Demands Specifications

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.

On This Page

Overview

The core of the Infusion IoC is the ability for a function call to resolve differently in a particular context. One way of thinking of this resolution is in terms of Aspect-Oriented Programming - the matching of the context specification could be seen as a kind of *joinpoint*, and the transformation of the original caller's function call, to resolve as the execution of a different function call, as a form of *advice*. One can set up configuration for this resolution by making calls to the function `fluid.demands()`, which accepts three parameters:

```
fluid.demands(demandingName, context, spec);
```

Name	Type	Description
demandingName	String	The name of the function requested by the original caller of the function, which is to be advised
context	String or Array of String	The name of the context(s) which need to match in order for this demands specification to be active
demandSpec	Object [Demands specification]	A record describing the modification to be performed when a dispatch to the original function is requested ("advice")

1. The name of the function which the original caller uses (the "demanding/demanded name")
2. The name of the context(s) for which the demand specification is valid, and
3. The demand specification ("demands block") itself.

The following sections describe the demands specification format, and several examples are provided below.

The Demands Specification Object

The demands specification object can occur in a few forms. The fullest form is described here, and then an abbreviated format is defined in terms of this full form.

`funcName`	the string name of the creator function for the demanding subcomponent. NOTE: If this property is not present, the original function name (agreeing with the first argument to `fluid.demands()`) is to be used instead
`args`	array of parameters to the function, or a single argument.
OR	
`options`	an object specifying how the `options` argument to a component's creator function (identified as a pseudoargument is to be assembled
`mergeOptions`	an array of objects composing the `options` argument to a component's creator function is to be assembled by merging several different IoC-resolved records together
`container`	an IoC-resolved reference (most likely of the form `{parentComponent}.dom.selectorName` expressing where the `container` argument to a View is to be taken from
any pseudoargument name	The name of any pseudoargument defined by the grade of the component which is to be created, representing the slot of a positional argument in the transformed argument list

Note that if the `args` record is used, NONE of the other mentioned elements can be present (none of `options`, `mergeOptions`, `container`, etc.). Also, ONLY ONE element may be used out of `options`, `mergeOptions`.

The demands specification may also simply consist of an array. In this case, the array is interpreted as being the array `args` above, and the `funcName` entry is interpreted as being empty.

Context references

The demands specification allows contextualised access to state which is held in locations around the component tree which are in scope from the site where the demands block is matched. That is, at the time the demands block is matched against the caller's use of the function, the *caller's* environment will be searched in order to *resolve* the references in the demands block. These references take the form of contextualised EL strings, where the context name is held in braces at the start of the string. The rest of the string consists of path segments which are a query into the object which matches the context name. For example:

```
    "{componentName}.model"
    "{componentName}.options.strings"
```

These contextualised values are resolved whenever a demands block is matched. The complete set of times when demands resolution occurs consists of i) a call to `fluid.initDependents()` or `fluid.initDependent`, used to construct a subcomponent, ii) a function call dispatched at an invoker, and iii) resolution of a boiled event. Demands resolution is also known as *function resolution*, which always involves this process of resolving contextualised EL references, which is itself known as *value resolution*. Value resolution also occurs by itself, without function resolution, during expansion and merging of options in an IoC tree.

Examples

```
fluid.demands("cspace.autocomplete.popup", "cspace.autocomplete",  {
    args: {
        resources: {
            template: {
                url: "../../html/AutocompleteAddPopup.html"
            }
        }
    }
});
```

In this example, the `demandSpec` does not include a `funcName`, so when the demand is resolved, the creator function will be assumed to be `cspace.autocomplete.popup`, the first argument to `fluid.demands()`, which is the same function the user originally requested.

```
fluid.demands("fluid.pager", "cspace.search.searchView",
                 ["{searchView}.dom.resultsContainer"]);
```

In this example, the demands specification (the third argument to `fluid.demands()`) does not include a `funcName`, so when the demand is resolved, the creator function will be assumed to be unchanged as `fluid.pager`, the function the user originally requested. The pager is dispatched a single argument, its container node, resolved through the DOM binder of the `searchView` component.

```
fluid.demands("cspace.autocomplete.closeButton", "cspace.autocomplete",
    ["{autocomplete}.autocompleteInput", "{options}"]
);
```

This example shows the use of the special contextual value {`options`} - this is a special value which does not match a component, but instead represents the block of options that were originally targetted at the subcomponent whose instantiation is matched by the demands specification, in the `fluid.defaults()` block entry for that subcomponent. This demands block is completely equivalent to the following simpler demands block which shows the use of the pseudoargument name `container`:

```
fluid.demands("cspace.autocomplete.closeButton", "cspace.autocomplete", {
    container: "{autocomplete}.autocompleteInput"
});
```

Note that in this case, both the `funcName` and `options` entries, through not appearing, cause the matching of the demands block to default to leaving the original behaviour that would be expected from the creator function call unchanged in these respects. The only change caused by resolution of this demands block is to cause the first argument of the creator function to be filled with the value drawn from the `autocomplete` component.

A note on "tricks"

It is important not to play "tricks" with demands blocks. That is, they should not be combined with any code whatsoever - that is, they should not be invoked from inside functions (except the global closure used to guard the scope of a file), they should not contain function calls or variable references from the Javascript language, but should be written exactly as they appear here - with three, literal concrete arguments taken from the JSON dialect of JavaScript. This is essential in order to enable interpretation of the structure of demands blocks throughout an file or entire application by development support tools (which are currently yet to be written). Demands blocks should never be "conditionally included" - except through either including or not including the files in which they appear into an application or web page. They should be properly scoped to appear in the same files as the components and functions they refer to - that is, they should not violate dependency rules by appearing in unrelated files.

TNT - Try No Tricks!