Demand Resolution

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

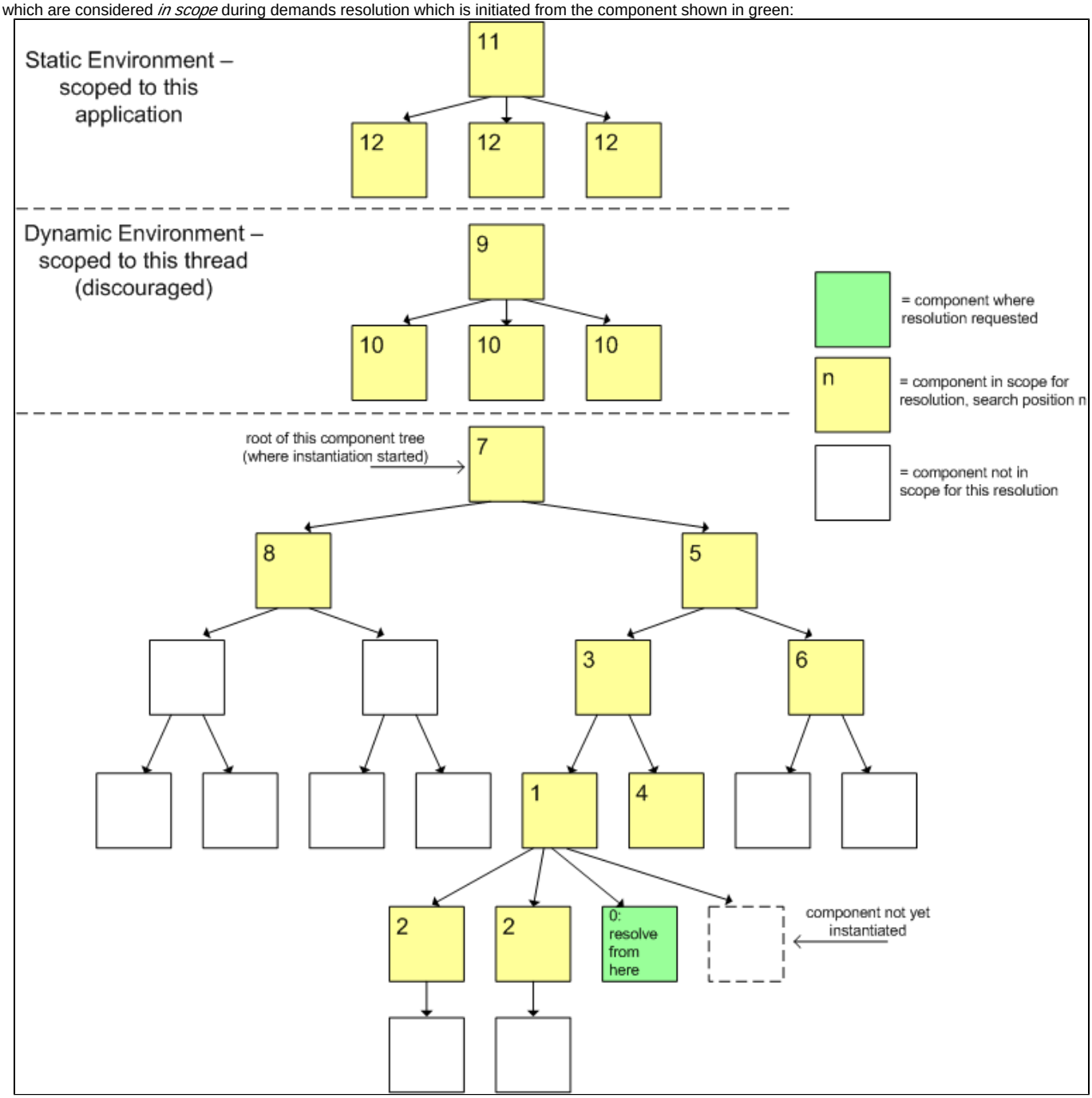
When does demand resolution occur?

Demand resolution is a process that is performed by the IoC system in order to convert a function call which has just been issued to a particular globally named function with a particular set of arguments, into a possibly different function call, one which differs either in the argument list supplied to the function, the function's global name, or both. Demand resolution currently occurs in the following three scenarios:

1. When constructing a subcomponent which has been configured into its parent's `components` block, and is now instantiating as a result of a call to `fluid.initDependents()` or `fluid.initDependent`
2. A function call which is being executed by an invoker (or a direct call to `fluid.invoke`)
3. resolution of a boiled event.

Demands Location and Function Resolution

The first stage of demand resolution is searching the demands blocks registered into the system to see if any of them match the case of the requested function name in the context that it is executing. This involves a search of the component tree around the site of the issued function call to assemble the set of *context names* which constitute the environment of the call. The diagram shown here shows graphically (in yellow) the set of components in the tree which are considered *in scope* during demands resolution which is initiated from the component shown in green:



The results of these simultaneous searches select one more demands blocks which match the required call - a demands block matches, if the context names it mentions match the context names found in the environment, as well as its demanding name matching the requested function.

Either:

1. **no demands are found matching the requested function call** - in this case, it is dispatched as normal
   OR
2. **one or more demands blocks are found matching the function call** - In this case, the system will select the *best match* from the set of matching demands, and use this block to transform the execution of the function and then dispatch it. For example, if the best match demands block specifies that a different function name should be selected, and/or different argument values should be supplied, these are used to replace the user's requested details before executing the function. These argument values are assembled using *value resolution* (described below). One demands block is considered a better match than another if it matches *more* of the context names that were found in the environment search, whilst not *mismatching* by being registered for context names which are *not* found in the environment.

Simple example of demands resolution for subcomponent creator function

Here is a simple example showing matching of a demands block for case 1) of demands resolution we described above. When a call to `fluid.initDependents` is executed (by the user, or by the framework) to construct the subcomponents of component `my.component`, demand resolution will occur for the demanded function names in that context. In this case, one of these will match the supplied demands block and be transformed, and the other will not and proceed as normal.

```
fluid.demands("subcomponent2", "my.component", {
    funcName: "my.otherSubcomponent"
});
fluid.defaults("my.component", {
    components: {
        subcomponent1: {
            // there is no demands block for this type, so
            // the function will be "my.subcomponent"
            type: "my.subcomponent"
        },
        subcomponent2: {
            // the demands block above will be used to
            // resolve this type to "my.otherSubcomponent"
            type: "subcompType2"
        }
    }
});
```

Value Resolution of Arguments

If a demands block is selected as a match for a particular function invocation, it may direct that the argument list of the function should be replaced with a different list, composed from values drawn from the environment surrounding the function call site, as well as from the original argument list. This process involves *value resolution*. Path expressions (contextualised "EL expressions"), strings of the following form may be used to represent values drawn from elsewhere in the tree, and during value resolution, will be replaced by the value which they match:

```
"{contextName}.path1.path2"
```

This matching of contexts proceeds by the same scoping rules as in the diagram above. The numbers in the top left corners of the yellow boxes show the sequence in which the environment of the call site will be searched for components matching the requested context name. The first match found in this sequence for `contextName` will select that component, and the patch `path1.path2`(etc.) will be used to resolve a property or sub-property of the component e.g.

```
fluid.demands("cspace.autocomplete.authoritiesDataSource",
              "cspace.autocomplete", {
    funcName: "cspace.URLDataSource",
    args: {url: "{autocomplete}.options.vocabUrl"}
});

fluid.demands("searchBox", ["cspace.header"],
    ["{header}.options.selectors.searchBox", "{options}"]);

fluid.demands("cspace.recordList", "cspace.relatedRecordsList", {
    funcName: "cspace.relatedRecordsList.provideRecordList",
    args: ["{relatedRecordsList}.container",
           "{relatedRecordsList}.options.selectors.recordListSelector",
           "{relatedRecordsList}.model.relations",
           "{relatedRecordsList}.options.related",
           "{options}"
    ]
});
```

The value `"{options}"` is an instruction to the Framework to insert any `options` found in the `components:` declaration for the subcomponent. If these include references to the parent component or other values in the environment, they will also be resolved via value resolution.

```
fluid.demands("cspace.searchBox", ["cspace.header"],
    ["{header}.options.selectors.searchBox", "{options}"]);

fluid.defaults("cspace.header", {
    components: {
        searchBox: {
            type: "cspace.searchBox",
            options: {
                permissions: "{header}.options.permissions",
                schema: "{header}.options.schema"
            }
        }
    }
}
```

In this example, the `searchBox` subcomponent will be instantiated by calling its creator function `cspace.searchBox()`. The arguments passed to it will be the selector named `searchBox` drawn from the `cspace.header` component's options (which must exist in the environment since the `"cspace.header"` context has matched), followed by the standard options specified in the `defaults`: an object with `permissions` and `schema`, drawn from the `cspace.header` component's options.

A more compact and readable way of writing the same demands block (with identical effect) makes use of the `container` pseudoargument (assuming that the searchBox does indeed have the grade of viewComponent):

```
fluid.demands("cspace.searchBox", ["cspace.header"], {
    container: "{header}.options.selectors.searchBox",
    options: {} // Optional
});
```