



# Ultimate Deep-Dive Guide to Servlets, External Tomcat, Embedded Tomcat & Full TODO REST API

This note is **rebuilt from scratch** to be extremely detailed and easier to understand.

It includes:

- ✓ What is a **Servlet**
- ✓ How a **Servlet Container** works internally
- ✓ What is **Apache Tomcat** (External Server)
- ✓ How to install + configure **External Tomcat**
- ✓ What is **web.xml** and why it exists
- ✓ How to create a **Servlet-based TODO REST API**
- ✓ What is an **Embedded Server** (Embedded Tomcat)
- ✓ How Spring Boot embeds Tomcat internally

---

## 1. What is a Servlet? (Super Beginner-to-Advanced Explanation)

A **Servlet** is a Java class that:

- Runs inside a **Servlet Container** like Tomcat
- Handles **HTTP requests** (GET, POST, PUT, DELETE)
- Sends back **HTTP responses**

**A Servlet = A Java program that runs on a server and communicates using HTTP.**

Example:

```
Client → HTTP request → Servlet → Response
```



## 2. How a Servlet Container Works Internally

A Servlet cannot run alone.

It needs a **Servlet Container**, which manages:

### Lifecycle of Servlet

- Loading class
- Creating object

- Calling `init()` once
- Calling `service()` for every request
- Calling `destroy()` when shutting down

## Multi-threading

Each incoming request = **new thread**. The Servlet object is shared, but each request gets its own thread.

## HTTP Parsing

Container handles:

- Parsing HTTP headers
- Reading request body
- Generating response



## 3. Servlet Lifecycle (Very Detailed)

```

START
↓
Servlet Loaded into memory
↓
init() → called once
↓
service() → called for every request
  ↴ doGet(), doPost(), doPut(), doDelete()
↓
destroy() → called once when server stops

```



## 4. What is Apache Tomcat (External Server)?

Apache Tomcat is:

- A **Web Server**
- A **Servlet Container**
- A **JSP engine**

### What Tomcat Does Internally

Task	Description
Start HTTP server	Opens port 8080
Manage servlets	Creates, loads, maps, destroys

Task	Description
Multi-threading	One thread per request
Deployment	Accepts WAR files
Classloading	Isolates each application

## 5. Installing & Configuring Apache Tomcat (Step-by-Step)

### Step 1 – Download

From: <https://tomcat.apache.org/>

### Step 2 – Extract ZIP

### Step 3 – Configure in IDE

#### IntelliJ

```
File → Settings → Build Tools → Application Servers → Add Tomcat
```

#### Eclipse

```
Servers tab → Add new server → Apache Tomcat
```

### Step 4 – Create Dynamic Web Project

This generates:

```
WEB-INF/web.xml
```

## 6. Folder Structure for Servlet Project (WAR)

```
myapp/
├── src/main/java/
│   └── com.example
│       └── TodoServlet.java
└── src/main/webapp/
```

```
    |   └── WEB-INF/  
    |       └── web.xml
```



## 7. What is web.xml? Why do we need it?

web.xml is the **deployment descriptor**. It tells Tomcat:

- Which servlet exists
- Which URL maps to that servlet

Example:

```
<web-app>  
    <servlet>  
        <servlet-name>TodoServlet</servlet-name>  
        <servlet-class>com.example.TodoServlet</servlet-class>  
    </servlet>  
  
    <servlet-mapping>  
        <servlet-name>TodoServlet</servlet-name>  
        <url-pattern>/todo</url-pattern>  
    </servlet-mapping>  
</web-app>
```



## 8. Creating a TODO REST API using Servlets (External Tomcat)

We will implement:

- GET → fetch all todos
- POST → add a new todo

### Step 1: Model Class

```
public class Todo {  
    private int id;  
    private String task;  
  
    public Todo(int id, String task) {  
        this.id = id;  
        this.task = task;  
    }
```

```
// getters + setters  
}
```

## Step 2: TodoServlet

```
@WebServlet("/todo")  
public class TodoServlet extends HttpServlet {  
  
    private List<Todo> todos = new ArrayList<>();  
    private int counter = 1;  
  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
throws IOException {  
    resp.setContentType("application/json");  
  
    StringBuilder json = new StringBuilder("[");  
    for (int i = 0; i < todos.size(); i++) {  
        Todo t = todos.get(i);  
        json.append(String.format("{\"id\":%d, \"task\":\"%s\"}",  
t.getId(), t.getTask()));  
        if (i < todos.size() - 1) json.append(",");  
    }  
    json.append("]");  
  
    resp.getWriter().write(json.toString());  
}  
  
    @Override  
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)  
throws IOException {  
    String task = req.getParameter("task");  
    Todo todo = new Todo(counter++, task);  
    todos.add(todo);  
    resp.getWriter().write("Todo added successfully");  
}  
}
```



## 9. Testing the Servlet API

### Add Todo (POST)

```
http://localhost:8080/myapp/todo?task=Learn Servlet
```

## Fetch All Todos (GET)

```
http://localhost:8080/myapp/todo
```

# 10. What is an Embedded Server? (Very Detailed)

An **embedded server** is a web server (like Tomcat, Jetty, Undertow) that runs *inside your Java application* instead of being installed separately.

There are **two types of Embedded Tomcat usage**:

- 1 Manual Embedded Tomcat (YOU configure everything)
- 2 Spring Boot Embedded Tomcat (Spring configures everything)

Let's understand both.

## A. Manual Embedded Tomcat (YOU handle servlet mapping)

If you embed Tomcat manually, Tomcat knows NOTHING about:

- `@WebServlet`
- `@GetMapping`
- Spring Boot
- Annotation scanning

So you must manually register every servlet + URL mapping.

### Example of MANUAL embedded Tomcat

```
Tomcat tomcat = new Tomcat();
tomcat.setPort(8080);

// Create context (root context)
Context context = tomcat.addContext("", null);

// Register servlet manually
Tomcat.addServlet(context, "HelloServlet", new HelloServlet());

// Map URL → servlet
context.addServletMappingDecoded("/hello", "HelloServlet");
```

```
// Start server
tomcat.start();
tomcat.getServer().await();
```

## Key Points

Feature	Manual Embedded Tomcat
Auto servlet discovery	✗ No
Auto URL mapping	✗ No
You must call <code>addServlet()</code>	✓ Yes
You must call <code>addServletMappingDecoded()</code>	✓ Yes

You must **configure everything by hand** because Tomcat is just a server — it does NOT understand Spring annotations.

## B. Spring Boot Embedded Tomcat (NO manual mapping required)

Spring Boot handles everything automatically:

### What Spring Boot does:

- ✓ Starts embedded Tomcat
- ✓ Creates `DispatcherServlet`
- ✓ Scans `@RestController` and `@RequestMapping`
- ✓ Registers controller methods as HTTP endpoints
- ✓ Does NOT require you to register servlets manually

### You NEVER write:

```
context.addServletMappingDecoded()
```

### Because Spring Boot internally registers:

```
/* ➔ DispatcherServlet
```

All requests go to **one servlet**, and Spring takes care of routing.



# Why manual mapping is not needed in Spring Boot?

Because Spring Boot uses:

- **DispatcherServlet** (master servlet)
- **HandlerMapping** (maps URLs to controller methods)
- **HandlerAdapter** (executes method)

So your controller becomes part of Spring MVC, NOT a raw servlet.

---



## Final Comparison: Manual vs Spring Boot Embedded Tomcat

Feature	Manual Embedded Tomcat	Spring Boot Embedded Tomcat
Write servlet class	✓ Yes	✗ No
Register servlet	✓ Manual	✗ Auto
Map URL	✓ Manual	✗ Auto (via @GetMapping)
Central DispatcherServlet	✗ No	✓ Yes
Annotation support	✗ No	✓ Full
Effort required	High	Very Low

---



## Key Takeaway

**Manual Embedded Tomcat → You must manually configure servlets and URL mapping**

**Spring Boot Embedded Tomcat → Spring does everything for you via DispatcherServlet**

---



## 10. What is an Embedded Server? (Very Detailed)

(continued below — original section preserved) An **embedded server** is a server (like Tomcat) that runs *inside* your Java application.

Spring Boot contains **Embedded Tomcat** via dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
``` (Very Detailed)
An **embedded server** is a server (like Tomcat) that runs *inside* your Java application.
```

Spring Boot contains **Embedded Tomcat** via dependency:

```
```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

## Embedded Server Advantages

Feature	Embedded Tomcat
No installation	✓ Built-in
Run as JAR	✓ <code>java -jar app.jar</code>
Easy microservices	✓
No WAR file	✓

## How Spring Boot Converts Controllers → Servlets

# How Spring Boot Converts Controllers Into Servlets (Deep Internal Explanation)

Spring Boot does **NOT** directly turn your controller class into a Servlet.

Instead, Spring Boot uses the **Spring MVC framework**, which internally uses one master servlet:

## 1. DispatcherServlet → The Heart of Spring MVC

When you add:

```
spring-boot-starter-web
```

Spring Boot automatically registers a servlet called:

```
DispatcherServlet
```

This is the **main servlet** that handles *all incoming HTTP requests*.

This means:

**Your controllers are NOT servlets → DispatcherServlet is the only servlet.**

## 2. Request Flow Inside Spring Boot (Internal Pipeline)

```
Browser
  ↓
Embedded Tomcat (HTTP Server)
  ↓
DispatcherServlet (Single Servlet)
  ↓
HandlerMapping (Finds which controller/method)
  ↓
Controller Method (@GetMapping/@PostMapping)
  ↓
Returns Java Object (Model / JSON)
  ↓
HttpMessageConverter (JSON/XML converters)
  ↓
Response sent back
```

## 3. Step-by-Step: How a Request Reaches Your Controller

### Step 1: Tomcat Receives HTTP Request

Embedded Tomcat listens on port **8080** and receives requests.

### Step 2: Tomcat Maps All URLs to DispatcherServlet

Spring Boot autoconfigures this mapping:

```
/* → DispatcherServlet
```

Meaning **all requests** go to `DispatcherServlet`.

### Step 3: DispatcherServlet Uses HandlerMapping

It checks:

- What URL did the user call?
- Which controller handles this URL?

Example:

```
@GetMapping("/hello")
public String hello() { ... }
```

Spring registers it internally as a **handler method**.

### Step 4: DispatcherServlet Calls the Controller Method

Once the correct controller + method is found, it executes it.

### Step 5: Return Value is Converted Into JSON (or HTML)

Using **HttpMessageConverters** such as:

- `MappingJackson2Http`