

Creating a resource for Services 3.x

We'll implement a simple service for notes that's exposed using the REST Server. Then we'll implement a simple JavaScript client for note-taking. This post is pretty much written like extended code comments and [the full code and article text is available here](#). *Note: this page contains a bug fix in the JavaScript testing logic for Create-POST and Update-Put that is not in that link.* Feel free to fork this repo and flesh out the text if you want to.

Agenda

- Getting the necessary modules
- Implementing a note service (resource)
- Creating an endpoint
- Writing a simple JavaScript client

Getting the necessary modules

First off we have to download the necessary modules of Drupal.org.

Services

Download the 6.x-3.x or 7.x-3.x version from [the Services project page at Drupal.org](#).

Chaos tool suite

The only dependency for services. Provides the framework for the endpoint definitions so that they can be exported and defined in both code and the database. Maybe further along the road it'll be used for a plugin system for the servers and authentication mechanisms.


- Download from [the CTools project page at Drupal.org](#).

REST Server

My server implementation of choice and what we'll be using to test our service. This is included in the Services download.

Autoload

Autoload is required by the REST Server for Drupal core 6.x. The Autoload functionality was moved into the Drupal 7 core.

 The Autoload module is a utility module. It allows other modules to leverage PHP 5's class autoloading capabilities in a unified fashion.

- Download from [the Autoload project page at Drupal.org](#).

Implementing a note service (resource)

Create a noteresource module that will contain our service implementation. The info file could look something like this:

```
name = Note Resource
description = Sample resource implementation
package = Notes example
dependencies[] = services
core = 6.x
php = 5.2
```

To get a really simple example I'll create an api for storing notes. In a real world scenario notes would probably be stored as nodes, but to keep things simple we'll create our own table for storing notes.

```
// noteresource.install
/**
 * Implements hook_install().
 */
function noteresource_install() {
  drupal_install_schema('noteresource');
}

/**
 * Implements hook_uninstall().
 */
function noteresource_uninstall() {
  drupal_uninstall_schema('noteresource');
}

/**
 * Implements hook_schema().
 */
function noteresource_schema() {
  return array(
    'note' => array(
      'description' => 'Stores information about notes',
      'fields' => array(
        'id' => array(
          'description' => 'The primary identifier for a note.',
          'type' => 'serial',
          'unsigned' => TRUE,
          'not null' => TRUE,
        ),
        'uid' => array(
          'description' => t('The user that created the note.'),
          'type' => 'int',
          'unsigned' => TRUE,
          'not null' => TRUE,
          'default' => 0,
        ),
        'created' => array(
          'description' => t('The timestamp for when the note was created.'),
          'type' => 'int',
          'unsigned' => TRUE,
          'not null' => TRUE,
          'default' => 0,
        ),
      ),
    ),
  );
}
```

```

    'modified' => array(
      'description' => t('The timestamp for when the note was modified.'),
      'type' => 'int',
      'unsigned' => TRUE,

      'not null' => TRUE,
      'default' => 0,
    ),
    'subject' => array(
      'description' => t('The subject of the note'),
      'type' => 'varchar',
      'length' => 255,
      'not null' => TRUE,
    ),
    'note' => array(
      'description' => t('The note'),
      'type' => 'text',
      'size' => 'medium',
    ),
  ),
  'primary key' => array('id'),
),
);
}

```

The familiar stuff

Now lets implement some basic hooks and API methods. We need some permissions that'll be used to decide what our users can and cannot do:

```

// noteresource.module
/**
 * Implements hook_perm().
 */
function noteresource_perm() {
  return array(
    'note resource create',
    'note resource view any note',
    'note resource view own notes',
    'note resource edit any note',
    'note resource edit own notes',
    'note resource delete any note',
    'note resource delete own notes',
  );
}

```

Now for some Drupal API methods for the basic CRUD operations for our notes. These will be used by the functions that are used as callbacks for our resource. But it's always a good idea to supply functions like these so that other Drupal modules have a nice and clean interface to your module's data.

```

// noteresource.module
/**
 * Gets a note object by id.
 *
 * @param int $id

```

```

* @return object
*/
function noteresource_get_note($id) {
    return db_fetch_object(db_query("SELECT * FROM {note} WHERE id=:id", array(
        ':id' => $id,
    )));
}

/**
 * Writes a note to the database
 *
 * @param object $note
 * @return void
 */
function noteresource_write_note($note) {
    $primary_key = !empty($note->id) ? array('id') : NULL;
    drupal_write_record('note', $note, $primary_key);
}

/**
 * Deletes a note from the database.
 *
 * @param int $id
 * @return void
 */
function noteresource_delete_note($id) {
    db_query("DELETE FROM {note} WHERE id=:id", array(
        ':id' => $id,
    ));
}

```

Defining our resource

All resources are defined through `hook_services_resources()`. The way resources are declared is quite similar to how the template and menu system works, it also bears a very close resemblance to how 2.x services are defined.

Notice how we define the basic CRUD methods here: create, retrieve, update, delete (and index). Most resources implement these methods, but it is also possible to implement actions, targeted actions and relationships. Those won't be covered here but their general nature is explained in [the REST Server README](#).

All the methods have `'file' => array('type' => 'inc', 'module' => 'noteresource', 'name' => 'noteresource')`, specified, which tells services that it can find the callback function in the file `noteresource.inc`, which is where we will write them all.

```

// noteresource.module
/**
 * Implements hook_services_resources().
 */
function noteresource_services_resources() {
    return array(
        'note' => array(
            'retrieve' => array(
                'help' => 'Retrieves a note',

```

```

'file' => array('type' => 'inc', 'module' => 'noteresource', 'name' => 'noteresource'),
'callback' => '_noteresource_retrieve',
'access callback' => '_noteresource_access',
'access arguments' => array('view'),

'access arguments append' => TRUE,
'args' => array(
    array(
        'name' => 'id',
        'type' => 'int',
        'description' => 'The id of the note to get',
        'source' => array('path' => '0'),
        'optional' => FALSE,
    ),
),
),
'create' => array(
    'help' => 'Creates a note',
    'file' => array('type' => 'inc', 'module' => 'noteresource', 'name' => 'noteresource'),
    'callback' => '_noteresource_create',
    'access arguments' => array('note resource create'),
    'access arguments append' => FALSE,
    'args' => array(
        array(
            'name' => 'data',
            'type' => 'struct',
            'description' => 'The note object',
            'source' => 'data',
            'optional' => FALSE,
        ),
    ),
),
),
'update' => array(
    'help' => 'Updates a note',
    'file' => array('type' => 'inc', 'module' => 'noteresource', 'name' => 'noteresource'),
    'callback' => '_noteresource_update',
    'access callback' => '_noteresource_access',
    'access arguments' => array('update'),
    'access arguments append' => TRUE,
    'args' => array(
        array(
            'name' => 'id',
            'type' => 'int',
            'description' => 'The id of the node to update',
            'source' => array('path' => '0'),
            'optional' => FALSE,
        ),
        array(
            'name' => 'data',
            'type' => 'struct',
            'description' => 'The note data object',
            'source' => 'data',
            'optional' => FALSE,
        ),
    ),
),
),

```

```

    },
    'delete' => array(
        'help' => 'Deletes a note',
        'file' => array('type' => 'inc', 'module' => 'noteresource', 'name' => 'noteresource'),

        'callback' => '_noteresource_delete',
        'access callback' => '_noteresource_access',
        'access arguments' => array('delete'),
        'access arguments append' => TRUE,
        'args' => array(
            array(
                'name' => 'nid',
                'type' => 'int',
                'description' => 'The id of the note to delete',
                'source' => array('path' => '0'),
                'optional' => FALSE,
            ),
        ),
    ),
),
'index' => array(
    'help' => 'Retrieves a listing of notes',
    'file' => array('type' => 'inc', 'module' => 'noteresource', 'name' => 'noteresource'),
    'callback' => '_noteresource_index',
    'access callback' => 'user_access',
    'access arguments' => array('access content'),
    'access arguments append' => FALSE,
    'args' => array(array(
        'name' => 'page',
        'type' => 'int',
        'description' => '',
        'source' => array(
            'param' => 'page',
        ),
        'optional' => TRUE,
        'default value' => 0,
    ),
    array(
        'name' => 'parameters',
        'type' => 'array',
        'description' => '',
        'source' => 'param',
        'optional' => TRUE,
        'default value' => array(),
    ),
),
),
);
}

```

There is another alternative when defining services (which I personally prefer) but that will probably be covered in a later article. Take a look at http://github.com/hugowetterberg/services_oop if you're curious.

Implementing the callbacks

Create the file `noteresource.inc` which is where we told services that it could find our callbacks.

We'll start with the create-callback. The method will receive an object describing the note that is about to be saved. The attributes we want are `subject` and `note` and we'll throw an error if those are missing. We return the id of the created note, and its uri so that the client knows how to access it. A get-request to the uri will return the full note.

```
// noteresource.inc
/**
 * Callback for creating note resources.
 *
 * @param object $data
 * @return object
 */
function _noteresource_create($data) {
    global $user;

    unset($data->id);
    $data->uid = $user->uid;
    $data->created = time();
    $data->modified = time();

    if (!isset($data->subject)) {
        return services_error('Missing note attribute subject', 406);
    }

    if (!isset($data->note)) {
        return services_error('Missing note attribute note', 406);
    }

    noteresource_write_note($data);
    return (object)array(
        'id' => $data->id,
        'uri' => services_resource_uri(array('note', $data->id)),
    );
}
```

The update callback works more or less the same, but we don't have to check that subject and note exists, there is no harm in allowing a client to just update the subject and leave the note alone.

```
// noteresource.inc
/**
 * Callback for updating note resources.
 *
 * @param int $id
 * @param object $data
 * @return object
 */
function _noteresource_update($id, $data) {
    global $user;

    $note = noteresource_get_note($id);

    unset($data->created);
    $data->id = $id;
```

```

    $data->uid = $note->uid;
    $data->modified = time();

    noterresource_write_note($data);

    return (object)array(
        'id' => $id,
        'uri' => services_resource_uri(array('note', $id)),
    );
}

```

The retrieve and delete callbacks are pretty trivial and probably don't need any further explanation.

```

// noterresource.inc
/**
 * Callback for retrieving note resources.
 *
 * @param int $id
 * @return object
 */
function _noterresource_retrieve($id) {
    return noterresource_get_note($id);
}

/**
 * Callback for deleting note resources.
 *
 * @param int $id
 * @return object
 */
function _noterresource_delete($id) {
    noterresource_delete_note($id);
    return (object)array(
        'id' => $id,
    );
}

```

The index callback fetches a users notes and returns them all. We specified some arguments for this method that we don't use. They are mostly here to show that it would be a good idea to support paging and filtering of a index listing.

```

// noterresource.inc
/**
 * Callback for listing notes.
 *
 * @param int $page
 * @param array $parameters
 * @return array
 */
function _noterresource_index($page, $parameters) {
    global $user;

    $notes = array();
    $res = db_query("SELECT * FROM {note} WHERE uid=:uid ORDER BY modified DESC", array(
        ':uid' => $user->uid,
    ));
}

```



```

));

while ($note = db_fetch_object($res)) {
    $notes[] = $note;
}

return $notes;
}

```

Access checking

Last but not least, we specified an access callback for all methods. This checks so that users don't overstep their bounds and starts looking at other people's notes without having the proper permissions. This function should be in the main .module file.

```

// noteresource.module
/**
 * Access callback for the note resource.
 *
 * @param string $op
 * The operation that's going to be performed.
 * @param array $args
 * The arguments that will be passed to the callback.
 * @return bool
 * Whether access is given or not.
 */
function _noteresource_access($op, $args) {
    global $user;
    $access = FALSE;

    switch ($op) {
        case 'view':
            $note = noteresource_get_note($args[0]);
            $access = user_access('note resource view any note');
            $access = $access || $note->uid == $user->uid && user_access('note resource view own notes');
            break;
        case 'update':
            $note = noteresource_get_note($args[0]->id);
            $access = user_access('note resource edit any note');
            $access = $access || $note->uid == $user->uid && user_access('note resource edit own notes');
            break;
        case 'delete':
            $note = noteresource_get_note($args[0]);
            $access = user_access('note resource delete any note');
            $access = $access || $note->uid == $user->uid && user_access('note resource delete own notes');
            break;
    }

    return $access;
}

```

As you can see neither the create nor the index function is represented here. That's because they both use `user_access()` directly. Unlike the other methods there are no considerations like note ownership to take into account. For creation the permission 'note resource create' is checked and for the index listing only 'access

content' is needed.

Creating an endpoint

The endpoint can actually be created in two ways either through the admin interface or through code. The easiest option is most often to create the endpoint through the interface, and then export it and copy paste it into your module.

Go to admin/structure/services and click "Add endpoint". Name your endpoint "notes" and call it something nice, like "Note API". Choose "REST" as your server and place the endpoint at "js-api".

Save and click the Resources tab/local task and enable all methods for the note resource. Then save your changes.

You should now have a proper working endpoint that exposes your note API. The easiest way to check that everything's working properly is to add a dummy note to your table. Then try to access it on js-api/note/[id].yaml, where [id] is the id of the note you created (probably 1).

Writing a simple JavaScript client

We'll put our javascript client in a module named noteresourcejs. The info file could look something like this:

```
name = Notes Javascript
description = Sample endpoint definition and javascript client implementation
package = Notes example

core = 6.x
php = 5.2
```

The javascript module will do two things: implement a javascript client; and provide the notes endpoint in code.

Defining the endpoint in code

First, we need to inform the Services module that our module implements it's API. We do this by implementing hook_ctools_plugin_api() as shown in the following:

```
/**
 * Implements hook_ctools_plugin_api().
 */
function noteresourcejs_ctools_plugin_api($owner, $api) {
  if ($owner == 'services' && $api == 'services') {
    return array(
      'version' => 3,
      'file' => 'noteresourcejs.services.inc', // Optional parameter to indicate the file name to load.
      'path' => drupal_get_path('module', 'noteresourcejs') . '/includes', // If specifying the file key, path is
      required.
    );
  }
}
```

Go to admin/structure/services and select Export for your Notes API endpoint. The code shown should be copy-pasted in a hook named hook_default_services_endpoint(), followed by the return value of return array(\$endpoint) (which is not provided in the Export code) or as shown in the example below:

```

// noteresourcejs.module
/**
 * Implements hook_default_services_endpoint().
 */
function noteresourcejs_default_services_endpoint() {
    $endpoints = array();

    $endpoint = new stdClass;
    $endpoint->disabled = FALSE; /* Edit this to true to make a default endpoint disabled initially */
    $endpoint->name = 'notes_js';
    $endpoint->title = 'Note API';
    $endpoint->server = 'rest_server';
    $endpoint->path = 'js-api';
    $endpoint->authentication = array();
    $endpoint->resources = array(
        'note' => array(
            'alias' => '',
            'operations' => array(
                'create' => array(
                    'enabled' => 1,
                ),
                'retrieve' => array(
                    'enabled' => 1,
                ),
                'update' => array(
                    'enabled' => 1,
                ),
                'delete' => array(
                    'enabled' => 1,
                ),
                'index' => array(
                    'enabled' => 1,
                ),
            ),
        ),
    );
    $endpoints[] = $endpoint;

    return $endpoints;
}

```

Notice that we don't return the endpoint as it is. But, as with views, we return an array containing the endpoint.

The client

Our client is quite trivial and will consist of one js file and one css file. I'm not going to write them both in their entirety here, but rather provide an excerpt that illustrates how you can communicate with a REST server using JavaScript. See [\[notes.js\]\(services-3.x-sample/blob/master/js/notes.js\)](#) and [\[notes.css\]\(services-3.x-sample/blob/master/css/notes.css\)](#) for the full versions.

```

// js/notes.js (excerpt)
// use an absolute URL path to prevent this from be local to the current page:
var noteapi = {

```

```

    'apiPath': '/js-api/note'
  });

  // REST functions.

  noteapi.create = function(note, callback) {
    $.ajax({
      type: "POST",
      url: this.apiPath,
      data: JSON.stringify({note: note}),
      dataType: 'json',
      contentType: 'application/json',
      success: callback
    });
  };

  noteapi.retreive = function(id, callback) {
    $.ajax({
      type: "GET",
      url: this.apiPath + '/' + id,
      dataType: 'json',
      success: callback
    });
  };

  noteapi.update = function(note, callback) {
    $.ajax({
      type: "PUT",
      url: this.apiPath + '/' + note.id,
      data: JSON.stringify({note: note}),
      dataType: 'json',
      contentType: 'application/json',
      success: callback
    });
  };

  noteapi.del = function(id, callback) {
    $.ajax({
      type: "DELETE",
      url: this.apiPath + '/' + id,
      dataType: 'json',
      success: callback
    });
  };

  noteapi.index = function (callback) {
    $.getJSON(this.apiPath, callback);
  };

```

Notice how we don't need to do anything odd to talk with our server. Everything maps to http verbs and a url, so there is no need for special client libraries.

The js and css is added in hook_init(), and will therefore be loaded on all pages in our Drupal install.

```
// noteresourcejs.module
```

```

/**
 * Implements hook_init().
 */
function noteresourcejs_init() {

  drupal_add_css(drupal_get_path('module', 'noteresourcejs') . '/css/notes.css');
  drupal_add_js(drupal_get_path('module', 'noteresourcejs') . '/js/notes.js');
}

```

By Example: Resource, Path, Arguments, Oy Vey!

Consider your Services API endpoint like a menu route. If you want a resource path like:

https://example.com/json_field/json_field/123/asdf

The API endpoint is "json_field" (the first one) and is configured in the .services.inc, the second "json_field" is the resource, defined in hook_services_resources, and the arguments are 123 and asdf, defined in hook_services_resources. The payload is still \$data, and is also defined in hook_services_resources. See example:

```

/**
 * Implements hook_services_resources().
 */
function json_field_services_resources() {
  /**
   * NOTE TO SELF:
   * When creating a new route, json_field.services.inc needs to be kept up to date.
   */
  return array(
    'json_field' => array(
      'update' => array(
        'help' => 'Update JSON Field',
        'file' => array('type' => 'inc', 'module' => 'json_field', 'name' => 'resources/update'),
        'callback' => 'json_field_resource_update',
        'access callback' => 'user_access',
        'access arguments' => array('access observer api'),
        'args' => array(
          array(
            'name' => 'id',
            'optional' => FALSE,
            'source' => array('path' => 0),
            'type' => 'string',
            'description' => 'The Entity ID whose field to update',
          ),
          array(
            'name' => 'field',
            'optional' => FALSE,
            'source' => array('path' => 1),
            'type' => 'string',
            'description' => 'The JSON field name to be updated',
          ),
          array(
            'name' => 'data',
            'optional' => FALSE,
            'source' => 'data',
            'type' => 'string',
            'description' => 'The complete, updated JSON array of values as {"1Z":1431983436,"1E":1432586713}.'.

```

```

    ),
    ),
    ),
    ),
);
}

```

The key is the 'source' in the 'args' array:

```
'source' => array('path' => 0),
```

"The source of this defined argument will be found in the 0th position in the path.

```
'source' => array('path' => 1),
```

"The source of this defined argument will be found in the 1st position in the path.

```
'source' => 'data',
```

"The source of this defined argument will be found as the payload (without a variable name)."

The callback function `json_field_resource_update()` defined in `./resources/update.inc` in the `json_field` module folder will have the following parameter structure:

```
function json_field_resource_update($entity_id = NULL, $field_name = NULL, $data = array()) {
```

Caveats / Troubleshooting

Given two REST Services APIs, even path-spaced accordingly (ie. `/apiv1` and `/api`), two resource names will conflict.

Ex. `/api/groups` and `/apiv1/groups`

Symptom: In the admin/structure/services/list/[api path]/resources config screen, one of the conflicting names may have "array()" instead of the endpoint description text. Odd behavior on the client end can be expected: ex. cannot authenticate to the new resource because the original one's authentication is actually not satisfied.

- [Getting Your First REST Server Working](#)
- [Preset Service Endpoint](#)

[< Working with REST Server](#)

[up](#)

[Getting Your First REST Server Working >](#)

Comments



Resource access callbacks must be a function

[gapple](#) he/they English *commented 11 years ago*

One gotcha that took me a while to figure out is that you must set the `access callback` value to a defined function. Unlike with `hook_menu()`, using a boolean value will not bypass the access check and instead results in a PHP error, as `services` passes the value straight through to `call_user_func_array()`.

`services.runtime.inc` - line 135

```

// Call default or custom access callback
if (call_user_func_array($controller['access callback'], $access_arguments) != TRUE) {
    // return 401 error
}

```

In addition, resource definitions are cached in the database, so if you're making modifications be sure to clear the cache or remove entries matching `services:%:resources` so that the new values in code are used.

[Log in](#) or [register](#) to post comments