Table of Contents

Why this token-based implementation? Can't I use HTTP basic authentication?

In theory you can. The only restriction to be truly stateless is to not use HTTP sessions at all. So if you go with basic authentication, you need to transfer the credentials back and forth every time.

Let's think about that. Keep in mind that your frontend is a pure HTML/Javascript application, consuming a REST API from the Grails side. So the first time, the Javascript application will make an API query and will receive a 401 response indicating that authentication is required. Then you present the user a form to enter credentials, you grab them, **encode** them with Base64 and in the next request, you send an HTTP header like `Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`.

Now remember you are doing RESTful application, so the session state is maintained in the client. That means that you would need to store that Base64 encoded string somewhere: cookies? HTML5 local storage? In any case, they are accessible using browser tools. And that's the point: there is a huge security risk because Base64 it's not encryption, just encoding. And it can be easily decoded.

You could argue that someone can access the token in the browser. Yes, but having the token will not allow him to obtain user's credentials. The tokens are just not decodable. And they can be revoked if necessary.

Fortunately for you, a token-based solution is not a magic idea that I only got; it's actually a specification: RFC 6750 - The OAuth 2.0 Authorization Framework: Bearer Token Usage.

Moreover, if you use tokens, you have the chance to implement expiration policies.

A couple of link with further explanations on the token-based flow:

- http://www.jamesward.com/2013/05/13/securing-single-page-apps-and-rest-services
- http://blog.brunoscopelliti.com/authentication-to-a-restful-web-service-in-an-angularjs-web-app

Why can't the API be secured with OAuth?

RFC 6749 - OAuth 2.0 specification does cover this scenario in what they call "public clients":

> Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

The OAuth 2.0 specification supports public clients with the implicit grant. This plugin supports that by default when you delegate the authentication to another OAuth provider. If it's you who are authenticating the users (via DB, LDAP, etc), the token-based flow of this plugin is *OAuth-ish*.

Why you didn't use any of the existing OAuth plugins? Why pac4j?

I'm aware of plugins like OAuth and Spring Security OAuth, but all of them rely on Spring Security Core's way of using HTTP sessions. So not acceptable.

I chose pac4j because:

1. They support major OAuth 2.0 providers out-of-the-box, whereas Scribe does not.
2. It's deadly simple and works just fine.

I'm also aware of a pac4j-spring-security module. See my previous response on HTTP sessions.

Dude, this is awesome. How can I compensate you?

I doubt you can :). You may try giving me free beers the next time you see me in a conference. Or you can just express your gratitude via Twitter.