ChangeApplier API

ChangeApplier API

This section explains and documents the various Javascript API calls for instantiating and working with ChangeAppliers.

Instantiating a ChangeApplier

Instantiating a ChangeApplier is extremely simple.

```
var applier = fluid.makeChangeApplier(model);
```

New in v1.3:

```
var applier = fluid.makeChangeApplier(model, options);
```

Please see Options below for more information about the new `options` parameter. Note that this new parameter is independent of the new Transaction support (which also uses options).

The `makeChangeApplier` call takes a single argument, which is the model object (the root of a tree of plain Javascript objects) to which this applier is to be attached. Note that by standard model semantics, whilst any subobjects and properties under this tree can and will change asynchronously, the essential identity of this model tree is defined by this exact object handle `model`. Therefore to establish agreement amongst model citizens and appliers about *which* model is being talked about, this object handle must be preserved, whatever happens to the object tree beneath it. The Fluid base framework contains various utility functions which help to make this easy - for example

```
fluid.clear(model);
```

is a call which unlinks all properties from the supplied model, preparing it for a wholesale change. A corresponding useful call is

```
fluid.model.copyModel(model, newModel)
```

On This Page

See Also

- ChangeApplier
- Infusion Event System

Still need help?

Join the infusion-users mailing list and ask your questions there.

which transfers all the state from `newModel` onto `model` - whilst still preserving the identity of `model` as `model` - and hence its association with any particular ChangeApplier. These two calls, `clear` followed by `copyModel` often occur together (and will be automated in a future version of the ChangeApplier).

After its construction, the particular model object to which a ChangeApplier is bound is available at `applier.model`.

New in v1.3:
Version 1.3 of Infusion includes a Sneak Peek view of our new transactional ChangeApplier. The ChangeApplier supports transactions by default, and can be configuring using an optional `options` parameter:

```
var applier = fluid.makeChangeApplier(model, options);
```

For more information see the fluid:Transactional ChangeApplier section below.

Firing a change using a ChangeApplier

There are two calls which can be used to fire a change request - one informal, using immediate arguments, and a more formal method which constructs a concrete ChangeRequest object.

```
applier.requestChange(path, value, type)
```

Parameter	Type	Description
`path`	`string`	An EL path into the model where the change is to occur.
`value`	`object`	An object which is to be either updated (or added if necessary), or removed from the model at path `path`
`type`	(optional) `"ADD"` or `"DELETE"`	A key string indicating whether this is an ADD request (the default) or a DELETE request (a request to unlink a part of the model)

```
applier.fireChangeRequest(changeRequest)
```

`requestChange` and `fireChangeRequest` reach exactly the same implementation - the only difference is in the packaging of the arguments. For `requestChange` they are spread out as a sequence of 3 arguments, whereas for `fireChangeRequest`, they are packaged up as named fields (`path`, `value` and `type`) of a plain Javascript object. Such an object is called a "ChangeRequest" and is a convenient package for these requests to pass around in in an event pipeline.

Registering interest in a ChangeApplier

Currently ChangeAppliers support two types of listeners (to be expanded).

Guard Listeners.

The first type, called **guard listeners**, are notified of an upcoming change request *before* it occurs. Guards can be registered and deregistered at the path `guards` with a call to `addListener`:

```
applier.guards.addListener(pathSpec, guard, namespace)
```

Parameter	Type	Description
`pathSpec`	v1.2 and earlier: `string` New in v1.3: `string` or `Object`	An EL expression, possibly with wildcards (* in place of a path component) which matches the set of EL paths, which on receiving a change request, will trigger this guard. New in v1.3: The `pathSpec` can also be an Object with the following properties: `{ path: an EL expression priority: an integer transactional: boolean (see Transactional Listeners below) }` The `priority` will affect when the listener is fired: All listeners will be sorted according to priority and fired in order. A higher number implies a higher priority. The default if unspecified will be the highest possible priority.
`guard`	`function`	A function pointer holding a guard
`namespace`	`string`	(Optional) - a namespace key, which is used to scope additions and removals of this guard (see Infusion Event System for interpretation of event namespaces)

A guard may be removed with a call to

```
applier.guards.removeListener(guard)
```

where `guard` holds either the original function reference, or else the `string` value supplied as the `namespace`.

A guard listener is just a function with a particular signature - for example, `guard` could be implemented as follows:

```
function guard(model, changeRequest) {
    if (changeRequest.value === null) {
        return false;
        }
    }
}
```

The behaviour of this guard is to reject any incoming change which would appy a null value to the model - by making a `false` return, the entire change cycle which triggered it would be cancelled, since guards have the semantics of *preventable events* (see Infusion Event System for the different event categories).

The arguments supplied to a guard are as follows:

Parameter	Description
model	The overall model attached to the ChangeApplier with which this guard is registered
changeRequest	The ChangeRequest object with which this request was started

modelChanged listeners

Registration and deregistration of `modelChanged` listeners is just as for guards -

```
applier.modelChanged.addListener(pathSpec, listener, namespace)
applier.modelChanged.removeListener(listener)
```

The signature and timing of the `listener` is different to guards. Unlike a guard, the listener is notified *after* the change has already been applied to the model - it is too late to affect this process and so this event is not *preventable*. The signature for these listeners is

```
function listener(model, oldModel, changeRequest)
```

Parameter	Description
model	The overal model attached to the ChangeApplier with which this listener is registered
oldModel	A "snapshot" of the previous model condition - created as if by `fluid.copyModel`.
changeRequest	The ChangeRequest object with which this request was started

New in 1.3: Options

In version 1.3, the ChangeApplier now supports an optional `options` parameter that allows users to configure how the Applier works. The currently supported options are:

Name	Description	Values	Default
`cullUnchanged`	This option allows users to configure a ChangeApplier to *not* fire a `modelChanged` event if a change request doesn't, in fact, modify the model.	boolean	`false`

Sneak Peek in v1.3: Transactional ChangeApplier

As of version 1.3 of Infusion, the ChangeApplier now supports transactions. Note that this feature is in Sneak Peek status, and so its APIs will change. We encourage users to provide feedback by emailing our infusion-users mailing list.

The functioning of the transactional ChangeApplier can be configured using the following options:

Name	Description	Values	Default
`thin`	By default, transactions create a copy of the model and apply changes to the copy, only modifying the original model on commit. The `thin` option allows users to configure a ChangeApplier to apply transactional edits directly to the mode. This can be useful where performance is an issue, but should be used with care: With this option, rolling back changes instead of committing them is not possible.	boolean	`false`

Using Transactions

A transaction can be opened using the new `initiate()` function which returns a transaction object:

```
var myApplier = fluid.makeChangeApplier(myModel);
var myTransaction = myApplier.initiate();
```

Once the transaction has been create, it can be used to request changes to the model:

```
myTransaction.fireChangeRequest(requestSpec1);
myTransaction.fireChangeRequest(requestSpec2);
...
```

The transaction can be completed using the new `commit()` function of the transaction object:

```
myTransaction.commit();
```

A single `modelChanged` event will be fired on completion of the commit, regardless of the number of change requests.

Post Guards

Just as guard listeners are notified of upcoming changes to the model (offering the opportunity to prevent those changes), *post guards* are notified of an upcoming transactional commit, offering the opportunity to prevent the completion of the entire transaction. Post guards are NOT notified with each change request, but only once, on commit.

Post guards are registered similarly to regular guards, using an `addListener` function:

```
myApplier.postGuards.addListener(pathSpec, guard, namespace);
```

Aside from when post guards are notified, they follow the same specification, function signature, etc. as fluid:regular guards, described above.

It is important to note that if the `thin` options is `true`, then transactional changes will have been applied to the model directly, and any post guards will NOT be able to prevent those changes. The ChangeApplier does not offer the ability to roll back changes when `thin` is `true` (though there's nothing stopping users from attempting to undo changes themselves, perhaps within a post guard).

Sneak Peek in v1.3: Transactional Listeners

As of version 1.3 of Infusion, the ChangeApplier now supports transactional listeners. Note that this feature is in Sneak Peek status, and so its APIs will change. We encourage users to provide feedback by emailing our infusion-users mailing list.

Transactional listeners provide users the opportunity to attach listeners that will create a transaction when they are invoked, so that any changes which are requested by the listeners will occur seemingly as one operation with the initial change request.

Specifying Transactional Guard Listeners

To specify that a guard listener should be a transactional listener, users can use one of two techniques:

An exclamation mark preface on a pathSpec string

```
myApplier.guards.addListener("!myModel.section.*", myGuardFunc);
```

A `transactional` flag in a pathSpec object

```
var pathSpec = {
    path: "myModel.section.*",
    transactional: true
};
myApplier.guards.addListener(pathSpec, myGuardFunc);
```