

# Table of Contents

Extracting credentials from the request	2
Logout	5

The [authentication filter](#) uses the default authenticationManager bean, which in turn uses all the registered authentication providers. See the [Spring Security Core guide](#) for more information about how to define your own providers. Note that you can easily plug any Spring Security sub-plugin (like the LDAP one) to use a different authentication strategy.

If the authentication is successful, a token generator is used to generate a token, and a token storage implementation is used to store the token. Finally, the JSON response sent back to the client is rendered by a restAuthenticationTokenJsonRenderer bean. See the token rendering documentation for more details.



This authentication filter will only be applied to the above configured URL and can also be disabled, in case a different approach for token creation is followed. In the rest of the cases, the request will continue through the filter chain, reaching Spring Security Core filters. Bear in mind that, by default, Spring Security Core 2.x locks down all URL's unless a explicit security rule has been specified for each of them.

See [Spring Security Core documentation](#) for more information.

The following are the Config.groovy properties available:

*Table 1. Authentication configuration options*

Config key	Default value
grails.plugin.springsecurity.rest.login.active	true
grails.plugin.springsecurity.rest.login.endpointUrl	/api/login
grails.plugin.springsecurity.rest.login.failureStatus Code	401

## Extracting credentials from the request

The plugin supports 2 ways of extracting the username and password: using request parameters, and using a JSON payload. To align with the RESTful principles, JSON payload is the default behaviour.

### From a JSON request

*Table 2. JSON credentials extraction configuration properties*

Config key	Default value
<code>grails.plugin.springsecurity.rest.login.useJsonCredentials</code>	<code>true</code>
<code>grails.plugin.springsecurity.rest.login.usernamePropertyName</code>	<code>username</code>
<code>grails.plugin.springsecurity.rest.login.passwordPropertyName</code>	<code>password</code>

The default implementation expects a request like this:

*Listing 1. Example JSON authentication request*

```
{
  "username": "john.doe",
  "password": "dontTellAnybody"
}
```

If you use `usernamePropertyName` and `passwordPropertyName` properties mentioned above, your JSON request can look like:

*Listing 2. Custom JSON authentication request*

```
{
  "login": "john.doe",
  "pwd": "dontTellAnybody"
}
```

With the following config:

### *Listing 3. Custom JSON authentication configuration properties*

```
grails.plugin.springsecurity.rest.login.usernamePropertyName = 'login'  
grails.plugin.springsecurity.rest.login.passwordPropertyName = 'pwd'
```

If your JSON request format is different, you can plug your own implementation by defining a class which extends <http://alvarosanchez.github.io/grails-spring-security-rest/latest/docs/gapi/grails/plugin/springsecurity/rest/credentials/AbstractJsonPayloadCredentialsExtractor.html> [AbstractJsonPayloadCredentialsExtractor]. The default implementation looks like this:

### *Listing 4. DefaultJsonPayloadCredentialsExtractor*

```
@Slf4j  
class DefaultJsonPayloadCredentialsExtractor extends  
    AbstractJsonPayloadCredentialsExtractor {  
  
    String usernamePropertyName  
    String passwordPropertyName  
  
    UsernamePasswordAuthenticationToken extractCredentials(HttpServletRequest  
    httpRequest) {  
        def jsonBody = getJsonBody(httpServletRequest)  
  
        if (jsonBody) {  
            String username = jsonBody."${usernamePropertyName}"  
            String password = jsonBody."${passwordPropertyName}"  
  
            log.debug "Extracted credentials from JSON payload. Username:  
            ${username}, password: ${password?.size()?'[PROTECTED]':'[MISSING]'}"  
  
            new UsernamePasswordAuthenticationToken(username, password)  
        } else {  
            log.debug "No JSONbody sent in the request"  
            return null  
        }  
    }  
}
```

Once you are done, register it in `resources.groovy` with the name `credentialsExtractor`.

## From request parameters

Note that the name of the parameters can also be customised:

*Table 3. Parameter extraction configuration options*

Config key	Default value
grails.plugin.springsecurity.rest.login.useRequestParamsCredentials	false
grails.plugin.springsecurity.rest.login.usernamePropertyName	username
grails.plugin.springsecurity.rest.login.passwordPropertyName	password

## Logout

Logout is not possible when using JWT tokens (the default strategy), as no state is kept in the server. If you still want to have logout, you can provide your own implementation by creating a subclass of `JwtTokenStorageService` and overriding the methods `storeToken` and `removeToken`.



Then, register your implementation in `resources.groovy` as `tokenStorageService`.

However, a more rational approach would be just to remove the token from the client (eg, browser's local storage) and let the tokens expire (they will expire anyway, unlike with other storages like Memcached or Redis where they get refreshed on every access).

The `logout filter` exposes an endpoint for deleting tokens. It will read the token from an HTTP header. If found, will delete it from the storage, sending a 200 response. Otherwise, it will send a 404 response.

You can configure it in `Config.groovy` using this properties:

*Table 4. Logout configuration options*

Config key	Default value
<code>grails.plugin.springsecurity.rest.logout.endpointUrl</code>	<code>/api/logout</code>
<code>grails.plugin.springsecurity.rest.token.validation.headerName</code>	<code>X-Auth-Token</code>