ChangeApplier

Overview

The ChangeApplier is a core (though still rapidly developing - first introduced in the 1.0 release) part of Fluid's architecture, which handles the function of data binding and coordinates model updates in general. Some notes are present at Notes on the ChangeApplier - the ChangeApplier was formerly (and still is, in its counterpart in the server-side RSF framework) known as the DARApplier, after the DataAlterationRequest structure which appears in its event APIs.

As well as being based on Fluid's model-directed thinking, the ChangeApplier is also implemented in terms of Fluid's Event System, which you should be familiar with before using the ChangeApplier.

Thinking behind the ChangeApplier

The ChangeApplier is a natural outgrowth of Fluid's focus on (transparent) model-directed programming - see our page on Component Model Interactions and API. The core contract is that a model should be "fully transparent" - meaning, that it consists of standard POJOs and is available for inspection by reading, using standard language constructs, at all times. For example, if `model` is a Javascript variable holding the overall model, accessing a field within the model is as simple as writing a standard Javascript expression `model.field1.subfield2` etc.

On This Page

See Also

- ChangeApplier API
- Infusion Event System

Still need help?

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

This is in contrast to other frameworks, which almost universally insist that a model is composed of some form of more or less "magic" objects - these might be derived from a base class or interface which is called "Model" or "Storage". This is a classically bad architectural smell. Allowing any "reasonable" body of objects to act as a model greatly increases flexibility and directness, as well as removing unspeakable "magic" from user interaction with their object trees.

Reads are transparent, writes require formality

However, this freedom (particularly in read access) implies a cost (or a formality) elsewhere, in write access. Since read access is essentially "unannounced" and asynchronous, it would be deeply unfortunate for a user of the model to simply reach in and change the model values, through the same base language rules that were used for read access, without any formality. The ChangeApplier is the central point of control, attached to a block of model, for applying updates to the model. It has a variety of responsibilities - as well as allowing users to react to "upcoming" changes (perhaps embodying validation rules, type constraints or state management), it is delegated to by the user to physically apply the change, and then will notify other citizens of the change which just happened. In some advanced uses, the ChangeApplier may also operate some kind of "transactional" semantics, allowing rollback of refused changes.

A helpful analogy

A useful analogy for thinking about this asymmetry and the reason behind it is perhaps an SVN server. The repository is exposed for read access over plain HTTP - as if it consisted of concrete files kept at the current HEAD revision. However, it is not possible to write to it over the same interface (this would correspond to the repository being writeable over plain WebDAV). This is because the repository consists of a set of "interesting public state" - a custom protocol is required for committing changes and revisions, simply because there are so many interested parties in the process, for reasons of authentication, versioning, atomicity, etc., and any other reactive hooks that the administrator of the repository may wish to install. "Interesting public state" is of just as much importance in an internal application design, as it is in the Internet at large, and this resulting solution of "transparent read, but formal write" is a good resolution of the same design tensions.