View interface extensions

`View` interface in current state of framework is rather simplistic (only allows to expose view as a Vaadin component). As part of the improvements we propose introduce an extension of the view interface with the working title `UiFrameworkView`. Its main purpose is to connect views to the IoC mechanism and `SessionStore`.

UiFrameworkView

```
public interface UiFrameworkView extends View {

    @Override
    default Component asVaadinComponent() {
                ...
    }

    /**
     * Cleans up the related bean store, un-registers this
     * view from the {@link ViewContextKeyRegistry view registry},
     * detaches the from the parent Vaadin component,
     * recursively destroys the child views.
     */
    default void destroy() {
                ...
    }

    /**
     * Put an instance into the view's bean storage, making it
     * injectable with {@link ViewComponentProvider}.
     */
    default <T> void bindInstance(Class<T> type, T instance) {
        accessViewBeanStore().put(type, instance);
    }

    /**
     * Create and bind an instance of the view context. The provided argument is typically an interface,
     * whose implementation is auto-generated on the fly.
     *
     * The instance of the view context is the stored in the view's bean storage and share
     * in similar fashion link {@link #bindInstance(Class, Object)} does.
     */
    default <T extends ViewContext> T bindContext(Class<? extends T> contextClass) {
        final T context = new ViewContextProxy().createViewContext(contextClass);
        accessViewBeanStore().put(contextClass, context);
        return context;
    }

    /**
     * Shares datasource definition information with the sub-views and components.
     * Datasource definition provides framework with the hints regarding which
     * implementation to choose for the domain-specific interfaces.
     */
    default void bindDatasourceDefinition(Object definition) {
        accessViewBeanStore().put(DatasourceHolder.class, new DatasourceHolder(definition));
    }

    /**
     * Convenience wrapper around {@link ComponentProvider} capabilities.
     * Creates and instance of a type specified by a passed {@link WithImplementation} instance.
     */
    default <T> T create(WithImplementation<T> definition, Object... args) {
        return getComponentProvider().newInstance(definition.getImplementationClass(), Util.appendToArray(args,
definition));
    }

    /**
     * Convenience wrapper around {@link ComponentProvider} capabilities.
     */
    default <T> T create(Class type, Object... args) {
```

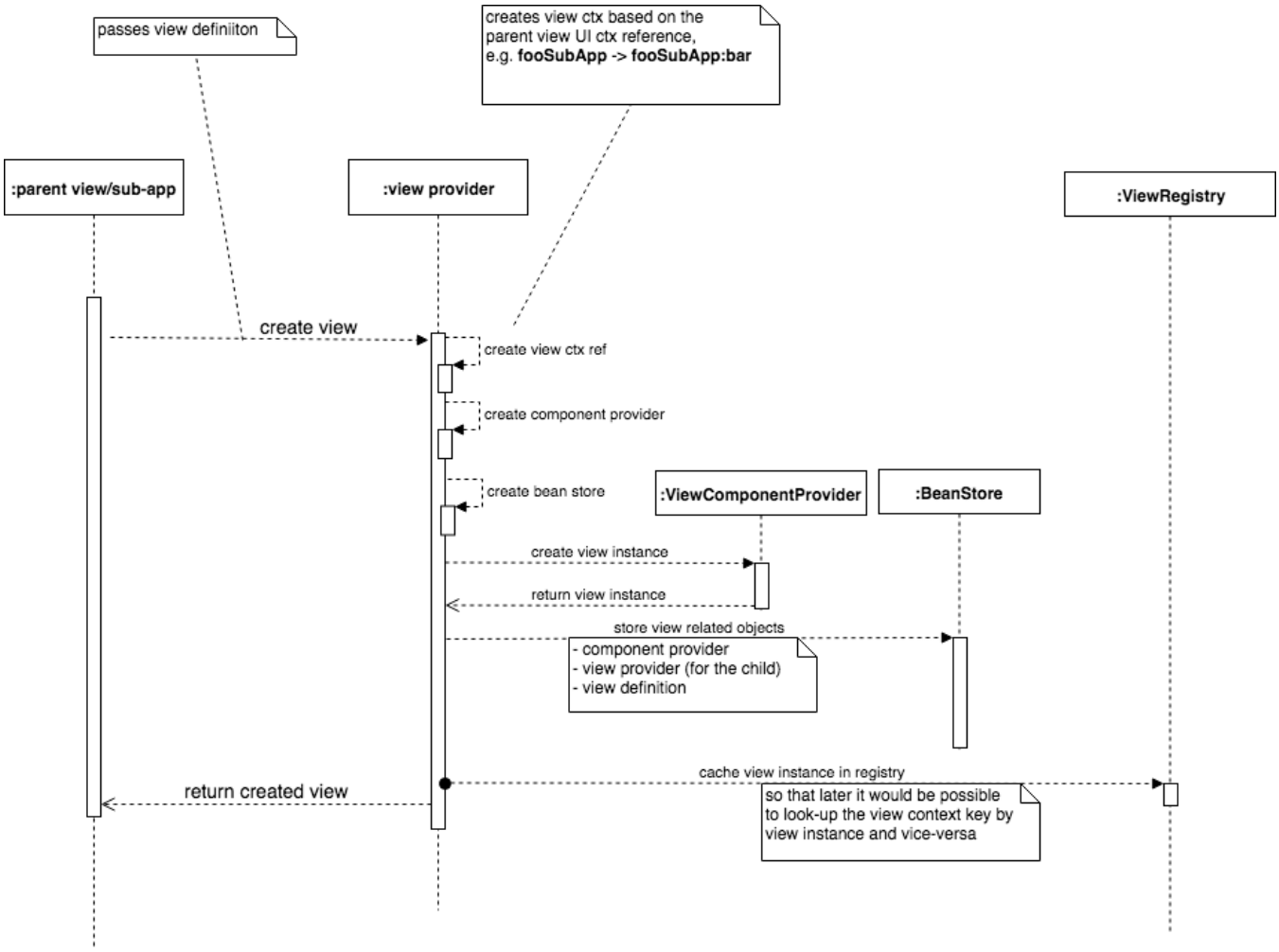
```
        //noinspection unchecked
        return getComponentProvider().newInstance((Class<T>) type, args);
    }

    default ViewProvider getViewProvider() {
        return new ViewProviderImpl(getCurrentViewReference());
    }

    default ComponentProvider getComponentProvider() {
        return new ViewComponentProvider(getCurrentViewReference());
    }

    default UiContextReference getCurrentViewReference() {
        return ViewContextKeyRegistry.access()
                .lookUp(this)
                .orElseGet(() -> CurrentUiContextReference.get().getUiContextReference());
    }

    default BeanStore accessViewBeanStore() {
        return SessionStore.access().getBeanStore(getCurrentViewReference());
    }
}
```

The interface is self-sufficient, all the extension methods are provided as default interface methods, so that the implementing types (view impls) are not enforces to extend any abstract super-class. Typically, a view implementation can extend a Vaadin component class (e.g. CustomComponent) and by implementing `UiFrameworkView` each such impl is automatically supplied with the following:

- `BeanStore` in the http session
- View-bound `ComponentProvider`
- `ViewProvider` utility, which allows to create child views

In order to align the views with the session store, each view has to be associated with a unique `UiContextReference`. Since we ship extensions as an interface, we cannot store such reference in the view itself. In order to persist the association, we introduce the special `ViewContextKeyRegistry` object, cached in the UI bean storage (i.e. there's one instance of such view registry per browser tab). `ViewProvider` automatically registers the view upon creation, later when the view is remove (via `destroy()` method), the association is removed.

Let us observe the enhanced view creation process in more detail below:



Creation of a child view

```
// We are in a context of a view responsible for the display of the content views
....
final ContentViewDefinition<T> contentViewDefinition = definition.getViews().get(viewId);
this.currentActiveContentView = getViewProvider().create(contentViewDefinition);
...
```