Using Commit Hooks

Contents

Overview

Pre- and post-commit hooks are invoked before or after a riak_object is persisted and can greatly enhance the functionality of any application. Commit hooks can:

- allow a write to occur with an unmodified object
- modify the object
- Fail the update and prevent any modifications

Post-commit hooks are notified after the fact and should not modify the riak_object. Updating riak_objects in post-commit hooks can cause nasty feedback loops which will wedge the hook into an infinite cycle unless the hook functions are carefully written to detect and short-circuit such cycles.

Pre- and post-commit hooks are defined on a per-bucket basis and are stored in the target bucket's properties. They are run once per successful response to the client.

Configuration

Configuring either pre- or post-commit hooks is very easy. Simply add a reference to your hook function to the list of functions stored in the correct bucket property. Pre-commit hooks are stored under the bucket property *precommit*. Post-commit hooks use the bucket property *postcommit*.

Pre-commit hooks can be implemented as named Javascript functions or as Erlang functions. The configuration for each is given below:

Javascript: `{"name": "Foo.beforeWrite"}` Erlang: `{"mod": "foo", "fun": "beforeWrite"}`

Post-commit hooks can be implemented in Erlang only, and is described in more detail under Advanced Commit Hooks. The reason for this restriction is Javascript cannot call Erlang code and, thus, is prevented from doing anything useful. This restriction will be revisited when the state of Erlang/Javascript integration is improved. Post-commit hooks use the same function reference syntax as pre-commit hooks.

See Advanced MapReduce for steps to define your own pre-defined Javascript named functions.

Pre-Commit Hooks

API & Behavior

Pre-commit hook functions should take a single argument, the riak_object being modified. Remember that deletes are also considered "writes" so pre-commit hooks will be fired when a delete occurs. Hook functions will need to inspect the object for the *X-Riak-Deleted* metadata entry to determine when a delete is occurring.

Erlang pre-commit functions are allowed three possible return values:

- A riak_object – This can either be the same object passed to the function or an updated version. This allows hooks to modify the object before they are written.
- `fail` – The atom *fail* will cause Riak to fail the write and send a 403 Forbidden along with a generic error message about why the write was blocked.
- `{fail, Reason}` – The tuple `{fail, Reason}` will cause the same

behavior as in #2 with the addition of `Reason` used as the error text.

Errors that occur when processing Erlang pre-commit hooks will be reported in the `sasl-error.log` file with lines that start with “problem invoking hook”.

Erlang Pre-commit Example

Erlang

```erlang
%% Limits object values to 5MB or smaller
precommit_limit_size(Object) ->
  case erlang:byte_size(riak_object:get_value(Object)) of
    Size when Size > 5242880 -> {fail, "Object is larger tl
    _ -> Object
  end.
```

Javascript pre-commit functions should also take a single argument, the JSON encoded version of the riak_object being modified. The JSON format is exactly the same as Riak's MapReduce. Javascript pre-commit functions are allowed three possible return values:

- A JSON encoded Riak object – Aside from using JSON, this is exactly the same as #1 for Erlang functions. Riak will automatically convert it back to it's native format before writing.
- `fail` – The Javascript string “fail” will cause Riak to fail the write in exactly the same way as #2 for Erlang functions.
- `{"fail": Reason}` – The JSON hash will have the same effect as #3 for Erlang functions. Reason must be a Javascript string.

Javascript Pre-commit Example

Javascript

```javascript
// Makes sure the object has JSON contents
function precommitMustBeJSON(object){
  try {
    Riak.mapValuesJson(object);
    return object;
  } catch(e) {
    return {"fail":"Object is not JSON"};
  }
}
```

Chaining

The default value of the bucket *precommit* property is an empty list. Adding one or more pre-commit hook functions, as documented above, to the list will cause Riak to start evaluating those hook functions when bucket entries are created, updated, or deleted. Riak stops evaluating pre-commit hooks when a hook function fails the commit.

Example

Pre-commit hooks can be used in many ways in Riak. One such way to use pre-commmit hooks is to validate data before it is written to Riak. Below is an example that uses Javascript to validate a JSON object before it is written to Riak.

Javascript

```javascript
//Sample Object
{
  "user_info": {
    "name": "Mark Phillips",
    "age": "25",
  },
  "session_info": {
    "id": 3254425,
    "items": [29, 37, 34]
  }
}


var PreCommit = {
  validate: function(obj){

    // A delete is a type of put in Riak so check and s
    // operation is doing

    if (obj.values[[0]][['metadata']][['X-Riak-Deleted'
      return obj;
    }
```

```
      // Make sure the data is valid JSON
      try{
         data = JSON.parse(obj.values[[0]].data);
         validateData(data);

      }catch(error){
        return {"fail": "Invalid Object: "+error}
      }
      return obj;
    }
  };

  function validateData(data){
    // Validates that user_info object is in the data
    // and that name and age aren't empty, finally
    // the session_info items array is checked and valida
    // being populated

    if(
      data.user_info != null &&
      data.user_info.name != null &&
      data.user_info.age != null &&
      data.session_info.items.length > 0
    ){
      return true;
    }else{
      throw( "Invalid data" );
    }
  }
```

Post-Commit Hooks

API & Behavior

Post-commit hooks are run after the write has completed successfully. Specifically, the hook function is called by riak_kv_put_fsm immediately before the calling process is notified of the successful write. Hook functions must accept a single argument, the riak_object instance just written. The return value of the function is ignored. As with pre-commit hooks, deletes are considered writes so post-commit hook functions will need to inspect object metadata for the presence of *X-Riak-Deleted* to determine when a delete has occurred. Errors that occur when processing post-commit hooks will be reported in the `sasl-error.log` file with lines that start with "problem invoking hook".

Example

Erlang

```erlang
%% Creates a naive secondary index on the email field of a
postcommit_index_on_email(Object) ->
    %% Determine the target bucket name
    Bucket = erlang:iolist_to_binary([riak_object:bucket(Ol
    %% Decode the JSON body of the object
    {struct, Properties} = mochijson2:decode(riak_object:ge
    %% Extract the email field
    {<<"email">>,Key} = lists:keyfind(<<"email">>,1,Propert
    %% Create a new object for the target bucket
    %% NOTE: This doesn't handle the case where the
    %%       index object already exists!
    IndexObj = riak_object:new(Bucket, Key,<<>>, %% no obje
                               dict:from_list(
                                 [
                                  {<<"content-type">>, "tex
                                  {<<"Links">>,
                                   [
                                    {{riak_object:bucket(Ol
                                 ])),
    %% Get a riak client
    {ok, C} = riak:local_client(),
    %% Store the object
    C:put(IndexObj).
```

Chaining

The default value of the bucket *postcommit* property is an empty list. Adding one or more post-commit hook functions, as documented above, to the list will cause Riak to start evaluating those hook functions immediately after data has been created, updated, or deleted. Each post-commit hook function runs in a separate process so it's possible for several hook functions, triggered by the same update, to execute in parallel. *All post-commit hook functions are executed for each create, update, or delete*.

These May Also Interest You

- Five-Minute Install
- Advanced Secondary Indexes
- Replication Properties
- Advanced Commit Hooks
- Advanced MapReduce
- Advanced Search