http://www.freebuf.com/articles/web/18493.html

CORS定义了两种跨域请求，简单跨域请求和非简单跨域请求。当一个跨域请求发送简单跨域请求包括：请求方法为HEAD，GET，POST;请求头只有4个字段，Accept，Accept-Language，Content-Language，Last-Event-ID;如果设置了Content-Type，则其值只能是application/x-www-form-urlencoded,multipart/form-data,text/plain。说起来比较别扭，简单的意思就是设置了一个白名单，符合这个条件的才是简单请求。其他不符合的都是非简单请求。

之所以有这个分类是因为浏览器对简单请求和非简单请求的处理机制是不一样的。当我们需要发送一个跨域请求的时候，浏览器会首先检查这个请求，如果它符合上面所述的简单跨域请求，浏览器就会立刻发送这个请求。如果浏览器检查之后发现这是一个非简单请求，比如请求头含有X-Forwarded-For字段。这时候浏览器不会马上发送这个请求，而是有一个preflight，跟服务器验证的过程。浏览器先发送一个options方法的预检请求。下图是一个示例。如果预检通过，则发送这个请求，否则就不拒绝发送这个跨域请求。

下面详细分析一下实现安全跨域请求的控制方式。先看一下非简单请求的预检过程。浏览器先发送一个options方法的请求。带有如下字段：

Origin: 普通的HTTP请求也会带有，在CORS中专门作为Origin信息供后端比对,表明来源域。Access-Control-Request-Method: 接下来请求的方法，例如PUT, DELETE等等Access-Control-Request-Headers: 自定义的头部，所有用setRequestHeader方法设置的头部都将会以逗号隔开的形式包含在这个头中

然后如果服务器配置了cors，会返回对应对的字段，具体字段含义在返回结果是一并解释。

Access-Control-Allow-Origin:

Access-Control-Allow-Methods:

Access-Control-Allow-Headers:

然后浏览器再根据服务器的返回值判断是否发送非简单请求。简单请求前面讲过是直接发送，只是多加一个origin字段表明跨域请求的来源。然后服务器处理完请求之后，会再返回结果中加上如下控制字段：

Access-Control-Allow-Origin: 允许跨域访问的域，可以是一个域的列表，也可以是通配符"\*"。这里要注意Origin规则只对域名有效，并不会对子目录有效。即http://foo.example/subdir/ 是无效的。但是不同子域名需要分开设置，这里的规则可以参照同源策略

Access-Control-Allow-Credentials: 是否允许请求带有验证信息，这部分将会在下面详细解释Access-Control-Expose-Headers: 允许脚本访问的返回头，请求成功后，脚本可以在XMLHttpRequest中访问这些头的信息(貌似webkit没有实现这个)

Access-Control-Max-Age: 缓存此次请求的秒数。在这个时间范围内，所有同类型的请求都将不再发送预检请求而是直接使用此次返回的头作为判断依据，非常有用，大幅优化请求次数

Access-Control-Allow-Methods: 允许使用的请求方法，以逗号隔开

Access-Control-Allow-Headers: 允许自定义的头部，以逗号隔开，大小写不敏感

然后浏览器通过返回结果的这些控制字段来决定是将结果开放给客户端脚本读取还是屏蔽掉。如果服务器没有配置cors，返回结果没有控制字段，浏览器会屏蔽脚本对返回信息的读取。

****安全隐患****

大家注意这个流程。服务器接收到跨域请求的时候，并没有先验证，而是先处理了请求。所以从某种程度上来说。在支持cors的浏览器上实现跨域的写资源，打破了传统同源策略下不能跨域读写资源。

再一个就是如果程序猿偷懒将Access-Control-Allow-Origin设置为允许来自所有域的跨域请求。那么cors的安全机制几乎就无效了。不过先别高兴的太早。其实这里在设计的时候有一个很好的限制。xmlhttprequest发送的请求需要使用“withCredentials”来带上cookie，如果一个目标域设置成了允许任意域的跨域请求，这个请求又带着cookie的话，这个请求是不合法的。（就是如果需要实现带cookie的跨域请求，需要明确的配置允许来源的域，使用任意域的配置是不合法的）浏览器会屏蔽掉返回的结果。javascript就没法获取返回的数据了。这是cors模型最后一道防线。假如没有这个限制的话，那么javascript就可以获取返回数据中的csrf token，以及各种敏感数据。这个限制极大的降低了cors的风险。

# CORS support in Spring Framework

<https://spring.io/blog/2015/06/08/cors-support-in-spring-framework>

For security reasons, browsers prohibit AJAX calls to resources residing outside the current origin. For example, as you’re checking your bank account in one tab, you could have the evil.com website in another tab. The scripts from evil.com shouldn’t be able to make AJAX requests to your bank API (withdrawing money from your account!) using your credentials.

[Cross-origin resource sharing](https://spring.io/understanding/CORS) (CORS) is a [W3C specification](http://www.w3.org/TR/cors/) implemented by [most browsers](http://caniuse.com/" \l "feat=cors)that allows you to specify in a flexible way what kind of cross domain requests are authorized, instead of using some less secured and less powerful hacks like IFrame or JSONP.

[Spring Framework 4.2 GA](https://spring.io/blog/2015/07/31/spring-framework-4-2-goes-ga) provides [first class support for CORS](https://jira.spring.io/browse/SPR-9278) out-of-the-box, giving you an easier and more powerful way to configure it than typical [filter based](http://software.dzhuvinov.com/cors-filter.html) solutions.

Spring MVC provides high-level configuration facilities, described bellow.

## **Controller method CORS configuration**

You can add to your @RequestMapping annotated handler method a [@CrossOrigin](http://docs.spring.io/spring-framework/docs/4.2.x/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html) annotation in order to enable CORS on it (by default @CrossOrigin allows all origins and the HTTP methods specified in the @RequestMapping annotation):

@RestController@RequestMapping("/account")public class AccountController {

@CrossOrigin

@RequestMapping("/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

}

@RequestMapping(method = RequestMethod.DELETE, value = "/{id}")

public void remove(@PathVariable Long id) {

// ...

}}

It is also possible to enable CORS for the whole controller:

@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)@RestController@RequestMapping("/account")public class AccountController {

@RequestMapping("/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

}

@RequestMapping(method = RequestMethod.DELETE, value = "/{id}")

public void remove(@PathVariable Long id) {

// ...

}}

In this example CORS support is enabled for both retrieve() and remove() handler methods, and you can also see how you can customize the CORS configuration using@CrossOrigin attributes.

You can even use both controller and method level CORS configurations, Spring will then combine both annotation attributes to create a merged CORS configuration.

@CrossOrigin(maxAge = 3600)@RestController@RequestMapping("/account")public class AccountController {

@CrossOrigin(origins = "http://domain2.com")

@RequestMapping("/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

}

@RequestMapping(method = RequestMethod.DELETE, value = "/{id}")

public void remove(@PathVariable Long id) {

// ...

}}

## **Global CORS configuration**

In addition to fine-grained, annotation-based configuration you’ll probably want to define some global CORS configuration as well. This is similar to using filters but can be declared withing Spring MVC and combined with fine-grained @CrossOrigin configuration. By default all origins and GET, HEAD and POST methods are allowed.

### **JavaConfig**

Enabling CORS for the whole application is as simple as:

@Configuration@EnableWebMvcpublic class WebConfig extends WebMvcConfigurerAdapter {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/\*\*");

}}

You can easily change any properties, as well as only apply this CORS configuration to a specific path pattern:

@Configuration@EnableWebMvcpublic class WebConfig extends WebMvcConfigurerAdapter {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/api/\*\*")

.allowedOrigins("http://domain2.com")

.allowedMethods("PUT", "DELETE")

.allowedHeaders("header1", "header2", "header3")

.exposedHeaders("header1", "header2")

.allowCredentials(false).maxAge(3600);

}}

### **XML namespace**

It is also possible to configure CORS with the [mvc XML namespace](https://jira.spring.io/browse/SPR-13046).

This minimal XML configuration enable CORS on /\*\* path pattern with the same default properties than the JavaConfig one:

<mvc:cors>

<mvc:mapping path="/\*\*" /></mvc:cors>

It is also possible to declare several CORS mappings with customized properties:

<mvc:cors>

<mvc:mapping path="/api/\*\*"

allowed-origins="http://domain1.com, http://domain2.com"

allowed-methods="GET, PUT"

allowed-headers="header1, header2, header3"

exposed-headers="header1, header2" allow-credentials="false"

max-age="123" />

<mvc:mapping path="/resources/\*\*"

allowed-origins="http://domain1.com" />

</mvc:cors>

## **How does it work?**

CORS requests ([including preflight ones with an OPTIONS method](https://github.com/spring-projects/spring-framework/blob/master/spring-webmvc/src/main/java/org/springframework/web/servlet/FrameworkServlet.java" \l "L906)) are automatically dispatched to the various HandlerMappings registered. They handle CORS preflight requests and intercept CORS simple and actual requests thanks to a [CorsProcessor](http://docs.spring.io/spring/docs/4.2.x/javadoc-api/org/springframework/web/cors/CorsProcessor.html) implementation ([DefaultCorsProcessor](https://github.com/spring-projects/spring-framework/blob/master/spring-web/src/main/java/org/springframework/web/cors/DefaultCorsProcessor.java) by default) in order to add the relevant CORS response headers (likeAccess-Control-Allow-Origin). [CorsConfiguration](http://docs.spring.io/spring/docs/4.2.x/javadoc-api/org/springframework/web/cors/CorsConfiguration.html) allows you to specify how the CORS requests should be processed: allowed origins, headers, methods, etc. It can be provided in various ways:

* [AbstractHandlerMapping#setCorsConfiguration()](http://docs.spring.io/spring/docs/4.2.x/javadoc-api/org/springframework/web/servlet/handler/AbstractHandlerMapping.html" \l "setCorsConfiguration-java.util.Map-) allows to specify a Map with several[CorsConfiguration](http://docs.spring.io/spring/docs/4.2.x/javadoc-api/org/springframework/web/cors/CorsConfiguration.html) mapped on path patterns like /api/\*\*
* Subclasses can provide their own CorsConfiguration by overridingAbstractHandlerMapping#getCorsConfiguration(Object, HttpServletRequest) method
* Handlers can implement [CorsConfigurationSource](http://docs.spring.io/spring/docs/4.2.x/javadoc-api/org/springframework/web/cors/CorsConfigurationSource.html) interface (like[ResourceHttpRequestHandler](https://github.com/spring-projects/spring-framework/blob/master/spring-webmvc/src/main/java/org/springframework/web/servlet/resource/ResourceHttpRequestHandler.java) now does) in order to provide a [CorsConfiguration](http://docs.spring.io/spring/docs/4.2.x/javadoc-api/org/springframework/web/cors/CorsConfiguration.html) for each request.

## **Spring Boot integration**

CORS support will be available in the upcoming Spring Boot 1.3 release, and is already available in the 1.3.0.BUILD-SNAPSHOT builds.

Using controller method CORS configuration with @CrossOrigin annotations in your Spring Boot application does not require any specific configuration.

Global CORS configuration can be defined by registering a WebMvcConfigurer bean with a customized addCorsMappings(CorsRegistry) method:

@Configurationpublic class MyConfiguration {

@Bean

public WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurerAdapter() {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/api/\*\*");

}

};

}}

## **Filter based CORS support**

In order to support CORS with filter-based security framework like Spring Security, or with other projects that does not support natively CORS yet like [Spring Data REST](https://jira.spring.io/browse/DATAREST-573), we also provide a [CorsFilter](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/filter/CorsFilter.html). In that case, instead of using @CrossOrigin orWebMvcConfigurer#addCorsMappings(CorsRegistry), you can for example declare the filter as following in your Spring Boot application:

@Configurationpublic class MyConfiguration {

@Bean

public FilterRegistrationBean corsFilter() {

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

CorsConfiguration config = new CorsConfiguration();

config.setAllowCredentials(true);

config.addAllowedOrigin("http://domain1.com");

config.addAllowedHeader("\*");

config.addAllowedMethod("\*");

source.registerCorsConfiguration("/\*\*", config);

FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));

bean.setOrder(0);

return bean;

}}