Holon Platform Vaadin 7 Module - Reference manual

Version 5.1.0

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. Vaadin integration . . . 2
4. Component builders . . . 2
5. UI Components . . . 3
   5.1. Input . . . 3
   5.2. ValidatableInput . . . 4
   5.3. Selectable Input . . . 6
      5.3.1. Rendering mode . . . 7
      5.3.2. Selection items data source . . . 8
      5.3.3. Item captions and icons . . . 8
      5.3.4. SingleSelect caption filter . . . 9
      5.3.5. Using the `Property` model with selectable Inputs . . . 9
   5.4. Grouping Inputs . . . 11
      5.4.1. Read-only properties . . . 13
      5.4.2. Hidden properties . . . 13
      5.4.3. Default property values . . . 13
   5.5. Input group validation . . . 13
   5.6. Input Forms . . . 15
   5.7. View components . . . 17
      5.7.1. View component Groups and Forms . . . 17
6. Item listing . . . 18
   6.1. BeanListing . . . 19
   6.2. PropertyListing . . . 19
   6.3. Items data source . . . 20
   6.4. Columns configuration . . . 21
   6.5. Column rendering . . . 22
   6.6. Listing configuration . . . 23
   6.7. Item selection . . . 24
   6.8. Item query configuration . . . 26
   6.9. Item set management . . . 26
      6.9.1. Buffered mode . . . 27
   6.10. Editing items . . . 28
7. Dialogs . . . 29
   7.1. Question dialogs . . . 30
8. Property renderers and presenters . . . 30

8.1. Property localization . . . 31
8.2. Custom property renderers . . . 32
9. Vaadin session scope . . . 32
10. Device information . . . 33
11. Navigator . . . 33
11.1. View configuration . . . 34
11.1.1. Parameters injection . . . 34
11.1.2. View lifecycle hooks . . . 35
11.1.3. Providing the View contents . . . 37
11.2. Navigator Configuration . . . 38
11.3. Excluding a View form navigation history . . . 40
11.4. Opening Views in a `Window` . . . 40
11.5. View navigation operations . . . 41
11.6. Sub views . . . 43
11.7. Context resources injection . . . 44
11.8. Obtain the ViewNavigator . . . 45
11.9. Authentication support . . . 45
12. Spring integration . . . 46
12.1. Spring view navigator configuration . . . 47
12.2. View context resources . . . 49
12.3. View authorization support . . . 49
13. Spring Boot integration . . . 51
13.1. Spring Boot starters . . . 52
14. Loggers . . . 52
15. System requirements . . . 52
15.1. Java . . . 52
15.2. Vaadin . . . 53

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** 7 module represents the platform support for the Vaadin **7.7.x** web applications framework.

This module makes available an API which matches at 99% the official Holon Vaadin module API (which requires Vaadin version **8.1.x**), only not compatible classes or methods are different for obvious reasons.

The main differences between the two modules API are:

- The Vaadin 7 module supports the `com.vaadin.ui.Field` interface for `Input` components and `Property` renderers, while the Vaadin 8 module relies on the `HasValue` interface
- The Vaadin 7 `Table` component is available as `ItemListing` backing component, in addition to the `Grid` component
- The `com.vaadin.data.provider.DataProvider` interface is not available in Vaadin 7, so it is not supported as item components data source

See the official Holon Vaadin module documentation to use Vaadin version **8** instead.

1.1. Sources and contributions

The Holon Platform **Vaadin** 7 module source code is available from the GitHub repository https://github.com/holon-platform/holon-vaadin7.

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.vaadin7</groupId>
<artifactId>holon-vaadin-bom</artifactId>
<version>5.1.0</version>
```

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

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.vaadin7</groupId>
      <artifactId>holon-vaadin-bom</artifactId>
      <version>5.1.0</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. Vaadin integration

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

Maven coordinates:

```
<groupId>com.holon-platform.vaadin7</groupId>
<artifactId>holon-vaadin</artifactId>
<version>5.1.0</version>
```

4. Component builders

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

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

All the builders can be obtained using the Components interface, which is organized in sub-interfaces by component kind.

Table 1. Components builders structure

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, 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.

5. UI Components

The Vaadin module provides a set of components which extend or decorates the standard components set. The available components are described below.

5.1. Input

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

Input components supports the *read-only* (to prevent the user from changing the value) and *required* (to show a suitable indicator along with the field) modes.

Input components supports `ValueChangeListener` registration to listen to input value changes.

```
Input<String> stringInput = Components.input.string().caption("String input")
.inputPrompt("Value here")
    .maxLength(100).styleName("mystyle").build(); ①

stringInput.setValue("test"); ②
String value = stringInput.getValue(); ③

stringInput.addValueChangeListener(e -> System.out.println("Value: " + e.getValue()));
④
```

① Create and configure a `String` value type `Input` field.

② Set a value

③ Get the value

④ Add a `ValueChangeListener`

The concrete Vaadin `Component`, which represents the UI input element, can be obtained through the `getComponent()` method.

```
Input<LocalDate> dateInput = Components.input.localDate().caption("Date input")
.lenient(true).build(); ①

Component inputComponent = dateInput.getComponent(); ②
```

① Create and configure a `LocalDate` value type `Input` field.

② Get the input `Component`, which can be used to display the input field in UI.

An `Input` component can be obtained from a standard Vaadin `Field` component using the `from(…)` static method.

```
Input<String> stringInput = Input.from(new TextField());
```

Another way to obtain a standard Vaadin `Field` component from an input component is to use the `asField()` input builder method.

```
Field<Locale> localeField = Components.input.singleSelect(Locale.class).items(Locale
.US, Locale.CANADA)
    .caption("Select Locale").fullWidth().withValue(Locale.US).asField(); ①
```

① Create select field using `Locale` data type and obtain it as `Field` component.

5.2. ValidatableInput

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

Value validation can be triggered each time the input value changes, using `setValidateOnValueChange` method.

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

By default, the standard Vaadin component error notification is used.

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

validatableInput.addValidator(Validator.email()); ②
validatableInput.addValidator(Validator.max(100)); ③

validatableInput.setValidationStatusHandler(e -> { ④
  if (e.isInvalid()) {
    Notification.show(e.getErrorMessage(), Type.ERROR_MESSAGE);
  }
});

validatableInput.validate(); ⑤

validatableInput.setValidateOnValueChange(true); ⑥
```

① Create a `ValidatableInput` from a String input

② Add a validator to check the input value is a valid e-mail address

③ Add a validator to check the input value is maximum 100 characters long

④ Set validation status handler which shows a notification when the input in invalid

⑤ Perform the input value validation

⑥ Triggers validation when input value changes

A `ValidatableInput` can be obtained from an existing `Input` or `Field` component instance or using the appropriate builder.

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

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

validatableInput = Components.input.string().validatable() ③
    .required("Value is required") ④
    .withValidator(Validator.max(100)).build(); ⑤
```

① Create a `ValidatableInput` from a String `Input`

② Create a `ValidatableInput` from a String type `Field` component

③ Create a `ValidatableInput` using the input builder

④ Add a default *required* (the input is not empty) validator using the provided error message

⑤ Add a standard validator

5.3. 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 interface, which provides methods to check if some item is selected, obtain the selected item/s and change the current selection. Furthermore, a `SelectionListener` can be registered to listen to selection changes.

SingleSelect example

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

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)
.caption("Multi select")
    .build(); ①

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

5.3.1. 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`

Rendering mode	SingleSelect UI component	MultiSelect UI component
OPTIONS	`com.vaadin.ui.OptionGroup`	`com.vaadin.ui.OptionGroup`

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

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

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

5.3.2. 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` data provider;

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

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

① Create a select input using an explicitly provided items set

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

5.3.3. Item captions and icons

The *selectable* input builders allow to explicitly set the selection item **captions** and **icons**. As for the items caption, the Holon platform **localization** architecture is supported, and the builders provides methods to specify the caption using a `Localizable` or a *message code*.

> In order for the localization to work, a `LocalizationContext` must be available as a context resource. See the Internationalization documentation for further information.

```
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, FontAwesome.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 ? FontAwesome.STAR : FontAwesome.STAR_O) ②
    .build();
```

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

② Set an item icon generator

5.3.4. SingleSelect caption filter

For `SingleSelect` inputs using **SELECT** rendering mode, the UI component allows the user to type a *filtering* text to search for a selection item, which is referred to item captions.

The `FilteringMode` can be changed using the `filteringMode(…)` builder method.

```
SingleSelect<TestData> singleSelect = Components.input.singleSelect(TestData.class)
    .items(new TestData(1), new TestData(2)) // set the items
    .filteringMode(FilteringMode.CONTAINS) ①
    .build();
```

① Set the caption filtering mode

5.3.5. Using the `Property` model with selectable Inputs

The Holon platform `Property` model is fully supported by the selectable input builder, which allow 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`.

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

```
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 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)) //
    .withQueryConfigurationProvider(() -> ID.gt(0L)) ①
    .itemCaptionGenerator(propertyBox -> propertyBox.getValue(DESCRIPTION)).build();
```

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

5.4. Grouping Inputs

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.

The property values are setted and obtained using the `PropertyBox` data container, with 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.

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

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

PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) ①
    .bind(ID, Components.input.number(Long.class).build()) ②
    .bind(DESCRIPTION, Components.input.string().maxLength(100).build()) ③
    .build();

group.setValue(PropertyBox.builder(PROPERTIES).set(ID, 1L).set(DESCRIPTION,
"TestDescription").build()); ④

PropertyBox value = group.getValue(); ⑤

group.addValueChangeListener(e -> { ⑥
  PropertyBox changedValue = e.getValue();
});
```

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

② Bind the `ID` property to a `Long` type Input field

③ Bind the `DESCRIPTION` property to a `String` type Input field

④ Set the property values using a `PropertyBox` instance

⑤ Get the property values as a `PropertyBox` instance

⑥ Add a group `ValueChangeListener`

The default Holon platform `PropertyRenderer` architecture can be used to automatically generate a suitable Input component for a `Property`, according to its type.

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

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

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

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

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

① The property `Input` fields are automatically generated using the available `PropertyRenderer` from

the default registry

5.4.1. 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.

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

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

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

5.4.2. 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.

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

5.4.3. 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.

5.5. Input group validation

A `PropertyInputGroup` support both property value and group value validation, using both standard Holon platform `Validator` and Vaadin validators.

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 alog with the property input component.

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

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

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

PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .withValidator(DESCRIPTION, Validator.max(100)) ①
    .required(ID) ②
    .required(ID, "The ID value is required") ③
    .withValidator(Validator.create(propertyBox -> propertyBox.getValue(ID) > 0,
        "The ID value must be greater than 0")) ④
    .build();
```

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

② Add a *required* validator to the `ID` property

③ Add a *required* validator to the `ID` property providing a custom validation error message

④ Add an overall validator which read the `ID` property value from the group `PropertyBox` and checks it is greater than 0

The input group validation is performed invoking all the available property validators, and then the overall group validators, if any.

```
PropertyInputGroup group = createInputGroup();

group.validate(); ①

PropertyBox value = group.getValue(); ②

value = group.getValue(false); ③

value = group.getValueIfValid().orElse(null); ④
```

① Explicit group validation: if some 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

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

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

Furthermore, the input group builder provides two methods to control if the validation failures are to be collected and all notified to the user or if the validation has to stop at first failure, both from a properties or a group point of view.

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

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

PropertyInputGroup group = Components.input.propertyGroup().properties(PROPERTIES) //
    .stopValidationAtFirstFailure(true) ①
    .stopOverallValidationAtFirstFailure(true) ②
    .validationStatusHandler(validationEvent -> { ③
      // ...
    }).propertiesValidationStatusHandler(validationEvent -> { ④
      // ...
    }).build();
```

① Set to stop overall validation at first failure

② Set to stop properties validation at first failure

③ Set the overall `ValidationStatusHandler`

④ Set the properties `ValidationStatusHandler`

> A property input group `ValidationStatusHandler` setter method which accept a `Label` is provided as a shorter to use a standard Vaadin `Label` to notify validation errors.

5.6. Input Forms

The PropertyInputForm represents a `Input` group bound to a `Property` model which is also a UI `Component`, and can be used to display and manage the `Input` components in the UI.

It is a Grouping Inputs, from which inherits all the `Input` generation, configuration and management features, using a `PropertyBox` to set and get the property values bound to each Input component.

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

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

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

The inputs composition is triggered using the `compose()` method. The `PropertyInputForm` builder provides a `composeOnAttach(…)` method to set whether to automatically trigger the components composition when the form is attached to a parent layout, and this is the standard behaviour.

Furthermore, a `initializer(…)` builder method is made available to perform custom configuration of the base layout component.

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

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

PropertyInputForm form = Components.input.form(new FormLayout()).properties(
PROPERTIES).required(ID).build(); ①

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

form.setValue(PropertyBox.builder(PROPERTIES).set(ID, 1L).set(DESCRIPTION, "Test")
.build()); ③

PropertyBox value = form.getValue(); ④
```

① Create a `PropertyInputForm` with given property set using a FormLayout as base layout, and using the default composer

② Set a custom `Composer` to compose the Input components on the form layout

③ Set the form value using a `PropertyBox`

④ Get the form value

A set of default builder factory methods are provided to use the most common Vaadin layout components as form base content.

```
PropertyInputForm form = Components.input.form().properties(PROPERTY_SET).build(); ①
form = Components.input.formVertical().properties(PROPERTY_SET).build(); ②
form = Components.input.formHorizontal().properties(PROPERTY_SET).build(); ③
form = Components.input.formGrid().properties(PROPERTY_SET)
    .initializer(gridLayout -> gridLayout.setSpacing(true)).build(); ④
```

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

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

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

④ Create a `PropertyInputForm` using a `GridLayout` as base content and setting an *initializer* function to perform GridLayout configuration

5.7. View components

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

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

The actual UI component to display is obtained form the `getComponent()` method.

A Vaadin `Label` is used for the default `ViewComponent` implementation, and a default view component can be built using the default builder which can be obtained from the `builder(Class valueType)` method or from the `Components.view` interface.

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

view.setValue("TestValue"); ②
String value = view.getValue(); ③
```

① Create and configure a `ViewComponent` with a `String` value type

② Set the value

③ Get the value

5.7.1. View component Groups and Forms

Just like the `Input` components, the `ViewComponent` s can be organized in *groups* (using the PropertyViewGroup) or in *forms* (using the PropertyViewForm). Unlike the *group*, the *form* is a UI component which can be displayed in the user interface, composing the view components on a base layout.

These component containers rely on the `Property` model to bind each view component to a `Property` value and they use a `PropertyBox` to provide the property values to show.

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

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 = Components.view.form(new FormLayout()).properties(PROPERTIES) //
    .composer((layout, source) -> { ③
      source.getValueComponents().forEach(c -> layout.addComponent(c.getComponent()));
    }).build();

viewForm.setValue(PropertyBox.builder(PROPERTIES).set(ID, 1L).set(DESCRIPTION, "Test"
).build()); ④

PropertyBox value = viewForm.getValue(); ⑤
```

① Create a `PropertyViewGroup`

② Create a `PropertyViewForm` using a `VerticalLayout` as base layout

③ Create a `PropertyViewForm` using a `FormLayout` as base layout and providing a custom components composer

④ Set the form value using a `PropertyBox`

⑤ Get the form value

6. Item listing

The ItemListing component can be used to display a set of items as tabular data. By default, it is backed by a Vaadin `Grid` as concrete UI component, but the Vaadin `Table` component is supported too.

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` interface 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` sub-interfaces are available to use this component:

- BeanListing: Uses a Java Bean as item type and the listing column ids are the bean property names.

- PropertyListing: Uses a `PropertyBox` as item type and the listing column ids are the `Property` of the provided property set.

6.1. BeanListing

A BeanListing can be obtained providing the Java Bean class to use as item type.

The listing columns will be generated inspecting the **bean class** to detect the available bean properties, and the column ids will be the bean property names.

```
private class TestData {

  private Long id;
  private String description;

  // getters and setters omitted

}

public void beanListing() {
  BeanListing<TestData> listing = Components.listing.items(TestData.class) ①
      .header("id", "The ID") ②
      .header("description", "The description") ③
      .build();

  listing.refresh(); ④

  listing = Components.listing.itemsUsingTable(TestData.class).build(); ⑤
}
```

① Create a `BeanListing` using the `TestData` bean class

② Set the column header for the `id` item (bean) property

③ Set the column header for the `description` item (bean) property

④ Refresh the items set

⑤ Create a `BeanListing` using a `Table` as backing component

6.2. PropertyListing

A PropertyListing can be obtained providing the `Property` set to use for the listing items.

The listing columns will be generated according to the item property set `Property` types and each column will be identified by the `Property` itself.

For each property:

- 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;

- 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 components in the cells.

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

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

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

listing = Components.listing.propertiesUsingTable(PROPERTIES).build(); ②
```

① Create a `PropertyListing` using given `Property` set

② Create a `PropertyListing` using a `Table` as backing component

6.3. Items data source

To build an item listing, the **data source** of the items must be provided using the ItemDataProvider interface.

```
ItemDataProvider<PropertyBox> dataProvider = getDataProvider();
PropertyListing listing = Components.listing.properties(PROPERTIES) //
    .dataSource(dataProvider) ①
    .build();
```

① Create a `PropertyListing` and provide an `ItemDataProvider` as data source

Furthermore, an ItemIdentifierProvider should be configured to provide the item **identifiers**, expecially for the item selection consistency. The default behaviour is to use item itself as its own identifier, which it is not always the right semantic to use, expecially for `PropertyBox` type objects.

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

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

listing = Components.listing.properties(PROPERTIES) //
    .dataSource(dataProvider, 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 id

② A shorter builder method to use the `ID` property as item identifier

The item listing builders make available convenience methods to use a `Datastore` as data source, providing the `DataTarget` to use as target of the query on the backend data (for example, the name of a RDBMS table).

```
Datastore datastore = getDatastore();

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

① Create a `PropertyListing` and use a `Datastore` to build a data source which performs a query using the `test` data target (which can be, for example, the name of a RDBMS table), the property set of the listing as query projection and the `ID` property as item identifier

6.4. Columns configuration

The listing component columns can be configured in a number of ways:

```
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

6.5. Column rendering

The default column rendering can be overridden using a custom vaadin `Renderer`. The `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, new Converter<String, Long>() { ②

      @Override
      public Long convertToModel(String value, Class<? extends Long> targetType,
Locale locale)
          throws ConversionException {
        return Long.valueOf(value);
      }

      @Override
      public String convertToPresentation(Long value, Class<? extends String>
targetType, Locale locale)
          throws ConversionException {
        return (value == null) ? null : String.valueOf(value);
      }

      @Override
      public Class<Long> getModelType() {
        return Long.class;
      }

      @Override
      public Class<String> getPresentationType() {
        return String.class;
      }
    }, new TextRenderer()).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

6.6. 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

6.7. Item selection

The item listing component supports items selection, both in single and multiple mode. The listing can be made selectable 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

6.8. Item 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 always to 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`

6.9. 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 item 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

6.9.1. 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

6.10. 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 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
    .editorSaveCaption("Save item") ②
    .editorCancelCaption("Discard") ③
    .editable(ID, false) ④
    .editor(DESCRIPTION, new TextField()) ⑤
    .withValidator(DESCRIPTION, Validator.max(100)) ⑥
    .required(ID) ⑦
    .build();
```

① Set the listing as editable

② Set the editor *save* button caption

③ Set the editor *cancel* button caption

④ Set a property as not editable

⑤ Set a custom editor field

⑥ Add a property value validator

⑦ Set the `ID` property as required, automatically adding a *not empty* validator

7. 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(FontAwesome.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

7.1. 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

8. 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 `com.vaadin.ui.Field` property renderer
- 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`

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

8.1. Property localization

When a `LocalizationContext` is available as a Context resource, it is used to *localize* any element to display which supports localization. Concerning to a `Property` related UI component, it 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 a a input field
- Obtain the **number format** to use to display a property value or to edit the property value through a a input field
- Obtain the default **boolean** localized text for the `true` and `false` values

See the Internationalization documentation for further information about localization.

8.2. 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

InputFieldPropertyRenderer<String> fieldRenderer = p -> new TextArea(); ④
```

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

④ Example of a `Input` property renderer which renders the property as a *text area* using a Vaadin `Field`

9. 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

10. 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 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();
  // ...

});
```

11. Navigator

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

Maven coordinates:

```
<groupId>com.holon-platform.vaadin7</groupId>
<artifactId>holon-vaadin-navigator</artifactId>
<version>5.1.0</version>
```

11.1. View configuration

11.1.1. 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` s (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;

  @Override
  public void enter(ViewChangeEvent event) {
  }

}
```

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

11.1.2. View lifecycle hooks

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 in present in the `View` class or in it's 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.

```java
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
    // ...
  }

  @Override
  public void enter(ViewChangeEvent event) {
  }

}
```

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

11.1.3. Providing the View contents

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

The ViewContentProvider interface can be implemented by a `View` class to control which UI content to provide through the `getViewContent()` method. The method is called by the `ViewNavigator` to display the `View` when it is activated.

```
class ViewExampleContent extends VerticalLayout implements View { ①

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

  @Override
  public void enter(ViewChangeEvent event) {
  }

}

class ViewExampleContentProvider implements View, ViewContentProvider { ②

  @Override
  public Component getViewContent() { ③
    boolean mobile = DeviceInfo.get().map(info -> info.isMobile()).orElse(false);
    return mobile ? buildMobileViewContent() : buildDefaultViewContent();
  }

  @Override
  public void enter(ViewChangeEvent event) {
  }

}
```

① Default `View` implementation extending a `Component` class (a `VerticalLayout`). The displayed `View`

content will be the component itself

② A `View` implementing the `ViewContentProvider` interface: the displayed UI content will be provided by the `getViewContent()` method

③ Provide the `View` content. 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

11.2. Navigator Configuration

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

- A `ViewProvider` to provide th `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`.

```java
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(new ViewChangeListener() { ⑨

          @Override
          public boolean beforeViewChange(ViewChangeEvent event) {
            // ...
            return true;
          }

          @Override
          public void afterViewChange(ViewChangeEvent event) {
            // ...

          }
        }).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

11.3. Excluding a View form navigation history

By default, the `ViewNavigator` tacks 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 {

  @Override
  public void enter(ViewChangeEvent event) {
  }

}
```

11.4. 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 {

  @Override
  public void enter(ViewChangeEvent event) {
  }

}
```

11.5. View navigation operations

The `ViewNavigator` API provides 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 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.

11.6. 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`, than 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.

```java
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(ViewNavigator.getViewContent(view), viewName, FontAwesome.PUZZLE_PIECE);
    return true;
  }

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

  @Override
  public void enter(ViewChangeEvent event) {
  }

}

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

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

  @Override
  public void enter(ViewChangeEvent event) {
  }

}
```

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

11.7. 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;

  @Override
  public void enter(ViewChangeEvent event) {
  }

}
```

11.8. Obtain the 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

11.9. 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).

12. Spring integration

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

Maven coordinates:

```
<groupId>com.holon-platform.vaadin7</groupId>
<artifactId>holon-vaadin-spring</artifactId>
<version>5.1.0</version>
```

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

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.

12.1. 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 registering 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 {

  @Override
  public void enter(ViewChangeEvent event) {
  }

}

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

  @Override
  public void enter(ViewChangeEvent event) {
  }

}

@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

12.2. 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.

12.3. 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 {

  @Override
  public void enter(ViewChangeEvent event) {
  }

}

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

  @Override
  public void enter(ViewChangeEvent event) {
  }

}

@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

13. 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.vaadin7</groupId>
<artifactId>holon-vaadin-spring-boot</artifactId>
<version>5.1.0</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

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

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

13.1. Spring Boot starters

The following *starter* artifacts are available to provide a quick project configuration setup using Maven dependency system:

1. The **Holon Vaadin application starter** provides the dependencies to the Holon Vaadin Spring Boot integration artifact, in addition to default Holon *core* Spring Boot starters, the default `spring-boot-starter-web` starter and the `spring-boot-starter-tomcat` to use Tomcat as the embedded servlet container:

Maven coordinates:

```
<groupId>com.holon-platform.vaadin7</groupId>
<artifactId>holon-starter-vaadin</artifactId>
<version>5.1.0</version>
```

2. The **Holon Vaadin application starter using Undertow**, to use Undertow instead of Tomcat as embedded servlet container:

Maven coordinates:

```
<groupId>com.holon-platform.vaadin7</groupId>
<artifactId>holon-starter-vaadin-undertow</artifactId>
<version>5.1.0</version>
```

14. 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`.

15. System requirements

15.1. Java

The Holon Platform Vaadin module requires Java **8** or higher.

15.2. Vaadin

The Holon Platform Vaadin module requires Vaadin 7.7 or higher.