New concept for choose dialogs.

Why we need a new concept?

Current implementation of choose dialog somewhat fights against the framework:

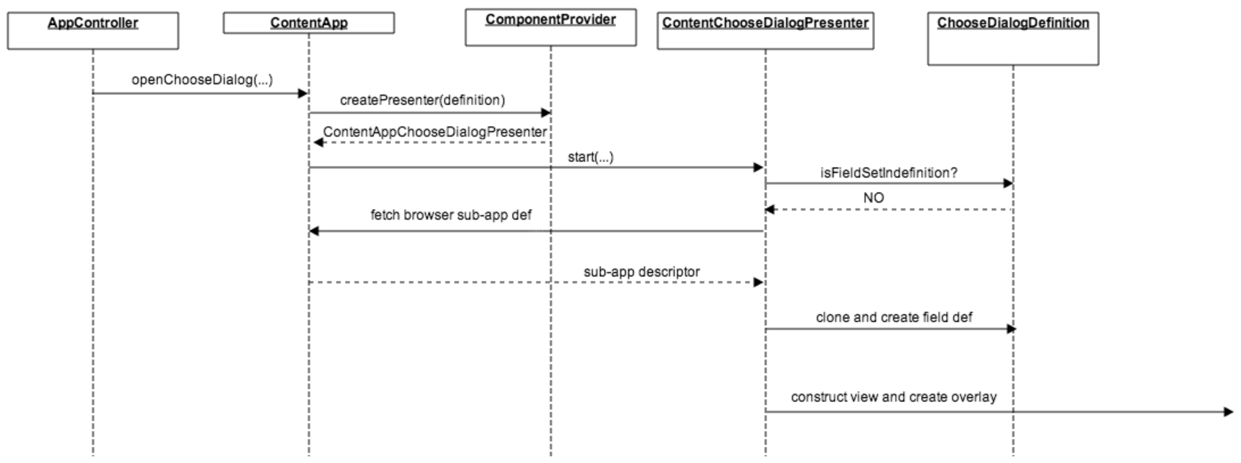
- No configuration in the tree is possible.
- Choose dialog actions are hard-coded and give no freedom in modifying the logic of choosing.
- Hacky and complex implementation of choose dialog for a content app is hidden behind a minimalistic ChooseDialogPresenter interface.
    - By default, when a content app is asked to provide a choose dialog - it duplicates a workbench and img provider def.
    - It uses WorkbenchChooseDialogPresenter and does all configuration in ContentApp class.
        - Child classes (e.g. AssetsApp) don't have a direct access to the implementation - everything has to be casted and a lot of assumptions have to be made in order to introduce modification.
- Choose dialog had to be closed manually by the consumer.

What can be done?

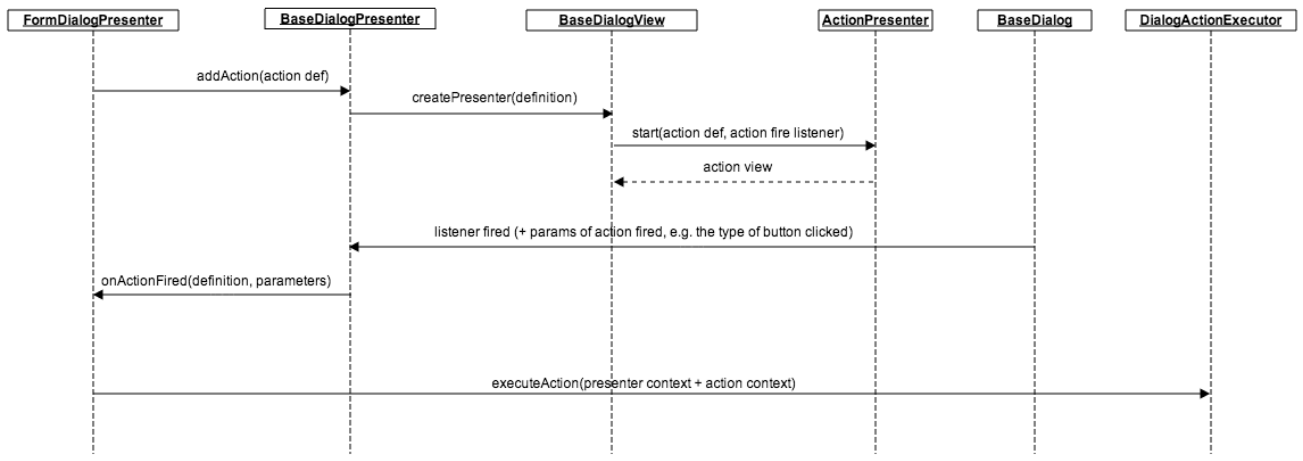
- **We can treat choose dialogs the same way we treat form dialogs.**
    - **UPDATED:** Content app `AppDescriptor` will contain definition of a choose dialog.
        - AppDescriptor will be extended for the case of content app.
        - AppContext has to expose AppDescriptor in order to let apps access it (the method was lost a long time ago during AppInstanceController introduction refactoring).
        - Choose dialog configuration can now reside on an app level.
        - ~~We extract the generic registry for base dialogs out of the form dialog registry and create choose dialog registry on its basis.~~
        - ~~Definition managers and providers can be created the same way.~~
    - Contains configurable actions (e.g. ChooseDialogCallback actions).
    - Configurable choose dialog UI-content.
        - Most configs like actions, title and label can be shared between form and choose dialogs.
        - The main difference - instead of configuring the whole form we only need one field, whose value would be used for selection.
        - Workbench can be adapted to a CustomField fairly easy (at least without modifying the workbench itself).
    - We set up a registry for choose dialogs analogous to the form dialog registry (they extend all the base classes).
    - We do not use a ChoseDialogPresenterFactory anymore.
        - In order to get a different choose dialog for an app a developer would have to implement a new factory and we don't want that.
        - We inject a ChooseDialogPresenter instead and it builds the view based on provided definition.
    - Choose dialogs can be now called by name (and/or by definition).
        - Looks like we can now have several different choose dialogs for a single app.
    - We can actually keep the old mechanism of creating a choose dialog as a fallback - in case absolutely no choose dialog is defined for an app at least try to construct it from the browser sub-app.
    - Actual choose process happens by means newly introduced ChooseDialogCallbackActions - similar to those DialogCallbackActions that fire a callback.
        - The only difference more or less is that besides an action name we pass a chosen value into the callback (maybe there should be some kind of a callback context object so it would look prettier).
        - Additional choose logic can be added effortlessly to these action.
            - To close or not to close dialog.
            - To accept selection or not to accept selection (had a request for forbidding selection nodes without children just yesterday, the solution involved creation of custom ContentPresenters).
            - To fire additional notifications if needed.
    - **UPDATE:** Customizable presenters.
        - Dialog presenters can be configured so that we could provide more complex dialogs.
        - It was requested by UX team multiple times that dialog actions are separated into groups in the footer.
            - DialogDefinition can have a list of secondary actions and DialogPresenter dispatches them into a proper place in the view.
        - There could be a potential benefit in configuring even dialog actions - instead of buttons we could provide combo-boxes and e.g. uploads.
            - A map of ActionPresenters is contained in the dialog.
            - Only special presenter are kept in the map, for all the rest - default presenter (a mere button).
        - BaseDialog should not do any action management, it should be just a chrome for dialogs.
            - All action management is done by presenter - listen for an action to be triggered by UI (click listener).
                - Callback from UI might include an array of additional parameters (type of mouse button clicked, id of chosen value in combobox). These parameters are then automatically merged with action executor's parameters and thus can be used in actions with injection.
            - Job of DialogView - to put an action view into the correct place and to bind the UI-listener.
            - Job of BaseDialog - to host Vaadin components.

Diagrams

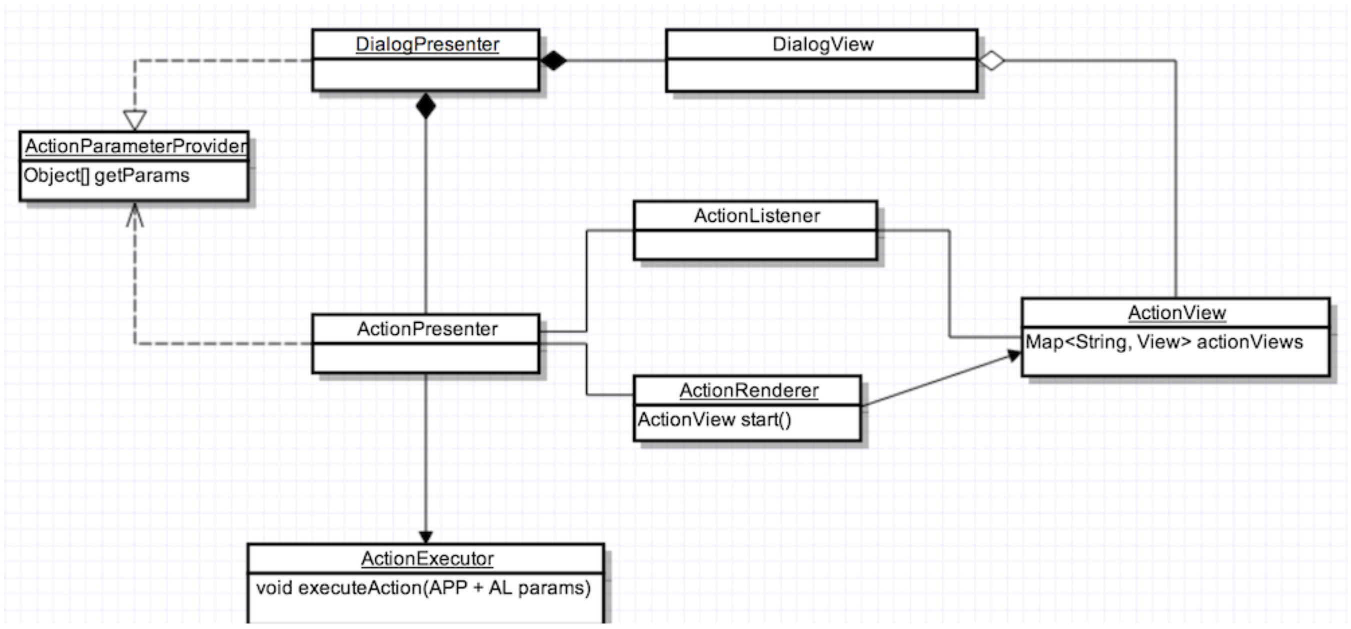
Choose dialog for content apps


AppController
ContentApp
ComponentProvider
ContentChooseDialogPresenter
ChooseDialogDefinition
openChooseDialog(...)
createPresenter(definition)
ContentAppChooseDialogPresenter
start(...)
isFieldSetIndefinition?
NO
fetch browser sub-app def
sub-app descriptor
clone and create field def
construct view and create overlay

Action Execution Workflow


FormDialogPresenter
BaseDialogPresenter
BaseDialogView
ActionPresenter
BaseDialog
DialogActionExecutor
addAction(action def)
createPresenter(definition)
start(action def, action fire listener)
action view
listener fired (+ params of action fired, e.g. the type of button clicked)
onActionFired(definition, parameters)
executeAction(presenter context + action context)

New dialog stack outline.


DialogPresenter
DialogView
ActionParameterProvider
Object[] getParams
ActionListener
ActionView
Map<String, View> actionViews
ActionPresenter
ActionRenderer
ActionView start()
ActionExecutor
void executeAction(APP + AL params)