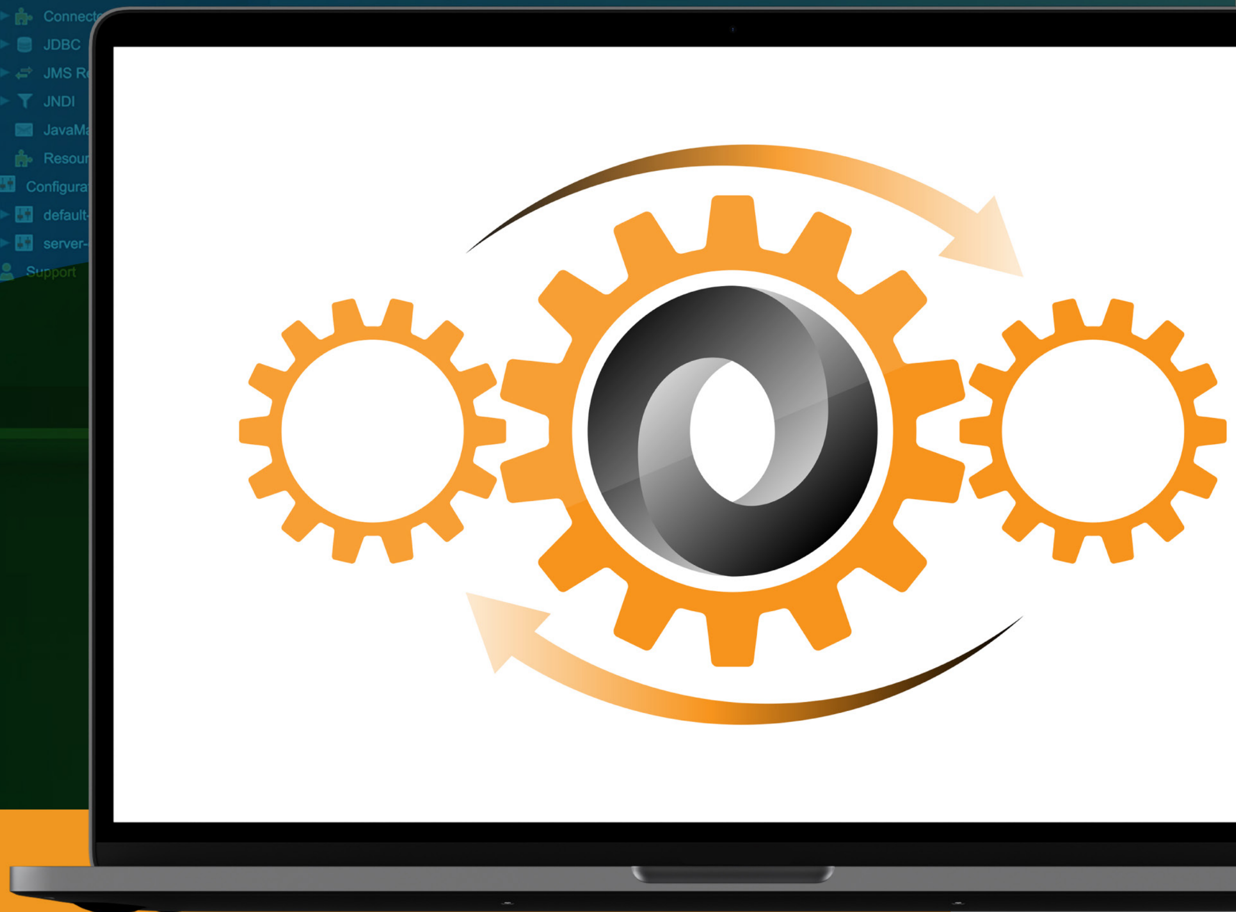
The Complete Guide To JSON Processing On The Jakarta EE Platform



The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

User Guide

Contents

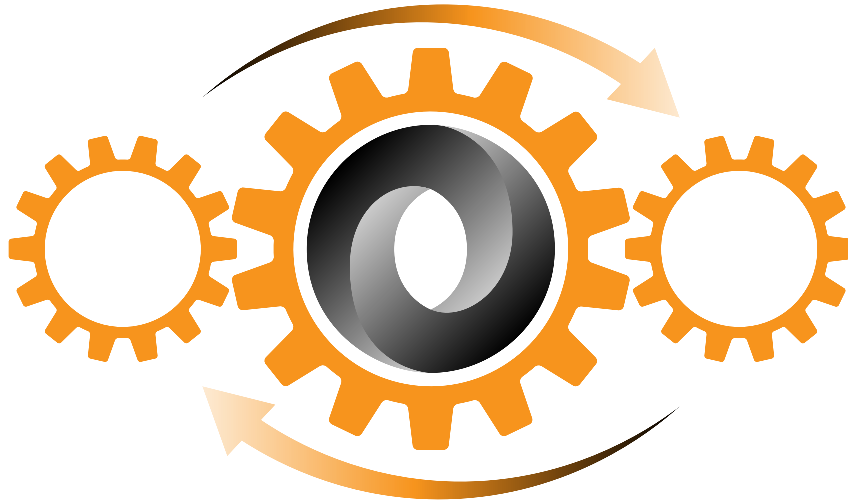
What is Jakarta EE? **1**
- What is a Specification? 2
- What is a Compatible Implementation? 2

What is Eclipse MicroProfile? **3**

Jakarta EE Application Development Process **3**
- Development 4

Jakarta JSON Binding **5**
- Configuration 7
  - Date and Number Formatting 8
  - Output Formatting 8
  - Null Values 8
  - Property Order 8
  - Strict Internet JSON (I-JSON) 8
  - Property Naming Strategy 9
  - Ignoring Fields 10
- Mapping 10
  - Mapping Individual Objects 11
  - Mapping Collections 11
  - Mapping Java Records 13
- Custom Adapters 15
- Use Cases 17
  - Test Resources 17
  - Simpler Data Transfer 17
  - Better REST Customization 17

Summary **18**



The Java Platform has been the platform of choice for enterprise application development for a lot of developers over the last two and half decades. There is no shortage of frameworks and platforms for developing all kinds of software applications using the Java Programming Language. One such platform that has stood the test of time is Jakarta EE.

This guide shows you how to handle the JavaScript Object Notation (JSON) on the Jakarta EE Platform. You will learn about Jakarta JSON-B - how to customise it, how to map objects and collections, how to map Java records and when to create custom adapters and serializers. By the end of this guide, you will have a good grasp of how to handle JSON in your Jakarta EE applications with Jakarta JSON-B.

What is Jakarta EE?

Jakarta EE is a set of community developed, abstract specifications that together form a platform for developing end-to-end, multi-tier enterprise applications. Jakarta EE is built on the Java Standard Edition, and aims to provide a stable, reliable and vendor neutral platform on which to develop cloud native applications.

Hitherto, Jakarta EE was called Java EE and was a property of Oracle Inc., evolved through the Java Community Process (JCP). However, in late 2017, Oracle decided to move the platform to an open foundation for a much broader community-led evolution. The Eclipse Foundation got chosen and Java EE, after the transfer, got rebranded to Jakarta EE.

What is a Specification?

As stated in the above definition, Jakarta EE is made up of a set of specifications that each cover a specific API for solving a specific software development need. For example, the Jakarta Contexts and Dependency Injection specification provides constructs for creating loosely coupled applications through dependency injection. These different specifications are combined into a single "umbrella" specification for each Jakarta EE release. As such, Jakarta EE 10 for instance, is released under the Jakarta EE 10 specification.

More technically, a specification is a formal proposal document made to the Jakarta EE Specification Committee through the Jakarta EE Specification Process (JESP) that outlines the functions of a given set of APIs. This document outlines what the expected behaviour should be for various invocations of the API. The specification then acts as the blueprint for the API.

What is a Compatible Implementation?

As a specification is merely a document that outlines the behaviour of a given API, it needs an implementation that realises the actual outcomes for each invocation of the API. For instance, the Jakarta Persistence specification provides the `EntityManager` interface that has the `persist() method`. This method, when called and passed an instance of a Jakarta Persistence entity, persists that entity instance as a database row to the underlying database. The "library" that does the actual work of taking that instance and making sure it gets stored to the durable storage when the `EntityManager#persist()` method is invoked, is called a compatible implementation of the Jakarta Persistence specification.

Each specification that makes up the full Jakarta EE platform has an implementation. As a specification itself, the Jakarta EE platform also has an implementation in the form of compatible products. As the specifications are separated from their implementations, you as a developer will generally code against the API constructs of the specification, and are free to pick any compatible implementation of the platform. With this abstraction, Jakarta EE implementation vendors can collaborate on the base, standard specifications and compete through innovations on top of the base platform.

An example of such invocation is the Payara Cloud offering from Payara. This innovation helps you realise the dream of true separation of your business domain application and the runtime that powers it. With Payara Cloud, you simply upload your Jakarta EE application web archive (.war file) and have it automatically deployed to the cloud, just as Jakarta EE was envisaged to have separation of business domain from the runtime. Another example of custom features available on the Payara Platform is remote CDI events. This feature, built on the Jakarta CDI specification, allows the firing of CDI events that can be observed by any listener in a given Hazelcast cluster.

What is Eclipse MicroProfile?

The Jakarta EE Platform is a general purpose platform for developing all kinds of applications. As modern application development paradigms have changed a lot over the past years, there is a need to evolve the platform to meet such changes. One such paradigm is cloud-native software application development.

As the base Jakarta EE Platform has always been geared towards enterprises, it has historically evolved at a much slower pace than changes in the software development space. It is for this reason that the Eclipse MicroProfile project was created as an extension to the base platform to provide cloud native APIs for developing modern cloud-based applications.

Eclipse MicroProfile, built upon Jakarta CDI, Jakarta REST and Jakarta JSON Processing, comes with the following APIs

- OpenTracing
- OpenAPI
- REST Client
- Config
- Fault Tolerance
- Metrics
- JWT Propagation
- Health

These APIs augment the much larger Jakarta EE Platform APIs to provide the developer with a cohesive set of APIs for developing, testing and deploying cloud native modern enterprise applications.

Jakarta EE Application Development Process

Jakarta EE application development follows the general software development lifecycle process of requirements gathering and planning, design and/or prototyping, actual development, testing, deployment and maintenance. Different companies use different combinations of these steps. However, development, testing and deployment are steps that almost every company's application development process will entail.

Development

Developing applications on the Jakarta EE Platform with Eclipse MicroProfile requires that the application declares a dependency on those two APIs. As Maven is the de facto build tool for most Jakarta EE applications, a typical dependency declaration would look like that shown in Listing 1-1.

Listing 1-1

```
<dependency>
                 <groupId>jakarta.platform</groupId>
                 <artifactId>jakarta.jakartaee-api</artifactId>
                 <version>10.0.0</version>
                 <scope>provided</scope>
           </dependency>
           <dependency>
                 <groupId>org.eclipse.microprofile</groupId>
                 <artifactId>microprofile</artifactId>
                 <version>5.0</version>
                 <type>pom</type>
                 <scope>provided</scope>
           </dependency>
```

Listing 1-1 showing Jakarta EE and MicroProfile dependency declaration.

With the above declaration in place, a compatible implementation like the Payara Platform can be used for development. Scoping the dependencies to ‘provided’ means the runtime, or compatible implementation will provide the implementation of the various ‘abstract’ APIs used. There is nothing special about using Jakarta EE for application development. Everything will be dependent upon application domain requirements. The Jakarta EE dependency makes all the specifications that make up the Jakarta EE 10 release available to your application, including Jakarta JSON-B, the subject of this guide.

To use JSON-B outside of a Jakarta runtime like Payara Server, you can add the following dependencies to your `pom.xml` file to have a JSON-B implementation available for instance in your tests.

```
<dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-binding</artifactId>
      <version>3.1.0</version>
      <scope>test</scope>
</dependency>
<dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-processing</artifactId>
      <version>3.1.0</version>
      <scope>test</scope>
</dependency>
```

With these two dependencies in place, you can use the JSON-B API outside of a Jakarta EE runtime and have an implementation available. The examples in this gude run the code as JUnnit test cases. You can find the full sample code here.

Jakarta JSON Binding

Jakarta JSON Binding is a high level API for serialising and deserializing Java objects to and from JSON. It is very much POJO based and easy to use. Listing 1-2 shows the reference Java class that will be used for this discussion. `HelloEntity` is a simple Plain Old Java Object (POJO).

Listing 1-2

```
@Getter
@Setter
public class HelloEntity implements Serializable {


      private Long id;
      private Long version;
      private String name;
      private String greeting;
      private LocalDateTime greetingDate;
      private LocalDateTime dateCreated;
}
```

Listing 1.3 shows how to use the Jakarta JSON Bind API to transform an instance of HelloEntity into a valid, RFC 7159 JSON format.

Listing 1-3

```
Jsonb jsonb = JsonbBuilder.create();
      var jsonString = jsonb.toJson(helloEntity);
      log.log(Level.INFO, () -> jsonString);
```

The above code snippet shows the use of the Jakarta JSON API to convert an instance of the `HelloEntity` to a JSON string. The `jakarta.json.bind.JsonbBuilder` interface has a number of methods for creating instances of jakarta.json.bind.Jsonb. The one used in the above snippet is the default that returns a Jsonb with default configurations. The above invocation generates the JSON string shown in Listing 1-4, representing the HelloEntity instance passed to the Jsonb#toJson method.

Listing 1-4

```
{
  "dateCreated": "2022-12-08T15:38:48.086089036",
  "greeting": "Hello, world!",
  "greetingDate": "2022-12-08T15:38:48.086077212",
  "id": 1,
  "name": "Arquillian",
  "version": 1
}
```

The HelloEntity instance that resulted in the above JSON string is shown in Listing 1-5

Listing 1-5

```
        var helloEntity = new HelloEntity();
        helloEntity.setGreeting("Hello, world!");
        helloEntity.setName("Arquillian");
        helloEntity.setGreetingDate(LocalDateTime.now(ZoneOffset.UTC));
        helloEntity.setDateCreated(LocalDateTime.now(ZoneOffset.UTC));
        helloEntity.setId(1L);
        helloEntity.setVersion(1L);
```

Configuration

The snippet shown in Listing 1.3 uses a default configuration to configure the behaviour of the Jsonb object. The `JsonbBuilder#create` method has an overloaded method version that allows for passing a `jakarta.json.bind.JsonbConfig` configuration object that customises the behaviour of the Jsonb object. Listing 1-6 shows the snippet shown in Listing 1-3 rewritten to pass configuration to the Jsonb.

Listing 1-6

```
JsonbConfig config = new JsonbConfig()
                .withDateFormat("yyyy-MM-dd HH:mm:ss", Locale.UK)
                .withFormatting(true)
                .withNullValues(true)
                .withPropertyOrderStrategy(PropertyOrderStrategy.
LEXICOGRAPHICAL)
                .withStrictIJSON(true)
.withPropertyNamingStrategy(PropertyNamingStrategy.LOWER_CASE_WITH_
UNDERSCORES);

        Jsonb jsonb = JsonbBuilder.create(config);

        var jsonString = jsonb.toJson(helloEntity);
        log.log(Level.INFO, () -> jsonString);
```

Listing 1-6 shows passing custom configurations to the Jsonb object to customise the generated JSON. Using the same HelloEntity instance as shown in Listing 1-5, the customised Jsonb object created in Listing 1-6 generates the JSON shown in Listing 1-7.

Listing 1-7

```
{
    "date_created": "2022-12-10 10:35:48",
    "entity_date": null,
    "greeting": "Hello, world!",
    "greeting_date": "2022-12-10 10:35:48",
    "hello_prize": null,
    "id": 1,
    "name": "Arquillian",
    "version": 1
}
```

Date and Number Formatting

Jsonb by default uses ISO formats to serialise date and numbers as shown by the dateCreated and greetingDate fields in the JSON shown in Listing 1-4. Date fields can be formatted with custom formats that can be parsed by `java.time.format.DateTimeFormatter`. These formats can be passed in two ways, one being globally as part of the configurations passed to Jsonb as done in Listing 1-6. This results in the formatted date-created and greeting-date fields of the generated JSON shown in Listing 1-7. Date formats can also be set per field in the Java class using the `@JsonbDateFormat` annotation, passing in a valid date format such as @JsonbDateFormat("dd.MM.yyyy").

Numbers can be formatted using the jakarta.json.bind.annotation.JsonbNumberFormat annotation such as `@JsonbNumberFormat("#0,000.00")`. The passed number format should be one that can be parsed by the java.text.DecimalFormat. Both the date and number format annotations can be used on field, getter/setter, type, parameter and package. When used on a class, all date or number fields will use the given format. When used only on a getter, the formatted field will be deserialized but not serialised. When used on only a setter, the formatted field will only be serialised and not deserialized.

Output Formatting

The JsonbConfig passed to the Jsonb in Listing 1-6 sets the formatting option to true with the `withFormatting(true)` option. This causes Jsonb to format the outputted JSON string.

Null Values

Null value options can be set in two ways. Using the `jakarta.json.bind.annotation.JsonbNillable` annotation on a field, setter/getter methods, class or package. Annotating any of these with `@JsonNillable` will have that field serialized to JSON even if it has null values. The second way to configure null values globally is by passing true to the withNullValues method when creating the `JsonbConfig` object as done in Listing 1-6.

Property Order

The order of serialised properties can be set either using the `jakarta.json.bind.annotation.JsonbPropertyOrder` or by passing in any of the constants in `jakarta.json.bind.config.PropertyOrderStrategy` to the `withPropertyOrderStrategy` method as done in Listing 1-6. There are three possible sort orders namely LEXICOGRAPHICAL, REVERSE and ANY. Lexicographical will sort the fields of the serialised class to its lexicographic order, reverse will reverse the fields lexicographically and any will sort the fields in an undefined order.

Strict Internet JSON (I-JSON)

You can enable full support for "Internet JSON" or I-JSON by passing true to the `withStrictIJSON` method. This causes the generated JSON to be much stricter and more interoperable.

Property Naming Strategy

We can set the property naming strategy by passing one of the constants from `PropertyNamingStrategy` class to the `withPropertyNamingStrategy` method, which will be applied globally. The possible naming strategies available are:

```
IDENTITY
LOWER_CASE_WITH_DASHES (my-mixed-case-property)
LOWER_CASE_WITH_UNDERSCORES (my_mixed_case_property)
UPPER_CAMEL_CASE (MyMixedCaseProperty)
UPPER_CAMEL_CASE_WITH_SPACES (My Mixed Case Property)
CASE_INSENSITIVE (mYmIxEdCaSePrOpErTy)
```

The default is IDENTITY, which generates and reads field names in the `lowerCamelCase` format.

Putting all the options discussed so far together, for the given `HelloEntity` instance shown in Listing 1-8 below, we can generate the JSON shown in Listing 1-9.

Listing 1-8

```
  var helloEntity = new HelloEntity();
        helloEntity.setGreeting("Hello, world!");
        helloEntity.setName("Arquillian");
        helloEntity.setGreetingDate(LocalDateTime.now(ZoneOffset.UTC));
        helloEntity.setDateCreated(LocalDateTime.now(ZoneOffset.UTC));
        helloEntity.setEntityDate(LocalDate.now(ZoneOffset.UTC));
        helloEntity.setHelloPrize(new BigDecimal("23500"));
        helloEntity.setVersion(1L);
```

Listing 1-9 shows the generated JSON from the HelloEntity instance above.

Listing 1-9

```
{
    "date_created": "2022-12-10 10:56:53",
    "entity_date": "10.12.2022",
    "greeting": "Hello, world!",
    "greeting_date": "2022-12-10 10:56:53",
    "hello_prize": "23,500.00",
    "id": null,
    "name": "Arquillian",
    "version": 1
}
```

Ignoring Fields

Fields can be excluded from being serialised or deserialised through the `@JsonbTransient` annotation. When used on a field, that field will not be serialised or deserialised to and from JSON. When used only on the getter method of a field, that field will be serialised to JSON but not deserialised, and when used only on the setter method of a field, that field will only be serialised.

Mapping

JSON-B, as you have seen so far, can generate JSON from POJO instances with the Jsonb#toJson method. It can also generate POJO instances from JSON through any of the Jsonb#fromJson methods. For example, Listing 1-10 reads back the JSON generated in Listing 1-9, saved in a file called helloEntity on the classpath.

Listing 1-10 reading back the generated JSON.

Listing 1-10

```
   try (var inputStream = HelloResourceJaxRsSeTest.class.getResourceAsStream("/
data/helloEntity.json")) {

                var helloEntityFromJson = jsonb.fromJson(inputStream,
                 HelloEntity.class);
                assertEquals(helloEntityFromJson.getId(), helloEntity.getId());
                assertEquals(helloEntityFromJson.getGreeting(), helloEntity.
                 getGreeting());
                assertEquals(helloEntityFromJson.getName(), helloEntity.
                 getName());
                assertEquals(helloEntityFromJson.getEntityDate(), helloEntity.
                 getEntityDate());
                assertEquals(helloEntityFromJson.getHelloPrize(), helloEntity.
                 getHelloPrize());

            }
```

The JSON file is read into an input stream that is passed as the source of the JSON to Jsonb#fromJson method, with the class `HelloEntity` as the second argument. This generates a new instance of `HelloEntity`, on which we perform some simple assertions to see if it matches the original instance from which the JSON was created.

Mapping Individual Objects

Our use of Jsonb so far has been to map to and from individual Java objects. The Jsonb#toJson method can take any valid java.lang.Object that can be converted to JSON. In our examples so far, we only converted single instances of HelloEntity with it. But Jsonb can also convert to and from collections.

Mapping Collections

Jsonb can also map to and from generic collections. To convert a list of HelloEntity instances to JSON, Listing 1-11 creates them and simply passes the collection to the toJson method on the Jsonb instance.

Listing 1-11 converting POJO collection to JSON

Listing 1-11

```
      var helloEntity = createEntity("Arquillian", "Hello, World!", new
BigDecimal("23500"));
        var anotherEntity = createEntity("Payara", "Hello Java!", new
BigDecimal("14500"));

        var helloEntityList = new ArrayList<HelloEntity>();
        helloEntityList.add(helloEntity);
        helloEntityList.add(anotherEntity);
        var helloEntityListJson = jsonb.toJson(helloEntityList);
```

The generated JSON list is shown inListing 1-12

Listing 1-12

```
[
  {
    "date_created": "2022-12-10 11:54:18",
    "entity_date": "10.12.2022",
    "greeting": "Hello, World!",
    "greeting_date": "2022-12-10 11:54:18",
    "hello_prize": "23,500.00",
    "name": "Arquillian",
    "version": 1
  },
  {
```

```
    "date_created": "2022-12-10 11:54:18",
    "entity_date": "10.12.2022",
    "greeting": "Hello Java!",
    "greeting_date": "2022-12-10 11:54:18",
    "hello_prize": "14,500.00",
    "name": "Payara",
    "version": 1
  }
]
```


The generated JSON can equally be read back to a generic collection, much like was done with reading back to a single instance in Listing 1-10. Listing 1-13 shows reading back the JSON list shown in Listing 1-12 into a generic `java.util`.List collection.

Listing 1-13 showing mapping from JSON list to a generic collection.

Listing 1-13

```
 try (var inputStream = HelloResourceJaxRsSeTest.class.getResourceAsStream("/
data/helloEntityList.json")) {
                List<HelloEntity> helloListFromJson = jsonb.
fromJson(inputStream, new TypeLiteral<List<HelloEntity>>() {
                }.getType());

                assertEquals(helloListFromJson.size(), helloEntityList.size());

            }
```

Notice for the second parameter to the Jsonb#fromJson method, the generic List<HelloEntity> is wrapped in a `jakarta.enterprise.util.TypeLiteral`. This is a CDI construct that supports inline instantiation of objects that represent parameterized types with actual type parameters. The `helloListFromJson` variable is now instantiated to a list of HelloEntities as read from the JSON file.

Mapping Java Records

Java 14 introduced records - a unified way to create immutable data objects without all the ceremonial code. JSON-B currently has support for records (with some caveats). Listing 1-14 shows a `HelloRecord` class.

Listing 1-14

```
public record HelloRecord(String name,
                          String greeting,
                          @JsonbProperty("greeting_date")
                          LocalDateTime greetingDate,
                          @JsonbProperty("date_created")
                          LocalDateTime dateCreated,
                          @JsonbProperty("hello_price")
                          BigDecimal helloPrice,
                          @JsonbDateFormat("dd.MM.yyyy")
                          @JsonbProperty("entity_date")
                          LocalDate entityDate) {
    @JsonbCreator
    public HelloRecord {
    }
}
```

This record is the same as the `HelloEntity` class. However, as part of the constructor initialization, we pass in some JSON-B annotations to customise how the fields should be serialised and deserialized. This is needed to work consistently across implementations as we set the `propertyNamingStrategy` to `PropertyNamingStrategy.LOWER_CASE_WITH_UNDERSCORES`. Without the `@JsonbProperty` annotation passing in the fields names, eg ("greeting_date"), JSON-B can serialise to JSON but not deserialize from it because it will be expecting the `camelCase` field names in the JSON string. We also need to pass in a custom implementation of the **jakarta.json.bind.config.PropertyVisibilityStrategy** to Jsonb through the `JsonbConfig` as was done in Listing 1-6. The `CustomPropertyVisibilityStrategy` is shown in Listing 1-15.

Listing 1-15 showing our custom PropertyVisibilityStrategy implementation

```
Listing 1-15
public class CustomPropertyVisibilityStrategy implements
PropertyVisibilityStrategy {
    @Override
    public boolean isVisible(Field field) {
        return true;
    }

    @Override
    public boolean isVisible(Method method) {
        return false;
    }
}
```

This is needed because JSON-B, by default, looks for the JavaBean getter/setter method styles of getXXX and setXXX. But records don't have that. So we implement the visibility strategy to cater for that. Listing 1-14 also shows the compact `HelloRecord` constructor annotated with `@JsonbCreator`. This annotation identifies the custom constructor or factory method to use when creating an instance of the associated class. With everything in place, `HelloRecord` can be converted to and from JSON as shown in Listing 1-16.

Listing 1-16 shows converting HelloRecord to and from JSON

```
Listing 1-16
var helloRecord = new HelloRecord("Arquillian", "Hello, World!", LocalDateTime.
now(ZoneOffset.UTC),
                    LocalDateTime.now(ZoneOffset.UTC), new BigDecimal("25000"),
LocalDate.now(ZoneOffset.UTC));
            var recordJson = jsonb.toJson(helloRecord);
            HelloRecord helloRecord1 = jsonb.fromJson(recordJson, HelloRecord.
class);

            assertEquals(helloRecord.name(), helloRecord1.name());
            assertEquals(helloRecord.greeting(), helloRecord1.greeting());
            assertEquals(helloRecord.helloPrice(), helloRecord1.helloPrice());
            assertEquals(helloRecord.entityDate(), helloRecord1.entityDate());
```

Listing 1-16 shows creating a `HelloRecord` object, converting it to a JSON string, then converting that JSON string back to another `HelloRecord` instance. The code then makes some assertions to confirm both objects have the same values.

Custom Adapters

JSON-B adapters allow you to customise the mapping process. Though you can customise how an object is mapped to and from JSON with the constructs discussed so far, there are times when you don't have control over the class, or need some fine grained control over the process. For instance you might want to marshal and unmarshal only specific fields of a class in some contexts. For such cases, you can register an implementation of `jakarta.json.bind.adapter.JsonbAdapter` when configuring Jsonb. Listing 1-17 shows a custom JsonbAdapter that converts only specific fields of the `HelloEntity` to and from JSON.

Listing 1-17 showing custom JsonbAdapter implementation

Listing 1-17

```
public class CustomAdapter implements JsonbAdapter<HelloEntity, JsonObject> {
    @Override
    public JsonObject adaptToJson(final HelloEntity obj) throws Exception {
        return Json.createObjectBuilder()
                .add("name", obj.getName())
                .add("greeting", obj.getGreeting())
                .add("id", obj.getId())
                .add("greeting_date", obj.getGreetingDate().toString())
                .build();
    }
    @Override
    public HelloEntity adaptFromJson(final JsonObject obj) throws Exception {
        var helloEntity = new HelloEntity();
        helloEntity.setName(obj.getString("name"));
        helloEntity.setGreeting(obj.getString("greeting"));
        helloEntity.setId(Long.valueOf(obj.get("id").toString()));
        helloEntity.setGreetingDate(LocalDateTime.parse(obj.
getString("greeting_date")));

        return helloEntity;
    }
}
```

The custom implementation takes a subset of the `HellEntity` fields and converts to a `jakarta.json.JsonObject` in the `adaptToJson` method of the interface. The inverse is done in the `adaptFromJson` method to convert from `JsonObject` to a `HelloEntity` instance. The `CustomAdapter` can now be passed to Jsonb through `JsonbConfig` and used for converting to and from JSON as shown in Listing 1-18.

Listing 1-18 shows passing `CustomAdapter` to Jsonb through `JsonbConfig`

Listing 1-18

```
JsonbConfig jsonbConfig = new JsonbConfig()
               .withNullValues(true)
               .withFormatting(true)
               .withAdapters(new CustomAdapter());

       try (Jsonb jsonb = JsonbBuilder.create(jsonbConfig)) {
           var json = jsonb.toJson(helloEntity);
           log.log(Level.INFO, () -> json);

           var convertedEntity = jsonb.fromJson(json, HelloEntity.class);
           assertEquals(helloEntity.getName(), convertedEntity.getName());
           assertEquals(helloEntity.getGreeting(), convertedEntity.
getGreeting());
           assertEquals(helloEntity.getGreetingDate(), convertedEntity.
getGreetingDate());
       }
```

Listing 1-18 shows the passing of our `CustomAdapter` to Jsonb through `JsonbConfig`. We then create a JSON string from a `HelloEntity` instance which we then convert back. We then make some assertions on the generated objects to be sure they're the same. Listing 1-19 shows the generated JSON from the call in Listing 1-18.

Listing 1-19 showing the generated JSON

Listing 1-19

```
{
   "name": "Arquillian",
   "greeting": "Hello, World!",
   "id": 1,
   "greeting_date": "2022-12-10T22:21:24.773712320"
}
```

Use Cases

So far we have discussed the use of Jakarta JSON-B to convert Java objects to and from JSON. You might be wondering what use cases suit the use of this API. First it's the default marshaller for Jakarta REST. JSON-B is the underlying API for converting to and from RESTful resources created using Jakarta REST. Other use cases include:

Test Resources

You can use JSON-B to read test data from JSON files to create repeatable data for test cases. As part of creating reliable, consistent and repeatable tests, it is good practice to have consistent test data for each scenario. One way that can be achieved is through the provision of test data as JSON data that is read to instantiate test artefacts at runtime. These JSON files can be read using JSON-B.

Simpler Data Transfer

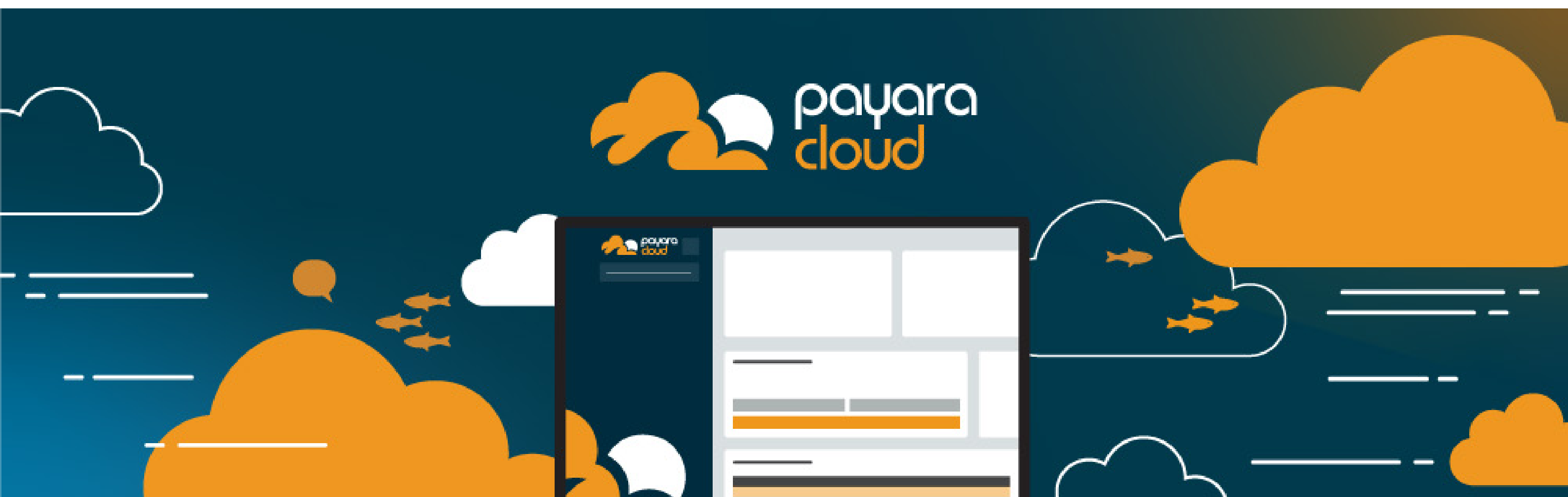
Objects can be converted to JSON strings, passed around an application much more conveniently and converted back to the Java type at the final point of use. Some very complex objects can also be converted to JSON and stored in a database, as most databases support JSON columns.

Better REST Customization

Knowing JSON-B will help you customise your REST resource return objects to better suit your consuming clients. As the default API for converting to and from JSON in Jakarta REST, you can easily extend and customise the marshalling and unmarshalling process much more easily with an understanding of how the API works.

Summary

In this guide, we looked at the theoretical foundations of Jakarta EE, what it is, what a specification is and how Jakarta EE relates to Eclipse MicroProfile. We then looked at how to get started with Jakarta EE and MicroProfile in a Maven application. We then took a look at Jakarta JSON-B, how to customise the serialisation and deserialisation process, mapping single and collection objects and records and finally briefly looked at some use cases for JSON-B.



CLICK HERE

sales@payara.fish

UK: +44 800 538 5490
Intl: +1 888 239 8941

www.payara.fish

Payara Services Ltd 2022 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ