

5000th Anniversary

Services + REST and OAuth

The purpose of services

Create a Drupal API for exposing web APIs

The official version

- Create a unified Drupal API for web services to be exposed in a variety of different server formats.
- Provide a service browser to be able to test methods.
- Allow distribution of API keys for developer access.

Services

Functionality split into three kinds of modules that provides:

- ✦ Servers
- ✦ Services
- ✦ Authentication mechanisms (new in 2.x)

Servers

- ◆ REST
- ◆ XMLRPC
- ◆ JSONRPC
- ◆ SOAP
- ◆ AMF (Binary Flash RPC protocol)

Services

- ✦ That exposes core
 - ✦ Node, user, taxonomy, menu, file,
- ✦ That exposes other modules
 - ✦ Views
- ✦ And additional services implemented in other contrib modules that I cant remember right now.

Authentication

- ◆ OAuth
- ◆ Key authentication

Implementing services

- ◆ Either as methods
- ◆ ...or (since version 2.x) as resources

Methods

- ✦ Pretty similar to the menu system
- ✦ Each service-implementing module returns a non-associative array with methods
- ✦ Method definitions contains a method attribute: “node.get”, “node.view”, “node.save”, “node.delete”
- ✦ ...and information about callbacks, parameters, access rules et cetera.

```
<?php
/**
 * Implementation of hook_service().
 */
function node_service_service() {
  return array(
    // node.get
    array(
      '#method'           => 'node.get',
      '#callback'         => 'node_service_get',
      '#access callback'  => 'node_service_get_access',
      '#file'             => array('file' => 'inc', 'module' => 'node_service'),
      '#args'             => array(
        array(
          '#name'         => 'nid',
          '#type'         => 'int',
          '#description'  => t('A node ID.')),
          ...
        ),
      '#return'          => 'struct',
      '#help'            => t('Returns a node data.')
    ),
  ),
}
```

Drawbacks

- ✦ No semantics
 - ✦ `node.view` is treated exactly like `node.delete`
- ✦ Lack of consistency
 - ✦ “`taxonomy.saveTerm`”, “`node.save`”
 - ✦ “`node.view`”, “`user.get`”
- ✦ Lack of structure makes it hard to alter through alter hooks.

Resources

- ✦ Adds semantics to the methods
- ✦ Natural grouping around resources
 - ✦ no more “taxonomy.saveTerm”
- ✦ Methods are divided into CRUD-operations, actions, targeted actions and relationships

Structure - CRUD

- ◆ Resource
 - ◆ Create
 - ◆ Retrieve
 - ◆ Update
 - ◆ Delete
 - ◆ (Index)

Extensions of CRUD

- ◆ Actions
 - ◆ Similar to static class methods:
`Node::publish_my_drafts()`
- ◆ Targeted actions
 - ◆ Like class methods: `$node->publish()`
- ◆ Relationships
 - ◆ Like targeted actions but for read-operations:
`$node->get_comments()`

All old services can be expressed as resources

- ✦ Direct translation through adding the old methods (taxonomy.saveTerm, saveVocabulary, getTree, selectNodes) as actions on the taxonomy resource.
- ✦ Or even better, create them as real resources (vocabulary and term).

OAuth

- ✦ Secure protocol for avoiding “the password anti-pattern”.
- ✦ A strong emerging standard.
- ✦ Client implementations available for most small and large languages.
- ✦ See <http://oauth.net/code>



OAuth workflow for the user

- ✦ Initiates the authorization process in a third-party application (consumer). Is redirected to our site (the provider).
- ✦ The user logs in to the provider and is asked to authorize the consumer.
- ✦ The user is sent back to the consumer. And were done!

Token-based security

- ◆ Three tokens (key+secret) are involved: consumer-token, request-token and access-token.
- ◆ The consumer uses it's consumer-token to retrieve a request token.
- ◆ The user authorizes our request token.
- ◆ The consumer uses it's request token to fetch a access token.
- ◆ The consumer can then use the consumer+access-token to access protected resources.

The REST server

- ♦ REST is designed to work as well as possible with HTTP.
- ♦ All resources are accessible through a url
 - ♦ Create: POST <http://example.com/node>
 - ♦ Retrieve: GET <http://example.com/node/123>
 - ♦ Index: GET <http://example.com/node>
 - ♦ Update: PUT <http://example.com/node/123>
 - ♦ Delete: DELETE <http://example.com/node/123>

The extensions to CRUD

- ◆ Actions
 - ◆ POST
http://example.com/node/publish_my_drafts
- ◆ Targeted actions
 - ◆ POST
<http://example.com/node/123/publish>
- ◆ Relationships
 - ◆ GET
<http://example.com/node/123/comments>

Multiple response formats

- ✦ XMLRPC always returns XML, JSONRPC returns JSON, SOAP returns XML+cruft and so on.
- ✦ REST is format agnostic and can give responses in different formats based on file endings and Accept-headers.
 - ✦ GET <http://example.com/node/123.json>
 - ✦ GET <http://example.com/node/123.xml>
 - ✦ GET <http://example.com/node/123.php>
- ✦ Other modules can add and alter response formats through `hook_rest_server_response_formatters_alter()`.

All response formats inherit from RESTServerView

```
/**
 * Base class for all response format views
 */
abstract class RESTServerView {
    protected $model;
    protected $arguments;

    function __construct($model, $arguments=array()) {
        $this->model = $model;
        $this->arguments = $arguments;
    }

    public abstract function render();
}
```

More advanced response formats

- ✦ The response formats that can't use simple serialization
 - ✦ RSS, iCal, xCal med flera
- ✦ The format can then demand that the method shall implement a data model that works like an adapter.

Example from xCal

```
function xcal_..._formatters_alter(&$formatters) {  
  $formatters['xcal'] = array(  
    'model' => 'ResourceTimeFeedModel',  
    'mime types' => array('application/xcal+xml'),  
    'view' => 'XCalFormatView',  
  );  
  $formatters['ical'] = array(  
    'model' => 'ResourceTimeFeedModel',  
    'mime types' => array('text/calendar'),  
    'view' => 'XCalFormatView',  
    'view arguments' => array('transform'=>'ical'),  
  );  
}
```

The resource declares support for the model, not the format

```
'models' => array(  
  'ResourceFeedModel' => array(  
    'class' => 'NodeResourceFeedModel',  
  ),  
  'ResourceTimeFeedModel' => array(  
    'class' => 'NodeResourceFeedModel',  
  ),  
)
```

Multiple input-formats

- ✦ Built in support for x-www-form-urlencoded, yaml, json and serialized php.
- ✦ Can be extended through `hook_rest_server_request_parsers_alter()`.
- ✦ Determined by the Content-type-header for the call and therefore matched to mime-types:
 - ✦ 'application/json' => 'RESTServer::parseJSON',
 - ✦ 'application/vnd.php.serialized' => 'RESTServer::parsePHP',

My view on the future of services - 3.x

- ✦ The old RPC-oriented methods are completely removed and are replaced by resources.
- ✦ Possibly support translation of method declarations to a resource with actions.
- ✦ Endpoints: modules and administrators will be able to publish and configure servers on arbitrary locations in the menu system.

Why endpoints?

Today all installed services are always available on all installed servers and they all have to use the same auth method.

Why Endpoints?

- ✦ Today it's not possible for modules to use services to expose an API.
- ✦ API = services + server + authentication mechanism
- ✦ Eller rättare sagt, endast ett API kan exponeras
- ✦ Or rather - only one API can be exposed at a time
- ✦ This becomes a problem if services is going to be used as a framework for other modules to publish API:s

Endpoints

- ✦ Can be configured independently of each other.
And you can choose:
- ✦ which server that should be used, and what path its should be placed on
- ✦ exactly what services should be exposed
- ✦ what authentication module that should be used, and how it should be configured

Endpoints makes it possible to

- ✦ Expose several different API:s on one Drupal install
- ✦ Define an API in your module that will become available when the module is installed.
- ✦ Package your API as a feature, this is planned to be supported through chaos tools.

Example of a endpoint- declaration

```
/**
 * Implementation of hook_services_endpoints().
 */
function conglomerate_services_endpoints() {
  return array(
    'conglomerate' => array(
      'title' => 'Conglomerate API',
      'server' => 'rest_server',
      'path' => 'api',
      'authentication' => 'services_oauth',
      'authentication_settings' => array(
        'oauth_context' => 'conglomerate',
      ),
      'resources' => array(
        'conglomerate-content' => array(
          'alias' => 'content',
          'operations' => array(
            'create' => array(
              'enabled' => TRUE,
              'oauth_credentials' => 'token',
              'oauth_authorization' => '*',
            ),
            'retrieve' => array(
              'enabled' => TRUE,
              'oauth_credentials' => 'unsigned_consumer',
              'oauth_authorization' => 'read',
            ),
          ),
        ),
      ),
    ),
  ),
);
```

OAuth and Endpoints

- ✦ OAuth now has support for contexts.
- ✦ Consumers are always placed in a context
- ✦ Authentications are therefore only valid within this context.
- ✦ Each context has it's own authorization levels
- ✦ Endpoints in services can either use separate contexts or share contexts.

OAuth context declaration in code

```
/**
 * Implementation of hook_oauth_default_contexts().
 */
function conglomerate_oauth_default_contexts() {
  return array(
    'conglomerate' => array(
      '*' => array(
        'title' => 'Yes, I want to connect !appname to !sitename',
        'description' => 'This will allow your site !appname to push content to !sitename',
        'weight' => -1,
      ),
      'read' => array(
        'title' => 'I want to connect, but just to get stuff from !sitename',
        'description' => 'This will allow !appname to fetch content from !sitename, but it will not
allow any information to be pushed to !sitename.',
        'weight' => 0,
      ),
    )
  );
}
```

Hugo Wetterberg

@hugowett

hugo@goodold.se

<http://github.com/hugowetterberg>