Contexts

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

Components or functions in general may have different requirements depending on the context in which they are operating. For example, a subcomponent might operate differently when running on a production server versus when testing locally off the file system, and differently still when operating in the context of automated tests. In a more fine-grained way, a component may behave differently when operating in a browser with different capabilities, or on behalf of a user who has expressed particular needs or preferences.

How context names are derived

Configuration material makes use of context names, when it is expanding - context names are derived from both particular components in the tree which have already instantiated (ancestors) and static environment. These names can be matched by the `context` parameter of `fluid.demands`, as well as names appearing in curly brackets at the beginning of EL path expressions like `"{contextName}.furtherPath"` . Each component in the tree can give rise to a context name through up to three strategies:

- The fully qualified name `typeName` of the component that is found in the context, e.g. `"fluid.uploader.progressiveStrategy"`
- The component's name when embedded within its own parent, e.g. `"strategy"` - this strategy is not available for components at the head of the tree. It is also the most unstable strategy since it depends on particular naming of members in the tree rather than remaining stable if the tree is reorganised. However, it is the only route for matching a particular *instance* of a component rather than matching components by their type or name in general.
- The component's "nickname" - which by default is derived from the last path segment of its fully qualified name, e.g. `"progressiveStrategy"`, but may be overridden by an option supplied to its creator function named `nickName`.

Sometimes you just need a context name, and don't need a component that does anything. In this case, you can use the special "vestigial" Fluid component which is created using the universal creator function `fluid.typeTag`. This takes as argument the name which is to be written onto the `typeName` member of the returned object, and returns a "component" which defines nothing else but the type tag. For example, an automated testing file might define a testing environment tag.

Where context names are looked for

Context names listed in a demands block are searched for at instantiation time, sequentially, in three kinds of "scopes" or "environments".

- Most immediately, context names are searched for in the tree of currently instantiating components, searching **upwards** through the tree from the parent of the component for which the demands a being resolved. This search currently cascades through **all** currently present components in the tree, although there are a few ways of marking a boundary for search - most easily, setting an option named `"fluid.visitComponents.fireBreak"` to `true` on your component.
- Following this, a search is made in the *dynamic environment*, which is notionally a separate tree of components bound to the currently executing thread (in a browser, there is just one thread, but the contents of the dynamic environment are still scoped to stack frames above a certain point in the call chain).
- Finally, context names are sought in the *static environment*, which is a tree of components attached globally to the JS VM as a whole. These are not intended to change during the lifetime of the VM, but currently there are some coding styles in use which can require these to be modified. It is probable that in a future release of the system, that the static environment will be split into "more mutable" and "less mutable" parts.

The use of the dynamic environment is not particularly reliable for this purpose - it will "go away" when the current execution stack returns. If one of the components in the tree tries to make use of deferred instantiation, this material will have gone away.

How context names are matched

If an *array* of context names is provided, they are conceptually ANDed. The IoC system will attempt to find registered demands that most closely match the `context` parameter. So, for example:

```
fluid.demands("subcomp", "comp2", demandspec1);
fluid.demands("subcomp", ["comp2", "testEnv"], demandspec2);
```

These two lines can be interpreted as follow:

- When `subcomp` is created by `comp2`, use `demandspec1`.

- When `subcomp` is created by `comp2` in the context of `testEnv`, use `demandspec2`.

The process of matching context names for the purpose of resolving a demands block is called **function resolution**, which is to be distinguished to **value resolution** which is used to match context names for the purposes of resolving values (appearing in curly brackets at the head of EL paths) held in demands blocks themselves and in component defaults. Given that **all** matched context names are considered during name resolution, the specific ordering of search presented in the previous section is not so relevant, although this may become relevant in future releases of the IoC system.

A demands block "outcompetes" another if it matches strictly more context names in its environment than another. If it happens that two blocks match the same set of names, a block may still be outcompeted if it declares more names that **mismatch** in the environment. If the number of matches and mismatches for two blocks are the same, for the two top ranked blocks for an invocation, the system declares that the invocation is ambiguous and fails the invocation.