Jackson Databind 3.1.3 Annotations ��? Testing Professor

Jackson Databind 3.1.3 — Complete Annotations Reference

> **Library:** tools.jackson.core:jackson-databind:3.1.3 **Annotation package (Jackson 3.x):** tools.jackson.annotation.* (core annotations) and tools.jackson.databind.annotation.* (databind-specific annotations). Jackson 3 renamed the root package from com.fasterxml.jackson to tools.jackson . The annotation set is largely the same as Jackson 2.x but lives under the new package.
>
> Maven coordinates:
>
> ```
> <dependency>
>   <groupId>tools.jackson.core</groupId>
>   <artifactId>jackson-databind</artifactId>
>   <version>3.1.3</version>
> </dependency>
> ```
>
> jackson-databind transitively brings in jackson-core and jackson-annotations (also under tools.jackson.core).

Table of Contents

    - Meta: `@JacksonAnnotation`, `@JacksonAnnotationsInside`, `@JacksonInject`, `@JacksonStdImpl`
2. Databind-specific annotations (`tools.jackson.databind.annotation`)
    - `@JsonSerialize`, `@JsonDeserialize`, `@JsonNaming`, `@JsonPOJOBuilder`, `@JsonAppend`, `@JsonTypeIdResolver`, `@JsonValueInstantiator`, `@NoClass`, `@EnumNaming`
3. Cross-cutting concerns
    - Inclusion rules, null handling, defaults, formats, locales, time zones
    - Polymorphic deserialization safety (default typing, `PolymorphicTypeValidator`)
    - Mix-in annotations
4. Complete worked examples

---

1. Core annotations (`tools.jackson.annotation`)

1.1 `@JsonProperty`

Marks a non-static method or field as a logical JSON property. Lets you customize the name, required flag, default value, access, and index.

Targets: field, method (getter/setter), constructor parameter, parameter.

Element	Type	Purpose
value	String	Logical JSON property name. Default: use Java name.
namespace	String	Namespace (used by formats like XML).
required	boolean	Whether the property must be present on deserialization.
index	int	Optional ordering hint.
defaultValue	String	Documentation-only default (not applied).
access	JsonProperty.Access	`AUTO`, `READ_ONLY`, `WRITE_ONLY`, `READ_WRITE`.

```
public class User {
   @JsonProperty("user_id")
   private long id;

   @JsonProperty(value = "email", required = true)
   private String email;

   @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
   private String password; // serialized OUT skipped, read IN accepted

   @JsonProperty(access = JsonProperty.Access.READ_ONLY)
   private Instant createdAt; // serialized OUT, ignored on input
}
```

JSON in/out:

```
{ "user_id": 7, "email": "a@b.com", "password": "secret" }
```

1.2 `@JsonAlias`

Defines alternative names accepted **`only during deserialization`**.

```
public class Person {
   @JsonProperty("name")
   @JsonAlias({"fullName", "full_name", "n"})
   public String name;
}
```

All of `{"name":"X"}`, `{"fullName":"X"}`, `{"full_name":"X"}`, `{"n":"X"}` deserialize successfully. Serialization always emits `"name"`.

1.3 `@JsonGetter` / `@JsonSetter`

Lower-level specializations of `@JsonProperty` for a single accessor.

```
public class Box {
   private int w, h;

   @JsonGetter("width")  public int getW() { return w; }
   @JsonSetter("width")  public void setW(int w) { this.w = w; }

   @JsonGetter("height") public int getH() { return h; }
   @JsonSetter("height") public void setH(int h) { this.h = h; }
}
```

`@JsonSetter` also supports `nulls` and `contentNulls` of type `Nulls`:

```
public class Order {
   @JsonSetter(nulls = Nulls.SKIP)      // ignore explicit null
   public String coupon;

   @JsonSetter(nulls = Nulls.AS_EMPTY)  // replace null with empty
   public List<String> tags = new ArrayList<>();

   @JsonSetter(nulls = Nulls.FAIL)      // throw on null
   public String customerId;

   @JsonSetter(contentNulls = Nulls.SKIP) // ignore nulls inside collection
   public List<String> items;
}
```

`Nulls` values: `SET`, `SKIP`, `FAIL`, `AS_EMPTY`, `DEFAULT`.

1.4 `@JsonAnyGetter` / `@JsonAnySetter`

Serialize/deserialize arbitrary, dynamic properties into a Map (or via paired methods).

```java
public class DynamicBag {
   public String fixed;

   private final Map<String, Object> extras = new LinkedHashMap<>();

   @JsonAnyGetter
   public Map<String, Object> getExtras() { return extras; }

   @JsonAnySetter
   public void put(String key, Object value) { extras.put(key, value); }
}
```

Input `{"fixed":"f","a":1,"b":"x"}` → `fixed="f"`, `extras={a=1, b="x"}`. Output flattens `extras` into the top-level JSON object.

`@JsonAnySetter` may also be placed on a `Map` field directly (Jackson 3 supports this on fields).

1.5 `@JsonKey` / `@JsonValue`

- `@JsonValue` — single method whose return value represents the entire serialized form of the object (typical for value wrappers/enums).
- `@JsonKey` — same idea but used when the object is a **`Map` key**.

```java
public final class Email {
   private final String address;
   public Email(String s) { this.address = s; }

   @JsonValue
   public String asString() { return address; } // serializes as "a@b.com"

   @JsonKey
   public String asKey() { return address.toLowerCase(); }
}
```

For deserialization, pair with `@JsonCreator`:

```java
public final class Email {
   private final String address;

   @JsonCreator
   public Email(String address) { this.address = address; }

   @JsonValue
   public String asString() { return address; }
}
```

1.6 `@JsonIgnore`

Excludes a field/getter/setter from both serialization and deserialization.

```
public class Account {
   public String username;
   @JsonIgnore public String passwordHash;
}
```

1.7 `@JsonIgnoreProperties`

Class- or property-level. Two main uses:

(a) `Class-level` — ignore listed properties and/or unknown properties.

```
@JsonIgnoreProperties(value = {"internalId", "debug"}, ignoreUnknown = true, allowGetters = false,
          allowSetters = false)
public class Product { ... }
```

Element	Meaning
value	Names to ignore in both directions.
ignoreUnknown	If `true`, unknown JSON properties on deserialization are silently skipped.
allowGetters	If `true`, listed properties may still be serialized.
allowSetters	If `true`, listed properties may still be deserialized.

(b) `Property-level` — ignore properties of the referenced object.

```
public class Order {
   @JsonIgnoreProperties({"orders"}) // break cycle when serializing User
   public User user;
}
```

1.8 `@JsonIgnoreType`

Class-level marker: all properties of the marked type are ignored anywhere it appears.

```
@JsonIgnoreType
public class InternalAuditTrail { ... }
```

1.9 `@JsonAutoDetect`

Configures visibility per accessor kind for a class.

```
@JsonAutoDetect(
   fieldVisibility    = JsonAutoDetect.Visibility.ANY,
   getterVisibility   = JsonAutoDetect.Visibility.NONE,
   isGetterVisibility = JsonAutoDetect.Visibility.NONE,
```

```
    setterVisibility   = JsonAutoDetect.Visibility.NONE,
    creatorVisibility  = JsonAutoDetect.Visibility.PUBLIC_ONLY
)
public class FieldOnly { private int a; private String b; }
```

`Visibility` values: `ANY`, `NON_PRIVATE`, `PROTECTED_AND_PUBLIC`, `PUBLIC_ONLY`, `NONE`, `DEFAULT`.

1.10 `@JsonInclude`

Controls which property values are written.

```
@JsonInclude(JsonInclude.Include.NON_NULL)
public class Dto {
   public String a;       // omitted if null
   public List<String> tags; // included always
}
```

Per-property override and content rules:

```
public class Dto {
   @JsonInclude(JsonInclude.Include.NON_EMPTY)
   public String name;        // omitted if null/empty

   @JsonInclude(value = JsonInclude.Include.NON_DEFAULT)
   public int retries;        // omitted if 0

   @JsonInclude(
      value       = JsonInclude.Include.ALWAYS,
      content     = JsonInclude.Include.NON_NULL // for Map values / collection elements
   )
   public Map<String, String> attrs;

   @JsonInclude(value = JsonInclude.Include.CUSTOM, valueFilter = ZeroIsAbsent.class)
   public int score;
}

public static class ZeroIsAbsent {
   @Override public boolean equals(Object other) { return other == null ||
           Integer.valueOf(0).equals(other); }
}
```

`Include` values: `ALWAYS`, `NON_NULL`, `NON_ABSENT`, `NON_EMPTY`, `NON_DEFAULT`, `CUSTOM`, `USE_DEFAULTS`.

1.11 `@JsonIncludeProperties`

Class- or property-level allow-list — the inverse of `@JsonIgnoreProperties`.

```
@JsonIncludeProperties({"id", "name"})
public class Slim { public long id; public String name; public String secret; }
```

Only `id` and `name` are visible to Jackson.

1.12 `@JsonCreator`

Marks a constructor or static factory method as the deserialization entry point.

```
public final class Money {
   public final String currency;
   public final BigDecimal amount;

   @JsonCreator
   public Money(
      @JsonProperty("currency") String currency,
      @JsonProperty("amount")   BigDecimal amount
   ) {
      this.currency = currency;
      this.amount   = amount;
   }
}
```

`Delegating` creator (single argument, no `@JsonProperty`):

```
public final class Email {
   private final String value;

   @JsonCreator(mode = JsonCreator.Mode.DELEGATING)
   public Email(String value) { this.value = value; }
}
```

`JsonCreator.Mode` : `DEFAULT` , `DELEGATING` , `PROPERTIES` , `DISABLED` .

Static factory:

```
public final class Point {
   public final int x, y;
   private Point(int x, int y) { this.x = x; this.y = y; }

   @JsonCreator
   public static Point of(@JsonProperty("x") int x, @JsonProperty("y") int y) {
      return new Point(x, y);
   }
}
```

> Jackson 3 detects all-arg canonical constructors of Java `record` types automatically; `@JsonCreator` is needed only when overriding the default.

1.13 `@JsonPropertyOrder`

Defines serialization order of properties.

```
@JsonPropertyOrder({"id", "name", "email"})
public class User { ... }

@JsonPropertyOrder(alphabetic = true)
public class Sorted { ... }
```

`alphabetic` puts unlisted properties after listed ones in alphabetic order.

1.14 `@JsonFormat`

Controls formatting for a value, especially dates, numbers and enums.

```
public class Event {
   @JsonFormat(shape = JsonFormat.Shape.STRING,
            pattern = "yyyy-MM-dd'T'HH:mm:ssXXX",
            timezone = "UTC",
            locale = "en_US")
   public OffsetDateTime when;

   @JsonFormat(shape = JsonFormat.Shape.STRING)
   public BigDecimal price;  // "12.34" instead of number

   @JsonFormat(shape = JsonFormat.Shape.OBJECT)
   public EnumStatus status; // serialize enum as object using its bean props

   @JsonFormat(shape = JsonFormat.Shape.NUMBER_INT)
   public DayOfWeek dow;     // 1..7

   @JsonFormat(with = { JsonFormat.Feature.WRITE_SINGLE_ELEM_ARRAYS_UNWRAPPED },
            without = { JsonFormat.Feature.ACCEPT_CASE_INSENSITIVE_PROPERTIES })
   public List<String> tags;
}
```

`Shape` values: `ANY`, `NATURAL`, `SCALAR`, `ARRAY`, `OBJECT`, `NUMBER`, `NUMBER_FLOAT`, `NUMBER_INT`, `STRING`, `BOOLEAN`, `POJO`, `BINARY`.

Common `JsonFormat.Feature`s: `ACCEPT_SINGLE_VALUE_AS_ARRAY`, `WRITE_SINGLE_ELEM_ARRAYS_UNWRAPPED`, `WRITE_DATES_WITH_ZONE_ID`, `WRITE_DATE_TIMESTAMPS_AS_NANOSECONDS`, `READ_DATE_TIMESTAMPS_AS_NANOSECONDS`, `WRITE_SORTED_MAP_ENTRIES`, `ACCEPT_CASE_INSENSITIVE_PROPERTIES`, `ACCEPT_CASE_INSENSITIVE_VALUES`, `WRITE_ENUMS_USING_TO_STRING`, `READ_ENUMS_USING_TO_STRING`, `READ_UNKNOWN_ENUM_VALUES_AS_NULL`, `READ_UNKNOWN_ENUM_VALUES_USING_DEFAULT_VALUE`, `ADJUST_DATES_TO_CONTEXT_TIME_ZONE`.

1.15 `@JsonRawValue`

Embed the property's String value as **`raw JSON`** without quoting/escaping.

```
public class Wrapper {
   public String name = "x";

   @JsonRawValue
   public String payload = "{\"already\":\"json\"}";
}
```

Output: `{"name":"x","payload":{"already":"json"}}`.

1.16 `@JsonUnwrapped`

Inlines all properties of a nested object into the enclosing one.

```
public class Address {
   public String street, city;
}

public class Customer {
   public String name;
   @JsonUnwrapped(prefix = "addr_") public Address address;
}
```

Output: `{"name":"Ada","addr_street":"...","addr_city":"..."}`. Optional `suffix` element is also available.

1.17 `@JsonMerge`

For deserialization into an existing object: merges JSON values into the current value instead of replacing.

```
public class Settings {
   @JsonMerge
   public Map<String, String> flags = new HashMap<>(Map.of("debug", "false"));
}
```

`OptBoolean` field `value()`: `TRUE` (default), `FALSE`, `DEFAULT`.

1.18 `@JsonAppend`

Class-level (in `tools.jackson.databind.annotation`) that **`virtually`** appends properties during serialization.

```
@JsonAppend(attrs = { @JsonAppend.Attr(value = "tenantId") })
public class Audited {
   public String name;
}

// usage:
```

```
mapper.writer().withAttribute("tenantId", "acme").writeValueAsString(new Audited());
// {"name":null,"tenantId":"acme"}
```

Also supports `props = @JsonAppend.Prop(...)` with a custom `VirtualBeanPropertyWriter`.

1.19 `@JsonPropertyDescription`

Documentation-only annotation (used by JSON-Schema modules) describing a property.

```
public class User {
   @JsonPropertyDescription("Unique e-mail address used as login")
   public String email;
}
```

1.20 `@JsonTypeInfo` + `@JsonSubTypes` + `@JsonTypeName` + `@JsonTypeId`

Polymorphic type handling.

```
@JsonTypeInfo(
   use      = JsonTypeInfo.Id.NAME,           // discriminator strategy
   include  = JsonTypeInfo.As.PROPERTY,         // where it appears
   property = "kind",                      // property name
   defaultImpl = Unknown.class,              // fallback type
   visible  = true                        // keep discriminator in POJO
)
@JsonSubTypes({
   @JsonSubTypes.Type(value = Dog.class, name = "dog"),
   @JsonSubTypes.Type(value = Cat.class, name = "cat")
})
public abstract class Animal {
   public String kind;
}

@JsonTypeName("dog")
public class Dog extends Animal { public boolean goodBoy = true; }

@JsonTypeName("cat")
public class Cat extends Animal { public int lives = 9; }
```

Serialized: `{"kind":"dog","goodBoy":true}`.

`JsonTypeInfo.Id`:

Id	Discriminator value used
`CLASS`	Fully qualified class name
`MINIMAL_CLASS`	Class name relative to base class
`NAME`	Logical name (`@JsonTypeName` / `@JsonSubTypes.Type.name`)
`SIMPLE_NAME`	Java simple class name

Id	Discriminator value used
CUSTOM	Resolved by @JsonTypeIdResolver
NONE	No type info
DEDUCTION	Deduce subtype from set of present properties

JsonTypeInfo.As :

As	Where discriminator goes
PROPERTY	Sibling property inside the object
EXISTING_PROPERTY	Reuse an already-declared property
WRAPPER_OBJECT	{ "dog": { ... } }
WRAPPER_ARRAY	["dog", { ... }]
EXTERNAL_PROPERTY	Sibling property on the enclosing object

@JsonTypeId marks a property whose value should be used as the type id (instead of writing the discriminator separately).

```
public class Shape {
   @JsonTypeId public String type;
}
```

1.21 @JsonManagedReference / @JsonBackReference

Break parent↔child cycles by serializing one side only.

```
public class Parent {
   public String name;
   @JsonManagedReference
   public List<Child> children = new ArrayList<>();
}
public class Child {
   public String name;
   @JsonBackReference
   public Parent parent;       // omitted from JSON, restored on deserialize
}
```

Multiple references can be named: @JsonManagedReference("orders") / @JsonBackReference("orders") .

1.22 @JsonIdentityInfo / @JsonIdentityReference

Object identity to handle cycles by reference rather than by inlining.

```java
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property  = "id"
)
public class Node {
    public long id;
    public String name;
    public Node next;
}
```

Other generators: `IntSequenceGenerator`, `UUIDGenerator`, `StringIdGenerator`, `None`.

Force only the id (always reference, never inline):

```java
public class Holder {
    @JsonIdentityReference(alwaysAsId = true)
    public Node node;
}
```

1.23 `@JsonView`

Conditional inclusion by view tag.

```java
public class Views {
    public interface Public {}
    public interface Internal extends Public {}
}

public class User {
    @JsonView(Views.Public.class)   public long id;
    @JsonView(Views.Public.class)   public String name;
    @JsonView(Views.Internal.class) public String email;
}

String publicJson  = mapper.writerWithView(Views.Public.class).writeValueAsString(u);
String internalJson= mapper.writerWithView(Views.Internal.class).writeValueAsString(u);
```

`MapperFeature.DEFAULT_VIEW_INCLUSION` controls whether unannotated properties are included by default.

1.24 `@JsonFilter`

Class-level marker that binds the class to a runtime filter id.

```java
@JsonFilter("userFilter")
public class User { public long id; public String name; public String email; }

SimpleFilterProvider filters = new SimpleFilterProvider()
    .addFilter("userFilter", SimpleBeanPropertyFilter.filterOutAllExcept("id", "name"));
```

```
String json = mapper.writer(filters).writeValueAsString(user);
```

1.25 `@JsonEnumDefaultValue`

Marks an enum constant as the fallback for unknown values (requires `READ_UNKNOWN_ENUM_VALUES_USING_DEFAULT_VALUE`).

```
public enum Status {
   NEW, OPEN, CLOSED,
   @JsonEnumDefaultValue UNKNOWN
}

mapper.coercionConfigDefaults()
    .setFeature(JsonFormat.Feature.READ_UNKNOWN_ENUM_VALUES_USING_DEFAULT_VALUE, true);
```

1.26 Meta-annotations

`@JacksonAnnotation`

Root marker every Jackson annotation carries. You generally do not use it directly.

`@JacksonAnnotationsInside`

Composes multiple Jackson annotations into a single custom one.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
@JacksonAnnotationsInside
@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
public @interface JsonDateOptional {}
```

`@JacksonInject`

Inject a value from `InjectableValues` rather than from JSON.

```
public class Request {
   @JacksonInject("requestId") public String requestId;
   public String body;
}

mapper.reader()
    .with(new InjectableValues.Std().addValue("requestId", "abc-123"))
    .forType(Request.class)
    .readValue("{\"body\":\"hi\"}");
```

Optional element `useInput = OptBoolean.FALSE` forces the injected value to win over JSON.

`@JacksonStdImpl`

Marker for built-in (de)serializer implementations. Internal — do not annotate your own code unless extending Jackson.

---

2. Databind-specific annotations

Package: `tools.jackson.databind.annotation`.

2.1 `@JsonSerialize`

Configure a custom serializer or modify serialization metadata.

```
public class Money {
   @JsonSerialize(using = MoneySerializer.class)
   public BigDecimal amount;

   @JsonSerialize(as = Number.class)       // serialize using static type
   public BigDecimal raw;

   @JsonSerialize(contentUsing = MoneySerializer.class) // for collection elements
   public List<BigDecimal> amounts;

   @JsonSerialize(keyUsing = LocaleKeySerializer.class) // for Map keys
   public Map<Locale, String> labels;

   @JsonSerialize(nullsUsing = EmptyStringSerializer.class)
   public String maybe;

   @JsonSerialize(converter = TrimConverter.class) // pre-process before default ser
   public String name;

   @JsonSerialize(typing = JsonSerialize.Typing.STATIC) // ignore runtime type
   public Object value;
}
```

Elements: `using`, `contentUsing`, `keyUsing`, `nullsUsing`, `as`, `keyAs`, `contentAs`, `typing`, `converter`, `contentConverter`, `include` (deprecated — prefer `@JsonInclude`).

2.2 `@JsonDeserialize`

Configure a custom deserializer or per-property behavior.

```
public class Money {
   @JsonDeserialize(using = MoneyDeserializer.class)
   public BigDecimal amount;

   @JsonDeserialize(as = LinkedHashMap.class, keyAs = String.class, contentAs = Integer.class)
   public Map<String, Integer> counts;

   @JsonDeserialize(contentUsing = MoneyDeserializer.class)
```

```
    public List<BigDecimal> amounts;

    @JsonDeserialize(keyUsing = LocaleKeyDeserializer.class)
    public Map<Locale, String> labels;

    @JsonDeserialize(converter = TrimConverter.class)
    public String name;

    @JsonDeserialize(builder = OrderBuilder.class)  // builder pattern
    public Order order;
}
```

Elements: `using`, `contentUsing`, `keyUsing`, `as`, `keyAs`, `contentAs`, `builder`, `converter`, `contentConverter`.

2.3 `@JsonNaming`

Class-level naming strategy override.

```
@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
public class User { public String firstName; public String lastName; }
// → {"first_name":"...","last_name":"..."}
```

Built-in strategies (in `tools.jackson.databind.PropertyNamingStrategies`):

Strategy class	Example transform
`SnakeCaseStrategy`	`firstName` → `first_name`
`UpperSnakeCaseStrategy`	`firstName` → `FIRST_NAME`
`LowerCamelCaseStrategy`	unchanged (default)
`UpperCamelCaseStrategy`	`firstName` → `FirstName`
`LowerCaseStrategy`	`firstName` → `firstname`
`KebabCaseStrategy`	`firstName` → `first-name`
`LowerDotCaseStrategy`	`firstName` → `first.name`

You can also pass any subclass of `PropertyNamingStrategy`.

2.4 `@JsonPOJOBuilder`

Configures a builder used with `@JsonDeserialize(builder = ...)`.

```
@JsonDeserialize(builder = User.Builder.class)
public class User {
    private final String name;
    private final int    age;

    private User(Builder b) { this.name = b.name; this.age = b.age; }
```

```
    @JsonPOJOBuilder(withPrefix = "set", buildMethodName = "build")
    public static class Builder {
        private String name; private int age;
        public Builder setName(String n) { this.name = n; return this; }
        public Builder setAge(int a)    { this.age = a;  return this; }
        public User build()           { return new User(this); }
    }
}
```

`withPrefix` defaults to `"with"`, `buildMethodName` defaults to `"build"`.

2.5 `@JsonTypeIdResolver`

Plug a custom resolver for polymorphic type ids (paired with `@JsonTypeInfo(use = CUSTOM)`).

```
@JsonTypeInfo(use = JsonTypeInfo.Id.CUSTOM, include = JsonTypeInfo.As.PROPERTY, property = "kind")
@JsonTypeIdResolver(AnimalTypeIdResolver.class)
public abstract class Animal { }

public class AnimalTypeIdResolver extends TypeIdResolverBase {
    @Override public String idFromValue(Object value) { return
            value.getClass().getSimpleName().toLowerCase(); }
    @Override public String idFromValueAndType(Object value, Class<?> type) { return
            idFromValue(value); }
    @Override public JavaType typeFromId(DatabindContext ctxt, String id) {
        return ctxt.constructType(switch (id) {
            case "dog" -> Dog.class;
            case "cat" -> Cat.class;
            default    -> Unknown.class;
        });
    }
    @Override public JsonTypeInfo.Id getMechanism() { return JsonTypeInfo.Id.CUSTOM; }
}
```

2.6 `@JsonValueInstantiator`

Plug a custom `ValueInstantiator` for constructing values.

```
@JsonValueInstantiator(MyInstantiator.class)
public class Special { ... }
```

2.7 `@JsonAppend` (databind)

Already covered in §1.18.

2.8 `@NoClass`

Placeholder class used as the default for `Class<?>` annotation elements (e.g. `@JsonDeserialize(as = ...)`) when "no value" is intended. Generally not used directly.

2.9 `@EnumNaming` *(Jackson 3 addition)*

Applies a naming strategy to enum constants for serialization/deserialization.

```
@EnumNaming(EnumNamingStrategies.CamelCaseStrategy.class)
public enum HttpMethod { GET, POST, PUT, DELETE }
```

Strategies live in `tools.jackson.databind.EnumNamingStrategies` (e.g. `CamelCaseStrategy`).

---

3. Cross-cutting concerns

3.1 Inclusion rules summary

Include	Property written when…
ALWAYS	always
NON_NULL	value is not `null`
NON_ABSENT	value is non-null and (if `Optional` / `AtomicReference`) is present
NON_EMPTY	not null, not empty `String` / `Collection` / `Map` /array
NON_DEFAULT	not equal to default of declaring type / default constructor value
CUSTOM	custom `valueFilter` / `contentFilter` returns `false` from `equals`
USE_DEFAULTS	inherit from upper scope (class → global)

3.2 Polymorphic deserialization safety

In Jackson 3 the legacy `enableDefaultTyping` is removed. Use:

```
PolymorphicTypeValidator ptv = BasicPolymorphicTypeValidator.builder()
    .allowIfSubType(Animal.class)
    .build();

JsonMapper mapper = JsonMapper.builder()
    .activateDefaultTyping(ptv, DefaultTyping.NON_FINAL, JsonTypeInfo.As.PROPERTY)
    .build();
```

Always prefer explicit `@JsonTypeInfo` + `@JsonSubTypes` over default typing.

3.3 Mix-in annotations

When you cannot modify a class, attach annotations through a mix-in:

```
public abstract class UserMixIn {
   @JsonProperty("user_id") abstract long getId();
   @JsonIgnore          abstract String getPasswordHash();
}

JsonMapper mapper = JsonMapper.builder()
   .addMixIn(User.class, UserMixIn.class)
   .build();
```

3.4 Property accessors on `record` types

Jackson 3 has first-class `record` support. Annotations may be placed on the record components:

```
public record User(
   @JsonProperty("user_id") long id,
   @JsonProperty(required = true) String email
) {}
```

---

4. Complete worked examples

4.1 Full ObjectMapper bootstrap (Jackson 3 style)

```
import tools.jackson.databind.*;
import tools.jackson.databind.json.JsonMapper;
import tools.jackson.databind.cfg.DeserializationFeature;
import tools.jackson.databind.cfg.SerializationFeature;
import tools.jackson.databind.PropertyNamingStrategies;
import tools.jackson.datatype.jsr310.JavaTimeModule; // separate module

JsonMapper mapper = JsonMapper.builder()
   .addModule(new JavaTimeModule())
   .propertyNamingStrategy(PropertyNamingStrategies.SNAKE_CASE)
   .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false)
   .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
   .serializationInclusion(JsonInclude.Include.NON_NULL)
   .build();
```

4.2 Putting many annotations together

```
@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonPropertyOrder({"id", "type", "name"})
@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type", include =
          JsonTypeInfo.As.EXISTING_PROPERTY, visible = true)
@JsonSubTypes({
   @JsonSubTypes.Type(value = AdminUser.class, name = "admin"),
   @JsonSubTypes.Type(value = GuestUser.class, name = "guest")
})
public abstract class BaseUser {
```

```
    @JsonProperty(value = "id", required = true)
    public long id;

    public String type;

    @JsonAlias({"full_name", "displayName"})
    public String name;

    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ssXXX")
    public OffsetDateTime createdAt;

    @JsonIgnore
    public String internalToken;

    @JsonAnyGetter
    public Map<String, Object> getAttrs() { return attrs; }
    @JsonAnySetter
    public void putAttr(String k, Object v) { attrs.put(k, v); }
    private final Map<String, Object> attrs = new LinkedHashMap<>();
}

@JsonTypeName("admin")
public class AdminUser extends BaseUser {
    @JsonProperty("admin_level") public int level;
}

@JsonTypeName("guest")
public class GuestUser extends BaseUser {
    @JsonProperty(access = JsonProperty.Access.READ_ONLY)
    public Instant expiresAt;
}
```

Sample serialized form for an `AdminUser`:

```
{
  "id": 42,
  "type": "admin",
  "name": "Ada",
  "created_at": "2026-05-27T12:00:00Z",
  "admin_level": 9,
  "department": "platform"
}
```

4.3 Polymorphic deserialization end-to-end

```
String json = """
    [
      {"type":"admin","id":1,"name":"Ada","admin_level":9},
      {"type":"guest","id":2,"name":"G"}
    ]
    """;

List<BaseUser> users = mapper.readValue(json, new TypeReference<>() {});
```

4.4 Builder pattern

```
@JsonDeserialize(builder = Order.Builder.class)
public final class Order {
   private final String id;
   private final List<String> items;
   private Order(Builder b) { this.id = b.id; this.items = List.copyOf(b.items); }

   @JsonPOJOBuilder(withPrefix = "", buildMethodName = "build")
   public static final class Builder {
      private String id;
      private List<String> items = new ArrayList<>();
      public Builder id(String id) { this.id = id; return this; }
      public Builder items(List<String> i) { this.items = i; return this; }
      public Order build() { return new Order(this); }
   }
}
```

4.5 Views, filters and injection

```
mapper.writerWithView(Views.Public.class).writeValueAsString(user);

SimpleFilterProvider filters = new SimpleFilterProvider()
   .addFilter("userFilter", SimpleBeanPropertyFilter.serializeAllExcept("email"));
mapper.writer(filters).writeValueAsString(user);

mapper.reader()
     .with(new InjectableValues.Std().addValue("requestId", "abc"))
     .forType(Request.class)
     .readValue(json);
```

Appendix — quick alphabetic index

Annotation	Package	One-liner
`@JacksonAnnotation`	`tools.jackson.annotation`	Root marker for Jackson annotations
`@JacksonAnnotationsInside`	`tools.jackson.annotation`	Compose into custom meta-annotation
`@JacksonInject`	`tools.jackson.annotation`	Inject value during deserialization
`@JacksonStdImpl`	`tools.jackson.databind.annotation`	Internal marker for built-in (de)serializers
`@JsonAlias`	`tools.jackson.annotation`	Extra accepted names (deser only)
`@JsonAnyGetter` / `@JsonAnySetter`	`tools.jackson.annotation`	Dynamic properties

Annotation	Package	One-liner
`@JsonAppend`	`tools.jackson.databind.annotation`	Virtual properties on serialization
`@JsonAutoDetect`	`tools.jackson.annotation`	Per-accessor visibility
`@JsonBackReference` / `@JsonManagedReference`	`tools.jackson.annotation`	Parent/child cycle break
`@JsonCreator`	`tools.jackson.annotation`	Mark constructor/factory for deser
`@JsonDeserialize`	`tools.jackson.databind.annotation`	Custom deserializer / target types
`@JsonEnumDefaultValue`	`tools.jackson.annotation`	Fallback enum constant
`@JsonFilter`	`tools.jackson.annotation`	Bind class to runtime filter id
`@JsonFormat`	`tools.jackson.annotation`	Format value (shape/pattern/locale/tz)
`@JsonGetter` / `@JsonSetter`	`tools.jackson.annotation`	Lower-level @JsonProperty for accessors
`@JsonIdentityInfo` / `@JsonIdentityReference`	`tools.jackson.annotation`	Object identity for cycles
`@JsonIgnore` / `@JsonIgnoreProperties` / `@JsonIgnoreType`	`tools.jackson.annotation`	Exclude from binding
`@JsonInclude` / `@JsonIncludeProperties`	`tools.jackson.annotation`	Inclusion rules / allow-list
`@JsonKey` / `@JsonValue`	`tools.jackson.annotation`	Map-key / whole-object representation
`@JsonMerge`	`tools.jackson.annotation`	Merge instead of replace
`@JsonNaming`	`tools.jackson.databind.annotation`	Naming strategy
`@JsonPOJOBuilder`	`tools.jackson.databind.annotation`	Configure builder used by deser
`@JsonProperty`	`tools.jackson.annotation`	Logical property + name/access/required
`@JsonPropertyDescription`	`tools.jackson.annotation`	Description for tooling/schema
`@JsonPropertyOrder`	`tools.jackson.annotation`	Serialization order
`@JsonRawValue`	`tools.jackson.annotation`	Emit raw JSON text
`@JsonSerialize`	`tools.jackson.databind.annotation`	Custom serializer / target types

Annotation	Package	One-liner
`@JsonSubTypes` / `@JsonTypeInfo` / `@JsonTypeName` / `@JsonTypeId`	`tools.jackson.annotation`	Polymorphism
`@JsonTypeIdResolver`	`tools.jackson.databind.annotation`	Custom type-id resolver
`@JsonUnwrapped`	`tools.jackson.annotation`	Inline nested object's properties
`@JsonValueInstantiator`	`tools.jackson.databind.annotation`	Custom value instantiator
`@JsonView`	`tools.jackson.annotation`	Conditional inclusion by view
`@EnumNaming`	`tools.jackson.databind.annotation`	Naming strategy for enum constants
`@NoClass`	`tools.jackson.databind.annotation`	Placeholder Class marker

---

13. Popular Annotations Used in the REST Assured Framework

REST Assured itself is a **`fluent DSL`** for HTTP testing — it ships with virtually no annotations of its own. In real projects, however, REST Assured tests rely heavily on annotations that come from three neighbouring ecosystems:

1. **`TestNG`** or **`JUnit 4 / JUnit 5 (Jupiter)`** — to structure, parameterise and order the tests.
2. **`Jackson (this library)`** — to serialize POJOs sent in `.body(...)` and to deserialize responses via `.as(Pojo.class)`.
3. **`Lombok / Bean Validation / Allure`** — frequently combined to reduce boilerplate, validate payloads and produce reports.

Below is a categorised cheat-sheet of the annotations you will see in 95% of REST Assured codebases, with short worked examples.

13.1 Test Lifecycle Annotations

TestNG

Annotation	Purpose
`@Test`	Marks a method as a test case. Supports `priority`, `groups`, `dataProvider`, `dependsOnMethods`, `enabled`, `description`, `expectedExceptions`.

Annotation	Purpose
`@BeforeSuite` / `@AfterSuite`	Runs once before/after the entire suite — perfect place to set `RestAssured.baseURI`, `basePath`, filters.
`@BeforeClass` / `@AfterClass`	Runs once per test class — typical place to fetch an auth token.
`@BeforeMethod` / `@AfterMethod`	Runs before/after every `@Test` — reset request specifications.
`@BeforeTest` / `@AfterTest`	Runs around a `<test>` tag in `testng.xml`.
`@DataProvider`	Supplies parameter sets to a `@Test(dataProvider = "...")`.
`@Parameters`	Injects parameters declared in `testng.xml`.
`@Listeners`	Plug in custom `ITestListener` / `IReporter` (Allure, ExtentReports).

```java
import io.restassured.RestAssured;
import io.restassured.builder.RequestSpecBuilder;
import io.restassured.specification.RequestSpecification;
import org.testng.annotations.*;

import static io.restassured.RestAssured.given;
import static org.hamcrest.Matchers.equalTo;

public class UserApiTests {

    private RequestSpecification spec;

    @BeforeSuite
    public void globalSetup() {
        RestAssured.baseURI = "https://api.example.com";
        RestAssured.basePath = "/v1";
    }

    @BeforeClass
    public void buildSpec() {
        spec = new RequestSpecBuilder()
                .setContentType("application/json")
                .addHeader("Authorization", "Bearer " + System.getenv("TOKEN"))
                .build();
    }

    @DataProvider(name = "userIds")
    public Object[][] userIds() {
        return new Object[][] { {1}, {2}, {3} };
    }

    @Test(dataProvider = "userIds", priority = 1,
          groups = {"smoke"}, description = "GET /users/{id} returns 200")
    public void getUserById(int id) {
        given().spec(spec)
        .when().get("/users/{id}", id)
        .then().statusCode(200).body("id", equalTo(id));
    }
```

```
    @AfterMethod
    public void resetRA() {
        RestAssured.reset();
    }
}
```

JUnit 5 (Jupiter) equivalents

JUnit 5	Purpose
`@Test`	Marks a test (no attributes — use `@DisplayName`, `@Tag`, `@Disabled`).
`@BeforeAll` / `@AfterAll`	Once per class (must be `static` unless lifecycle is `PER_CLASS`).
`@BeforeEach` / `@AfterEach`	Per test method.
`@DisplayName("...")`	Human-readable name shown in reports.
`@Tag("smoke")`	Categorise tests (replaces TestNG groups).
`@Disabled("reason")`	Skip a test.
`@Nested`	Inner test class for behaviour grouping.
`@ExtendWith(Xxx.class)`	Register an extension (Allure, WireMock, Mockito).
`@TestMethodOrder(OrderAnnotation.class)` + `@Order(n)`	Deterministic ordering for end-to-end flows.
`@RepeatedTest(5)`	Run a test N times.
`@Timeout(10)`	Fail if it takes longer.

```
import org.junit.jupiter.api.*;
import static io.restassured.RestAssured.given;

@DisplayName("Users API — JUnit 5")
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class UserApiJupiterTests {

    @BeforeAll
    static void init() { io.restassured.RestAssured.baseURI = "https://api.example.com"; }

    @Test @Order(1) @Tag("smoke") @DisplayName("Create user")
    void createUser() {
        given().contentType("application/json")
               .body("""{ "name": "Ada" }""")
        .when().post("/users")
        .then().statusCode(201);
    }

    @RepeatedTest(3) @Tag("regression")
    void healthCheckIsStable() {
        given().when().get("/health").then().statusCode(200);
```

```
    }
}
```

JUnit 4 (legacy)

`@Test` , `@Before` , `@After` , `@BeforeClass` , `@AfterClass` , `@Ignore` , `@RunWith(Parameterized.class)` , `@Parameters` , `@Rule` , `@ClassRule` , `@FixMethodOrder(NAME_ASCENDING)` , `@Category(SmokeTests.class)` .

13.2 Parameterised Test Annotations (JUnit 5)

REST Assured suites are typically data-driven. The Jupiter `@ParameterizedTest` machinery is the modern replacement for TestNG `@DataProvider` .

Annotation	Purpose
`@ParameterizedTest`	Marks the method as parameterised (replaces `@Test`).
`@ValueSource(ints={1,2,3})`	Single primitive/String array of values.
`@CsvSource({"1, OK", "2, NOT_FOUND"})`	Inline CSV rows.
`@CsvFileSource(resources="/payloads.csv", numLinesToSkip=1)`	External CSV.
`@MethodSource("provider")`	Stream/Iterable provider method.
`@EnumSource(HttpStatus.class)`	All values of an enum.
`@ArgumentsSource(MyArgs.class)`	Custom `ArgumentsProvider` .
`@NullSource` / `@EmptySource` / `@NullAndEmptySource`	Edge values.

```
import org.junit.jupiter.params.*;
import org.junit.jupiter.params.provider.*;
import static io.restassured.RestAssured.when;
import static org.hamcrest.Matchers.notNullValue;

class StatusCodeMatrixTests {

    @ParameterizedTest(name = "GET /users/{0} -> {1}")
    @CsvSource({
        "1, 200",
        "999999, 404"
    })
    void getUserReturnsExpectedStatus(int id, int expected) {
        when().get("https://api.example.com/users/{id}", id)
        .then().statusCode(expected).body(notNullValue());
    }

    @ParameterizedTest
    @MethodSource("invalidPayloads")
```

```
    void rejectsBadPayload(String body) {
        io.restassured.RestAssured.given()
            .contentType("application/json").body(body)
        .when().post("https://api.example.com/users")
        .then().statusCode(400);
    }

    static java.util.stream.Stream<String> invalidPayloads() {
        return java.util.stream.Stream.of("{}", "{\"name\":\"\"}", "not json");
    }
}
```

13.3 Jackson Annotations on POJOs Sent / Received by REST Assured

REST Assured uses Jackson under the hood whenever you call `.body(pojo)` or `response.as(Pojo.class)`. Every annotation from sections 1-12 of this document applies, but these are the ones that show up most often in API test code:

Annotation	Why it matters in REST Assured tests
`@JsonProperty("snake_name")`	Map between Java camelCase fields and the JSON snake_case the API actually returns.
`@JsonInclude(Include.NON_NULL)`	Don't send `null` fields in PATCH/POST payloads — many APIs treat `null` as "clear field".
`@JsonIgnore` / `@JsonIgnoreProperties(ignoreUnknown = true)`	Survive new fields added by the backend without breaking deserialization.
`@JsonNaming(SnakeCaseStrategy.class)`	Convert an entire class at once instead of per-field.
`@JsonFormat(shape = STRING, pattern = "yyyy-MM-dd'T'HH:mm:ssXXX")`	Pin date/time format for assertions.
`@JsonCreator` + `@JsonProperty`	Deserialize immutable response DTOs (records or Lombok `@Value`).
`@JsonAlias({"id", "userId"})`	Accept either field name when the API is inconsistent across versions.
`@JsonRootName` + `WRAP_ROOT_VALUE`	When the API wraps responses in `{ "user": { ... } }`.
`@JsonTypeInfo` / `@JsonSubTypes`	Polymorphic responses (e.g. `Event` with subtypes `Login`, `Purchase`).

```
// Request DTO sent through .body(new CreateUser(...))
import tools.jackson.annotation.*;
import com.fasterxml.jackson.databind.PropertyNamingStrategies.SnakeCaseStrategy;
import tools.jackson.databind.annotation.JsonNaming;
```

```java
@JsonNaming(SnakeCaseStrategy.class)
@JsonInclude(JsonInclude.Include.NON_NULL)
public record CreateUser(
        String firstName,
        String lastName,
        @JsonProperty("email_address") String email,
        @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
        java.time.LocalDate birthDate
) {}

// Test
import static io.restassured.RestAssured.given;
class CreateUserTest {
    @org.junit.jupiter.api.Test
    void createsUser() {
        CreateUser dto = new CreateUser("Ada", "Lovelace",
                "ada@example.com", java.time.LocalDate.of(1815, 12, 10));

        UserResponse resp =
            given().contentType("application/json").body(dto)
            .when().post("https://api.example.com/users")
            .then().statusCode(201)
            .extract().as(UserResponse.class);  // Jackson deserialization
    }
}

// Response DTO — tolerant of unknown fields
@JsonIgnoreProperties(ignoreUnknown = true)
public record UserResponse(
        long id,
        @JsonAlias({"first_name", "givenName"}) String firstName,
        @JsonProperty("email_address") String email
) {}
```

13.4 Allure Reporting Annotations (the de-facto standard in REST Assured projects)

Add `io.qameta.allure:allure-rest-assured` + `allure-testng` / `allure-junit5`, then register `AllureRestAssured` as a `RestAssured.filters(...)` — these annotations decorate the generated report.

Annotation	Effect on the report
`@Epic("Billing")`	Top-level grouping.
`@Feature("Invoices")`	Sub-grouping under an Epic.
`@Story("Create invoice")`	Behavioural unit under a Feature.
`@Severity(SeverityLevel.CRITICAL)`	Coloured severity badge.
`@Description("...")`	Long form description.
`@Step("Send POST /users with {0}")`	Marks a helper method as a report step (with parameter substitution).

Annotation	Effect on the report
`@Issue("JIRA-123")` / `@TmsLink("TC-42")`	Hyperlink to JIRA / TestRail.
`@Owner("qa-team")`	Test ownership.
`@Attachment(value = "payload", type = "application/json")`	Method return value is attached to the report.
`@Flaky` / `@Muted`	Mark known-unstable tests.

```java
import io.qameta.allure.*;
import org.junit.jupiter.api.Test;
import static io.restassured.RestAssured.given;

@Epic("Users") @Feature("CRUD")
class UserCrudAllureTest {

    @Test @Story("Fetch user") @Severity(SeverityLevel.CRITICAL)
    @Issue("JIRA-512") @Owner("backend-qa")
    @Description("Verifies GET /users/{id} returns the expected payload")
    void fetchUser() {
        getUser(1);
    }

    @Step("GET /users/{0}")
    void getUser(int id) {
        given().when().get("https://api.example.com/users/{id}", id)
               .then().statusCode(200);
    }

    @Attachment(value = "Response body", type = "application/json")
    byte[] attach(String body) { return body.getBytes(); }
}
```

13.5 Lombok Annotations Commonly Combined with REST Assured POJOs

Lombok is not strictly required, but it removes a huge amount of DTO boilerplate. Jackson reads the generated getters/setters and constructors transparently.

Annotation	Generates
`@Data`	Getters, setters, `equals`, `hashCode`, `toString`.
`@Value`	Immutable equivalent of `@Data` (all fields `private final`).
`@Builder` / `@Jacksonized @Builder`	Fluent builder; `@Jacksonized` makes Jackson use it for deserialization.
`@NoArgsConstructor` / `@AllArgsConstructor` / `@RequiredArgsConstructor`	Constructors. Jackson needs a no-args constructor unless you use `@JsonCreator`.

Annotation	Generates
`@Getter` / `@Setter`	Per-field accessor generation.
`@ToString(exclude = "password")`	Safe logging in test output.
`@EqualsAndHashCode`	For comparing expected vs actual DTOs in assertions.
`@SneakyThrows`	Avoid `throws` clutter in test helpers.
`@Slf4j`	Inject a logger into test utility classes.

```
import lombok.*;
import lombok.extern.jackson.Jacksonized;

@Value
@Builder
@Jacksonized          // tells Jackson to deserialize via the builder
public class Invoice {
    long id;
    String customer;
    java.math.BigDecimal amount;
}

// Usage in a REST Assured test
Invoice expected = Invoice.builder().id(1).customer("Ada").amount(new
            java.math.BigDecimal("9.99")).build();
Invoice actual   = io.restassured.RestAssured.get("/invoices/1").as(Invoice.class);
org.assertj.core.api.Assertions.assertThat(actual).isEqualTo(expected);
```

13.6 Bean Validation Annotations on Request DTOs

Useful when you want to sanity-check that the payload you are about to send is well-formed before hitting the API (e.g. with `Validator`/Hibernate Validator inside a `@BeforeEach`).

`@NotNull`, `@NotBlank`, `@NotEmpty`, `@Size(min=, max=)`, `@Min`, `@Max`, `@Email`, `@Pattern(regexp=...)`, `@Positive`, `@PastOrPresent`, `@Valid`, `@AssertTrue`.

```
import jakarta.validation.constraints.*;
public record CreateUser(
        @NotBlank @Size(max = 50) String firstName,
        @NotBlank @Email          String email,
        @Min(0) @Max(120)          int    age) {}
```

13.7 Spring-based REST Assured Tests (when the SUT is a Spring Boot app)

When you run REST Assured against an in-process Spring Boot app via `RestAssuredMockMvc` or `port = RANDOM_PORT`, the following Spring annotations appear:

Annotation	Purpose
`@SpringBootTest(webEnvironment = RANDOM_PORT)`	Boots the whole app on a random port.
`@LocalServerPort`	Inject that port into `RestAssured.port`.
`@AutoConfigureMockMvc`	Enables `RestAssuredMockMvc`.
`@ActiveProfiles("test")`	Choose a test profile.
`@TestConfiguration`	Provide test-only beans (e.g. WireMock).
`@MockBean` / `@SpyBean`	Replace a bean with a Mockito mock/spy.
`@Sql("/seed.sql")`	Seed the DB before a test.
`@DynamicPropertySource`	Inject Testcontainers URLs at runtime.
`@Transactional`	Roll the DB back after each test.

```
import io.restassured.RestAssured;
import org.junit.jupiter.api.*;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;

import static org.springframework.boot.test.context.SpringBootTest.WebEnvironment.RANDOM_PORT;

@SpringBootTest(webEnvironment = RANDOM_PORT)
class UserApiIT {

   @LocalServerPort int port;

   @BeforeEach void setUp() { RestAssured.port = port; }

   @Test void contextLoadsAndUsersAreReachable() {
      RestAssured.given().when().get("/users").then().statusCode(200);
   }
}
```

13.8 Quick Cheat-Sheet

Concern	TestNG	JUnit 5
Mark a test	`@Test`	`@Test`
Setup once / class	`@BeforeClass`	`@BeforeAll` (static)
Setup per test	`@BeforeMethod`	`@BeforeEach`
Skip	`@Test(enabled=false)`	`@Disabled`
Group / categorise	`groups = {"smoke"}`	`@Tag("smoke")`
Order	`priority = n`	`@Order(n)` + `@TestMethodOrder`
Data-driven	`@DataProvider`	`@ParameterizedTest` + `@CsvSource` etc.

Concern	TestNG	JUnit 5
Expect exception	`expectedExceptions = ...`	`assertThrows(...)` (no annotation)
Listeners / extensions	`@Listeners(...)`	`@ExtendWith(...)`
Friendly name	`description = "..."`	`@DisplayName("...")`

> **Rule of thumb for a REST Assured project**: *Test framework annotations* (TestNG / JUnit) decide **when** code runs, *Jackson annotations* (this document) decide **what JSON looks like on the wire**, and *Allure / Lombok / Validation annotations* decide **how the test is reported and how DTOs are written**.