OAuth 2.0 Service Provider Implementation in Apache Shindig

OAuth 2.0 Service Provider for Apache Shindig

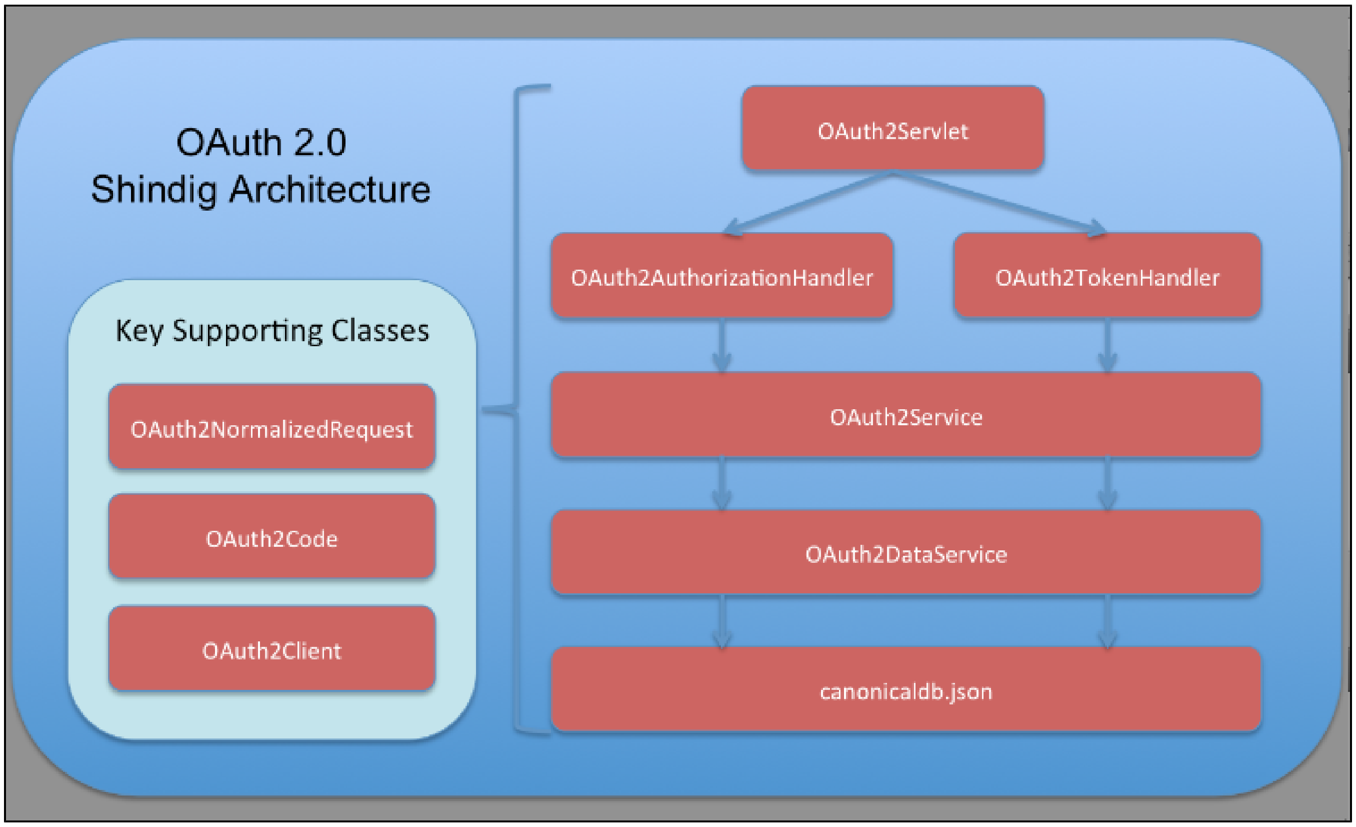
Eric Woods and Matthew Marum are implementing support for an OAuth 2.0 Service Provider in Apache Shindig. This article provides an overview of the implementation including high level design, supported flows, common How-Tos, and future considerations.

The OAuth 2.0 specification is here: http://tools.ietf.org/html/draft-ietf-oauth-v2-20

OAuth 2.0 support is ENABLED by default. However, clients can still anonymously access the Social APIs using the default AuthenticationHandlerProvider. To prevent anonymous access to Shindig's Social APIs, modify the existing one or create your own AuthenticationHandlerProvider. See the How-Tos section below for details.

For more information on the OAuth 2.0 Consumer, click here: TODO

High Level Design



All OAuth 2.0 requests are received by the OAuth2Servlet. Per the spec, there are two OAuth 2.0 related endpoints: /oauth/authorize and /oauth /token. The request is quickly delegated to either the OAuth2AuthorizationHandler or the OAuth2TokenHandler accordingly.

The first thing either handler does is normalize the incoming OAuth 2.0 request using the OAuth2NormalizedRequest class. This class is designed to receive the incoming HttpServletRequest and normalize all OAuth 2.0-related fields. For example, a pre-registered client may authenticate using Basic Authentication or by including "client_secret" as a URL parameter. These permutations make it difficult to process an incoming request consistently. OAuth2NormalizedRequest will make the client secret accessible by getClientSecret() regardless of authentication method, thus "normalizing" the method that was used to authenticate, pass a token, pass a grant type, etc.

After normalization, the request is processed. We have two distinct layers to process OAuth requests: OAuth2Service and OAuth2DataService. The OAuth2Service handles all OAuth processing and enforcement as defined by the OAuth 2.0 specification. For example, the OAuth2Service handles client authentication and request validation for an authorization code, access token, refresh token, or resource. The OAuth2Service also handles generation of authorization codes, access tokens, and refresh tokens.

The second layer, OAuth2DataService, represents the data store to manage authorization codes, access tokens, refresh tokens, and association with a pre-registered client. For example, the OAuth2DataService manages registering/unregistering the various OAuth 2.0 codes/tokens, retrieving a client by ID, and retrieving a code/token by value.

We provide "poor-man" implementations of the two service interfaces; the OAuth2ServiceImpl uses in-memory data structures, and the OAuth2DataServiceImpl uses canonical.json.

OAuth 2.0 Flow Support

- Authorization Code Flow (done)
- Implicit Grant Flow (done)
- Client Credential Flow (done)

User Authentication

User authentication for this OAuth 2.0 service provider is handled through the sample ShiroFilter that Shindig also used for the OAuth 1.0a implementation. This means our service provider makes assumptions that requests to our authorization endpoint have been user authenticated by the time they are received.

How-Tos

How to Enable/Disable OAuth 2.0 Security

OAuth 2.0 support is enabled in Shindig's default AuthenticationHandlerProvider.

OAuth 2.0 support can be added to your existing AuthenticationHandlerProvider by adding the OAuth2AuthenticationHandler to the chain. The OAuth2AuthenticationHandler ensures that requests for protected resources (the Social REST APIs) include valid access tokens in the form of Bearer tokens as defined in the OAuth 2.0 specs.

If your AuthenticationHandlerProvider doesn't include the OAuth2AuthenticationHandler, access tokens will not be checked when clients access protected resources (the Social REST APIs).

If necessary, you can disable the OAuth 2.0 token and authorization endpoints by removing mappings for the OAuth2Servlet in the Shindig web.xml.

How to Register a Client

Clients are pre-registered in canonicaldb.json. Client registration, authorization codes, access tokens, and refresh tokens are keyed by client. Here's the current structure (subject to change):

```
// Registry of OAuth 2.0 clients with Shindig's service provider.
"oauth2" : {
    "advancedOpenSocialClient" : {
        "registration" : {
            "id" : "advancedOpenSocialClient",
            "secret": "advancedOpenSocialClient_secret",
            "title": "Most Advanced OpenSocial Client Ever!",
            "redirectUri" : "http://localhost:8080/oauthclients/OpenSocialClient",
            "type" : "confidential",
            "flow" : "authentication_code"
        },
        "authorizationCodes" : {
            "advancedClient_authcode_1" : {
                // Authentication code has been consumed since associatedSignature exists
                "redirectUri" : "http://localhost:8080/oauthclients/OpenSocialClient",
                "associatedSignature" : "advancedClientOS_accesstoken_1"
            },
            "advancedClient_authcode_2" : {
                "redirectUri" : "http://localhost:8080/oauthclients/OpenSocialClient"
            }
        },
        "accessTokens" : {
            "advancedClient_accesstoken_1" : {
                "redirectUri" : "http://localhost:8080/oauthclients/OpenSocialClient"
            }
        }
    },
   "testClient" : {
        "registration" : {
            "id" : "testClient",
            "redirectUri" : "http://localhost:8080/oauthclients/OpenSocialClient",
            "type" : "public"
        }
    }
}
```

Clients can only be registered with a single flow at a given time. Some flows have restrictions. For example, Client Credentials flow requires a confidential client.

How to Add a Custom Grant Validator

Per the OAuth 2.0 specification, grant types are extensible beyond the 4 pre-defined types (authorization_code, client_credentials, refresh_token, and password). To accommodate, validation of new grant types, grant handlers are registered with the OAuth2Service. The appropriate grant handler is identified by its registered "grant_type" and invoked within OAuth2Service.validateRequestForAccessToken(...) to validate requests.

TODO: where and how are grant handlers registered?

Future Considerations

Currently, the entire OAuth 2.0 implementation is located in a single package within Shindig's social-api module. We need to determine the most appropriate way to integrate this package into shindig. We could integrate this package into social-api following the existing patterns, or it may even be broken out as an entirely new module within Shindig. To be determined...

Additionally, client registration is currently facilitated by modifying canonicaldb.json. We could expose client registration services via a REST API and provide a UI for clients to register themselves.