Modern Docking

Release 1.4

Andrew Auclair

Jun 10, 2026

CONTENTS:

Welcome! This documentation explains the user facing features of Modern Docking, its API, and some of the internals.

Modern Docking provides a rich set of features to enhance any Java Swing User Interface with a desire to provider user customizable layouts of UI components.

Java 11 and newer are supported.

CHAPTER

ONE

USER FACING FEATURES

Users of your application will interact with several UI components provided by Modern Docking. The framework is built around using Java Swing's `JTabbedPane` and `JSplitPane` to provide the docking layouts. The positioning of the applications dockable components within these ``JTabbedPane``s and ``JSplitPane``s is done by dragging the dockables. When the dockable is being dragged it will float above the window of the application and show Docking Handles and Docking Regions. Docking Handles allow the user to place the dockable precisely where they want it and Docking Regions provide extra visual feedback of this action. Docking Regions are also a quick way to "snap" dockables into place without needing to precisely drag to a Docking Handle.

1.1 Using Docking Handles to Dock

When dragging a dockable hovering over another dockable will display docking handles in the center of the dockable. These handles provide easy access to dock to the North, South, East and West regions.

The dockable can also be docked to any of these regions by hovering over the region itself and dropping the dockable.

The root also has North, South, East and West handles to dock the dockable directly to the root of the panel.

1.2 Adjusting Split

Splits can be adjusted and perform continuous layout. Double-clicking the split will return it to 50-50 split for the 2 sides.

1.3 Creating Tab Group

Panels can be grouped into tabbed panes by dragging a dockable to the center region of another dockable.

1.4 Closing Panels

Panels can be closed using the X button on their headers. This option can be disabled in the source code. Test

1.5 Floating a Panel

Panels can be floated as their own window by dragging them by their header and dropping them outside the frame. This creates a new JFrame with the dockable in it. More dockables can then be docked to this dockable.

1.6 Auto Hide a Panel

Panels can be set to Auto Hide with View Mode > Auto Hide from the settings button on the panel header. This option will display the panel on a side toolbar as a button which can be pressed to display the panel. Clicking the button again, or clicking off the panel, will return it to the toolbar. To return the panel to normal, select the View Mode > Auto Hide option again to deselect it.

1.7 Option Panel in New Window

Panels can be opened in their own window using View Mode > Window.

1.8 Additional Settings Options

Display custom settings options on the settings menu.

CHAPTER

TWO

PACKAGING

Modern Docking is provided on Maven Central as a collection of packages.

Most applications will use `modern-docking-api` and `modern-docking-single-app`. There is an additional UI package (*Modern Docking UI*) that adds special support for the FlatLaf look and feel library.

In addition to the above packages, there is a package for applications that launch multiple instances in a single JVM, which is how IntelliJ IDEA works. This package is available as `modern-docking-multi-app` and replaces `modern-docking-single-app` for the multi-app in JVM use case.

CHAPTER

THREE

MODERN DOCKING API

This section of topics explains the Modern Docking API that can be used by an application. This API is used to create and maintain the dockables that users interact with.

3.1 Hello World

This page will demonstrate a hello world application with Modern Docking.

At the end of this page you'll know how to:

- create and register dockables
- create and register root panels
- dock programmatically (covered more in depth *Docking Programmatically*

First we will create `HelloWorld` and `MainFrame` classes. `HelloWorld` will contain our `main` method and `MainFrame` will extend `JFrame` to provide the main frame for our application.

```
public class HelloWorld {
    public static class MainFrame extends JFrame {
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new MainFrame().setVisible(true));
    }
}
```

That will give us a very basic application to start working from.

Now we need to create a constructor that does the following:

- sets a size
- initializes Modern Docking
- adds a `RootDockingPanel`
- creates, registers and docks some dockables

First we set a size so that our `JFrame` shows up on screen in a reasonable way

```
setSize(400, 300);
```

Next step is to initialize the docking framework.

```
Docking.initialize(this);
```

This will initialize the docking framework with our `MainFrame` class as the Main Frame of the application. This also initializes internal components in the framework that are needed later.

Next, we create and add a `RootDockingPanel`. This can be further customized if you want to use a different layout or have other custom components surrounding the dockable area. For now, we're going to simple use a `BorderLayout` (the default `JFrame` layout manager) and let the root consume all the space.

```
RootDockingPanel root = new RootDockingPanel(this);
add(root, BorderLayout.CENTER);
```

This creates a new root and adds it to the `JFrame`. `this` is passed to tell the `RootDockingPanel` what `Window` it belongs to. This is used to register the panel with the docking framework and to create any toolbars required for pinning.

Next, we will create a dockable. This will involve creating a new `JPanel` that extends the `Dockable` interface. Most of the methods in the `Dockable` interface have default implementations and are not required to be implemented.

The following methods are required to be implemented:

Method	Description
`getPersistentID`	A unique identifier for the dockable within the docking framework
`getTabText`	The text to display on the titlebar and on a `JTabbedPane` tab, if docked into a tab group

We will make a simple panel that takes text as the lone constructor argument and uses it as the persistentID and tab text.

```
static class DockingPanel extends JPanel implements Dockable {
    private final String text;

    public DockingPanel(String text) {
        this.text = text;

        Docking.registerDockable(this);
    }

    @Override
    public String getPersistentID() {
        return text;
    }

    @Override
    public String getTabText() {
        return text;
    }
}
```

Now that we've created a panel that implements `Dockable` we can start creating dockables in our constructor.

```
DockingPanel helloWorld = new DockingPanel("Hello World");
```

This will register the dockable with the docking framework. The persistentID will be used to uniquely identify the dockable throughout the framework. In more complicated applications we would call `Docking.registerDockable`

from within the constructor of our panel, as shown below.

We can now dock the dockable. First, we can do this with the panel reference directly.

```
Docking.dock(helloWorld, this);
```

or, we can use the persistentID value:

```
Docking.dock("Hello World", this);
```

In both cases `this` again refers to our `MainFrame`, requesting that the framework dock our panel to this frame.

Now we have a complete sample that will create a `JFrame` with a dockable with the display text of "Hello World". Full sample is shown below:

```
public class HelloWorld {
    static class DockingPanel extends JPanel implements Dockable {
        private final String text;

        public DockingPanel(String text) {
            this.text = text;

            Docking.registerDockable(this);
        }

        @Override
        public String getPersistentID() {
            return text;
        }

        @Override
        public String getTabText() {
            return text;
        }
    }

    public static class MainFrame extends JFrame {
        public MainFrame() {
            setSize(400, 300);

            Docking.initialize(this);

            RootDockingPanel root = new RootDockingPanel(this);
            add(root, BorderLayout.CENTER);

            DockingPanel helloWorld = new DockingPanel("Hello World");

            Docking.dock(helloWorld, this);
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new MainFrame().setVisible(true));
    }
}
```

3.2 Dockable Properties

The `@DockingProperty` attribute is used to dynamically add properties to dockable components and was inspired by the excellent Java command line parsing library Picocli.

For example, say we have a simple `JCheckBox` on our Dockable component and wish to persist it together with the Docking layout. We could create a **`boolean`** member variable in our Dockable component class and tie it to a property.

```
public class Example implements Dockable {
    @DockingProperty(name = "enabled", defaultValue = "false")
    private boolean enabled;

    @Override
    public void updateProperties() {
        // enabled value is now usable
    }
}
```

These dockable properties can be done for the following Java types:

- byte
- short
- int
- long
- float
- double
- char
- boolean
- String

3.3 Dockable Interface

The `Dockable` interface is required to be implemented by any component that will be dockable.

Typically, this interface is implemented by a class that extends from `JPanel`.

The `Dockable` interface has a minimum subset of methods that must be implemented by the application and a set of methods that are optional. These optional methods provide extra customization points for the application per dockable.

> **Warning**
>
> The return values for the `Dockable` interface methods should be constant after the creation of the dockable component. Certain methods can be dynamic and the framework provides methods to update their values within the framework. Any method that doesn't specify such a method can lead to unexpected behavior if changed after the dockable component is created.
>
> Generally, if you wish to modify the return values after creating the dockable component, do so while the dockable is not docked. This can be checked with `Docking.isDocked`.

3.3.1 Mandatory Methods

These two methods do not provide default implementations in the interface and must be implemented by the application.

Persistent ID

```
String persistentID()
```

Provides a unique ID for the dockable to use within Modern Docking

Tab Text

```
String getTabText()
```

Provides the text that will be displayed on a tab when the dockable is in a `JTabbedPane`.

Important

If the text displayed on the tab ever changes, the application must call `Docking.updateTabText` with the dockables persistentID to force the framework to update the text. This should be done anytime the text is changed, just in case the dockable is displaying in a JTabbedPane. Undocking and docking the dockable again will also update the tab text.

3.3.2 Optional Methods

The following methods are provided to the application with a default. This means the application only needs to implement the methods if it wishes to change the default.

Type

```
int getType()
```

Default
0

Provides an int to Modern Docking. This represents a unique type category for the dockable. Modern Docking will use this value when docking to determine which dockables to dock to.

Title Text

```
String getTitleText()
```

Default
value of `getTabText`

Provides the text that Modern Docking should display on the header. This string can be different than `getTabText`

Tab Tooltip

```
String getTabTooltip()
```

Default
`null`

Used by the framework to get the text to display as a tooltip on `JTabbedPane` tabs.

Icon

```
Icon getIcon()
```

Default
`null`

Used by the framework to get the icon for the dockable to use in a `JTabbedPane` tab.

Is Floating Allowed

```
boolean isFloatingAllowed()
```

Default
`true`

Tells Modern Docking if the dockable is allowed to be opened in its own window

Is Limited to Window

```
boolean isLimitedToWindow()
```

Default
`false`

Allows the application to limit the dockable to the window it was initially docked in.

Style

```
DockableStyle getStyle()
```

Default
`DockableStyle.BOTH`

The docking style of the dockable which can be `DockableStyle.VERTICAL`, `DockableStyle.HORIZONTAL`, `DockableStyle.BOTH` or `DockableStyle.CENTER_ONLY`. Modern Docking will use this to determine which docking regions to allow when docking this dockable. Docking handles that do not match this style will be hidden.

Auto Hide Style

```
DockableStyle getAutoHideStyle()
```

Default
`DockableStyle.BOTH`

Determines which toolbars this dockable can be displayed on. Uses the same values as `getStyle`. `DockableStyle.VERTICAL` will allow the dockable on the east and west auto hide toolbars. `DockableStyle.HORIZONTAL` will allow the dockable on the south auto hide toolbar. `DockableStyle.CENTER_ONLY` is invalid for this method.

Closable

```
boolean isClosable()
```

Default
`true`

Indicates to the docking framework whether the Dockable component can be closed and undocked.

Request Close

```
boolean requestClose()
```

Default
`true`

Called by Modern Docking when the dockable is in the process of closing due to `undock`. This allows the application to stop the dockable from closing. For example, maybe the user has unsaved changes and the application wishes to confirm closing of the dockable.

Auto Hide Allowed

```
boolean isAutoHideAllowed()
```

Default
`false`

Determines if the dockable can be set to the auto hide toolbars.

Min Max Allowed

```
boolean isMinMaxAllowed()
```

Default
`false`

Determines if the dockable can be maximized so that it takes up all the space in the window.

Wrappable in Scrollpane

```
boolean isWrappableInScrollpane()
```

Default
`false`

Allows the application to specify whether the docking framework should automatically wrap the Dockable component in a `JScrollPane`.

Has More Options

```
boolean getHasMoreOptions()
```

Default
`false`

Flag that tells Modern Docking that this dockable has more menu items it wishes to add to the context menu. If this method returns true then Modern Docking will call `addMoreOptions`

Tab Preference

```
DockableTabPreference getTabPreference()
```

Default
`DockableTabPreference.NONE`

Gives the dockables preferred tab location when in a `JTabbedPane`

Add More Options

```
void addMoreOptions(JPopupMenu menu)
```

Adds this dockables menu items to the context menu

Create Header UI

```
DockingHeaderUI createHeaderUI(HeaderController headerController, HeaderModel␣
↪headerModel)
```

Default

```
DockingInternal.createDefaultHeaderUI(headerController, headerModel)
```

Creates the header UI for this dockable. The default implementation will create the default Modern Docking header.

Update Properties

```
void updateProperties()
```

Modern Docking will call this method after setting the values of any fields annotated with `DockingProperty`. If there are no fields with that annotation then this method is not called

3.4 Docking Programmatically

3.4.1 Dock

`dock` methods of the `Docking` class are used to dock dockables. There are several variations of this method that allow docking directly to a window or to specific regions of other dockables. There are variations that allow specifying the divider proportions to use for `JSplitPane` and each form of `dock` allows using the `persistentID` directly or an instance of `Dockable`

```
dock(String persistentID, Window window)
dock(Dockable dockable, Window window)
```

Allows docking the dockable to a given Window. This will only work if the root docking panel of the window is empty.

```
dock(String persistentID, Window window, DockingRegion region)
dock(Dockable dockable, Window window, DockingRegion region)
```

Docks the dockable to the specified root region of the window. The divider proportion is set to .25

```
dock(String persistentID, Window window, DockingRegion, double dividerProportion)
dock(Dockable dockable, Window window, DockingRegion, double dividerProportion)
```

Dock the dockable to the specific root region of the window with a user provided divider proportion.

```
dock(String sourcePersistentID, String targetPersistentID, DockingRegion region)
dock(String sourcePersistentID, Dockable target, DockingRegion region)
dock(Dockable source, String targetPersistentID, DockingRegion region)
dock(Dockable source, Dockable target, DockingRegion region)
```

Dock the dockable to a specific region of another dockable.

```
dock(String sourcePersistentID, String targetPersistentID, DockingRegion region, double␣
↪dividerProportion)
dock(Dockable source, Dockable target, DockingRegion region, double dividerProportion)
```

Dock the dockable to a specific region of another dockable with a user provided divider proportion.

3.4.2 Undock

```
undock(String persistentID)
undock(Dockable dockable)
```

Undocks the dockable. Nothing is done if the dockable is not docked

```
newWindow(Dockable dockable)
newWindow(String persistentID, Point location, Dimension size)
newWindow(Dockable dockable, Point location, Dimension size)
```

Opens the dockable in a new `FloatingFrame` instance. If *location* and *size* are provided, they are used to size the new frame.

3.4.3 Bring to Front

```
void bringToFront(Dockable dockable)
void bringToFront(String persistentID)

Brings the dockable to the front if it is not showing. If the dockable is in a tab group␣
↪it will be made the active tab. If the dockable is hidden due to the Auto Hide feature,
↪ then it will be shown. Finally, the frame containing the dockable will be brought to␣
↪the front with the ``JFrame::toFront`` function.
```

3.4.4 Display

This method is a combination of `dock` and `bringToFront`. If the dockable is not docked it will be docked and then brought to the front</p>

```
display(Dockable dockable)</code>
display(String persistentID)</code>
```

3.4.5 isDocked

Checks if a dockable is already docked

3.4.6 isMaximized

Checks if a dockable is currently maximized

3.5 Menu Items

Modern Docking provides several specialized menu classes explained below

3.5.1 LayoutsMenu

Displays a list of all layouts that the layout manager knows about. These are automatically added and removed. When clicked the menu item will load that layout.

3.5.2 ApplicationLayoutMenuItem

Menu item specific to one layout. Simply displays the name and when clicked loads the layout with `Docking.restore()`

3.5.3 DockableMenuItem

Displays a single dockable. Shows a checkmark if the dockable is docked. If the dockable is not docked, docks it, if it is docked, it displays it.

3.6 Customization Points

3.6.1 Tab Placement

By default, Modern Docking places all tabs in a `JTabbedPane` at the bottom. This can be changed using `Settings.alwaysDisplayTabsMode` which will switch the tab placement to the top.

3.6.2 Tab Layout Policy

The tab layout policy of `JTabbedPane` can be customized with `Settings.setTabLayoutPolicy`. The default tab layout policy is `JTabbedPane.SCROLL_TAB_LAYOUT` and it can be set to either `JTabbedPane.SCROLL_TAB_LAYOUT` or `JTabbedPane.WRAP_TAB_LAYOUT`.

3.6.3 Active Highlighter

By default, Modern Docking will highlight the dockable that the house is currently over. This can be disabled before initializing the docking framework by using `Settings.setActiveHighlighterEnabled`.

3.6.4 Custom Dockable Header

Create your own implementation of the header UI and return it in Dockable.

3.7 Exceptions

3.7.1 DockableNotFoundException

Thrown when a dockable is not found when restoring a DockingLayout

> **Note**
>
> Thrown by `DockingState.restoreState`

3.7.2 DockableRegistrationFailureException

Thrown when a dockable with the `persistentID` has already been registered

> **Note**
>
> Thrown by `Docking.registerDockable`

3.7.3 DockingLayoutException

This exception is thrown when there is an issue saving or loading a layout file. The exception provides the file that failed and the failure type

> **Note**
>
> Thrown by `AppState.restore`, `LayoutPersistence.saveLayoutToFile` and `LayoutPersistence.loadApplicationLayoutFromFile`

3.7.4 NotDockedException

This exception is thrown when Modern Docking attempts to use a dockable that should be docked but isn't. Thrown when the target dockable when docking is not docked or when attempting to bring a dockable to front that isn't already docked

> **Note**
>
> Thrown by `Docking.dock` and `Docking.bringToFront`

3.7.5 RootDockingPanelNotFoundException

Thrown when the root for a window is not found

> **Note**
>
> Thrown by `Docking.configurePinning`, `Docking.dock` and `DockingComponentUtils.rootForWindow`

3.7.6 RootDockingPanelRegistrationFailureException

Thrown when Modern Docking fails to register a `RootDockingPanel` because one is already registered for the window

> **Note**
>
> Thrown by `Docking.registerDockingPanel`

3.8 Events

3.8.1 DockingEvent

This event is fired when dockables are docked, undocked, shown, hidden, pinned or unpinned. Shown and hidden are used when a dockable is in a `JTabbedPane` and the active tab changes. Pinned and Unpinned

are used when the dockable is added to a toolbar or removed from a toolbar. Shown and hidden will also be fired when a pinned dockable is shown and hidden.

Temporary events are fired when a user starts dragging a dockable. This allows any listeners to perform actions only based on permanent events (i.e. events that do not have the temporary flag set).JTabbedPane

3.8.2 DockingLayoutEvent

Fired when layouts are added to or removed from `DockingLayouts` and when layouts are restored or persisted to a file

3.9 Listeners

3.9.1 DockingLayoutListener

Listen for when layouts are added to or removed from `DockingLayouts` and when layouts are restored or saved to a file

3.9.2 DockingListener

Listen for any changes to active dockables. This includes docking, undocking, shown, hidden, auto hide

3.9.3 MaximizeListener

Listen for maximize events triggered by `Docking.maximize`

3.9.4 NewFloatingFrameListener

Listen for the creation of new floating frames. This allows the application to set icons, titles, menu bars, etc. on the new frame.

3.10 Working With Layouts

Layouts can be created in memory from scratch with a builder, pulled from the current window or application layout and saved to files for long term storage.

In memory layouts are created using the `WindowLayoutBuilder` class and are built using persistentIDs only, by calling similar `dock` functions to the ones provided in the `Docking` class. These persistentIDs do not need to be currently registered in the docking framework to be added to `WindowLayoutBuilder`. An error will be thrown if the persistentID already exists in the layout. When done building the layout, call `build()` to create a `WindowLayout` or `buildApplicationLayout()` to directly build an `ApplicationLayout`. If you're building a layout for multiple windows, you can manually create an `ApplicationLayout` and use `WindowLayoutBuilder` to create layouts and add them to `ApplicationLayout` with `addFrame`.

The `WindowLayout` and `ApplicationLayout` can then be saved to XML, as well as loaded, with the `WindowLayoutXML` and `ApplicationLayoutXML` classes, respectively.

The docking framework can store these layouts for you and provides a special `JMenuItem` that can restore named layouts on the application.

Layouts can be restored by using the `restoreApplicationLayout` and `restoreWindowLayout` methods of the `DockingState` class. This undocks all dockables from the window (or entire application for an ApplicationLayout) and docks the dockables specified by the layout.

Default layout management and restore is discussed in *Persistence*

3.11 Persistence

Modern Docking will persist the current layout of the application to a file specified through the API. When auto persistence is enabled, this file is saved after a number of different UI actions listed below. A delay mechanism is employed to avoid unnecessarily saving the file, such as when the user is dragging splitters.

The persistence feature defaults to off and can be enabled by calling the `setPersist`. The file that Modern Docking should use to persist the layout can be configured with `setPersistFile`. Finally, a default layout can be configured with `setDefaultApplicationLayout` for when persistence is disabled or Modern Docking fails to load the current auto persist file.

3.12 Look and Feel

3.12.1 Colors

Modern Docking handles all colors by using properties in the UIManager. For all the colors listed here, Modern Docking will first attempt to use the Modern Docking property name, then the theme color or a custom configured property from the user. If none of these are found, Modern Docking will default to a predefined color.

Docking Handle Background

This setting controls the color used for the background of Docking Handles.

UIManager property used for the background color on Docking Handles. This property can be modified by calling `DockingSettings.setHandleBackgroundProperty`

Modern Docking property: `ModernDocking.handleBackground`

Default UIManager property: `TableHeader.background`

Docking Handle Foreground

This setting controls the color used for the foreground of Docking Handles. The foreground color is used both for the borders and the mouse over color.

UIManager property used for the foreground color on Docking Handles. This property can be modified by calling `DockingSettings.setHandleForegroundProperty`

Modern Docking property: `ModernDocking.handleForeground`

Default UIManager property: `TableHeader.foreground`

Dockable Header Background

This setting controls the color used for the background of the default dockable header provided by Modern Docking.

This property can be modified by calling `DockingSettings.setHeaderBackgroundProperty`

Modern Docking property: `ModernDocking.headerBackground`

Default UIManager property: `TableHeader.background`

Dockable Header Foreground

This setting controls the color used for the foreground of the default dockable header provided by Modern Docking.

This property can be modified by calling `DockingSettings.setHeaderForegroundProperty`

Modern Docking property: `ModernDocking.headerForeground`

Default UIManager property: `TableHeader.foreground`

Docking Overlay Background

This setting controls the color used for the background of the docking overlay. An alpha value less than 100% is typically used for this color.

This property can be modified by calling `DockingSettings.setOverlayBackgroundProperty`

Modern Docking property: `ModernDocking.overlayBackground`

Active Dockable Highlighter Selected Border Color

This setting controls the color used for the border color when the mouse is over a dockable and the active dockable highlighter is enabled.

This property can be modified by calling `DockingSettings.setHighlighterSelectedBorderProperty`

Modern Docking property: `ModernDocking.highlighterSelectedBorder`

Default UIManager property: `Component.focusColor`

Active Dockable Highlighter Not Selected Border Color

This setting controls the color used to reset to the default border when a dockable is no longer under the mouse. Modern Docking will use this color when the active dockable highlighter is active. The color should be set to the theme's default border color.

This property can be modified by calling `DockingSettings.setHighlighterNotSelectedBorderProperty`

Modern Docking property: `ModernDocking.highlighterNotSelectedBorder`

Default UIManager property: `Component.borderColor`

CHAPTER

FOUR

MODERN DOCKING UI

Modern Docking UI is an extension to Modern Docking that changes png's to svg's for the Settings and Close icons using the FlatLaf library.

4.1 Initializing

Using our *Hello World* example, we can update it to initialize the Modern Docking UI like below, which is all that's required to use Modern Docking UI.

```
public static class MainFrame extends JFrame {
    public MainFrame() {
        setSize(400, 300);

        Docking.initialize(this);
        DockingUI.initialize();

        RootDockingPanel root = new RootDockingPanel(this);
        add(root, BorderLayout.CENTER);

        DockingPanel helloWorld = new DockingPanel("Hello World");

        Docking.dock(helloWorld, this);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new MainFrame().setVisible(true));
    }
}
```

CHAPTER

FIVE

INTERNAL WORKINGS

These topics cover internal functionality of the framework for anyone wanting to become more familiar with how Modern Docking works. This knowledge is not required for using Modern Docking.

5.1 ActiveDockableHighlighter

The `ActiveDockableHighlighter` is responsible for drawing a border around the dockable that the mouse is currently over. Using an AWT event listener on the AWT Toolkit lets us listen for all mouse events in the entire application.

5.2 AppStatePersister

Used to call `AppState.persist` whenever a `Window` instance resizes, moves or changes state.