🚀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 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>
```

Embedded Server Advantages

Feature	Embedded Tomcat
No installation	✔️Built-in
Run as JAR	✔️ `java -jar app.jar`
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:

- MappingJackson2HttpMessageConverter (for JSON)
- StringHttpMessageConverter

Step 6: Response Sent → Browser

DispatcherServlet sends the final HTTP response.

🎠4. Why Controllers in Spring Boot Are NOT Servlets

Servlet	Spring MVC Controller
Extends HttpServlet	Plain class with annotations
doGet(), doPost()	@GetMapping, @PostMapping
Manual JSON creation	Automatic JSON conversion
One servlet per URL	One servlet (DispatcherServlet) maps all URLs

Spring Boot simplifies everything by letting **DispatcherServlet** do the servlet work.

🎠5. Complete Internal Architecture Diagram

```
HTTP Request
     ↓
Embedded Tomcat
     ↓
DispatcherServlet (Main Servlet)
     ↓
HandlerMapping (Find controller)
     ↓
HandlerAdapter (Prepare method call)
     ↓
Your Controller Method
     ↓
HttpMessageConverter
     ↓
HTTP Response
```

🎠6. Summary (Extremely Important)

- Spring Boot does **NOT** convert each controller into a servlet.
- Instead, it uses **ONE servlet** → DispatcherServlet.
- DispatcherServlet internally forwards the request to the correct controller.
- Controllers are plain Java classes with annotations.
- HttpMessageConverters handle JSON/XML output.

Spring Boot abstracts the entire servlet layer while still running **on top of servlets**.

```
@RestController
 ↓
HandlerMapping
 ↓
DispatcherServlet
 ↓
Embedded Tomcat
 ↓
HTTP Server
```

You don't write `web.xml`.
Spring Boot uses **auto-configuration**.

🏠11. Building TODO REST API using Embedded Tomcat (Spring Boot)

Step 1 → Create Spring Boot Project

Add dependency:

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

Step 2 → Model

```
public class Todo {
    private int id;
    private String task;
}
```

Step 3 → Controller

```
@RestController
@RequestMapping("/todo")
public class TodoController {

    private List<Todo> list = new ArrayList<>();
    private int counter = 1;

    @GetMapping
    public List<Todo> getAll() {
        return list;
    }

    @PostMapping
    public String add(@RequestParam String task) {
        list.add(new Todo(counter++, task));
        return "Todo added";
    }
}
```

🛖12. External Tomcat vs Embedded Tomcat (Final Comparison)

Feature	External Tomcat	Embedded Tomcat
Deployment	WAR	JAR
Needs Installation	Yes	No
Configuration	web.xml	annotations
Startup	Manual	Automatic
Used by	Traditional Java EE	Spring Boot, Microservices

🚢Final Summary

- Servlets are Java classes that handle HTTP
- External Tomcat must be installed manually
- web.xml maps URLs to Servlets
- Embedded Tomcat comes inside Spring Boot
- You can build REST APIs using both approaches

If you want, I can add diagrams, Servlet Filters, JSP MVC flow, Sessions/Cookies, Thread Model, or convert this into a PDF.