackson2ObjectMapperBuilder()
```

```
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT) ①
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS) ②
            //more later
        .createXmlMapper(false) ③
        .build();
```

① optional pretty print indentation

② option to use ISO-based strings versus binary values and arrays

③ same Spring builder creates both XML and JSON ObjectMappers

Use Injection When Inside Container

When inside the container, have the `Jackson2ObjectMapperBuilder` injected (i.e., not locally-instantiated) in order to pick up external and property configurations/customizations.

By default, Jackson will marshal zone-based timestamps as a decimal number (e.g., `-6106031876.123456789`) and generic date/times as an array of values (e.g., `[ 1776, 7, 4, 8, 2, 4, 123456789 ]` and `[ 1966, 1, 9 ]`). By disabling this serialization feature, Jackson produces ISO-based strings for all types of timestamps and generic date/times (e.g., `1776-07-04T08:02:04.123456789Z` and `2002-02-14`)

The following example from the class repository shows a builder customizer being registered as a `@Bean` factory to be able to adjust Jackson defaults used by the server. The returned lambda function is called with a builder each time someone injects a `Jackson2ObjectMapper` — provided the Jackson AutoConfiguration has not been overridden.

Example Jackson2ObjectMapperBuilder Custom Configuration

```
/**
 * Execute these customizations first (Highest Precedence) and then the
 * properties second so that properties can override Java configuration.
 */
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public Jackson2ObjectMapperBuilderCustomizer jacksonMapper() {
    return (builder) -> { builder
        //spring.jackson.serialization.indent-output=true
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
        //spring.jackson.serialization.write-dates-as-timestamps=false
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
        //spring.jackson.date-
format=info.ejava.examples.svc.content.quotes.dto.ISODateFormat
        .dateFormat(new ISODateFormat());
    };
}
```

6.1.2. Jackson JSON Marshalling/Unmarshalling

The mapper created from the builder can then be used to marshal the POJO to JSON.

Marshal DTO to JSON using Jackson

```
private ObjectMapper mapper;

public <T> String marshal(T object) throws IOException {
    StringWriter buffer = new StringWriter();
    mapper.writeValue(buffer, object);
    return buffer.toString();
}
```

The mapper can just as easy — unmarshal the JSON to a POJO instance.

Unmarshal DTO from JSON using Jackson

```
public <T> T unmarshal(Class<T> type, String buffer) throws IOException {
    T result = mapper.readValue(buffer, type);
    return result;
}
```

A packaged set of marshal/unmarshal convenience routines have been packaged inside `ejava-dto-util`.

6.1.3. Jackson JSON Maven Aspects

For modules with only DTOs with Jackson annotations, only a direct dependency on jackson-annotations is necessary.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
</dependency>
```

Modules that will be marshalling/unmarshalling JSON will need the core libraries that can be conveniently brought in through a dependency on one of the following two starters.

- spring-boot-starter-web
- spring-boot-starter-json

```
org.springframework.boot:spring-boot-starter-web:jar
+- org.springframework.boot:spring-boot-starter-json:jar
|  +- com.fasterxml.jackson.core:jackson-databind:jar
|  |  +- com.fasterxml.jackson.core:jackson-annotations:jar
|  |  \- com.fasterxml.jackson.core:jackson-core
```

```
|  +- com.fasterxml.jackson.datatype:jackson-datatype-jdk8:jar ①
|  +- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:jar ①
|  \- com.fasterxml.jackson.module:jackson-module-parameter-names:jar
```

① defines mapping for java.time types

Jackson has built-in ISO mappings for Date and java.time

Jackson has built-in mappings to ISO for `java.util.Date` and `java.time` data types.

6.2. JSON-B

JSON-B (the standard) and Yasson (the reference implementation of JSON-B) can pretty much render a JSON view of our simple DTO class right out of the box. Customizations can be applied using JSON-B annotations. In the following example, the `ignore` Java property is being excluded from the JSON output.

Example Pre-Tweaked JSON-B Payload

```
{
"author": "Reg Nullify",
"date": "1986-05-20", ①
"id": 0,
"ignored": "ignored",
"text": "In the beginning, the Universe
was created. This has made a lot of
people very angry and been widely
regarded as a bad move."
}
```

① LocalDate looks to already be in an ISO-8601 format

Example QuoteDTO with JSON-B Annotation(s)

```
...
import jakarta.json.bind.annotation
.JsonbTransient;
...
public class QuoteDTO {
    private int id;
    private String author;
    private String text;
    private LocalDate date;
    @JsonbTransient ①
    private String ignored;
}
```

① `@JsonbTransient` used to identify unmapped Java properties

6.2.1. JSON-B Initialization

JSON-B provides all mapping through a `Jsonb` builder object that can be configured up-front with various options.

JSON-B Imports

```
import jakarta.json.bind.Jsonb;
import jakarta.json.bind.JsonbBuilder;
import jakarta.json.bind.JsonbConfig;
```

JSON-B Initialization

```
JsonbConfig config=new JsonbConfig()
        .setProperty(JsonbConfig.FORMATTING, true); ①
Jsonb builder = JsonbBuilder.create(config);
```

① adds pretty-printing features to payload

Jsonb is no longer `javax`

The Jsonb package has changed from `javax.json.bind` to `jakarta.json.bind`.

```
import javax.json.bind.Jsonb;   //legacy
import jakarta.json.bind.Jsonb; //modern
```

6.2.2. JSON-B Marshalling/Unmarshalling

The following two examples show how JSON-B marshals and unmarshals the DTO POJO instances to/from JSON.

Marshall DTO using JSON-B

```
private Jsonb builder;

public <T> String marshal(T object) {
    String buffer = builder.toJson(object);
    return buffer;
}
```

Unmarshal DTO using JSON-B

```
public <T> T unmarshal(Class<T> type, String buffer)  {
    T result = (T) builder.fromJson(buffer, type);
    return result;
}
```

6.2.3. JSON-B Maven Aspects

Modules defining only the DTO class require a dependency on the following API definition for the annotations.

```
<dependency>
    <groupId>jakarta.json</groupId>
    <artifactId>jakarta.json-api</artifactId>
</dependency>
```

Modules marshalling/unmarshalling JSON documents using JSON-B/Yasson implementation require

dependencies on `binding-api` and a runtime dependency on `yasson` implementation.

```
org.eclipse:yasson:jar
+- jakarta.json.bind:jakarta.json.bind-api:jar
+- jakarta.json:jakarta.json-api:jar
\- org.glassfish:jakarta.json:jar
```

Chapter 7. XML Content

XML is preferred by many data exchange services that require rigor in their data definitions. That does not mean that rigor is always required. The following two examples are XML renderings of a `QuoteDTO`.

The first example is a straight mapping of Java class/attribute to XML elements. The second example applies an XML namespace and attribute (for the `id` property). Namespaces become important when mixing similar data types from different sources. XML attributes are commonly used to host identity information. XML elements are commonly used for description information. The sometimes arbitrary use of attributes over elements in XML leads to some confusion when trying to perform direct mappings between JSON and XML — since JSON has no concept of an attribute.

Example Vanilla XML Document

```
<QuoteDTO> ①
  <id>0</id> ②
  <author>Zaphod Beeblebrox</author>
  <text>Nothing travels faster than the speed of light with the possible exception of
bad news, which obeys its own special laws.</text>
  <date>1927</date> ③
  <date>6</date>
  <date>11</date>
  <ignored>ignored</ignored> ④
</QuoteDTO>
```

① root element name defaults to variant of class name

② all properties default to `@XmlElement` mapping

③ `java.time` types are going to need some work

④ all properties are assumed to not be ignored

Collections Marshall Unwrapped

The three (3) `date` elements above are elements of an ordered collection marshalled without a wrapping element. If we wanted to keep the collection (versus marshalling in ISO format), it would be common to define a wrapping element to encapsulate the collection — much like parentheses in a sentence.

Example XML Document with Namespaces, Attributes, and Desired Shaping

```
<quote xmlns="urn:ejava.svc-controllers.quotes" id="0"> ① ② ③
  <author>Zaphod Beeblebrox</author>
  <text>Nothing travels faster than the speed of light with the possible exception of
bad news, which obeys its own special laws.</text>
  <date>1927-06-11</date>
</quote> ④
```

① `quote` is our targeted root element name

② `urn:ejava.svc-controllers.quotes` is our targeted namespace

③ we want the `id` mapped as an attribute — not an element

④ we want certain properties from the DTO not to show up in the XML

7.1. Jackson XML

Like Jackson JSON, Jackson XML will attempt to map a Java class solely on Java reflection and default mappings. However, to leverage key XML features like namespaces and attributes, we need to add a few annotations. The partial example below shows our POJO with Lombok and other mappings excluded for simplicity.

Example QuoteDTO with Jackson XML Annotations

```
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;
...
@JacksonXmlRootElement(localName = "quote", ①
   namespace = "urn:ejava.svc-controllers.quotes") ②
public class QuoteDTO {
    @JacksonXmlProperty(isAttribute = true) ③
    private int id;
    private String author;
    private String text;
    private LocalDate date;
    @JsonIgnore ④
    private String ignored;
}
```

① defines the element name when rendered as the root element

② defines namespace for type

③ maps `id` property to an XML attribute — default is XML element

④ reuses Jackson JSON general purpose annotations

7.1.1. Jackson XML Initialization

Jackson XML initialization is nearly identical to its JSON sibling as long as we want them to have the same options. In all of our examples I will be turning off array-based, numeric dates expression in favor of ISO-based strings.

Jackson XML Imports

```
import com.fasterxml.jackson.databind.SerializationFeature;
import com.fasterxml.jackson.dataformat.xml.XmlMapper;
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
```

Jackson XML Initialization

```
XmlMapper mapper = new Jackson2ObjectMapperBuilder()
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT) ①
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS) ②
        //more later
        .createXmlMapper(true) ③
        .build();
```

① pretty print output

② use ISO-based strings for time-based fields versus binary numbers and arrays

③ XmlMapper extends ObjectMapper

7.1.2. Jackson XML Marshalling/Unmarshalling

Marshall DTO using Jackson XML

```
public <T> String marshal(T object) throws IOException {
    StringWriter buffer = new StringWriter();
    mapper.writeValue(buffer, object);
    return buffer.toString();
}
```

Unmarshal DTO using Jackson XML

```
public <T> T unmarshal(Class<T> type, String buffer) throws IOException {
    T result = mapper.readValue(buffer, type);
    return result;
}
```

7.1.3. Jackson XML Maven Aspects

Jackson XML is not broken out into separate libraries as much as its JSON sibling. Jackson XML annotations are housed in the same library as the marshalling/unmarshalling code.

Jackson Dependency for XML-specific Annotation and Marshalling Support

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

7.2. JAXB

JAXB is more particular about the definition of the Java class to be mapped. JAXB requires that the root element of a document be defined with an `@XmlRootElement` annotation with an optional name and namespace defined.

JAXB Requires @XmlRootElement on Root Element of Document

```
com.sun.istack.SAXException2: unable to marshal type
"info.ejava.examples.svc.content.quotes.dto.QuoteDTO"
as an element because it is missing an @XmlRootElement annotation
```

Required @XmlRootElement supplied

```
...
import jakarta.xml.bind.annotation.XmlRootElement;
...
@XmlRootElement(name = "quote", namespace = "urn:ejava.svc-controllers.quotes")
public class QuoteDTO { ① ②
```

① default name is `quoteDTO` if not supplied

② default to no namespace if not supplied

JAXB 4.x is no longer `javax`

JAXB 4.x used in Spring Boot 3/Spring 6 is no longer `javax`. The Java package changed from `javax.xml.bind` to `jakarta.xml.bind`

```
import javax.xml.bind.annotation.XmlRootElement; //Spring Boot 2
import jakarta.xml.bind.annotation.XmlRootElement; //Spring Boot 3
```

7.2.1. Custom Type Adapters

JAXB has no default definitions for `java.time` classes and must be handled with custom adapter code.

JAXB has no default mapping for java.time classes

```
INFO: No default constructor found on class java.time.LocalDate
java.lang.NoSuchMethodException: java.time.LocalDate.<init>()
```

This has always been an issue for Date formatting even before `java.time` and can easily be solved with a custom adapter class that converts between a String and the unsupported type. We can locate packaged solutions on the web, but it is helpful to get comfortable with the process on our own.

We first create an adapter class that extends `XmlAdapter<ValueType, BoundType>` — where ValueType is a type known to JAXB and BoundType is the type we are mapping. We can use DateFormatter.ISO_LOCAL_DATE to marshal and unmarshal the LocalDate to/from text.

Example JAXB LocalDate Adapter

```
import jakarta.xml.bind.annotation.adapters.XmlAdapter;
...
```

```
public static class LocalDateJaxbAdapter extends extends XmlAdapter<String, LocalDate>
{
    @Override
    public LocalDate unmarshal(String text) {
        return text == null ? null : LocalDate.parse(text, DateTimeFormatter
.ISO_LOCAL_DATE);
    }
    @Override
    public String marshal(LocalDate timestamp) {
        return timestamp==null ? null : DateTimeFormatter.ISO_LOCAL_DATE.format
(timestamp);
    }
}
```

We next annotate the Java property with `@XmlJavaTypeAdapter`, naming our adapter class.

Example Mapping Custom Type to Adapter for Class Property

```
import jakarta.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
...
@XmlAccessorType(XmlAccessType.FIELD) ②
public class QuoteDTO {
...
    @XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ①
    private LocalDate date;
```

① custom adapter required for unsupported types

② must manually set access to FIELD when annotating attributes

7.2.2. JAXB Initialization

There is no sharable, up-front initialization for JAXB. All configuration must be done on individual, non-sharable JAXBContext objects. However, JAXB does have a package-wide annotation that the other frameworks do not. The following example shows a `package-info.java` file that contains annotations to be applied to every class in the same Java package.

JAXB Package Annotations

```
//package-info.java
@XmlSchema(namespace = "urn:ejava.svc-controllers.quotes")
package info.ejava.examples.svc.content.quotes.dto;

import jakarta.xml.bind.annotation.XmlSchema;
```

The same feature could be used to globally apply adapters package-wide.

Example Mapping Custom Type to Adapter for Package

```
//package-info.java
```

```
@XmlSchema(namespace = "urn:ejava.svc-controllers.quotes")
@XmlJavaTypeAdapter(type= LocalDate.class, value=JaxbTimeAdapters.
LocalDateJaxbAdapter.class)
package info.ejava.examples.svc.content.quotes.dto;

import jakarta.xml.bind.annotation.XmlSchema;
import jakarta.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import java.time.LocalDate;
```

7.2.3. JAXB Marshalling/Unmarshalling

JAXB Imports

```
import jakarta.xml.bind.JAXBContext;
import jakarta.xml.bind.JAXBException;
import jakarta.xml.bind.Marshaller;
import jakarta.xml.bind.Unmarshaller;
```

Marshalling/Unmarshalling starts out by constructing a `JAXBContext` scoped to handle the classes of interest. This will include the classes explicitly named and the classes they reference. Therefore, one would only need to create a `JAXBContext` by explicitly naming the input and return types of a Web API method.

Marshall DTO using JAXB

```
public <T> String marshal(T object) throws JAXBException {
    JAXBContext jbx = JAXBContext.newInstance(object.getClass()); ①
    Marshaller marshaller = jbx.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true); ②

    StringWriter buffer = new StringWriter();
    marshaller.marshal(object, buffer);
    return buffer.toString();
}
```

① explicitly name primary classes of interest

② adds newline and indentation formatting

Unmarshal DTO using JAXB

```
public <T> T unmarshal(Class<T> type, String buffer) throws JAXBException {
    JAXBContext jbx = JAXBContext.newInstance(type);
    Unmarshaller unmarshaller = jbx.createUnmarshaller();

    ByteArrayInputStream bis = new ByteArrayInputStream(buffer.getBytes());
    T result = (T) unmarshaller.unmarshal(bis);
    return result;
}
```

7.2.4. JAXB Maven Aspects

Modules that define DTO classes only will require a direct dependency on the `jakarta.xml-bind-api` library for annotations and interfaces.

```
<dependency>
    <groupId>jakarta.xml.bind</groupId>
    <artifactId>jakarta.xml.bind-api</artifactId>
</dependency>
```

Modules marshalling/unmarshalling DTO classes using JAXB will require a dependency on the `jaxb-runtime` artifact.

```
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
</dependency>
```

Deprecated javax.xml.bind Dependencies

The `jaxb-api` artifact contains the deprecated `javax.xml.bind` interface types.

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
</dependency>
```

The following two artifacts contain the deprecated `javax.xml.bind` implementation classes. `jaxb-core` contains visible utilities used map between Java and XML Schema. `jaxb-impl` is more geared towards runtime. Since both are needed, I am not sure why there is not a dependency between one another to make that automatic.

```
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
</dependency>
```

Chapter 8. Configure Server-side Jackson

8.1. Dependencies

Jackson JSON will already be on the classpath when using `spring-boot-web-starter`. To also support XML, make sure the server has an additional `jackson-dataformat-xml` dependency.

Server-side Dependency Required for Jackson XML Support

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

8.2. Configure ObjectMapper

Both XML and JSON mappers are instances of ObjectMapper. To configure their use in our application — we can go one step higher and create a builder for jackson to use as its base. That is all we need to know as long as we can configure them identically.

Jackson's AutoConfiguration provides a layered approach to customizing the marshaller. One can configure using:

- spring.jackson properties (e.g., `spring.jackson.serialization.*`)
- Jackson2ObjectMapperBuilderCustomizer — a functional interface that will be passed a builder pre-configured using properties

Assigning a high precedence order to the customizer will allow properties to flexibly override the Java code configuration.

Server-side Jackson Builder @Bean Factory

```
...
import com.fasterxml.jackson.databind.SerializationFeature;
import
org.springframework.boot.autoconfigure.jackson.Jackson2ObjectMapperBuilderCustomizer;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;

@SpringBootApplication
public class QuotesApplication {
    public static void main(String...args) {
        SpringApplication.run(QuotesApplication.class, args);
    }

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE) ②
    public Jackson2ObjectMapperBuilderCustomizer jacksonMapper() {
```

```
        return (builder) -> { builder ①
            //spring.jackson.serialization.indent-output=true
            .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
            //spring.jackson.serialization.write-dates-as-timestamps=false
            .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
            //spring.jackson.date-
format=info.ejava.examples.svc.content.quotes.dto.ISODateFormat
            .dateFormat(new ISODateFormat());
        };
}
```

① returns a lambda function that is called with a `Jackson2ObjectMapperBuilder` to customize. Jackson uses this same definition for both XML and JSON mappers

② highest order precedence applies this configuration first, then properties — allowing for overrides using properties

8.3. Controller Properties

We can register what MediaTypes each method supports by adding a set of consumes and produces properties to the `@RequestMapping` annotation in the controller. This is an array of MediaType values (e.g., `["application/json", "application/xml"]`) that the endpoint should either accept or provide in a response.

Example Consumes and Produces Mapping

```
@RequestMapping(path= QUOTES_PATH,
        method= RequestMethod.POST,
        consumes = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE
},
        produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE
})
public ResponseEntity<QuoteDTO> createQuote(@RequestBody QuoteDTO quote) {
    QuoteDTO result = quotesService.createQuote(quote);

    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()
            .replacePath(QUOTE_PATH)
            .build(result.getId());
    ResponseEntity<QuoteDTO> response = ResponseEntity.created(uri)
            .body(result);
    return response;
}
```

The `Content-Type` request header is matched with one of the types listed in `consumes`. This is a single value and the following example uses an `application/json` Content-Type and the server uses our Jackson JSON configuration and DTO mappings to turn the JSON string into a POJO.

Example POST of JSON Content

```
POST http://localhost:64702/api/quotes
```

```
sent: [Accept:"application/xml", Content-Type:"application/json", Content-Length:"108
"]
{
  "id" : 0,
  "author" : "Tricia McMillan",
  "text" : "Earth: Mostly Harmless",
  "date" : "1991-05-11"
}
```

If there is a match between Content-Type and consumes, the provider will map the body contents to the input type using the mappings we reviewed earlier. If we need more insight into the request headers — we can change the method mapping to accept a RequestEntity and obtain the headers from that object.

Example Alternative Content Mapping

```
    @RequestMapping(path= QUOTES_PATH,
        method= RequestMethod.POST,
        consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
//    public ResponseEntity<QuoteDTO> createQuote(@RequestBody QuoteDTO quote) {
    public ResponseEntity<QuoteDTO> createQuote(RequestEntity<QuoteDTO> request) {①
        QuoteDTO quote = request.getBody();
        log.info("CONTENT_TYPE={}", request.getHeaders().get(HttpHeaders.CONTENT_TYPE
));
        log.info("ACCEPT={}", request.getHeaders().get(HttpHeaders.ACCEPT));
        QuoteDTO result = quotesService.createQuote(quote);
```

① injecting raw input RequestEntity versus input payload to inspect header properties

The log statements at the start of the methods output the following two lines with request header information.

Example Header Output

```
QuotesController#createQuote:38 CONTENT_TYPE=[application/json;charset=UTF-8]
QuotesController#createQuote:39 ACCEPT=[application/xml]
```

Whatever the service returns (success or error), the `Accept` request header is matched with one of the types listed in the `produces`. This is a list of N values listed in priority order. In the following example, the client used an `application/xml` Accept header and the server converted it to XML using our Jackson XML configuration and mappings to turn the POJO into an XML response.

Review: Original Request Headers

```
sent: [Accept:"application/xml", Content-Type:"application/json", Content-
Length:"108"]
```

Response Header and Payload

```
rcvd: [Location:"http://localhost:64702/api/quotes/1", Content-Type:"application/xml",
Transfer-Encoding:"chunked", Date:"Fri, 05 Jun 2020 19:44:25 GMT", Keep-
Alive:"timeout=60", Connection:"keep-alive"]
<quote xmlns="urn:ejava.svc-controllers.quotes" id="1">
  <author xmlns="">Tricia McMillan</author>
  <text xmlns="">Earth: Mostly Harmless</text>
  <date xmlns="">1991-05-11</date>
</quote>
```

If there is no match between Content-Type and consumes, a `415/Unsupported Media Type` error status is returned. If there is no match between Accept and produces, a `406/Not Acceptable` error status is returned. Most of this content negotiation and data marshalling/unmarshalling is hidden from the controller.

Chapter 9. Client Marshall Request Content

If we care about the exact format our POJO is marshalled to or the format the service returns, we can no longer pass a naked POJO to the client library. We must wrap the POJO in a `RequestEntity` and supply a set of headers with format specifications. The following shows an example using RestTemplate.

RestTemplate Content Headers Example

```
RequestEntity<QuoteDTO> request = RequestEntity.post(quotesUrl) ①
        .contentType(contentType) ②
        .accept(acceptType) ③
        .body(validQuote);
ResponseEntity<QuoteDTO> response = restTemplate.exchange(request, QuoteDTO.class);
```

① create a POST request with client headers

② express desired Content-Type for the request

③ express Accept types for the response

The following example shows the request and reply information exchange for an `application/json` Content-Type and Accept header.

Example JSON POST Request and Reply

```
POST http://localhost:49252/api/quotes, returned CREATED/201
sent: [Accept:"application/json", Content-Type:"application/json", Content-Length:"
146"]
{
  "id" : 0,
  "author" : "Zarquon",
  "text" : "Whatever your tastes, Magrathea can cater for you. We are not proud.",
  "date" : "1920-08-17"
}
rcvd: [Location:"http://localhost:49252/api/quotes/1", Content-Type:"application/json
", Transfer-Encoding:"chunked", Date:"Fri, 05 Jun 2020 20:17:35 GMT", Keep-Alive:
"timeout=60", Connection:"keep-alive"]
{
  "id" : 1,
  "author" : "Zarquon",
  "text" : "Whatever your tastes, Magrathea can cater for you. We are not proud.",
  "date" : "1920-08-17"
}
```

The following example shows the request and reply information exchange for an `application/xml` Content-Type and Accept header.

Example XML POST Request and Reply

```
POST http://localhost:49252/api/quotes, returned CREATED/201
```

```
sent: [Accept:"application/xml", Content-Type:"application/xml", Content-Length:"290"]
<quote xmlns="urn:ejava.svc-controllers.quotes" id="0">
  <author xmlns="">Humma Kavula</author>
  <text xmlns="">In the beginning, the Universe was created. This has made a lot of
people very angry and been widely regarded as a bad move.</text>
  <date xmlns="">1942-03-03</date>
</quote>

rcvd: [Location:"http://localhost:49252/api/quotes/4", Content-Type:"application/xml",
Transfer-Encoding:"chunked", Date:"Fri, 05 Jun 2020 20:17:35 GMT", Keep-
Alive:"timeout=60", Connection:"keep-alive"]
<quote xmlns="urn:ejava.svc-controllers.quotes" id="4">
  <author xmlns="">Humma Kavula</author>
  <text xmlns="">In the beginning, the Universe was created. This has made a lot of
people very angry and been widely regarded as a bad move.</text>
  <date xmlns="">1942-03-03</date>
</quote>
```

Chapter 10. Client Filters

The runtime examples above showed HTTP traffic and marshalled payloads. That can be very convenient for debugging purposes. There are two primary ways of examining marshalled payloads.

Switch accepted Java type to String

Both our client and controller declare they expect a `QuoteDTO.class` to be the response. That causes the provider to map the String into the desired type. If the client or controller declared they expected a `String.class`, they would receive the raw payload to debug or later manually parse using direct access to the unmarshalling code.

Add a filter

Both RestTemplate and WebClient accept filters in the request and response flow. RestTemplate is easier and more capable to use because of its synchronous behavior. We can register a filter to get called with the full request and response in plain view — with access to the body — using RestTemplate. WebClient, with its asynchronous design has a separate request and response flow with no easy access to the payload.

10.1. RestTemplate and RestClient

The following code provides an example of a filter that will work for the synchronous RestTemplate and RestClient. It shows the steps taken to access the request and response payload. Note that reading the body of a request or response is commonly a read-once restriction. The ability to read the body multiple times will be taken care of within the `@Bean` factory method registering this filter.

Example RestTemplate/RestClient Logging Filter

```
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.http.client.ClientHttpResponse;
...
public class RestTemplateLoggingFilter implements ClientHttpRequestInterceptor {
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,①
                        ClientHttpRequestExecution execution) throws IOException {
        ClientHttpResponse response = execution.execute(request, body); ①
        HttpMethod method = request.getMethod();
        URI uri = request.getURI();
        HttpStatusCode status = response.getStatusCode();
        String requestBody = new String(body);
        String responseBody = this.readString(response.getBody());
        //... log debug
        return response;
    }
    private String readString(InputStream inputStream) { ... }
    ...
}
```

① interceptor has access to the client request and response

RestTemplateLoggingFilter is for all Synchronous Requests

The example class is called `RestTemplateLoggingFilter` because `RestTemplate` was here first, the filter is used many times in many examples, and I did not want to make the generalized name change at this time. It is specific to synchronous requests, which includes `RestClient`.

The following code shows an example of a `@Bean` factory that creates `RestTemplate` instances configured with the debug logging filter shown above.

Example @Bean Factory Registering RestTemplate Filter

```
@Bean
ClientHttpRequestFactory requestFactory() {
    return new SimpleClientHttpRequestFactory(); ③
}

@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder,
    ClientHttpRequestFactory requestFactory) { ③
    return builder.requestFactory(
            //used to read the streams twice -- so we can use the logging filter
            ()->new BufferingClientHttpRequestFactory(requestFactory)) ②
        .interceptors(new RestTemplateLoggingFilter()) ①
        .build();
}

@Bean
public RestClient restClient(RestClient.Builder builder,
    ClientHttpRequestFactory requestFactory) { ③
    return builder  //requestFactory used to read stream twice
            .requestFactory(new BufferingClientHttpRequestFactory(requestFactory)) ②
            .requestInterceptor(new RestTemplateLoggingFilter()) ①
            .build();
}
```

① the overall intent of this `@Bean` factory is to register the logging filter

② must configure client with a buffer (`BufferingClientHttpRequestFactory`) for body to enable multiple reads

③ providing a `ClientRequestFactory` to be forward-ready for SSL communications

10.2. WebClient

The following code shows an example request and response filter. They are independent and are implemented using a Java 8 lambda function. You will notice that we have no easy access to the request or response body.

Example WebClient Logging Filter

```
package info.ejava.examples.common.webflux;

import org.springframework.web.reactive.function.client.ExchangeFilterFunction;
...
public class WebClientLoggingFilter {
    public static ExchangeFilterFunction requestFilter() {
        return ExchangeFilterFunction.ofRequestProcessor((request) -> {
                //access to
                //request.method(),
                //request.url(),
                //request.headers()
            return Mono.just(request);
        });
    }
    public static ExchangeFilterFunction responseFilter() {
        return ExchangeFilterFunction.ofResponseProcessor((response) -> {
                //access to
                //response.statusCode()
                //response.headers().asHttpHeaders())
            return Mono.just(response);
        });
    }
}
```

The code below demonstrates how to register custom filters for injected WebClient instances.

Example @Bean Factory Registering WebClient Filters

```
@Bean
public WebClient webClient(WebClient.Builder builder) {
    return builder
            .filter(WebClientLoggingFilter.requestFilter())
            .filter(WebClientLoggingFilter.responseFilter())
            .build();
}
```

Chapter 11. Date/Time Lenient Parsing and Formatting

In our quote example, we had an easy LocalDate to format and parse, but that even required a custom adapter for JAXB. Integration of other time-based properties can get more involved as we get into complete timestamps with timezone offsets. So lets try to address the issues here before we complete the topic on content exchange.

The primary time-related issues we can encounter include:

Table 1. Potential Time-related Format Issues

Potential Issue	Description
type not supported	We have already encountered that with JAXB and solved using a custom adapter. Each of the providers offer their own form of adapter (or serializer/deserializer), so we have a good headstart on how to solve the hard problems.
non-UTC ISO offset style supported	There are at least four or more expressions of a timezone offset (Z, +00, +0000, or +00:00) that could be used. Not all of them can be parsed by each provider out-of-the-box.
offset versus extended offset zone formatting	There are more verbose styles (Z[UTC]) of expressing timezone offsets that include the ZoneId
fixed width or truncated	Are all fields supplied at all times even when they are 0 (e.g., `1776-07-04T00:00:00.100+00:00`) or are values truncated to only include significant values (e.g., '1776-07-04T00:00:00.1Z'). This mostly applies to fractions of seconds.

We should always strive for:

- consistent (ISO) standard format to marshal time-related fields
- leniently parsing as many formats as possible

Let's take a look at establishing an internal standard, determining which providers violate that standard, how to adjust them to comply with our standard, and how to leniently parse many formats with the Jackson parser since that will be our standard provider for the course.

11.1. Out of the Box Time-related Formatting

Out of the box, I found the providers marshalled `OffsetDateTime` and `Date` with the following format. I provided an `OffsetDateTime` and `Date` timestamp with varying number of nanoseconds (123456789, 1, and 0 ns) and timezone UTC and -05:00) and the following table shows what was marshalled for the DTO.

Table 2. Default Provider OffsetDateTime and Date Formats

Provider	OffsetDateTime	Trunc	Date	Trunc
Jackson	1776-07-04T00:00:00.123456789Z 1776-07-04T00:00:00.1Z 1776-07-04T00:00:00Z 1776-07-03T19:00:00.123456789-05:00 1776-07-03T19:00:00.1-05:00 1776-07-03T19:00:00-05:00	Yes	1776-07-04T00:00:00.123+00:00 1776-07-04T00:00:00.100+00:00 1776-07-04T00:00:00.000+00:00	No
JSON-B	1776-07-04T00:00:00.123456789Z 1776-07-04T00:00:00.1Z 1776-07-04T00:00:00Z 1776-07-03T19:00:00.123456789-05:00 1776-07-03T19:00:00.1-05:00 1776-07-03T19:00:00-05:00	Yes	1776-07-04T00:00:00.123Z[UTC] 1776-07-04T00:00:00.1Z[UTC] 1776-07-04T00:00:00Z[UTC]	Yes
JAXB	(not supported/ custom adapter required)	n/a	1776-07-03T19:00:00.123-05:00 1776-07-03T19:00:00.100-05:00 1776-07-03T19:00:00-05:00	Yes/ No

Jackson and JSON-B — out of the box — use an ISO format that truncates nanoseconds and uses "Z" and "+00:00" offset styles for `java.time` types. JAXB does not support `java.time` types. When a non-UTC time is supplied, the time is expressed using the targeted offset. You will notice that Date is always modified to be UTC.

Jackson Date format is a fixed length, no truncation, always expressed at UTC with an `+HH:MM` expressed offset. JSON-B and JAXB Date formats truncate milliseconds/nanoseconds. JSON-B uses extended timezone offset (`Z[UTC]`) and JAXB uses "+00:00" format. JAXB also always expresses the Date in `EST` in my case.

11.2. Out of the Box Time-related Parsing

To cut down on our choices, I took a look at which providers out-of-the-box could parse the different timezone offsets. To keep things sane, my detailed focus was limited to the Date field. The table shows that each of the providers can parse the "Z" and "+00:00" offset format. They were also able to process variable length formats when faced with less significant nanosecond cases.

Table 3. Default Can Parse Formats

Provider	ISO Z	ISO +00	ISO +0000	ISO +00:00	ISO Z[UTC]
Jackson	**Yes**	Yes	Yes	**Yes**	No
JSON-B	**Yes**	No	No	**Yes**	Yes
JAXB	**Yes**	No	No	**Yes**	No

The testing results show that timezone expressions "Z" or "+00:00" format should be portable and something to target as our marshalling format.

- Jackson - no output change
- JSON-B - requires modification

- JAXB - requires no change

11.3. JSON-B DATE_FORMAT Option

We can configure JSON-B time-related field output using a `java.time` format string. `java.time` permits optional characters. `java.text` does not. The following expression is good enough for Date output but will create a parser that is intolerant of varying length timestamps. For that reason, I will not choose the type of option that locks formatting with parsing.

JSON-B global DATE_FORMAT Option

```
JsonbConfig config=new JsonbConfig()
    .setProperty(JsonbConfig.DATE_FORMAT, "yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]") ①
    .setProperty(JsonbConfig.FORMATTING, true);
builder = JsonbBuilder.create(config);
```

① a fixed formatting and parsing candidate option rejected because of parsing intolerance

11.4. JSON-B Custom Serializer Option

A better JSON-B solution would be to create a serializer — independent of deserializer — that takes care of the formatting.

Example JSON-B Default Serializer

```
public class DateJsonbSerializer implements JsonbSerializer<Date> {
    @Override
    public void serialize(Date date, JsonGenerator generator, SerializationContext
serializationContext) {
        generator.write(DateTimeFormatter.ISO_INSTANT.format(date.toInstant()));
    }
}
```

We add `@JsonbTypeSerializer` annotation to the field we need to customize and supply the class for our custom serializer.

Example JSON-B Annotation Applied

```
@JsonbTypeSerializer(JsonbTimeSerializers.DateJsonbSerializer.class)
private Date date;
```

With the above annotation in place and the JsonConfig unmodified, we get output format we want from JSON-B without impacting its built-in ability to parse various time formats.

- 1776-07-04T00:00:00.123Z
- 1776-07-04T00:00:00.100Z
- 1776-07-04T00:00:00Z

11.5. Jackson Lenient Parser

All those modifications shown so far are good, but we would also like to have lenient input parsing — possibly more lenient than built into the providers. Jackson provides the ability to pass in a `SimpleDateFormat` format string or an instance of class that extends `DateFormat`. `SimpleDateFormat` does not make a good lenient parser, therefore I created a lenient parser that uses DateTimeFormatter framework and plugged that into the `DateFormat` framework.

Example Custom DateFormat Class Implementing Lenient Parser

```
public class ISODateFormat extends DateFormat implements Cloneable {
    public static final DateTimeFormatter UNMARSHALLER = new DateTimeFormatterBuilder
()
                //...
                    .toFormatter();
    public static final DateTimeFormatter MARSHALLER = DateTimeFormatter
.ISO_OFFSET_DATE_TIME;
    public static final String MARSHAL_ISO_DATE_FORMAT = "yyyy-MM-
dd'T'HH:mm:ss[.SSS]XXX";

    @Override
    public Date parse(String source, ParsePosition pos) {
        OffsetDateTime odt = OffsetDateTime.parse(source, UNMARSHALLER);
        pos.setIndex(source.length()-1);
        return Date.from(odt.toInstant());
    }
    @Override
    public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition pos)
{
        ZonedDateTime zdt = ZonedDateTime.ofInstant(date.toInstant(), ZoneOffset.UTC);
        MARSHALLER.formatTo(zdt, toAppendTo);
        return toAppendTo;
    }
    @Override
    public Object clone() {
        return new ISODateFormat(); //we have no state to clone
    }
}
```

I have built the lenient parser using the Java interface to DateTimeFormatter. It is designed to

- handle variable length time values
- different timezone offsets
- a few different timezone offset expressions

DateTimeFormatter Lenient Parser Definition

```
public static final DateTimeFormatter UNMARSHALLER = new DateTimeFormatterBuilder()
    .parseCaseInsensitive()
```

```
    .append(DateTimeFormatter.ISO_LOCAL_DATE)
    .appendLiteral('T')
    .append(DateTimeFormatter.ISO_LOCAL_TIME)
    .parseLenient()
    .optionalStart().appendOffset("+HH", "Z").optionalEnd()
    .optionalStart().appendOffset("+HH:mm", "Z").optionalEnd()
    .optionalStart().appendOffset("+HHmm", "Z").optionalEnd()
    .optionalStart().appendLiteral('[').parseCaseSensitive()
                    .appendZoneRegionId()
                    .appendLiteral(']').optionalEnd()
    .parseDefaulting(ChronoField.OFFSET_SECONDS,0)
    .parseStrict()
    .toFormatter();
```

An instance of my `ISODateFormat` class is then registered with the provider to use on all interfaces.

```
mapper = new Jackson2ObjectMapperBuilder()
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
        .dateFormat(new ISODateFormat()) ①
        .createXmlMapper(false)
        .build();
```

① registering a global time formatter for Dates

In the server, we can add that same configuration option to our builder `@Bean` factory.

```
@Bean
public Jackson2ObjectMapperBuilderCustomizer jacksonMapper() {
    return (builder) -> { builder
        .featuresToEnable(SerializationFeature.INDENT_OUTPUT)
        .featuresToDisable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
        .dateFormat(new ISODateFormat()); ①
    };
}
```

① registering a global time formatter for Dates for JSON and XML

At this point we have the insights into time-related issues and knowledge of how we can correct.

Chapter 12. Summary

In this module we:

- introduces the DTO pattern and contrasted it with the role of the Business Object
- implemented a DTO class with several different types of fields
- mapped our DTOs to/from a JSON and XML document using multiple providers
- configured data mapping providers within our server
- identified integration issues with time-related fields and learned how to create custom adapters to help resolve issues
- learned how to implement client filters
- took a deeper dive into time-related formatting issues in content and ways to address