Framework Concepts

Framework Concepts

The Fluid (Infusion) Framework is built on a number of important concepts. This page outlines some of the fundamental design goals of Infusion, providing terminology to help clarify these goals. Descriptions elsewhere in the documentation may also make use of particular terms in certain ways, and this page collects together the conceptual and background knowledge you might need to help you interpret these descriptions.

On This Page

See Also

- Infusion Framework
- Fluid Component API
- Events for Component Developers

Model Objects

The most overriding goal of the framework is *transparency*—data is pure data, and markup is pure markup. Infusion doesn't impose any inheritance hierarchies or strict contracts on how you structure your data model.

In Infusion, a tree of objects has the status of being a Model Object if it can successfully be copied by the framework utility function `fluid.copy()`. This function duplicates an object tree in a very clear way: each object is cloned recursively, except for immutable values such as strings, numbers, functions, etc. which are copied across. If everything that is important to you about the object has been successfully cloned, then it is a suitable model object.

Note that this does not prohibit functions, DOM nodes, etc. from appearing in a model, just so long as the meaning of the object tree is not destroyed through the operation of `fluid.copy()`. The point here is that Infusion doesn't impose a formal contract on model objects: as long as `fluid.copy()` succeeds, you're good. For reference, the current implementation of `fluid.copy()` simply forwards the operation to `jQuery.extend(true, ...)`. In practice, your model objects will often be JSON-compatible data suitable for sharing between your JavaScript code and other services on the Web.

An analogy from another environment, Java, is a POJO or "Plain Old Java Object". The J could just as easily be substituted for "JavaScript" in this case.

EL Paths

In some parts of the framework, we refer to "EL expressions". This is a somewhat historical phrase that perhaps sounds like it says more than it is trying to. All we mean by EL expressions are dot-separated paths built of names—for example if you had defined an object `zar` with a member `boo` which has a member `baz`, you could access the nested Javascript property by writing the expression `zar.boo.baz`. Now, as a string, this expression becomes an EL expression—that is, the string `"zar.boo.baz"` is an EL expression which designates the same piece of data. This is useful because it abstracts references to pieces of data from the actual data itself. It is possible to replace one object tree with another, but still to maintain a stable reference to the same subproperty `baz`, whoever it happens to be today. This is particularly important in web applications where data claiming to be "your data" can suddenly arrive from anywhere (a JSON feed, some persistence, a particularly aggressive version management system, etc). However it got here, you know it is your data because it is at the right path.

EL expressions within Fluid can be evaluated by the framework utilities `fluid.model.getBeanValue()` and `fluid.model.setBeanValue()`, and also global functions can be similarly invoked by `fluid.invokeGlobalFunction()`. For more information about these functions, see Framework Functions. EL path expressions of this sort are fundamental to Fluid's model-oriented thinking, and the operation of the Infusion ChangeApplier.

Events

Events in Infusion are model-oriented, and aren't specific to the DOM. Much like every other concept in Infusion, Fluid Events are really not anything special at all—an event here is really just another kind of function call. There is no special kind of "Event Object" that gets handed around to event listeners, and anyone can easily define a new event by simply calling `fluid.event.getEventFirer()`. In practice, many events are automatically instantiated for you as a result of the initialisation of Fluid Components. Just by writing the event name in a component's options structure, an event type is auomatically created. No code required.

Fluid Events are so close to the language that they are a suitable replacement/implementation for features found in other frameworks, such as the "delegates" found in Mac OS X's Cocoa environment. In Infusion, delegates can be implemented simply as a unicast event. Since event signatures are completely free (free as in "like a bird", not like either beer or speech), any function can potentially actually be an event listener. It is all a matter of perspective.

MVC

Fluid infusion is not formally a Model-View-Controller framework, although it embodies the separation of concerns that MVC implies. The weak link in MVC is the controller layer. Controllers are typically conceived of as the "glue" of an application: all the code that connects models and views together. Controllers are often the most brittle and least reusable part of an architecture. There's no controller layer in Infusion; Fluid takes the approach that the glue should be taken care of for you. Infusion applications consist, as much as possible, of pure view and model code. Even event binding itself can be performed declaratively, eliminating yet another source of controller code.

Infusion has a clear concept of a view: the Component. As mentioned above, models are central to the framework as well. The overall design of Fluid is aimed to reduce to zero the code at the controller level of an application. Instead, Infusion emphasizes declarative configuration, the powerful and flexible #Events system, and the automated approach to data binding enabled by the #EL system and the ChangeApplier.

Declarative Configuration

Less code is better. If you can represent aspects of your application as data rather than as imperative logic, there is a clear benefit to application design: data is transparent, and is easier to operate on and transform with algorithms. It's also easier to understand by both code in the application and by humans reading the design. In order to understand a data-oriented component, one merely needs to understand the layout of the data it works with, rather than looking at its implementation or understanding all the details of the contract behind its API documentation.

Standard conventions also help simplify your application architecture. Wherever possible, Infusion applies a standard layout to commonly appearing options for a component. For example, there is a common convention for declaring a component's events, selectors which bind to markup for both styling as well as function, and relationship to its model. Each component is also supplied with a set of defaults which also take the form of declaratively-specified model data.

The Fluid Renderer takes the concept of declarative configuration to a level not achieved with other frameworks. The renderer reduces the entire binding and controller function of an application to a declarative form, a component tree. In this form, the user interface is extremely amenable to inspection, interpretation and re-processing.

Historically, the desire to be able to treat logic as data has strong roots, for example in the LISP community. However, where *all* application code is on a common footing, designs become tangled and hard to interpret. By providing domain-specific forms for carefully selected parts of an application's functionality, typically at the fluid:Controller level, the complexity of code operating on this data can be reduced and transparency increased. It is a productive middle ground, between all application code becoming a candidate to be data (as in LISP), and none of it (as in Java).

IoC

IoC stands for *Inversion of Control*, the traditional name given to this programming concept by its early promoters, Ralph Johnson and Martin Fowler. IoC is actually the technique that naturally results when trying to make sure that dependencies in a codebase are correctly organised. The goal is to eliminate cycles of knowledge and points of dependency weakness within an application architecture.

IoC is a crucial mechanism of avoiding tight binding between framework components in Infusion. Whilst we do not yet have a full formal IoC container in the framework, most of the essential mechanisms are already there as part of Fluid's Subcomponent system. The subcomponents for a top-level component can be expressed in a fluid:declarative structure, where a tight binding is avoided between component and subcomponent implementations. This enables a more flexible design, including the ability to swap out the implementation of one subcomponent for another. We accomplish this loose-coupling through the use of #EL to hold the name of the subcomponent to be instantiated, rather than requiring the user to instantiate the subcomponent themselves directly in code.

This "don't call us, we'll call you" approach to instantiation is one of the cornerstones of IoC programming. Instantiation is performed by the framework after it has performed all dependent lookups, rather than being in the hands of users whereby they may get themselves into dependency tangles and requiring other components to have too much knowledge of the other parts of the system.

This somewhat vague-seeming description is an attempt to codify a way of thinking that can really only be internalised by repeated (and sometimes painful!) experience. Those unfamiliar with the benefits should read around some of the links above, and also experiment with the relationships between some framework components and their subcomponents, particularly the Decorators such as the pairing of Inline Edit and Undo.

Markup agnosticism

Markup agnosticism is ubiquitous throughout Infusion. We don't bake in assumptions about the nature and structure of a user interface's markup, recognizing that components need to be customized and adapted for different context.

Markup agnosticism reflects the same basic stance that leads to #IoC, #Declarative Configuration and fluid:Model Transparency: effective organisation of dependencies. In practice, users of a framework want to be able to use their own markup and customize the look and feel of existing components. Virtually every other client-side framework ships their widgets as "black boxes:" markup is baked into code and is invisible to the user. This makes it very difficult to customize the widget without cracking open the code and forking it.

The framework DOM Binder, a universal service which is supplied to every Fluid Component, is the primary vehicle for delivering markup agnosticism. The DOM Binder allows users to supply their own jQuery selectors to configure a component, allowing them to change the markup and inform the component of how to find important things in the DOM. This entirely removes the hard-baked assumptions about markup within a component, allowing the user to control how it looks and is structured.

Infusion's other tool for DOM agnosticism is the Fluid Renderer. By abstracting the entire process of dynamic markup generation behind a purely data-oriented component tree, the field is left clear for 100% control of markup down to the tag level by component users. An important exhibition of this control is in the Pager component. Other client-side table controls tend to impose a model where one logical table row corresponds to a <tr> tag, and one table cell corresponds to a <td> tag. The Infusion Pager, on the other hand, breaks down this association and allows a flexible association of markup to data. The Pager addresses the universal case of "regularly repeating markup," so that the association to <tr> and <td> may be one to many, many to one. Entirely different tags can even be chosen to render the "table." All layout could, for example, be performed in CSS.