Why Inversion of Control?

This functionality is Sneak Peek status. This means that the **APIs may change**. We welcome your feedback, ideas, and code, but please use caution if you use this new functionality.

The Infusion Framework is designed to support highly flexible aggregation of reusable components. The Inversion of Control (IoC) system provides the ability for reusable components to **pull** dependencies from the environment rather than having to push them in initialisers. Inverting control this way allows users of the framework and components to customize implementations by plugging in alternative subcomponents, without having to modify the parent component code.

Compare the following examples:

Without IoC

```
/**
 * Component
creator function
 */
fluid.myComponent =
function
(container,
options) {
    var that =
fluid.initView
("fluid.
myComponent",
container, options);
    that.model = {
        ...
    };
    that.field =
that.locate
("inputField");

    //
Instantiation of
first subcomponent
    that.sub1 =
fluid.
initSubcomponent
(that, "sub1",
        [that.
options.selectors.
sub1container, {
            model:
that.model,
            field:
that.field
        }]);

    //
Instantiation of
second subcomponent
    that.sub2 =
fluid.
initSubcomponent
(that, "sub2",
        [that.
model]);

    //
Instantiation of
third subcomponent
    that.sub3 =
fluid.
initSubcomponent
(that, "sub3",
        [that.
container, that.
events.onSave]);

    return that;
};
```

This component creator function initializes three subcomponents. Each initialization requires a call to `fluid.initSubcomponent()` with the specific parameters that the subcomponent implementation requires (the array of parameters is the third argument to `fluid.initSubcomponent()`). The second argument to `fluid.initSubcomponent()` is the string name of the subcomponent, as specified in the defaults (not necessarily the actual function name).

```
/**
 * Default options
for the component
 */
fluid.defaults
("fluid.
myComponent", {
    sub1: {
        type:
"fluid.
mySubComponent1Impl"
    },
    sub2: {
        type:
"fluid.
mySubComponent2Impl"
    },
    sub3: {
        type:
"fluid.
mySubComponent3Impl"
    },
    selectors: {
        inputField:
".flc-myComponent-
inputField",

sub1container: ".
flc-subComponent1"
    },
    events: {
        onSave:
null,
        afterSave:
null
    }
});
```

The `fluid.defaults()` call declares the default function to use for the subcomponents, by name. Users can override these defaults by specifying a different function name, but the arguments to the customized replacement must be the same, or the user will have to modify the code that calls `fluid.initSubcomponent()` for that subcomponent.

With IoC

```
/**
 * Component creator function
 */
fluid.myComponent = function
(container, options) {
    var that = initView("fluid.
myComponent", container, options);
    that.model = {
        ...
    };
    that.field = that.locate
("inputField");

    // initDependents() will
instantiate all three subcomponents
    fluid.initDependents(that);

    return that;
};
```

With IoC, the component creator function does not initialize each subcomponent individually: `fluid.initDependents()` initializes anything that is defined in the `components` section of the defaults provided to `fluid.defaults()`.

```
/**
 * Default options for the component
 */
fluid.defaults("fluid.myComponent", {
    gradeNames: "fluid.viewComponent",
    components: {
        sub1: {
            type: "fluid.
subComponent1Impl"
        },
        sub2: {
            type: "fluid.
subComponent2Impl"
        },
        sub3: {
            type: "fluid.
subComponent3Impl"
        }
    },
    selectors: {
        inputField: ".flc-myComponent-
inputField"
    },
    events: {
        onSave: null,
        afterSave: null
    }
});
```

In the component defaults, the *types* of components to use for the subcomponents are defined inside a property called `components`. The `fluid.initDependents()` will look inside this property for the list of dependents to initialize.

The values for the `type` properties may be function names, but this is not necessary. The function to use for the types specified, as well as their arguments, are specified through `fluid.demands()`, shown below.

With IoC and "autoInit" grade

It is encouraged to use IoC system in conjunction with the component grade system. Using "autoInit" allows for simpler and more declarative implementation of component. The example of such "autoInit" component is below:

```
/**
 * Default options for the component
with autoInit grade.
 */
fluid.defaults("fluid.myComponent", {
    gradeNames: ["fluid.viewComponent",
"autoInit"],
    preInitFunction: "fluid.myComponent.
preInitFunction",
    postInitFunction: "fluid.
myComponent.postInitFunction",
    components: {
        sub1: {
            type: "fluid.
subComponent1Impl"
        },
        sub2: {
            type: "fluid.
subComponent2Impl"
        },
        sub3: {
            type: "fluid.
subComponent3Impl"
        }
    },
    selectors: {
        inputField: ".flc-myComponent-
inputField"
    },
    events: {
        onSave: null,
        afterSave: null
    }
});
fluid.myComponent.preInitFunction =
function (that) {
    that.model = {
        ...
    };
};
fluid.myComponent.postInitFunction =
function (that) {
    that.field = that.locate
("inputField");
};
```

With autoInit grade, the component creator function will be generated from component defaults. Any code present in the old style creator function can then be distributed into appropriate lifecycle functions such as preInitFunction, postInitFunction and finalInitFunction.

```
/**
 * Subcomponent demands
 */
fluid.demands("fluid.
subComponent1Impl", "myComponent", {
    container: "{myComponent}.options.
selectors.sub1container",
    options: {
        model: "{myComponent}.model",
        field: "{myComponent}.field"
    }
});

fluid.demands("fluid.
subComponent2Impl", "myComponent", {
    funcName: "fluid.
mySubComponent2Impl",
    args: ["{myComponent}.model"]
});

fluid.demands("fluid.
subComponent3Impl", "myComponent", {
    container: "{myComponent}.
container",
    options: {
        events: {
            onSave: "{myComponent}.
onSave"
        }
    }
});
```

The calls to `fluid.demands()` specify both

- what function to use for a given context, and
- what arguments to provide to that function

The second call to `fluid.demands()`, for example, says:

- use the function `fluid.mySubComponent2Impl` when instantiating `fluid.subComponent2Impl` in the context of `myComponent`
- pass the `model` property of the parent component as a parameter

```
/**
 * Subcomponent demands
 */
fluid.demands("fluid.
subComponent1Impl", ["testEnvironment",
"myComponent"], {
    funcName: "fluid.
mySubComponent1Impl",
    args: ["#test-markup", {
                model: testData.model,
                field: "{myComponent}.
field",
            }]
});

fluid.demands("fluid.
subComponent2Impl", ["testEnvironment",
"myComponent"], {
    funcName: "fluid.
mySubComponent2Impl",
    args: [testData.model]
});

fluid.demands("fluid.
subComponent3Impl", ["testEnvironment",
"myComponent"], {
    funcName: "fluid.myTest3Impl",
    args: [$("#test-markup"),
testOnSaveHandler]
});
```

Additional calls to `fluid.demands()` can be used to demand different arguments, or even functions, in different contexts. This example shows different demands when the subcomponents are being used by `myComponent` in a test environment. Test data can be provided instead of actual data, if desired. The third subcomponent is demanding a completely different implementation.

These additional demands specification can be made in a different file, if desired. For example, demands specific to a testing environment can be placed in the test files.