METAPROGRAMMING IN COLDFUSION

By Andy Hill

ABOUT ME

- Senior Systems/Analyst at Indiana University as well as freelance work
- Maintain multiple web sites in ColdFusion, PHP, and SharePoint
- Graduated from I.U. with B.S. in Computer Science in 2003
- All about CFSCRIPT

WHAT IS METAPROGRAMMING?

According to Wikipedia,

> *“ **Metaprogramming** is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime. ”*

METAPROGRAMMING TO ME

Code that inspects or maniplulates its environment at compile or run time.

- Reduce redundant code
- Increase flexibility
- Extend a component with only the functionality you need

OVERVIEW

We'll look at the following in ColdFusion:

1. Evaluate()
2. Introspection
3. Function Passing
4. Overloading
5. Code Generation
6. OnMissingMethod()
7. Mixins

EVALUATE()

According to CF docs, **Evaluate**:

“ Evaluates one or more string expressions, dynamically, from left to right. (The results of an evaluation on the left can have meaning in an expression to the right.) Returns the result of evaluating the rightmost expression. ”

EVALUATE() (CONTINUED)

- The example Adobe gives,

```
Evaluate("qNames.#colname#[#index#]");
```

  I would just write as

```
qNames[colname][index];
```

- If you do use it, be sure to sanitize any user data before passing to Evaluate()
- Not nearly as capable as Ruby's eval()
- I haven't used recently, so my example is contrived

INTROSPECTION

From **Wikipedia**:

“ In computing, type introspection is the ability of a program to examine the type or properties of an object at runtime. Some programming languages possess this capability. Introspection should not be confused with reflection, which goes a step further and is the ability for a program to manipulate the values, meta-data, properties and/or functions of an object at runtime. ”

INTROSPECTION (METHODS)

- WriteDump/CFDump to inspect the returned metadata
- GetMetaData()
- CF10:
  - GetApplicationMetadata()
  - SessionGetMetaData()
  - CallStackGet()

GETMETADATA: NATIVE CF VARIABLES

- Returns an instance of Java.lang.class
- Can use any methods of that class on the metadata object
- Probably not very useful unless using other Java classes

GETMETADATA: FUNCTIONS

- Depends on how explicit your function declarations are.
- Struct keys:
  - Name: Name of the function
  - Parameters: Array of function parameters. Each array entry is a struct with 'name' and 'required' fields, other cfargument attributes returned if declared
  - Access: Not defined by default (public/private)
  - Return Type: Not defined by default
  - Any other attributes to cffunction

GETMETADATA: COMPONENTS

- By default, has 'fullname', 'name', 'path', 'type', and an 'extends' struct with the same fields for the component's parent
- If there are any methods defined, returns 'functions', an array of structs with metadata for each function
- Returns any other defined cfcomponent attribute

CF10

- GetApplicationMetadata(): Application (timeout, session, client, cookie, etc. settings)
- SessionGetMetaData(): Only current key is 'starttime'
- CallStackGet(): How did I get here? Works anywhere. Array of structs representing the call stack (from most recent back to application.cfc.). Struct keys are 'Function', 'LineNumber', and 'Template.'

FUNCTION PASSING

- Pass a function (callback) to another function as an argument
- CF10 supports inline functions (a subset of **closures**), making traditional function passing somewhat irrelevant, but still valuable if passing the same function to more than one function.
- CF10 also supports passing functions as arguments to native functions such as ArraySort() for custom sorting

CF9

```
d = { 'one'='a', 'two'='b' };
string function serializer(data, func) {
        return func(data);
}
//// helpers
string function data2json(required any data) {
        return serializeJSON(data);
}
string function data2xml(required any data) {
        var toxml = new toxml(); //// toxml.cfc from
raymondcamden.com
        if (isQuery(data)) return toxml.queryToXML(data, 'items',
'item');
        else if (isArray(data)) return toxml.arrayToXML(data, 'ite
ms', 'item');
        if (isStruct(data)) return toxml.structToXML(data, 'items'
, 'item');
        else return toxml.listToXML(data, 'items', 'item');
}
//// returns {"one":"a","two":"b"}
jsonstr = serializer(d, data2json);

//// returns <?xml version="1.0" encoding="UTF-8"?><items><item><o
ne>a</one><two>b</two></item></items>
xmlstr = serializer(d, data2xml);
```

CF10

```
serializer(d, function(data) { return serializeJson(data); });

//// CF ArraySort with custom sorting function
//// from raymondcamden.com
arraySort(data, function(a,b) {
        //remove the The
        var first = a;
        var second = b;

        first = replace(first, "The ","");
        second = replace(second, "The ","");

        return first gt second;
});
```

CURRYING

```
function function serializeCurry(required function func) {
        return function(required any data) {
                return func(data);
        };
}

xmlSerializer = serializeCurry(data2xml);
jsonSerializer = serializeCurry(function(data)  {
        return serializeJSON(data);
});

//// returns xml
xmlSerializer(d);
//// returns json
jsonSerializer(d);
```

OVERLOADING

- Overloading is having several methods with the same name which differ from each other in the type of the input and/or output of the function.
- In strongly typed languages, such as Java or C#, overloading is accomplished by having multiple functions with the same name with different method signatures
- In ColdFusion and other weakly typed languages, this is accomplished via type-sniffing and optional arguments
- jQuery: $(obj).html() versus $(obj).html('Hello World!')

OVERLOADING EXAMPLE

```
public any function query(required string sql, any args, string ds
n=This.dsn) {
                //// query without arguments, default dsn
                if (!IsDefined('args')) {
                        args = This.dsn;
                }
                ////query with arguments
                if (isArray(args)) {
                        return This.safeQuery(sql, args, dsn);
                ////query without args
                } else {
                        dsn = args;
                        return This.safeQuery(sql, [], dsn);
                }
        }

db.query('select * from my_table');
db.query('select * from my_table where id=?', [ 5 ]);
db.query('select * from their_table', 'their_dsn');
db.query('select * from their_table where id=?', [ 5 ], 'their_dsn
');
```

CODE GENERATION

- Write out code to a file (.cfc, .cfm, .js, .css)
- Execute it

EXAMPLE: GETDSOBJECT()

- Used in determining the correct cfsqltype for cfsqlparam
- Checks if there is a .cfc file for the requested table in the specified directory (e.g., /cfc/dbtables/tablename.cfc)
- If not,
  - Query the database for the table metadata (e.g., information_schema.columns)
  - Generate code for a .cfc file, which associates column names with their types
  - Write code to .cfc file
- Instantiate and return the object (e.g., return createObject('component', 'cfc.dbtables.tablename');

ONMISSINGMETHOD()

- Called when a component method is called which does not exist
- Ruby's version, method_missing, uses it for its Active Record model
- I would imagine something similar is used for CF ORM (Hibernate)
- The idea being that methods like get{columnName}() is not manually written, but generated on the fly based on the object's knowledge of the table metadata

MIXINS

- Add functionality to component at runtime
- Mixins can assume variables/methods of parent will be there

CODE

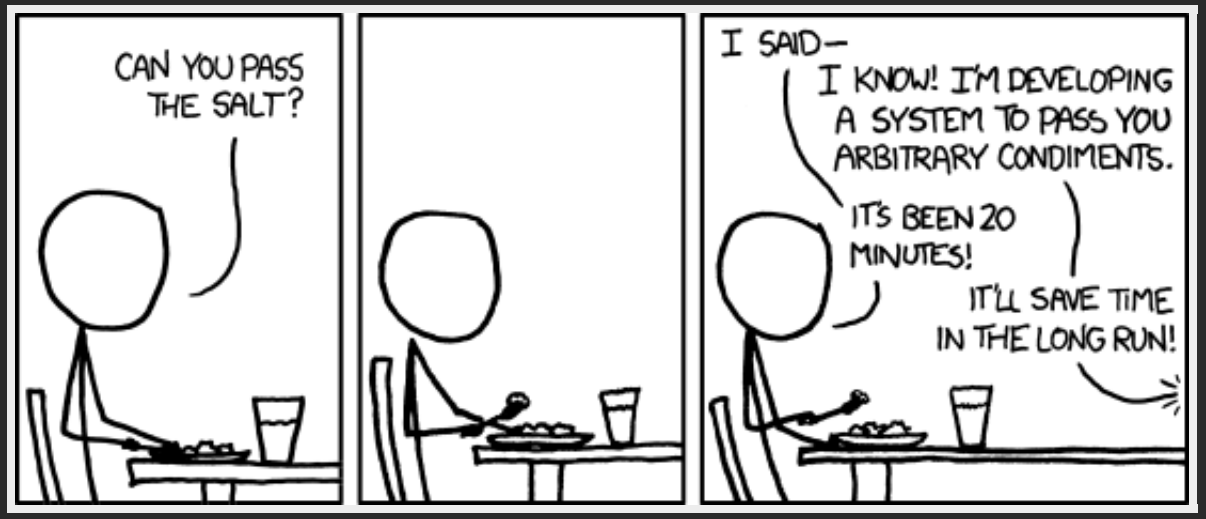
```
//// Modified from corfield.org
public function mixin(type) {
        var target = createObject("component",arguments.type);
        structAppend(this,target);
        structAppend(variables,target);
        if (structKeyExists(target, 'getVariables')) {
                structAppend(variables, target.getVariables());
        }
}

//// Add to mixin to allow access to variables scope
//// However, this will expose variables scope
public struct function getVariables() {
        return variables;
}
```

REVIEW

- Introspection: Use metadata from your programming environment
- Function Passing and Currying: Pass functions by name or inline to add custom behavior (e.g., sorting, parsing)
- Overloading: Before writing a similar function, consider adapting the function based on parameter types
- Code Generation: Generate code files on the fly for custom behavior
- OnMissingMethod(): Groups common method behavior by name to avoid repetitive code
- Mixins: Extend component at run time based on just the extended behavior needed

WRAP UP



via **xkcd**

GitHub: **https://github.com/athill/cfmeta/**

Slideshow engine: **Reveal.js**

Questions? **andy@andyhill.us**