

## ***Beans, Beans, the Musical Fruit***

by Hal Helms and Ben Edwards

The Mach-II framework for web applications (version 1.0.7) introduced built in support of *beans*. This article looks at what beans are and how they should be used.

In class-based object orientation, the type employed by both Java and CFCs, the application is modeled as a series of classes. Classes are an encapsulation of both data and methods. A bean is simply a specialized kind of object, one that maintains strict rules for the accessing of data. If the name, *bean*, seems an odd choice for a programming construct, the blame must be layed on that period of time shortly after Java became popular when all things Java-related were given coffee-related names. The idea of beans comes from the Java world, as does the odd name.

### **Bean Best-practices**

Beans are primarily carrier objects, used for passing encapsulated data between application layers (model-view-controller or architectural tiers). They typically contain minimal business logic (if any), and they have simple, consistent interfaces.

For a class to be considered a bean, it must have methods for getting and setting the values of its data members. Bean getters and setters (also called *accessors*) are formalized so that for each data member, **xyz**, there exists a **getXyz()** and a **setXyz()** method. Here, for example, is a bean CFC with a single data member, `taxRate`, along with the requisite accessors:

```
<cfcomponent>
  <cfset variables.taxRate = 0 />

  <cffunction
    name="getTaxRate"
    access="public"
    returntype="numeric"
    output="false">
    <cfreturn variables.taxRate />
  </cffunction>

  <cffunction
    name="setTaxRate"
    access="private"
    returntype="void"
    output="false">
    <cfargument name="taxRate" type="numeric" required="true" />
    <cfset variables.taxRate = arguments.taxRate />
  </cffunction>
```

`</cfcomponent>`

As you can see from this example, the **access** attributes for getters and setters can be determined by the bean writer. In the case of bean CFCs, the possible values would be **public**, **private**, **package**, or **remote**. Regardless of the access value, the data member (**taxRate** in our example) should be private—which we achieve by assigning it to the variables scope. This ensures that the only way to gain access to the data is by use of an accessor. Aside from this rule, there are other best practices that should be adhered to for consistency. These include:

- Bean data members/properties should begin with a lower case letter and should use mixed-cased format. Examples of properly named properties include **taxRate**, **length**, and **companyName**.
- Setters should accept a single argument bearing the same name and type as the data member itself. See the example above for an illustration of this.
- Getters accept no arguments since their job consists simply of returning a data member's value.
- Beans may have other beans as data members. For example, a **Company** bean might have an **Address** bean as a data member.
- All bean CFCs should have an `init()` function that has no required arguments and returns the bean itself (`this`). The `init()` function may accept arguments, but they should not be required. The `init()` function should use setters to set any data passed to `init()` in the bean.
- All access to data members—even by other non-accessor methods within a bean—should be done by means of an accessor method. Here is a method, **computeTax()**, that accepts a **Product** bean and uses its own **getTaxRate()** method to compute the appropriate tax:

`<cffunction`

```
    name="computeTax"
```

```
    access="public"
```

```
    returntype="numeric"
```

```
    output="false">
```

```
    <cfargument name="product" type="Product" required="true" />
```

```
    <cfreturn arguments.product.getPrice() * getTaxRate() />
```

`</cffunction>`

Methods other than accessors may begin with **get** and **set**. We could, for example, have renamed `computeTax()` to `getTax()`. Even with the name change, the method is not a “getter” as it does not follow the requisite format.

While we consider the above to be essential to writing well-formed beans, we only (highly) recommend the practice of grouping getters and setters for a single data member together in code. This would mean that **getTaxRate()** and **setTaxRate()** should be grouped together in code as opposed to grouping all getters together and all setters together.

Why the call for rigor in consistency in writing beans? Consistency in code saves developers valuable time and effort. In the case of beans, a consistent approach ensures that you and others will be able to concentrate on the important task at hand. Put another way, it saves brain cycles (similar to CPU cycles!) for more important work.

A full bean example:

```

<cfcomponent
  displayName="Address"
  hint="An address bean.">

  <!--- PROPERTIES --->
  <cfset variables.street = "" />
  <cfset variables.city = "" />
  <cfset variables.state = "" />
  <cfset variables.postalCode = "" />

  <!--- CONSTRUCTOR --->
  <cffunction name="init" access="public" returnType="Address"
output="false">
    <cfargument name="street" type="string" required="false" default="" />
    <cfargument name="city" type="string" required="false" default="" />
    <cfargument name="state" type="string" required="false" default="" />
    <cfargument name="postalCode" type="string" required="false" default=""
/>

    <cfset setStreet(arguments.street) />
    <cfset setCity(arguments.city) />
    <cfset setState(arguments.state) />
    <cfset setPostalCode(arguments.postalCode) />

    <cfreturn this />
  </cffunction>

  <!--- GETTERS/SETTERS --->
  <cffunction name="getStreet" access="public" returnType="string"
output="false">
    <cfreturn variables.street />
  </cffunction>
  <cffunction name="setStreet" access="public" returnType="void"
output="false">
    <cfargument name="street" type="string" required="true" />
    <cfset variables.street = arguments.street />
  </cffunction>

```

```

    <cffunction name="getCity" access="public" returnType="string"
output="false">
        <cfreturn variables.city />
    </cffunction>
    <cffunction name="setCity" access="public" returnType="void"
output="false">
        <cfargument name="city" type="string" required="true" />
        <cfset variables.city = arguments.city />
    </cffunction>

    <cffunction name="getState" access="public" returnType="string"
output="false">
        <cfreturn variables.state />
    </cffunction>
    <cffunction name="setState" access="public" returnType="void"
output="false">
        <cfargument name="state" type="string" required="true" />
        <cfset variables.state = arguments.state />
    </cffunction>

    <cffunction name="getPostal Code" access="public" returnType="string"
output="false">
        <cfreturn variables.postal Code />
    </cffunction>
    <cffunction name="setPostal Code" access="public" returnType="void"
output="false">
        <cfargument name="postal Code" type="string" required="true" />
        <cfset variables.postal Code = arguments.postal Code />
    </cffunction>

</cfcomponent>

```

## Mach-II Bean Support

Version 1.0.7 of the Mach-II framework similarly saves you time and effort by supporting bean creation and population. This is accomplished with a new XML tag, **<event-bean>**. This tag, a sub-element of the **<event-handler>** tag, has the following attributes:

- **name**: the name of the bean to be created
- **type**: the CFC bean to be used
- **fields**: the event fields to be used to populate the bean's data members (optional).

Suppose that we have a bean CFC, **Address**. The properties for this method might be **street**, **city**, **state**, and **postalCode**. Since it is a well-formed bean, we must have a **getStreet()**, **setAddress()**, **getCity()**, **setCity()**, **getState()**, **setState()**, **getPostalCode()**, and **setPostalCode()**.

Let's further suppose that we have a form on a web page that asks for this information. When the form is submitted to our Mach-II application, we wish to create a new **Address** bean and populate it with information from the form. The resulting bean should be placed in the event arguments. We could, of course, write code in a listener to create the bean and populate its data members, but the **<event-bean>** tag makes it even simpler, so long as the form field names match those of our bean:

```
<event-bean
  name="address"
  type="model.Address"
  fields="street, city, state, postalCode" />
```

Mach-II will create a bean of type, **model.Address**, and will call the setters for each of the data members specified, using the matching event variables as arguments for these setters.

Let's say that we have another bean, **Registrant**, that has properties of **firstName**, **lastName**, and **address**. While the data type of the first two properties are strings, **address** points to an **Address** bean. Here's how our XML would appear:

```
<event-handler event="register">
  <event-bean
    name="address"
    type="model.Address"
    fields="street, city, state, postalCode" />
  <event-bean
    name="registrant"
    type="model.Registrant"
    fields="firstName, lastName, address" />
  ...
</register>
```

The variables, **firstName** and **lastName**, will have already been placed in the event's argument collection. Once the bean, **address**, is created (through the use of an **<event-bean>**), it will now be accessible to be used in the creation of the bean, **registrant**. The register event will now have two beans, **address** and **registrant**, which used in business logic as encapsulated units, rather than a loose collection of fields.

Also, the **<event-bean>** tag can populate a bean using its **init()** function. By not specifying the **fields** attribute, when the bean is created the event arguments will be passed as arguments to the bean's **init()** function. This allows us to have the framework populate the bean with a single method call and, optionally, place additional initialization logic in the bean.

Beans are a simple but powerful tool, but their power depends on the consistency with which they are used. This article shows the proper use of beans and explains how Mach-II makes the creation and use of beans easier still.