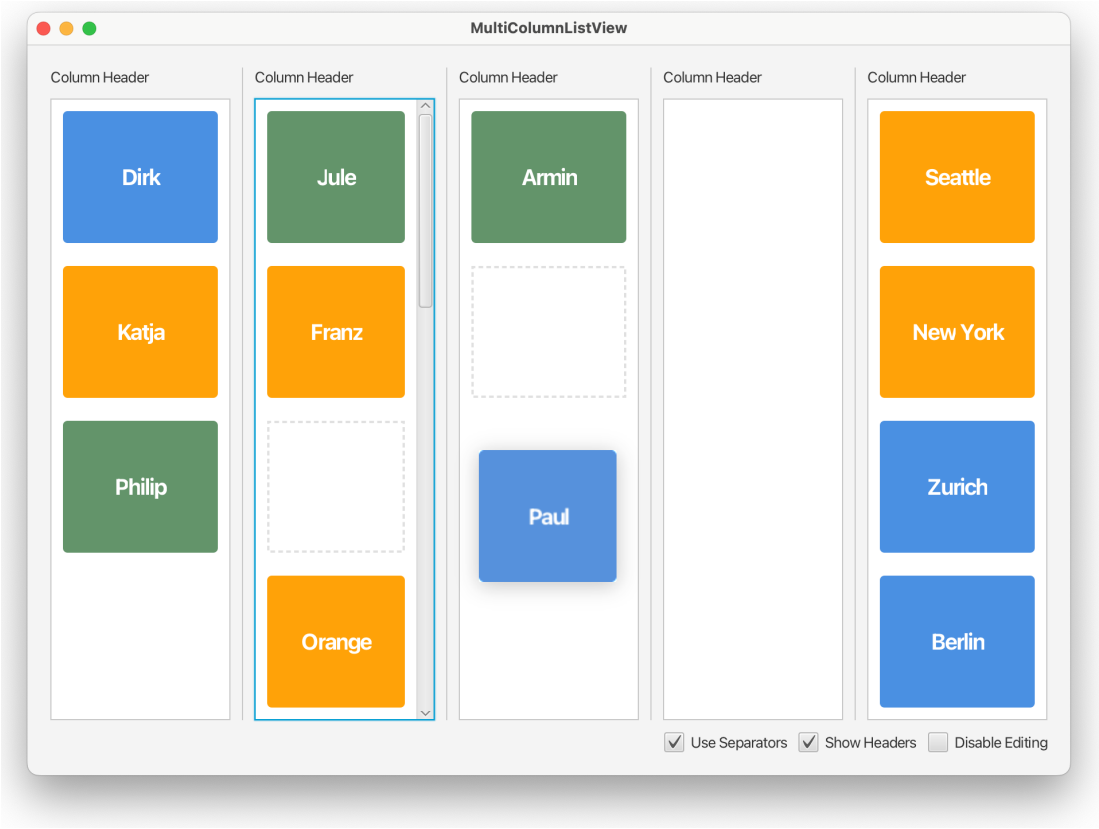
MultiColumnListView

Developer Manual

GemsFX • com.dlsc.gemsfx • JavaFX Custom Controls



A MultiColumnListView with five columns, headers and separators.

Table of Contents

1. Introduction

The **MultiColumnListView** is a JavaFX control that displays a configurable number of columns, where each column consists of an optional header node and a `ListView`. Users can reorder items inside a column and drag items from one column to another, which makes the control a natural fit for Kanban boards, issue trackers, task planners, and any other workflow that moves entities through a series of states.

The control lives in the package `com.dlsc.gemsfx`, its skin in `com.dlsc.gemsfx.skins.MultiColumnListViewSkin`, and its default stylesheet in `com/dlsc/gemsfx/multi-column-list-view.css`.

1.1 Key features

- Any number of columns, each with its own header node, item list and optional user object.
- Built-in drag and drop between and within columns, including visual "from" and "to" drop markers.
- Two callbacks to veto drags and drops (`dragPossibleCallback`, `dropPossibleCallback`).
- A rich event API (`MultiColumnListViewEvent`) covering the whole drag and drop lifecycle.
- Pluggable factories for list views, cells and column separators.
- A placeholder node that is shown when no columns have been defined.
- Integrated loading state via `LoadingPane` (status, size, progress indicator).
- Two styleable CSS properties: `-fx-show-headers` and `-fx-disable-drag-and-drop`.
- Localized default texts and an accessible role of `AccessibleRole.LIST_VIEW`.

1.2 Type parameter

`MultiColumnListView<T>` is generic. `T` is the type of the application's own model objects, for example *Issue*, *Ticket* or *Task*. Applications only ever deal with objects of type `T`; the wrapping into `ColumnItem<T>` instances is an implementation detail of the control (see chapter 6).

1.3 Maven dependency

```
<dependency>
    <groupId>com.dlsc.gemsfx</groupId>
    <artifactId>gemsfx</artifactId>
    <version>${gemsfx.version}</version>
</dependency>
```

When using the Java Platform Module System, add `requires com.dlsc.gemsfx;` to your *module-info.java*.

2. Getting started

A minimal setup consists of creating the control, creating one `ListViewColumn` per column, filling the columns with model objects and adding the columns to the view.

```
MultiColumnListView<Issue> view = new MultiColumnListView<>();

ListViewColumn<Issue> todo = new ListViewColumn<>();
todo.setHeader(new Label("To Do"));
todo.setUserObject("todo");
todo.getItems().setAll(new Issue("Fix login"), new Issue("Update docs"));

ListViewColumn<Issue> inProgress = new ListViewColumn<>();
inProgress.setHeader(new Label("In Progress"));
```

```
inProgress.setUserObject("in-progress");

ListViewColumn<Issue> done = new ListViewColumn<>();
done.setHeader(new Label("Done"));
done.setUserObject("done");

view.getColumns().setAll(todo, inProgress, done);
```

The control sizes itself like any other JavaFX control. Inside a `VBox` it is usually combined with `VBox.setVgrow(view, Priority.ALWAYS)` so that the columns fill the available height. All columns receive the same width because the skin assigns identical column constraints to every column.

> The control sets **focusTraversable** to `false`. Keyboard focus is handled by the individual list views inside the columns.

3. Control API

3.1 Properties of MultiColumnListView

Property	Type	Default	Description
`columns`	ListProperty<ListViewColumn<T>>	empty	The columns shown by the view. Each column carries its own header, items and user object.
`placeholder`	ObjectProperty<Node>	dashed label	Node shown instead of the columns while the column list is empty. Set to `null` to show nothing.
`showHeaders`	BooleanProperty (styleable)	true	Whether the column headers are shown. Toggling triggers a rebuild of the view.
`disableDragAndDrop`	BooleanProperty (styleable)	false	Turns drag and drop off completely, making the view read-only.
`dragPossibleCallback`	ObjectProperty<Callback<T, Boolean>>	item -> true	Vetoes the start of a drag operation for individual items.
`dropPossibleCallback`	ObjectProperty<Callback<DropParameter<T>, Boolean>>	param -> true	Vetoes a drop on a given target column.
`cellFactory`	ObjectProperty<Callback<MultiColumnListView<T>, ColumnListCell<T>>>	ColumnListCell::new	Creates the cells used by all columns.
`listViewFactory`	ObjectProperty<Callback<MultiColumnListView<T>, ListView<ColumnItem<T>>>>	AutoscrollListView	Creates the `ListView` for each column.
`separatorFactory`	ObjectProperty<Callback<Integer, Node>>	region with style class "column-separator"	Creates the node placed between two columns. Set to `null` for no separators.
`draggedItem`	ObjectProperty<T>	null	The model object that is currently being dragged.
`loadingStatus`	ObjectProperty<LoadingPane.Status>	Status.OK	Switches the internal `LoadingPane` between content, progress indicator and error.
`loadingStatusSize`	ObjectProperty<LoadingPane.Size>	Size.MEDIUM	Visual size of the loading indicator.
`progressIndicator`	ObjectProperty<ProgressIndicator>	CircleProgressIndicator	The indicator shown while loading.

3.2 Additional methods

Member	Returns	Description
`getDraggedItems()`	ObservableList<T>	All selected model objects that participate in the current drag gesture.
`getFromPlaceholder()`	ColumnItem<T>	The marker item visualizing the origin of the ongoing drag operation.
`getToPlaceholder()`	ColumnItem<T>	The marker item visualizing the future drop location.
`getDraggedColumnItem()`	ColumnItem<T>	The wrapper of the item that is currently being dragged, or `null`.
`getClassCssMetaData()`	List<CssMetaData>	The styleable properties contributed by this control.

3.3 ListViewColumn<T>

A column is a pure model object — it is not a node. It stores the data and the header of a single column.

Member	Type	Description
`items`	ListProperty<T>	The model objects shown in this column. This is the list applications read from and write to.
`header`	ObjectProperty<Node>	The node shown above the column. Defaults to a label reading "Column Header".
`userObject`	ObjectProperty<Object>	Optional application data attached to the column, e.g. an identifier or the represented status.
`getItemWrappers()`	ObservableList<ColumnItem<T>>	Read-only view of the internal wrapper list kept in sync with `items`. Managed by the control.

> Always modify **items**, never **itemWrappers**. The wrapper list is synchronized automatically in both directions and may temporarily contain the two drag and drop marker items.

4. Drag and drop

Drag and drop is enabled by default. When the user starts to drag a cell, the control replaces the dragged item with the "from" marker and inserts the "to" marker at the position the item would land in. Both markers are rendered by ordinary cells, which can be styled through the CSS pseudo classes `:from` and `:to`.

4.1 Restricting drags and drops

Two callbacks control what may be dragged and where it may be dropped. Both return `true` by default.

```
// only items that are not yet finished may be dragged
view.setDragPossibleCallback(issue -> !"done".equals(issue.getStatus()));

// a column may not hold more than ten items
view.setDropPossibleCallback(param -> param.getColumn().getItems().size() < 10);
```

`DropParameter<T>` carries the item currently under the mouse pointer (`getItem()`) and the target column (`getColumn()`). When the callback returns `false`, neither the drop markers appear nor is the drop performed.

To switch drag and drop off entirely, use `view.setDisableDragAndDrop(true)` or the CSS property `-fx-disable-drag-and-drop: true;`.

4.2 Events

The view fires `MultiColumnListViewEvent` instances for every relevant step, including the steps rejected by the callbacks above. Each event exposes `getDraggedItem()`, `getColumn()` and `getIndex()`.

Event type	Fired when
`ANY`	Super type of all events of this control — register here to observe everything.
`DRAG_STARTED`	A drag gesture started and `dragPossibleCallback` allowed it.
`DRAG_NOT_POSSIBLE`	A drag gesture was rejected by `dragPossibleCallback`.
`DRAG_OVER`	A dragged item hovers over a valid drop location.
`DROP_NOT_POSSIBLE`	A location was rejected by `dropPossibleCallback`, either while hovering or on drop.
`ITEM_MOVED`	An item was dropped successfully and now sits in a new column / at a new index.
`DRAG_ENDED`	Reserved for applications implementing their own drag and drop handling; not fired by the control itself.

```
view.addEventHandler(MultiColumnListViewEvent.ITEM_MOVED, evt ->
        System.out.println(evt.getDraggedItem()
                + " moved to column " + evt.getColumn().getUserObject()
                + " at index " + evt.getIndex()));

// a single handler for every drag and drop step
view.addEventHandler(MultiColumnListViewEvent.ANY, System.out::println);
```

ITEM_MOVED is the right place to persist a change. Because the wrapper list is kept in sync with `ListViewColumn.getItems()`, the item lists of the involved columns are already up to date when the event is delivered.

4.3 Multi-item drags

When a drag starts, the control copies all selected model objects of the source list view into `getDraggedItems()`. The default implementation moves only the single item under the mouse pointer;

applications that want true multi-item moves can read this list in an `ITEM_MOVED` handler and move the remaining items themselves.

5. Customizing the cells

Cells are created by the `cellFactory` and must be instances of `ColumnListCell<T>`, because that class carries the entire drag and drop implementation. Subclasses override `updateUserObject(T userObject, boolean empty)` instead of `updateItem(...)`, which is `final`.

```
public class IssueListCell extends ColumnListCell<Issue> {

    public IssueListCell(MultiColumnListView<Issue> view) {
        super(view);
        getStyleClass().add("issue-list-cell");
    }

    @Override
    protected void updateUserObject(Issue issue, boolean empty) {
        getStyleClass().removeAll("todo", "in-progress", "done");

        if (isFromPlaceholder()) {
            setText("From");
        } else if (isToPlaceholder()) {
            setText("To");
        } else if (issue != null && !empty) {
            setText(issue.getTitle() + " (" + issue.getStatus() + ")");
            getStyleClass().add(issue.getStatus());
        } else {
            setText("");
        }
    }
}

view.setCellFactory(mclv -> new IssueListCell(mclv));
```

5.1 Useful members of ColumnListCell

Member	Description
`updateUserObject(T, boolean)`	Override point called whenever the cell's item changes. The model object is `null` for empty cells and for both markers.
`getUserObject()`	The model object shown by this cell, or `null` for empty / marker cells.
`getColumn()`	The `ListViewColumn` this cell belongs to.
`getMultiColumnListView()`	The control the cell is used in.
`placeholderProperty()`	Read-only, true while the cell shows either marker ("from" or "to").
`fromPlaceholderProperty()`	Read-only, true while the cell shows the "from" marker.
`toPlaceholderProperty()`	Read-only, true while the cell shows the "to" marker.
`getContentPane()` / `getContentLabel()`	The dashed pane and centered label used by the default rendering; reusable by subclasses.
`getSnapshotNode()`	Override to control which node is snapshotted for the drag image (default: the cell itself).

> If a custom cell renders its content into a wrapper node, override `getSnapshotNode()` and return that wrapper. Otherwise the drag image may include the cell's own background and selection decoration.

5.2 Custom list views and separators

The `listViewFactory` creates one list view per column; the default returns an `AutoscrollListView`, which scrolls automatically while the user drags near the top or bottom edge. Use the factory to configure the selection mode or to install a custom placeholder for empty columns.

```
view.setListViewFactory(mclv -> {
    AutoscrollListView<ColumnItem<Issue>> listView = new AutoscrollListView<>();
    listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
    return listView;
});

// separators between the columns
view.setSeparatorFactory(index -> new Separator(Orientation.VERTICAL));
view.setSeparatorFactory(null); // no separators at all
```

6. Internals: ColumnItem

Internally every model object is wrapped in a `ColumnItem<T>`. The wrappers allow the control to insert the two drag and drop markers into a list view without requiring the application to provide model objects for them. The wrapper list of a column is kept in sync with its item list in both directions, and existing wrappers are reused whenever possible so that cells are not rebuilt unnecessarily.

Member	Description
`getUserObject()`	The wrapped model object; `null` for both markers.
`isPlaceholder()`	True if the item is the "from" or the "to" marker.
`isFromPlaceholder()`	True if the item marks the origin of the ongoing drag operation.
`isToPlaceholder()`	True if the item marks the prospective drop location.

Applications normally never create `ColumnItem` instances themselves. They encounter the type only in the signatures of the `listViewFactory` and when working directly with a column's list view.

7. Styling

The control ships with the stylesheet `multi-column-list-view.css`, which is returned by `getUserAgentStylesheet()` and therefore applied automatically.

7.1 Style classes and pseudo classes

Selector	Applies to
`.multi-column-list-view`	The control itself.
`.multi-column-list-view > .placeholder`	The node shown when no columns are defined.
`.grid-pane`	The internal grid pane laying out headers, columns and separators.
`.column-separator`	The default separator region between two columns.
`.column-background-region`	A region behind each column, spanning header and list.
`.column-foreground-region`	A mouse-transparent region above each column; gets the `:hover` pseudo class while the mouse or a drag hovers the column.
`.column-list-cell`	Every cell of every column.
`.column-list-cell > .content-pane`	The dashed pane used by the default cell rendering.
`.content-label`	The centered label inside the content pane.
`.column-list-cell:from`	A cell showing the "from" marker.
`.column-list-cell:to`	A cell showing the "to" marker.
`.list-view .placeholder:drag-over`	The empty-column placeholder while a valid drag hovers over it.

7.2 Styleable CSS properties

CSS property	Type	Default	Java property
`-fx-show-headers`	Boolean	true	showHeaders
`-fx-disable-drag-and-drop`	Boolean	false	disableDragAndDrop

```
.multi-column-list-view {
    -fx-show-headers: true;
    -fx-disable-drag-and-drop: false;
}

.multi-column-list-view .column-list-cell:to > .content-pane {
    -fx-border-color: green;
}
```

8. Loading state, placeholder and i18n

8.1 Loading state

The skin wraps the grid of columns in a `LoadingPane`. While data is being fetched, set the status to `LOADING` and the pane shows the configured progress indicator instead of the columns.

```
view.setLoadingStatus(LoadingPane.Status.LOADING);
view.setLoadingStatusSize(LoadingPane.Size.LARGE);
// ... load data on a background thread, then, on the FX thread:
view.setLoadingStatus(LoadingPane.Status.OK);
```

8.2 Placeholder

When the column list is empty, the skin shows the `placeholder` node instead of the columns and resizes it to fill the control. The default is a label reading "No columns defined." surrounded by a dashed border. Setting the property to `null` removes it.

```
view.setPlaceholder(new Label("Please select a project first."));
view.setPlaceholder(null); // show nothing at all
```

Independently of this, every column's list view has its own placeholder for the case that the column contains no items. The skin creates one automatically if the list view does not have one and wires it up so that items can be dropped onto an empty column.

8.3 Localization

The default texts are resolved through `ResourceBundleManager` using the bundle `multi-column-list-view`. Translations are provided for Arabic, Chinese, Danish, Dutch, English, Finnish, French, German, Italian, Japanese, Norwegian, Portuguese, Spanish, Swedish and Ukrainian.

Key	English default
`column.header.default`	Column Header
`placeholder.from`	From
`placeholder.to`	To
`placeholder.no-columns`	No columns defined.

8.4 Accessibility

The control sets `AccessibleRole.LIST_VIEW` in its constructor via `AccessibilityUtil`. Custom cells should provide meaningful text so that screen readers can announce the content of the columns.

9. Complete example

The following example is a condensed version of the demo application `com.dlsc.gemsfx.demo.MultiColumnListViewApp`.

```
public class KanbanApp extends Application {

    @Override
    public void start(Stage stage) {
        MultiColumnListView<Issue> view = new MultiColumnListView<>();
        view.setCellFactory(IssueListCell::new);
        view.getColumns().setAll(createColumns());

        // no item may leave the "done" column
        view.setDragPossibleCallback(issue -> !"done".equals(issue.getStatus()));

        // the "in progress" column is limited to three items
        view.setDropPossibleCallback(param ->
                !"in-progress".equals(param.getColumn().getUserObject())
                        || param.getColumn().getItems().size() < 3);

        view.addEventHandler(MultiColumnListViewEvent.ITEM_MOVED,
                evt -> save(evt.getDraggedItem(), evt.getColumn()));

        VBox.setVgrow(view, Priority.ALWAYS);

        VBox root = new VBox(10, view);
        root.setPadding(new Insets(20));

        stage.setScene(new Scene(root, 1000, 800));
        stage.setTitle("Kanban Board");
        stage.show();
    }

    private List<ListViewColumn<Issue>> createColumns() {
        ListViewColumn<Issue> todo = new ListViewColumn<>();
        todo.setHeader(new Label("To Do"));
        todo.setUserObject("todo");
        todo.getItems().setAll(new Issue("Fix login"), new Issue("Write tests"));

        ListViewColumn<Issue> progress = new ListViewColumn<>();
        progress.setHeader(new Label("In Progress"));
        progress.setUserObject("in-progress");

        ListViewColumn<Issue> done = new ListViewColumn<>();
        done.setHeader(new Label("Done"));
        done.setUserObject("done");

        return List.of(todo, progress, done);
    }
}
```

The demo application can be started from the repository root with:

```
mvn javafx:run -f gemsfx-demo/pom.xml \
    -Dmain.class=com.dlsc.gemsfx.demo.MultiColumnListViewApp
```

10. Best practices and pitfalls

- **Modify the items, not the wrappers.** Write to `ListViewColumn.getItems()`; the wrapper list is maintained by the control.
- **Wrapper identity matters.** The synchronization compares items by identity, not by `equals()`. Avoid replacing model objects with equal copies if you want cells to be reused.

- **Update the model in ITEM_MOVED.** This is the only event that signals a completed move; the other events also fire for rejected gestures.
- **Do not override updateItem().** It is `final`; override `updateUserObject()` instead.
- **Handle marker cells.** Custom cells receive `null` for both markers — check `isFromPlaceholder()` and `isToPlaceholder()` before rendering.
- **Changing showHeaders, factories or the column list rebuilds the view.** Avoid doing so in tight loops or animations.
- **A read-only board** is achieved with `setDisableDragAndDrop(true)` rather than by removing the cell factory.

11. Reference

Artifact	Location
`Control`	com.dlsc.gemsfx.MultiColumnListView
`Skin`	com.dlsc.gemsfx.skins.MultiColumnListViewSkin
`Stylesheet`	com/dlsc/gemsfx/multi-column-list-view.css
`Resource bundle`	multi-column-list-view[_xx].properties
`Demo application`	com.dlsc.gemsfx.demo.MultiColumnListViewApp
`Demo stylesheet`	com/dlsc/gemsfx/demo/multi-column-app.css