Holon Platform Vaadin UI Module - Reference manual

Version 5.1.2

Table of Contents

1. Introduction . . . 1
  1.1. Sources and contributions . . . 1
2. Obtaining the artifacts . . . 1
  2.1. Using the Platform BOM . . . 2
3. What's new in version 5.1.x . . . 2
  3.1. Migrating from version 5.0.x . . . 3
    3.1.1. Deprecations . . . 3
    3.1.2. Item identifiers in `Property` based Item listings . . . 3
4. Vaadin UI integration . . . 4
5. Component builders . . . 4
6. Component builders common features . . . 5
  6.1. Component configurators . . . 7
  6.2. Messages localization support . . . 8
    6.2.1. Defer localization . . . 9
  6.3. Drag and Drop support . . . 9
7. UI Components . . . 10
  7.1. Input . . . 10
    7.1.1. Create an `Input` component . . . 11
    7.1.2. Obtain an Input component from a `HasValue` Component . . . 13
    7.1.3. Obtain a `HasValue` Component from Input builders . . . 13
    7.1.4. Input value conversion . . . 13
    7.1.5. Listening to `Input` value changes . . . 14
  7.2. ValidatableInput . . . 16
    7.2.1. Registering Validators and perform validation . . . 16
    7.2.2. Automatic validation at `Input` value change . . . 17
    7.2.3. Validation errors notification . . . 18
  7.3. Selectable Input . . . 19
    7.3.1. Rendering mode . . . 20
    7.3.2. Using *selectable* Inputs for Enumerations . . . 21
    7.3.3. Selection items data source . . . 21
    7.3.4. Item captions and icons . . . 22
    7.3.5. `SingleSelect` caption filter . . . 23
    7.3.6. Using the `Property` model with *selectable* Inputs . . . 23
  7.4. Input groups . . . 25
    7.4.1. Property Inputs rendering . . . 26
    7.4.2. Explicit `Input` and `HasValue` binding . . . 26
    7.4.3. Input value convertes . . . 26
    7.4.4. Manage the property values using a `PropertyBox` . . . 27

7.4.5. Read-only properties . . . 27
7.4.6. Hidden properties . . . 28
7.4.7. Default property values . . . 28
7.5. Input group validation . . . 28
7.5.1. `Property` validators . . . 30
7.5.2. Validation configuration . . . 30
7.5.3. Validation errors notification . . . 31
7.6. Handling value changes . . . 32
7.7. Dealing with *Virtual* properties . . . 33
7.8. Input Forms . . . 33
7.8.1. Base layout initialization . . . 34
7.8.2. `Input` components composition . . . 34
7.8.3. `Input` components width mode . . . 35
7.8.4. `Input` components configuration . . . 36
7.9. View components . . . 36
7.9.1. View component Groups and Forms . . . 37
7.9.2. `ViewComponent` captions . . . 38
8. Item listing . . . 38
8.1. PropertyListing . . . 39
8.1.1. Visible properties . . . 39
8.1.2. Items type and identifiers . . . 40
8.1.3. Items data source . . . 40
8.1.4. Item identifier configuration . . . 41
8.1.5. Item query configuration . . . 42
8.1.6. Using `VirtualProperty` for `Component` type columns . . . 42
8.2. BeanListing . . . 43
8.2.1. Bean items data source . . . 44
8.2.2. Dynamically generated columns . . . 45
8.3. Item listing API and configuration . . . 46
8.3.1. Columns configuration . . . 46
8.3.2. Columns rendering . . . 47
8.3.3. Item listing configuration . . . 47
8.3.4. Items selection . . . 48
8.3.5. Items query configuration . . . 50
8.3.6. Item set management . . . 50
8.3.7. Buffered mode . . . 51
8.3.8. Editing items . . . 52
9. Dialogs . . . 53
9.1. Question dialogs . . . 54
10. Property renderers and presenters . . . 55
10.1. Property localization . . . 56

10.2. Custom property renderers . . . 56
11. Vaadin session scope . . . 57
12. Device information . . . 57
13. Navigator . . . 58
13.1. View parameters injection . . . 58
13.2. View lifecycle management . . . 59
13.3. Providing the View contents . . . 62
13.4. ViewNavigator Configuration . . . 62
13.5. Excluding a View form navigation history . . . 64
13.6. Opening Views in a `Window` . . . 65
13.7. ViewNavigation API operations . . . 65
13.8. Sub views . . . 67
13.9. Context resources injection . . . 68
13.10. Obtain the current ViewNavigator . . . 69
13.11. Authentication support . . . 69
14. Spring integration . . . 70
14.1. Default View navigation strategy . . . 71
14.2. Spring view navigator configuration . . . 71
14.3. View context resources . . . 73
14.4. View authorization support . . . 73
14.5. Spring Security support . . . 75
15. Spring Boot integration . . . 75
15.1. Vaadin servlet configuration . . . 76
15.2. Spring Boot starters . . . 76
16. Loggers . . . 77
17. System requirements . . . 77
17.1. Java . . . 77
17.2. Vaadin . . . 77

Copyright © 2016-2018

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1. Introduction

The Holon **Vaadin** module represents the platform support for the Vaadin web applications framework, focusing on the user interface components and data binding features.

This module provides integration with the platform foundation architecture, such as;

- The Property model and the Datastore API.
- The Holon Platform authentication and authentication architecture.
- The Holon Platform internationalization support.

This module provides a *view navigation* system which allows to create web applications focusing on the application functions, which can be represented by *virtual pages*, relying on a robust navigation architecture, with parameters and view lifecycle hooks support.

Furthermore, a complete set of *fluent* builders is available, to build web application UI components quickly and easily.

A complete integration with the **Spring framework** and **Spring Boot** is also available.

1.1. Sources and contributions

The Holon Platform **Vaadin** module source code is available from the GitHub repository https://github.com/holon-platform/holon-vaadin.

See the repository `README` file for information about:

- The source code structure.
- How to build the module artifacts from sources.
- Where to find the code examples.
- How to contribute to the module development.

2. Obtaining the artifacts

The Holon Platform uses Maven for projects build and configuration. All the platform artifacts are published in the **Maven Central Repository**, so there is no need to explicitly declare additional repositories in your project `pom` file.

At the top of each *section* of this documentation you will find the Maven *coordinates* (group id, artifact id and version) to obtain the artifact(s) as a dependency for your project.

A **BOM (Bill Of Materials)** `pom` is provided to import the available dependencies for a specific version in your projects. The Maven coordinates for the core BOM are the following:

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin-bom</artifactId>
<version>5.1.2</version>
```

The BOM can be imported in a Maven project in the following way:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.vaadin</groupId>
      <artifactId>holon-vaadin-bom</artifactId>
      <version>5.1.2</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2.1. Using the Platform BOM

The Holon Platform provides an **overall Maven BOM (Bill of Materials)** to easily obtain all the available platform artifacts.

See Obtain the platform artifacts for details.

3. What's new in version 5.1.x

- Support for the the `PropertySet` **identifier properties** to make more easy and quick to setup data bound UI objects. See Items type and identifiers.
- Full support of Vaadin version 8.3, including for example the Navigator `@PushStateNavigation` strategy support. See [PushStateNavigation].
- Better support for bean based data bound UI objects, with new builders and a Java API to create and configure the UI components just like the `Property` based one. See BeanListing.
- Support for components *drag and drop* configuration at Java API level. See Drag and Drop support.
- New component builder APIs for Vaadin `TabSheet` and `Accordion` components. See Component builders.

3.1. Migrating from version 5.0.x

3.1.1. Deprecations

- **PropertyListing builder API**: dataSource(ItemDataProvider dataProvider, Property... identifierProperties). Since the `PropertySet` identifier properties support is used by default by the `PropertyListing` builder to obtain the item identifier properties, this method is no longer useful. To customize the item identifiers strategy, the more general `dataSource(ItemDataProvider dataProvider, ItemIdentifierProvider itemIdentifierProvider)` builder method can be used.

3.1.2. Item identifiers in `Property` based Item listings

The `Property` based Item listing components now supports the PropertySet identifier properties configuration and use them as the default strategy to identify a `PropertyBox` type item.

So, when the `PropertySet` used with an item listing component declares one ore more *identifier properties*, there is no longer the need to declare the item identifier properties, for example, when configuring a `Datastore` based item data source. The *identifier properties* of the `PropertySet` will be used by default to provide the item identifier values.

For example, given the following Property model definition:

```
private final static PathProperty<Long> ID = PathProperty.create("id", Long.class);
private final static PathProperty<String> DESCRIPTION = PathProperty.create(
"description", String.class);

private final static PropertySet<?> PROPERTIES = PropertySet.of(ID, DESCRIPTION);
```

A `PropertyListing` component, prior to version 5.1.0, had to be built this way, for example using a `Datastore` as item data source:

```
Datastore datastore = getDatastore();

PropertyListing listing = Components.listing.properties(PROPERTIES) ①
    .dataSource(datastore, DataTarget.named("test"), ID) ②
    .build();
```

① Provide the item listing `PropertySet`

② Configure a `Datastore` as item data source, providing the `ID` property as item identifier property

From version 5.1.x, the Property model definition can include the `PropertySet` *identifier properties* like this:

```
private final static PathProperty<Long> ID = PathProperty.create("id", Long.class);
private final static PathProperty<String> DESCRIPTION = PathProperty.create(
"description", String.class);

private final static PropertySet<?> PROPERTIES = PropertySet.builderOf(ID,
DESCRIPTION).identifier(ID).build(); ①
```

① Set the `ID` property as property set identifier property

And so the item identifier properties to use are no longer required during the `PropertyListing` component data source configuration, since the `PropertySet` *identifier properties* are used by default:

```
Datastore datastore = getDatastore();

PropertyListing listing = Components.listing.properties(PROPERTIES) ①
    .dataSource(datastore, DataTarget.named("test")) ②
    .build();
```

① The item listing `PropertySet` provides the property set identifier properties

② There is no longer the need to declare the `ID` property as item identifier property at data source configuration time, since the configured `PropertySet` *identifier properties* will be used by default

4. Vaadin UI integration

The `holon-vaadin` artifact is the main entry point to use the Holon platform Vaadin integration.

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin</artifactId>
<version>5.1.2</version>
```

5. Component builders

A complete set of *fluent* builders is available to create and configure the most common Vaadin **components** and the additional components provided by the Holon platform Vaadin module.

> See UI Components for a list of the additional components made available by the Holon platform Vaadin module.

All the builders can be obtained by using the Components API, which is organized in sub-APIs by components category.

The available **component categories** are the following:

Sub-interface	Provides
[NONE]	A set of *configure(*)* methods to setup existing standard component instances and a set of methods to obtain the *fluent* builders for standard Vaadin components, such as `Label`, `Button`, and standard layout components.
input	Builders for different data type `Input` components, including selection components. Furthermore, it provides builder to create input *forms*.
view	Builders for `ViewComponent` components and view *forms*.
listing	Builders for components which represent a list of *items*.

For each UI displayed text of a component (for example, the component caption or description), the builders provide the opportunity to set a *localizable* text, using a `Localizable` object or directly providing the *message code* and the *default message* of the localizable text. In order for the localization to work, a `LocalizationContext` must be available as a context resource. See the Internationalization documentation for further information.

6. Component builders common features

The Holon Platform Vaadin component builders make available a set of common features, partly related to standard Vaadin features configuration and partly dedicated to the integration with some core Holon Platform APIs.

For each UI component which extends a standard Vaadin `Component` hierarchy, the builder APIs provide methods to configure all the component attributes and register the supported *listeners*. In addition to the common component configuration attributes, each component builder makes available methods to configure the component specific attributes and supported behaviors.

Label builder example

```
Label lbl = Components.label() ①
    .fullWidth() ②
    .height(50, Unit.PIXELS) ③
    .styleName("my-style").styleName(ValoTheme.LABEL_BOLD) ④
    .icon(VaadinIcons.CHECK) ⑤
    .caption("The caption") ⑥
    .captionAsHtml() ⑦
    .description("The description") ⑧
    .withData(new MyData()) ⑨
    .hidden() ⑩
    .disabled() ⑪
    .responsive() ⑫
    .withAttachListener(event -> { ⑬
      // ...
    }).withDetachListener(event -> { ⑭
      // ...
    }).withContextClickListener(event -> { ⑮
      event.isDoubleClick();
      // ...
    }).errorHandler(event -> { ⑯

    })
    // Label specific configuration
    .content("Label content") ⑰
    .html() ⑱
    .build();
```

① Obtain a `com.vaadin.ui.Label` component builder

② Set the component width to `100%`

③ Set the component height to `50px`

④ Add two css style class names

⑤ Set the component icon

⑥ Set the *caption*

⑦ Set the caption to be rendered as HTML

⑧ Set the *description* (tooltip)

⑨ Set a generic *data* value bound to the component

⑩ Set the component as hidden (not visible)

⑪ Set the component as disabled

⑫ Set the component as responsive

⑬ Register a `AttachListener`

⑭ Register a `DetachListener`

⑮ Register a `ContextClickListener`

⑯ Set the component `ErrorHandler`

⑰ Set the Label content

⑱ Set the Label content mode as HTML

For *component container* component types (for example, Layout components), the builder APIs provides a set of methods to add components to the container to build, also providing layout configuration attributes such as alignment and expand ratio, when supported.

VerticalLayout builder example

```
VerticalLayout verticalLayout = Components.vl() ①
    .margin() ②
    .spacing() ③
    .add(COMPONENT_1).add(COMPONENT_2, COMPONENT_3) ④
    .addAndAlign(COMPONENT_4, Alignment.TOP_CENTER) ⑤
    .addAndExpand(COMPONENT_5, 0.5f) ⑥
    .addAndExpandFull(COMPONENT_6) ⑦
    .addAlignAndExpand(COMPONENT_7, Alignment.MIDDLE_CENTER, 1f) ⑧
    .deferLocalization().build();
```

① Obtain a `com.vaadin.ui.VerticalLayout` component builder

② Enable layout margins

③ Enable layout spacing

④ Add components to the layout in given order

⑤ Add a component and set the component alignment

⑥ Add a component and set the component expand ratio

⑦ Add a component and set the component expand ratio to 1 (full)

⑧ Add a component and set the component alignment and expand ratio

6.1. Component configurators

When a component instance is already available, the Components API makes available a set of *component configurator* APIs, which can be used to configure a component instance, using the same API as the component *builders*.

The `configure(AbstractComponent component)` method can be used to obtain a component configurator for a generic `com.vaadin.ui.AbstractComponent`, to setup common component configuration attributes.

Other component confgurator APIs are provided to configure specific components, such a `Label` and `Button`, and the standard Vaadin layout components.

```
Label label = new Label();

Components.configure(label).fullWidth().content("my content").contentMode(ContentMode
.PREFORMATTED); ①

VerticalLayout verticalLayout = new VerticalLayout();

Components.configure(verticalLayout).spacing().addAndAlign(COMPONENT_1, Alignment
.TOP_CENTER) ②
    .addAndExpandFull(COMPONENT_2);
```

① Label component configurator

② VerticalLayout component configurator

6.2. Messages localization support

The component builder methods that can be used to configure a *text to display* for a component supports the **message localization** using the standard Holon Platform internationalization APIs.

The message localization attributes can be provided either using:

- A `Localizable` object instance;
- Or directly providing the message localization attributes, such as the **message code**, the **default message** and the optional message localization *arguments*.

A set of *localizable* message builder methods are available, for example, for standard Vaadin component attributes, such as the component *caption* and *description*.

```
Button btn = Components.button().caption("DefaultCaption", "button.caption.code") ①
    .build();

Label lbl = Components.label()
    .content(Localizable.builder().message("DefaultMessage").messageCode(
"my.message.code").build()) ②
    .build();
```

① Set the Button *caption* using a default caption message (`DefaultCaption`) and a message localization code (`button.caption.code`)

② Set a *localizable* Label content using a `Localizable`

> In order for the message localization to work, a valid `LocalizationContext` instance must be available as a **context resource** using the default `LocalizationContext` resource key (i.e. the `LocalizationContext` class name). See the LocalizationContext documentation for details about the `LocalizationContext` API and the Context resources documentation for information about the Holon Platform *Context* architecture.

This module makes available a Vaadin session scope to use the Vaadin session as *context resources scope*.

6.2.1. Defer localization

By default, the component messages localization is performed immediately during the component building, when a localizable message is provided at component configuration time.

Since the Holon Platform localization APIs require that a `LocalizationContext` resource is available to perform messages localization, when this is not possible the messages localization will not work at component configuration time. The `LocalizationContext` resource must be also *localized*, i.e. the current `Locale` is configured and available, in order to ensure messages localization support.

When the `LocalizationContext` resource configuration or localization availability is expected *after* the component build time, the component builder APIs provide a method to *defer* the component messages localization: `deferLocalization()`.

This method instructs the builder to resolve any message localization (for example the component caption and description) only when the component is attached to a parent layout, and this means only when the parent layout too is attached to the application UI.

```
Label lbl = Components.label() //
    .caption("Default caption", "caption.message.code") //
    .deferLocalization() ①
    .build();
```

① Set to defer the component messages localization at UI display time

6.3. Drag and Drop support

The Vaadin module component builders supports component **drag and drop** configuration, allowing to declare a component as *draggable* and configure the drag mode and attributes and to declare a component as a *drop target* providing the target configuration and logic to react to a drop event.

The standard Vaadin `DragSourceExtension` and `DropTargetExtension` extensions are used to configure components drag and drop capabilities.

The `dragSource` builder method can be used to declare a component as *draggable*. The provided `Consumer` can be used to configure the `DragSourceExtension` for the component.

The `dropTarget` builder method can be used to declare a component as a *drop target* and the provided `BiConsumer` can be used to configure the `DropTargetExtension` for the component and to obtain the actual component instance when a drop event occurs.

```
Label lbl = Components.label().content("Draggable").dragSource(dragSource -> { ①
  dragSource.setEffectAllowed(EffectAllowed.MOVE); ②
  dragSource.setDataTransferText("hello receiver");
  dragSource.addDragStartListener(e -> Notification.show("Drag event started"));
}).build();

VerticalLayout droppableLayout = Components.vl().fullWidth().caption("Drop things
inside me").dropTarget((dropTarget, component) -> { ③
  dropTarget.setDropEffect(DropEffect.MOVE); ④
  dropTarget.addDropListener(dropEvent -> { ⑤
    dropEvent.getDragSourceComponent().ifPresent(dragged -> component.addComponent
(dragged));
    Notification.show("DropEvent with data transfer text: " + dropEvent
.getDataTransferText());
  });
}).build();
```

① Configure the Label as *draggable*

② The `DragSourceExtension` can be configured using the provided `Consumer` function

③ Configure the VerticalLayout as a *drop target*

④ The `DropTargetExtension` can be configured using the provided `BiConsumer` function

⑤ The second argument of the `BiConsumer` function is the actual VerticalLayout instance, used in this case to add the dropped component to the layout when a drop event occurs

7. UI Components

The Vaadin module provides a set of additional UI components which can be used besides the standard Vaadin components.

7.1. Input

The Input interface represents a component that provides a user-editable value, which can be read and setted using the methods provided by the The ValueHolder API.

The `Input` API is parametrized on the value type it manages and the value management is provided by the ValueHolder API, which provides methods to:

- Set the Input value.
- Read the input value.
- Provide and *empty* value representation.
- Check whether a value is available.
- Listen to value changes registering one or more ValueChangeListener.

Furthermore, the `Input` API supports two component **state** configuration options:

- **Read-only mode:** When an `Input` is in read-only mode, the user can't change the Input value.
- **Required mode:** When an `Input` is in required mode, a *required indicator* symbol is visible.

The `Input` API is a general and abstract representation of an UI object which holds a value, the actual Vaadin `Component` representation can be obtained though the `getComponent()` method and used with the standard Vaadin APIs.

```
Input<String> stringInput = Components.input.string().build(); ①

Component component = stringInput.getComponent(); ②
VerticalLayout vl = Components.vl().add(component).build(); ③
```

① Create a String type input

② Get the actual Input `Component`

③ Add the Input `Component` to a vertical layout

7.1.1. Create an `Input` component

An `Input` component can be created using the Components API, through the **input** sub-interface methods.

A *typed* `Input` builder method is available for each available `Input` value type. Besides the base `Input` types, as set of methods are available to create **single and multi select** Input components. See Selectable Input for details.

```
Input<String> stringInput = Components.input.string().build(); ①
stringInput = Components.input.string(true).build(); ②
Input<Integer> integerInput = Components.input.number(Integer.class).build(); ③
Input<Boolean> booleanInput = Components.input.boolean_().build(); ④
Input<Date> dateInput = Components.input.date(Resolution.DAY).build(); ⑤
Input<LocalDate> localDateInput = Components.input.localDate().build(); ⑥
Input<LocalDateTime> localDateTimeInput = Components.input.localDateTime().build(); ⑦

SingleSelect<MyEnum> enumSingleSelect = Components.input.enumSingle(MyEnum.class)
.build(); ⑧
MultiSelect<MyEnum> enumMultiSelect = Components.input.enumMulti(MyEnum.class).build(
); ⑨
```

① `String` type Input

② `String` type Input represented as a *text area*

③ Numeric (`Integer` in this case) type Input

④ `Boolean` type Input

⑤ `Date` type Input with *resolution* configuration

⑥ `LocalDate` type Input

⑦ `LocalDateTime` type Input

⑧ `Enum` type Input represented with a *single select*

⑨ `Enum` type Input represented with a *multi select*

Input component configuration:

The `Input` builder API provides methods both for the `Input` attributes al listeners configuration and for the associated Input `Component` configuration, including API methods for specific `Input` and `Component` types.

```
Input<String> stringInput = Components.input.string() //
    // Component configuration
    .fullWidth().styleName("my-style").caption("The caption") ①
    // Input configuration
    .readOnly() ②
    .tabIndex(10) ③
    .withValue("Initial value") ④
    .withValueChangeListener(event -> { ⑤
      event.getValue();
      event.getOldValue();
      // ...
    }).withFocusListener(event -> { ⑥
      // focused
    })
    // Specific String Input configuration
    .inputPrompt("The prompt") ⑦
    .maxLength(100) ⑧
    .blankValuesAsNull(true) ⑨
    .textChangeEventMode(ValueChangeMode.BLUR) ⑩
    .build();
```

① Input `Component` configur

② Set the Input as read-only

③ Set the *tabulator index*

④ Set the Input initial value

⑤ Register a `ValueChangeListener`

⑥ Register a `FocusListener`

⑦ Set the Input *prompt*

⑧ Set the String Input max length (the maximum number of cahracters)

⑨ Set to treat *blank* String values (with 0 length or whitespaces only) as `null` values.

⑩ Set the mode how the Input triggers vaue change events

See the next sections for details about the `Input` configuration attributes and features.

7.1.2. Obtain an Input component from a HasValue Component

An Input component can be obtained from a standard Vaadin HasValue Component using the from static method of the Input API.

```
TextField field = new TextField();

Input<String> stringInput = Input.from(field); ①
```

① Create a String type Input using a Vaadin TextField as concrete implementation

A com.vaadin.data.Converter can also be provided to obtain an Input component with a different value type from the original field type, providing the presentation to model type value conversion logic and vice-versa.

```
TextField field = new TextField();

Input<Integer> integerInput = Input.from(field, new StringToIntegerConverter(
"conversion error")); ①
```

① Create a Integer type Input using a Vaadin TextField as concrete implementation and providing a Converter to convert String type presentation values into Integer type model values and vice-versa

7.1.3. Obtain a HasValue Component from Input builders

The Input builder APIs can provide the Input instance also a standard Vaadin HasValue Component, which is represented by the convenience Field interface.

To obtain a Field type Input component, the Input builder API asField() method can be used.

```
Field<LocalDate> field = Components.input.localDate() ①
    // configuration omitted
    .asField(); ②

HasValue<LocalDate> hasValue = field; ③
Component component = field;
```

① Obtain a LocalDate type Input builder

② Get the Input component as a LocalDate type Field

③ The obtained Field is a HasValue<LocalDate> and a Component

7.1.4. Input value conversion

The Input interface makes available a set of methods to create an Input from another Input or from a standard Vaadin HasValue Component with a different value type, providing a suitable *converter* to perform value conversions from the presentation value type to the model value type and vice-

versa.

The stardard Vaadin `com.vaadin.data.Conveter` API is supported to provide the value conversion logic.

```
Input<String> stringInput = Components.input.string().build();

Input<Integer> integerInput = Input.from(stringInput, new StringToIntegerConverter(
"Conversion error")); ①

Input<Boolean> booleanInput = Input.from(integerInput, ②
    Converter.from(value -> Result.ok((value == null) ? Boolean.FALSE : (value
.intValue() > 0)),
        value -> (value == null) ? null : (value ? 1 : 0)));

Input<Long> longInput = Input.from(new TextField(), new StringToLongConverter(
"Conversion error")); ③
```

① Convert a `String` presentation type `Input` into a `Integer` model type `Input` using the standard Vaadin `StringToIntegerConverter`

② Obtain a Boolean type `Input` from the `Integer` type one providing the value conversion logic

③ Obtain a `Long` type `Input` from a `String` type `TextField` using the `StringToLongConverter` converter

Besides the standard Vaadin `com.vaadin.data.Conveter`, the Holon `PropertyValueConverter` can be used, providing the `Property` to which the converter is bound.

See the Property value conversion documentation for information about the `PropertyValueConverter` API.

```
Input<Integer> integerInput = Components.input.number(Integer.class).build();

final Property<Boolean> BOOL_PROPERTY = PathProperty.create("bool", Boolean.class); ①

Input<Boolean> booleanInput = Input.from(integerInput, BOOL_PROPERTY,
    PropertyValueConverter.numericBoolean(Integer.class)); ②
```

① The `Boolean` type `Property` to use for the `PropertyValueConverter`

② Create a `Boolean` type `Input` from an `Integer` type `Input`, using a the default *numericBoolean* `PropertyValueConverter`

7.1.5. Listening to `Input` value changes

The `Input` API supports ValueChangeListener registration to listen to `Input` value changes.

The value change event provide the current (changed) `Input` value, the previous (old) `Input` value, the value change source (i.e. the reference to the `Input` component which triggered the value change listeners) and whether the value change was originated by a user action or from server side

code.

```
Input<String> stringInput = Components.input.string() //
    .withValueChangeListener(event -> { ①
      String currentValue = event.getValue(); ②
      String previousValue = event.getOldValue(); ③
      boolean byUser = event.isUserOriginated(); ④
      ValueHolder<String> source = event.getSource(); ⑤
      // ...
    }).build();
```

① Register a `ValueChangeListener` for the `String` type `Input`

② Get the new value that triggered the value change event

③ Get the value of the `Input` before the value change event occurred

④ Get whether the event was triggered by user interaction, on the client side, or programmatically, on the server side

⑤ Get the source of the value change event, i.e. the `Input` component itself

Value change mode

The mode and the frequency with which the value change events are triggered can be customized for the `Input` components which supports it.

The MaySupportValueChangeMode API, extended by the `Input` API, allows to check if the value change mode configuration is supported by a specific `Input` component through the `isValueChangeModeSupported()` method.

If so, the value change mode can be configured specifying:

- The `com.vaadin.shared.ui.ValueChangeMode` enumeration value which represents when and how often `Input` value changes are transmitted from the client to the server.
- Set the value change *timeout*, i.e. how often value change events are triggered, when the `ValueChangeMode` is either `LAZY` or `TIMEOUT`.

```
Input<String> stringInput = Components.input.string().withValueChangeListener(event ->
{
  // ...
}).valueChangeMode(ValueChangeMode.LAZY) ①
    .valueChangeTimeout(1000) ②
    .build();

boolean supported = stringInput.isValueChangeModeSupported(); ③
if (supported) {
  stringInput.setValueChangeMode(ValueChangeMode.BLUR); ④
}
```

① Configure the `ValueChangeMode` using the Input builder. The builder methods to configure the

value change mode are available only for Input components which support it.

② Configure the value change timeout using the Input builder

③ Check if the value change mode configuration is supported by the `Input` component

④ Set the value change mode to `BLUR` using the `Input` API

7.2. ValidatableInput

The ValidatableInput interface represents an `Input` component which supports **value validation** using the standard Holon Platform Validator API.

A `ValidatableInput` can be obtained in the following ways:

1. From an existing `Input` or `HasValue` component: Using the `from` static method of the `ValidatableInput` API.

```
Input<String> stringInput = Components.input.string().build();

ValidatableInput<String> validatableInput = ValidatableInput.from(stringInput); ①

validatableInput = ValidatableInput.from(new TextField()); ②
```

① Create a `ValidatableInput` from a String Input

② Create a `ValidatableInput` from a `TextField`

2. Using the `Input` builder: Using the `validatable()` builder method, which returns a `ValidatableInputBuilder` API that can be used to configure the `ValidatableInput`, for example adding one or more `Validator`, before returning it.

```
validatableInput = Components.input.string() //
    .caption("The caption") //
    .validatable() ①
    .required("Value is required") ②
    .withValidator(Validator.max(100)) ③
    .build();
```

① Get the builder API to obtain a `ValidatableInput`

② Add a default *required* (the input is not empty) validator using the provided error message

③ Add a `Validator

7.2.1. Registering Validators and perform validation

The `addValidator` method of the `ValidatableInput` API can be used to register value validators for the `Input` component.

The `Validator` API can be used to obtain a set of builtin validators, allowing to specify the validation error message, even using a *localizable* message code. See the Holon Platform Validator API for

details.

```
ValidatableInput<String> validatableInput = Components.input.string().validatable()
.build(); ①

validatableInput.addValidator(Validator.max(100)); ②
validatableInput.addValidator(Validator.email("Must be a valid e-mail address",
"invalid.email.message.code")); ③
validatableInput
    .addValidator(Validator.create(value -> value.length() >= 3, "Must be at least 3
characters long")); ④
```

① Create a `String` type `ValidatableInput`

② Add a builtin `Validator` to check the Input value is maximum 100 characters long, using the default validation error message

③ Add a builtin `Validator` to check the Input value is a valid e-mail address, providing a *localizable* validation error message

④ Add a `Validator` to check the Input value is at least 3 characters long and provide the validation error message

The actual value validation can be performed using the Validatable API, which provides the following methods:

- `validate()`: Validate the current `Input` value using the registered `Validator` s. Throws a `ValidationException` if the value is not valid, which provides the validation error message/s. The `ValidationException` is `Localizable`, providing optional message code and arguments for validation message localization.
- `isValid()`: Returns whether the current `Input` value is valid, swallowing any `ValidationException`.

```
try {
  validatableInput.validate(); ①
} catch (ValidationException e) {
  // do something at validation failure
}

boolean valid = validatableInput.isValid(); ②
```

① Validate the `Input` value. The `ValidationException` is thrown if the validation fails.

② Checks whether the current `Input` value is valid, swallowing any `ValidationException`.

7.2.2. Automatic validation at `Input` value change

Value validation can be automatically triggered each time the `Input` value changes.

To enable this behaviour, either the `setValidateOnValueChange` method of the `ValidatableInput` API or the `validateOnValueChange` method of the `ValidatableInput` builder API can be used.

```
ValidatableInput<String> validatableInput = Components.input.string().validatable() //
    .validateOnValueChange(true) ①
    .build();

validatableInput.setValidateOnValueChange(false); ②
```

① Use the `ValidatableInput` builder API to enable the automatic value validation at `Input` value change

② Use the `ValidatableInput` API to disable the the automatic value validation at `Input` value change

7.2.3. Validation errors notification

By default, the standard Vaadin **component error notification** is used to notify validation errors on `ValidatableInput` components. As per default behaviour, an invalid component is decorated with a proper CSS style class (for example to show a red border around the component itself) and the validation error is notified using a *tooltip*.

To change this behavior and to control how the validation errors are notified to the user, the ValidationStatusHandler API can be used.

A `ValidationStatusHandler` listens to validation status change events and the provided validation event can be used to:

- Obtain the current component **validation status**: `UNRESOLVED` (no value validation has been made yet), `VALID` (validation passed), `INVALID` (validation failed).
- Obtain the current **validation error messages** if the component is in the `INVALID` status. The error messages can be obtained as `Localizable` to support message localization.
- Get the **source component** against whom the validation was performed.

The `ValidationStatusHandler` can be configured either using the `setValidationStatusHandler` method of the `ValidatableInput` API or the `validationStatusHandler` method of the `ValidatableInput` builder API.

```
ValidatableInput<String> validatableInput = Components.input.string().validatable() //
    .validationStatusHandler(event -> { ①
      Status status = event.getStatus();
      // ....
    }).build();

validatableInput.setValidationStatusHandler(event -> { ②
  event.getError();
  // ...
});
```

① Use the `ValidatableInput` builder API to configure a `ValidationStatusHandler`

② Use the `ValidatableInput` API to configure a `ValidationStatusHandler`

The `ValidationStatusHandler` also provides static methods to obtain a default validation status handler either using a `Label` component or a Vaadin `Notification` component.

```
Label statusLabel = new Label();
ValidationStatusHandler vsh = ValidationStatusHandler.label(statusLabel); ①

ValidationStatusHandler usingNotifications = ValidationStatusHandler.notification();
②
```

① Create a `ValidationStatusHandler` using a `Label` to display validation errors.

② Create a `ValidationStatusHandler` using a `Notification` to notify validation errors.

7.3. Selectable Input

When the Input values must be selected from a specific set, a *selectable* `Input` type can be used. Two *selectable* Input types are available:

- The SingleSelect Input, in which **at most one item** can be selected at a time;
- The MultiSelect Input, in which **multiple items** can be selected at the same time.

Both *selectable* `Input` types extends the Selectable API, which provides methods to check if some item is selected, obtain the selected item/s and change the current selection.

A *selectable* `Input` can be obtained using the Components API, through the `singleSelect` and `multiSelect` methods and their specializations.

Furthermore, a SelectionListener can be registered to listen to selection changes.

SingleSelect example

```
SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class) ①
    .caption("Single select").build();

singleSelect.setValue(new TestData(1)); ②
singleSelect.select(new TestData(1)); ③

singleSelect.clear(); ④
singleSelect.deselectAll(); ⑤

boolean selected = singleSelect.isSelected(new TestData(1)); ⑥

singleSelect.addSelectionListener(
    s -> s.getFirstSelectedItem().ifPresent(i -> Notification.show("Selected: " + i
.getId()))); ⑦
```

① Create a `SingleSelect` Input using the `TestData` class as selection item type

② Select a value using the default `setValue()` Input method

③ Select a value using `select()` method

④ Deselect the value using the default `clear()` Input method

⑤ Deselect the value using the `deselectAll()` method

⑥ Check whether a value is selected

⑦ Add a selection listener

MultiSelect example

```
MultiSelect<TestData> multiSelect = Components.input.multiSelect(TestData.class) ①
    .caption("Multi select").build();

Set<TestData> values = new HashSet<>();
values.add(new TestData(1));
values.add(new TestData(2));

multiSelect.setValue(values); ②
multiSelect.select(new TestData(3)); ③

multiSelect.deselect(new TestData(3)); ④

multiSelect.clear(); ⑤
multiSelect.deselectAll(); ⑥

boolean selected = multiSelect.isSelected(new TestData(1)); ⑦

multiSelect.addSelectionListener(s -> Notification.show(s.getAllSelectedItems().
stream()
    .map(i -> String.valueOf(i.getId())).collect(Collectors.joining(";", "Selected: ",
"")))); ⑧
```

① Create a `MultiSelect` Input using the `TestData` class as selection item type

② Select a set of values using the default `setValue()` Input method

③ Add a value to current selection using the `select()` method

④ Remove a value from current selection

⑤ Deselect the value using the default `clear()` Input method

⑥ Deselect the value using the `deselectAll()` method

⑦ Check whether a value is selected

⑧ Add a selection listener

7.3.1. Rendering mode

The visual aspect of the *selectable* Input can be configured using the `RenderingMode` enumeration. According to the rendering mode, a suitable UI component will be used to implement the Input.

The default rendering modes are `SELECT` for `SingleSelect` inputs and `OPTIONS` for `MultiSelect` inputs.

Rendering mode	SingleSelect UI component	MultiSelect UI component
NATIVE_SELECT	`com.vaadin.ui.NativeSelect`	*not supported*
SELECT	`com.vaadin.ui.ComboBox`	`com.vaadin.ui.ListSelect`
OPTIONS	`com.vaadin.ui.CheckBoxGroup`	`com.vaadin.ui.RadioButtonGroup`

```
SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class,
RenderingMode.OPTIONS) ①
    .build();

MultiSelect<TestData> multiSelect = Components.input.multiSelect(TestData.class,
RenderingMode.SELECT) ②
    .build();
```

① Create a `SingleSelect` using the **OPTIONS** rendering mode: a `CheckBox` group UI component will be used

② Create a `MultiSelect` using the **SELECT** rendering mode: a `ListSelect` UI component will be used

7.3.2. Using *selectable* Inputs for Enumerations

The Components API provides convenience methods to build a *selectable* `Input` using the values of an **enumeration** as selection items.

The enumeration *selectable* `Input` can be obtained either as a **single** select or a **multiple** select, even specifying the `RenderingMode` to use.

```
SingleSelect<MyEnum> singleSelect = Components.input.enumSingle(MyEnum.class).build();
①
singleSelect = Components.input.enumSingle(MyEnum.class, RenderingMode.NATIVE_SELECT)
.build(); ②

MultiSelect<MyEnum> multiSelect = Components.input.enumMulti(MyEnum.class).build(); ③
```

① Create a `SingleSelect` Input for the `MyEnum` enumeration

② Create a `SingleSelect` Input for the `MyEnum` enumeration using `NATIVE_SELECT` as rendering mode

③ Create a `MultiSelect` Input for the `MyEnum` enumeration

7.3.3. Selection items data source

The data source of the available items for a *selectable* `Input` can be configured in the following ways:

- Using a fixed set of items, which can be provided using the `addItem(…)` or `items(…)` methods of the select input builder.
- Using a Holon Platform ItemDataProvider.
- Using a standard Vaadin `DataProvider`.

```
SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class)
    .items(new TestData(1), new TestData(2)) ①
    .build();

singleSelect = Components.input.singleSelect(TestData.class)
    .dataSource(ItemDataProvider.create(q -> 2, (q, o, l) -> Stream.of(new TestData(1
), new TestData(2)))) ②
    .build();

singleSelect = Components.input.singleSelect(TestData.class)
    .dataSource(DataProvider.ofItems(new TestData(1), new TestData(2))) ③
    .build();
```

① Create a select input using an explicitly provided items set

② Create a select input using a Holon platform `ItemDataProvider`

③ Create a select input using a Vaadin `DataProvider`

7.3.4. Item captions and icons

The *selectable* input builders allow to explicitly set the selection item **captions** and **icons**. For the items caption configuration, the Holon Platform **localization** architecture is supported, and the builders provide methods to specify the captions using a `Localizable` object or a *message code*.

See Messages localization support for details about messages localization.

```
final TestData ONE = new TestData(1);
final TestData TWO = new TestData(2);

SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class)
.items(ONE, TWO)
    .itemCaption(ONE, "One") ①
    .itemCaption(ONE, "One", "caption-one-message-code") ②
    .itemIcon(ONE, VaadinIcons.STAR) ③
    .build();
```

① Set the item caption for the `ONE` item

② Set the localizable item caption for the `ONE` item, providing a default caption and a translation message code

③ Set the item icon for the `ONE` item

Furthermore, the `ItemCaptionGenerator` and the `ItemIconGenerator` interfaces can be used to provide custom selection items **captions** and **icons**.

By default, the item caption is obtained using the `toString()` method of the item class and no icon is displayed.

```
SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class)
    .items(new TestData(1), new TestData(2)) // set the items
    .itemCaptionGenerator(i -> i.getDescription()) ①
    .itemIconGenerator(i -> i.getId() == 1 ? VaadinIcons.STAR : VaadinIcons.STAR_O) ②
    .build();
```

① Set an item caption generator which provides the item description as item caption

② Set an item icon generator

7.3.5. `SingleSelect` caption filter

For the `SingleSelect` type Inputs which use the **SELECT** rendering mode, the UI component allows the user to type a *filtering* text to search for a selection item, and such filter by default is referred to item captions.

When using a **fixed item set**, the in-memory caption filter can be customized using the `filteringMode(…)` builder method.

```
SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class)
    .items(new TestData(1), new TestData(2)) // set the items
    .filteringMode(
        (itemCaption, filterText) -> itemCaption.toLowerCase().startsWith(filterText
.toLowerCase())) ①
    .build();
```

When using a `ItemDataProvider` or a Vaadin `DataProvider`, an overloaded form of the `dataSource(…)` method can be used, which accept as second parameter a function to provide a suitable query filter using the current user filtering text.

```
final TestData ONE = new TestData(1);
final TestData TWO = new TestData(2);

SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class)
.dataSource(
    ItemDataProvider.create(q -> 2, (q, o, l) -> Stream.of(ONE, TWO)),
    filterText -> QueryFilter.startsWith(ConstantExpression.create("description"),
filterText, true))
    .build();
```

7.3.6. Using the `Property` model with *selectable* Inputs

The Holon platform Property model is fully supported by the selectable Input builders, which allows to create selectable Inputs using `PropertyBox` type items and a specific `Property` as selection property (typically, the property which acts as item id).

The selection data type will be the same as the selection property type.

The simplest way to configure a `Property` based selectable input, is by using a Datastore.

```
Datastore datastore = obtainDatastore();

final PathProperty<Long> ID = PathProperty.create("id", Long.class);
final PathProperty<String> DESCRIPTION = PathProperty.create("description", String
.class);

SingleSelect<Long> singleSelect = Components.input.singleSelect(ID) ①
    .dataSource(datastore, DataTarget.named("testData"), PropertySet.of(ID,
DESCRIPTION)) ②
    .itemCaptionGenerator(propertyBox -> propertyBox.getValue(DESCRIPTION)) ③
    .build();

singleSelect.setValue(Long.valueOf(1)); ④
Long selectedId = singleSelect.getValue(); ⑤

singleSelect.refresh(); ⑥
```

① Create a `SingleSelect` using the `ID` property as selection property

② Set the select data source using a `Datastore`, specifying the `DataTarget` and the set of property to use as query projection

③ Set an item caption generator which provides the value of the `DESCRIPTION` property as item caption

④ Set the selected value, which will be of type `Long`, consistent with the `ID` selection property

⑤ Get the selected value, which will be of type `Long`, consistent with the `ID` selection property

⑥ Refresh the selection items, querying the `Datastore`

One or more `QueryConfigurationProvider` can be used to control the `Datastore` query configuration, providing additional query filters and/or sorts.

```
Datastore datastore = obtainDatastore();

final PathProperty<Long> ID = PathProperty.create("id", Long.class);
final PathProperty<String> DESCRIPTION = PathProperty.create("description", String
.class);

SingleSelect<Long> singleSelect = Components.input.singleSelect(ID)
    .dataSource(datastore, DataTarget.named("testData"), PropertySet.of(ID,
DESCRIPTION), // Datastore
        () -> ID.gt(0L)) ①
    .itemCaptionGenerator(propertyBox -> propertyBox.getValue(DESCRIPTION)).build();
```

① Set a `QueryConfigurationProvider` which provides a `QueryFilter` to select only the items with the `ID` property that is greater than 0

Selection model converter

When the Holon platform `Property` model is used for selectable Inputs and the selection value type is not a `PropertyBox`, i.e. the selection items are not of the same type of the `Input` value type, a conversion function is required to obtain the `PropertyBox` item from a selection value when the value is setted in the Input component, either using `setValue(⋯)` or `select(⋯)`.

For example, if a `String` type `Property` is declared as selection property, the `Input` value will be the the value of the selection property (of `String` type), while the selection items are `PropertyBox`.

When a `Datastore` is provided as the data source of the select, a default item converter is configured. The default item converter tries to obtain the `PropertyBox` item by performing a query on the `Datastore` and using a filter predicate to get the item for which the selection property value is equal to the value to select, assuming that the selection property acts as primary key.

To change the default conversion strategy, a custom item converter can be provided to the select input builder using the `itemConverter(⋯)` method:

```
Datastore datastore = obtainDatastore();

final PathProperty<Long> ID = PathProperty.create("id", Long.class);
final PathProperty<String> DESCRIPTION = PathProperty.create("description", String
.class);
final DataTarget<?> TARGET = DataTarget.named("testData");
final PropertySet<?> PROPERTIES = PropertySet.of(ID, DESCRIPTION);

SingleSelect<Long> singleSelect = Components.input.singleSelect(ID)
    .dataSource(datastore, DataTarget.named("testData"), PROPERTIES)
    .itemConverter(
        value -> datastore.query().target(TARGET).filter(ID.eq(value)).findOne
(PROPERTIES).orElse(null)) ①
    .build();
```

① Set an `ItemConverter` providing the query to perform to obtain the item `PropertyBox` from the selection `ID` property value

7.4. Input groups

The PropertyInputGroup component allows to group and manage a set of `Input` fields, relying on the `Property` model and using `PropertyBox` objects to collect, set and provide the Input values.

See the Property documentation for further information about the Holon Platform `Property` model.

A `PropertyInputGroup` is bound to a *set of properties*, and for each property an `Input` field is made available to set and get the property value. Each property has to be bound to an `Input` field with a consistent value type.

7.4.1. Property Inputs rendering

By default, the Holon platform Property rendering architecture is used to automatically generate a suitable `Input` component for each `Property` of Input group property set, according to the `Property` type.

The `Input` type is used as target rendering type to generate the `Input` components.

> See Property renderers and presenters for further information about Property renderers.

```
final NumericProperty<Long> ID = NumericProperty.longType("id");
final StringProperty DESCRIPTION = StringProperty.create("description");

final PropertySet<?> PROPERTIES = PropertySet.of(ID, DESCRIPTION);

PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) ①
    .build();
```

① Configure the input group *property set*. The property `Input` fields are automatically generated using the available *Property Rendereres* from the default registry

7.4.2. Explicit `Input` and `HasValue` binding

To explicitly set the `Input` component to use for a `Property` of the input group property set, the `bind(…)` builder methods of the `PropertyInputGroup` builder API can be used.

The `bind(…)` builder method can therefore be used to override the default property `Input` generation only when a custom `Input` type is required.

The standard `HasValue` Components are also supported to define a `Property` binding.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES)
    .bind(ID, Components.input.number(Long.class).build()) ①
    .bind(DESCRIPTION, new TextField()) ②
    .build();
```

① Bind the `ID` property to a `Long` type `Input` component

② Bind the `DESCRIPTION` property to a `TextField`

7.4.3. Input value convertes

When an `Input` component or `HasValue` component must be bind to a Property with a **different type**, a *value converter* can be provided to convert the Input value to the required Property type and vice-versa.

Specialized `bind` methods of the `PropertyInputGroup` builder API are available to use a standard Vaadin `com.vaadin.data.Converter` to provide the conversion logic.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES)
    .bind(ID, Components.input.string().build(), new StringToLongConverter("Conversion
error")) ①
    .bind(ID, new TextField(), new StringToLongConverter("Conversion error")) ②
    .build();
```

① Bind the `ID` property to a `String` type `Input` component and use a `StringToLongConverter` to convert the String values to the required `Long` property type

② Bind the `ID` property to a `TextField` and use a `StringToLongConverter` to convert the String values to the required `Long` property type

7.4.4. Manage the property values using a `PropertyBox`

The property values are setted and obtained using the `PropertyBox` data container, using the same *property set* of the input group.

The property values are automatically updated in the current `PropertyBox` according to any user modification made through the Input's UI components.

The property input group supports `ValueChangeListener` registration to be notified when the group `PropertyBox` value changes.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES)
.build();

group.setValue(PropertyBox.builder(PROPERTIES).set(ID, 1L).set(DESCRIPTION,
"TestDescription").build()); ①

PropertyBox value = group.getValue(); ②

group.addValueChangeListener(e -> { ③
  PropertyBox changedValue = e.getValue();
});
```

① Set the input group property values using a `PropertyBox` with a matching *property set*

② Get the input group property values as a `PropertyBox` instance

③ Add an input group value change listener

7.4.5. Read-only properties

A property can be setted as **read-only**, to prevent the modification of its value. When a property is read-only, the `Input` component will be in *read-only mode* too, preventing the user from changing the value.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .readOnly(ID) ①
    .build();
```

① The `ID` property is setted as read-only, preventing the user from changing the property value

7.4.6. Hidden properties

A property can be setted as **hidden**. When a property is hidden, it is part of the group property set and its value will be preserved in the `PropertyBox` instances, but no `Input` component will be generated and bound to the property itself, so its value will never be visible on the user interface.

Inputs bound to hidden properties are never returned from the `PropertyInputGroup` methods which allow to inspect the available binding, such as `getInputs()`, `getInput(Property<T> property)` or `stream()`.

The `hidden(Property property)` method of the group builder can be used to set a property as hidden.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .hidden(ID) ①
    .build();
```

① The `ID` property is setted as hidden and will not be displayed nor managed by the input group

7.4.7. Default property values

A **default value** can be provided for a property. If available, the default value will be used as property value when the corresponding `Input` component is empty (i.e. has no value).

The `defaultValue(Property<T> property, DefaultValueProvider<T> defaultValueProvider)` method of the group builder can be used to set a default value for a property.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .defaultValue(DESCRIPTION, property -> "Default") ①
    .build();
```

① Use a `DefaultValueProvider` to provide the `DESCRIPTION` property default value

7.5. Input group validation

A `PropertyInputGroup` support both **property values** and **group value** validation, supporting both the Holon Platform `Validator` API and the standard Vaadin validators.

When a validator in bound to the overall Input group, the value to validate will be represented by the `PropertyBox` type, which contains all the current property values.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .withValidator(DESCRIPTION, Validator.max(100)) ①
    .withValidator(ID, com.vaadin.data.Validator.from(id -> id != null, "Id must be
not null")) ②
    // group validation
    .withValidator(Validator.create(propertyBox -> propertyBox.getValue(ID) > 0,
        "The ID value must be greater than 0")) ③
    .build();
```

① Add a `Validator` bound to the `DESCRIPTION` property to check that the value size is not greater than 100 characters

② Add a `com.vaadin.data.Validator` to check the `ID` property value is not null

③ Add an overall (group) `Validator` which reads the `ID` property value from the group `PropertyBox` value and checks it is greater than 0

Furthermore, a set of `required(…)` group builder methods are provided to add a **required validator** to a property (i.e. a validator that check the value is not `null` or empty) and to **show the required indicator symbol** along with the property `Input` component.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .required(ID) ①
    .required(ID, "The ID value is required") ②
    .build();
```

① Add a *required* validator to the `ID` property

② Add a *required* validator to the `ID` property providing a custom validation error message

The input group validation can be requested using the `validate()` and it is performed by invoking:

1. All the available **`Property` validators** for each property of the input group property set;
2. Then the **overall group validators**, if any.

Besides the `validate()` method, the input group validation is performed by default when the Input group `PropertyBox` value is requested using the `getValue()` method. To skip the group validation when requesting the Input group value, a `getValue(boolean validate)` method is available and `false` can be specified for the *validate* parameter.

A further `getValueIfValid()` method can be used to obtain the Input group value **only when it is valid**, after performing the validations. If validation fails, an empty `Optional` is returned.

```
PropertyInputGroup group = createInputGroup();

try {

  group.validate(); ①

  PropertyBox value = group.getValue(); ②

} catch (ValidationException e) {
  // validation failed
}

PropertyBox value = group.getValue(false); ③

group.getValueIfValid().ifPresent(propertyBox -> { ④
  // ...
});
```

① Explicit group validation: if some kind of validation failure occurs, a `ValidationException` is thrown

② Using the default `getValue()` method, the validation is triggered automatically, and a `ValidationException` is thrown if validation fails

③ The `getValue(boolean validate)` method can be used to skip value validation

④ The `getValueIfValid()` returns an `Optional`, which contains the `PropertyBox` value only if the validation succeded

7.5.1. `Property` validators

Since the Property API supports direct validators registration for each `Property` instance, the property validators are **inherited by default** as Input group validators for each Property of the Input group property set.

This behavior can be disabled using the `ignorePropertyValidation()` method of the `PropertyInputGroup` builder API.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .ignorePropertyValidation() ①
    .build();
```

① Set to ignore any Property `Validator` when binding the property to an `Input` component, i.e. to not inherit property validators when the property-Input binding is performed.

7.5.2. Validation configuration

The Input group provides some configuration options to tune the validation strategy. The validation configuration can be controlled through the `PropertyInputGroup` builder API in order to:

- Enable or disable the automatic `Input` validation when the Input value changes (this behavior is enabled by default).
- Whether to stop property validation at first validation failure. When active, only the first `ValidationException` is thrown. Otherwise, a `ValidationException` which groups all the occurred validation exceptions is thrown, providing all the validation error messages (the default behavior).
- Whether to stop overall group validation at first validation failure. When active, only the first `ValidationException` is thrown. Otherwise, a `ValidationException` which groups all the occurred validation exceptions is thrown, providing all the validation error messages (the default behavior).

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .validateOnValueChange(false) ①
    .stopValidationAtFirstFailure(true) ②
    .stopOverallValidationAtFirstFailure(true) ③
    .build();
```

① Disable the automatic validation at `Input` value change

② Stop property validation at first validation failure

③ Stop overall group validation at first validation failure

7.5.3. Validation errors notification

A ValidationStatusHandler can be used to control how the validation errors are notified to the user, listening to validation status change events of the `PropertyInputGroup`.

See Validation errors notification for details about the `ValidationStatusHandler` API.

The property input group supports `ValidationStatusHandler` configuration both for property and overall validation.

```
Label statusLabel = new Label();

PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .validationStatusHandler(validationEvent -> { ①
      // ...
    }).propertiesValidationStatusHandler(validationEvent -> { ②
      // ...
    }).validationStatusHandler(statusLabel) ③
    .build();
```

① Configure a `ValidationStatusHandler` for the overall (group) validation

② Configure a `ValidationStatusHandler` for the properties validation

③ Convenience method to use a `Label` as validation status handler

7.6. Handling value changes

The `PropertyInputGroup` builder API provides methods to register one or more `ValueChangeListener` in order to:

- Listen to `Input` value changes for the `Input` components bound to group *property set*.
- Listen to the Input group value changes, where the value type is represented by the `PropertyBox` type.

Furthermore, the `PropertyInputGroup` builder API makes available a set of methods to configure the value change notification strategy, i.e. the mode and the frequency with which the value change events are triggered, both for property level and group level value change events.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .withValueChangeListener(e -> { ①
      PropertyBox value = e.getValue();
      // ...
    }).withValueChangeListener(DESCRIPTION, e -> { ②
      String description = e.getValue();
      // ...
    }) //
    .valueChangeMode(ValueChangeMode.BLUR) ③
    .valueChangeMode(DESCRIPTION, ValueChangeMode.LAZY) ④
    .valueChangeTimeout(DESCRIPTION, 500) ⑤
    .build();
```

① Add an overall value change listener

② Add a value change listener to the `Input` bound to the `DESCRIPTION` property

③ Set the default `ValueChangeMode` for all inputs

④ Set the `ValueChangeMode` only for the `Input` bound to the `DESCRIPTION` property

⑤ Set the value change timeout to 500 ms only for the `Input` bound to the `DESCRIPTION` property

In addition to the standard `ValueChangeListener`, the PropertyInputValueChangeListener interface is also supported to listen to **Property/Input pairs value changes**.

The `PropertyInputValueChangeListener` provides the current `PropertyInputBinder` reference, besides of the standard `ValueChangeEvent`. The `PropertyInputBinder` makes available several methods to inspect the input group to which the value change source belongs, for example in order to obtain the values of other `Input` components.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .withValueChangeListener(DESCRIPTION, (event, binder) -> { ①
      String description = event.getValue();
      Long id = binder.requireInput(ID).getValue();
      // ...
    }).build();
```

① Add a `PropertyInputValueChangeListener` to the Input bound to the `DESCRIPTION` property, which provides the current `PropertyInputBinder` to inspect all available group Inputs

7.7. Dealing with *Virtual* properties

By default, the `PropertyInputGroup` renders as an Input all the properties of the property set that has been provided at build time, including any VirtualProperty.

Since a `VirtualProperty` is a *read-only* property, the `Input` bound to a `VirtualProperty` is configured in **read-only mode** by default.

If the `VirtualProperty` value is calculated using the values of other properties of the `PropertyInputGroup` property set, can be useful to *refresh* the displayed `Input` value bound to the virtual property when one or more property of set changes.

For this purpose, the `refresh(…)` methods of the `PropertyInputGroup` can be used. This methods are available from the `PropertyInputBinder` super-interface. You can refresh a specific property or all the properties of the set, even only the *read-only* ones.

```
final VirtualProperty<String> VIRTUAL = VirtualProperty.create(String.class,
    propertyBox -> propertyBox.getValue(ID) + " -" + propertyBox.getValue(DESCRIPTION
));

final PropertySet<?> PROPERTIES = PropertySet.of(ID, DESCRIPTION, VIRTUAL);

PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .withValueChangeListener(DESCRIPTION, (event, binder) -> { ①
      binder.refresh(VIRTUAL);
    }) //
    .build();
```

① Add a `PropertyInputValueChangeListener` to the `Input` bound to the `DESCRIPTION` property, and refresh the `VIRTUAL` property `Input` value when the `DESCRIPTION` property value changes

To disable the read-only properties rendering and management, the `excludeReadOnlyProperties()` builder method can be used.

```
PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .excludeReadOnlyProperties() ①
    .build();
```

① Exclude any *read-only* property from `Input` rendering and binding

7.8. Input Forms

The PropertyInputForm component represents a PropertyInputGroup which is also a UI `Component`, and thus can be directly used to display and manage the `Input` components in the UI.

It extends the ComposableComponent API, which represents an UI component with a **base layout** and the capability to ***compose* a set of UI components on this layout**.

The Composer API is used to *compose* the `Input` components on the UI, and must be provided to the `PropertyInputForm` at configuration time, along with the base layout component on which the `Input` components will be organized.

When a `PropertyInputForm` component is obtained using the Components API, a set of methods are available to create a `PropertyInputForm` using one of the standard Vaadin layouts. Using these methods, a default `ComponentContainer` *composer* is provided, which adds the `Input` components to the base layout in the order they are provided by the `PropertyInputForm` property set.

> The `ComposableComponent` interface provides the `componentContainerComposer()` static method to obtain a default *composer* which uses a Vaadin `ComponentContainer` as base layout.

```
PropertyInputForm form = Components.input.form().properties(PROPERTIES).build(); ①
form = Components.input.formVertical().properties(PROPERTIES).build(); ②
form = Components.input.formHorizontal().properties(PROPERTIES).build(); ③
form = Components.input.formGrid().properties(PROPERTIES).build(); ④
```

① Create a `PropertyInputForm` using a `FormLayout` as base content layout

② Create a `PropertyInputForm` using a `VerticalLayout` as base content layout

③ Create a `PropertyInputForm` using a `HorizontalLayout` as base content layout

④ Create a `PropertyInputForm` using a `GridLayout` as base content layout

7.8.1. Base layout initialization

A `initializer(…)` builder method is available to perform custom configuration of the *base layout* component of the `PropertyInputForm`.

```
PropertyInputForm form = Components.input.formGrid().properties(PROPERTIES) //
    .initializer(gridLayout -> { ①
      gridLayout.setSpacing(true);
      gridLayout.addStyleName("my-style");
    }).build();
```

① Provide base layout component initialization using an *initializer*

7.8.2. `Input` components composition

The Composer API is used to *compose* the `Input` components on the `PropertyInputForm` base layout, and can be provided at configuration time either to provide a custom component composition strategy.

The `Input` components *composition* using the currently configured `Composer` is triggered by the `compose()` method. By default, the components composition is **automatically performed when the**

`PropertyInputForm` **component is a attached to a parent layout**.

```
PropertyInputForm form = Components.input.form(new FormLayout()) ①
    .properties(PROPERTIES).composer((layout, source) -> { ②
      source.getValueComponents().forEach(c -> layout.addComponent(c.getComponent()));
    }).build();
```

① A `FormLayout` instance is provided as base layout component

② Configure a `Composer` to *compose* the `Input` components on the form base layout. The `PropertyComponentSource` type argument can be used to obtain the available `Input` components

To disable this behaviour, the `composeOnAttach(…)` method of the `PropertyInputForm` builder API can be used.

```
PropertyInputForm form = Components.input.form(new FormLayout()).properties(
PROPERTIES) //
    .composer((layout, source) -> {
      source.getValueComponents().forEach(c -> layout.addComponent(c.getComponent()));
    }).composeOnAttach(false) ①
    .build();

form.compose(); ②
```

① Disable the automatic `Input` components composition when the `PropertyInputForm` component is a attached to a parent layout

② Explicitly use the `compose()` method to perform components composition

7.8.3. `Input` components width mode

When the `Input` components are *composed* on a `PropertyInputForm` content component, the ComponentsWidthMode enumeration can be used to configure the strategy to use to handle the **component widths**.

The available strategies are:

- **NONE**: The width of each `Input` component will not be altered.
- **AUTO**: The width of each `Input` component will adjusted according to parent layout width: if the parent layout is 100% wide, the components width will be setted to 100%.
- **FULL**: The width of each `Input` component will always be setted to 100%.

The **default strategy** is `AUTO`.

The strategy can be changed using the `componentsWidthMode` method of the `PropertyInputForm` builder API.

```
PropertyInputForm form = Components.input.form().properties(PROPERTIES)
    .componentsWidthMode(ComponentsWidthMode.FULL) ①
    .build();
```

① Set the `ComponentsWidthMode` to `FULL`: all the form `Input` components will be 100% wide

7.8.4. `Input` components configuration

The `ComposableComponent` type builders, including the `PropertyInputForm` builder API, allow to provide custom *configurator* functions in order to configure the `Input` components just after they are *rendered*.

For each `Property` of the `PropertyInputForm` property set, you can provide a `Consumer` to use as `ComponentConfigurator`, which makes available a complete set of methods to configure the Input `Component` associated to a `Property` before rendering it in UI. For example, you can set the component size, the icon, the description and so on.

```
PropertyInputForm form = Components.input.form().properties(PROPERTIES)
    .componentConfigurator(ID, cfg -> cfg.styleName("id-input").description("The ID"))
①
    .componentConfigurator(DESCRIPTION, cfg -> cfg.icon(VaadinIcons.CLIPBOARD_TEXT))
②
    .build();
```

① Configure the `ID` property `Component`

② Configure the `DESCRIPTION` property `Component`

7.9. View components

The ViewComponent API represents a UI component which can be used to **display a value** on the user interface. Unlike an `Input` type component, the **value cannot be changed by the use**.

As a `ValueHolder`, the view component value can be setted and obtained using the `setValue` and `getValue` methods.

Just like an `Input` type component, the actual UI component to display is obtained form the `getComponent()` method.

The Components API can be used to obtain a default `ViewComponent` implementation (backed by a Vaadin `Label` component) for a specific value type, using the `view.component(Class<? extends T> valueType)` method.

```
ViewComponent<String> view = Components.view.component(String.class) ①
    .caption("TheCaption", "caption.message.code") ②
    .icon(VaadinIcons.CAMERA) //
    .styleName("my-style") //
    .build();

view.setValue("TestValue"); ③

String value = view.getValue(); ④
```

① Get a `ViewComponent` builder using a `String` value type

② Configure the `ViewComponent` component

③ Set the `ViewComponent` value

④ Get the `ViewComponent` value

7.9.1. View component Groups and Forms

Just like the `Input` components, a `ViewComponent` set can be organized in *groups* and handled using *forms*.

- The PropertyViewGroup API con be used to create `ViewComponent` **groups**.
- The PropertyViewForm API con be used to create `ViewComponent` **form** components.

These component containers rely on the Holon Platform the Property model to bind each `ViewComponent` to a `Property` and they use the `PropertyBox` type to provide the property values to show.

```
final NumericProperty<Long> ID = NumericProperty.longType("id");
final StringProperty DESCRIPTION = StringProperty.create("description");

final PropertySet<?> PROPERTIES = PropertySet.of(ID, DESCRIPTION);

PropertyViewGroup viewGroup = Components.view.propertyGroup().properties(PROPERTIES)
.build(); ①

PropertyViewForm viewForm = Components.view.formVertical().properties(PROPERTIES)
.build(); ②

viewForm.setValue(PropertyBox.builder(PROPERTIES).set(ID, 1L).set(DESCRIPTION, "Test"
).build()); ③
PropertyBox value = viewForm.getValue(); ④
```

① Create a `PropertyViewGroup` using the `PROPERTIES` property set

② Create a `PropertyViewForm` using the `PROPERTIES` property set and a `VerticalLayout` as base layout

③ Set the form value using a `PropertyBox`

④ Get the form value as a `PropertyBox`

The `PropertyViewGroup` and `PropertyViewForm` provide all the base components configuration capabilities of their corresponding PropertyInputGroup and PropertyInputForm implementations.

So, for example, it is possible to configure a `Composer` to customize the `ViewComponent` components composition strategy for a `PropertyViewForm` or to provide component *initializers* at configuration time.

```
PropertyViewForm viewForm = Components.view.form(new FormLayout()).properties
(PROPERTIES) //
    .initializer(layout -> layout.setMargin(true)) ①
    .composer((layout, source) -> { ②
      source.getValueComponents().forEach(c -> layout.addComponent(c.getComponent()));
    }) //
    .componentConfigurator(DESCRIPTION, cfg -> cfg.styleName("my-style")) ③
    .componentsWidthMode(ComponentsWidthMode.FULL) ④
    .build();
```

① Base layout initializer

② Custom component `Composer`

③ `DESCRIPTION` property `ViewComponent` component configurator

④ Set the `ComponentsWidthMode` to `FULL`: all the form `ViewComponent` components will be 100% wide

7.9.2. `ViewComponent` captions

When the `PropertyViewForm` builder API makes available some additional methods to directly configure the *captions* of the `ViewComponent` managed by the form.

For each `Property` of the form property set, the `ViewComponent` *caption* can be setted using the `propertyCaption` method or hidden using the `hidePropertyCaption` method.

```
PropertyViewForm viewForm = Components.view.form(new FormLayout()).properties
(PROPERTIES) //
    .propertyCaption(DESCRIPTION, "My caption") ①
    .hidePropertyCaption(ID) ②
    .build();
```

① Set the caption for the `ViewComponent` bound to the `DESCRIPTION` property

② Hide the caption for the `ViewComponent` bound to the `ID` property

8. Item listing

The ItemListing API can be used to display a set of items as tabular data using a `Grid` type UI component and to manage the data items.

Each item *property* to display is bound to a *column*, and a property **id** is used to bind the item property to the listing column.

The `ItemListing` is a Selectable component, supporting **items selection** in single or multiple mode.

The `ItemListing` API provides a set of methods to control the listing appearance, showing and hiding columns, sorting and refreshing the item set, and so on.

Two `ItemListing` implementations are available to use this component:

- PropertyListing: Uses a `PropertyBox` as item type and the listing column ids are the `Property` instances of the provided *property set*.
- BeanListing: Uses a **Java Bean** as item type and the listing column ids are the Bean property names.

8.1. PropertyListing

A PropertyListing implementation can be obtained using the Components API, through the `listing.properties` method.

The `Property` set to use to configure the `PropertyListing` must be provided, for each `Property` of the property set a listing column and configured to the `Property` types. Each column will be identified by the `Property` itself.

In particular, for each `Property` of the *property set*:

- The property *caption*, if available, will be used as **column header**. The caption/header *localization* is fully supported and performed if a `LocalizationContext` is available. See Messages localization support for details.
- The cells of the column which corresponds to a property will be rendered according to the **property type** and relying on the available property value *presenters*.
- For any property of `com.vaadin.ui.Component` type, a `ComponentRenderer` will be automatically used to render the Vaadin Component in the listing column cells.

```
final NumericProperty<Long> ID = NumericProperty.longType("id");
final StringProperty DESCRIPTION = StringProperty.create("description");

final PropertySet<?> PROPERTIES = PropertySet.of(ID, DESCRIPTION);

PropertyListing listing = Components.listing.properties(PROPERTIES).build(); ①
```

① Create a `PropertyListing` using given `PROPERTIES` property set

See the Item listing API and configuration to learn about the item listing API operations and configuration options.

8.1.1. Visible properties

By default, all the properties of the listing *property set* are rendered as listing columns, in the same order they are provided by the property set itself.

To select which properties are to be rendered as listing columns and to declare the columns order, the `PropertyListing` builder API provides specialized version of the `build()` method, which accept an array or an Iterable of `Property` to declare the visible columns and their order.

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .build(DESCRIPTION, ID); ①
```

① Set the listing visible columns: the `DESCRIPTION` property first and then the `ID` property

8.1.2. Items type and identifiers

The PropertyBox type is used to represent a `PropertyListing` *item*, i.e. a listing *row*. For each property listing row, the `PropertyBox` type item collects the values of each `Property` bound to the listing columns.

By default, the listing property set **identifier properties** are used to provide the `PropertyBox` items identifiers, if available. See the property set identifier properties documentation for details.

```
final NumericProperty<Long> ID = NumericProperty.longType("id");
final StringProperty DESCRIPTION = StringProperty.create("description");

final PropertySet<?> PROPERTIES = PropertySet.builderOf(ID, DESCRIPTION).identifier(
ID).build(); ①

PropertyListing listing = Components.listing.properties(PROPERTIES).build();
```

① Set the `ID` property as property set identifier property

8.1.3. Items data source

The `PropertyBox` type item set handled by the `PropertyListing` is provided using an items **data source**, which can be configured using the `PropertyListing` builder API using the `dataSource` methods.

The ItemDataProvider API is used to represent the items data source.

An `ItemDataProvider` implementation provides methods to obtain the item set **size** and to load a subset of the items using a **limit** which represents the subset size and a **offset** index to specify which subset to return. For a `PropertyListing`, the items subset is returned as a Stream of `PropertyBox` instances.

The `ItemDataProvider` methods accepts a `QueryConfigurationProvider` instance, to provide the `QueryFilter` and the `QuerySort` to be used to query the item set.

See the Query documentation for information about the *query* definition using the Holon Platform Property model.

```
ItemDataProvider<PropertyBox> dataProvider = ItemDataProvider.create(cfg -> 0, ①
    (cfg, offset, limit) -> Stream.empty()); ②

PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .dataSource(dataProvider) ③
    .build();
```

① Create a `ItemDataProvider` which always returns a 0 size item set

② Return an empty Stream as item set

③ Create a `PropertyListing` using the `dataProvider` as items data source

A convenience method is available to use the Datastore API as items data source, providing the `DataTarget` to use to perform the query on the concrete persistence source.

```
Datastore datastore = getDatastore();

PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .dataSource(datastore, DataTarget.named("test")) ①
    .build();
```

① Use a `Datastore` as data source. The `Datastore` API will be used to perform the queries to obtain the items set, with the given `DataTarget` as query target

8.1.4. Item identifier configuration

When the property set of the `PropertyListing`, used to represent the `PropertyBox` type items, does not provide one or more *identifier properties* as described in the Items type and identifiers section, the ItemIdentifierProvider interface can be used to provide the item identifiers.

```
ItemDataProvider<PropertyBox> dataProvider = getDataProvider();

PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .dataSource(dataProvider, item -> item.getValue(ID)) ①
    .build();

listing = Components.listing.properties(PROPERTIES) //
    .dataSource(getDatastore(), DataTarget.named("test"), ID) ②
    .build();
```

① Create a `PropertyListing` and provide an `ItemDataProvider` as data source, along with an `ItemIdentifierProvider` which provides the value of the `ID` property as item identifier

② A shorter builder method to use the `ID` property as item identifier when a `Datastore` is used as items data source

8.1.5. Item query configuration

One or more `QueryConfigurationProvider` can be registered in the `PropertyListing` to provide additional and dynamic items query configuration elements, such as `QueryFilter` and `QuerySort`.

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .dataSource(getDatastore(), DataTarget.named("test")) //
    .withQueryConfigurationProvider(new QueryConfigurationProvider() { ①

      @Override
      public QueryFilter getQueryFilter() {
        return ID.gt(0L);
      }

      @Override
      public QuerySort getQuerySort() {
        return DESCRIPTION.asc();
      }

    }).build();
```

① Add a `QueryConfigurationProvider` to the `PropertyListing`

8.1.6. Using `VirtualProperty` for `Component` type columns

When a `Property` of the `PropertyListing` property set is declared as `com.vaadin.ui.Component` type, the listing configures by default a suitable *renderer* to display the property value as a Vaadin Component, instead of a `String`.

When a listing `Component` type column has to be **dinamically generated** when the listing is rendered, the VirtualProperty property type can be used.

Since the `VirtualProperty` provides its value through a `PropertyValueProvider`, the provider is invoked when the listing column cells are rendered to obtain the `Component` to display, providing the current row values as `PropertyBox`, which can be used for component generation.

> Of course, a `VirtualProperty` can be used in a listing component not only to provide a `Component` value, but any other supported type.

The `VirtualProperty` can be declared in the listing property set as any other Property, but when it is only used for a specific item listing column to render a `Component` it can be only declared as **visible column**, in the `build(…)` method. This way, it will not be part of the listing property set (and it will not be included in the row `PropertyBox` properties) and will only be used as *generated column*.

```
final VirtualProperty<Component> EDIT = VirtualProperty.create(Component.class)
.message("Edit") ①
    .valueProvider( ②
        row -> Components.button().styleName(ValoTheme.BUTTON_ICON_ONLY).icon
(VaadinIcons.EDIT)
            .onClick(e -> {
              Long rowId = row.getValue(ID); ③
              // perform edit action ...
            }).build());

PropertyListing listing = Components.listing.properties(PROPERTIES) ④
    .build(EDIT, ID, DESCRIPTION); ⑤

listing = Components.listing.properties(PROPERTIES) //
    .build(PropertySet.builder().add(PROPERTIES).add(EDIT).build()); ⑥
```

① Declare a `VirtualProperty` of `Component` type. The configured message will be used as column header.

② The property *value provider* returs a `Button` component

③ Since the row `PropertyBox` is provided at column cell generation time, we can use it to obtain a property value for the current row. In this example, the `ID` property value is obtained.

④ The listing property set do not includes the `EDIT` virtual property

⑤ The `EDIT` virtual property is declared as visible column, in the first position in this case

⑥ Another example in which the visible columns property set is composed joining the `PROPERTIES` set and the `EDIT` virtual property

8.2. BeanListing

A BeanListing implementation can be obtained using the Components API, through the `listing.items` method.

The **Bean class** to use as items type must be provided as construction time.

The Holon Platform Bean introspection API is used to detect the Bean class properties to use as listing columns and the Bean property **names** are used as column ids.

All the suitable bean property configuration attributes supported by the Holon Platform Bean introspection API are used to automatically configure the listing component, for example to set the column *headers* using the localizable property captions.

See the Bean property configuration section to learn about the Bean introspection API configuration capabilities and the available builtin annotations which can be used to provide a set of configuration attributes, such as `@Caption` or `@Sequence`.

For example, given the following bean class:

```
private class TestData {

  private Long id;
  private String description;

  // getters and setters omitted
```

A `BeanListing` which uses the bean class can be obtained as follows:

```
BeanListing<TestData> listing = Components.listing.items(TestData.class) ①
    .build();
```

① Create a `BeanListing` using `TestData` as items data type

See the Item listing API and configuration to learn about the item listing API operations and configuration options.

8.2.1. Bean items data source

The **data source** to provide the bean items for a `BeanListing` can be configured using the `dataSource` builder API methods.

1. Using an `ItemDataProvider` as items data source:

The ItemDataProvider API can be used as items data source.

For example, given a `List` of `TestData` bean instances, an in-memory `ItemDataProvider` can be configured in the following way:

```
final List<TestData> data = Arrays.asList(new TestData(1, "One"), new TestData(2, "
Two"),
    new TestData(3, "Three"));

BeanListing<TestData> listing = Components.listing.items(TestData.class) //
    .dataSource(ItemDataProvider.create(cfg -> data.size(), ①
        (cfg, offset, limit) -> data.stream().skip(offset).limit(limit)))
    .build();
```

① Set an `ItemDataProvider` which uses the `data` List as items data source

2. Using a `Datastore` as items data source:

A Datastore implementation can be used as items data source to obtain the bean items from a persistence source.

The properties which will be used to perform query operations using the `Datastore` API will be the Bean properties obtained as a BeanPropertySet using the Holon Platform Bean introspection API.

```
final Datastore datastore = getDatastore();

BeanListing<TestData> listing = Components.listing.items(TestData.class) //
    .dataSource(datastore, DataTarget.named("test")) ①
    .build();
```

① Use a `Datastore` and given `DataTarget` as items data source

The `BeanListing` API makes available a `refresh()` method to *refresh* the current items set, causing items reloading from the items data source.

```
BeanListing<TestData> listing = Components.listing.items(TestData.class) //
    .dataSource(getDatastore(), DataTarget.named("test")).build();

listing.refresh(); ①
```

① Refresh the items set

8.2.2. Dynamically generated columns

The `BeanListing` component supports *virtual* columns definition to declare **generated** column types, which contents are generated when the listing is rendered, for example to provide `Component` type column contents.

The `withVirtualColumn(String id, Class<V> type, ValueProvider<T, V> valueProvider)` method of the `BeanListing` builder API allows to declare a *virtual* column, providing:

- The virtual column **id** and **type**
- A `ValueProvider` which will be used to generate the column cells, using the row item **bean instance**

The declared virtual column id can later be used to include the virtual column in the visible column in the required position.

```
BeanListing<TestData> listing = Components.listing.items(TestData.class)
    .withVirtualColumn("delete", Button.class, bean -> { ①
      return Components.button().icon(VaadinIcons.TRASH).onClick(e -> {
        Long rowId = bean.getId(); ②
      }).build();
    }).build("delete", "id", "description"); ③
```

① Declare a virtual column of `Button` type and `delete` as column id

② The bean instance of the current row can be used to obtain row values, for example the id attribute in this case

③ Declare the `delete` column as visible and render it as first column

8.3. Item listing API and configuration

The item listing component *builder* API provides a wide set of configuration methods which can be used to:

- Configure the listing columns, header and footer sections.
- Register a number of *listeners* for different event types.
- Configure the items data source query.

See the next sections for details.

8.3.1. Columns configuration

The listing component columns can be configured in a number of ways. Below is provided an example which uses a `PropertyListing` and shows the available column configuration methods:

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .header(ID, "Custom ID header") ①
    .columnHidingAllowed(true) ②
    .hidable(ID, false) ③
    .columnReorderingAllowed(true) ④
    .alignment(ID, ColumnAlignment.RIGHT) ⑤
    .hidden(DESCRIPTION, true) ⑥
    .resizable(ID, false) ⑦
    .width(ID, 100) ⑧
    .expandRatio(DESCRIPTION, 1) ⑨
    .minWidth(DESCRIPTION, 200) ⑩
    .maxWidth(DESCRIPTION, 300) ⑪
    .style(ID, (property, item) -> item.getValue(DESCRIPTION) != null ? "empty" :
"not-empty") ⑫
    .withPropertyReorderListener((properties, userOriginated) -> { ⑬
      // ...
    }).withPropertyResizeListener((property, widthInPixel, userOriginated) -> { ⑭
      // ...
    }).withPropertyVisibilityListener((property, hidden, userOriginated) -> { ⑮
      // ...
    }).build();
```

① Set a custom header caption for the `ID` property/column

② Set whether the listing columns can be hidden

③ Set that the `ID` property/column cannot be hidden

④ Set whether the listing columns can reordered

⑤ Set the `ID` property/column cell contents alignment

⑥ Set the `DESCRIPTION` property/column as hidden by default

⑦ Set the `ID` property/column as not resizable

⑧ Set the `ID` property/column width in pixels

⑨ Set the `DESCRIPTION` property/column expand ratio

⑩ Set the `DESCRIPTION` property/column minimum width

⑪ Set the `DESCRIPTION` property/column maximum width

⑫ Set the `ID` property/column CSS style generator

⑬ Add a listener to be notified when the listing columns order changes

⑭ Add a listener to be notified when a property/column is resized

⑮ Add a listener to be notified when a property/column is shown or hidden

8.3.2. Columns rendering

By default, the cells of a column bound to a `Property` are *rendered* using the Holon Platform *presenters* available from the default registry, to provide the `String` representation of the cell value.

> See the String value presenters documentation for details.

The default column rendering can be overridden using a custom Vaadin `Renderer`. The provided `Renderer` must handle the same data type of the property/column for which it is configured.

Furthermore, a property/column value can be converted to a different value type, and, in this case, a suitable `Renderer` for the converted type must be provided.

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .render(ID, new NumberRenderer(Locale.US)) ①
    .render(ID, value -> String.valueOf(value), new TextRenderer()) ②
    .build();
```

① Set a custom `NumberRenderer` for the `ID` property/column (which is of type `Long`)

② Set a converter to convert the `Long` type value of the `ID` property/column into a `String` and a suitable render according to the new value type

8.3.3. Item listing configuration

The item listing component is highly configurable. Through the builders you can configure:

- The **header appeareance**, adding and removing header rows and joining header columns
- Showing and configuring a **footer** section
- Set a number of column as **frozen**
- Provide **rows CSS style** using a style generator
- Provide item **description** (tooltip) for item rows
- Provide a **item click listener** to listen for user clicks on item rows
- Provide a **row details component** which can be shown for each item row

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .heightByContents() ①
    .frozenColumns(1) ②
    .hideHeaders() ③
    .withRowStyle(item -> { ④
      return item.getValue(DESCRIPTION) != null ? "has-des" : "no-nes";
    }) //
    .itemDescriptionGenerator(item -> item.getValue(DESCRIPTION)) ⑤
    .detailsGenerator(item -> { ⑥
      VerticalLayout component = new VerticalLayout();
      // ...
      return component;
    }).withItemClickListener((item, property, event) -> { ⑦
      // ...
    }).header(header -> { ⑧
      header.getDefaultRow().setStyleName("my-header");
    }).footer(footer -> { ⑨
      footer.appendRow().setStyleName("my-footer");
    }).footerGenerator((source, footer) -> { ⑩
      footer.getRowAt(0).getCell(ID).setText("ID footer");
    }).withPostProcessor(grid -> { ⑪
      // ...
    }).build();
```

① Set the height of the listing defined by its contents

② Set the *frozen* columns count

③ Hide all the listing headers

④ Set the row CSS style generator

⑤ Set the item/row description (tooltip) generator

⑥ Set the item/row *details* component generator

⑦ Register a item click listener

⑧ Add a style class name to the default header row

⑨ Append a row to the listing footer and set a style class name for it

⑩ Set the footer *generator* to invoke to update the listing footer contents

⑪ Set a post-processor to customize the internal `Grid` component

8.3.4. Items selection

The item listing component supports items selection, both in single and multiple mode. The listing can be made selectable by using the `setSelectionMode` method or the `selectionMode` builder method.

Just like any other `Selectable` component, the item listing component provides methods to inspect the current selected item/s and change the current selection.

Single selection mode

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .selectionMode(SelectionMode.SINGLE) ①
    .build();

final PropertyBox ITEM = PropertyBox.builder(PROPERTIES).set(ID, 1L).build();

PropertyBox selected = listing.getFirstSelectedItem().orElse(null); ②

boolean isSelected = listing.isSelected(ITEM); ③

listing.select(ITEM); ④

listing.deselectAll(); ⑤
```

① Set the listing *selectable* in `SINGLE` selection mode

② Get the currently selected item, if any

③ Check whether the given item is selected

④ Select the given item

⑤ Deselect all (clear current selection)

Multiple selection mode

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .build();

listing.setSelectionMode(SelectionMode.MULTI); ①

final PropertyBox ITEM = PropertyBox.builder(PROPERTIES).set(ID, 1L).build();

Set<PropertyBox> selected = listing.getSelectedItems(); ②

boolean isSelected = listing.isSelected(ITEM); ③

listing.select(ITEM); ④

listing.deselectAll(); ⑤

listing.selectAll(); ⑥
```

① Set the listing *selectable* in `MULTI` selection mode

② Get the selected items

③ Check whether the given item is selected

④ Select the given item

⑤ Deselect all items

⑥ Select all items

> The **select all** checkbox shown in the top left corner of the listing is visible by default. You can control the checkbox visibility using the `selectAllCheckBoxVisibility(…)` method of the item listing builder.

8.3.5. Items query configuration

The item listing builder provides a number of methods to control the data source items query, allowing to:

- Set a property/column as **sortable** or not and **override the default sort behaviour**, for example using another property as query sort target or providing a custom `QuerySort`;
- Set a **fixed sort**, which will be always appended to current query sorts
- Set a **default sort**, to be used when no other query sort is configured
- Set a **fixed filter**, which will be always applied to the query
- Register one or more `QueryConfigurationProvider` to provide custom query configuration logic

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .sortable(ID, true) ①
    .sortUsing(ID, DESCRIPTION) ②
    .sortGenerator(ID, (property, ascending) -> { ③
      return ascending ? ID.asc() : ID.desc();
    }) //
    .fixedSort(ID.asc()) ④
    .defaultSort(DESCRIPTION.asc()) ⑤
    .fixedFilter(ID.gt(0L)) ⑥
    .withQueryConfigurationProvider(() -> DESCRIPTION.isNotNull()) ⑦
    .build();
```

① Set the `ID` property/column as sortable

② Set to use the `DESCRIPTION` property to sort the `ID` property/column

③ Set a custom `QuerySort` for the `ID` property/column

④ Set a fixed `QuerySort`, which will always to appended to current query sorts

⑤ Set the default `QuerySort`, to be used when no other query sort is configured

⑥ Set a fixed `QueryFilter`

⑦ Add a `QueryConfigurationProvider`

8.3.6. Item set management

The item set component provides methods to manage the item set which is provided as the listing data source, to add, update and remove one or more items in the set.

To reflect the item set management operations in the underlyng data source (for example, a

RDBMS), the `CommitHandler` interface is used. The `commit(…)` method of that interface is invoked at each item set modification, providing the *added*, *updated* and *removed* items. The `CommitHandler` implementation should persist the item set modifications in the concrete data source.

> When using the `dataSource(…)` builder method which accepts a `Datastore` as data source, a default `CommitHandler` is automatically configured, using the provided `Datastore` to perform the item persistence operations.

```
Datastore datastore = getDatastore();

PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .dataSource(datastore, DataTarget.named("test"), ID) ①
    .commitHandler((addedItems, modifiedItems, removedItems) -> { ②
      // ...
    }).build();

final PropertyBox ITEM = PropertyBox.builder(PROPERTIES).set(ID, 777L).set(
DESCRIPTION, "A description")
    .build();

listing.addItem(ITEM); ③

listing.refreshItem(ITEM); ④

listing.removeItem(ITEM); ⑤
```

① When using the `Datastore` method of the data source builder, a default `CommitHandler` is automatically configured

② You can provide a custom `CommitHandler` using the `commitHandler(…)` builder method

③ Add an item

④ Refresh (update) the item

⑤ Remove the item

8.3.7. Buffered mode

The item set component supports a **buffered** mode to handle its items set. When in *buffered* mode, the item set modifications are cached internally and not reflected to the concrete data source until the `commit()` method is called.

A `discard()` method is provided to discards all changes since last commit.

The same `CommitHandler` as before is used to persist the item set modifications, but it is invoked only when the `commit()` method is called.

```
Datastore datastore = getDatastore();

PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .dataSource(datastore, DataTarget.named("test"), ID) //
    .buffered(true) ①
    .build();

final PropertyBox ITEM = PropertyBox.builder(PROPERTIES).set(ID, 777L).set(
DESCRIPTION, "A description")
    .build();

listing.addItem(ITEM); ②
listing.refreshItem(ITEM); ③
listing.removeItem(ITEM); ④

listing.commit(); ⑤
listing.discard(); ⑥
```

① Set the listing in *buffered* mode

② Add an item

③ Refresh (update) the item

④ Remove the item

⑤ *commit* the item set modifications

⑥ *discard* the item set modifications since the last commit

8.3.8. Editing items

The item listing component supports line-based editing, where double-clicking a row opens the row editor, using the default Vaadin `Grid` editor. The item listing editor must be enabled using the `editable(true)` method of the builder in order to activate the row editor at double-click.

The **editor fields** to use for each listing property/column will be auto-generated according to the property/column type. This behaviour can be overridden providing custom editor fields for one or more of the listing properties.

The item listing component editor supports **validation**, allowing to register value validators both for the single property/column fields and for the overall value validation.

```
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .editable(true) ①
    .editorBuffered(true) ②
    .editorSaveCaption("Save item") ③
    .editorCancelCaption("Discard") ④
    .editable(ID, false) ⑤
    .editor(DESCRIPTION, new TextField()) ⑥
    .withValidator(Validator.create(pb -> pb.getValue(DESCRIPTION) != null,
"Description must be not null")) ⑦
    .withValidator(DESCRIPTION, Validator.max(100)) ⑧
    .required(ID) ⑨
    .build();
```

① Set the listing as editable

② Set the row editor in buffered mode

③ Set the editor *save* button caption

④ Set the editor *cancel* button caption

⑤ Set a property as not editable

⑥ Set a custom editor field

⑦ Add an overall value validator

⑧ Add a property value validator

⑨ Set the `ID` property as required, automatically adding a *not empty* validator

9. Dialogs

The Holon Vaadin module provides builders to create and show **dialog** windows.

A dialog window in represented by the Dialog interface, which makes available methods to set the dialog as *modal*, open and close the dialog window and registere a `CloseListener` to be invoked when the dialog is closed.

```
Dialog dialog = Components.dialog() ①
    .draggable(false) ②
    .closable(true) ③
    .resizable(true) ④
    .modal(true) ⑤
    .message("Dialog message", "dialog.message.code") ⑥
    .okButtonConfigurator(cfg -> cfg.caption("Done").icon(VaadinIcons.CHECK_CIRCLE_O))
⑦
    .withCloseListener((window, action) -> { ⑧
      // ...
    }).build();

dialog.open(); ⑨

dialog.close(); ⑩
```

① Dialog builder

② Set as not draggable

③ Set as closable

④ Set as resizable

⑤ Set as modal

⑥ Set the localizable dialog message

⑦ Configure the dialog *OK* button

⑧ Add a close listener

⑨ Open (show) the dialog

⑩ Close (hide) the dialog

9.1. Question dialogs

A *question* dialog is a `Dialog` with a message and two buttons (instead of one) to answer *yes* or *no* to a question.

A `QuestionCallback` can be used when the user selects one of the two buttons and the dialog is closed. The callback notifies the selected answer.

```
Components.questionDialog() ①
    .message("Can I do it for you?") ②
    .yesButtonConfigurator(cfg -> cfg.caption("Ok, let's do it")) ③
    .noButtonConfigurator(cfg -> cfg.caption("No, thanks")) ④
    .callback(answeredYes -> { ⑤
      Notification.show("Ok selected: " + answeredYes);
    }).build().open(); ⑥
```

① Question dialog builder

② Set the dialog message (the question)

③ Change the *yes* button caption

④ Change the *no* button caption

⑤ Set the answer callback

⑥ Build and open the dialog

10. Property renderers and presenters

The Vaadin module takes advantage of the platform foundation *property model* to provide the most suitable UI components and display values when a `Property` is used.

When a `Property` must be rendered in UI, the components made available by the Vaadin module try to use a suitable `PropertyRenderer`, if available.

By default, the following renderers are automatically registered by the module:

- A Input property renderer
- A Field property renderer, which represents a `HasValue` Vaadin `Component`
- A ViewComponent property renderer

The `Input` and `Field` property renderers generate a UI component to edit a property value, relying on the property type.

The following types are supported:

- **String**: renders the property as a `TextField`
- **Boolean**: renders the property as a `CheckBox`
- **Enum**: renders the property as a `ComboBox`
- **LocalDate and LocalDateTime**: renders the property as a `DateField` or a `DateTimeField`
- **java.util.Date**: renders the property as a `DateField` or a `DateTimeField`
- **Number**: renders the property as a `TextField` which only accepts numbers and decimal/group separators

```
final PathProperty<String> TEXT = PathProperty.create("text", String.class);
final PathProperty<Long> LONG = PathProperty.create("long", Long.class);

Input<String> input = TEXT.render(Input.class); ①

Field<Long> field = LONG.render(Field.class); ②

ViewComponent<String> view = TEXT.render(ViewComponent.class); ③
```

① Render the property as `String` type `Input`

② Render the property as `Long` type `Field` (`HasValue` component)

③ Render the property as `String` type `ViewComponent`

10.1. Property localization

When a `LocalizationContext` is available as a *context resource*, it is used to **localize** any element to display which supports localization.

> See the LocalizationContext documentation for details about the `LocalizationContext` API and the Context resources documentation for information about the Holon Platform *Context* architecture.

Concerning to a `Property` related UI component, the `LocalizationContext` is used to:

- Localize the property **caption**, for example when it is used as a listing column header or as a input field caption
- Obtain the **date and time format** to use to display a property value or to edit the property value through an input field
- Obtain the **number format** to be used to display a property value or to edit the property value through an input field
- Obtain the default **boolean** localized text for the `true` and `false` values

> See the Internationalization documentation for further information about localization.

10.2. Custom property renderers

Relying on the standard property renderers features, custom renderers can be registered to:

- Customize the rendered UI component under certain conditions for the default provided rendering types (`Input`, `Field` and `ViewComponent`)
- Provide the rendering for a custom type

For example, to render a specific `String` type property as **text area**, instead of the default *text field*, an appropriate renderer can be registered and bound to a suitable property condition:

```
final PathProperty<String> TEXT = PathProperty.create("text", String.class);

InputPropertyRenderer<String> textAreaInputRenderer = p -> Components.input.string
(true).build(); ①

PropertyRendererRegistry.get().register(p -> p == TEXT, textAreaInputRenderer); ②

Input<String> input = TEXT.render(Input.class); ③
```

① Create a `Input` property renderer which renders the property as a *text area*

②

Register the renderer and bind it to a condition which checks the property is exactly the `TEXT` property

③ From now on, rendering the `TEXT` property as an `Input` field will generate a *text area* input component

11. Vaadin session scope

A **Vaadin session** platform scope is automatically registered by the Holon Vaadin module.

This `ContextScope` is bound to the current `VaadinSession`, looking up context resources using Vaadin Session attributes.

The scope uses `VaadinSession.getCurrent()` to obtain the current Vaadin session, so this scope is active only when invoked within a Vaadin server request thread.

> See the Context documentation for further information about context scopes and resources.

```
VaadinSession.getCurrent().setAttribute(LocalizationContext.CONTEXT_KEY,
    LocalizationContext.builder().withInitialLocale(Locale.US).build()); ①

LocalizationContext.getCurrent().ifPresent(localizationContext -> { ②
  // LocalizationContext obtained from current Vaadin session
});
```

① Create a `LocalizationContext` and set it in Vaadin session using the default context key as attribute name

② Get the current `LocalizationContext` context resource, which will be obtained form the Vaadin session, if available

12. Device information

The Holon platform Vaadin module makes available the DeviceInfo interface, to obtain informations about the **device** in which the application is running.

The `DeviceInfo` object provides informations about the *user agent* and screen dimensions, and can be obtained by using a `VaadinRequest` or relying on the currently available `VaadinSession`.

```
DeviceInfo.get().ifPresent(info -> {

  info.getScreenWidth();
  info.getScreenHeight();
  info.getViewPortWidth();
  info.getViewPortHeight();
  info.isMobile();
  info.isTablet();
  info.isSmartphone();
  info.isAndroid();
  info.isIPhone();
  // ...

});
```

13. Navigator

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin-navigator</artifactId>
<version>5.1.2</version>
```

The `holon-vaadin-navigator` artifact makes available an extension of the default Vaadin **view navigator**, represented by the ViewNavigator API.

The `ViewNavigator` API, compared to the standard Vaadin Navigator one, provides additional features and configuration capabilities:

- `View` navigation **history tracking**, allowing to navigate back to the previous view.
- `View` **parameters management**, allowing to obtain the parameters values **by injection** in a `View` instance.
- Easy `View` **lifecycle** management, supporting `@OnShow` and `@OnLeave` annotated methods to be notified when the view is loaded and unloaded in the UI.
- **Default view** support, acting as the application's "home page".
- The possibility to **show a view in a `Window`**, instead of using the default view display component.
- Support for Context resources injection in `View` instances.

13.1. View parameters injection

The *navigation parameters* can be injected in the `View` class fields using the ViewParameter annotation.

The parameter **name** for which the value has to be injected in an annotated field is assumed to be

equal to the *field name*. Otherwise, it can be specified using the annotation `value()` property.

When a named parameter value is not available (not provided by the current navigation state), the parameter field will be setted to `null` or the default value for primitive types (for example `0` for `int`, `long`, `float` and `double` types, false for `boolean` types).

The supported parameter value types are:

- `String`
- `Number` (including primitive types)
- `Boolean` (including primitive type)
- `Enum`
- `Date`
- `LocalDate`
- `LocalTime`
- `LocalDateTime`

A view parameter can be declared as **required** using the `required()` annotation property. A navigation error will be thrown if a required parameter value is missing.

Furthermore, a **default value** can be provided for a parameter, using the `defaultValue()` annotation property. See the ViewParameter javadocs for information about the default value representation.

```
class ViewExample implements View {

  @ViewParameter("myparam") ①
  private String stringParam;

  @ViewParameter(defaultValue = "1") ②
  private Integer intParam;

  @ViewParameter(required = true) ③
  private LocalDate requiredParam;

}
```

① View parameter injection using the `myparam` parameter name

② View parameter with a default value. The parameter name will be the `intParam` field name

③ A required view parameter declaration

13.2. View lifecycle management

In addition to the standard `enter(…)` method of the `View` interface, two annotations can be used to intercept `View` lifecycle events using `public` View methods:

- OnShow, called by the view navigator right before the view is shown (i.e. rendered in target

display component)

- OnLeave, called by the view navigator when the view is about to be deactivated (i.e. a navigation to another view was triggered)

If more than one `OnShow` or `OnLeave` annotated method is present in the `View` class or in its class hierarchy, all these methods will be called and the following behaviour will be adopted:

- Methods will be called following the class hierarchy, starting from the top (the first superclass after `Object`)
- For methods of the same class, no calling order is guaranteed

The `@OnShow` and `@OnLeave` annotated method supports an optional parameter, which can be either of the standard `com.vaadin.navigator.ViewChangeEvent` type or of the extended `ViewNavigatorChangeEvent` type, to obtain informations about the current view navigator, the previous or next `View`, the `View` name and parameters and the optional `Window` in which the View is displayed.

The `OnShow` annotation provides an additional `onRefresh()` property which, if setted to `true`, instruct the navigator to invoke the `OnShow` annotated method also at browser page refresh.

```
class ViewExample2 implements View {

  @ViewParameter
  private String myparam;

  @OnShow
  public void onShow() { ①
    // ...
  }

  @OnShow(onRefresh = true) ②
  public void onShowAtRefreshToo() {
    // ...
  }

  @OnShow
  public void onShow2(ViewChangeEvent event) { ③
    String name = event.getViewName(); ④
    View oldView = event.getOldView(); ⑤
    // ...
  }

  @OnShow
  public void onShow3(ViewNavigatorChangeEvent event) { ⑥
    ViewNavigator navigator = event.getViewNavigator(); ⑦
    Optional<Window> viewWindow = event.getWindow(); ⑧
    // ...
  }

  @OnLeave
  public void onLeave() { ⑨
    // ...
  }

  @OnLeave
  public void onLeave2(ViewNavigatorChangeEvent event) { ⑩
    View nextView = event.getNewView(); ⑪
    // ...
  }

}
```

① Basic `OnShow` annotated method, invoked right before the view is shown

② `OnShow` method with `onRefresh()` setted to `true`: this method will be invoked also at browser page refresh

③ `OnShow` method with a default `ViewChangeEvent` parameter

④ Get the View name

⑤ Get the previous View

⑥ `OnShow` method with a `ViewNavigatorChangeEvent` parameter

⑦ Get the `ViewNavigator`

⑧ Get the `Window` in which the view is displayed if it was requested to open the View in a window

⑨ Basic `onLeave` annotated method

⑩ `onLeave` method with a `ViewNavigatorChangeEvent` parameter

⑪ Get the View being activated

13.3. Providing the View contents

By default, the `View` is rendered as the UI component which is represented by the `View` class.

You can override the standard Vaadin 8 `getViewComponent()` method of the `View` class to control which `Component` to provide as `View` content.

```
class ViewExampleContent extends VerticalLayout implements View { ①

  public ViewExampleContent() {
    super();
    addComponent(new Label("View content"));
  }

}

class ViewExampleContentProvider implements View {

  @Override
  public Component getViewComponent() { ②
    boolean mobile = DeviceInfo.get().map(info -> info.isMobile()).orElse(false);
    return mobile ? buildMobileViewContent() : buildDefaultViewContent();
  }

}
```

① Default `View` implementation extending a `Component` class (a `VerticalLayout`). The displayed `View` content will be the component itself

② Provide the `View` content using the `getViewComponent()` method. In this example, a different UI component is provided according to the device type, checking if the user is running the application in a *mobile* device

13.4. ViewNavigator Configuration

Just like the standard Vaadin navigator, the `ViewNavigator` implementation requires two elements which must be configured in order to work properly:

- A `ViewProvider` to provide the `View` instances by view *name*

- A `View` display component to show the view contents in the UI, which can be a `ComponentContainer` (replacing the contents of the container with the active `View` content), a `SingleComponentContainer` (using the `setContent(…)` method to set the active `View` content) or a `ViewDisplay` object to implement a custom `View` display logic.

In addition to the required configuration elements, the `ViewNavigator` supports:

- Setting a **default view** name, which will be used as target of the `ViewNavigator.navigateToDefault()` method and as a fallback by the `ViewNavigator.navigateBack()` method if no other `View` is available in navigation history
- Setting a **error view** or a error `ViewProvider` to provide the `View` to be displayed when no other `View` matches a navigation state
- Setting the max **navigation history** size
- Register a `ViewChangeListener` for listening to `View` changes before and after they occur

The `builder()` method of the `ViewNavigator` interface provides a *fluent* builder to configure a navigator and bind it to a Vaadin `UI`.

```
class AppUI extends UI {

  @Override
  protected void init(VaadinRequest request) {
    ViewNavigator navigator = ViewNavigator.builder() ①
        .viewDisplay(this) ②
        .addProvider(getViewProvider()) ③
        .defaultViewName("home") ④
        .errorView(MY_ERROR_VIEW) ⑤
        .errorViewProvider(getErrorViewProvider()) ⑥
        .maxNavigationHistorySize(1000) ⑦
        .navigateToDefaultViewWhenViewNotAvailable(true) ⑧
        .withViewChangeListener(e -> { ⑨
          // ...
          return true;
        }).buildAndBind(this); ⑩
  }

}
```

① Obtain the `ViewNavigator` builder

② Set the component to which the `View` contents display is delegated

③ Add a `ViewProvider`

④ Set the *default* `View` name

⑤ Set the error `View`

⑥ Set the error `View` provider

⑦ Set the max navigation history size

⑧ Configure the navigator to navigate to *default* `View` (if availablr) when a view is not available according to current navigation state

⑨ Add a `ViewChangeListener`

⑩ Build the `ViewNavigator` and bind it to the `UI`

The `ViewNavigator` builder makes available a default `View` provider, which supports a fixed set of `Views` configured using a View *name* bound to a View *class*.

Stateful views are supported by this provider. To declare a View as stateful, the StatefulView annotation can be used on the View class.

View instances will be created according to view scope: for *stateful* views, an instance is created at first request (for each UI) and the same instance is returned to subsequent view requests. On the contrary, for standard views, a new instance is created and returned to navigator for every view request.

To register this view provider and configure the `View` set, the `ViewNavigator` builder method `withView(String viewName, Class<? extends View> viewClass)` can be used.

```
UI ui = getUI();
ViewNavigator.builder() //
    .viewDisplay(ui) //
    .withView("view1", View1.class) ①
    .withView("view2", View2.class) ②
    .defaultViewName("view1") ③
    .buildAndBind(ui);
```

① Register the view class `View1` and bind it the `view1` name

② Register the view class `View2` and bind it the `view2` name

③ Set the `view1` View as the default view

13.5. Excluding a View form navigation history

By default, the `ViewNavigator` tracks views navigation history, to made available navigation operations such `navigateBack()` to navigate to the previous view.

To exclude a `View` from the navigation history, skipping such View when a `navigateBack()` operation is performed or the user presses the browser *back* button, the VolatileView annotation can be used on the `View` class.

```
@VolatileView
class VolatileViewExample implements View {

}
```

13.6. Opening Views in a `Window`

The `ViewNavigator` supports displaying a `View` content in a application `Window`. The view navigation is tracked in navigator history just like any other navigation operation.

To open a `View` in a `Window`, the `navigateInWindow(…)` view navigator methods can be used (see the next section), or a `View` class can be forced to be always opened in a window using the WindowView annotation.

If the `@WindowView` annotation is found on a view class, such view will be always opened in a window, regardless of the navigation operation that is used.

The `Window` is automatically created by the `ViewNavigator`, and can be configured using either a `ViewWindowConfigurator`, when the navigator method is used, or the `@WindowView` annotation attributes.

```
@WindowView(windowWidth = "50%")
class WindowViewExample implements View {

}
```

13.7. ViewNavigation API operations

The `ViewNavigator` API makes available navigation methods to provide a set of **parameters** to the `View` being activated.

These parameters are serialized in the navigation state String, and are automatically bound to the any `View` class field annotated with the `@ViewParameter` annotation.

See View parameters injection for further informations about `View` parameters injection.

> When using `@ViewParameter` annotated `View` methods, the parameter value types provided to a view navigation method through the *name-value Map* must be consistent with the corresponding `@ViewParameter` view class field type.

The `ViewNavigator` API provides the following operations for `View` navigation:

- `navigateTo(String viewName, Map<String, Object> parameters)`: navigate to given `View` name, providing an optional Map of parameters with parameters names and corresponding values.
- `navigateTo(String viewName)`: navigate to given `View` name without providing any view parameter.

```
ViewNavigator navigator = getViewNavigator();

navigator.navigateTo("myView"); ①

Map<String, Object> parameters = new HashMap<>();
parameters.put("parameter1", "test");
parameters.put("parameter2", 34.5);

navigator.navigateTo("myView", parameters); ②
```

① Navigate to the view named `myView`

② Navigate to the view named `myView` providing a map of **parameters** (names and values)

- `navigateInWindow(…)`: `View` navigation methods that force the View content to be displayed in an `Window`, supporting a `ViewWindowConfiguration` Consumer function to setup the view `Window` features.

```
ViewNavigator navigator = getViewNavigator();

navigator.navigateInWindow("myView"); ①

Map<String, Object> parameters = new HashMap<>();
parameters.put("parameter1", "test");
parameters.put("parameter2", 34.5);

navigator.navigateInWindow("myView", windowConfig -> {
  windowConfig.fullWidth();
  windowConfig.styleName("my-window-style");
}, parameters); ②
```

① Navigate to the view named `myView` and display the view in a Window

② Navigate to the view named `myView` and display the view in a Window, providing a window configuration consumer and a map of **parameters**

- Using the `NavigationBuilder` through the `toView(String viewName)` method, which accepts the view name and the optional view parameters using a *fluent* builder style.

```
ViewNavigator navigator = getViewNavigator();

navigator.toView("myView").withParameter("parameter1", "test").withParameter(
"parameter2", 34.5).navigate(); ①

navigator.toView("myView").navigateInWindow(); ②

navigator.toView("myView").navigateInWindow(windowConfig -> {
  windowConfig.fullWidth();
  windowConfig.styleName("my-window-style");
}); ③
```

① Navigate to the view named `myView` using given parameters

② Navigate to the view named `myView` and display the view in a Window

③ Navigate to the view named `myView` and display the view in a Window, providing a window configuration consumer

- The `navigateToDefault()` method can be used to navigate to the default view, if configured.
- The `navigateBack()` method can be used to navigate back to previous `View`, if any. In no previous `View` is available and a default view is defined, navigator will navigate to the default view.

13.8. Sub views

The `ViewNavigator` supports the concept of **sub view**: a sub view is a `View` which is always displayed using a parent *view container*, and the sub view display strategy is delegated to the container.

A sub view container is a conventional `View` which implements the SubViewOf interface.

Any sub view must declare its parent container view name using the WindowView annotation.

When the navigation to a sub view is triggered, the `ViewNavigator` first of all will navigate to the declared sub view container, displaying it as a normal `View`, then the `display(View view, String viewName, Map<String, String> parameters)` method of the sub view container is invoked providing the sub view instance, the sub view name and any navigation parameter. At this point, the container should display the sub view contents in a suitable way.

The `getViewContent(View view)` static method of the `ViewNavigator` interface can be used to obtain the view contents from a `View` instance.

```
class SubViewContainerExample extends TabSheet implements SubViewContainer { ①

  public SubViewContainerExample() {
    super();
    setSizeFull();
  }

  @Override
  public boolean display(View view, String viewName, Map<String, String> parameters)
      throws ViewNavigationException {
    addTab(view.getViewComponent(), viewName, VaadinIcons.PIN);
    return true;
  }

  @Override
  public View getCurrentView() {
    return (View) getSelectedTab();
  }

}

@SubViewOf("mycontainer")
public class SubViewExample extends VerticalLayout implements View { ②

  public SubViewExample() {
    super();
    setMargin(true);
    addComponent(new Label("The sub view 1"));
  }

}
```

① A view declared as `SubViewContainer` using a `TabSheet` to display each sub view in a new tab. We suppose this view is bound to the `mycontainer` view name

② A sub view bound to the `mycontainer` sub view container using the `@SubViewOf` annotation

13.9. Context resources injection

The `ViewNavigator` supports **Context** resources injection into the `View` instances. A resource available form the Holon platform `Context` registry can be injected in a `View` using the ViewContext annotation on a `View` field.

The resource to be injected is looked up by *key*, and by the default the resource key is assumed to be the fully qualified class name of the injectable field **type**. To override this strategy, the `value()` annotation attribute of the `ViewContext` annotation can be used to provide the resource key to look up.

See the Context documentation for further information about context resources.

```
class ViewContextExample implements View {

  @ViewContext
  private LocalizationContext localizationContext;

  @ViewContext
  private AuthContext authContext;

}
```

13.10. Obtain the current ViewNavigator

The `ViewNavigator` interface provides methods to obtain the current navigator using the following strategy:

- If the `ViewNavigator` is available as *Context* resource using the default navigator resource key, that instance is returned;
- If a current Vaadin `UI` is available and a `ViewNavigator` is bound to the `UI`, that instance is returned.

See the Context documentation for further information about context resources.

```
Optional<ViewNavigator> navigator = ViewNavigator.getCurrent(); ①

ViewNavigator viewNavigator = ViewNavigator.require(); ②
```

① Try to obtain the `ViewNavigator` from context or current `UI`

② Obtain the `ViewNavigator` from context or current `UI`, failing with an exception if not found

13.11. Authentication support

The `ViewNavigator` architecture provides support for `View` authentication, relying on the default Holon platform `AuthContext` API.

See the Realm and AuthContext documentation for information about the Holon platform authentication and authorization architecture.

In order for the authentication to work, an `AuthContext` instance must be available as a *context* resource, and it will be used to perform user authentication and resource access control, relying on the `Realm` bound to the auth context.

The authentication support is enabled by default, but it can be configured using the `authenticationEnabled(…)` `ViewNavigator` builder method.

The **authentication** support is enabled through the standard

`com.holonplatform.auth.annotations.Authenticate` annotation, which can be used on a `View` class or on the application `UI` class.

When the `Authenticate` annotation is used at `UI` level, all the views managed by the navigator bound to such `UI` will be under authentication, and the access to any `View` will be denied if an authenticated subject is not available from the current `AuthContext`.

Each time the navigation to a protected `View` is requested, the `AuthContext` is checked, and if it not authenticated, the following strategy is implemented:

1. An *implicit* authentication attempt is performed, using the current `VaadinRequest` and the optional authentication schemes which can be specified using the `schemes()` attribute of the `Authenticate` annotation. This behaviour can be used, for example, to support authentication using the current HTTP request and schemes such the `Authorization` HTTP header.
2. If the *implicit* authentication is not successfull and a valid **redirect URI** is provided through the `redirectURI()` property of the `Authenticate` annotation, the navigation is redirected to that URI. If the redirect URI does not specify a scheme, or the scheme is equal to the special `view://` scheme, the navigation is redirected to the navigation state specified by the redirect URI (escluding the `view://` part, if present). This way, the redirect URI can be used to delegate to a `View` an *explicit* authentication entry point (for example using conventional username and password credentials).

14. Spring integration

The `holon-vaadin-spring` artifact provides support and integration with the Spring framework.

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin-spring</artifactId>
<version>5.1.2</version>
```

This artifact provides a `ViewNavigator` extension with Spring support, represented by the SpringViewNavigator API.

The `SpringViewNavigator` implementation relies upon the standard Vaadin Spring integration add-on, and supports all its functionalities and configuration features.

See the Vaadin Spring tutorial for the documentation.

The following annotations are available for `View` configuration:

- DefaultView: can be used on a `View` class to declare it as the **default** view, i.e the view which will be used as target of the `ViewNavigator.navigateToDefault()` method and as a fallback by the `ViewNavigator.navigateBack()` method if no other `View` is available in navigation history.
- ErrorView: can be used on a `View` class to declare it as the default **error** view, i.e. the `View` to be displayed when no other `View` matches a navigation state.

14.1. Default View navigation strategy

When a statically declared *default* `View` (using for example the `@DefaultView` annotation) is not suitable for an application, the DefaultViewNavigationStrategy interface can be used to implement a custom strategy to provide the default navigation state.

When a `DefaultViewNavigationStrategy` type Spring bean is found in application context, it is automatically configured in View navigator as default view navigation stategy implementation.

14.2. Spring view navigator configuration

The `SpringViewNavigator` API provides a *builder* to create a navigator instance, and can be used to explicitly build and configure a `SpringViewNavigator` instance. The bulder can be obtained through the static `builder()` method of the `SpringViewNavigator` interface.

The easiest way to setup a Spring view navigator, is to use the provided EnableViewNavigator configuration annotation.

The `@EnableViewNavigator` can be used on Spring configuration classes to automatically setup the default Vaadin Spring integration and register a UI-scoped `SpringViewNavigator` bean. The standard `@SpringViewDisplay` annotation can be used to configure the views display component and the default Vaadin Spring `ViewProvider` will be used.

> The `@EnableViewNavigator` annotation includes the standard `com.vaadin.spring.annotation.@EnableVaadin` annotation behaviour, which is not required anymore on configuration classes.

The `@EnableViewNavigator` annotation makes available a number of properties to control the navigator configuration, for example to explicitly configure the default and error views or to set the max navigation history size. See the EnableViewNavigator javadocs for further information.

```
@Configuration ①
@ComponentScan(basePackageClasses = ViewOne.class) ②
@EnableViewNavigator ③
@EnableBeanContext ④
class SpringConfig {

}

@SpringView(name = "view1") ⑤
@DefaultView ⑥
class ViewOne extends VerticalLayout implements View {

}

@SpringView(name = "view2") ⑦
@UIScope ⑧
class ViewTwo extends VerticalLayout implements View {

}

@SpringUI ⑨
@SpringViewDisplay ⑩
class AppUI extends UI {

  @Autowired
  ViewNavigator navigator; ⑪

  @Override
  protected void init(VaadinRequest request) {
    // ...
  }

}
```

① Declare the class as a Spring configuration class

② Set the *component scan* rule to auto detect the `View` beans

③ Enable the Spring `ViewNavigator`

④ Enable the Holon platform Spring *context* scope, to provide context resource instances as Spring beans

⑤ Declare the view as Spring view (which will be automatically registered in the navigator view provider), and bind it to the `view1` name

⑥ Declare the view as the default view

⑦ Create another view and enable it as a Spring view using the `view2` name

⑧ Declare the view bean scope as `UI`

⑨ Create the application `UI` and declare it as a Spring `UI`, which will be automatically detected and configured by Spring

⑩ Use the `UI` as `View` display container

⑪ The `ViewNavigator` will be made avaialable as Spring (UI-scoped) bean, so it can be obtained using dependency injection

14.3. View context resources

The EnableViewContext annotation can be used on Spring configuration classes to enable `View` context resource injection using the ViewContext annotation.

See Context resources injection for further information.

14.4. View authorization support

In addition to the `ViewNavigator` authentication support (see Authentication support), the Spring view navigator provides View **authorization** support using default `javax.annotation.security.*` annotations (`@RolesAllowed`, `@PermitAll`, `@DenyAll`).

The authorization support can be enabled using the EnableViewAuthorization annotation and, just like the authentication support, relies on the current `AuthContext` to perform authorization control, so it must be available as a *context* resource.

By using the `@EnableBeanContext` configuration annotation, Spring beans can be automatically configured as *context* resources. See the SpringContextScope documentation for further information.

The default Vaadin Spring `ViewAccessControl` and `ViewInstanceAccessControl` view access control methods are fully supported too, and can be used along with the security annotations.

The AccessDeniedView annotation can be used on a Spring `View` class to declare it as the view to show when the user is not authorized to access a view, either according to a `javax.annotation.security.*` annotation or to a `ViewAccessControl` or `ViewInstanceAccessControl` rule.

```
@Configuration
@ComponentScan(basePackageClasses = ViewOne.class)
@EnableViewNavigator
@EnableBeanContext ①
@EnableViewAuthorization ②
class SpringConfig {

  @Bean ③
  @VaadinSessionScope
  public AuthContext authContext() {
    AccountProvider ap = id -> {
      // Only a user with username 'username1' is available
      if ("username1".equals(id)) {
```

```
        // setup the user password and assign the role 'role1'
        return Optional.of(Account.builder(id).credentials(Credentials.builder()
.secret("s3cr3t").build())
            .permission("role1").build());
      }
      return Optional.empty();
    };
    return AuthContext.create(Realm.builder()
        // authenticator using the AccountProvider
        .authenticator(Account.authenticator(ap))
        // default authorizer
        .withDefaultAuthorizer().build());
  }

}

@SpringView(name = "view1")
@PermitAll ④
class ViewOne extends VerticalLayout implements View {

}

@SpringView(name = "view2")
@RolesAllowed("role1") ⑤
class ViewTwo extends VerticalLayout implements View {

}

@AccessDeniedView ⑥
@UIScope
@SpringView(name = "forbidden")
class AccessDenied extends VerticalLayout implements View {

  private static final long serialVersionUID = 1L;

  private final Label message;

  public AccessDenied() {
    super();
    Components.configure(this).margin()
        .add(message = Components.label().styleName(ValoTheme.LABEL_FAILURE).build());
  }

  @Override
  public void enter(ViewChangeEvent event) {
    message.setValue("Access denied [" + event.getViewName() + "]");
  }

}
```

① Use `@EnableBeanContext` to enable Spring beans as context resources (in this example, the

`AuthContext` bean will be available as context resource)

② Enable views authorization using `javax.annotation.security.*` annotations

③ Configure the `AuthContext` and declare it as a session-scoped Spring bean

④ Use `@PermitAll` on this view to skip authorization control

⑤ Use `@RolesAllowed` to declare that the view is available only for the authenticated subjects with the `role1` role

⑥ Create a custom *access denied* view using the `@AccessDeniedView` annotation

14.5. Spring Security support

The `View` access control using the Spring Security `@Secured` annotation can be enabled using the EnableSecuredView annotation.

This access control strategy relies on the Spring Security `SecurityContext` to check current authentication and performs View access control checking current user granted authorities against the security attributes specified through the `@Secured` annotation.

The Spring Security artifacts must be explicitly included as a project dependency.

15. Spring Boot integration

The `holon-vaadin-spring-boot` artifact provides integration with Spring Boot for Vaadin application and view navigator auto configuration.

To enable Spring Boot auto-configuration the following artifact must be included in your project dependencies:

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin-spring-boot</artifactId>
<version>5.1.2</version>
```

The Spring Boot auto-configuration includes the default Spring Boot Vaadin add-on auto configuration features, with the following additional behaviour:

- The configured view navigator will be a Spring `ViewNavigator`.
- The `View` authorization support using the `javax.annotation.security.*` annotations is enabled by default.
- If Spring Security is available and configured, the `@Secured` annotation based `View` access control is enabled.

To disable this auto-configuration feature the `HolonVaadinAutoConfiguration` class can be excluded:

```
@EnableAutoConfiguration(exclude={HolonVaadinAutoConfiguration.class})
```

To disable the Spring Security `@Secured` access control support the `HolonVaadinSpringSecurityAutoConfiguration` class can be excluded:

```
@EnableAutoConfiguration(exclude={HolonVaadinSpringSecurityAutoConfiguration.class})
```

15.1. Vaadin servlet configuration

By default, a Holon extension of the Vaadin servlet is auto-configured.

This servlet allows automatic `com.vaadin.server.SessionInitListener` and `com.vaadin.server.SessionDestroyListener` registration if they are defined as Spring beans and made available in the Spring context.

To disable this auto-configuration feature, the `HolonVaadinServletAutoConfiguration` class can be excluded:

```
@EnableAutoConfiguration(exclude={HolonVaadinServletAutoConfiguration.class})
```

15.2. Spring Boot starters

The following *starter* artifacts are available to provide a quick project configuration setup using Maven dependency system:

1. The **Vaadin application starter** provides the dependencies to the Holon Platform Vaadin Spring Boot integration artifact `holon-vaadin-spring-boot`, in addition to:

- The Holon Platform Core Module Spring Boot integration base starter (`holon-starter`).
- The Spring Boot `spring-boot-starter-web` starter.
- The Spring Boot `spring-boot-starter-tomcat` to use Tomcat as the embedded servlet container.

See the Spring Boot starters documentation for details on Spring Boot *starters*.

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-starter-vaadin</artifactId>
<version>5.1.2</version>
```

2. The **Vaadin application starter using Undertow** provides the same dependencies as the default Vaadin application starter, but using *Undertow* instead of Tomcat as embedded servlet container.

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-starter-vaadin-undertow</artifactId>
<version>5.1.2</version>
```

16. Loggers

By default, the Holon platform uses the SLF4J API for logging. The use of SLF4J is optional: it is enabled when the presence of SLF4J is detected in the classpath. Otherwise, logging will fall back to JUL (`java.util.logging`).

The logger name for the **Vaadin** module is `com.holonplatform.vaadin`.

17. System requirements

17.1. Java

The Holon Platform Vaadin module requires Java **8** or higher.

17.2. Vaadin

The Holon Platform Vaadin module requires Vaadin **8.1** or higher.