Component Lifecycle and autoInit

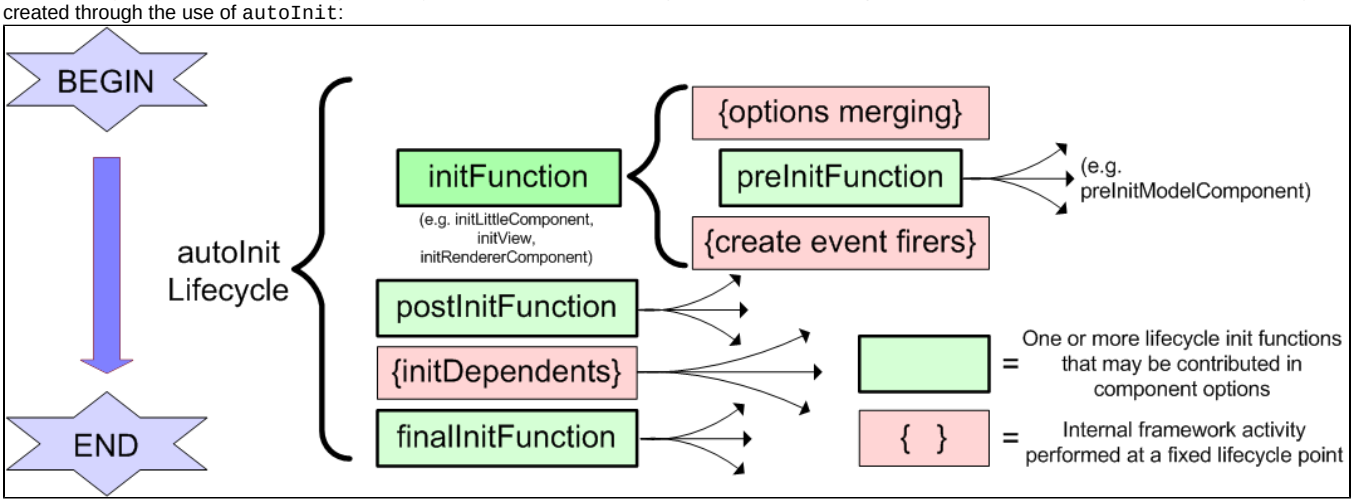
New in v1.4

Basic Component Lifecycle

The basic component lifecycle (as per older versions of the framework) is described at Component Lifecycle. The information held there is still accurate and the techniques described there will continue to be supported at least until Infusion v2.0 and probably much beyond. However, new in the 1.4 version of Infusion is a new selection of directives and a wider lifecycle that accompanies the introduction of the new Inversion of Control (IoC) system. This wider lifecycle, which is activated by supplying the Grade name of `autoInit`, comprises the lifecycle described by `fluid.initView`, but as well as subdividing it more finely to allow more interception, extends it to comprise a call to `fluid.initDependents` and further interception points. The use of the call `fluid.initSubcomponents` to initialise subcomponents is discouraged, although since the IoC system which replaces it is not yet in full production status, it cannot be deprecated at this time.

The lifecycle enabled by `autoInit`

The following diagram shows the complete lifecycle which is followed during initialisation of a component which has its creator function automatically created through the use of `autoInit`:



The lifecycle proceeds from its beginning at the top of the diagram, where the component instance is first created, to the bottom, where it is fully initialised and ready to use, following the direction of the blue arrow. This diagram shows the "extended lifecycle" which is operated by the framework for a component created with the `autoInit` grade, enclosed in the large brace shown at the left. For "old-style" components which have a creator function manually invoked, the lifecycle which the framework takes charge of is only the smaller segment enclosed by the smaller brace shown in the centre, attached to the box labelled "initFunction".

Lifecycle listener functions

These functions are all free functions, either registered in the global namespace or supplied literally, and all have the same signature - they all accept a single argument, conventionally named `that` which is the component instance which is currently under construction.

Simple example of autoInit lifecycle component

Here is an illustration of a simple component which is defined using the `autoInit` grade:

```
fluid.defaults("fluid.myAutoComponent", {
    gradeNames: ["fluid.eventedComponent", "fluid.modelComponent", "autoInit"],
    preInitFunction: {
        priority: "last",
        listener: "fluid.myAutoComponent.preInit"
        },
    finalInitFunction: "fluid.myAutoComponent.finalInit",
    events: {
        componentReady: null
    }
};

fluid.myAutoComponent.preInit = function(that) {
    that.applier.requestChange("myPath", 3);
    };

fluid.myAutoComponent.finalInit = function(that) {
    that.events.componentReady.fire(that);
    };



...


var myComponent = fluid.myAutoComponent();
```

This block of defaults defines an automatically created component (that is, the component's creator function `fluid.myAutoComponent` is created by the framework rather than using user code) which configures two functions to intercept lifecycle points during creation, and one event named `componentReady`. This component inherits from both of the standard framework grades `fluid.modelComponent` which causes it to have a model initialised on startup (in fact, by use of a framework-contributed `preInitFunction`) and a set of event firers instantiated.

Since we specified the `autoInit` grade, this component has its creator function, named `fluid.myAutoComponent` created by the framework. This function is ready immediately after the call to `fluid.defaults` finishes for us to conveniently use a namespace to hang off our lifecycle listener functions. This is a helpful convention to follow if the listener functions are tightly bound to the identity of the component itself, but nothing prevents the use of a different naming convention and/or lifecycle listener functions being shared amongst multiple components. In fact, this kind of reusability is precisely one of the aims of the `autoInit` system.

The `preInitFunction`

In this case, we want to ensure that the `preInitFunction` we contribute is definitely called *after* the framework's standard `preInitFunction` attached to the model-bearing grade `fluid.modelComponent`. As a result, we use a slightly longer form of declaring the listener which supplies a `priority` field. The syntax and semantics of lifecycle listeners are exactly the same as those for standard event listeners - all of the directives, including `listener`, `namespace` and `priority` are supported in the same way, as well as the various forms for supplying listeners, and/or arrays of listeners. We want our `preInitFunction` to be called last during the lifecycle point since we want to make use of the component's ChangeApplier, which is created at the standard location `applier` attached to the component's `model` by the framework's own `preInitFunction`.

The `finalInitFunction`

The `finalInitFunction` that we configure here, we decide we are uninterested in expressing any preference for ordering amongst other `finalInitFunction`s - so it is just listed as a raw function name registered into the configuration. However, in real life, given the precise use we make of the lifecycle point (we fire a standard Infusion event to signal the component has finished constructing) we would be wise to register this lifecycle listener using the priority `"last"` to ensure that all constructional activity for the component really had completed before the event was fired.

Choice of lifecycle points

The choice of which lifecycle points to use is governed by how much of the component's functionality the listener needs to use - and conversely, which parts of the further construction lifecycle the component needs to contribute to. In this case, we used a `preInitFunction` so that we could guarantee that some initial material was set up in the component's model - this might be used by the component's subcomponents, or queried in listeners attached to later lifecycle points. Since the event firing listener has to execute last, the choice of `finalInitFunction` for that one is forced. The table below shows some discussion of the uses which these listeners might be put.

Lifecycle points and their possible uses

Lifecycle point	Description	Recommended Use

preInitFunction	Executes immediately after options merging	Set up material for core responsibilities of the component that needs to be visible throughout the early lifecycle - for example, a correct initial state of the model, also any functions or methods which will be used as event handlers
postInitFunction	Executes after event handlers have been initialised	Set up general methods on the component - the main purpose of this lifecycle point is to set up modifications to the component's options and members that need to be visible before the call to initDependents. Note that if methods will be used as event handlers, they must be attached during the `preInit` phase.
finalInitFunction	Executes just before the component is returned	No events should be fired until this phase, and in general no methods should be called until this phase, since the component until now has not been "generally ready for use". However, *references* to methods can be taken, if they exist, during the postInit phase for use during this phase.

Benefits of the `autoInit` and lifecycle system

The primary benefit of the `autoInit` system is that it reduces what would ordinarily have been literal and naturally error-prone user code to declarative form. For the first time, it is actually possible to define a complete functional component without one line of code at all - for example, the following code constructs a fully functional renderer component:

```
fluid.defaults("fluid.myRendererComponent", {
    gradeNames: ["fluid.rendererComponent", "autoInit"],
    selectors: {
        input: ".flc-rendererComponent-input"
    },
    protoTree: {
        input: "${value}"
    }
});
```

This defines a model-bound renderer component which renders a single input field. It could be constructed, for example, using a lines such as

```
    var model = {value: "My String"};
    var component = fluid.myRendererComponent(".flc-rendererComponent", {model: model});
```

The value in the supplied model would be automatically kept up to date with changes made by the user as a result of the `autoBind` facility of the renderer.

Other benefits include removing error-prone aspects of the component workflow. For example, having the component creator lifecycle in code creates the chance that no standard framework init function may be called, or these functions may be called in the wrong order or with inappropriate arguments - for example, a call to `fluid.initDependents` at an inappropriate time. Seeing that a component is registered with `autoInit` puts the reader's mind at rest that all of these aspects are taken care of automatically and that the component will behave in a standard way when combined with others. Special case uses remain for components with non-standard workflows but these can now be highlighted, warning readers to look more closely when seeing a manual creator function written out.

Finally, the use of lifecycle init functions of this kind increase the potential for reusability of code. Since these all enjoy a standard component signature (accepting `that` as a single argument) and are packaged as public functions out in the open, any common workflow that can be found between different component types is maximally likely to be reusable for the different components. In the "old world" where component initialisation material was typically kept in private scope functions in long streams of activity, this kind of reuse was unlikely to be realised.

Future of the lifecycle system

The system described here is stable but is not in the final form that we ultimately desire. Component writers still need to *think* about WHICH lifecycle point to attach their listeners to, based on their purpose - rather than having the framework be able to automatically infer the correct time to fire the listener based on an understanding of its purpose. This relies on an upcoming framework feature known as the *globally ginger world* that will interleave the two processes of {`merge options`} and {`initDependents`} shown in the diagram into a single unified instantiation process that will proceed across all instantiation activities in an order determined by their inter-dependencies. This is a tail for which the world is not yet ready.