Dependency injection additions

Best practices changes

We aim for maximum configurability and flexibility of out UI views/presenters. Shipping them as Guice components like below seems to go against such idea:

Module descriptor excerpt

```
    <component>
      <type>info.magnolia.ui.contentapp.browser.BrowserView</type>
      <implementation>info.magnolia.ui.contentapp.browser.BrowserViewImpl</implementation>
    </component>

    <component>
      <type>info.magnolia.ui.contentapp.ContentSubAppView</type>
      <implementation>info.magnolia.ui.contentapp.ContentSubAppViewImpl</implementation>
    </component>

    <component>
      <type>info.magnolia.ui.contentapp.browser.BrowserPresenter</type>
      <implementation>info.magnolia.ui.contentapp.browser.BrowserPresenter</implementation>
    </component>
```

- It only complicates overriding them with custom extensions
- Views/presenters never have any scope attached to them (i.e. they're always one-shot instances)
- It is hard to provide required information to UI components with Guice (e.g. typically some sort of definition is required and then we inject the sub-app descriptor just cause it is the only injectable definition, and read sub-definitions from there).
- Shipping views as Guice components restricts usage of generics: one cannot inject e.g `TreeView<T> treeView` since Guice needs to know explicit generic type upon injector creation (which will fail blaming such arg as under-specified).

In order to mitigate the above and in order to avoid weird patterns when presenter first injects the view somehow, and then initialises it with an obscure and bulky method like e.g.:

`public TreeView start(WorkbenchDefinition workbenchDefinition, EventBus eventBus, String viewTypeName, ContentConnector contentConnector)`

it is advised to follow the practices described in Design patterns in UI content app and use the additional view API from View interface extensions.

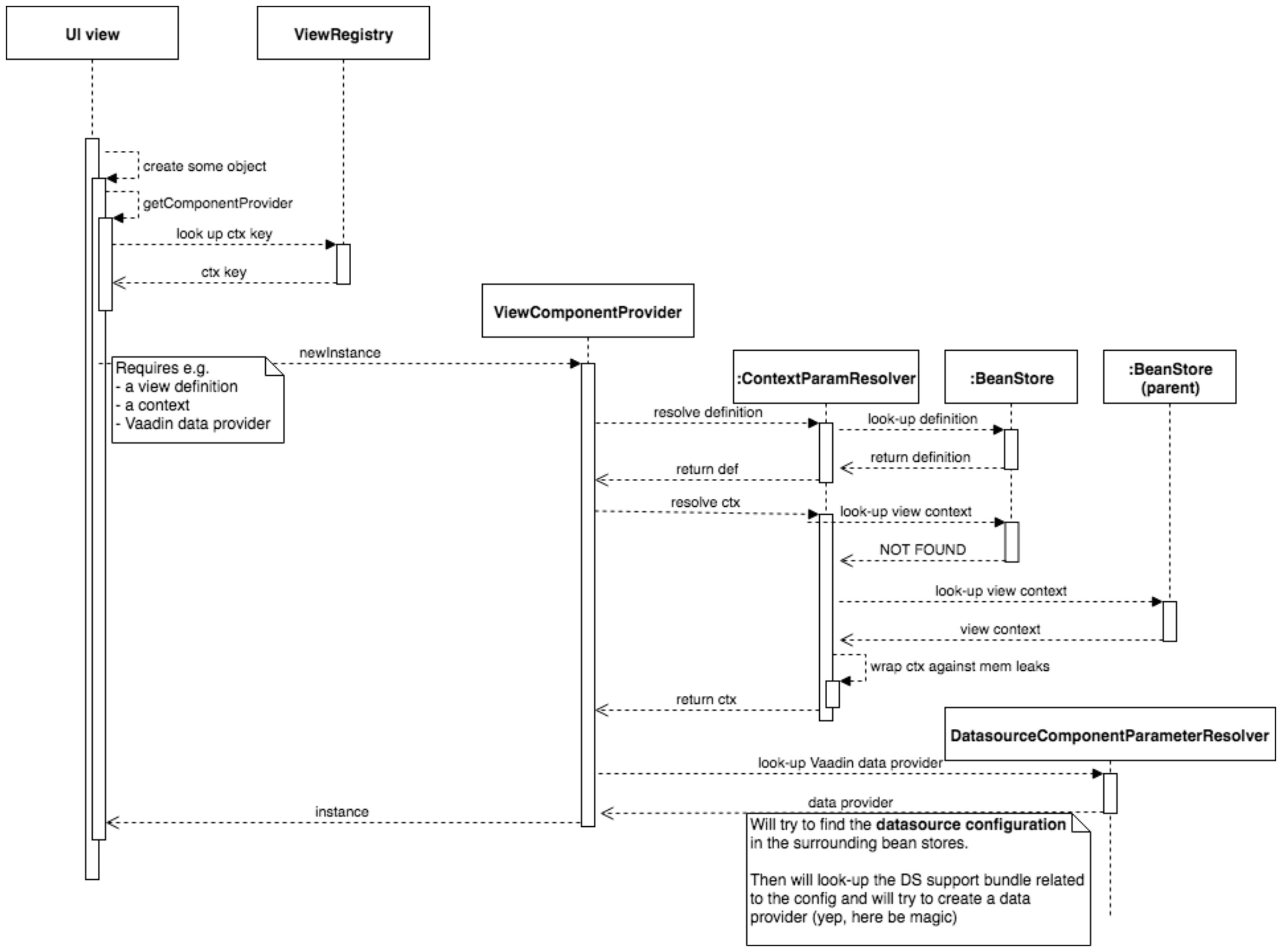
Additional points:

- `WithImplementation` interface proposed to 'standardise' the definitions which incorporate implementation type. `UiFrameworkView` has convenience API that allows to 'un-pack' and instantiate objects from such definitions. `ViewDefinition` of course implements it.
- ❓ There's no convention re: presenter provisioning, typically views just create them manually atm, but we could also introduce smth like `PresentedView` which allows to specify (and configure) the presenter class on definition level and then auto-create it with the view.

ComponentProvider goodies

As it is mentioned in View interface extensions concept, each framework view is automatically provisioned with its own `ComponentProvider` instance (called `ViewComponentProvider`). It roughly the same thing as any other `UiContextBoundComponentProvider`, i.e. it'll point Guice to look for scoped instances in the session store. The distinguishing difference of `ViewComponentProvider` though, that it adds two additional `ParameterResolvers` to the blend, making `ComponentProvider#newInstance` API more powerful in case of the views:

- `ViewContextParameterResolver` looks up c-tor arguments in the bean stores that are in the context of the current views (its own and its parents), which allows us to avoid having to pass around a bunch of objects like *definitions, selected items* etc.
- `DatasourceComponentParameterResolver` described in Datasource bundles



Example: injection of definition and value context in `TreeView`

```
...
// parent view will create TreeView like this (not even knowing that it is a TreeView).
getViewProvider().create(contentViewDefinition);
...
// The definition will be resolved as an explicit instance passed to #create(...) method. ValueContext will be
resolved directly from the sub-apps root view
public TreeView(ValueContext<T> valueContext, TreeViewDefinition<T> definition)
```