One Minute Distraction

Experiments with Java, JavaEE, Web technologies and who knows what else

GO

← Building Your Own Presentation Server | Screencast: Creating a Hybrid Mobile Application →

Using OAuth For Your JavaEE Login

APRIL 29, 2014 19 COMMENTS

In my previous blog I describe how we could add additional security layer to Java EE's login process. The application server holds a set of valid username/password; users are authenticated against this set of credentials.

Managing username/password is hard work; what if you are only interested about the user but not necessarily in maintaining the user. If your security requirement is not that stringent, then I could let somebody that I trust validate that user. Once the user is validated, I can find out about the user. So you don't maintain any username/password database. As far as you are concern there is no user.

Enter OAuth; it is a standard at allows an application to access data that a user holds in a 3rd party application. In order to access the data, the user first have to authenticated into that 3rd party application; once we pass the authentication stage, then we are authorized to access the data. The really nice thing about OAuth is that you never have to reveal your username or password to the application.

A use case might be an application would like a user's valid email; one source of this might be from Google. So by using OAuth, the user can authorize Google to release this piece of information to the application. For more details on OAuth watch this video.

For the remainder of this blog, we will look at

1. how to use Google's OAuth2 provider for authentication and authorization; we'll look at what are the information/resources we can release and how to go about doing that
2. how to integrate/use Google's OAuth2 as our authentication mechanism for a web application.

Register and Configure Your Application

To start using Google as our OAuth2 provider, you'll need first do the following list of things

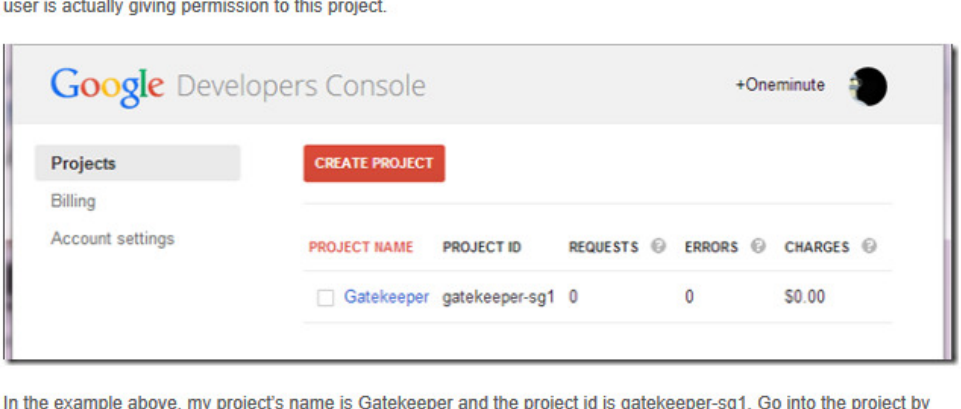
1. Create a project or service account (in Google)
2. Create a new client id
3. Select the information/resources that your application wishes to use
4. Optionally, configure the consent screen

Here is how everything hangs together:

1. An user Fred, attempts to access some restricted resources/page on the JavaEE application. The JavaEE application server determines that Fred need to first login.
2. The web application redirect Fred to Google's login page. Fred proceeds logins to his Google account; if Fred has already login to his account, then he will not need to relogin.
3. After Fred has successfully login, Google OAuth2 provider will display the consent page. The consent page provides details about the web application and more importantly what resources is it trying to access, eg Fred's email and Google+ id, his drive, etc.
4. If Fred consents, Google now redirects the browser back to our web application

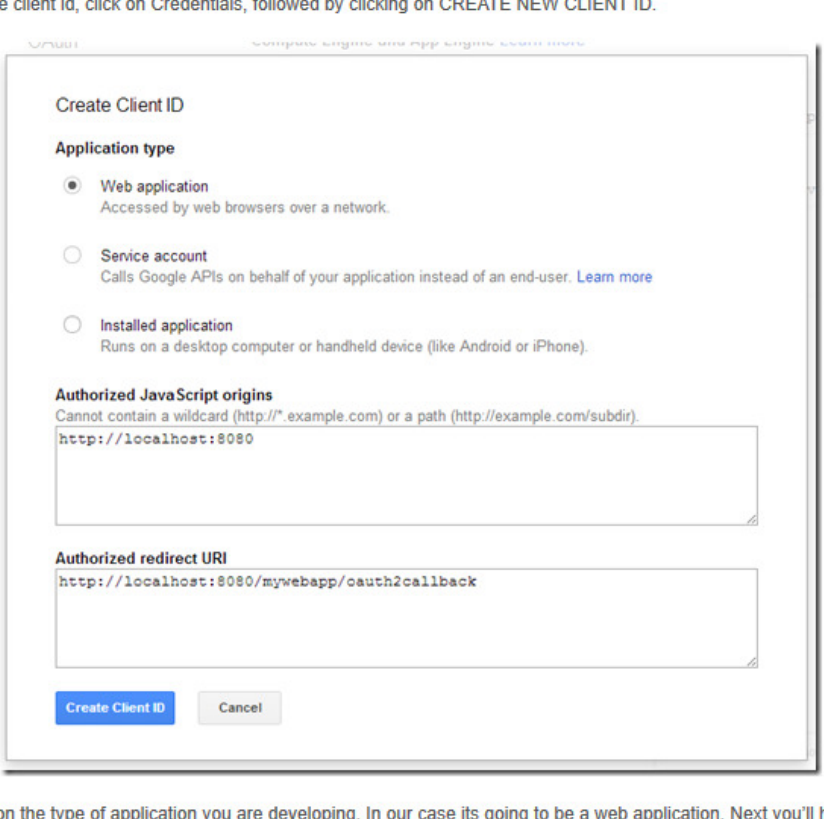
Now lets look at these steps in greater detail.

The first thing you'll need to do is to create a project using Developer Console. This project is our JavaEE's representation in Google's universe. When a user gives permission to our JavaEE application to access his email, the user is actually giving permission to this project.

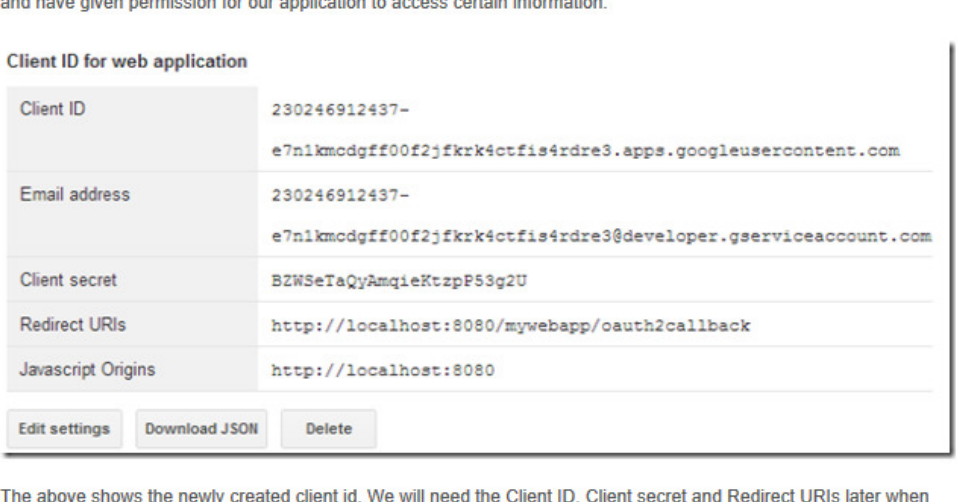


In the example above, my project's name is Gatekeeper and the project id is gatekeeper-sg1. Go into the project by clicking on the project's name. On the left hand side, expand 'API & auth' menu item and you should see APIs, Credentials and Consent Screen.

To create client id, click on Credentials, followed by clicking on CREATE NEW CLIENT ID.



Decide on the type of application you are developing. In our case its going to be a web application. Next you'll have to configure 2 pieces of additional information associated with Web Application. The Authorized JavaScript is your application's address in this case is the application's domain. Since I'm writing a demo, that will be localhost with the port number. The Authorized redirect URI is the URI that Google will redirect to after the user has successfully login and have given permission for our application to access certain information.



Client ID for web application

Client ID	230246912437-e7n1kmcdgff00f2jfkrk4ctfis4rdre3.apps.googleusercontent.com
Email address	230246912437-e7n1kmcdgff00f2jfkrk4ctfis4rdre3@developer.gserviceaccount.com
Client secret	BZWSeTaQyAmqieKtzpP53g2U
Redirect URIs	http://localhost:8080/mywebapp/oauth2callback
Javascript Origins	http://localhost:8080

Edit settings Download JSON Delete

The above shows the newly created client id. We will need the Client ID, Client secret and Redirect URIs later when we write our web application.

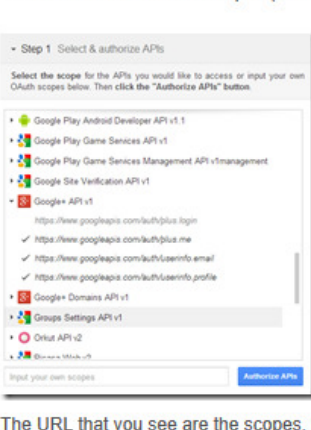
So the next thing we need to decide is what do we want to access from a user's Google account. Click API under APIs & auth. You will now see a list of all the resources available. For our case, we are only interested in basic information, so we're going to enable Google+ API. Enable all the API that the application will be needing. Note that most of these API have a daily limit (for free use). We'll look at how to use the Google+ API that we have enabled in the next section

One final, you can customize the consent screen by entering the web application's url, image, T&C, etc under the Consent screen menu.

Exploring the API

As I've mentioned earlier, Google provide lots of resource that we can access. In our case we are interested in a user's Google+ details. All of these resources are access using REST. Google provides a playground for you to experiment with these API. The playground allows you to

- Find out what are the resources available eg. under Google+ there are different resources like email, profile, login
- How to invoke them
- What is the structure of the returned JSON data
- What are the OAuth2 scopes (more on scopes later) namespaces



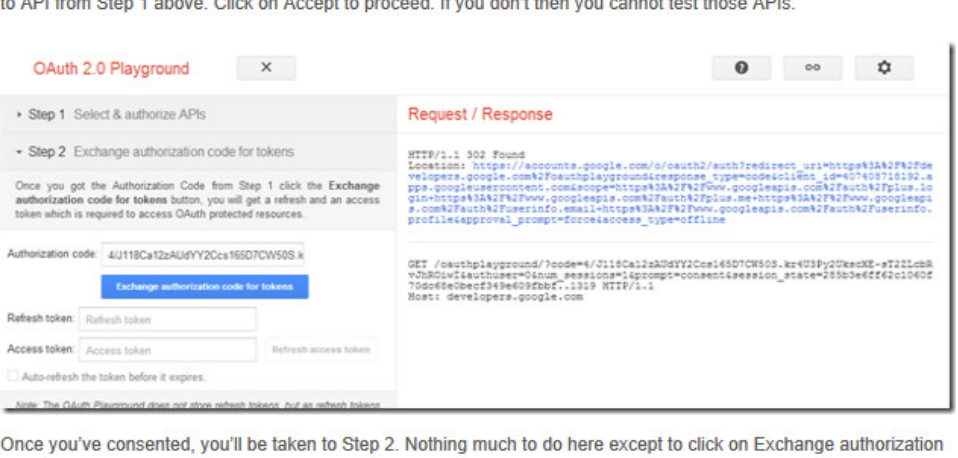
Assume we are going to get basic detail about a user from his Google+, so let see how we do this using playground.

Open up playground. Under Step 1, look for Google+ API v1 (don't forget to enable it in your application). Under that you will see list of information accessible. Select the information that the application will need to access; what we are saying here is that if a user eg Fred logins to our web application, then Fred will have to provide these information to the web application. The consent page will display these details. So check whatever details you want to access. As a best practice, ask for the minimum amount of information in order for your application to establish a trust relationship with the user.

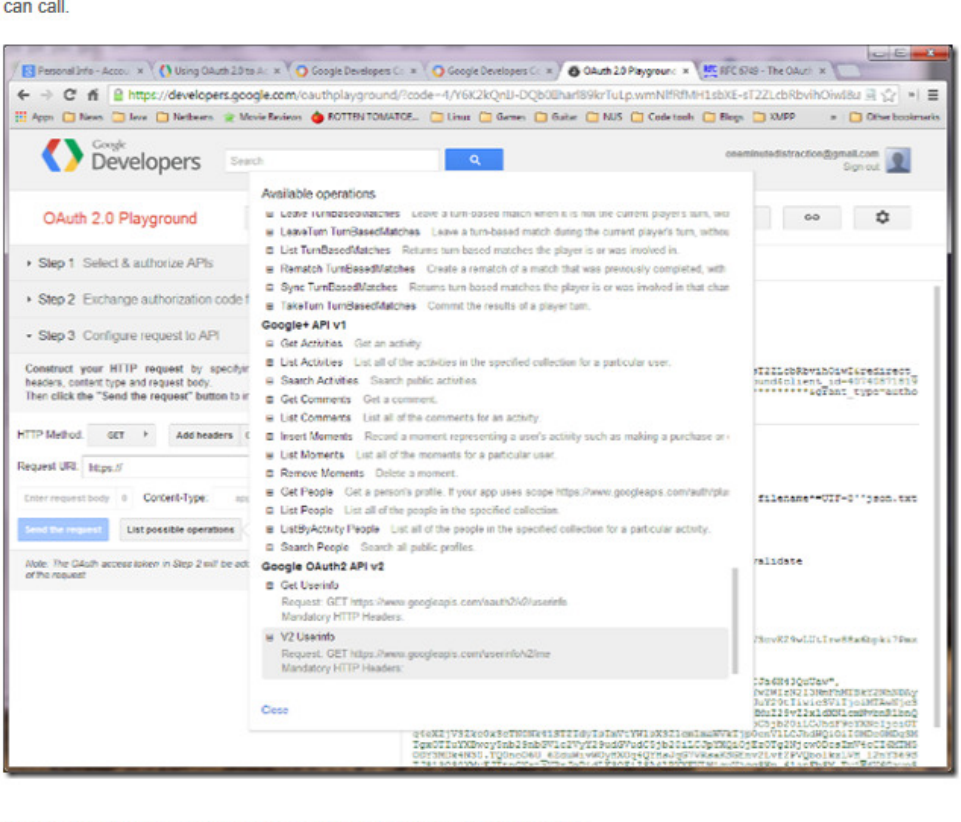
The URL that you see are the scopes. Make a note of these eg. if the example on the left, I have 3 scopes: `https://www.googleapis.com/auth/plus.me`, `https://www.googleapis.com/auth/userinfo.email` and `https://www.googleapis.com/auth/userinfo.profile`.

Once you've selected all the API, click on the Authorize APIs button. This is to authorize playground application (for testing) to access these resources from your Google account.

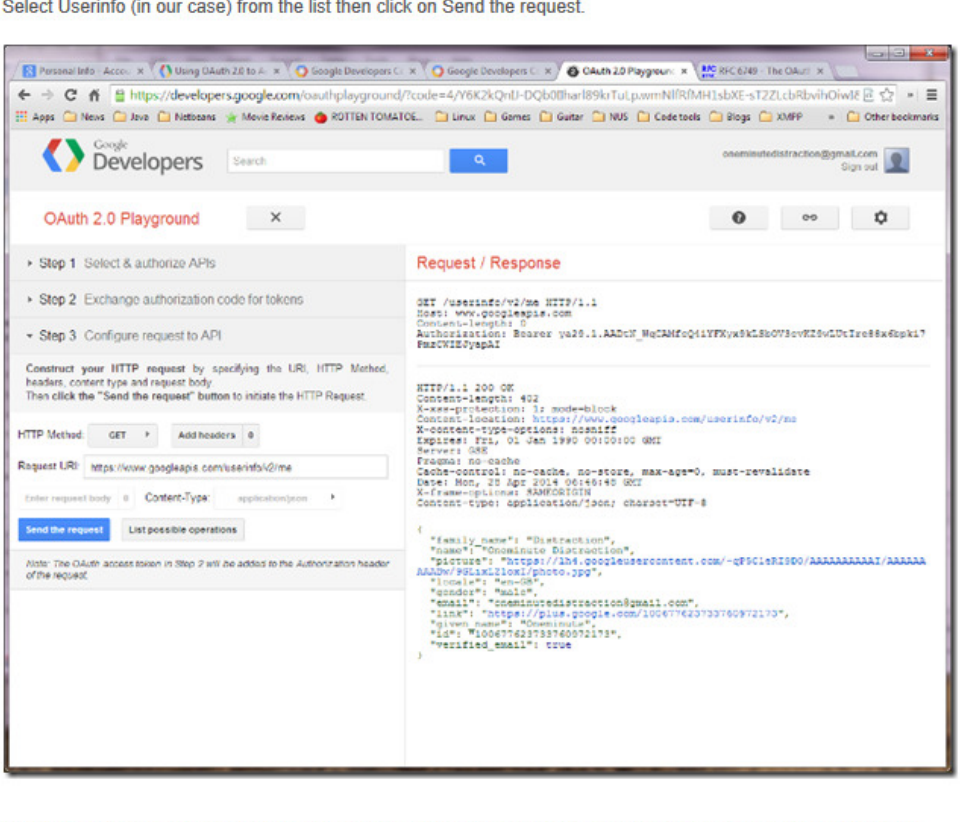
If you're using this the first time, then you'll probably see Playground's consent screen requesting permission to access to API from Step 1 above. Click on Accept to proceed. If you don't then you cannot test those APIs.



Once you've consented, you'll be taken to Step 2. Nothing much to do here except to click on Exchange authorization code button which will now take you to Step 3. We are now ready to test the Google+ API. Click on List possible operations button. Depending on what you've request in Step1, you will now be presented with a list of URL that you can call.



Select Userinfo (in our case) from the list then click on Send the request.



From the response, you can see the JSON structure of the resource. So by using the playground, you can find out

1. What is the URI for a particular to get a particular piece of information
2. What is the HTTP method to use
3. What is the JSON structure of the result
4. What are the required scopes (from step 1)

Integrating with JavaEE

We now have everything we need from the OAuth provider, we will now look at how to add this to our web application.

Start by configuring your web application to use form based authentication. You should provide a login page with a button to login via Google.

```
<form method="GET" action="googleplus">
   <button type="submit">Login</button>
</form>
```

You could also provide the option of using the traditional way of login with `j_security_check`.

We will now have to write a servlet that maps to `googleplus`. This servlet will be performing the OAuth2 setup and login. Lets look at the code first

```
@WebServlet("/googleplus")
public class GooglePlusServlet extends HttpServlet {

   private static final String CLIENT_ID = "client id here";
   private static final String CLIENT_SECRET = "client secret here";

   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {

      //Configure
      ServiceBuilder builder= new ServiceBuilder();
      OAuthService service = builder.provider(Google2Api.class)
         .apiKey(CLIENT_ID)
         .apiSecret(CLIENT_SECRET)
         .callback("http://localhost:8080/mywebapp/oauth2callback&quot;)
         .scope("openid profile email " +
               "https://www.googleapis.com/auth/plus.login " +
               "https://www.googleapis.com/auth/plus.me")
         .debug(System.out)
         .build(); //Now build the call

      HttpSession sess = req.getSession();
      sess.setAttribute("oauth2Service", service);

      resp.sendRedirect(service.getAuthorizationUrl(null));
   }
}
```

`GooglePlusServlet` is the servlet that will handle the `OAuth` login.Google has documented the authentication exchange here. I'm however going to using the excellent Scribe library. From the `ServiceBuider` we lookup Google's OAuth2 Provider class. Once we have the `OAuthService`, we need to set the client id, the client secret and the URI callback . All these 3 pieces of information must match the project we created earlier.

Next comes the scope information; scope is use to indicate to Google what resources are we interested in accessing. In other words what is the scope of this request. The scope in this case must always start with the string `openid` followed by a space delimited scope namespaces. The 2 standard namespaces are `profile` and `email`. If we just have those 2 namespace, then we are saying that we are interested in the user's profile and email. Google supports additional scopes for its services. This is where playground comes in. If you recall, when we were exploring the API, we are also shown the scope. If we want the request to access the user's G+ profile and circles, then add their scope namespaces to the `scope()` method as shown above.

Note: Scribe supports may OAuth providers out of the box. However Google is **NOT** one of them; the reason is found here. This gist adds support for Google. Add the Google2Api class into your project and you're good to go.

After constructing `OAuthService`, save that in the session cause we'll be using it in the next phase of the authentication.

Now its time to forward our request to Google. Get the authorization URL from `service.getAuthorizationUrl()` and perform a redirect to it. So what is going to happen now is

- If you are not login to your Google account, you will now be asked to do so.
- If this is the first time you are using the web application, you will be shown the consent screen. You can now choose to accept or reject. Once you've consented to the request, this permission is saved into the user's account permissions.

Google will now perform a callback to the configured callback URL. To handle the callback, we're going to need another servlet

```
@WebServlet(urlPatterns={"/oauth2callback", asyncSupported=true)
public class OAuth2CallbackServlet extends HttpServlet { @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws IOException, ServletException {

      //Check if the user have rejected
      String error = req.getParameter("error");
      if ((null != error) && ("access_denied".equals(error.trim())) {
         HttpSession sess = req.getSession();
         sess.invalidate();
         resp.sendRedirect(req.getContextPath());
         return;
      }

      //OK the user have consented so lets find out about the user
      AsyncContext ctx = req.startAsync();
      ctx.start(new GetUserInfo(req, resp, asyncCtx));
   }
}

public class GetUserInfo implements Runnable {
   private HttpServletRequest req;
   private HttpServletResponse resp;
   private AsyncContext asyncCtx;
   public GetUserInfo(HttpServletRequest req, HttpServletResponse resp,
AsyncContext asyncCtx) {
      this.req = req;
      this.resp = resp;
      this.asyncCtx = asyncCtx;
   }

   @Override
   public void run() {
      HttpSession sess = req.getSession();
      OAuthService serivce =
(OAuthService)sess.getAttribute("oauth2Service");

      //Get the all important authorization code
      String code = req.getParameter("code");
      //Construct the access token
      Token token = service.getAccessToken(null, new Verifier(code));
      //Save the token for the duration of the session
      sess.setAttribute("token", token);

      //Perform a proxy login
      try {
         req.login("fred", "fredfred");
      } catch (ServletException e) {
         //Handle error - should not happen
      }

      //Now do something with it - get the user's G+ profile
      OAuthRequest oReq = new OAuthRequest(Verb.GET,
            "https://www.googleapis.com/oauth2/v2/userinfo&quot;);
      service.signRequest(token, oReq);
      Response oResp = oReq.send();

      //Read the result
      JsonReader reader = Json.createReader(new ByteArrayInputStream(
            oResp.getBody().getBytes()));
      JsonObject profile = reader.readObject();
      //Save the user details somewhere or associate it with
      sess.setAttribute("name", profile.getString("name"));
      sess.setAttribute("email", profile.getString("email"));
         ...
      asyncCtx.complete();
   }
}
```

The callback is routed to this servlet; we check if the user have rejected the request. If not we proceed to get the user's details. I'm using an async servlet here because we'll be making request out to Google for the user's info and we do not want to hold on to the request processing thread.

If the user have consented to the request, then get the `code` from the query string. This is the most important parameter. Using the code we construct the access token. For every request to Google, we will need the access token to use the token to sign it. Constructing a new token is quite straight forward. You can also refresh an old token by passing in the token as the first parameter in `getAccessToken()`. Once you have gotten your token, save it in the session.

We are now ready to get the user's profile. For your experimentations in the playground (step3), you now know which URL to use and what HTTP method to use. Construct that using `OAuthRequest` class. Sign it with the token, before making the call.

When the response, returns parse it using a Json parser. Since I'm using EE7, I'll be using the Json API to perform this. Again, from playground, we know the structure of the Json response from Google. You can now save all the relevant information.

One last thing before we end. So we have successfully authenticated with Google using OAuth. But we have not login to the application. To do that, I've decided to create a proxy user in the authentication realm of the application server. After you have authenticated with Google, login to this proxy user so that the application server will honour all the security declarations. Just be aware that you cannot use `getRemoteUser()` or use `getPrincipal()` to get a user's detail. A more tightly integrated approach would be to develop a custom realm (Glassfish in this example). However custom realms may not be portable.

One More Time…

There are may details and configurations so before I end lets go through the steps again

Blog Stats

154,183 hits

One Minute Tweet

Photostream Application - Part 3/3: youtu.be/GXI-aVfnZnE?a via @YouTube 1 year ago

Photostream Application - Part 2/3: youtu.be/NqAtauTKKtw?a via @YouTube 1 year ago

Photostream Application - Part 1/3: youtu.be/iM4L-xWkcis?a via @YouTube 1 year ago

@speakjava Congrats! A dream job! 1 year ago

I liked a @YouTube video youtu.be/5gDsXbMTBd8?a Tai Chi 24 form - slow version 1 year ago

@javapassion @TIME once in everybody's lifetime 1 year ago

CORS Support for JavaEE Web Applications oneminutedistraction.wordpress.com/2015/07/14/cor… 1 year ago

Hybrid Mobile Application with Ionic/Cordova: youtu.be/jbopML8dnqg?a via @YouTube 1 year ago

Try MongoDB realm for Glassfish (and Payara) bit.ly/1vmgOke 2 years ago

Build a Yet Another Weather App with the Angular Workshop Kit oneminutedistraction.wordpress.com/2014/10/14/ang… 2 years ago

Follow @1MinDistraction

Follow •••

All the screencast in one place

The One Minute Distraction Channel

The Lair of the Jabberwocky

Project home

Contact

oneminutedistraction at g mail

My Posts

Posts

April 2014

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

« Mar May »

My tags

angularjs cdi cordova dataform java vorpal xep-0004 jabberwocky feature glassfish glassfish-3.1 jabberwocky netbeans-7.0 java glassfish-v3 xep-0114 container java xmpp jabber html5 invokedynamic jsr-292 ionicframework jabber jabber xmpp conversation cdi vorpal jabberwocky jabber xmpp java vorpal glassfish xep-030 disco#info disco#item jabber xmpp netbeans glassfish vorpal jabberwocky screencast java java cdi xmpp client javaee javaee6 javaee7 java glassfish netbeans xmpp vorpal presence jabber java jabber xmpp xep-0114 tinder smack java jabber xmpp xep-0114 tinder smack dom4j java javaee7 websocket socketio javascript java jsf totp glassfish javaee security faces-flow java rest javaee javascript javascript java xmpp jabber smack pep XEP163 196 user gaming java xmpp jabber vorpal glassfish jax-rs jquery jsonp mongo netbeans jabberwocky glassfishv3 xmpp xep-0114 oauth2 pep pubsub xep-163 xep-115 presentation pubsub revealjs rss scribe servlet smack SSE vorpal vorpal2 xmpp game multiplayer muc vorpal2 xmpp java client vorpal xmpp java conversation vorpal xmpp java tools netbeans webpofile xep-0030 disco#info disco#items java vorpal glassfish netbeans xep-0114 glassfish xmpp jabber xep-0114 java jabber framework vorpal xep-0114 java smack whack xep-0115 pep xep-163 java smack xmpp xmpp conversation vorpal jabberwocky stateful xmpp glassfish xep-0114 jmx jabberwocky xmpp jabber openfire smack java xmpp jabberwocky vorpal glassfish java cdi xmpp java jabber smack service discovery google_talk jabber.org xmpp java smack pubsub disco#item disco#info xmpp java xep-0114 presence vorpal xmpp vorpal conversation muc