Advanced Spring MVC

HandlerInterceptor (preHandle/postHandle/afterCompletion) and the Filter vs. Interceptor distinction; registering interceptors and configuring global CORS with WebMvcConfigurer; same-origin policy, the preflight request, and @CrossOrigin; file uploads with MultipartFile and size limits.

Advanced 45 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/advanced-spring-mvc

---

Advanced Spring MVC

In Spring MVC Fundamentals' "The Journey of an HTTP Request: Request Lifecycle" section, we saw how a request reaches DispatcherServlet, and from there gets routed to a controller method via `HandlerMapping` / `HandlerAdapter`. On top of that journey, there are two ways to add **cross-cutting** behavior to every request: `Filter` (at the Servlet API level, outside DispatcherServlet) and `HandlerInterceptor` (at the Spring MVC level, inside DispatcherServlet). This lesson covers both, the `WebMvcConfigurer` both of them are configured through, how browsers restrict cross-origin requests (CORS), and how file uploads (`multipart/form-data`) fit into this pipeline.

What Is HandlerInterceptor?

`HandlerInterceptor` is a Spring MVC interface that lets you write code that runs **before** a controller method is called, **after** it's called, and once the response is fully done -- for behavior like logging, authentication, or timing that repeats across many endpoints but doesn't belong to any single endpoint's business logic:

```
interface MinimalInterceptor {
    boolean preHandle(Object request, Object response, Object handler);
    void afterCompletion(Object request, Object response, Object handler, Exception ex);
}
```

The real interface uses `jakarta.servlet.http.HttpServletRequest` / `HttpServletResponse` and has a third method (`postHandle`) as well -- the topic of "The HandlerInterceptor Interface: preHandle, postHandle, afterCompletion."

Why Does It Exist?

Adding the same logging/auth code by hand to every controller method is another instance of the repetition problem we saw in Validation & Exception Handling's "Why Does It Exist?" section -- the rule repeats everywhere, and it's easy to forget in one place. `HandlerInterceptor` moves that cross-cutting behavior to **one place** and lets `WebMvcConfigurer` decide, centrally, which URLs it applies to; the controllers themselves stay unaware of it.

History

The `HandlerInterceptor` interface has been around since Spring MVC's earliest versions -- it's as old as Spring MVC itself. Spring 5.0 (2017) made all three methods **default** (before that you had to extend the abstract class `HandlerInterceptorAdapter` just to override one method); in the version this project uses, `HandlerInterceptorAdapter` is no longer needed. CORS support arrived in Spring 4.2 (2015) with `@CrossOrigin`, and Spring 4.3 added global support through `WebMvcConfigurer.addCorsMappings` -- before that, CORS meant writing a `Filter` by hand. Multipart support entered the Servlet API with Servlet 3.0 (2009), and from there into Spring MVC through `MultipartResolver`.

Filter vs. Interceptor: Both "Get in the Way," but Where?

Both wrap code around a request, but at different layers:

```
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;

import java.io.IOException;

// Both a Filter and a HandlerInterceptor can run code "around" a request, but they
// belong to two different layers: Filter is part of the Servlet API itself (the
// container calls it, before Spring even enters the picture); HandlerInterceptor is
// a Spring MVC concept (DispatcherServlet calls it, only for requests that reach a
// handler mapping).
class FilterVsInterceptorExample {

    // A Filter sees EVERY request the servlet container receives -- static resources,
    // 404s, anything -- because it sits in front of DispatcherServlet, not inside it.
    static class LoggingFilter implements Filter {
        public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
                throws IOException, ServletException {
            System.out.println("[Filter] before -- runs for ANY request the container
handles");
            chain.doFilter(request, response);
            System.out.println("[Filter] after");
        }
    }

    // A HandlerInterceptor only sees requests DispatcherServlet has already matched to
    // a handler -- it never runs for a request that 404s before a handler is found.
    static class LoggingInterceptor implements HandlerInterceptor {
        public boolean preHandle(jakarta.servlet.http.HttpServletRequest request,
                jakarta.servlet.http.HttpServletResponse response, Object handler) {
            System.out.println("[Interceptor] preHandle -- only for requests that matched a
@Controller method");
            return true;
        }
    }

    public static void main(String[] args) {
        System.out.println("See 'The Path a Request Follows: Filter Chain + Interceptor
Chain Together'");
        System.out.println("for how these two actually nest around each other in a real
request.");
    }
}
```

`Filter` is part of the Servlet API -- the container (embedded Tomcat) runs every request through the filter chain before it ever reaches DispatcherServlet; that means even a request for a static file, or one that will end in a 404, still passes through filters. `HandlerInterceptor` only kicks in once DispatcherServlet has actually matched a request to a handler -- if there's no match, it never runs at all. We'll see exactly how these two layers nest around each other in "The Journey of a Request: Filter Chain and Interceptor Chain Together."

The HandlerInterceptor Interface: preHandle, postHandle, afterCompletion

The three callbacks correspond to three different moments:

```
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

// HandlerInterceptor has three callback points, all default methods (so you only
// override the ones you need). DispatcherServlet calls them at three different
// moments around the actual handler method call.
class HandlerInterceptorLifecycleExample implements HandlerInterceptor {

    public boolean preHandle(jakarta.servlet.http.HttpServletRequest request,
            jakarta.servlet.http.HttpServletResponse response, Object handler) {
        // Runs BEFORE the handler method. Returning false stops the chain right here --
        // the handler method (and postHandle) never run at all.
        System.out.println("1. preHandle");
        return true;
    }

    public void postHandle(jakarta.servlet.http.HttpServletRequest request,
            jakarta.servlet.http.HttpServletResponse response, Object handler, ModelAndView
modelAndView) {
        // Runs AFTER the handler method, but only if preHandle returned true AND the
        // handler didn't throw. Still has access to the ModelAndView -- can still
        // change what gets rendered.
        System.out.println("2. handler method runs here (between preHandle and
postHandle)");
        System.out.println("3. postHandle");
    }

    public void afterCompletion(jakarta.servlet.http.HttpServletRequest request,
            jakarta.servlet.http.HttpServletResponse response, Object handler, Exception
ex) {
        // Runs after the view has been rendered (or an exception occurred) -- the
        // last callback, good for cleanup/logging regardless of success or failure.
        System.out.println("4. afterCompletion" + (ex != null ? " (exception: " + ex + ")"
: ""));
    }

    public static void main(String[] args) throws Exception {
        HandlerInterceptorLifecycleExample interceptor = new
HandlerInterceptorLifecycleExample();

        // Simulating exactly what DispatcherServlet does around a successful request:
        boolean proceed = interceptor.preHandle(null, null, null);
        if (proceed) {
            interceptor.postHandle(null, null, null, null);
        }
        interceptor.afterCompletion(null, null, null, null);
        // 1. preHandle
        // 2. handler method runs here (between preHandle and postHandle)
        // 3. postHandle
        // 4. afterCompletion
    }
}
```

`preHandle` runs **before** the handler method -- returning `false` stops the chain right there, and neither the handler nor `postHandle` runs (see "Stopping a Request in preHandle: A Simple Auth/Logging Example"). `postHandle` runs after the handler completes successfully,

but **before** the view is rendered -- it can still modify the `ModelAndView`. `afterCompletion` runs after the view has been rendered, even if the handler threw an exception -- which makes it the most reliable place for cleanup/logging (see "Appendix: Mini Project — An Interceptor That Logs Request Duration").

The Journey of a Request: Filter Chain and Interceptor Chain Together

Filters and interceptors nest inside the same request:

```
import java.util.List;

// Filters wrap the ENTIRE DispatcherServlet call -- including view rendering.
// Interceptors only wrap the handler method call, and afterCompletion runs once the
// view has already been rendered, but still before the filter's "after" code, because
// the filter is still the outermost layer. This simulates that nesting order with
// plain method calls, no real Filter/HandlerInterceptor interfaces involved.
class RequestPipelineSimulationExample {

    static void filter(Runnable next) {
        System.out.println("1. Filter -- before (runs for every request the container
sees)");
        next.run();
        System.out.println("6. Filter -- after");
    }

    static void interceptorChain(List<String> interceptorNames, Runnable handler) {
        for (String name : interceptorNames) {
            System.out.println("2. " + name + ".preHandle");
        }
        System.out.println("3. Handler method runs");
        handler.run();
        for (int i = interceptorNames.size() - 1; i >= 0; i--) {
            System.out.println("4. " + interceptorNames.get(i) + ".postHandle (reverse
order)");
        }
        System.out.println("5. View rendered, then afterCompletion for each interceptor
(reverse order)");
    }

    public static void main(String[] args) {
        List<String> interceptors = List.of("AuthInterceptor", "LoggingInterceptor");

        filter(() -> interceptorChain(interceptors, () -> System.out.println("   (view
built from Model)")));
        // 1. Filter -- before (runs for every request the container sees)
        // 2. AuthInterceptor.preHandle
        // 2. LoggingInterceptor.preHandle
        // 3. Handler method runs
        //    (view built from Model)
        // 4. LoggingInterceptor.postHandle (reverse order)
        // 4. AuthInterceptor.postHandle (reverse order)
        // 5. View rendered, then afterCompletion for each interceptor (reverse order)
        // 6. Filter -- after
    }
}
```

A filter wraps DispatcherServlet's **entire** call, including view rendering -- interceptors only wrap the handler call; even `afterCompletion` runs after the view is rendered, but still before the filter's "after" code. Knowing this order answers "where should this code be logged" -- if you need to see every request (static files included), use a `Filter`; if you only care about requests that reach a controller, use a `HandlerInterceptor`.

WebMvcConfigurer: Registering an Interceptor

Implementing a `HandlerInterceptor` isn't enough on its own -- unlike a `@Component` from Component Scanning, it isn't found automatically; it has to be registered explicitly:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

// Implementing a HandlerInterceptor isn't enough by itself -- unlike a @Component
// (which Component Scanning picks up automatically), an interceptor has to be
// registered explicitly. WebMvcConfigurer is the extension point Spring MVC looks
// for at startup; a @Configuration class implementing it can override
// addInterceptors to register any number of interceptors.
@Configuration
class InterceptorRegistrationExample implements WebMvcConfigurer {

    static class SimpleLoggingInterceptor implements HandlerInterceptor {
        public boolean preHandle(jakarta.servlet.http.HttpServletRequest request,
                jakarta.servlet.http.HttpServletResponse response, Object handler) {
            System.out.println("SimpleLoggingInterceptor.preHandle: " + request);
            return true;
        }
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new SimpleLoggingInterceptor());
    }

    public static void main(String[] args) {
        // Without a real ApplicationContext, we can only show that the registration
        // call compiles and reads the way it would in a real @Configuration class --
        // registry.addInterceptor(...) is what DispatcherServlet's HandlerMapping
        // consults at startup to build its interceptor list.
        System.out.println("addInterceptors would register: " + SimpleLoggingInterceptor.
class.getSimpleName());
    }
}
```

`WebMvcConfigurer` is the extension point Spring MVC looks for at startup -- a `@Configuration` class that implements it and overrides `addInterceptors` gets every interceptor added via `registry.addInterceptor(...)` added to `HandlerMapping`'s interceptor list. This project doesn't register an interceptor right now -- `WebConfig.java` is a `@Configuration` class that only defines a `LocaleResolver` bean; if an interceptor were added, overriding `addInterceptors` on that same class would be the natural place for it.

addPathPatterns and excludePathPatterns: Scoping an Interceptor

Not every interceptor needs to run on every URL:

```
import org.springframework.util.AntPathMatcher;

// registry.addInterceptor(...).addPathPatterns(...).excludePathPatterns(...) limits
// which URLs an interceptor actually runs for. Internally, Spring MVC compares each
// incoming path against these Ant-style patterns using AntPathMatcher -- the same
// class this example uses directly, standing in for what
// InterceptorRegistration/MappedInterceptor do for you.
class PathPatternScopingExample {

    // In a real @Configuration class:
    //
    // registry.addInterceptor(new AuthInterceptor())
    //         .addPathPatterns("/topics/**")
    //         .excludePathPatterns("/topics/public/**");

    static boolean shouldRunFor(String path) {
        AntPathMatcher matcher = new AntPathMatcher();
        boolean included = matcher.match("/topics/**", path);
        boolean excluded = matcher.match("/topics/public/**", path);
        return included && !excluded;
    }

    public static void main(String[] args) {
        System.out.println(shouldRunFor("/topics/spring-mvc-fundamentals"));
        // true -- matches the include pattern, not the exclude pattern

        System.out.println(shouldRunFor("/topics/public/announcement"));
        // false -- matches the include pattern too, but the exclude pattern wins

        System.out.println(shouldRunFor("/"));
        // false -- doesn't match the include pattern at all
    }
}
```

`addPathPatterns("/topics/**")` limits an interceptor to URLs matching that pattern; `excludePathPatterns(...)` then carves a subset back out of an included pattern. Path Variables and Request Parameters covered the URL patterns used by `@GetMapping` -- the `/**` here uses the exact same Ant-style matching, the only difference is that it scopes an interceptor instead of a handler method.

Multiple Interceptors: Ordering and Chaining

With more than one interceptor registered, order matters:

```
import java.util.ArrayList;
import java.util.List;

// With more than one interceptor registered, preHandle runs in REGISTRATION order,
// but postHandle/afterCompletion run in REVERSE order -- the same "wrapping" pattern
// as try-with-resources closing multiple resources, or a stack of middleware. This
// matters when interceptors depend on each other (e.g. one sets a request attribute
// another one reads).
class MultipleInterceptorOrderExample {

    record NamedInterceptor(String name) {
    }

    static void runChain(List<NamedInterceptor> interceptors) {
        List<NamedInterceptor> executedPreHandle = new ArrayList<>();

        for (NamedInterceptor interceptor : interceptors) {
            System.out.println(interceptor.name() + ".preHandle");
            executedPreHandle.add(interceptor);
        }

        System.out.println("(handler runs)");

        // postHandle/afterCompletion only run for interceptors whose preHandle
        // already completed, and in reverse -- this is what makes it safe for
        // AuthInterceptor to assume LoggingInterceptor's preHandle already ran.
        for (int i = executedPreHandle.size() - 1; i >= 0; i--) {
            System.out.println(executedPreHandle.get(i).name() + ".postHandle");
        }
    }

    public static void main(String[] args) {
        runChain(List.of(new NamedInterceptor("AuthInterceptor"), new NamedInterceptor
("LoggingInterceptor")));
        // AuthInterceptor.preHandle
        // LoggingInterceptor.preHandle
        // (handler runs)
        // LoggingInterceptor.postHandle
        // AuthInterceptor.postHandle
    }
}
```

`preHandle` calls run in **registration order**; `postHandle` and `afterCompletion` run in **reverse order** -- the same "stack" pattern as try-with-resources closing multiple resources. This lets one interceptor safely assume another interceptor's `preHandle` has already run -- for instance, a logging interceptor can safely read, in its `postHandle`, a user attribute an auth interceptor placed on the request.

Stopping a Request in preHandle: A Simple Auth/Logging Example

`preHandle`'s ability to return `false` also makes it usable for a simple access check:

```
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;

// preHandle returning false stops the chain immediately -- neither the handler
// method nor any later interceptor's preHandle runs. This is the standard place for
// cross-cutting checks (auth, rate limiting) that should reject a request before any
// business logic executes.
class AuthLoggingInterceptorExample implements HandlerInterceptor {

    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) {
        String apiKey = request.getHeader("X-Api-Key");
        System.out.println("Checking X-Api-Key: " + apiKey);

        if (apiKey == null) {
            response.setStatus(401);
            System.out.println("-> rejected, handler will NOT run");
            return false;
        }

        System.out.println("-> accepted, handler will run");
        return true;
    }

    // A tiny fake HttpServletRequest/HttpServletResponse, only implementing the two
    // methods this example actually calls -- everything else throws, since a real
    // implementation would need a full servlet container.
    static HttpServletRequest fakeRequest(String apiKeyHeaderValue) {
        return (HttpServletRequest) java.lang.reflect.Proxy.newProxyInstance(
                HttpServletRequest.class.getClassLoader(),
                new Class<?>[]{HttpServletRequest.class},
                (proxy, method, methodArgs) -> {
                    if (method.getName().equals("getHeader")) {
                        return apiKeyHeaderValue;
                    }
                    throw new UnsupportedOperationException(method.getName());
                });
    }

    static HttpServletResponse fakeResponse(int[] statusHolder) {
        return (HttpServletResponse) java.lang.reflect.Proxy.newProxyInstance(
                HttpServletResponse.class.getClassLoader(),
                new Class<?>[]{HttpServletResponse.class},
                (proxy, method, methodArgs) -> {
                    if (method.getName().equals("setStatus")) {
                        statusHolder[0] = (int) methodArgs[0];
                        return null;
                    }
                    throw new UnsupportedOperationException(method.getName());
                });
    }

    public static void main(String[] args) {
        AuthLoggingInterceptorExample interceptor = new AuthLoggingInterceptorExample();

        int[] status = {200};
        boolean allowed = interceptor.preHandle(fakeRequest(null), fakeResponse(status),
```

```
null);
        System.out.println("allowed=" + allowed + ", status=" + status[0]);
        // Checking X-Api-Key: null
        // -> rejected, handler will NOT run
        // allowed=false, status=401

        boolean allowedWithKey = interceptor.preHandle(fakeRequest("secret-123"),
fakeResponse(status), null);
        System.out.println("allowed=" + allowedWithKey);
        // Checking X-Api-Key: secret-123
        // -> accepted, handler will run
        // allowed=true
    }
}
```

The `response.setStatus(401)` call matters here -- returning `false` stops the chain, but unless you set the status code **yourself**, the client still gets a default `200`. This is a heavily simplified version of what a real security framework (like Spring Security) does -- this project doesn't use Spring Security, but the core idea (reject the request before it reaches a handler) is exactly the same.

What Is CORS? Same-Origin Policy and the Preflight Request

Browsers block a page from reading data from a different origin (scheme+host+port) by default -- the **same-origin policy**. CORS is the standard way for a server to say "I allow this origin":

```
import org.springframework.web.cors.CorsConfiguration;

// Same-origin policy: a browser blocks a page at origin A (scheme+host+port) from
// reading a response from origin B, unless B's server explicitly allows it via CORS
// (Cross-Origin Resource Sharing) response headers. For "unsafe" requests (anything
// other than a simple GET/HEAD/POST with a plain content type), the browser sends a
// preflight OPTIONS request FIRST, asking "would you allow this?" -- and only sends
// the real request if the answer is yes. Spring's CorsConfiguration is the object
// that answers that question; this example uses it directly, no HTTP involved.
class CorsPreflightExample {

    public static void main(String[] args) {
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedOrigin("https://learning-platform.example.com");
        config.addAllowedMethod("GET");
        config.addAllowedMethod("POST");
        config.addAllowedHeader("Content-Type");

        // checkOrigin returns the allowed origin to echo back in the response header,
        // or null if this origin isn't allowed at all.
        System.out.println(config.checkOrigin("https://learning-platform.example.com"));
        // https://learning-platform.example.com

        System.out.println(config.checkOrigin("https://evil.example.com"));
        // null -- browser will block the response from reaching JavaScript

        // checkHttpMethod returns the allowed methods, or null if the requested one
        // (what the preflight's Access-Control-Request-Method asked about) isn't allowed.
        System.out.println(config.checkHttpMethod("DELETE"));
        // null -- DELETE was never added as an allowed method
    }
}
```

For a request that isn't "simple" (say, one carrying a custom header, or using a method other than `GET` / `POST`), the browser first sends a **preflight** -- without sending the real request at all, it asks, using `OPTIONS`, "am I allowed to make this request?" If the server doesn't answer with the right `Access-Control-Allow-*` headers, the browser never sends the real request. `CorsConfiguration` is the object that produces that answer -- its `checkOrigin` / `checkHttpMethod` methods compute the response to whatever the browser is asking.

@CrossOrigin: CORS at the Controller/Method Level

`@CrossOrigin` is the way to declare CORS for individual endpoints:

```
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

// @CrossOrigin is the per-controller/per-method alternative to a global CORS config --
// Spring reads it with reflection at startup (the same mechanism the Reflection lesson
// covered) and builds a CorsConfiguration from its attributes, exactly like the one
// CorsPreflightExample builds by hand.
@RestController
class CrossOriginAnnotationExample {

    @CrossOrigin(origins = "https://learning-platform.example.com", methods = {
            org.springframework.web.bind.annotation.RequestMethod.GET})
    @GetMapping("/api/topics")
    public String listTopics() {
        return "[]";
    }

    public static void main(String[] args) throws NoSuchMethodException {
        Method method = CrossOriginAnnotationExample.class.getMethod("listTopics");
        CrossOrigin annotation = method.getAnnotation(CrossOrigin.class);

        System.out.println("origins: " + java.util.Arrays.toString(annotation.origins()));
        // origins: [https://learning-platform.example.com]
        System.out.println("methods: " + java.util.Arrays.toString(annotation.methods()));
        // methods: [GET]

        // This is conceptually all Spring itself does at startup: scan each handler
        // method for a @CrossOrigin (via getAnnotation, just like above), and if
        // present, register a matching CorsConfiguration for that mapping.
        Annotation[] all = method.getAnnotations();
        System.out.println("total annotations on listTopics: " + all.length);
        // total annotations on listTopics: 2  -- @CrossOrigin and @GetMapping
    }
}
```

The same `getAnnotation` mechanism from the Reflection lesson is at work here too -- at startup, Spring scans every handler method, and if `@CrossOrigin` is present, builds a `CorsConfiguration` from its attributes (`origins` , `methods` , ...) and stores it for that mapping. It's the exact same mechanism that reads `@RequestMapping` and its shortcuts (the Mapping Annotations and HTTP Methods lesson).

Global CORS Configuration with WebMvcConfigurer

Instead of adding `@CrossOrigin` to every controller, you can also configure CORS for all of `/api/**` in one place:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

// The same WebMvcConfigurer that registers interceptors (see "WebMvcConfigurer:
// Registering an Interceptor") also has an addCorsMappings hook -- the global
// alternative to sprinkling @CrossOrigin over every controller method. One
// CorsRegistration per URL pattern, each building the same kind of CorsConfiguration
// CorsPreflightExample constructed by hand.
@Configuration
class GlobalCorsConfigExample implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("https://learning-platform.example.com")
                .allowedMethods("GET", "POST")
                .allowedHeaders("Content-Type")
                .allowCredentials(true)
                .maxAge(3600);
    }

    public static void main(String[] args) {
        System.out.println("addCorsMappings applies to every /api/** endpoint --");
        System.out.println("no per-controller @CrossOrigin needed, and no risk of
forgetting one.");
    }
}
```

`addCorsMappings` is another method on the same `WebMvcConfigurer` interface as `addInterceptors` from "WebMvcConfigurer: Registering an Interceptor" -- both can live in the same `@Configuration` class. If a URL pattern matches more than one `CorsRegistration` (say, one global and one via `@CrossOrigin`), Spring tries to merge them, but in practice it's usually best to pick **either** global **or** annotation-based, not both, to avoid confusion.

Multipart File Upload: Taking a MultipartFile with @RequestParam

File uploads use a different mechanism than `@RequestBody` (Request and Response Handling), which reads a single JSON body:

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;

// multipart/form-data is the content type browsers use for file uploads -- unlike
```

```
// @RequestBody (which reads one JSON body via a single HttpMessageConverter), a
// multipart request is split into named parts, and MultipartFile binds one of them
// straight to a controller parameter, the same way @RequestParam binds a plain form
// field (see Path Variable'lar ve Request Parametreleri).
@RestController
class MultipartUploadControllerExample {

    @PostMapping("/upload")
    public String upload(@RequestParam("file") MultipartFile file) {
        if (file.isEmpty()) {
            return "empty file";
        }
        return "received " + file.getOriginalFilename() + " (" + file.getSize() + " bytes)";
    }

    // A minimal hand-written MultipartFile, standing in for the real implementation
    // Spring builds from the actual HTTP request -- just enough to exercise upload()
    // without a servlet container.
    static class InMemoryMultipartFile implements MultipartFile {
        private final String originalFilename;
        private final byte[] content;

        InMemoryMultipartFile(String originalFilename, byte[] content) {
            this.originalFilename = originalFilename;
            this.content = content;
        }

        public String getName() {
            return "file";
        }

        public String getOriginalFilename() {
            return originalFilename;
        }

        public String getContentType() {
            return "text/plain";
        }

        public boolean isEmpty() {
            return content.length == 0;
        }

        public long getSize() {
            return content.length;
        }

        public byte[] getBytes() {
            return content;
        }

        public InputStream getInputStream() {
            return new ByteArrayInputStream(content);
        }

        public void transferTo(File dest) throws IOException, IllegalStateException {
            throw new UnsupportedOperationException("not needed for this example");
        }
```

```
    }

    public static void main(String[] args) {
        MultipartUploadControllerExample controller = new
MultipartUploadControllerExample();

        System.out.println(controller.upload(new InMemoryMultipartFile("notes.txt", "hello".
getBytes())));
        // received notes.txt (5 bytes)

        System.out.println(controller.upload(new InMemoryMultipartFile("empty.txt", new byte
[0])));
        // empty file
    }
}
```

`multipart/form-data` splits a request into named **parts** -- each part can be a plain form field or a file. `MultipartFile` is bound with `@RequestParam`, the same way as "@RequestParam: Reading a Value from the Query String" in Path Variables and Request Parameters, but what it reads isn't a query parameter, it's a part of the request -- accessed through methods like `getOriginalFilename()`, `getSize()`, `getBytes()`.

Multipart Configuration and Size Limits

`spring.servlet.multipart.max-file-size` / `max-request-size` set an upper bound Spring rejects before a request ever reaches a handler:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.multipart.MaxUploadSizeExceededException;

// spring.servlet.multipart.max-file-size / max-request-size (application.properties)
// set hard limits Spring enforces BEFORE your controller method ever runs -- exceeding
// them throws MaxUploadSizeExceededException, which reaches DispatcherServlet as a
// regular exception. It fits the same @RestControllerAdvice mechanism from Validation
// & Exception Handling -- turning an error into a consistent, standard response.
@RestControllerAdvice
class MultipartSizeLimitExample {

    @ExceptionHandler(MaxUploadSizeExceededException.class)
    public ProblemDetail handleTooLarge(MaxUploadSizeExceededException ex) {
        ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.PAYLOAD_TOO_LARGE);
        problem.setTitle("File Too Large");
        problem.setDetail("Uploaded file exceeds the configured size limit.");
        return problem;
    }

    public static void main(String[] args) {
        MultipartSizeLimitExample advice = new MultipartSizeLimitExample();

        MaxUploadSizeExceededException ex = new MaxUploadSizeExceededException(5 * 1024 *
1024);
        ProblemDetail problem = advice.handleTooLarge(ex);

        System.out.println(problem.getStatus() + " " + problem.getTitle() + " -- " +
problem.getDetail());
        // 413 File Too Large -- Uploaded file exceeds the configured size limit.

        // In application.properties, the limit that triggered this would be:
        //   spring.servlet.multipart.max-file-size=5MB
        //   spring.servlet.multipart.max-request-size=10MB
    }
}
```

The `MaxUploadSizeExceededException` thrown when the limit is exceeded is caught the same way as in Validation & Exception Handling's "Catching Errors at the Controller Level: @ExceptionHandler" and "Global Error Handling: @RestControllerAdvice" sections; returning a standard error body like "ProblemDetail: A Standard Error Body with RFC 7807" is a much better behavior than letting a raw stack trace leak to the client.

Best Practices

- **Only reach for `Filter` when you genuinely need to see every request (static files included); otherwise prefer `HandlerInterceptor`** -- the latter works closer to Spring's own mechanisms (Model, exception handling) (see Filter vs. Interceptor: Both "Get in the Way," but Where?).

- **Put cleanup/logging code in `afterCompletion`, not `postHandle`** -- only `afterCompletion` runs even if the handler throws (see "The HandlerInterceptor Interface: preHandle, postHandle, afterCompletion").
- **Manage CORS either globally (`WebMvcConfigurer.addCorsMappings`) or via annotation (`@CrossOrigin`), not a mix of both** -- mixing the two makes it harder to track which rule applies to which endpoint (see "Global CORS Configuration with WebMvcConfigurer").
- **Always configure multipart size limits explicitly** -- the defaults (1MB in Spring Boot) are either far too low or simply left unconsidered for most real upload scenarios; either way, it should be a deliberate decision (see "Multipart Configuration and Size Limits").

Common Mistakes

1. Assuming `Filter` and `HandlerInterceptor` are interchangeable. A `Filter` has no access to Spring concepts like `Model`/`HandlerMethod` -- it only sees the raw `ServletRequest`/`ServletResponse`; if you need to know which controller a request matched, `HandlerInterceptor` is the right tool (see Filter vs. Interceptor: Both "Get in the Way," but Where?).

2. Returning `false` from `preHandle` and forgetting to set the response status. The chain stops, but the client still gets a default `200 OK` -- you have to call `response.setStatus(...)` before returning `false` (see "Stopping a Request in preHandle: A Simple Auth /Logging Example").

3. Assuming `postHandle` also runs in registration order with multiple interceptors. `preHandle` runs forward, `postHandle`/`afterCompletion` run in reverse -- forgetting this can lead an interceptor to read another one's state at the wrong moment (see "Multiple Interceptors: Ordering and Chaining").

4. Looking for a CORS error in server logs, thinking it's a server-side failure. When a preflight fails, the browser never sends the real request at all -- nothing may show up in the server log; the error only appears in the browser's own developer console (see "What Is CORS? Same-Origin Policy and the Preflight Request").

5. Adding `@CrossOrigin` only at the class level and forgetting that a method-level one overrides it. A class-level `@CrossOrigin` applies to every method by default, but a method that declares its own `@CrossOrigin` **completely replaces** the class-level one rather than merging with it -- this can unexpectedly strip CORS permissions from some endpoints (see "@CrossOrigin: CORS at the Controller/Method Level").

6. Setting only `max-file-size` and forgetting `max-request-size`. In a request with multiple files, each one can stay under the per-file limit while the total still exceeds `max-request-size` -- they're two separate limits, and both need to be configured (see "Multipart Configuration and Size Limits").

Summary, Cheat Sheet, and Glossary

`Filter` and `HandlerInterceptor` are two layers for adding cross-cutting behavior around a request; `WebMvcConfigurer` is the central place both (interceptor registration, CORS) get configured; CORS and multipart are two concrete scenarios real-world applications run into often. Key points:

- `Filter`: Servlet API level, sees **every** request (outside DispatcherServlet)
- `HandlerInterceptor`: Spring MVC level, only sees matched requests (`preHandle` / `postHandle` / `afterCompletion`)
- If `preHandle` returns `false`, the chain stops -- the handler and `postHandle` never run
- `afterCompletion` always runs, even on exception -- the most reliable place for cleanup /logging
- Multiple interceptors: `preHandle` runs forward, `postHandle` / `afterCompletion` run in reverse
- `WebMvcConfigurer.addInterceptors` / `addCorsMappings`: extension points for interceptor registration and global CORS
- CORS: the server-side relaxation of the same-origin policy; preflight is a permission question asked ahead of time with `OPTIONS`
- `@CrossOrigin`: CORS at the controller/method level, an alternative to `WebMvcConfigurer`
- `MultipartFile`: an interface representing one part of a `multipart/form-data` request, bound with `@RequestParam`
- `max-file-size` / `max-request-size`: two separate size limits for multipart uploads

Quick reference:

```
@Configuration
class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new AuthInterceptor())
                .addPathPatterns("/api/**")
                .excludePathPatterns("/api/public/**");
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("https://example.com")
                .allowedMethods("GET", "POST");
    }
}

@RestController
class UploadController {
    @PostMapping("/upload")
    ResponseEntity<String> upload(@RequestParam("file") MultipartFile file) {
        return ResponseEntity.ok(file.getOriginalFilename() + ": " + file.getSize() + "
bytes");
    }
}
```

Glossary

`Filter` — Part of the Servlet API; an interface the container runs every request through before it reaches DispatcherServlet, used for cross-cutting behavior.

`HandlerInterceptor` — A Spring MVC-specific interface that wraps only requests matched to a handler, with `preHandle` / `postHandle` / `afterCompletion` callbacks.

`WebMvcConfigurer` — The extension point `@Configuration` classes implement to configure Spring MVC concerns like interceptor registration and CORS.

Same-origin policy — The browser security rule that blocks a page from reading data from a different origin by default.

CORS (Cross-Origin Resource Sharing) — The HTTP header mechanism that lets a server relax the same-origin policy for specific origins.

Preflight request — The browser's advance `OPTIONS` request asking the server for permission before sending a "non-simple" request.

`@CrossOrigin` — An annotation that declares CORS permissions at the controller class or method level.

`MultipartFile` — A Spring interface representing a single file part of a `multipart/form-data` request.

`MaxUploadSizeExceededException` — The exception Spring throws when a configured size limit is exceeded, catchable with `@ExceptionHandler`.

Appendix: Mini Project — An Interceptor That Logs Request Duration

Bringing this lesson's `HandlerInterceptor` mechanics together in a realistic scenario: an interceptor that starts a timer in `preHandle` and logs the elapsed time in `afterCompletion`, whether the handler succeeded or not:

```
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;

// Mini project, part 1/2: a small but realistic request-timing interceptor -- starts
// a timer in preHandle, logs the elapsed time in afterCompletion. Using
// afterCompletion (not postHandle) matters here: it still runs even if the handler
// threw an exception, so a slow-then-failing request is still logged.
class RequestLoggingInterceptorExample implements HandlerInterceptor {

    private static final String START_TIME_ATTRIBUTE = "requestStartTime";

    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) {
        request.setAttribute(START_TIME_ATTRIBUTE, System.currentTimeMillis());
        return true;
    }

    public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler,
            Exception ex) {
        long startTime = (long) request.getAttribute(START_TIME_ATTRIBUTE);
        long elapsedMs = System.currentTimeMillis() - startTime;

        String outcome = ex == null ? "OK" : "FAILED (" + ex.getClass().getSimpleName() +
")";
        System.out.println(request.getRequestURI() + " -> " + outcome + " in " + elapsedMs
+ "ms");
    }
}
```

```
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.lang.reflect.Proxy;
import java.util.HashMap;
import java.util.Map;

// Mini project, part 2/2: drives RequestLoggingInterceptorExample through a
// successful request and a failing one, using a tiny reflection-based fake
// HttpServletRequest (same technique as AuthLoggingInterceptorExample) that actually
// backs setAttribute/getAttribute with a real Map, since the interceptor depends on
// that round-trip.
class RequestLoggingInterceptorDemo {

    static HttpServletRequest fakeRequest(String uri) {
        Map<String, Object> attributes = new HashMap<>();
        return (HttpServletRequest) Proxy.newProxyInstance(
                HttpServletRequest.class.getClassLoader(),
                new Class<?>[]{HttpServletRequest.class},
                (proxy, method, methodArgs) -> switch (method.getName()) {
                    case "setAttribute" -> {
                        attributes.put((String) methodArgs[0], methodArgs[1]);
                        yield null;
                    }
                    case "getAttribute" -> attributes.get((String) methodArgs[0]);
                    case "getRequestURI" -> uri;
                    default -> throw new UnsupportedOperationException(method.getName());
                });
    }

    public static void main(String[] args) throws InterruptedException {
        RequestLoggingInterceptorExample interceptor = new
RequestLoggingInterceptorExample();
        HttpServletResponse fakeResponse = (HttpServletResponse) Proxy.newProxyInstance(
                HttpServletResponse.class.getClassLoader(),
                new Class<?>[]{HttpServletResponse.class},
                (proxy, method, methodArgs) -> {
                    throw new UnsupportedOperationException(method.getName());
                });

        HttpServletRequest okRequest = fakeRequest("/topics/advanced-spring-mvc");
        interceptor.preHandle(okRequest, fakeResponse, null);
        Thread.sleep(5);
        interceptor.afterCompletion(okRequest, fakeResponse, null, null);
        // /topics/advanced-spring-mvc -> OK in Xms

        HttpServletRequest failingRequest = fakeRequest("/topics/does-not-exist");
        interceptor.preHandle(failingRequest, fakeResponse, null);
        interceptor.afterCompletion(failingRequest, fakeResponse, null,
                new IllegalStateException("Topic not found"));
        // /topics/does-not-exist -> FAILED (IllegalStateException) in Xms
    }
}
```

Using `afterCompletion` is a deliberate choice -- as we saw in "The HandlerInterceptor Interface: preHandle, postHandle, afterCompletion," this callback runs even if the handler

throws, so a request that's both slow **and** failing still gets logged correctly. `RequestLoggingInterceptorDemo` simulates both a successful and a failing request to show both getting logged.

Appendix: Mini Project — A CORS-Enabled File Upload Endpoint

The last mini project brings three of this lesson's topics together (`@CrossOrigin`, `MultipartFile`, and `@ExceptionHandler` for an exceeded size limit) into a single endpoint:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MaxUploadSizeExceededException;
import org.springframework.web.multipart.MultipartFile;

// Mini project, part 1/2: a single endpoint combining three mechanics from this
// lesson -- @CrossOrigin (so a browser-based frontend on a different origin can call
// it), MultipartFile (the actual upload), and an @ExceptionHandler for when the file
// is too large (same idea as MultipartSizeLimitExample, kept local to this
// controller instead of a separate @RestControllerAdvice).
@RestController
class FileUploadCorsController {

    private static final long MAX_BYTES = 1024; // deliberately tiny, to make the demo
trigger it

    @CrossOrigin(origins = "https://learning-platform.example.com")
    @PostMapping("/api/avatar")
    public ResponseEntity<String> uploadAvatar(@RequestParam("file") MultipartFile file) {
        if (file.getSize() > MAX_BYTES) {
            throw new MaxUploadSizeExceededException(MAX_BYTES);
        }
        return ResponseEntity.ok("Stored " + file.getOriginalFilename() + " (" + file.
getSize() + " bytes)");
    }

    @ExceptionHandler(MaxUploadSizeExceededException.class)
    public ProblemDetail handleTooLarge(MaxUploadSizeExceededException ex) {
        ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.PAYLOAD_TOO_LARGE);
        problem.setTitle("Avatar Too Large");
        problem.setDetail("Avatar must be at most " + MAX_BYTES + " bytes.");
        return problem;
    }
}
```

```
import org.springframework.http.ProblemDetail;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.multipart.MaxUploadSizeExceededException;
import org.springframework.web.multipart.MultipartFile;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.InputStream;

// Mini project, part 2/2: calls FileUploadCorsController.uploadAvatar directly with
// a small file (succeeds) and a large one (throws, then dispatch() catches it and
// routes it to the controller's own @ExceptionHandler -- the same pattern
// UserRegistrationDemo used in Validation & Exception Handling, just for a different
// exception type).
class FileUploadCorsDemo {

    static class InMemoryMultipartFile implements MultipartFile {
        private final String originalFilename;
        private final byte[] content;

        InMemoryMultipartFile(String originalFilename, byte[] content) {
            this.originalFilename = originalFilename;
            this.content = content;
        }

        public String getName() {
            return "file";
        }

        public String getOriginalFilename() {
            return originalFilename;
        }

        public String getContentType() {
            return "image/png";
        }

        public boolean isEmpty() {
            return content.length == 0;
        }

        public long getSize() {
            return content.length;
        }

        public byte[] getBytes() {
            return content;
        }

        public InputStream getInputStream() {
            return new ByteArrayInputStream(content);
        }

        public void transferTo(File dest) {
            throw new UnsupportedOperationException("not needed for this example");
        }
    }

    static String dispatch(FileUploadCorsController controller, MultipartFile file) {
        try {
```

```
            ResponseEntity<String> response = controller.uploadAvatar(file);
            return response.getStatusCode() + " " + response.getBody();
        } catch (MaxUploadSizeExceededException ex) {
            ProblemDetail problem = controller.handleTooLarge(ex);
            return problem.getStatus() + " " + problem.getTitle() + ": " + problem.
getDetail();
        }
    }

    public static void main(String[] args) {
        FileUploadCorsController controller = new FileUploadCorsController();

        System.out.println(dispatch(controller, new InMemoryMultipartFile("avatar.png",
"small".getBytes())));
        // 200 OK Stored avatar.png (5 bytes)

        byte[] tooLarge = new byte[2048];
        System.out.println(dispatch(controller, new InMemoryMultipartFile("huge.png",
tooLarge)));
        // 413 Avatar Too Large: Avatar must be at most 1024 bytes.
    }
}
```

`FileUploadCorsController` carries `@CrossOrigin` so it can be called from a different origin (a separate frontend application, for instance), takes the file through a `MultipartFile` parameter, and returns a `ProblemDetail` for a file that exceeds the size limit through its own `@ExceptionHandler` (this time **local** to the controller, instead of a `@RestControllerAdvice` like in Validation & Exception Handling). `FileUploadCorsDemo` shows two calls -- a successful one with a small file, and a 413 result with a large one.