Advanced Spring MVC

HandlerInterceptor (preHandle/postHandle/afterCompletion) ve Filter vs Interceptor ayrımı; WebMvcConfigurer ile interceptor kaydı ve global CORS yapılandırması; same-origin policy, preflight request ve @CrossOrigin; MultipartFile ile dosya yükleme ve boyut sınırları.

İleri 45 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/advanced-spring-mvc

---

Advanced Spring MVC

Spring MVC Fundamentals'ın "Bir HTTP İsteğinin Yolculuğu: Request Lifecycle" bölümünde bir isteğin DispatcherServlet'e kadar nasıl geldiğini, oradan `HandlerMapping` / `HandlerAdapter` ile bir controller metoduna nasıl yönlendirildiğini gördük. O yolculuğun üzerine, her isteğe **kesişen** (cross-cutting) davranış eklemenin iki yolu var: `Filter` (Servlet API seviyesinde, DispatcherServlet'in dışında) ve `HandlerInterceptor` (Spring MVC seviyesinde, DispatcherServlet'in içinde). Bu ders bu ikisini, ikisinin de yapılandırıldığı `WebMvcConfigurer`'ı, tarayıcıların farklı origin'ler arası istekleri nasıl kısıtladığını (CORS) ve dosya yüklemenin (`multipart/form-data`) bu boru hattına nasıl oturduğunu ele alıyor.

HandlerInterceptor Nedir?

`HandlerInterceptor`, bir controller metodu çağrılmadan **önce**, çağrıldıktan **sonra** ve yanıt tamamen bittiğinde çalışacak kod yazmanı sağlayan bir Spring MVC arayüzü -- loglama, kimlik doğrulama, performans ölçümü gibi, birçok endpoint'te tekrar eden ama endpoint'in kendi iş mantığına ait olmayan davranışlar için:

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

Gerçek arayüz `jakarta.servlet.http.HttpServletRequest` / `HttpServletResponse` kullanır ve üçüncü bir metot (`postHandle`) daha taşır -- "HandlerInterceptor Arayüzü: preHandle, postHandle, afterCompletion" bölümünün konusu.

Neden Var?

Her controller metoduna aynı loglama/auth kodunu elle eklemek, Validation & Exception Handling dersindeki "Neden Var?" bölümünde gördüğümüz tekrar sorununun bir başka örneği -- kural her yerde tekrar eder, bir yerde unutulması kolaydır. `HandlerInterceptor`, bu kesişen davranışı **tek bir yere** taşır ve `WebMvcConfigurer` üzerinden hangi URL'lere uygulanacağını merkezi olarak belirler; controller'ların kendisi bundan habersiz kalır.

Tarihçe

`HandlerInterceptor` arayüzü Spring'in ilk sürümlerinden beri var -- Spring MVC'nin kendisi kadar eski. Spring 5.0 (2017), üç metodu da **default** yaptı (öncesinde soyut sınıf `HandlerInterceptorAdapter`'dan türetmek gerekiyordu, yalnızca ihtiyaç duyulan metodu override etmek için); bu projenin kullandığı sürümde `HandlerInterceptorAdapter` artık gereksiz. CORS desteği Spring 4.2'de (2015) `@CrossOrigin` ile, Spring 4.3'te de global `WebMvcConfigurer.addCorsMappings` ile geldi -- ondan önce CORS için elle bir `Filter` yazmak gerekiyordu. Multipart desteği ise Servlet 3.0 (2009) ile Servlet API'sine, oradan da Spring MVC'ye `MultipartResolver` üzerinden girdi.

Filter vs Interceptor: İkisi de "Araya Girer" ama Nerede?

İkisi de bir isteğin etrafına kod sarar, ama farklı katmanlarda:

```
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`, Servlet API'nin bir parçası -- container (embedded Tomcat) her isteği DispatcherServlet'e ulaştırmadan önce filter zincirinden geçirir; bu yüzden statik bir dosya isteği ya da 404 ile sonuçlanacak bir istek bile filter'lardan geçer. `HandlerInterceptor` ise yalnızca DispatcherServlet bir isteği gerçekten bir handler'a eşleştirdiğinde devreye girer -- eşleşme yoksa hiç çalışmaz. Bu iki katmanın gerçekte nasıl iç içe geçtiğini "Bir İsteğin İzlediği Yol: Filter Chain + Interceptor Chain Birlikte" bölümünde göreceğiz.

HandlerInterceptor Arayüzü: preHandle, postHandle, afterCompletion

Üç callback, üç farklı ana karşılık gelir:

```
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`, handler metodundan **önce** çalışır -- `false` dönmesi zinciri hemen durdurur, ne handler ne `postHandle` çalışır (bkz. "preHandle'da İsteği Durdurmak: Basit Bir Auth/Logging Örneği"). `postHandle`, handler başarıyla tamamlandıktan sonra, view render edilmeden **önce**

çalışır -- hâlâ `ModelAndView`'i değiştirebilir. `afterCompletion` ise view render edildikten sonra çalışır, handler bir exception fırlatmış olsa bile -- bu yüzden temizlik/loglama için en güvenilir nokta odur (bkz. "Ek: Mini Proje — İstek Süresini Loglayan Bir Interceptor").

Bir İsteğin İzlediği Yol: Filter Chain + Interceptor Chain Birlikte

Filter'lar ile interceptor'lar aynı istekte iç içe çalışır:

```
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
    }
}
```

Filter, DispatcherServlet'in **tüm** çağrısını (view render dahil) sarar -- interceptor'lar ise yalnızca handler çağrısını sarar, `afterCompletion` bile view render edildikten sonra ama filter'ın "after" kodundan önce çalışır. Bu sıralamayı bilmek, "hangi kod nerede loglanmalı" sorusuna doğru cevabı verir -- tüm istekleri (statik dosyalar dahil) görmek istiyorsan `Filter`, yalnızca controller'a ulaşan istekleri görmek istiyorsan `HandlerInterceptor`.

WebMvcConfigurer: Interceptor'ı Kaydetmek

Bir `HandlerInterceptor` implement etmek yetmez -- Component Scanning dersindeki `@Component` gibi otomatik bulunmaz, açıkça kaydedilmesi gerekir:

```
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`, Spring MVC'nin başlangıçta aradığı bir genişletme noktası -- `@Configuration` işaretli bir sınıf bunu implement edip `addInterceptors`'ı override ettiğinde, `registry.addInterceptor(...)` ile eklenen her interceptor, `HandlerMapping`'in interceptor listesine katılır. Bu proje şu an bir interceptor kaydetmiyor -- `WebConfig.java`, yalnızca `LocaleResolver` bean'i tanımlayan bir `@Configuration` sınıfı; bir interceptor eklenecek olsa `addInterceptors`'ı override ederek aynı sınıfa taşınabilirdi.

addPathPatterns ve excludePathPatterns: Interceptor'ı Sınırlamak

Her interceptor her URL'de çalışmak zorunda değil:

```
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/**")` bir interceptor'ı yalnızca o desenle eşleşen URL'lere sınırlar; `excludePathPatterns(...)` ise dahil edilmiş bir desen içinden belirli bir alt kümeyi hariç tutar. Path Variable'lar ve Request Parametreleri dersinde `@GetMapping`'in URL desenlerini gördük -- buradaki `/**` de aynı Ant-style eşleştirmeyi kullanıyor, tek fark bunun bir handler metodunu değil bir interceptor'ı kapsaması.

Çoklu Interceptor: Sıralama ve Zincirleme

Birden fazla interceptor kayıtlıysa, sıralama önemli:

```
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` çağrıları **kayıt sırasıyla** çalışır; `postHandle` ve `afterCompletion` ise **ters sırayla** -- try-with-resources'ın kaynakları kapatma sırasına benzer bir "yığın" (stack) deseni. Bu, bir interceptor'ın diğerinin `preHandle`'ının zaten çalıştığını güvenle varsayabilmesini sağlar -- örneğin bir loglama interceptor'ı, bir auth interceptor'ın request'e koyduğu kullanıcı bilgisine `postHandle`'da güvenle erişebilir.

preHandle'da İsteği Durdurmak: Basit Bir Auth/Logging Örneği

`preHandle`'ın `false` dönme yeteneği, onu basit bir erişim kontrolü için de kullanılabilir kılar:

```
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
    }
}
```

Burada `response.setStatus(401)` çağrısı önemli -- `false` dönmek zinciri durdurur ama yanıt kodunu **kendin** ayarlamazsan istemci varsayılan 200 alır. Bu, gerçek bir güvenlik framework'ünün (Spring Security gibi) yaptığının çok basitleştirilmiş bir hâli -- bu proje Spring Security kullanmıyor, ama mekanizmanın temel fikri (isteği handler'a ulaşmadan reddetmek) birebir aynı.

CORS Nedir? Same-Origin Policy ve Preflight Request

Tarayıcılar, bir sayfanın farklı bir origin'den (şema+host+port) veri okumasını varsayılan olarak engeller -- **same-origin policy**. CORS, sunucunun "bu origin'e izin veriyorum" demesinin standart yolu:

```
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
    }
}
```

Basit olmayan bir istek (örneğin özel bir header taşıyan ya da `GET` / `POST` dışında bir metotla yapılan istek) için tarayıcı önce bir **preflight** gönderir -- gerçek isteği hiç göndermeden, `OPTIONS` metoduyla "bu isteği yapabilir miyim?" diye sorar. Sunucu doğru `Access-Control-Allow-*` header'larıyla yanıt vermezse, tarayıcı gerçek isteği hiç göndermez. `CorsConfiguration`, bu kararı üreten nesne -- `checkOrigin` / `checkHttpMethod` metotları, tarayıcının sorduğu sorulara verilecek cevabı hesaplar.

@CrossOrigin: Controller/Metot Seviyesinde CORS

CORS'u tek tek endpoint'lere tanımlamanın yolu `@CrossOrigin`:

```
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
    }
}
```

Reflection dersinde gördüğümüz `getAnnotation` mekanizması burada da aynen işliyor -- Spring, uygulama başlarken her handler metodunu tarar, `@CrossOrigin` varsa attribute'larından (`origins`, `methods`, ...) bir `CorsConfiguration` inşa eder ve o mapping için saklar. `@RequestMapping` ve arkadaşlarının nasıl okunduğuyla (Mapping Annotation'ları ve HTTP Metotları dersi) birebir aynı mekanizma.

WebMvcConfigurer ile Global CORS Yapılandırması

Her controller'a `@CrossOrigin` eklemek yerine, tek bir yerden tüm `/api/**` için CORS tanımlamak da mümkün:

```
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`, "WebMvcConfigurer: Interceptor'ı Kaydetmek" bölümündeki `addInterceptors` ile aynı `WebMvcConfigurer` arayüzünün başka bir metodu -- ikisi de aynı `@Configuration` sınıfında bir arada bulunabilir. Bir URL deseni birden fazla `CorsRegistration` 'la eşleşirse (biri global, biri `@CrossOrigin` ile) Spring bunları birleştirmeye çalışır, ama pratikte karışıklığı önlemek için genelde **ya global ya da annotation tabanlı** bir yaklaşım seçilir, ikisi birden değil.

Multipart File Upload: @RequestParam ile MultipartFile Almak

Dosya yükleme, `@RequestBody` 'nin (Request ve Response Handling dersi) tek bir JSON gövdeyi okumasından farklı bir mekanizma kullanır:

```
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`, isteği adlandırılmış **parçalara** ayırır -- her parça ayrı bir form alanı ya da dosya olabilir. `MultipartFile`, Path Variable'lar ve Request Parametreleri dersindeki "@RequestParam: Query String'den Değer Okumak" bölümündeki gibi `@RequestParam`'la bağlanır, ama okuduğu şey bir query parametresi değil, isteğin bir parçası -- `getOriginalFilename()`, `getSize()`, `getBytes()` gibi metotlarla o parçaya erişilir.

Multipart Yapılandırması ve Boyut Sınırları

`spring.servlet.multipart.max-file-size` / `max-request-size`, Spring'in handler'a hiç ulaşmadan reddedeceği bir üst sınır tanımlar:

```
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
    }
}
```

Sınır aşıldığında fırlatılan `MaxUploadSizeExceededException`, Validation & Exception Handling dersindeki "Controller-Seviyesinde Hata Yakalama: @ExceptionHandler" ve "Global Hata Yönetimi: @RestControllerAdvice" bölümlerinde gördüğümüz mekanizmayla aynı şekilde yakalanır; "ProblemDetail: RFC 7807 ile Standart Hata Gövdesi" bölümündeki gibi standart bir hata gövdesi döndürmek, ham bir stack trace'i istemciye sızdırmaktan çok daha iyi bir davranış.

Best Practices

- **`Filter`'ı yalnızca gerçekten her isteği (statik dosyalar dahil) görmen gerektiğinde kullan, aksi halde `HandlerInterceptor`'ı tercih et** -- ikincisi Spring'in kendi mekanizmalarına (Model, exception handling) daha yakın çalışır (bkz. Filter vs Interceptor: İkisi de "Araya Girer" ama Nerede?).

- **Temizlik/loglama kodunu `postHandle` değil `afterCompletion`'a koy** -- yalnızca `afterCompletion` handler bir exception fırlatsa bile çalışır (bkz. "HandlerInterceptor Arayüzü: preHandle, postHandle, afterCompletion").
- **CORS'u ya global (`WebMvcConfigurer.addCorsMappings`) ya da annotation tabanlı (`@CrossOrigin`) yönet, ikisini karıştırma** -- karışık kullanım, hangi kuralın hangi endpoint'e uygulandığını takip etmeyi zorlaştırır (bkz. "WebMvcConfigurer ile Global CORS Yapılandırması").
- **Multipart boyut sınırlarını her zaman açıkça yapılandır** -- varsayılan sınırlar (Spring Boot'ta 1MB) çoğu gerçek dosya yükleme senaryosu için ya çok düşük ya da hiç düşünülmeden bırakılmış olabilir; her iki durumda da bilinçli bir karar olmalı (bkz. "Multipart Yapılandırması ve Boyut Sınırları").

Yaygın Hatalar

1. `Filter` ile `HandlerInterceptor`'ı birbirinin yerine geçebilir sanmak. Bir `Filter`, Spring'in `Model` / `HandlerMethod` gibi kavramlarına erişemez -- yalnızca ham `ServletRequest` / `ServletResponse` görür; bir handler'ın hangi controller'a eşleştiğini bilmesi gerekiyorsa doğru araç `HandlerInterceptor`'dır (bkz. Filter vs Interceptor: İkisi de "Araya Girer" ama Nerede?).

2. `preHandle`'da `false` dönüp yanıt kodunu ayarlamayı unutmak. Zincir durur ama istemci hâlâ varsayılan `200 OK` alır -- `false` dönmeden önce `response.setStatus(...)` çağırmak gerekir (bkz. "preHandle'da İsteği Durdurmak: Basit Bir Auth/Logging Örneği").

3. Birden fazla interceptor'da `postHandle`'ın da kayıt sırasıyla çalıştığını sanmak. `preHandle` ileri sırada, `postHandle` / `afterCompletion` ise ters sırada çalışır -- bu farkı unutmak, bir interceptor'ın diğerinin state'ine yanlış zamanda erişmesine yol açabilir (bkz. "Çoklu Interceptor: Sıralama ve Zincirleme").

4. CORS hatasını sunucu tarafında bir hata sanıp sunucu loglarında aramak. Tarayıcı, preflight başarısız olduğunda gerçek isteği hiç göndermez -- sunucu logunda hiçbir şey görünmeyebilir; hata yalnızca tarayıcının geliştirici konsolunda görünür (bkz. "CORS Nedir? Same-Origin Policy ve Preflight Request").

5. `@CrossOrigin`'i yalnızca `@RestController` sınıfına ekleyip metotların kendi `@CrossOrigin`'ini unutmak. Sınıf seviyesindeki `@CrossOrigin`, o sınıftaki tüm metotlara varsayılan olarak uygulanır, ama bir metot kendi `@CrossOrigin`'ini tanımlarsa sınıf seviyesindekini **tamamen** geçersiz kılar, birleştirmez -- bu, beklenmedik şekilde bazı endpoint'lerin CORS izinlerini kaybetmesine yol açabilir (bkz. "@CrossOrigin: Controller /Metot Seviyesinde CORS").

6. Multipart boyut sınırını yalnızca `max-file-size` ile ayarlayıp `max-request-size`'ı unutmak. Birden fazla dosya içeren bir istekte her dosya tek başına sınırın altında kalabilir ama toplamı `max-request-size`'ı aşabilir -- ikisi ayrı sınırlar, ikisi de ayrı ayrı yapılandırılmalı (bkz. "Multipart Yapılandırması ve Boyut Sınırları").

Özet, Cheat Sheet ve Terimler Sözlüğü

`Filter` ve `HandlerInterceptor`, bir isteğin etrafına kesişen davranış eklemenin iki farklı katmandaki yolu; `WebMvcConfigurer` ikisinin de (interceptor kaydı, CORS) yapılandırıldığı merkezi nokta; CORS ve multipart ise gerçek dünya uygulamalarının sıkça karşılaştığı iki somut senaryo. Öne çıkan noktalar:

- `Filter`: Servlet API seviyesinde, **tüm** istekleri görür (DispatcherServlet'in dışında)
- `HandlerInterceptor`: Spring MVC seviyesinde, yalnızca eşleşen istekleri görür (`preHandle` / `postHandle` / `afterCompletion`)
- `preHandle` `false` dönerse zincir durur -- handler ve `postHandle` hiç çalışmaz
- `afterCompletion`, exception olsa bile her zaman çalışır -- temizlik/loglama için en güvenilir nokta
- Çoklu interceptor: `preHandle` ileri sırada, `postHandle` / `afterCompletion` ters sırada
- `WebMvcConfigurer.addInterceptors` / `addCorsMappings`: interceptor kaydı ve global CORS için genişletme noktaları
- CORS: same-origin policy'nin sunucu tarafından gevşetilmesi; preflight, `OPTIONS` ile önceden sorulan bir izin sorusu
- `@CrossOrigin`: controller/metot seviyesinde CORS, `WebMvcConfigurer`'a alternatif
- `MultipartFile`: `multipart/form-data` isteğinin bir parçasını temsil eden, `@RequestParam` ile bağlanan arayüz
- `max-file-size` / `max-request-size`: multipart yükleme için iki ayrı boyut sınırı

Hızlı referans:

```
@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");
    }
}
```

Terimler Sözlüğü

`Filter` — Servlet API'nin bir parçası; container'ın DispatcherServlet'e ulaşmadan önce her isteği geçirdiği, kesişen davranış için kullanılan arayüz.

`HandlerInterceptor` — Spring MVC'ye özel, yalnızca bir handler'a eşleşen istekleri saran, `preHandle` / `postHandle` / `afterCompletion` callback'lerine sahip arayüz.

`WebMvcConfigurer` — Interceptor kaydı ve CORS gibi Spring MVC yapılandırmalarının yapıldığı, `@Configuration` sınıflarının implement ettiği genişletme noktası.

Same-origin policy — Tarayıcıların, bir sayfanın farklı bir origin'den veri okumasını varsayılan olarak engelleyen güvenlik kuralı.

CORS (Cross-Origin Resource Sharing) — Sunucunun, belirli origin'lere same-origin policy'yi gevşeterek izin vermesini sağlayan HTTP header mekanizması.

Preflight request — Tarayıcının, "basit olmayan" bir isteği göndermeden önce `OPTIONS` metoduyla sunucuya izin sorduğu ön istek.

`@CrossOrigin` — CORS izinlerini controller sınıfı ya da metodu seviyesinde tanımlayan annotation.

`MultipartFile` — Bir `multipart/form-data` isteğindeki tek bir dosya parçasını temsil eden Spring arayüzü.

`MaxUploadSizeExceededException` — Yapılandırılmış boyut sınırı aşıldığında Spring'in fırlattığı, `@ExceptionHandler` ile yakalanabilen exception.

Ek: Mini Proje — İstek Süresini Loglayan Bir Interceptor

Bu dersin `HandlerInterceptor` mekaniğini gerçekçi bir senaryoda birleştiriyoruz: `preHandle` 'da bir zamanlayıcı başlatıp `afterCompletion`'da (handler başarılı olsa da olmasa da) geçen süreyi loglayan bir interceptor:

```
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
    }
}
```

`afterCompletion`'ın kullanılması bilinçli bir seçim -- "HandlerInterceptor Arayüzü: preHandle, postHandle, afterCompletion" bölümünde gördüğümüz gibi, bu callback handler bir

exception fırlatsa bile çalışır, yani yavaş **ve** başarısız olan bir istek de doğru şekilde loglanır. `RequestLoggingInterceptorDemo`, hem başarılı hem başarısız bir isteği simüle ederek ikisinin de loglandığını gösteriyor.

Ek: Mini Proje — CORS Destekli Dosya Yükleme Endpoint'i

Son mini proje, bu dersin üç konusunu (`@CrossOrigin`, `MultipartFile`, boyut sınırı aşıldığında `@ExceptionHandler`) tek bir endpoint'te birleştiriyor:

```
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`, farklı bir origin'den (örneğin ayrı bir frontend uygulamasından) çağrılabilmesi için `@CrossOrigin` taşıyor, `MultipartFile` parametresiyle dosyayı alıyor, ve boyut sınırını aşan bir dosya için kendi `@ExceptionHandler`'ıyla (Validation & Exception Handling dersindeki `@RestControllerAdvice` yerine, bu kez controller'a **yerel** olarak) bir `ProblemDetail` döndürüyor. `FileUploadCorsDemo`, küçük bir dosyayla başarılı, büyük bir dosyayla da 413 sonucu üreten iki çağrıyı gösteriyor.