Content app framework improvements

Child pages

Workshop 23.05.2018

The recordings of the workshop are available at the file server:

- afp://fileserver.magnolia-cms.com/All/Product Development/Knowledge Transfer/2018/20180522-UI-Workshop-part-1.mp4
- afp://fileserver.magnolia-cms.com/All/Product Development/Knowledge Transfer/2018/20180522-UI-workshop-part-2.mp4

Related repos:

Project with UI framework extensions and reference content app implementation: https://git.magnolia-cms.com/users/apchelintcev/repos/content-app-poc/browse

UI fork with necessary minimal changes: https://git.magnolia-cms.com/users/jsimak/repos/ui-bottom-lift/browse

Related JIRA issues:

MGNLUI-4958 - UI framework implementation and design upgrade IN PROGRESS

DEV-884 - Jira project doesn't exist or you don't have permission to view it.

DEV-885 - Jira project doesn't exist or you don't have permission to view it.

DEV-788 - Jira project doesn't exist or you don't have permission to view it.

DEV-910 - Jira project doesn't exist or you don't have permission to view it.

⚠ DEV-947 - Jira project doesn't exist or you don't have permission to view it.

Main goals

- Adapt UI framework to Vaadin 8 concepts
- Data binding in forms, dialogs and grids - simplify the existing solution

Model View Presenter

Initial investigation concept: Design patterns in UI content app

Reasons to bother	What changes	Benefits

• Way too many interfaces where a mere class would do (see IoC part as well)
• The way we used it before is cumbersome and not too beneficial.
• Concept of **Model** is not well defined.
• **Poor** component **isolation** patterns
  • views know about sub-views, presenters know about sub-presenters [... and sub-views].
  • poor state management (the parent views /presenters may just have most of the child view's state/business logic).
	• We try to prefer **classes over the interfaces**. If we introduce interfaces, we try to keep them small and function-oriented.
	• Instead of MVP we use something like **MVVMP** (Model-View-ViewModel-Presenter)
	  • **M**odel is the good ol' datasource like JCR workspace.
	  • **V**iew is now the entry point and the driver of the pattern. Views may compose and even interact with each other (though we we should not encourage that!)
	  • **V**iew**M**odel is the state of the View (all the properties that define the view in real time)
	  • **P**resenter - optional, completely isolated part which is used to help View interact with Model /ViewModel
	• We implement the ViewModel part via so called `ViewContexts` - interfaces that describe a certain part of mutable state (*selection*, *value*, *location etc*).
	  • We provide the **views** with the ability to **bind** contexts and **share** them with the sub-views (see more in the IoC section).
	  • For the ease of use, we do not enforce the developer to actually implement the interfaces, we **generate** them. ? We use some reactive technology for convenient subscription.
		• **Views** are more **component-like** and are easier to compose.
		  • Interfaces/classes do not expose methods like `#refresh()`, that encourage external control over the view.
		  • All the internal view **state management** happens in the **view itself** (no case when sub-app reacts on events and updates the state of the sub-views/components).
		  • All the necessary context can be injected via `ViewContexts`
		• Some synergy with how client-side frameworks manage the UI's (Redux/React contexts do similar things though in more conventional way)
		• Less code to write (less noisy listener interfaces, less abstractions)

Inversion of control (IoC) capabilities in UI

Reasons to bother	What changes	Outcome

• Too many Guice components
  • type mappings should just do the trick
• Guice does not support generics
• No good support for View-scoped components
  • Can only share sub-app components which is not enough often
• Hard to share such essential objects as definitions
	ViewProvider API
	
	Special factory API that creates the view with all the following additional features.
	
	Bean Storages for the views
	
	Each framework view gets the bean storage (same as app/sub-app) and can store objects in there. Each view can create sub-views and their storages will form an hierarchy with the parent's which makes it easy to traverse the parent storages from the child ones.
	
	Sharing objects between views
	
	Each view can bind view contexts (see MVP pattern changes) and publish other objects like definitions so that all the child views can access them effortlessly
	
	ComponentProvider for every view
	
	Each view gets its own `ComponentProvider` bound to the view's UI key. Each such component provider is enhanced with two additional `ParameterResolvers`:
	
	• `ViewContextParameterResolver` - upon `ComponentProvider#newInstance` calls this one looks up the constructor dependencies from the bean storage hierarchy (e.g. shared with the mechanism above).
	• `DatasourceComponentParameterResolver` - provides support for the datasource related components injection (see ContentConnector changes description)
		Benefits:
		
		• Ability to share configuration/resources between the views without coding overhead.
		  • E.g. a root view can share some context; and then all the sub-views on all the levels can inject that context without any need to pass it through all the intermediate parents.
		• Better support for the pattern improvements described above.
		
		Questions:
		
		• ? How to know when to clean up the view context?
		  • We only have one similar example - choosers, their context is cleaned up when dialog is closed.
		  • Should we also attach this logic to maybe Vaadin attach/detach events?
		  • Should we communicate that the views /sub-views should be closed manually?
		• ? How to attach the sub-views properly?
		  • should we do it manually?
		  • should we pass the parent container /attachment lambda? (same might work for closing of the view as well)
		• ? We probably require additional tooling to make the thing

ContentConnector

Reasons to bother	What changes	Outcome

• CC seems to be **incomplete**, over-promising interface
  • while claiming to be "the connection" to the data source it merely provides the conversion between different forms of the item (id → Vaadin Item → url fragment and back);
  • CC doesn't provide any child/parent relations API;
• part of CC's **functionality** becomes **obsolete** with **Vaadin 8**
  • Vaadin Items are no longer there, the corresponding CC methods can /should go too;
• It is **hard to extend** the CC interface. E.g. if we want to add API that provides support for item version look-up or support for item creation - we have to create the extending interface and then cast the CC to the extension;
• CC is **not generic** (partially cause it is not possible to inject generics with Guice);
	Instead of single CC interface we can introduce a concept of a `DataSourceSupport`:
	
	• it would be based on an arbitrary DS definition attached to app/sub-app/view
	• it would manage so-called 'bundles' of utilities that provide various functionality realted to the data source.
	  • once an app with JCR DS configured needs e.g. a Vaadin `HierarchicalDataProvider`, we can ask the DS support to give one. It looks up the bundle corresponding to JCR DS definition and sees if the bundle provides an HDataProvider.
	  • Bundles are completely arbitrary, they may contain domain specific utlitites and components
		Benefits
		
		• + Flexible and versatile provisioning of DS components (properysets, data providers, hierarchy look-ups, to-string converters etc)
		
		Concerns
		
		• ? Such solution adds a bit of magic - DSSupport and DS Bundles are yet two more layers of indirection, though still they typically should be concealed behind IoC - maybe that is not a big deal?
		• ⚠ It wouldn't be easy to provision generic DS components via Guice, the #newInstance(...) variant will work just fine.

Data binding in grids and tree grids

Reasons to bother	What changes	
Vaadin 8 brings in new concepts to the data binding • no more Item abstraction → no more JcrNodeAdapter • no more Container → no more JcrContainer	**DataProvider** • Container concept replacement: interface for querying the data from a datasource • Works with domain objects (e.g. JCR Nodes) instead of Items • Stateless by design (one can implement caching manually though) **PropertySet** • Item concept replacement • Unlike Item is more or less stateless component itself - only describes how to interact with the domain object to access [and modify] its properties via funtional primitives (lambdas), e.g. via bean getter/setter method references.	**Benefits** • Simpler and slimmer way to connect to the datasources • Less memory consumption (no additional abstractions over the domain objects) • Easier to implement various providers (REST, ORM...) • Supposedly better implementation hierarchical data providers **Concerns** • ⚠️ DataProvider still requires item indexing API • ⚠️ Size query API still needs to be implemented (bummer for JCR list views) • ❓ Observation of the DataProvider changes is still to be drafted!

Content change events

Reasons to bother	What changes	Outcome
Quirky way of communicating the data source changes • Requires **manual** actions of both **sending** the events and **handling** them. I.e. if I change smth in JCR and forget to send CCE, no one will ever know. • Apps that are not related to the changed data source still have to handle its CCE and see if they should react (hence `ContentConnector#canHandleItem`).	We should try to expose the data source observation utilities and subscribe to them where needed. • e.g. we have already the JCR observation mechanism that Vaadin's `DataProvider` can subscribe to in this or that way and just push the change notification to the UI automatically. • Whenever we change smth in the data source the UI is notified eventually. • For the data sources that do not have real time observation, we could easily provide a timer-based implementation that merely refreshes the UI periodically.	**Benefits** • Clear and re-usable data source observation mechanism. • Less boilerplate for the app developers **Questions** • ❓ Need to provide un-registration strategy. For some cases a mere weak-hashmap solution might work ❓. We though should also make sure that once the view is dead, no observation left-overs are still hanging around

Field definitions

Reasons to bother	What changes	Benefits
• Not generic • value type is communicated in "JCR" style, i.e. via strings	`FieldDefinition` becomes generic	Better compatibility with the new, more type-safe Vaadin data-binding API's

Form definition

Reasons to bother	What changes	Outcome

• Form definition is nailed to the concept of the tabs
  • presentation definition is entangled with the model
	`FormDefinition` separates field/property definitions from the layout definition
	• Tabbed layout becomes just a concrete case of the layout definition
	• We provide other types of layouting possibilities - HTML/Vaadin declarative layout/Custom Component
	
	? Possibly we introduce an alternative term 'editor' which is more generic than the form
		Benefits
		• More flexibility
		• Possibility to define complex fields with the same definitions that we use for the forms
		  • i.e. complex fields also become forms
		
		Questions
		• ? How to provide compatibility between the current form definitions and the new ones?
		  • Resolve on programmatic level, i.e. allow special definitions that take the old one and then the app/dialog transforms them into the new ones?

I18N support in dialogs

Column definitions

Complex fields