SubPages

So far we have talked about DomUI **pages**. These are classes derived from UrlPage which are uniquely identified by an URL containing a class name and a set of parameters. Changing to another Page means changing the URL, and causing a Browser page change. A DomUI Page, once rendered, is fully AJAX, but changing pages is not AJAX but is closer to a traditional HTML page. This was a good match for the first application DomUI was written for.

Modern applications try to be fully AJAX: they load the main page once, and from then on all changes are made to that page with AJAX calls. They never reload the entire page. While it is possible to make this with DomUI 1.x it is hard because having everything in a single Page means that state management difficult because everything uses the same Conversation (and thus the same QDataContext).

DomUI 2.0 has a new concept called *SubPages* which aims to fix this. This are part of the SPI (Single Page Interface) implementation for DomUI.

This is preliminary info; the implementation can still change quite a bit.

What is a SubPage

A SubPage is a class that extends the class **SubPage** instead of UrlPage. A SubPage is a DomUI fragment represented as a Div which can be added anywhere in the DomUI DOM. A SubPage is always part of an UrlPage, but an UrlPage can contain multiple SubPage's if so desired. SubPages themselves can *also* contain other SubPages.

SubPages are added just like any other node: by calling add() on some node to add the SubPage.

Subpage state management: the SubConversationContext

SubPages also implement state management which is similar to the state management that is done by UrlPages. Like an UrlPage a SubPage has a ConversationContext (called SubConversationContext). These ConversationContexts are children of the main ConversationContext that is attached to the UrlPage, and have the same lifecycle states.

A SubPage's conversation context is created as soon as the SubPage is added to an UrlPage, either directly or indirectly. If the SubPage does not yet have a conversation one gets created and it gets added to the main ConversationContext of the UrlPage. From now on the SubPage's conversation follows the lifecycle of the main ConversationContext: if the main Conversation is detached so will the SubConversation. In this way we are sure that resources belonging to the SubConversation are properly managed.

A SubPage and its underlying conversation is destroyed when it is removed from a page. The destruction is not immediate but done when the request finishes. This allows you to temporarily disconnect a SubPage and reconnect it somewhere else within the same request.

The shared QDataContext on SubPages

Calls to getDataContext() on nodes that belong to a SubPage will get their data context from the SubConversation belonging to the closest SubPage parent (remember that SubPages can be nested too!). This means that a SubPage **has its own database connection and session**. In this way the SubPage acts as a "fence" between different connections and objects.

This shared context will be managed as usual, its connection will be released as soon as the subconversation detaches.

Entity (data) objects in SubPages

The normal way for creating SubPages is to have some class with a constructor, passing the data objects into the constructor. This is completely different from UrlPage's: these get their objects from "outside", and because of this these objects are always part of the QDataContext (session) of that UrlPage.

But when you create a SubPage and pass Entity objects in the Constructor those objects belong to the QDataContext of the *parent* of the SubPage. But the SubPage has its own QDataContext, and using the object as-is would cause trouble as all changes to the object would be done in the parent database context, not in the context inside the SubPage.

DomUI tries to fix this as follows: as soon as your SubPage is added to the page DomUI will scan the SubPage class for (private) fields and getters. All fields and getters that contain an object which is determined to be an Entity object (as detected by MetaManager.findClassInfo) will need to be *re-injected* in the SubPage. Re-injecting means: a new instance for the object is loaded from the QDataContext that is part of the SubPage. This will always be a new instance. The new instance is then put back into the field or the property. This should mean that all references to those objects now refer to the *copy* in the SubPage, not to the original.

To indicate that a property or field contains an entity they should be annotated with a @UIReInject annotation:

```
public abstract class CdbFormPage<T extends AbstractBaseEntity> extends SubPage {
    @UIReinject
    private T model;
```

The subpage injector

The SubPageInjector class is responsible for checking everything related to SubPage data. It contains a list of ISubpageInjectorFactory instances, and each instance scans the SubPage class for properties or fields that might need manipulation. The default injectors scan for fields that contain entities, but you can easily add more by implementing a factory and adding it to the default SubPageInjector (obtained from DomApplication.getSubpageInjector()).

Exceptions from the injector

When the server is running in Development mode the injector will add injections for all fields, even the unannotated and unsuitable ones to make sure that none of them contain entities. If the checking injector detects an uninjected field containing an entity it will throw a SubFieldInjectionException describing the problem. This usually means that you either have to fix whatever prevents injection, or you need to explicitly specify @ReInject(false) to indicate you accept the fact that the field contains a reference inside another QDataContext (Session):

```
public abstract class CdbSubListPage<T extends AbstractBaseEntity,P> extends CdbListPage<T> {
    @UIReinject(false)
    private P parentModel;
```