Mary Jo Sminkey: Robust Error Handling Part 3 - Ajax

Posted At : September 8, 2017 4:52 PM | Posted By : Mark Kruger
Related Categories: ColdFusion

Part 3 in the super-guru's Mary Jo Sminkey's series on robust error handling. Enjoy!

So in my last post on error handling, I promised in this next one to cover logging of Ajax requests. As more and more of our site involves Ajax requests, this has become an important part of my error logging and debug work. If you want to know how to do this read on!

Ajax Error Handling

As with the page tracker, we're going to initialize an array for the tracker in onSessionStart( ):

```
session.ajaxTracker = [];
```

Now we'll add our new tracker method to the SessionManager.cfc. If you pass any secure data elements into Ajax calls, be sure to strip those out so they won't be included in the tracker data (or just modify them in some way if you do still want to get some info on them). Duplicate() is used so we don't change the argument scope itself but just make a copy of it, before deleting or changing the elements prior to logging. We typically pass in the method for the Ajax call as an additional argument that is used for this logging request (and delete it from the arguments struct copy so it won't show up twice). Also, since my code has an Ajax handler that will attempt 3 times to complete it if there's a timeout, I also include code to prevent each attempt from being logged by checking for a duplicate request (also prevents logging more than once if an impatient user tries to rerun the request).

```
<cfscript>public void function logAjax( numeric maxReqests=20){
var secureVars = 'password,newpassword,ccNumber,ccv';
var methodArgs = Duplicate(arguments);
var methodName = 'notLogged';
if (structKeyExists(methodArgs,"methodName")) {
    methodName = methodArgs.methodName;
structDelete(methodArgs,"methodName");
}
for (local.item in secureVars) {
    if (structKeyExists(methodArgs, local.item)) {
     methodArgs[local.item] = 'removed';
    }
}
var ajaxData = {     ajaxRequest = cgi.script_name,
methodName = methodName,
            methodArgs = methodArgs,
requestTime = dateFormat(Now())&'-'&timeFormat(Now()) };
// skip duplicate ajax requests
if ( arrayLen(session.ajaxTracker) IS 0
OR session.ajaxTracker[1].ajaxRequest IS NOT ajaxData.ajaxRequest
         OR session.ajaxTracker[1].methodName IS NOT ajaxData.methodName ) {

lock scope="session" type="exclusive" timeout="10" {
     if ( arrayLen(session.ajaxTracker) LT arguments.maxReqests ) {
```

```
                arrayPrepend(session.ajaxTracker, ajaxData);
             } else {
                arrayDeleteAt(session.ajaxTracker,arrayLen(session.ajaxTracker));
                arrayPrepend(session.ajaxTracker, ajaxData);
                }
             }
          }
}

</cfscript>
```

Since I use this logging method in most of my remote methods, it works best putting it into a Base.cfc component that the remote ones extend from. You can include other common things in a Base CFC like an onMissingMethod() to prevent errors from trying to call a method that doesn't exist. I'm using wirebox for persisting my CFCs and dependency injection.

```
<cfscript>

component name="RemoteBase" output="false" hint="Parent component for all remote methods" {

    variables.sessionManager = application.wirebox.getInstance("SessionManager");

    function init() {
        return this;
    }

    remote function onMissingMethod( missingMethodName, missingMethodArguments ) {
        return "onMissingMethod() called for method call - #arguments.missingMethodName#";
    }

    public void function logAjax( struct strArguments, string methodName='Unknown Method' ){
        strArguments.methodName = arguments.methodName;
        variables.sessionManager.logAjax(argumentCollection=strArguments);
    }
}
</cfscript>
```

So now to log the Ajax requests, we simply add this line of code to all the remote methods in our ColdFusion APIs (making sure those CFCs extend the above RemoteBase.cfc):

```
logAjax(arguments, "RemoteCartService - AddtoCart');
```

In some cases, we also want to log the Ajax response data as well, or at least whether it completed successfully. For instance, we tend to get a lot of emails from customers that are having problems logging into the site. We have 3 different sites that run off the same code base, and different messages are returned on login requests based on if the user didn't have an account yet, was using an account on the wrong website, entered a wrong password, etc. By including the response from the Ajax call, we can easily see what the issue with their login attempt was without having to dig into the database to see what status their account is in. So this additional method in the SessionManager.cfc can be used for adding the remote method response data to the log entry for the Ajax request. It will match up the name of the method given when logging the request to add additional information to a "response" key in the same object in the array.

```
<cfscript>
public void function logAjaxResponse(methodName, result){
if ( arrayLen(session.ajaxTracker) GT 0
AND session.ajaxTracker[1].methodName IS arguments.methodName ) {
lock scope="session" type="exclusive" timeout="10" {
        session.ajaxTracker[1].ajaxResponse = result;
    }
}
}
</cfscript>
```

And then we'll add this to the RemoteBase.cfc:

```
<cfscript>
public void function logAjaxResponse( string methodName='Unknown Method', any result = {} ){
    variables.sessionManager.logAjaxResponse(arguments.methodName, result);
}

</cfscript>
```

And finally add the call to this method with the response data prior to returning it to the Ajax call. Here's an example of a completed remote method:

```
<cfscript>
remote struct function checkAccountExists() {
    logAjax(arguments, 'RemoteCustomerService - checkAccountExists');
    var result = customerService.checkAccountExists(argumentCollection=arguments);
    logAjaxResponse('RemoteCustomerService - checkAccountExists', result);
    return result;
}
</cfscript>
```

Or if we just want to log when the request successfully is completed and not load all the response data into the session:

```
<cfscript>
remote function deleteProduct() {
    arguments.shoppingCartId = structKeyExists(cookie, 'shoppingCartId') ?
cookie.shoppingCartId : 0;
    logAjax(arguments, 'CartService - deleteProduct');
    var result = cartService.deleteProduct(argumentCollection=arguments);
    logAjaxResponse('CartService - deleteProduct', "success");
    return result;
}

</cfscript>
```

Now that we have both a page tracker as well as this Ajax tracker in our session, we may want to use it other than just for error messages. For instance, I drop this data into the emails that get sent to our support email account(s) from the site. This is one of the advantages of using a "Contact Us" page on your site that directs the user through an email form that can be sent to ColdFusion other than just posting an email address (reducing spam being another obvious one!) In addition to typical dumps of the user's browser data, cookie/customer information, etc. that I drop into this page, I include the page tracker and ajax tracker as well as an output of their current shopping cart and comprehensive system information obtained via the commercial BrowserHawk tool (https://www.cyscape.com/). With this level of information to help see what the actions the user was taking on the site prior to emailing us, often we can figure out and handle the problem even when the email itself from the customer has the (not uncommon) lack of detail about what they were doing when the problem occurred.

So back to error tracking. As I mentioned in my previous article, we don't use BugSnagHQ any more for error logging. Not that it isn't a great tool, but as we started doing more and more client-side JS code on our sites, logging of Javascript errors became more critical. I did some basic error logging with some simple tools for capturing and logging a global JS error (BrowserHawk actually now includes a JS library to include on your site for doing this) but I couldn't get enough information with the error in most cases to really be able to figure out the cause. When I went looking for something better, I found dozens of commercial offerings but one of the most common ones that came up as recommended was BugSnag (www.bugsnag.com).

After integrating it and trying it out for a short time, we knew this was the tool we were looking for and it was an easy decision to subscribe to a paid plan. Pricing starts at $9/seat per month for unlimited projects and logging, but is free for open source and nonprofit groups. BugSnag can log not just Javascript errors, but virtually ANY kind of error. It has notifiers available for JS as well as most popular languages like Ruby, PHP, Python, Java, .NET, Node.js, and even some popular applications like Wordpress as well as mobile apps on iOS and Android. In addition, you can send notifications for BugSnag error to multiple systems, so you can for instance post to a Slack channel or send a SMS in addition to the usual emails. Even more powerful is the ability to send BugSnag errors to a wide range of project tracking systems, like Jira, Bitbucket, Bugzilla, GitHub, Lighthouse, Unfuddle, and many more. You can of course always build your own notifications particularly for any system that has a REST API to tap into. As with emails, adding tickets is very customizable so that for instance you only log the first instance of a particular bug.

So let's look at logging our ColdFusion errors in BugSnag. First, we have to set up a project. Since there's no built-in option for CF, I used the Server-Other option. Once you create your project, you'll have your API key for sending notifications.

To log your errors from ColdFusion, there are of course numerous ways to do this, I use a CFC I built specifically for BugSnag, that using wirebox I can inject as needed into other components as well as use in the global error handler that I've created. It takes information about the error, browser data, our page and Ajax tracker objects, and any available user information. Here's the code for this CFC. You'll notice when instantiated we set a "site version" and "environment" as well as the other BugSnag specific settings. This allows us to filter our local, dev and staging errors from production and also filter errors on a specific release version, an application setting that we update with production releases. Also note the format used for the ajax and page trackers. We need to convert them to structs to ensure that BugSnag will give them their own tabs on the Detailed error view. The 'notifier' can just be set to something like "BugSnag ColdFusion".

```
<cfscript>
```

```
component name="BugSnag" accessors="true" output="false" {

    property string apiKey;
    property string notifier;
    property string version;
    property string url;
    property string siteVersion;
    property string environment;

    function init( required string apiKey,
required string notifier,
             required string version,
required string url,
             required string siteVersion,
required string environment ) {

        setApiKey(arguments.apiKey);
        setNotifier(arguments.notifier);
        setVersion(arguments.version);
        setURL(arguments.url);
        setSiteVersion(arguments.siteVersion);
        setEnvironment(arguments.environment);

        return this;
    }

    function sendLog( required string errorType,
required string message,
required string page,
                required string osVersion,
required string hostname,
                struct user = {},
array ajaxTracker = [],
array pageTracker = [],
struct scope = {},
                array stacktrace = [] ) {

        var payload = {
     'apiKey' = getApiKey(),
     'notifier' = {
     'name' : getNotifier(),
     'version' : getVersion(),
     'url' : getURL()
     },
     'events' = [{
         'payloadVersion': "2",
          'exceptions' = [{
     'errorClass' : arguments.errorType,
     'message' : arguments.message,
     'stacktrace' : []
     }],
     'context': arguments.page,
         'app': {
             'version': getSiteVersion(),
             'releaseStage': getEnvironment()
         },
         'device': {
         'osVersion': arguments.osversion,
                   'hostname': arguments.hostname
         },
          'metaData': {
              'pageTracker': { 'Page Tracker' = arguments.pageTracker },
              'ajaxTracker': { 'Ajax Requests' = arguments.ajaxTracker },
'scopeData' : arguments.scope
          }
         }]
     };
     if (structKeyExists(arguments.user,"id")) {
     payload.events[1]['user'] = {
              'id': arguments.user.id,
              'name': arguments.user.name,
              'email': arguments.user.email
         };
     } else {
         payload.events[1]['user'] = {
              'id': cgi.REMOTE_ADDR
         };
     }
    for (local.s = 1; local.s LTE arraylen(arguments.stacktrace); local.s++ ) {
     var trace = {
     'file' : arguments.stacktrace[local.s].template,
     'lineNumber' : arguments.stacktrace[local.s].line,
     'columnNumber' : arguments.stacktrace[local.s].column,
     'method'         : "cfml"
     };
     arrayAppend(payload.events[1].exceptions[1].stacktrace, trace);
     }

     cfhttp(url="https://notify.bugsnag.com",result="local.result",method="post") {
         cfhttpparam(type="header", name="Content-Type", value="application/json");
         cfhttpparam(type="body", value=serializejson(payload) );
        }
    }

}

</cfscript>
```

Now it's time to go back to the error handler and make some updates for using BugSnag. First, we want to add a new variable for the "stacktrace" that we send as part of the error data. For a global thrown error, this gets set to "error.rootcause.TagContext" and for caught (try/catch) errors it gets set to "cfcatch.TagContext". For our custom errors, we'll use a custom variable, using a ternary operator to see if it was set when the error handler was called (localVars is the struct we use for all variables in the error handler template, that gets stripped out when outputting data):

```
localVars.stacktrace = structKeyExists(request,"stackTrace") ? request.stackTrace :[] ;
```

I also like to use a variable in my error handler called "addTicket" which allows me to programmatically decide when I'm going to send data to BugSnag. By default I log any errors that I already send by email (which is one copy of each error every 4 hours, and also for every 25 copies of the same error that occur). However, I may want to also log errors that I don't send by email... typically common errors that I can't fix and don't need cluttering up my email box, like database timeout messages. I still want to log those when they occur so I can keep track of how often they occur. I also want to set my error handler to log ALL errors that occur during user login and checkout attempt, since these are the most common reasons users will email us about site problems and I want to log anything that might have happened for them. Here's how I do that as well as include the CF page that the error occurred on in the error name:

```
<cfscript>
if ( structKeyExists(localVars.errorData, "TagContext") AND
isArray(localVars.errorData.TagContext) AND arrayLen(localVars.errorData.TagContext) IS NOT
0 ) {
    localVars.template = localVars.errorData.TagContext[1].Template;
    localVars.errormess &= ": #localVars.template#";
    if ( FindNoCase('/login', localVars.template) OR
findNoCase("/shopping",localVars.template) ) {
        localVars.logAllErrors = true;
    }
}

</cfscript>
```

So once we have collected and sanitized all the scope data, we're ready to send the notification to BugSnag. We want to include the full file URL where the error occurred, which will get included as a clickable link in the BugSnag console, as well as any information we have about the user - their browser, OS, and account information for our site. This information can be used to filter and search errors in BugSnag.

```
<cfscript>
if (localVars.addTicket ) {
    errorPage = "#LCase(ListFirst(cgi.server_protocol,
"/"))#://#cgi.server_name##cgi.script_name##cgi.path_info#";
    if ( cgi.query_string IS NOT "" ) {
        errorPage &= "?#cgi.query_string#";
    }
    bugsnag = application.wirebox.getInstance("BugSnag");
    bugsnag.sendLog( errorType=request.errorType,
            message=localVars.errormess,
            page=errorPage,
            osVersion=cgi.http_user_agent,
            hostname=hostaddress,
            user = { id = session.customerId,
name = structKeyExists(cookie, "CUSTOMERNAME") ? cookie.CUSTOMERNAME :
'',                email = structKeyExists(session, "email") ? session.email : ''},
ajaxTracker = session.ajaxTracker,
            pageTracker = session.pageTracker,
            scope = localVars.strScopes,
            stacktrace = localVars.stacktrace );
    }
</cfscript>
```

So here's our new BugSnag Dashboard showing our logged CF errors which we can filter by the stage (environment), version, IP address, user email, customer ID, error message, context and more. You can select errors to mark as fixed, or ignore, send to your ticket system, assign to a user, etc.

Click to view the detailed view, and here we find all the data we so carefully collected and logged with the error. You can see that BugSnag puts the Page Tracker (and Ajax Tracker if it has data) on the custom tab and the full scope dump on a tab of its own, as is the User data. You can add your own custom elements to the data logged as well if you want. Note that BugSnag doesn't do any formatting of the data, it just outputs the raw JSON data. I've come up with a neat solution to this which allows me to include a URL in the error reports that will open up a separate page with a nice formatted collapsible tree view of the full scoped data structure. You'll have to wait for another blog article on how I do this... as well as my custom error logging for Ajax and other JS errors to BugSnag.

That's it for the ColdFusion error logging for now. Hopefully you've gotten some good ideas on how to improve your own error handling. I'm sure I'll continue to improve on mine... next item on the list is to redo the entire error handler code into script (I really hate tag syntax!) But I'm usually too busy fixing errors and finding even better ways to report them!