**Complex JSON to POJO Example**

**🌱 1️⃣ Purpose of the Example**

This example shows **how to read a complex JSON file** (having nested objects and arrays) and **convert it into Java objects (POJOs)** using **Jackson API**.

**🧰 2️⃣ What Is a Complex JSON?**

A *complex JSON* means it contains:

* **Simple properties** → like id, firstName, active
* **HAS-A relationship** → like an address object inside customer
* **Arrays/Lists** → like languages: ["Java","Python"]

Example JSON:

{

"id": 10,

"firstName": "Sachin",

"lastName": "Tendulkar",

"active": true,

"address": {

"street": "25/1",

"city": "Mumbai",

"state": "Maharashtra",

"zip": "560026",

"country": "IND"

},

"languages": ["Java", "C#", "Python", "JavaScript"]

}

**🧱 3️⃣ Java Class Design**

We create **two POJO classes** to represent this JSON:

* Customer → main object
* Address → sub-object (HAS-A relation)

Customer contains:

* simple fields (id, firstName, lastName, etc.)
* one Address type field (address)
* one List<String> field (languages)

So we are representing both **object nesting** and **arrays** in Java.

**⚙️ 4️⃣ Tools and Dependencies Used**

We use two main libraries:

1. **Jackson Databind**
   * To read and write JSON data
   * Provides ObjectMapper class
2. **Lombok**
   * To reduce boilerplate code
   * Automatically adds getters, setters, toString, etc. using @Data

👉 Without Lombok, you’d manually write 20–30 lines of getters/setters.

**🧠 5️⃣ Internal Working**

When you call:

Customer customer = mapper.readValue(new File("data/sample.json"), Customer.class);

Here’s what happens internally:

1. Jackson reads the JSON file.
2. For every key (like "id": 10), it looks for a matching setter method in the POJO (like setId(10)).
3. It creates an object of Customer, fills the data, and also creates the nested Address object automatically.
4. The whole Java object is ready for use.

**🔁 6️⃣ Important Concept**

* ObjectMapper → main class that performs conversion
  + readValue() → JSON → Java object
  + writeValue() → Java object → JSON

**🧩 7️⃣ HAS-A Relationship**

In the JSON, address is a sub-object:

"address": {

"street": "25/1",

"city": "Mumbai",

...

}

This is represented in Java as:

private Address address;

This means **Customer HAS-A Address** — a perfect example of *object composition*.

**💡 8️⃣ List or Array Conversion**

The "languages" array in JSON is automatically mapped to a List<String> in Java:

private List<String> languages;

Jackson handles this conversion internally without needing loops.

**📦 9️⃣ Output**

When you print the customer object:

System.out.println(customer);

Lombok’s @Data generates a readable toString() method automatically, showing full details including nested address and list elements.

Example output:

Customer(id=10, firstName=Sachin, lastName=Tendulkar, active=true,

address=Address(street=25/1, city=Mumbai, state=Maharashtra, zip=560026, country=IND),

languages=[Java, C#, Python, JavaScript])

**📝 10️⃣ Summary Notes**

| **Concept** | **Explanation** |
| --- | --- |
| **ObjectMapper** | Used to convert JSON ↔ Java Object |
| **@Data (Lombok)** | Automatically adds getters/setters/toString |
| **HAS-A Property** | Nested object (Address inside Customer) |
| **Array in JSON** | Converted to List in Java |
| **Mapping Rule** | JSON key → Java setter (e.g., id → setId()) |

Eg: ComplexJsonObject

**@JsonIgnoreProperties — Detailed Notes**

**🌟 1️⃣ What It Is**

@JsonIgnoreProperties is an annotation provided by the **Jackson library**.

It tells Jackson:

“Ignore some fields from JSON while converting it to a Java object.”

**🌟 2️⃣ Why We Need It**

When JSON contains **extra properties** that your Java class doesn’t have,  
then while converting JSON → POJO, Jackson will throw an error like 👇

UnrecognizedPropertyException: Unrecognized field "age"

This happens because Jackson by default expects **every JSON key** to match a **Java variable name**.

But in **real-world APIs**, sometimes the JSON data we receive:

* Has extra fields we don’t need
* Comes from other teams or systems with more data
* Changes frequently

Instead of changing your class every time,  
you simply tell Jackson:

“Ignore any unknown fields — don’t crash!”

That’s what @JsonIgnoreProperties(ignoreUnknown = true) does.

**🌟 3️⃣ Basic Syntax**

@JsonIgnoreProperties(ignoreUnknown = true)

public class Customer {

private Integer id;

private String name;

}

📌 Now if the JSON looks like this 👇

{

"id": 101,

"name": "Sachin",

"country": "India"

}

➡️ Even though country doesn’t exist in the Customer class,  
Jackson will **ignore it silently** and still convert JSON successfully.

**🌟 4️⃣ What Happens Internally**

When ObjectMapper tries to map JSON to your class:

* It loops through each key in the JSON.
* If it finds a matching field → it calls the setter (like setName("Sachin")).
* If the key **doesn’t exist** in your POJO:
  + Normally → it throws an UnrecognizedPropertyException.
  + But with @JsonIgnoreProperties(ignoreUnknown = true) →  
    it just skips that field and continues mapping the rest.

So it prevents your app from **breaking** when the JSON is **not an exact match**.

**🌟 5️⃣ You Can Also Ignore Specific Fields**

Instead of ignoring all unknown fields,  
you can ignore **specific known fields** using the same annotation.

@JsonIgnoreProperties({"salary", "country"})

public class Employee {

private int id;

private String name;

private double salary;

private String country;

}

Now, when you convert JSON → POJO,  
Jackson will completely **skip reading or writing** those fields.

Even if salary or country exist in JSON or POJO —  
they will be **ignored during serialization and deserialization.**

**🌟 6️⃣ Types of Use**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| ignoreUnknown = true | Ignores all unknown fields from JSON | @JsonIgnoreProperties(ignoreUnknown = true) |
| {"field1", "field2"} | Ignores specific known fields | @JsonIgnoreProperties({"salary", "age"}) |

**🌟 7️⃣ Real-Time Use Case (Very Common in APIs)**

Imagine your backend expects JSON like this:

{

"id": 101,

"name": "Sachin"

}

but your frontend team sends this:

{

"id": 101,

"name": "Sachin",

"country": "India",

"team": "MI"

}

Without @JsonIgnoreProperties(ignoreUnknown = true),  
your backend will crash ❌.

With it → your backend runs smoothly ✅  
and still maps only what it needs.

**🌟 8️⃣ Reverse Direction (Java → JSON)**

Even when converting from **Java → JSON**,  
fields marked in @JsonIgnoreProperties will not be included in the generated JSON.

Example:

@JsonIgnoreProperties({"salary"})

public class Employee {

private int id;

private String name;

private double salary;

}

When converted to JSON, output will be:

{

"id": 101,

"name": "Sachin"

}

salary is skipped while generating JSON.

**🌟 9️⃣ Common Mistake (and Fix)**

❌ **Mistake:**  
Adding @JsonIgnoreProperties but not enabling ignoreUnknown.

@JsonIgnoreProperties

public class Customer { ... }

This alone does nothing — you must use:

@JsonIgnoreProperties(ignoreUnknown = true)

✅ This tells Jackson to skip unknown properties.

**🌟 🔟 Summary Table**

| **Concept** | **Meaning** | **Example** |
| --- | --- | --- |
| **Unknown JSON Fields** | Extra keys in JSON not in POJO | "country": "India" |
| **Without Annotation** | Jackson throws UnrecognizedPropertyException | ❌ |
| **With @JsonIgnoreProperties(ignoreUnknown=true)** | Jackson skips extra fields | ✅ |
| **With @JsonIgnoreProperties({"x","y"})** | Jackson ignores specific fields | ✅ |
| **Used in** | POJO classes that map to JSON | Common in REST APIs |

**💡 Analogy**

Think of @JsonIgnoreProperties like a **filter** when reading JSON:

🧩 JSON = a parcel box with many items  
📦 Your POJO = can only hold specific items  
🪶 @JsonIgnoreProperties(ignoreUnknown=true) = tells the delivery guy:

“If there are any extra items that don’t fit, just leave them out.”

Eg: JsonIgnoreProperties

**Correct Understanding (Simplified)**

Yes —  
👉 **@PathVariable** and **@RequestBody** are the main triggers that **automate** the data binding (conversion) process between **HTTP requests** and **Java objects**.  
But — they handle *different kinds of automation*.

Let’s break this carefully 👇

**🌿 1️⃣ @PathVariable → Path Data Automation**

**🔹 Purpose:**

Used when **data is part of the URL path**.

Example URL:

/api/customer/id/10

**🔹 Code:**

@GetMapping("/id/{theId}")

public Customer getCustomerById(@PathVariable("theId") Integer id) {

// "10" automatically converted to Integer and injected

}

**🔹 What Happens Internally:**

1. Client sends URL /id/10.
2. Spring reads {theId} = 10.
3. It automatically **converts the value** (String "10") → (Integer 10)  
   ✅ No need for parsing manually.
4. Spring **injects** that into your method parameter id.

🧠 **So @PathVariable automates:**  
➡ Extracting and converting *path values* into Java variables.

**🌾 2️⃣ @RequestBody → JSON Body Automation**

**🔹 Purpose:**

Used when **data is sent in the request body** (usually in POST or PUT requests).

**🔹 Code:**

@PostMapping("/save")

public String saveCustomer(@RequestBody Customer customer) {

// Incoming JSON automatically converted into Customer object

}

**🔹 What Happens Internally:**

1. Client sends JSON:
2. {
3. "id": 101,
4. "name": "Virat",
5. "country": "India"
6. }
7. Spring detects @RequestBody.
8. It uses **HttpMessageConverter** → internally calls **Jackson library**.
9. Jackson automatically converts JSON → Customer (Java object).
10. Injects the created object into your method parameter.

🧠 **So @RequestBody automates:**  
➡ Reading JSON → creating & filling Java objects automatically.

**🌻 3️⃣ Output Automation (Bonus)**

When you **return** a Java object from a @RestController method,  
Spring automatically converts that object → JSON before sending it as a response.

Example:

@GetMapping("/get")

public Customer getCustomer() {

Customer c = new Customer();

c.setId(1);

c.setName("Sachin");

return c; // <-- Auto converted to JSON

}

✅ JSON Response:

{

"id": 1,

"name": "Sachin"

}

🧠 No need to manually use ObjectMapper.writeValueAsString() — Spring + Jackson handle it.

**✨ So in short:**

| **Annotation** | **Automates What** | **Direction** | **Example** |
| --- | --- | --- | --- |
| @PathVariable | Extracts & converts **path parameters** | URL → Java | /id/{theId} |
| @RequestBody | Converts **JSON body → Java object** | JSON → Java | POST JSON |
| @RestController (output) | Converts **Java → JSON** | Java → JSON | API response |

**🧩 Combined Together:**

These 3 work like a team:

| **Step** | **Action** | **Done by** |
| --- | --- | --- |
| 1️⃣ | Client sends request (with path + JSON body) | Browser/Postman |
| 2️⃣ | Spring reads the path | @PathVariable |
| 3️⃣ | Spring reads the JSON body | @RequestBody |
| 4️⃣ | Converts both automatically | Jackson + HttpMessageConverter |
| 5️⃣ | You process logic | Your Controller/Service |
| 6️⃣ | You return object | @RestController |
| 7️⃣ | Spring converts object → JSON | Jackson again |
| 8️⃣ | Sends back to client | HTTP Response |

**🧠 Quick Memory Tip:**

🗺️ @PathVariable → data in the URL path  
📦 @RequestBody → data in the request body  
🪄 Together → no need for manual JSON handling ever again!

**🧩 Notes on @RestController and Java → JSON Conversion**

**🔹 1️⃣ What is @RestController**

* It is a **specialized version** of @Controller used in RESTful web services.
* It is a combination of two annotations:
* @Controller + @ResponseBody
* It tells Spring:

“Don’t look for a view page (like JSP or HTML) — instead, send the return value **directly as the HTTP response body**.”

**🔹 2️⃣ How JSON Conversion Happens Automatically**

* When a method inside @RestController returns a **Java Object (POJO)**,  
  Spring automatically converts that object into **JSON format** before sending it to the client.
* This conversion is done using an internal library called **Jackson**.

🧠 Internally uses:

MappingJackson2HttpMessageConverter

which calls:

ObjectMapper.writeValueAsString(object)

**🔹 3️⃣ When You Hit the URL**

1. You call the endpoint (e.g., /api/customer/get)
2. Spring finds the controller method that matches
3. The method returns a **Java object (Customer)**
4. Spring’s DispatcherServlet sends it to the **Jackson converter**
5. Jackson converts that Java object → JSON text
6. JSON is sent as the **HTTP response**

**🔹 4️⃣ Why @ResponseBody Is Not Needed**

* Normally in @Controller, you would need to write:
* @ResponseBody
* public Customer getCustomer() { ... }
* But since @RestController already includes @ResponseBody,  
  you don’t have to write it separately.

**🔹 5️⃣ What Can Be Returned**

With @RestController, you can return:

* A **single POJO**
* A **List of objects**
* A **Map**
* A **String**
* A **ResponseEntity<>** (for status + body)

All of them are automatically converted to **JSON** by Spring.

**🔹 6️⃣ Common Use Cases**

✅ Sending object data to frontend (React, Angular, Vue)  
✅ Exposing APIs for mobile apps  
✅ Returning structured data responses  
✅ Building microservices or REST APIs

**🔹 7️⃣ Example Output Explanation**

If method returns:

new Customer(101, "Sachin Tendulkar", "sachin@gmail.com");

Then JSON output will be:

{

"id": 101,

"name": "Sachin Tendulkar",

"email": "sachin@gmail.com"

}

Here:

* Each **Java field** → becomes a **JSON key**
* Each **Java value** → becomes a **JSON value**

**💬 Analogy**

Think of @RestController as an **automatic translator**:

You give it a Java object,  
it automatically “speaks JSON” to the outside world —  
so clients (like Postman, React, or any browser) can understand it easily.

Eg: JavaObjectToJson

Earlier, we saw:  
➡️ **Java → JSON** using @RestController (Spring converts your object into JSON).

Now we’ll see the reverse:  
➡️ **JSON → Java** using @RequestBody (Spring converts incoming JSON into your object automatically).

# 📝 Notes: Using @RequestBody (JSON → Java Object)

### 1️⃣ What is @RequestBody?

* @RequestBody is an annotation in Spring MVC.
* It **tells Spring to read JSON data from the HTTP request body** and **convert it into a Java object** automatically.
* Works with any HTTP client: Postman, browser fetch, Angular, React, etc.

### 2️⃣ How it Works (Step by Step)

1. **Client sends JSON** → in the HTTP request body.  
   Example:
2. {
3. "id": 101,
4. "name": "Sachin Tendulkar",
5. "email": "sachin@gmail.com"
6. }
7. @RequestBody tells Spring to **map the JSON to your Java POJO**.
8. **Jackson library** does the conversion automatically.
   * It looks for **matching JSON keys and Java field names**.
   * Uses **setters** to fill the object.
9. Your controller method receives the **ready-to-use Java object**.
10. You can **process, save, or respond** as normal in Java.

### 3️⃣ POJO Requirements

* Must have **default constructor** → Jackson uses it to create object.
* Must have **getters and setters** → Jackson sets values via setters.
* Field names in Java **must match JSON keys** (case-sensitive).
* Data types are handled automatically (int, String, boolean, etc.).

### 4️⃣ Example Flow

| **Step** | **Description** | **Who Handles It** |
| --- | --- | --- |
| 1️⃣ | Send JSON in request body | Client (Postman, Angular, etc.) |
| 2️⃣ | Convert JSON → Java Object | Spring MVC (@RequestBody) |
| 3️⃣ | Map JSON keys to POJO fields | Jackson Library |
| 4️⃣ | Controller receives Customer object | Spring Framework |
| 5️⃣ | Process object like normal Java | Developer |

### 5️⃣ Key Notes

* @RequestBody = JSON → Java object converter.
* **Jackson** = library that maps JSON to POJO.
* **Default constructor** is mandatory.
* **Field names must match JSON keys.**
* Works for **complex nested objects** too.

### 6️⃣ Analogy

Think of @RequestBody like a **“magic converter machine”**:

* You throw JSON in the request.
* Spring + Jackson **magically creates a Java object** ready for you to use.

### 7️⃣ Quick Controller Example

@PostMapping("/save")

public ResponseEntity<String> saveCustomer(@RequestBody Customer customer) {

System.out.println("Received Customer: " + customer);

return new ResponseEntity<>("Customer " + customer.getName() + " saved successfully!", HttpStatus.CREATED);

}

✅ **Takeaway**:  
Whenever you need **client → server communication with JSON**, @RequestBody is the easiest and cleanest way to convert it into a Java object without manually parsing.

Eg: JsonToPojoObject