**REST API Security Guidelines**

**authentication** is the process of ascertaining that “**is user really someone who he claims to be**“. In technical terms, it’s process of login to system through username/password or any similar mechanisms e.g. finger print scanning, security token, security questions or SAML token received from SSO login. There must be something which can identify the user from others.

Once user is inside system, **authorization** refers to rules that determine **“what user is allowed to do”**and what he is not e.g. a normal user can post a message in any public group, but users only with editor role will be able to delete something. Authorization is often seen as both the introductory setting up of permissions by a system administrator and the checking of the permission values that have already been set up when a user is getting access to the system.

It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (**for example, a 403 Forbidden**).

#### BASIC Authentication

It’s simplest of all techniques and probably most used as well. You use login/password forms – it’s basic authentication only. You input your username and password and submit the form to server, and application identify you as a user – you are allowed to use the system – else you get error.

The main problem with this security implementation is that credentials are propagated in a plain way from the client to the server. Credentials are merely encoded with Base64 in transit, but not encrypted or hashed in any way. This way, any sniffer could read the sent packages over the network.

HTTPS is, therefore, typically preferred over or used in conjunction with Basic Authentication which makes the conversation with the web server entirely encrypted. The best part is that nobody can even guess from the outside that Basic Auth is taking place.

#### DIGEST Authentication

This authentication method makes use of a hashing algorithms to encrypt the password (called **password hash**) entered by the user before sending it to the server. This, obviously, makes it much safer than the basic authentication method, in which the user’s password travels in plain text that can be easily read by whoever intercepts it.

**Read More:** [Generate Encrypted Passwords](https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/)

There are many such hashing algorithms in java also, which can prove really effective for password security such as MD5, SHA, BCrypt, SCrypt and PBKDF2WithHmacSHA1 algorithms.

Please remember that once this password hash is generated and stored in database, you can not convert it back to original password. Each time user login into application, you have to regenerate password hash again, and match with hash stored in database. So, if user forgot his/her password, you will have to send him a temporary password and ask him to change it with his new password. Well, it’s common trend now-a-days.

#### Client CERT Authentication

This is a mechanism in which a trust agreement is established between the server and the client through certificates. They must be signed by an agency established to ensure that the certificate presented for authentication is legitimate, which is known as CA.

Using this technique, when the client attempts to access a protected resource, instead of providing a username or password, it presents the certificate to the server. The certificate contains the user information for authentication including security credentials, besides a unique private-public key pair. The server then determines if the user is legitimate through the CA. Additionally, it must verify whether the user has access to the resource. This mechanism must use HTTPS as the communication protocol as we don’t have a secure channel to prevent anyone from stealing the client’s identity.

#### OAUTH2 API Keys

If you have ever developed applications which interact other with other applications over cloud e.g. facebook integration or twitter authentication etc. then you have already used this. They require you to provide API key and API secret to rightly identify you. These API key and secret are some random encoded string which is impossible to guess.

To understand how it works, let’s assume you are using a Flickr (photo sharing application) and want to post some of your photos using it’s REST API. You build the request as documented in Flickr docs, then send it.

## REST API Security Implementations

Apart from above concepts, you will usually need to secure your RESTful APIs in your company using below methods.

#### 3.1. JAX-RS SecurityContext instance

The javax.ws.rs.core.SecurityContext interface provides access to security-related information for a request and is very similar to javax.servlet.http.HttpServletRequest.

You access the SecurityContext by injecting an instance into a class field, setter method, or method parameter using the javax.ws.rs.core.Context annotation e.g. in below code sc.isUserInRole() is used to check authorization for user.

|  |
| --- |
| @GET  @Produces("text/plain;charset=UTF-8")  @Path("/hello")  public String sayHello(@Context SecurityContext sc) {          if (sc.isUserInRole("admin"))              return "Hello World!";          throw new SecurityException("User is unauthorized.");  } |

#### 3.2. JAR-RS annotations for method level authorization

This technique is widely used in enterprise application and used to verify roles and responsibities of an authenticated used – for any certain operation. JAX-RS provides below annotations for this purpose.

* [@PermitAll](https://docs.oracle.com/javaee/6/api/javax/annotation/security/PermitAll.html)
* [@DenyAll](https://docs.oracle.com/javaee/6/api/javax/annotation/security/DenyAll.html)
* [@RolesAllowed](https://docs.oracle.com/javaee/6/api/javax/annotation/security/RolesAllowed.html)

An example use of annotation can be:

|  |
| --- |
| @RolesAllowed("ADMIN")  @PUT  @Path("/users/{id}")  public Response updateUserById(@PathParam("id") int id)  {      //Update the User resource      UserDatabase.updateUser(id);      return Response.status(200).build();  } |

## REST API Security Best Practices

Let’s note down some important points while designing security for your RESTful web services.

1. Use only HTTPS protocol so that your whole communication is always encrypted.
2. Never send auth credentials or API keys as query param. They appear in URL and can be logged or tracked easily.
3. Use hardest encryption level always. It will help in having more confidence.
4. For resources exposed by RESTful web services, it’s important to make sure any PUT, POST, and DELETE request is protected from Cross Site Request Forgery.
5. Always validate the input data asap it is received in server method. Use only primitive data as input parameter as much as possible.
6. Rely on framework provided validation features as they are tested by large community already.

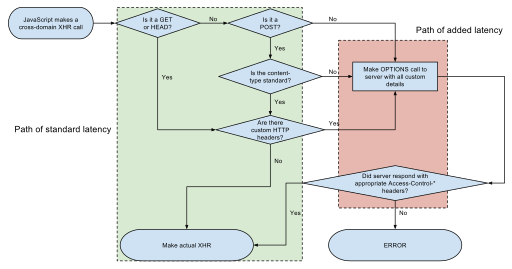
<https://howtodoinjava.com/resteasy/jax-rs-resteasy-basic-authentication-and-authorization-tutorial/>

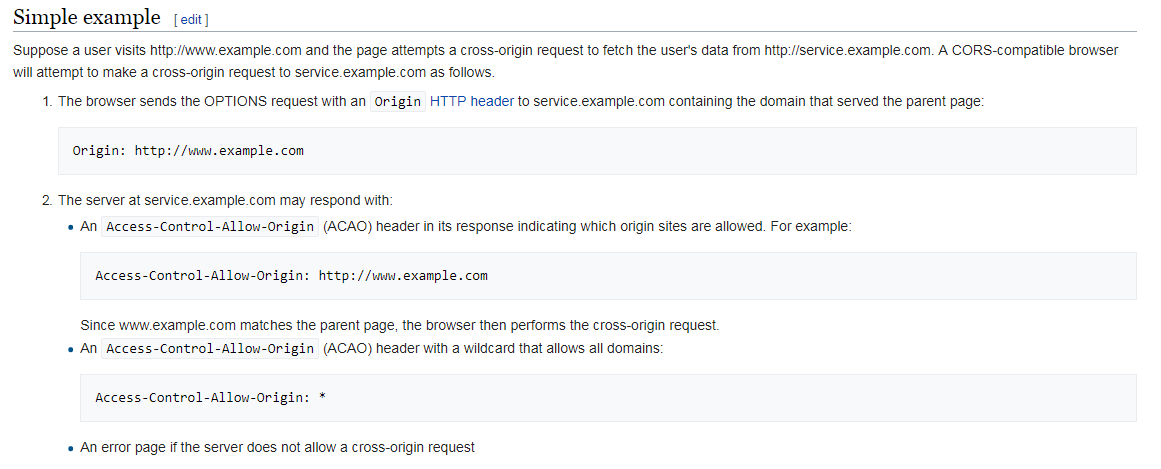


I faced CORS issue in my application after allowing origin domain then only it will be success else failure

GOOD POST to understand this concept

<https://en.wikipedia.org/wiki/Cross-origin_resource_sharing>





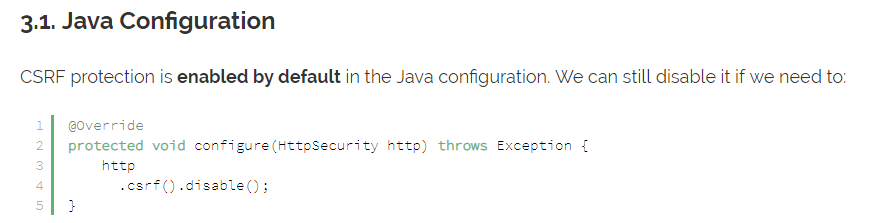
<https://www.baeldung.com/spring-boot-security-autoconfiguration>

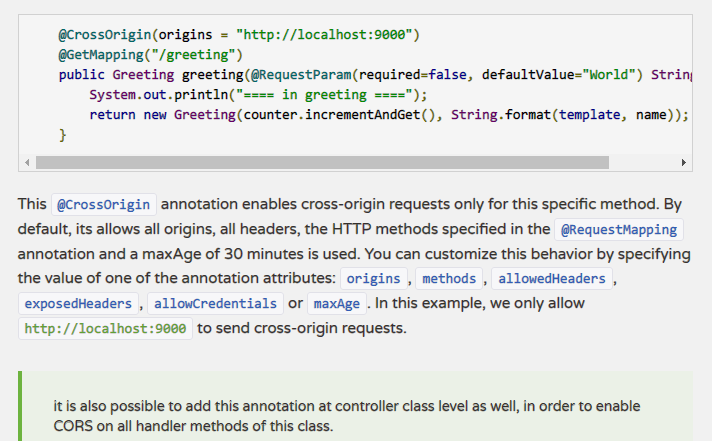
<https://spring.io/guides/gs/rest-service-cors/>

# A Guide to CSRF Protection in Spring Security

# <https://www.baeldung.com/spring-security-csrf> Good one

Cross-Site Request Forgery CSRF attacks and how to prevent them using Spring Security.





### Protect Privileged Actions and Sensitive Resource Collections

Not every user has a right to every web service. This is vital, as you don't want administrative web services to be misused:

https://example.com/admin/exportAllData

The session token or API key should be sent along as a cookie or body parameter to ensure that privileged collections or actions are properly protected from unauthorized use.

### Protect Against Cross-site Request Forgery

For resources exposed by RESTful web services, it's important to make sure any PUT, POST, and DELETE request is protected from Cross Site Request Forgery. Typically one would use a token-based approach.

CSRF is easily achieved even using random tokens if any XSS exists within your application, so please make sure you understand how to prevent XSS.

### Insecure Direct Object References

It may seem obvious, but if you had a bank account REST web service, you'd have to make sure there is adequate checking of primary and foreign keys:

In this case, it would be possible to transfer money from any account to any other account, which is clearly absurd. Not even a random token makes this safe.

This is essentially a data-contextual access control enforcement need. A URL or even a POSTed form should NEVER contain an access control "key" or similar that provides automatic verification. A data contextual check needs to be done, server side, with each request.

## 2/5 - Input Validation

Everything you know about input validation applies to RESTful web services, but add 10% because automated tools can easily fuzz your interfaces for hours on end at high velocity. So:

Assist the user > Reject input > Sanitize (filtering) > No input validation

Assisting the user makes the most sense, as the most common scenario is "problem exists between keyboard and chair" (PEBKAC).

Help the user input high quality data into your web services, such as ensuring a Zip code makes sense for the supplied address, or the date makes sense. If not, reject that input. If they continue on, or it's a text field or some other difficult to validate field, input sanitization is a losing proposition but still better than XSS or SQL injection.

If you're already reduced to sanitization or no input validation, make sure output encoding is very strong for your application.

Log input validation failures, particularly if you assume that client-side code you wrote is going to call your web services.

The reality is that anyone can call your web services, so assume that someone who is performing hundreds of failed input validations per second is up to no good.

Consider rate limiting the API to a certain number of requests per hour or day to prevent abuse.

### URL Validations

Web applications/web services use input from HTTP requests (and occasionally files) to determine how to respond.

Attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields, to try to bypass the site’s security mechanisms.

Common names for common input tampering attacks include forced browsing, command insertion, cross-site scripting, buffer overflows, format string attacks, SQL injection, cookie poisoning, and hidden field manipulation.

### Secure Parsing

Use a secure parser for parsing the incoming messages. If you are using XML, make sure to use a parser that is not vulnerable to XXE and similar attacks.

### Strong Typing

It's difficult to perform most attacks if the only allowed values are true or false, or a number, or one of a small number of acceptable values. Strongly type incoming data as quickly as possible.

### Validate Incoming Content-types

When POSTing or PUTting new data, the client will specify the Content-Type (e.g. application/xml or application/json) of the incoming data.

**The server should never assume the Content-Type,**it should always check that the Content-Type header and the content are the same type. A lack of Content-Type header or an unexpected Content-Type header should result in the server rejecting the content with a **406 Not Acceptable response**.

### Validate Response Types

It is common for REST services to allow multiple response types (e.g. application/XML or application/JSON, and the client specifies the preferred order of response types by the Accept header in the request.

Do NOT simply copy the Accept header to the Content-type header of the response. Reject the request (ideally with a 406 Not Acceptable response) if the Accept header does not specifically contain one of the allowable types.

Because there are many MIME types for the typical response types, it's important to document for clients specifically which MIME types should be used.

### XML Input Validation

XML-based services must ensure that they are protected against common XML-based attacks by using secure XML-parsing.

This typically means protecting against XML External Entity attacks, XML-signature wrapping etc.

## 3/5 - Output Encoding

### Security Headers

To make sure the content of a given resources is interpreted correctly by the browser, the server should always send the Content-Type header with the correct Content-Type, and preferably the Content-Type header should include a charset.

The server should also send an X-Content-Type-Options: nosniff to make sure the browser does not try to detect a different Content-Type than what is actually sent (can lead to XSS).

Additionally, the client should send an X-Frame-Options: deny to protect against drag'n drop clickjacking attacks in older browsers.

### JSON Encoding

A key concern with JSON encoders is preventing arbitrary JavaScript remote code execution within the browser... or, if you're using node.js, on the server. It's vital that you use a proper JSON serializer to encode user-supplied data properly to prevent the execution of user-supplied input on the browser.

When inserting values into the browser DOM, strongly consider using .value/.innerText/.textContent rather than .innerHTML updates, as this protects against simple DOM XSS attacks.

### XML Encoding

XML should never be built by string concatenation. It should always be constructed using an XML serializer. This ensures that the XML content sent to the browser is parseable and does not contain XML injection. For more information, please see the Web Service Security Cheat Sheet.

## 4/5 - Cryptography

### Data in Transit

Unless the public information is completely read-only, the use of TLS should be mandated, particularly where credentials, updates, deletions, and any value transactions are performed. The overhead of TLS is negligible on modern hardware, with a minor latency increase that is more than compensated by safety for the end user.

Consider the use of mutually authenticated client-side certificates to provide additional protection for highly privileged web services.

### Data in Storage

Leading practices are recommended as per any web application when it comes to correctly handling stored sensitive or regulated data.

### Message Integrity

In addition to HTTPS/TLS, JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

JWT can not only be used to ensure the message integrity but also authentication of both message sender/receiver.

The JWT includes the digital signature hash value of the message body to ensure the message integrity during the transmission.

## 5/5 - HTTP Status Codes

HTTP defines status code. When design REST API, don't just use 200 for success or 404 for error.

Here are some guideline to consider for each REST API status return code. Proper error handle may help to validate the incoming requests and better identify the potential security risks.

* **200 OK** - Response to a successful REST API action. The HTTP method can be GET, POST, PUT, PATCH or DELETE.
* **400 Bad Request** - The request is malformed, such as message body format error.
* **401 Unauthorized** - Wrong or no authentication ID/password provided.
* **403 Forbidden** - It's used when the authentication succeeded but authenticated user doesn't have permission to the request resource.
* **404 Not Found** - When a non-existent resource is requested.
* **405 Method Not Allowed** - The error checking for unexpected HTTP method. For example, the RestAPI is expecting HTTP GET, but HTTP PUT is used.
* **429 Too Many Requests** - The error is used when there may be DOS attack detected or the request is rejected due to rate limiting

### 401 vs 403

**401 “Unauthorized”** really means Unauthenticated, “You need valid credentials for me to respond to this request”.

**403 “Forbidden”** really means Unauthorized, “I understood your credentials, but so sorry, you’re not allowed!”