Holon Platform Vaadin Flow Module - Reference manual

Table of Contents

1. Introduction . . . 3
   1.1. Sources and contributions . . . 4
2. Obtaining the artifacts . . . 4
   2.1. Using the Platform BOM . . . 5
3. What's new in version 5.3.x . . . 5
4. Vaadin Flow UI components . . . 5
   4.1. Datastore API integration . . . 5
      4.1.1. Item types . . . 6
      4.1.2. Filter types . . . 7
      4.1.3. Custom item and filter type . . . 7
      4.1.4. `Datastore` Query configuration . . . 8
      4.1.5. Setup a `Datastore` data provider using builders . . . 10
   4.2. Internationalization . . . 10
      4.2.1. LocalizationContext and LocalizationProvider . . . 11
      4.2.2. Use a LocalizationContext as I18NProvider . . . 13
   4.3. The `HasComponent` interface . . . 14
   4.4. The `ValueHolder` interface . . . 15
   4.5. Component builders and providers . . . 16
      4.5.1. Component builders structure . . . 17
      4.5.2. Base components builders . . . 17
      4.5.3. Base components configurators . . . 21
   4.6. Dialogs . . . 21
      4.6.1. Message Dialog . . . 22
      4.6.2. Confirm Dialog . . . 22
      4.6.3. Question Dialog . . . 22
      4.6.4. Dialog configuration options . . . 23
      4.6.5. Adding components to the Dialog . . . 24
   4.7. List data in grids . . . 24
      4.7.1. PropertyListing . . . 24
      4.7.2. BeanListing . . . 25
      4.7.3. Item listing data source . . . 27
      4.7.4. Using a `Datastore` as item listing data source . . . 28
      4.7.5. Item listing *frozen* data source modality . . . 30
      4.7.6. Item listing data source additional items . . . 30

4.7.7. Item listing configuration . . . 31
4.7.8. Listen to listing row clicks . . . 32
4.7.9. Configure the item listing columns . . . 32
4.7.10. Default column value rendering strategy . . . 36
4.7.11. Render columns using Components . . . 37
4.7.12. Render columns using a `ViewComponent` . . . 37
4.7.13. Add a context menu . . . 37
4.7.14. Manage row details components . . . 38
4.7.15. Header and footer configuration . . . 39
4.7.16. Handling row selection . . . 40
4.7.17. Editing the listing items . . . 41
4.8. View components and forms . . . 46
4.8.1. Value conversion and formatting . . . 48
4.8.2. Using the property value presenters . . . 48
4.8.3. Create a `ViewComponent` from a generic `Component` . . . 49
4.8.4. `ViewComponent` property renderer . . . 51
4.8.5. Organize view components in groups and forms . . . 51
4.9. Input components and forms . . . 56
4.9.1. `Input` property renderer . . . 59
4.9.2. Input value conversion . . . 60
4.9.3. Input adapters for `HasValue` components . . . 61
4.9.4. Select type inputs . . . 61
4.9.5. Extend the Input API using adapters . . . 68
4.9.6. Validatable inputs . . . 69
4.9.7. Organize `Input` components in groups and forms . . . 70
5. Vaadin session scope . . . 77
6. Device information . . . 78
7. Routing and navigation . . . 78
7.1. Navigation parameters handling . . . 79
7.1.1. Query parameter URL encoding and decoding . . . 80
7.1.2. Using the injected parameter values . . . 80
7.1.3. Direct query parameter values deserialization . . . 81
7.1.4. Built in query parameter types . . . 83
7.1.5. Optional query parameter values . . . 84
7.1.6. Multiple query parameter values . . . 84
7.1.7. Adding query parameter type support . . . 84
7.1.8. Required query parameters . . . 86
7.1.9. Default query parameter values . . . 87
7.2. Using `@OnShow` on route target classes . . . 87
7.3. The `Navigator` API . . . 88
7.3.1. The navigation builder API . . . 91

7.3.2. Get the URL of a navigation target . . . 93
7.3.3. Listening to navigation changes . . . 93
7.3.4. Navigation links . . . 93
7.4. Authentication support for UI routes . . . 95
7.5. Authorization support for UI routes . . . 96
8. Spring integration . . . 97
8.1. Provide the `Navigator` API as a Spring bean . . . 97
8.2. Using Spring Security for authorization control . . . 98
9. Spring Boot integration . . . 98
9.1. `Navigator` API auto-configuration . . . 98
9.2. `LocalizationContext` integration auto configuration . . . 99
9.3. Spring Boot starters . . . 99
10. Loggers . . . 100
11. System requirements . . . 100
11.1. Java . . . 100
11.2. Vaadin . . . 100

Copyright © 2016-2019

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 **Vaadin Flow** module provides integration between the Holon Platform architecture and the Vaadin Flow web applications platform.

The module main features are:

- A complete set of UI components *fluent* builders, to make the UI development fast and easy.
- A full integration with the Holon Platform Property model and Datastore API for data bound UI components.
- A complete support for UI components and messages localization, using the Holon Platform internationalization support.
- A convenient *navigation* API to manage the UI *views*, with typed parameters support and components lifecycle management.
- Integration with the Holon Platform authentication and authentication architecture.
- A complete integration with **Spring** and **Spring Boot**, with application auto-configuration facilities.

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-flow.

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-flow-bom</artifactId>
<version>5.5.0</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-flow-bom</artifactId>
      <version>5.5.0</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. What's new in version 5.3.x

- **Default navigation error handler routes**: The default `UnauthorizedNavigationError` and `ForbiddenNavigationError` error handler routes, bound to the `UnauthorizedNavigationException` and `ForbiddenNavigationException` respectively, were removed to avoid conflicts with the Vaadin flow routes auto-scan when user-defined routes to handle the same exceptions are present in classpath.

4. Vaadin Flow UI components

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

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin-flow</artifactId>
<version>5.5.0</version>
```

It provides integration between the Vaadin Flow core architecture and the Holon Platform one, providing also a set of UI components specifically designed to ensure a seamless integration with the Holon Platform Property model, Datastore API, authentication and internationalization APIs.

4.1. Datastore API integration

The Holon Platform Datastore API can be used to bind a UI component to backend data in a simple and powerful way, relying on the Holon Platform Property model to represent the underlying data model.

The DatastoreDataProvider API is a standard Vaadin Flow `com.vaadin.flow.data.provider.DataProvider` which uses the `Datastore` API to fetch data from backend and makes available a set of configuration methods to control and setup the queries to perform.

Since it is a standard `DataProvider`, it can be seamlessy used in any Vaadin UI component which support data binding.

The `DatastoreDataProvider` provides a set of `create(..)` and `builder(..)` methods to build and configure a new `Datastore` based `DataProvider`. At least the following elements are required to

create a DatastoreDataProvider:

- The reference to the Datastore instance to use.
- A DataTarget declaration, which represents the data model *entity* to use as data source (for example, a RDBMS table name).

4.1.1. Item types

By default, the Holon Platform Property model is used to represent the backend data model attributes, and the PropertyBox type to represent a data source *item*, intended as a collection of properties and their values.

So, besides the Datastore reference and the DataTarget definition, the **property set** to use has to be provided to the DatastoreDataProvider builder methods.

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

final PropertySet<?> SUBJECT = PropertySet.of(ID, NAME); ①

final DataTarget<?> TARGET = DataTarget.named("subjects"); ②

final Datastore datastore = obtainDatastore(); ③

DataProvider<PropertyBox, QueryFilter> dataProvider = DatastoreDataProvider.create
(datastore, TARGET, SUBJECT); ④

dataProvider = DatastoreDataProvider.create(datastore, TARGET, ID, NAME); ⑤
```

① Property set declaration

② Data target definition

③ Datastore reference

④ Create a new DatastoreDataProvider using the SUBJECT property set

⑤ Create a new DatastoreDataProvider providing the ID and NAME properties as property set

Furthermore, the DatastoreDataProvider API provides methods to use a **bean class** as item type. The item property set is obtained form the bean class attributes, using the bean property names as property set property names.

See the Java Beans and property model documentation section for information about the conventions and the configuration options which can be used when a Java Bean is used with the Holon Platform property model.

```
final DataTarget<?> TARGET = DataTarget.named("subjects"); ①

final Datastore datastore = obtainDatastore(); ②

DataProvider<MyBean, QueryFilter> dataProvider = DatastoreDataProvider.create
(datastore, TARGET, MyBean.class); ③
```

① Data target definition

② Datastore reference

③ Create a new `DatastoreDataProvider` using the `MyBean` class as item type

4.1.2. Filter types

By default, the `QueryFilter` type is used used as data provider filter type, since it is the default property model representation for the `Datastore` API query restrictions definition.

To use a different data provider filter type, a **filter converter function** can be configured for the `DatastoreDataProvider`, to provide the custom filter type to `QueryFilter` type conversion logic.

In the example below, a `String` type filter is used and the conversion function creates a `QueryFilter` which represents the condition "The `NAME` data model attribute value starts with the given filter value":

```
DataProvider<PropertyBox, String> dataProvider = DatastoreDataProvider.create
(datastore, TARGET, SUBJECT,
    stringValue -> NAME.startsWith(stringValue));
```

Similarly when a bean item type is used:

```
DataProvider<MyBean, String> dataProvider = DatastoreDataProvider.create(datastore,
TARGET, MyBean.class,
    stringValue -> NAME.startsWith(stringValue));
```

4.1.3. Custom item and filter type

The `DatastoreDataProvider` API makes also available methods to customize both the item type and the data provider filter type, providing:

- The `PropertyBox` type to the custom item type conversion function.
- The custom filter type to the `QueryFilter` type conversion function.

For example, using a `MyItem` item class and a `String` filter type:

```
DataProvider<MyItem, String> dataProvider = DatastoreDataProvider.create(datastore,
TARGET, //
    SUBJECT, ①
    propertyBox -> new MyItem(propertyBox.getValue(ID), propertyBox.getValue(NAME)),
②
    stringValue -> NAME.startsWith(stringValue)); ③
```

① The `SUBJECT` property set is used as query projection

② Item conversion function: from the `PropertyBox` type object to the `MyItem` item type

③ Filter conversion function: from a `String` type filter to a `QueryFilter` type

4.1.4. `Datastore` Query configuration

The queries performed by the `Datastore` to fetch the data from backend can be configured using the `DatastoreDataProvider` **builder API**.

Query filters and sorts

The query filters and sorts can be configured in two ways:

1. At construction time, using the appropriate builder methods:

The `DatastoreDataProvider` builder API provide methods to:

- Add one or more `QueryFilter` to the query definition. These filters will always be included in the query execution.
- Add one or more `QuerySort` to the query definition. These sorts will always be included in the query execution.
- Set a default `QuerySort`, which will be used when no other sort is configured for the query.

```
DataProvider<?, ?> dataProvider = DatastoreDataProvider.builder(datastore, TARGET,
SUBJECT) ①
    .withQueryFilter(NAME.isNotNull()) ②
    .withQuerySort(NAME.asc()) ③
    .withDefaultQuerySort(ID.desc()) ④
    .build();
```

① Get a `DatastoreDataProvider` builder

② Add a query filter

③ Add a query sort

④ Set the default query sort

2. Dynamically, using a `QueryConfigurationProvider`:

One or more `QueryConfigurationProvider` references can be configured for the query using the `DatastoreDataProvider` builder API.

At each query execution, the `getQueryFilter()` and `getQuerySort()` methods of each available `QueryConfigurationProvider` will be invoked to obtain the filters and sorts to add to the query, if any.

```
DataProvider<?, ?> dataProvider = DatastoreDataProvider.builder(datastore, TARGET,
SUBJECT) ①
    .withQueryConfigurationProvider(new QueryConfigurationProvider() { ②

      @Override
      public QueryFilter getQueryFilter() { ③
        return null;
      }

      @Override
      public QuerySort getQuerySort() { ④
        return null;
      }

    }).build();
```

① Get a `DatastoreDataProvider` builder

② Add a `QueryConfigurationProvider`

③ At each query execution, this method will be invoked to obtain the query filters, if any

④ At each query execution, this method will be invoked to obtain the query sorts, if any

`QuerySortOrder` conversion

When the standard Vaadin Flow `com.vaadin.flow.data.provider.QuerySortOrder` type is used for the data provider query configuration, the `DatastoreDataProvider` can be configured to control how a `QuerySortOrder` declaration is converted into a `QuerySort` definition.

For this purpose, the `DatastoreDataProvider` builder API allows to configure a `QuerySortOrder` conversion function.

```
DataProvider<?, ?> dataProvider = DatastoreDataProvider.builder(datastore, TARGET,
SUBJECT) ①
    .querySortOrderConverter(querySortOrder -> { ②
      // QuerySortOrder to QuerySort conversion logic omitted
      return null;
    }).build();
```

① Get a `DatastoreDataProvider` builder

② Set the `QuerySortOrder` conversion function

Item identifier

By default, the item itself is used as *item identifier* to discern each item from another within the query results. The item identification relies on the standard Java `equals` and `hashCode` object

methods.

When a `PropertyBox` type item is used, the default item identification strategy relies on the property set **identifier properties** values, if declared.

See the Identifier properties documentation section for details.

If a custom item identifier should be used, an item identifier provider function can be configured using the `DatastoreDataProvider` builder API. This function must provide the item identifier for each item obtained form query execution.

Example using a PropertyBox type item

```
DataProvider<?, ?> dataProvider = DatastoreDataProvider.builder(datastore, TARGET,
SUBJECT)
    .itemIdentifierProvider(propertyBox -> propertyBox.getValue(ID)) ①
    .build();
```

① Set an item identifier provider function which provides the `ID` property value as item identifier

Example using a bean item

```
DataProvider<?, ?> dataProvider = DatastoreDataProvider.builder(datastore, TARGET,
MyBean.class)
    .itemIdentifierProvider(bean -> bean.getId()) ①
    .build();
```

① Set an item identifier provider function which provides the bean id value as item identifier using the `getId()` method

4.1.5. Setup a `Datastore` data provider using builders

The Holon Platform Vaadin Flow module *builder* APIs provides covenient methods to setup a `Datastore` based data provider for UI components which support data binding.

Example for a *single select*, implemented using a Vaadin `ComboBox` component:

```
SingleSelect<Long> select = Components.input.singleSelect(ID).dataSource(datastore,
TARGET).build();
```

See Component builders and providers to learn about the available component builders.

4.2. Internationalization

The Holon Platform Vaadin Flow module provides a complete integration with the Holon Platform internationalization architecture, making UI components and messages localization simple and conistent with the full application stack.

Each localizable messages is represented by:

- A required **message code**, i.e. the `String` key to be used to obtain the message localization for a given `Locale`.
- An optional **default message**, to be used if a localized message is not available for a `Locale`.
- A set of optional **localization arguments**, which can be used to replace a specific *placeholder* with the actual values.

The localizable message attributes described above can be declared and represented using the Localizable interface, which provides convenient methods to create `Localizable` instances providing the message localization attributes.

The UI component builders which allow to setup localizable messages and texts, always support the `Localizable` API to provide the message localization attributes, providing also convenient methods to directly set the message localization attributes.

Example using a *button* component builder:

```
Button button = Components.button().text("Not localizable").build(); ①

button = Components.button().text("Default message", "message.localization.code")
.build(); ②

button = Components.button().text("Default message", "message.localization.code",
"arg1", "arg2").build(); ③

Localizable message = Localizable.of("Default message", "message.localization.code");
button = Components.button().text(message).build(); ④
```

① Set the button text using a not localizable message

② Set the button text using a localizable message, providing the default message and the localization code

③ Set the button text using a localizable message, also providing two message localization arguments

④ Set the button text using a `Localizable` reference

See Component builders and providers to learn about the available component builders.

4.2.1. LocalizationContext and LocalizationProvider

The Holon Platform internationalization support relies on the LocalizationContext API, which acts as message resolver, localized data formatter and holder for the current user `Locale`.

In order for the UI components localization to work, one of the following conditions must be met:

- A `LocalizationContext` is available as *context* resource and a `Locale` is configured. See the Holon Platform Context architecture for information about the context resources management.

- Otherwise, a Vaadin Flow `I18NProvider` is available from the current Vaadin Service.

See Vaadin session scope to learn about the *context scope* for resources bound to the current Vaadin application session.

The LocalizationProvider API is used by the UI components and builders to obtain the localization information and to localize the messages to display.

The `LocalizationProvider` API makes available a set of static methods to seamlessy integrate the messages localization in any part of the application, delegating the actual internationalization logic to the concrete localization provider, if available, which can be either a Holon Platform `LocalizationContext` or a Vaadin Flow `I18NProvider`.

The **actual localization service to use** is automatically detected with the following strategy:

- If a Vaadin Flow `I18NProvider` is available from current `VaadinService`, it is used for messages localization.
- Otherwise, a `LocalizationContext` is used if available as a *context* resource.

This way, the `LocalizationProvider` API `getLocalization` static methods can be used as follows:

```
final Localizable message = Localizable.of("Default message",
"message.localization.code");

Optional<String> localized = LocalizationProvider.getLocalization(Locale.US, message);
①
String localizedOrDefault = LocalizationProvider.getLocalization(Locale.US, "Default
message",
    "message.localization.code"); ②
```

① Get the localization for the given `Localizable` message using the given `Locale`

② Get the localization for the given `message.localization.code` message code using the given `Locale` and the default `Default message` message if the localization is not available

If the `Locale` is not explicitly provided, the `LocalizationProvider` API `localize` static methods can be used and the **current Locale** is obtained with the following strategy:

- If a current `UI` is available and a `Locale` is configured for the `UI`, this one is used.
- If a `LocalizationContext` is available as a *context* resource and it is localized, the `LocalizationContext` current `Locale` is used.
- If a Vaadin Flow `I18NProvider` is available from current `VaadinService`, the first provided `Locale` is used if available.

```
final Localizable message = Localizable.of("Default message",
"message.localization.code");

Optional<Locale> locale = LocalizationProvider.getCurrentLocale(); ①

Optional<String> localized = LocalizationProvider.localize(message); ②
String localizedOrDefault = LocalizationProvider.localize("Default message",
"message.localization.code"); ③
```

① Get the current `Locale`, if available

② Get the localization for the given `Localizable` message using the current `Locale`

③ Get the localization for the given `message.localization.code` message code using the current `Locale`: the default `Default message` message is returned if the message localization or the current `Locale` is not available

4.2.2. Use a LocalizationContext as I18NProvider

The Holon Platform `LocalizationContext` can be used as a standard Vaadin Flow `I18NProvider` through the LocalizationContextI18NProvider API.

See the LocalizationContext documentation for information about the `LocalizationContext` configuration and use.

The `LocalizationContextI18NProvider` provides static methods to build a `I18NProvider` instance which uses a `LocalizationContext` as message localizations provider.

```
LocalizationContext localizationContext = LocalizationContext.builder()
    .withMessageProvider(MessageProvider.fromProperties("messages").build()) ①
    .withInitialLocale(Locale.US) ②
    .build();

I18NProvider i18nProvider = LocalizationContextI18NProvider.create(
localizationContext); ③
```

① Set the messages localization source

② Set the initial `Locale`

③ Create a `I18NProvider` instance using the `LocalizationContext` as message localization provider

A `LocalizationContextI18NProvider` which uses the current `LocalizationContext` (if available as a *context* resource) can be obtained with the `create()` method:

```
LocalizationContext localizationContext = LocalizationContext.builder()
    .withMessageProvider(MessageProvider.fromProperties("messages").build())
.withInitialLocale(Locale.US)
    .build(); ①

VaadinSession.getCurrent().setAttribute(LocalizationContext.class,
localizationContext); ②

I18NProvider i18nProvider = LocalizationContextI18NProvider.create(); ③
```

① Create a `LocalizationContext`

② Put the `LocalizationContext` instance as a Vaadin Session attribute

③ Create a `I18NProvider` instance which uses the current `LocalizationContext`

4.3. The `HasComponent` interface

Many of the additional UI components provided by the Holon Platform Vaadin Flow module extends the HasComponent interface to make available the actual Vaadin `com.vaadin.flow.component.Component` which can be used in the application UI.

The `getComponent()` method should be used to obtain the actual `Component`, for example to add it to a layout.

This because all the Holon UI components are represented using an *interface*, while the Vaadin `Component` object is a *class*. For this reason, the UI component interface cannot directly extend a class and the actual Vaadin `Component` is thus provided through the `getComponent()` method.

Example for an `Input` type component:

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

VerticalLayout layout = new VerticalLayout();
layout.add(input.getComponent()); ②
```

① Create a String type `Input`

② Use the `getComponent()` method to add the actual input component to a layout

All the Holon UI components interfaces are located in the `com.holonplatform.vaadin.flow.components` package.

Furthermore, the `HasComponent` API provides methods to check if the actual component supports a set of features and to obtain the appropriate interface which allow to manage a specific feature.

Some of the component features which can be checked and accessed are:

- *Enabled mode*: using `hasEnabled()`.
- *Size*: using `hasSize()`.

- *Style*: using `hasStyle()`.
- *Label*: using `hasLabel()`.

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

input.hasEnabled().ifPresent(e -> e.setEnabled(false)); ②
input.hasSize().ifPresent(s -> s.setSizeFull()); ③
input.hasStyle().ifPresent(s -> s.addClassName("my-style")); ④
input.hasLabel().ifPresent(l -> l.setLabel("My label")); ⑤
```

① Create a String type `Input`

② Check if the component supports the enabled mode and if so set it to `false`

③ Check if the component supports size and if so set it to full

④ Check if the component supports style and if so add a CSS style class name

⑤ Check if the component supports a label and if so set it to `My label`

4.4. The `ValueHolder` interface

The ValueHolder API is the abstract representation for components which handle a **typed value**, allowing to get and set the value according to the value type with which they are declared.

A set of convenience methods are provided to check if the value holder is *empty* (i.e. has no value) and to clear the value.

See View components and forms and Input components and forms to check two `ValueHolder` implementations.

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

input.setValue("String value"); ②
String value = input.getValue(); ③
Optional<String> optionalValue = input.getValueIfPresent(); ④

boolean empty = input.isEmpty(); ⑤
input.clear(); ⑥
```

① Create a `String` type `Input`, which extends `ValueHolder`

② Set the value

③ Get the value

④ Get the value, if present

⑤ Check whether the input is empty

⑥ Clear the value

Furthermore, the `ValueHolder` type components allow to listen to value changes, using a

`ValueChangeListener`. The value change event provides both the changed and the previous value, and the information about the value change context, including whether the value change was originated by a user action or programmatically.

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

input.addValueChangeListener(event -> { ②
  String oldValue = event.getOldValue(); ③
  String newValue = event.getValue(); ④
  boolean byUser = event.isUserOriginated(); ⑤
});
```

① Create a `String` type `Input`, which extends `ValueHolder`

② Add a `ValueChangeListener`

③ Get the previous value

④ Get the changed value

⑤ Get whether the value change was originated by a user action or programmatically

4.5. Component builders and providers

The Components API is **the main entry point** to build and configure the UI components.

It provides a complete set of methods to obtain *fluent* builders to create and configure both standard Vaadin Flow components and the additional UI components provided by the Holon Platform Vaadin Flow module.

The `Components` API 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 base Vaadin components, such as `Button` and standard layout components. See Base components builders.
dialog	Message and question dialogs builders. See Dialogs.
view	Builders for view components and forms. See View components and forms.
input	Builders input components and forms. See Input components and forms.
listing	Builders for grid type components to list data. See List data in grids.

4.5.1. Component builders structure

Each component builder API is composed by a set of specific builders, each related to a specific feature of the component. The feature builders are independent from the specific UI component to build, thus providing a common API which makes component configuration simple and intuitive.

For example, the HasTextConfigurator represents a builder part for UI components which supports a *text*, and it is included in any builder for UI components with text support.

Similarly, the the HasSizeConfigurator API is included in component builders which supports the component *size* configuration.

This ensures a consistent and common API to build and configure any UI component.

In the example below, the `HasTextConfigurator` and the `HasSizeConfigurator` are used in the same way to configure a *label* and a *button* component text and size:

```
Div label = Components.label() ①
    .text("Label text") ②
    .width("200px") ③
    .build();

Button button = Components.button() ④
    .text("Button text") ⑤
    .width("200px") ⑥
    .build();
```

① Get a *label* component builder

② Set the label text (using the `HasTextConfigurator` API)

③ Set the label width (using the `HasSizeConfigurator` API)

④ Get a *button* component builder

⑤ Set the button text (using the `HasTextConfigurator` API)

⑥ Set the button width (using the `HasSizeConfigurator` API)

4.5.2. Base components builders

Button

The `Components.button()` method provides a standard Vaadin `Button` component builder, with localizable text support.

```
Button button = Components.button() ①
    .text("Button text") ②
    .text("Default text", "message.code") ③
    .description("The description") ④
    .fullWidth() ⑤
    .styleName("my-style") ⑥
    .withThemeVariants(ButtonVariant.LUMO_PRIMARY) ⑦
    .icon(VaadinIcon.CHECK) ⑧
    .disableOnClick() ⑨
    .onClick(event -> { ⑩
      // do something
    }).build();

button = Components.button("Button text", e -> {
  /* do something */ }); ⑪
```

① Get the `Button` builder

② Set the button text

③ Set the button localizable text, providing a default text and a message localization code

④ Set the button description

⑤ Set the button width to 100%

⑥ Add a CSS style class name

⑦ Add a theme variant

⑧ Set the button icon

⑨ Automatically disables button when clicked

⑩ Add a click event listener

⑪ Shorter to directly create a `Button` providing the text and the click handler

The `Components.nativeButton()` builder can be used to build a `NativeButton` component.

Label

A *label* component can be used to display a text, which can be **localizable**, and can be implemented using a set of tags. The default tag used by the `Components.label()` method is `DIV`.

See Internationalization for information about the UI components text localization.

```
Div label = Components.label() ①
    .text("Label text") ②
    .text("Default text", "message.code") ③
    .description("The description") ④
    .fullWidth() ⑤
    .styleName("my-style") ⑥
    .build();
```

① Get a label builder using a `Div` component

② Set the label text

③ Set the label localizable text, providing a default text and a message localization code

④ Set the label description

⑤ Set the label width to 100%

⑥ Add a CSS style class name

Other builders are available for the tags: `P`, `SPAN`, `H1`, `H2`, `H3`, `H4`, `H5`, `H6`.

```
Paragraph p = Components.paragraph().text("text").build(); ①
Span span = Components.span().text("text").build(); ②
H1 h1 = Components.h1().text("text").build(); ③
H2 h2 = Components.h2().text("text").build(); ④
H3 h3 = Components.h3().text("text").build(); ⑤
H4 h4 = Components.h4().text("text").build(); ⑥
H5 h5 = Components.h5().text("text").build(); ⑦
H6 h6 = Components.h6().text("text").build(); ⑧
```

① Get a label builder using a `Paragraph` component

② Get a label builder using a `Span` component

③ Get a label builder using a `H1` component

④ Get a label builder using a `H2` component

⑤ Get a label builder using a `H3` component

⑥ Get a label builder using a `H4` component

⑦ Get a label builder using a `H5` component

⑧ Get a label builder using a `H6` component

Layouts

The Vaadin Flow base layouts can be created and configured using the following builders:

- *Vertical layout: * `Components.vl()`.
- *Horizontal layout: * `Components.hl()`.
- *Form layout: * `Components.formLayout()`.

```
VerticalLayout verticalLayout = Components.vl() ①
    .spacing() ②
    .margin() ③
    .alignItems(Alignment.CENTER) ④
    .justifyContentMode(JustifyContentMode.END) ⑤
    .fullSize() ⑥
    .styleName("my-style") ⑦
    .add(Components.button().text("Button text").build()) ⑧
    .add(Components.label().text("Label1").build(), Components.label().text("Label2")
.build()) ⑨
    .addAndAlign(myComponent1, Alignment.START) ⑩
    .build();

HorizontalLayout horizontalLayout = Components.hl() ⑪
    .add(myComponent1) ⑫
    .add(myComponent2) ⑬
    .flexGrow(1, myComponent1) ⑭
    .build();

FormLayout formLayout = Components.formLayout() ⑮
    .add(myComponent1) ⑯
    .withFormItem(myComponent1, "Label 1") ⑰
    .responsiveSteps(new ResponsiveStep("100px", 2)) ⑱
    .build();
```

① Get a `VerticalLayout` builder

② Enable spacing

③ Enable margins

④ Set the default component alignment

⑤ Set the justify content mode

⑥ Set width and height to 100%

⑦ Add a CSS style class name

⑧ Add a Button component

⑨ Add two label components

⑩ Add a component and set its alignment

⑪ Get a `HorizontalLayout` builder

⑫ Add a component

⑬ Add another component

⑭ Set the flex grow ratio for the first component

⑮ Get a `FormLayout` builder

⑯ Add a component

⑰ Add a FormItem slot with a component and its label

⑱ Set the responsive steps

Context menu

The `Components.contextMenu()` method can be used to configure a `ContextMenu` and bind it to a component.

```
Components.contextMenu() ①
    .withItem("Item 1").onClick(e -> {
      // do something
    }).add() ②
    .withItem("Item 2", "message.code").disabled().add() ③
    .withItem(itemComponent).add() ④
    .withOpenedChangeListener(e -> { ⑤
      e.isOpened();
    }).build(myComponent); ⑥
```

① Get a `ContextMenu` builder

② Add a menu item with a text and a click handler

③ Add disabled a menu item with a localizable text

④ Add a menu item using a component

⑤ Add a menu open change listener

⑥ Build the `ContextMenu` and bind it to the `myComponent` component

4.5.3. Base components configurators

Besides the builders, the `Components` API provides a set of *configurator* methods, which can be used to **configure existing UI components**. The available configurators acts on some base UI components: label, Button and layouts.

```
Button btn = new Button();

Components.configure(btn).text("Default text", "message.code"); ①

VerticalLayout vl = new VerticalLayout();

Components.configure(vl).withoutSpacing().add(myComponent); ②
```

① Configure the `Button`, setting a localizable text

② Configure the `VerticalLayout`, disabling spacing and adding a component

4.6. Dialogs

The `Components.dialog` sub-interface provides a set of method to build and configure a Vaadin Flow `Dialog`. Three types of dialogs are available, as described below.

4.6.1. Message Dialog

A **message dialog** is a `Dialog` which can be used to display a simple message to the user.

```
Dialog dialog = Components.dialog.message().text("Message").build(); ①

Components.dialog.message().text("Default text", "message.code").open(); ②

Components.dialog.showMessage("Default text", "message.code"); ③
```

① Build a message `Dialog` with given message text

② Build a message `Dialog` with given localizable message text and open it

③ Shorter to build and show a message `Dialog` with a localizable message text

4.6.2. Confirm Dialog

A **confirm dialog** is a `Dialog` which can be used to display a simple message to the user and provides a `OK` button at the bottom right position.

The `OK` button can be configured using the dialog builder.

```
Components.dialog.confirm() ①
    .text("Default text", "message.code") ②
    .okButtonConfigurator(button -> { ③
      button.text("My text").icon(VaadinIcon.CHECK);
    }).open(); ④

Components.dialog.showConfirm("Default text", "message.code"); ⑤
```

① Get a confirm `Dialog` builder

② Set the localizable dialog message

③ Configure the confirm dialog OK button

④ Build and open the dialog

⑤ Shorter to build and show a confirm `Dialog` with a localizable message text

4.6.3. Question Dialog

A **question dialog** is a `Dialog` which can be used to ask a question to the user and handle the user answer. A `QuestionDialogCallback` function must be provided to handle the user answer.

Two buttons are displayed at the bottom right position, which by default represents the `yes` or `no` answers. The buttons can be configured using the dialog builder.

```
Components.dialog.question(confirm -> { ①
  // handle user response (true/false)
}).text("Default text", "message.code") ②
    .confirmButtonConfigurator(button -> { ③
      // confirm button configuration
    }).denialButtonConfigurator(button -> { ④
      // deny button configuration
    }).open(); ⑤

Components.dialog.showQuestion(confirm -> {
  /* handle user response */ }, "Default text", "message.code"); ⑥
```

① Get a question `Dialog` builder, providing the user answer callback

② Set the localizable dialog message

③ Configure the *confirm* dialog button

④ Configure the *deny* dialog button

⑤ Build and open the dialog

⑥ Shorter to build and show a question `Dialog` with a localizable message text

4.6.4. Dialog configuration options

All the dialog builders provides a set of common `Dialog` configuration options, including:

- Dialog size configuration
- Dialog style configuration
- Dialog closing mode configuration and open change listener

```
Components.dialog.message().text("Default text", "message.code") //
    .width("200px") ①
    .height("200px") ②
    .styleName("my-style") ③
    .closeOnEsc(false) ④
    .closeOnOutsideClick(true) ⑤
    .withOpenedChangeListener(event -> { ⑥
      // open changed
    }).build();
```

① Set the dialog width

② Set the dialog height

③ Add a CSS style class name

④ Disable closing when `ESC` pressed

⑤ Enable closing when user clicks outside the dialog

⑥ Add a open change listener

4.6.5. Adding components to the Dialog

The dialog builders allow to add custom components to the `Dialog` content. The components can be added in two positions:

- In the main dialog content: will be placed after the dialog message label, if any.
- In the dialog bottom toolbar: will be placed before the standard dialog buttons, if any.

```
Components.dialog.message().text("Default text", "message.code") //
    .withComponent(Components.label().text("My label").build()) ①
    .withToolbarComponent(Components.button().text("My button").build()) ②
    .open();
```

① Add a label component in the dialog content area

② Add a button component in the dialog toolbar

4.7. List data in grids

The ItemListing API represents a UI component which can be used to display data in a tabular way.

Extends `HasComponent` (see The `HasComponent` interface) and it is actually implemented using a Vaadin Flow `Grid` component.

A `ItemListing` is designed to display a set of data **items** of a specific type, each one rendered as a list *row*. An item is a collection of values, and each value is bound to a **property** definition of a specific type.

Each `ItemListing` column is bound to an item *property* by default, even if *virtual* or calculated columns can be added to the listing.

Two `ItemListing` implementations are available:

- `PropertyListing`: to use the Holon Platform *property model* to represent the item properties and the `PropertyBox` type as item representation. See PropertyListing.
- `BeanListing`: to use a Java Bean class as item type and the bean property names as item properties. See BeanListing.

Each item listing implementation provides one or more suitable `builder` methods to obtain the *fluent* builder to use to create and configure an item listing instance.

The same builders are also available from the `Components` API, through the `Components.listing` sub interface. See Component builders and providers.

4.7.1. PropertyListing

The PropertyListing API is the default `ItemListing` implementation which uses the **Holon Platform Property model** to represent and manage the data items.

- A **property set** must be provided at `PropertyListing` costruction time. It represents the available item properties, each one bound by default to a listing *column*.
- The `PropertyBox` type is used to represent an *item instance*, using the listing property set. So each listing row is bound to a `PropertyBox` instance, which collects the property values to be displayed for each listing column.

For example, given the following property model definition:

```
static final NumericProperty<Long> ID = NumericProperty.longType("id");
static final StringProperty NAME = StringProperty.create("name");
static final PropertySet<?> SUBJECT = PropertySet.of(ID, NAME);
```

A `PropertyListing` can be created in the following way:

```
PropertyListing listing = PropertyListing.builder(SUBJECT).build(); ①

listing = Components.listing.properties(ID, NAME).build(); ②

new VerticalLayout().add(listing.getComponent()); ③
```

① Build a `PropertyListing` using the `SUBJECT` property set. The listing will provide two columns, one bound to the `ID` item property and the other bound to the `NAME` item property

② Build a `PropertyListing` using the `Components` API, directly providing the `ID` and `NAME` properties as listing property set

③ Just like any `HasComponent` component type, the `getComponent()` method should be used to obtain the actual listing Component

4.7.2. BeanListing

The BeanListing API is the default `ItemListing` implementation to use a **Java Bean class** as item type.

- A **bean class** must be provided at `BeanListing` costruction time. The bean class *property names* will be used as listing property set and so rendered as listing *columns*. For this reason the item listing property definition type is a `String`.
- Each listing *row* is bound to a bean class instance, and the bean class *getters* will be used to obtain the property value to be rendered for each listing column.

For example, given the following bean class definition:

```
class MyBean {

  private Long id;
  private String name;

  public MyBean() {
    super();
  }

  public MyBean(Long id, String name) {
    super();
    this.id = id;
    this.name = name;
  }

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

}
```

A `BeanListing` can be created in the following way:

```
BeanListing<MyBean> listing = BeanListing.builder(MyBean.class).build(); ①

listing = Components.listing.items(MyBean.class).build(); ②

new VerticalLayout().add(listing.getComponent()); ③
```

① Build a `BeanListing` using the `MyBean` class. The listing will provide two columns, one bound to the `id` bean property name and the other bound to the `name` bean property name

② Build a `BeanListing` using the `Components` API

③ Just like any `HasComponent` component type, the `getComponent()` method should be used to obtain the actual listing Component

4.7.3. Item listing data source

To obtain the item set to display, an item listing needs a *data source* to be configured. This can be achieved either using the `items(…)` or the `datasource(…)` builder methods, to provide a static set of items or a backend data connection respectively.

1. Using a static set of items:

The item listing item set can be provided at listing costruction time, using the `items(…)` or `addItem(…)` builder methods. The provided item instances must be of the same type of the listing item type.

The provided item will be displayed in the same order they are provided at construction time.

Example using a `PropertyListing`:

```
PropertyListing listing = PropertyListing.builder(SUBJECT) ①
    .items(PropertyBox.builder(SUBJECT).set(ID, 1L).set(NAME, "One").build()).build();

listing = PropertyListing.builder(SUBJECT) ②
    .addItem(PropertyBox.builder(SUBJECT).set(ID, 1L).set(NAME, "One").build())
    .addItem(PropertyBox.builder(SUBJECT).set(ID, 2L).set(NAME, "Two").build()).build
();
```

① Set the listing items

② Set the listing items using `addItem(…)`

Example using a `BeanListing`:

```
BeanListing<MyBean> listing = BeanListing.builder(MyBean.class) ①
    .items(new MyBean(1L, "One"), new MyBean(2L, "Two")).build();

listing = BeanListing.builder(MyBean.class) ②
    .addItem(new MyBean(1L, "One")) //
    .addItem(new MyBean(2L, "Two")).build();
```

① Set the listing items

② Set the listing items using `addItem(…)`

2. Using a `DataProvider`:

A Vaadin `com.vaadin.flow.data.provider.DataProvider` can be used as items data source. This allows to provide a *dynamic* set of items, for example using a backend service.

The `DataProvider` data type must be consistent with the listing item type.

Example using a `PropertyListing`:

```
DataProvider<PropertyBox, ?> dataProvider = getPropertyBoxDataProvider(); ①

PropertyListing listing = PropertyListing.builder(SUBJECT).dataSource(dataProvider) ②
    .build();
```

① Obtain a `DataProvider` using the `PropertyBox` data type

② Set the data provider as listing data source

Example using a `BeanListing`:

```
DataProvider<MyBean, ?> dataProvider = getBeanDataProvider(); ①

BeanListing<MyBean> listing = BeanListing.builder(MyBean.class).dataSource
(dataProvider) ②
    .build();
```

① Obtain a `DataProvider` using the `MyBean` data type

② Set the data provider as listing data source

When a `DataProvider` is used as data source, the `refresh()` and `refreshItem(T item)` can be used to refresh the listing item set:

- The `refresh` method can be used to refresh the whole item set. In most cases, the data provider configured as data source will execute again a query to obtain the data from the backend.
- The `refreshItem` method can be used to refresh a single item, providing an updated instance to be included in the listing item set, replacing the previous one.

```
PropertyListing listing = PropertyListing.builder(SUBJECT).dataSource(getDataProvider
()).build(); ①

listing.refresh(); ②
listing.refreshItem(itemToRefresh); ③
```

① Create a listing and use a data provider as data source

② Refresh all the listing items

③ Refresh a specific item

4.7.4. Using a `Datastore` as item listing data source

The Holon Platform Datastore API can be used to implement a listing data source, and maybe it is the most straight and easy way to connect a *property model* based listing to a backend data source.

This first way to use a `Datastore` as listing data source is to use the `DataProvider` adapter, available through the DatastoreDataProvider API.

See the Datastore API integration section to learn how to create and configure a `Datastore` based

`DataProvider`.

When a `Datastore` is used, a `DataTarget` definition is required to declare the data model *entity* to use as items data source (for example a RDBMS table name, or a collection name in a document based database).

Supposing to use the following `DataTarget` definition:

```
static final DataTarget<?> TARGET = DataTarget.named("subjects");
```

A `PropertyListing` data source can be declared as follows:

```
Datastore datastore = getDatastore();

PropertyListing listing = PropertyListing.builder(SUBJECT) ①
    .dataSource(DatastoreDataProvider.create(datastore, TARGET, SUBJECT)) ②
    .build();
```

① Create a `PropertyListing` using the `SUBJECT` property set

② Use the `DatastoreDataProvider` API to create a data provider which uses the provided `Datastore` and `DataTarget` to perform backend data queries

But the most straight and easy way to use a `Datastore` as listing data source is to rely on the **listing builder methods** available for this purpose. Furthermore, the builder methods allow to avoid to specify the query projection **property set** again, using the listing property set by default.

```
Datastore datastore = getDatastore();

PropertyListing listing = PropertyListing.builder(SUBJECT) ①
    .dataSource(datastore, TARGET) ②
    .build();
```

① Create a `PropertyListing` using the `SUBJECT` property set

② Directly use the `dataSource` builder method to create a data provider which uses the provided `Datastore` and `DataTarget` to perform backend data queries. In this case, there is no need to specify the `SUBJECT` property set, since the listing property set is used as query projection

The `dataSource` builder method turns the root listing builder in a Datastore-aware builder, making available a set of additional methods which can be used to configure the `Datastore` based data provider, for example to **add query filters and sorts**. The `Datastore` based data provider configuration methods corresponds to the ones provided by the `DatastoreDataProvider` API. See `Datastore` Query configuration for details.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .dataSource(getDatastore(), TARGET) ①
    .withQueryFilter(NAME.isNotNull()) ②
    .withQuerySort(ID.asc()) ③
    .build();
```

① Set the listing data source using a `Datastore` and providing the data target to use

② Configure the `Datastore` data provider adding a query filter

③ Configure the `Datastore` data provider adding a query sort

4.7.5. Item listing *frozen* data source modality

The item listing data source can be set in a *frozen* state using the item listing API:

- When the item listing is in *frozen* state, it **never shows any item** and **no fetch is performed** from the data provider.
- When the item listing `refresh()` method is invoked, the *frozen* state is automatically disabled, allowing the listing to fetch and display the items.

The *frozen* state can be useful when you don't want to fetch the items just after the listing is displayed in UI, maybe because it involves an onerous backend query. The backend query is only performed when the `refresh()` method is called, typically by a user explicit action.

The item listing *frozen* state can be configured either from the buider API or from the `ItemListing` API:

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .frozen(true) ①
    .build();

listing.setFrozen(true); ②

boolean frozen = listing.isFrozen(); ③

listing.refresh(); ④
```

① Use the builder API to set the listing in *frozen* state

② Use the listing API to set the listing in *frozen* state

③ Check whether the listing is in *frozen* state

④ Calling `refresh()` will disable the *frozen* state, allowing the listing to fetch and display the items

4.7.6. Item listing data source additional items

The `ItemListing` API supports *additional* data source items. Additional items can be added using the `ItemListing` API and they will appear as conventional listing items, even they are not part of the items returned by the concrete data source.

Additional items can be used, for example, to allow the user to **add** an item to the listing, edit its data and then saving the item in the concrete datastore. At this point, the additional item can be removed, since the saved item will now be returned by the data source query.

The `ItemListing` API provides method to manage additional items: add an additional item, remove an additional item, remove all additional items and obtain the current additional items.

```
PropertyListing listing = PropertyListing.builder(SUBJECT).build();

listing.addAdditionalItem(myItem); ①

List<PropertyBox> additionalItems = listing.getAdditionalItems(); ②

listing.removeAdditionalItem(myItem); ③

listing.removeAdditionalItems(); ④
```

① Add an additional item

② Get the current additional items

③ Remove the additional item

④ Remove all the additional items

4.7.7. Item listing configuration

The item listing builders provides a set of methods to configure the item listing component, most of them coming from the Vaadin `Grid` component configuration options.

This includes for example setting the page size, setting whether the listing column are resizable and can be reordered, setting the *frozen* columns and so on.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .pageSize(50) ①
    .columnReorderingAllowed(true) ②
    .resizable(true) ③
    .frozenColumns(1) ④
    .heightByRows(true) ⑤
    .verticalScrollingEnabled(true) ⑥
    .multiSort(true) ⑦
    .autoWidth(ID) ⑧
    .columnsAutoWidth() ⑨
    .withColumnResizeListener(evt -> { ⑩
      // ...
    }).withColumnReorderListener(evt -> { ⑪
      // ...
    }).build();
```

① Sets the page size (the number of items fetched at a time from the data source)

② Set whether the columns can by reordered by the user

③ Set whether the columns can by resized by the user

④ Set the number of *frozen* columns, i.e. the fixed columns, not involved in horizontal scrolling

⑤ Set whether the listing's height is defined by the number of its rows

⑥ Set whether the vertical scrolling is enabled

⑦ Set whether multiple column sorting is enabled on the client-side

⑧ Set the column auto-width mode for the `ID` property

⑨ Set the auto-width mode for all the listing columns

⑩ Add a listener for listing columns resize events

⑪ Add a listener for listing columns reordering events

4.7.8. Listen to listing row clicks

One ore more **item click listener** can be added to the listing to handle user clicks on the listing rows. The click event provides the clicked item instance and a reference to the listing on which the click occurred.

Furthermore, a set of additional click information are provided, such as the click count, the modifier keys, the screen coordinates and so on.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .withItemClickListener(event -> { ①
      PropertyBox clickedItem = event.getItem(); ②
      PropertyListing source = event.getSource(); ③
      event.getClickCount(); ④
      event.getButton(); ⑤
      event.isCtrlKey(); ⑥
      /* other getters omitted */
    }).build();
```

① Add an item click listener

② Get the item instance on which the click occurred

③ Get the item listing reference

④ Get the consecutive clicks count

⑤ Get the mouse button, if available

⑥ Check whether the CTRL key was down when the click occurred

4.7.9. Configure the item listing columns

The item listing columns are auto-generated by default using the provided listing **property set**.

- `PropertyListing`: each property declared in the property set will be rendered as a column. You can refer to a column using the corresponding *property definition*.

- `BeanListing`: each available bean property will be rendered as a column. You can refer to a column using the corresponding bean *property name*.

Change the column order and visibility

A set of builder methods are available to change the listing column display order. The builder methods allow to:

- Explicitly provide the column to show and their order.
- Declare the column position relative to another column (before or after)
- Display a column as first or last in the columns set
- Set a column as hidden

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .visibleColumns(NAME, ID) ①
    .displayAsFirst(NAME) ②
    .displayAsLast(ID) ③
    .displayBefore(NAME, ID) ④
    .displayAfter(ID, NAME) ⑤
    .hidden(ID) ⑥
    .build();

BeanListing<MyBean> listing2 = BeanListing.builder(MyBean.class) ⑦
    .visibleColumns("name", "id").displayAsFirst("name").hidden("id").build();
```

① Declare the visibile columns and the display order

② Set the `NAME` column to be displayed as first

③ Set the `ID` column to be displayed as last

④ Set the `NAME` column to be displayed before the `ID` column

⑤ Set the `ID` column to be displayed after the `NAME` column

⑥ Set the `ID` column as hidden

⑦ Example using a `BeanListing`

Column configuration

The item listing builder provides a set of methods to configure a listing column, including:

- Se the column *header*, with **localizable messages** support. See Internationalization.
- Se the column *footer*, with **localizable messages** support. See Internationalization.
- Set the column width and alignment
- Set whether the column is resizable by the user
- Set whether the column is sortable and optionally provide the sort logic
- Set the column value provider and renderer

- Set whether the column is *frozen*, i.e. fixed and not involved in horizontal scrolling

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .header(NAME, "The name") ①
    .header(NAME, "The name", "name.message.code") ②
    .headerComponent(NAME, new Button("name")) ③
    .width(NAME, "100px") ④
    .flexGrow(NAME, 1) ⑤
    .alignment(NAME, ColumnAlignment.CENTER) ⑥
    .footer(NAME, "Footer text") ⑦
    .resizable(NAME, true) ⑧
    .sortable(NAME, true) ⑨
    .sortUsing(NAME, ID) ⑩
    .sortProvider(NAME, direction ->
    /* sort logic omitted */
    null) ⑪
    .valueProvider(NAME, item -> item.getValue(NAME)) ⑫
    .renderer(NAME, new TextRenderer<>()) ⑬
    .frozen(NAME, true) ⑭
    .build();
```

① Set the `NAME` column header

② Set the `NAME` column header using a localizable message

③ Set the `NAME` column header using a `Component`

④ Set the `NAME` column width

⑤ Set the `NAME` column flex grow ratio

⑥ Set the `NAME` column alignment

⑦ Set the `NAME` column footer

⑧ Set whether the `NAME` column is resizable

⑨ Set whether the `NAME` column is sortable

⑩ Set that the `ID` property must be used to sort the `NAME` column

⑪ Set the `NAME` column sort logic

⑫ Set the `NAME` column value provider

⑬ Set the `NAME` column renderer

⑭ Set the `NAME` column as frozen

Adding columns

New *virtual* columns can be added to an item listing, i.e. columns which are not directly bound to an item property.

A virtual column can be used for example to display a **calculated** result, using the item instance property values, or to display a **UI component**.

1. Add a virtual column:

A new column can be added to the listing using the `withColumn` builder method and providing a function to obtain the column value, given the current **row item instance** (a `PropertyBox` for a `PropertyListing` and a bean class instance for a `BeanListing`.

The added column can be **configured**, with the same configuration options described in the section above. Then, the column is added to the listing using the `add()` method.

Example using a `PropertyListing`:

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .withColumn(item -> "Virtual: " + item.getValue(ID)) ①
    .header("Virtual") ②
    .displayBefore(NAME) ③
    .add() ④
    .build();
```

① Add a new virtual column, which uses the row item to provide its value

② Configure the added column setting the header text

③ Set to display the added column before the `NAME` column

④ Add the column to the listing

Example using a `BeanListing`:

```
BeanListing<MyBean> listing2 = BeanListing.builder(MyBean.class) //
    .withColumn(item -> "Virtual: " + item.getId()) ①
    .header("Virtual") ②
    .displayAsFirst() ③
    .add() ④
    .build();
```

① Add a new virtual column, which uses the row item to provide its value

② Configure the added column setting the header text

③ Set to display the added column as first

④ Add the column to the listing

2. Add a `Component` type column:

A convenience `withComponentColumn` method is available to add a Vaadin `Component` type column, automatically configuring a suitable Component renderer for the column itself.

Just like a virtual column, the column can be configured and then the `add()` method should be used to add it to the listing.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .withComponentColumn(item -> new Button(item.getValue(NAME))) ①
    .header("Component") ②
    .sortUsing(NAME) ③
    .add() ④
    .build();
```

① Add a new `Component` type column, which provides a `Button` and uses the row item to set the button text

② Configure the added column setting the header text

③ Set that the `NAME` property has to be used to sort the added column

④ Add the column to the listing

4.7.10. Default column value rendering strategy

If an explicit column renderer is not configured for a column, the default column value rendering strategy uses the Holon Platform StringValuePresenter API to render a column value.

This ensures a consistent behaviour across the application on how the values are displayed and the current `LocalizationContext` is used, if available, to apply the proper internationalization conventions to format, for example, a numeric value.

See the StringValuePresenter API documentation for details.

Furthermore, for a `PropertyListing` implementation, the PropertyValuePresenter architecture is used to render the column value, allowing to extend and tune the presentation strategy at higher level, ensuring consistency across the application UI and backend.

See the Property value presentation documentation for details.

For example, suppose we want to add a `#` prefix to each value of the `ID` property. We can declare a `PropertyValuePresenter` and bind it to the `ID` property in the property presenters registry:

```
PropertyValuePresenterRegistry.getDefault() ①
    .forProperty(ID, (property, value) -> "#" + value); ②
```

① Get the default `PropertyValuePresenterRegistry`

② Register a `PropertyValuePresenter`, bound to the `ID` property, which adds a `#` prefix to the property value

From now on, any `PropertyListing` instance will use the new presenter to render the `ID` property column value.

```
PropertyListing listing = PropertyListing.builder(SUBJECT).build(); ①

String value = ID.present(1L); ②
```

① The `PropertyListing` instance will use the new presenter to render the `ID` property column value (so the `ID` column cells will show `#1`, `#2` and so on)

② Any other property presentation service will use the new presenter for the `ID` property value. In this example, the `ID` property is directly used to present the `1` value (declared as a Long) and the result will be the String `#1`.

4.7.11. Render columns using Components

The `PropertyListing` builder makes available helper methods to render a column using a `Component`, providing the function to use to obtain the `Component` instance for each cell, given the current row `PropertyBox` item.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .componentRenderer(NAME, item -> { ①
      return new Button(item.getValue(NAME));
    }).build();
```

① Render the column identified by the `NAME` property as a `Component`. In this example, a `Button` is returned, with button text setted as the current item `NAME` property value

4.7.12. Render columns using a `ViewComponent`

A `PropertyListing` column can be rendered using a `ViewComponent`, leveraging the `ViewComponent` rendering architecture and thus providing a consistent data rendering strategy across all the application UI components.

See View components and forms for information about `ViewComponent` purpose and configuration.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .renderAsViewComponent(NAME) ①
    .build();
```

① The column identified by the `NAME` property will be rendered using a `ViewComponent`

4.7.13. Add a context menu

The item listing supports a **context menu**, which can be added to the listing and will open by default when a *right click* or a *long tap* is performed by the user on a listing row.

The item listing context menu builder provides a set of methods to:

- Add context menu item using a **text**, with message localization support.
- Add context menu item using a **Component**.
- Set the menu item **click handler**, providing an event form which to obtain the **listing item instance** of the row on which to context menu was opened and a reference to the item listing itself.
- Configure the context menu items.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .contextMenu() ①
    .withItem("Context menu item 1") ②
    .withClickListener(e -> { ③
      PropertyBox rowItem = e.getItem(); ④
      e.getItemListing(); ⑤
      Notification.show("Context menu item clicked on row id " + rowItem.getValue(ID)
);
    }).add() ⑥
    .withItem("Context menu item 2", "item.message.code", e -> {
      /* do something */}) ⑦
    .withItem(new Button("Context menu item 3"), e -> {
      /* do something */}) ⑧
    .add() ⑨
    .build();
```

① Create a context menu for the listing: a context menu builder is returned to configure the menu items

② Add a menu item with given text

③ Configure the menu item adding the click handler

④ The click event provides the listing item instance of the row on which to context menu was opened

⑤ The parent item listing can be obtained from the click event

⑥ Add the context menu item to the context menu

⑦ Add another context menu item, directly providing a localizable text and the item click handler

⑧ Add a `Component` type context menu item, directly providing a `Button` and the item click handler

⑨ Add the configured context menu to the item listing

4.7.14. Manage row details components

The item listing supports **row details** components, which can be opened and closed under each listing row.

To enable the item row details, a function which provides the details content for each row has to be provided. For each row, the **listing row item** instance is provided as function argument.

The item details contents can be rendered in two ways:

- **As a text:** using the `itemDetailsText` builder method.
- **As a `Component`:** using the `itemDetailsComponent` builder method.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .itemDetailsText(item -> "Detail of: " + item.getValue(ID)) ①
    .itemDetailsComponent(item -> new Button(item.getValue(NAME))) ②
    .build();
```

① Set the item details renderer using text

② Set the item details renderer using a `Component`

By default, the item details are shown/hidden for each listing row **when the user clicks on a row**. This behaviour can be disabled using the `itemDetailsVisibleOnClick` builder method.

In this case, the item details content should be opened or closed programmatically. For this purpose, the `setItemDetailsVisible` method of the item listing API can be use to show or hide the item details for a specific item.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .itemDetailsText(item -> "Detail of: " + item.getValue(ID)) ①
    .itemDetailsVisibleOnClick(false) ②
    .withItemClickListener(e -> { ③
      e.getSource().setItemDetailsVisible(e.getItem(), true);
    }).build();
```

① Set the item details renderer using text

② Disable the item details display at user click

③ Add an item click listener to handle the item details display, using the `setItemDetailsVisible` method

4.7.15. Header and footer configuration

The item listing header and footer sections can be completely customized, setting the cells contents and joining two or more cells together.

The header and footer sections handlers can be obtainer either using:

- The builder API, using the `header` and `footer` methods.
- The item listing API, using the `getHeader()` and `getFooter()` methods.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .header(header -> { ①
      header.prependRow().join(ID, NAME).setText("A text");
    }).footer(footer -> { ②
      footer.appendRow().getCell(ID).ifPresent(cell -> cell.setText("ID footer"));
    }).build();

listing.getHeader().ifPresent(header -> { ③
  /* customize header */
});
listing.getFooter().ifPresent(footer -> { ④
  /* customize footer */
});
```

① Use the builder to customize the header section

② Use the builder to customize the footer section

③ Get the header section

④ Get the footer section

4.7.16. Handling row selection

The item listing supports both **single** and **multiple** row selection modes. When the listing is in multiple selection mode, a *checkbox* is shown before each listing row to allow user selection. Additionaly, a *select all* checkbox is available in the listing header first cell.

The item listing selection mode can be enabled either using:

- The builder API, using the `selectionMode` method (`NONE`, `SINGLE` or `MULTI`). The `singleSelect()` and `multiSelect()` convenience methods are provided as shorters for the single and multiple selection mode.
- The item listing API, using the `setSelectionMode` method (`NONE`, `SINGLE` or `MULTI`).

A `SelectionListener` can be added to the item listing in order to listen to item selection events and obtain the currently selected items.

Furthermore, the item listing API provides a set of methods to select or deselect one or more items programmatically.

```
PropertyListing listing = PropertyListing.builder(SUBJECT) //
    .selectionMode(SelectionMode.SINGLE) ①
    .singleSelect() ②
    .multiSelect() ③
    .withSelectionListener(event -> { ④
      Set<PropertyBox> items = event.getAllSelectedItems(); ⑤
      Optional<PropertyBox> item = event.getFirstSelectedItem(); ⑥
    }).build();

listing.setSelectionMode(SelectionMode.MULTI); ⑦
listing.addSelectionListener(event -> { ⑧
});

listing.select(myItem); ⑨
listing.deselect(myItem); ⑩
listing.deselectAll(); ⑪
```

① Set the listing selection mode

② Set the listing selection mode as single

③ Set the listing selection mode as multiple

④ Add a selection listener

⑤ Get the selected items, if any

⑥ Get the first selected item, if any

⑦ Use the listing API to change the selection mode

⑧ Add a selection listener

⑨ Programmatically select an item

⑩ Programmatically deselect an item

⑪ Deselect all items

4.7.17. Editing the listing items

The item listing supports an **edit mode**, which can be enabled using the `editable()` builder method.

When a listing is editable, a row can be setted in edit mode using the `editItem(T item)` method of the item listing API. When `editItem` is called, each cell content is replaced by an **input component** to allow to change the item property value which corresponds to each column.

The updated row item values can be saved using the `saveEditingItem()` method of the item listing API, or discarded using the `cancelEditing()` method. Then the row editor is closed, restoring the default cell contents.

A set of listeners can be added to the listing to control the editing process:

- `EditorSaveListener`: for editor save events, which corresponds to the `saveEditingItem()` method call. Can be used for example to update the data in the backend.

- `EditorCancelListener`: for editor cancel events, which corresponds to the `cancelEditing()` method call.
- `EditorOpenListener` / `EditorCloseListener`: for editor open and close events.

The component to use as input to edit each item property value can be customized using the `editorComponent(…)` builder method.

The row editor provides a **buffered** mode: when enabled, the edited values are setted into the item instance only when the `saveEditingItem()` method is called, for example using a provided *save* button. The `editorBuffered` builder method can be used to enable or disable the buffered mode.

```
PropertyListing.builder(SUBJECT) //
    .editable() ①
    .editorBuffered(true) ②
    .withComponentColumn(item -> Components.button("Edit", e -> listing.editItem(item
)) ③
    ).editorComponent(new Div( ④
        Components.button("Save", e -> listing.saveEditingItem()),
        Components.button("Cancel", e -> listing.cancelEditing())))
    .displayAsFirst() ⑤
    .add() ⑥
    .withEditorSaveListener(event -> { ⑦
      PropertyBox item = event.getItem(); ⑧
      /* update the item in backend */
    }).build();
```

① Enable the editable mode

② Set the editor in buffered mode

③ Add a new column to provide the `Edit` button for each row, using the `editItem` method to edit the current row item

④ Set the editor component for the added column: it will be displayed when the row is in edit mode, providing the `Save` and `Cancel` buttons, which invoke the `saveEditingItem()` and the `cancelEditing()` methods respectively

⑤ Configure the added column to be displayed as first column

⑥ Add the column to the listing

⑦ Add an editor save listener to update the item data in backend

⑧ The save event provides the edited item instance

Default editor components and customization

By default, when a row is turned into edit mode, the Holon PropertyRenderer architecture is used to obtain an `Input` type component for each item property to edit (See Input components and forms for information on the `Input` component API).

See the Property renderering documentation for details about the property renderers architecture.

The editor component for a listing item property (and so for a listing column), can be configured using the following builder methods:

- `editor`: to set a `Input` as editor component. See Input components and forms.
- `editorField`: to use a standard Vaadin `HasValue` component as editor component.
- `editorComponent`: to use generic `Component` as editor component.

```
PropertyListing.builder(SUBJECT) //
    .editor(NAME, Input.string().build()) ①
    .editor(NAME, property -> Input.string().build()) ②
    .editor(ID, Input.string().build(), new StringToLongConverter("")) ③
    .editorField(NAME, new TextField()) ④
    .editorField(ID, new TextField(), new StringToLongConverter("")) ⑤
    .editorComponent(ID, new Button()) ⑥
    .build();
```

① Set the editor `Input` to use for the `NAME` property/column

② Set the editor `Input` to use for the `NAME` property/column

③ Set the editor `Input` to use for the `ID` property/column, providing a `Long` to `String` and back converter

④ Set the editor `HasValue` component to use for the `NAME` property/column

⑤ Set the editor `HasValue` component to use for the `ID` property/column, providing a `Long` to `String` and back converter

⑥ Set a generic component as `ID` property/column editor component

Since the `PropertyRenderer` based architecture is used to obtain the editor components by default using the `Input` rendering type, the editor component can customized also by adding a new `PropertyRenderer` for a property definition. This ensures consistency across the application, and can be used to automatically configure the editor components for any item listing instance.

```
PropertyRendererRegistry.getDefault().forProperty(NAME, ①
    InputPropertyRenderer.create(property -> Input.stringArea().build()));

PropertyListing.builder(SUBJECT).editable().build(); ②
```

① Add a new `PropertyRenderer` to the default registry to render the `NAME` property using the provided *stringArea* type `Input` component.

② Any item listing which includes the `NAME` property will use the registered renderer to obtain the `Input` component to use as the `NAME` property/column editor

Data validation

To validate the user input when a listing row is in edit mode, the standard Holon Validator API can be used.

When the editor *save* action is performed (using the `saveEditingItem()` item listing API method), all the registered validators are invoked and the save operation is interrupted if one ore more validators fails, notifying the validation errors to the user.

1. Property level validators:

One or more validators can be bound to an item property to perform the property value validation. The property value validation is performed when the user leaves an editor component and when the editor *save* action in invoked.

Any validator bound to a property at property definition time are **automatically inherited** as item property value validators.

New validators can be added to an item property using the `withValidator` item listing builder method.

```
final StringProperty NAME = StringProperty.create("name").withValidator(Validator
.notBlank()); ①

PropertyListing.builder(SUBJECT) //
    .withValidator(NAME, Validator.max(10)) ②
    .build();
```

① The *notBlank* validator is added to the `NAME` property at definition time, so it will be inherited from the item listing editor validation

② A *max* validator is added to the `NAME` property to ensure the String value is maximum 10 characters long

2. Row level validators:

One or more row level validators can be added to an item listing using the `withValidator` builder method. The row level validators are invoked when the editor *save* action is performed, and the current item state s provided to validators to perform the required checks.

```
PropertyListing.builder(SUBJECT) //
    .withValidator(Validator.create(item -> item.getValue(ID) != null, "Id value must
be not null")) ①
    .build();
```

① Add a row level validator

Validation status handler

By default, the validation errors originated by the registered validators are notified to the user in

the following way:

- For property level validators, if the editor component supports error messages notification (using the Vaadin `HasValidation` interface), the validation errors are setted as component errors.
- For row level validators, a Dialog is used to notify the validation errors.

The validation status errors notification can be customized using the ValidationStatusHandler interface, both at property and row level.

The `ValidationStatusHandler` validation event provides the validation status (valid, invalid or unresolved) and the current validation errors, if any.

To customize both the **property level** and the **row level** validation status notification, the `validationStatusHandler` builder method can be used.

```
PropertyListing.builder(SUBJECT) //
    .validationStatusHandler(event -> { ①
      if (event.isInvalid()) {
        Notification.show("Validation falied: " + event.getErrorMessage());
      }
    }).validationStatusHandler(NAME, event -> { ②
      /* omitted */
    }).build();
```

① Set a custom row level validation status handler

② Set a custom property level validation status handler for the `NAME` property

Furthermore, the GroupValidationStatusHandler interface can be used to override all the validation status notification, customizing both the property level and the row level validation status notification using a single handler.

```
PropertyListing.builder(SUBJECT) //
    .groupValidationStatusHandler(event -> { ①
      event.getGroupStatus(); ②
      event.getInputsValidationStatus(); ③
      event.getGroupErrorMessages(); ④
      event.getInputsValidationStatus().forEach(s -> s.getErrorMessages()); ⑤
    }).build();
```

① Set the `GroupValidationStatusHandler` for the row editor

② Get the row-level validation status

③ Get the single editor components validation status

④ Get the row-level validation errors

⑤ Get the single editor components validation errors

Listening to editor components value change

When a listing row is in edit mode, one ore more `ValueChangeListener` can be added to the editor input components to listen to value change, using the `withValueChangeListener` builder method.

```
PropertyListing.builder(SUBJECT) //
    .withValueChangeListener(NAME, event -> { ①
      event.getOldValue();
      event.getValue();
    }).build();
```

① Add a value change listener for the `NAME` property

4.8. View components and forms

The ViewComponent component can be used to display data, providing consistent formatting and style across the whole UI, including localization and **internationalization** support.

The `ViewComponent` API extends:

- `ValueHolder`: it handles a typed value. See The `ValueHolder` interface.
- `HasComponent`: it can used as UI component. See The `HasComponent` interface.

A `ViewComponent` instance can be created either using the `ViewComponent` API builder methods, providing the value type or the `Components` API, through the `Components.view` sub interface (see Component builders and providers).

The `ViewComponent` builder API provides methods to configure the UI component, including methods to set a **localizable label** for the component and to add component click listeners.

```
ViewComponent<String> viewComponent = ViewComponent.builder(String.class) ①
    .fullWidth() ②
    .styleName("my-style") ③
    .label("My label") ④
    .label("My label", "label.message.code") ⑤
    .description("My description") ⑥
    .onClick(event -> { ⑦
      /* handle the click event */
    }).withValue("Initial value") ⑧
    .withValueChangeListener(event -> { ⑨
      String oldValue = event.getOldValue(); ⑩
      String newValue = event.getValue(); ⑪
    }).build();

viewComponent = ViewComponent.create(String.class); ⑫

viewComponent = Components.view.component(String.class) ⑬
    /* configuration omitted */
    .build();
```

① Get a `String` type `ViewComponent` builder

② Set the component width to 100%

③ Add a CSS style class name

④ Set the label

⑤ Set the label using a localizable message

⑥ Set the description

⑦ Add a click handler

⑧ Set an initial value

⑨ Add value change listener

⑩ The previous value

⑪ The changed value

⑫ Direct `ViewComponent` creation method

⑬ The `Components.view.component` method of the `Components` API can be used to create and configure a `ViewComponent`

A `ViewComponent` instance can be used as *value holder* to handle the value to display (see The `ValueHolder` interface).

```
ViewComponent<String> viewComponent = ViewComponent.create(String.class); ①

new VerticalLayout().add(viewComponent.getComponent()); ②

viewComponent.setValue("My value"); ③
String value = viewComponent.getValue(); ④
value = viewComponent.getValueIfPresent().orElse("Default value"); ⑤

viewComponent.clear(); ⑥
boolean empty = viewComponent.isEmpty(); ⑦
```

① Create a `String` type `ViewComponent`

② Add the component to a layout using the `getComponent()` method

③ Set the component value

④ Get the component value

⑤ Get the component optional value

⑥ Clear the component value

⑦ Check if the component is empty

4.8.1. Value conversion and formatting

A `ViewComponent` converts the value it holds to a `String` in order to display it in UI. For value conversion and formatting the `ViewComponent` API relies on the Holon Platform StringValuePresenter API.

This ensures a consistent behaviour across the application on how the values are displayed and the current `LocalizationContext` is used, if available, to apply the proper internationalization conventions to format, for example, a numeric value.

See the StringValuePresenter API documentation for details.

To provide a custom value conversion logic, a value conversion function can be provided at `ViewComponent` build time:

```
ViewComponent<Integer> viewComponent = ViewComponent.<Integer>builder(value -> String
.valueOf(value)).build(); ①
```

① Create a `Integer` type `ViewComponent`, providing the function to convert the `Integer` value to `String`

4.8.2. Using the property value presenters

When a `Property` value has to be displayed, a `ViewComponent` can be created and bound to the `Property` definition, inheriting the property value type as component type and enabling the PropertyValuePresenter based strategy to convert and format the value in a consistent way across the application.

See the Property value presentation documentation for details on property value presentation.

```
NumericProperty<Integer> MY_PROPERTY = NumericProperty.integerType("my_property"); ①

ViewComponent<Integer> viewComponent = ViewComponent.create(MY_PROPERTY); ②
```

① `Integer` type property definition

② Create a `ViewComponent` using the `MY_PROPERTY` property definition. The property value presenters will be used to convert the value to a `String` and apply formatting

This allows to **generalize the property value presentation** and make it consistent across all the application UI components which support property value presenters, including `Input` components (see Input components and forms) and listing components (see List data in grids).

New `PropertyValuePresenter` can be registered and used transparently to present the value for a specific type or a specific property configuration.

For example, suppose we want to add a `#` prefix to each value of the `MY_PROPERTY` property. We can declare a `PropertyValuePresenter` and bind it to the `MY_PROPERTY` property in the property presenters registry:

```
PropertyValuePresenterRegistry.getDefault() ①
    .forProperty(MY_PROPERTY, (property, value) -> "#" + value); ②
```

① Get the default `PropertyValuePresenterRegistry`

② Register a `PropertyValuePresenter`, bound to the `MY_PROPERTY` property, which adds a `#` prefix to the property value

From now on, any `ViewComponent` instance will use the new presenter to render the `MY_PROPERTY` property value.

```
ViewComponent<Integer> viewComponent = ViewComponent.create(MY_PROPERTY); ①

String value = MY_PROPERTY.present(1); ②
```

① The `ViewComponent` instance will use the new presenter to render the `MY_PROPERTY` property value

② Any other property presentation service will use the new presenter for the `MY_PROPERTY` property value. In this example, the `MY_PROPERTY` property is directly used to present the `1` value and the result will be the String `#1`.

4.8.3. Create a `ViewComponent` from a generic `Component`

You can use the `ViewComponent` API `adapt` static methods to transform any `Component` into a `ViewComponent`. The `Component` will be used as `ViewComponent` content and a suitable function must be provided to setup the `Component` instance each time the value changes, for example to display the

value using the custom `Component` instance.

A generic `Component` can be turn into a `ViewComponent` either providing:

1. The `Component` to use a `ViewComponent` content and a *consumer* function to setup the `Component` instance each time the value changes:

```
ViewComponent<String> viewComponent = ViewComponent.adapt(String.class, new Span(),
(span, value) -> {
  span.setText(value);
}).build(); ①
```

① A `Span` is used as `String` type `ViewComponent` content, setting the Span text with the provided value each time the `ViewComponent` value changes

2. A function to provided a custom `Component` instance each time the value changes, to be used as `ViewComponent` content:

```
ViewComponent<String> viewComponent = ViewComponent.adapt(String.class, value -> {
  if (value != null) {
    return new RouterLink(value, MyView.class, value); ①
  }
  return null;
}).build();
```

① Each time the value changes, a new `RouterLink` is created and used as `ViewComponent` content

The `adapt` methods return a builder API consistent with the default `ViewComponent` builder API, allowing to configure the `ViewComponent` component.

```
ViewComponent<String> viewComponent = ViewComponent.adapt(String.class, value -> {
  if (value != null) {
    return new RouterLink(value, MyView.class, value); ①
  }
  return null;
}) ①
    .fullWidth() ②
    .styleName("my-style") ③
    .label("My label") ④
    .label("My label", "label.message.code") ⑤
    .description("My description") ⑥
    .onClick(event -> { ⑦
      /* handle the click event */
    }).withValue("Initial value") ⑧
    .withValueChangeListener(event -> { ⑨
      String oldValue = event.getOldValue();
      String newValue = event.getValue();
    }).build();
```

① Create a `ViewComponent` which shows a `RouterLink` each time the value changes

② Set the component width to 100%

③ Add a CSS style class name

④ Set the label

⑤ Set the label using a localizable message

⑥ Set the description

⑦ Add a click handler

⑧ Set an initial value

⑨ Add value change listener

4.8.4. `ViewComponent` property renderer

When the `holon-vaadin-flow` artifact is in classpath, a default `PropertyRenderer` is automatically registered to render a `Property` as a `ViewComponent`.

See the Property renderering documentation for details about the property renderers architecture.

So the `ViewComponent.class` type can be used to render any property as a `ViewComponent`. In this scenario, the property **localizable message** is used if available as `ViewComponent` label.

```
BooleanProperty PROPERTY = BooleanProperty.create("test") ①
    .message("caption").messageCode("caption.message.code"); ②

ViewComponent<Boolean> viewComponent = PROPERTY.render(ViewComponent.class); ③
```

① Declare a boolean type property

② Set the property localizable message

③ Create `ViewComponent` instance using the property renderer. The property localizable message will be used as `ViewComponent` label.

4.8.5. Organize view components in groups and forms

A set of `ViewComponent` type components can be grouped and organized using a PropertyViewGroup, using the Holon Platform Property model to bind each `ViewComponent` to a property definition.

The property definitions will be used as **references** to get and configure the `ViewComponent` instances managed by the group and to **set the group values** to display using a `PropertyBox` type.

The `PropertyViewGroup` API provides method to inspect the available properties and the `ViewComponent` instances bound to each property. Through the `ValueHolder` API, the group value can be setted and managed (see The `ValueHolder` interface).

To create a `PropertyViewGroup`, the **property set** to use must be provided at build time. Both the

`PropertyViewGroup` API and the `Components` API (through the `Components.view` sub interface) can be used to obtain a property view group builder.

By default, the property rendering architecture is used to **automatically generate** the `ViewComponent` instances for each property of the group property set (see `ViewComponent` property renderer).

For example, given the following property model definition:

```
static final NumericProperty<Long> ID = NumericProperty.longType("id");
static final StringProperty NAME = StringProperty.create("name");
static final PropertySet<?> SUBJECT = PropertySet.of(ID, NAME);
```

A `PropertyViewGroup` can be created in the following way:

```
PropertyViewGroup group = PropertyViewGroup.builder(SUBJECT).build(); ①
group = Components.view.propertyGroup(SUBJECT).build(); ②

Collection<Property<?>> properties = group.getProperties(); ③
Stream<ViewComponent<?>> components = group.getElements(); ④
group.getBindings().forEach(binding -> { ⑤
  Property<?> property = binding.getProperty();
  ViewComponent<?> component = binding.getElement();
});

Optional<ViewComponent<?>> element = group.getElement(NAME); ⑥
ViewComponent<?> component = group.requireElement(NAME); ⑦

PropertyBox value = PropertyBox.builder(SUBJECT).set(ID, 1L).set(NAME, "One").build();

group.setValue(value); ⑧
value = group.getValue(); ⑨

group.clear(); ⑩
group.isEmpty(); ⑪
```

① Create a `PropertyViewGroup` using the `SUBJECT` property set. The `ViewComponent` for each property will be automatically generated using the property renderers

② Create a `PropertyViewGroup` from the `Components` API

③ Get the group property set

④ Get the group `ViewComponent` elements

⑤ Get the group bindings, i.e. the `ViewComponent` bound to each property

⑥ Get the `ViewComponent` bound to the `NAME` property, if available

⑦ Require the `ViewComponent` bound to the `NAME` property

⑧ Set the group value using a `PropertyBox`: each property value available from the `PropertyBox`

instance will be setted into the corresponding `ViewComponent`, if available

⑨ Get the current group value

⑩ Clear the group value

⑪ Check whether the group is empty, i.e. has no value

To customize this behaviour and provide the `ViewComponent` to use for one or more property, the `PropertyViewGroup` builder API provides a set of `bind` methods, which can be use to explicitly bind a `ViewComponent` instance to a property.

```
PropertyViewGroup group = PropertyViewGroup.builder(SUBJECT) ①
    .bind(NAME, ViewComponent.create(String.class)) ②
    .bind(NAME, property -> ViewComponent.create(String.class)) ③
    .build();
```

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

② Bind a `ViewComponent` instance to the `NAME` property

③ Bind a `ViewComponent` instance to the `NAME` property using a function

Group configuration

The `PropertyViewGroup` builder API provides method to configure the group, allowing to:

- Hide one or more property: the hidden property won't be bound to a `ViewComponent`
- Add a *post processor* to configure the `ViewComponent` instances just before the binding occurs
- Add group value change listeners

```
PropertyViewGroup group = PropertyViewGroup.builder(SUBJECT) ①
    .hidden(ID) ②
    .withPostProcessor((property, component) -> { ③
      /* ViewComponent configuration */
      component.hasStyle().ifPresent(s -> s.addClassName("my-style"));
    }) //
    .withValueChangeListener(event -> { ④
      /* handle value change */
      PropertyBox value = event.getValue();
    }).build();
```

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

② Set the `ID` property as hidden

③ Add a binding post processor

④ Add a value change listener

Using a `PropertyViewForm` to make the group a UI component

The `PropertyViewGroup` API handles a group of view components bound to a property set, but is it

not a UI component itself. The PropertyViewForm API can be used to provide a UI component for the group and use it as a *form* in application UI.

A PropertyViewGroup extends the HasComponent interface to provide the actual group UI component (see The HasComponent interface) and the Composable API to declare the UI component to use as group elements container and the strategy to apply to organize the ViewComponent group elements into the group content area.

The Composer API can be used to implement the composition logic, i.e. to add the available ViewComponent elements to the UI component declared as group content. The Composer function provides:

- The group UI content element.
- A reference to the group API to obtain the ViewComponent elements (and the corresponding property bindings) to add to the group UI content element.

A default Composer for a com.vaadin.flow.component.HasComponents type UI element can be obtain using the componentContainerComposer() method of the Composable API. This composer adds the group elements sequentially to the HasComponents group content element, in the order they are provided from the group property set.

```
PropertyViewForm form = PropertyViewForm.builder(new VerticalLayout(), SUBJECT) ①
    .composer(Composable.componentContainerComposer()) ②
    .build();

myLayout.add(form.getComponent()); ③

form.setValue(PropertyBox.builder(SUBJECT).set(ID, 1L).set(NAME, "One").build()); ④
```

① Create a PropertyViewForm using the SUBJECT property set, a VerticalLayout as group UI element

② The default component container composer is used

③ The getComponent() method can be used to add the PropertyViewForm to a layout

④ Set a form value

A set of convenience builder methods are available for the base Vaadin layouts:

```
PropertyViewForm form = PropertyViewForm.verticalLayout(SUBJECT).build(); ①
form = PropertyViewForm.horizontalLayout(SUBJECT).build(); ②
form = PropertyViewForm.formLayout(SUBJECT).build(); ③

// Using the Components API:
form = Components.view.formVertical(SUBJECT).build();
form = Components.view.formHorizontal(SUBJECT).build();
form = Components.view.form(SUBJECT).build();
```

① Create a PropertyViewForm using the SUBJECT property set and a VerticalLayout as group UI element

② Create a `PropertyViewForm` using the `SUBJECT` property set and a `HorizontalLayout` as group UI element

③ Create a `PropertyViewForm` using the `SUBJECT` property set and a `FormLayout` as group UI element

A custom `Composer` can be used to control and customize the group element composition strategy:

```
PropertyViewForm form = PropertyViewForm.verticalLayout(SUBJECT) //
    .composer((content, group) -> { ①
      group.getBindings().forEach(binding -> {
        content.add(binding.getElement().getComponent());
      });
    }).build();
```

① Provide a custom composition logic

Form configuration

A `PropertyViewForm` can be configured using the builder API in order to:

- Hide one or more property: the hidden property won't be bound to a `ViewComponent`
- Add an *initializer* to configure the form UI content
- Add a *post processor* to configure the `ViewComponent` instances just before the binding occurs
- Add form value change listeners
- Configure the property *captions*, i.e. to set the label to use for the `ViewComponent` bound to a property

```
PropertyViewForm.formLayout(SUBJECT) ①
    .hidden(ID) ②
    .initializer(content -> { ③
      /* content configuration */
      FormLayout fl = content;
    }).withPostProcessor((property, component) -> { ④
      /* ViewComponent configuration */
      component.hasStyle().ifPresent(s -> s.addClassName("my-style"));
    }) //
    .withValueChangeListener(event -> { ⑤
      /* handle value change */
      PropertyBox value = event.getValue();
    }) //
    .hidePropertyCaption(ID) ⑥
    .propertyCaption(NAME, "My name", "name.message.code") ⑦
    .build();
```

① Create a `PropertyViewForm` using the `SUBJECT` property set and a `FormLayout`

② Set the `ID` property as hidden

③ Set the form content initializer

④ Set the binding post processor

⑤ Add a value change listener

⑥ Hide the `ID` property caption

⑦ Set the localizable caption for the `NAME` property

4.9. Input components and forms

The Input API can be used to create UI components to handle **user input**, ensuring a consistent data type handling and input format patterns.

The `Input` API extends:

- `ValueHolder`: it handles a typed value. See The `ValueHolder` interface.
- `HasComponent`: it can used as UI component. See The `HasComponent` interface.

Furthermore, the `Input` API provides methods to set the UI component in **read only** mode and to display required indicators and validation errors.

A `Input` instance can be created either using the `Input` API builder methods, providing the value type, or the `Components` API, through the `Components.input` sub interface (see Component builders and providers).

The `Input` builder API provides methods to configure the UI component, including methods to set a localizable label for the component and a complete set of configuration methods according to the input value type.

```
Input<String> input = Input.string() ①
    .fullWidth() ②
    .styleName("my-style") ③
    .label("My label") ④
    .label("My label", "label.message.code") ⑤
    .description("My description") ⑥
    .blankValuesAsNull(true) ⑦
    .autocapitalize(Autocapitalize.WORDS) ⑧
    .maxLength(50) ⑨
    .placeholder("My placeholder", "placeholder.message.code")⑩
    .autofocus(true)⑪
    .prefixComponent(new Button("Prefix"))⑫
    .tabIndex(99) ⑬
    .readOnly() ⑭
    .withBlurListener(event -> {⑮
      /* handle blur event */
    }).withValueChangeListener(event -> { ⑯
      String oldValue = event.getOldValue(); ⑰
      String newValue = event.getValue(); ⑱
    }).build();

input = Components.input.string() ⑬
    /* configuration omitted */
    .build();
```

① Get a `String` type `Input` builder

② Set the width to 100%

③ Add a CSS style class name

④ Set the input label

⑤ Set the input label using a localizable message

⑥ Set the input description

⑦ Set to treat blank values as `null` values.

⑧ Set the auto-capitalize mode

⑨ Set the max characters count allowed for input

⑩ Set the input placeholder using a localizable message

⑪ Enable auto focus

⑫ Set the prefix component

⑬ Set the input tab index

⑭ Set the input as read only, preventing the user from changing the input value

⑮ Add a blur (focus lost) listener

⑯ Add value change listener

⑰ The previous value

⑱ The changed value

A set of `Input` builders for the most common types are available both form the `Input` API and from the `Components.input` API.

Each `Input` is rendered using a suitable UI component according to the input value type. For example, a `TextField` is used for String type inputs and a `Checkbox` for boolean type inputs.

```
Input<String> input1 = Input.string().build(); ①
Input<String> input2 = Input.stringArea().build(); ②
Input<Boolean> input3 = Input.boolean_().build(); ③
Input<Double> input4 = Input.number(Double.class).build(); ④
Input<LocalDate> input5 = Input.localDate().build(); ⑤
Input<LocalDateTime> input6 = Input.localDateTime().build(); ⑥
Input<LocalTime> input7 = Input.localTime().build(); ⑦
Input<Date> input8 = Input.date().build(); ⑧

Optional<Input<String>> input = Input.create(String.class); ⑨
```

① Create an `Input` for `String` type values

② Create an `Input` for `String` type values using a *text area* to render the input component

③ Create an `Input` for `Boolean` type values

④ Create an `Input` for a numeric type value, a `Double` in this case

⑤ Create an `Input` for `LocalDate` type values

⑥ Create an `Input` for `LocalDateTime` type values

⑦ Create an `Input` for `LocalTime` type values

⑧ Create an `Input` for `Date` type values

⑨ Create an `Input` for the given value type, if a suitable implementation is available

The `Components.input` API provides the same methods to obtain the input builders.

A `Input` instance can be used as *value holder* to handle the input value (see The `ValueHolder` interface).

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

myLayout.add(input.getComponent()); ②

input.setValue("My value"); ③
String value = input.getValue(); ④
value = input.getValueIfPresent().orElse("Default value"); ⑤

input.clear(); ⑥
boolean empty = input.isEmpty(); ⑦

input.setReadOnly(true); ⑧
input.isReadOnly(); ⑨

input.addValueChangeListener(event -> { ⑩
  /* handle value change */
});
```

① Create a `String` type `Input`

② Add the input component to a layout using the `getComponent()` method

③ Set the input value

④ Get the input value

⑤ Get the input optional value

⑥ Clear the input value

⑦ Check if the input is empty (i.e. has no value)

⑧ Set the input as read only

⑨ Check whether the input is read only

⑩ Add a value change listener

4.9.1. `Input` property renderer

When the `holon-vaadin-flow` artifact is in classpath, a default `PropertyRenderer` is automatically registered to render a `Property` as an `Input`.

See the Property renderering documentation for details about the property renderers architecture.

So the `Input.class` type can be used to render any property as a `Input`. In this scenario, the property **localizable message** is used if available as `Input` label.

The `Input` API makes available the `create(Property<T> property)` method to directly create an `Input` for a given property.

```
NumericProperty<Integer> MY_PROPERTY = NumericProperty.integerType("my_property"); ①

Input<Integer> input = Input.create(MY_PROPERTY); ②

Input<Integer> viewComponent = MY_PROPERTY.render(Input.class); ③
```

① Declare a `Integer` type property

② Create a `Input` to handle the `MY_PROPERTY` property value using the `Input` API

③ Create a `Input` to handle the `MY_PROPERTY` property value using the property renderer for the `Input` class

The property renderers architecture is used **consistently across the application UI**, so when a renderer is bound to a specific property condition it is used by all the UI components with property rendering support, including input groups/forms (see Organize `Input` components in groups and forms) and item listing editors (see Editing the listing items).

For example, suppose we want to use a custom `MyInput` class to render the `Input` component for the `MY_PROPERTY` property definition. A custom property renderer can be created and registered:

```
PropertyRendererRegistry.getDefault() ①
    .forProperty(MY_PROPERTY, InputPropertyRenderer.create(property -> new MyInput())
); ②
```

① Get the default `PropertyRendererRegistry`

② Register a `InputPropertyRenderer`, bound to the `MY_PROPERTY` property, to provide a `MyInput` instance

From now on, the `Input` component for the `MY_PROPERTY` property definition will be a `MyInput` instance.

4.9.2. Input value conversion

The `Input` API provides methods to use an `Input` with a different type than the value type to handle, providing a suitable `com.vaadin.flow.data.converter.Converter` to perform the value conversions to the required type.

```
Input<String> stringInput = Input.string().build(); ①
Input<Integer> integerInput = Input.from(stringInput, new StringToIntegerConverter(
"Conversion failed")); ②
```

① Create a `String` type `Input`

② Create a `Integer` type `Input` from the previous one, providing the String to Integer and back converter

A Holon Platform `PropertyValueConverter` can also be use to provide the value conversion logic when the input is bound to a `Property` definition.

```
BooleanProperty PROPERTY = BooleanProperty.create("test");

Input<Integer> integerInput = Input.number(Integer.class).build(); ①
Input<Boolean> booleanInput = Input.from(integerInput, PROPERTY,
    PropertyValueConverter.numericBoolean(Integer.class)); ②
```

① Create a `Integer` type `Input`

② Create a `Boolean` type `Input` for the `PROPERTY` property, using the previous input and a *numeric boolean* `PropertyValueConverter`

4.9.3. Input adapters for `HasValue` components

The `Input` API provides methods to *adapt* a standard Vaadin `HasValue` type input component as an `Input`, including value conversion support using a `com.vaadin.flow.data.converter.Converter`.

```
Input<String> stringInput = Input.builder(new TextField()).build(); ①

Input<Integer> input = Input.from(new TextField(), new StringToIntegerConverter(
"Conversion failed")); ②
```

① Create a `String` type `Input` from a `TextField`

② Create a `Integer` type `Input` from a `TextField`, providing the value converter

4.9.4. Select type inputs

The SingleSelect and MultiSelect APIs are `Input` extensions which represent *selectable* input components, i.e. input components which provide a **set of items** from which the input value can be selected, either supporting **single or multiple** selection values.

The `Input` API and the corresponding Components.input API provides methods to create `SingleSelect` and `MultiSelect` type input components, providing the **selectable value type**.

For `SingleSelect` input types, the input component can be rendered as:

- **Simple select**: a Vaadin `Select` component is used as input component.
- **Filterable select**: a Vaadin `ComboBox` component is used as input component.
- **Options select**: a `RadioButtonGroup` component is used as input component.

```
SingleSelect<String> singleSelect = Input.singleSelect(String.class).build(); ①
singleSelect = Input.singleOptionSelect(String.class).build(); ②

MultiSelect<String> multiSelect = Input.multiOptionSelect(String.class).build(); ③
```

① Create a `String` type `SingleSelect` input

② Create a `String` type `SingleSelect` input using the *options* rendering mode

③ Create a `String` type `MultiSelect` input

The select input components can be used just like any other *value holder* type component to set and get the selected item/s (see The `ValueHolder` interface). The `MultiSelect` type input, since it allows multiple items selection, uses a `java.util.Set` to handle the input value.

The `Components.input` API provides the same methods to obtain the select type input builders.

```
SingleSelect<String> singleSelect = Input.singleSelect(String.class).build();

singleSelect.setValue("A value"); ①
String value = singleSelect.getValue(); ②
singleSelect.clear(); ③
singleSelect.isEmpty(); ④

MultiSelect<String> multiSelect = Input.multiOptionSelect(String.class).build();

multiSelect.setValue(Arrays.asList("Value 1", "Value 2").stream().collect(Collectors
.toSet())); ⑤
Set<String> values = multiSelect.getValue(); ⑥
```

① Set the input value, i.e. select the given value

② Get the input (selected) value

③ Clear the input value (deselect any value)

④ Check whether the input is empyt (no value is selected)

⑤ Set the input values, i.e. select the given values, using a `Set`

⑥ Get the input (selected) values as a `Set`

Besides the `Input` API, the selectable inputs implements the Selectable API, which allows to manage and obtain the selected items.

```
SingleSelect<String> singleSelect = Input.singleSelect(String.class).build();

singleSelect.select("A value"); ①
String value = singleSelect.getSelectedItem().orElse(null); ②
singleSelect.deselectAll(); ③
```

① Select given value

② Get the selected value

③ Deselect any value

Select items data source

A select type `Input`, whether it's a `SingleSelect` or a `MultiSelect`, requires an **item set** to be configured, which represents the source for the selectable values. Only the available items will be

used as valid selectable input values.

The builder API provides the `items` and `addItem` methods to provides the selection items set.

```
SingleSelect<String> singleSelect = Input.singleSelect(String.class) //
    .addItem("Value 1") ①
    .addItem("Value 2") ②
    .items("Value 1", "Value 2") ③
    .build();
```

① Add a selection item

② Add another selection item

③ Set the selection items, replacing any previous item set

A standard Vaadin `DataProvider` can be used as items data source.

```
SingleSelect<String> singleSelect = Input.singleSelect(String.class) //
    .dataSource(DataProvider.ofItems("Value 1", "Value 2")) ①
    .build();
```

① Use a `DataProvider` as items data source

When **the item type does not match the selection value type**, a ItemConverter con be used to provide the item to selection value and back conversion logic.

```
SingleSelect<String> singleSelect = Input.singleSelect(String.class, MyBean.class,
ItemConverter.create( ①
    item -> item.getName(), ②
    value -> Optional.of(new MyBean(value)) ③
)).addItem(new MyBean("One")).addItem(new MyBean("Two")).build();
```

① Create a `String` type `SingleSelect` which uses a `MyBean` class as items type, providing a suitable `ItemConverter`

② Convert a `MyBean` type item into a `String` type selection value

③ Convert a `String` type selection value into a `MyBean` type item

Use the `Datastore` API as items data source

When the property model is used to build a select type input, **the selection items are represented using the `PropertyBox` type by default**, and the property which represents the selection value has to be declared a construction time.

The select input type will match the **selection property type** and the value provided as selection value will be obtained using the selection property definition from the `PropertyBox` type item instances.

For example, given the following property model definition:

```
static final NumericProperty<Long> ID = NumericProperty.longType("id");
static final StringProperty NAME = StringProperty.create("name");
static final PropertySet<?> SUBJECT = PropertySet.of(ID, NAME);
```

A `SingleSelect` to select the `ID` property can be created as follows:

```
SingleSelect<Long> singleSelect = Input.singleSelect(ID) ①
    .addItem(PropertyBox.builder(SUBJECT).set(ID, 1L).set(NAME, "One").build()) ②
    .build();
```

① Create a `SingleSelect` using the `ID` property as selection property: the select input type will be `Long` like the property type

② Add a `PropertyBox` type selection item

When a property model bound select is used, the most simple and powerful way to provided the selection items is to use a `Datastore` as data source. The `Datastore` API makes easy and straightful to bind the select input to a backend data source.

When a `Datastore` is used, a `DataTarget` definition is required to declare the data model *entity* to use as items data source (for example a RDBMS table name, or a collection name in a document based database).

A `Datastore` based data source can be configured using a standard DataProvider, through the `DatastoreDataProvider` API: see the Datastore API integration section to learn how to create and configure a `Datastore` based `DataProvider`.

Furthermore, the select input builder APIs provides convenience methods to set and configure a data source using a `Datastore`.

By default, **only the selection property is used as items property set**. To specify a different property set (i.e. a different query projection), the property set must be provided at data source configuration time.

For example, given the following `DataTarget` definition;

```
static final DataTarget<?> TARGET = DataTarget.named("subjects");
```

The select of the previous example can be configured as follows:

```
Datastore datastore = getDatastore();

SingleSelect<Long> singleSelect = Input.singleSelect(ID) ①
    .dataSource(datastore, TARGET) ②
    .build();

singleSelect = Input.singleSelect(ID) //
    .dataSource(datastore, TARGET, SUBJECT) ③
    .build();
```

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

② Use a `Datastore` as items data source, providing the `DataTarget` to use

③ Declare to use the `SUBJECT` property set as items property set (hence as query projection)

The `Datastore` based data source can be configured, for example to **add query filters and sorts**. The configuration methods provided by the builder API corresponds to the ones provided by the `DatastoreDataProvider` API. See `Datastore` Query configuration for details.

```
SingleSelect<Long> singleSelect = Input.singleSelect(ID) ①
    .dataSource(getDatastore(), TARGET, SUBJECT) ②
    .withQueryFilter(NAME.isNotNull()) ③
    .withQuerySort(ID.asc()) ④
    .withQueryConfigurationProvider(myQueryConfigurationProvider) ⑤
    .build();
```

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

② Use a `Datastore` as items data source, providing the `DataTarget` to use

③ Add a fixed query filter

④ Add a fixed query sort

⑤ Add a `QueryConfigurationProvider` reference to provide dynamic query filters and sorts

Filterable select inputs

When a `SingleSelect` in the **filterable select** mode is used, since it is rendered using a `ComboBox` component, the selection items to select can be *suggested* by the input component, **using the user input to filter the items set**.

When a standard `DataProvider` is used, the `filteringBy*` methods can be used to provide the items filtering strategy.

```
SingleSelect<String> input = Input.singleSelect(String.class) ①
    .dataSource(DataProvider.ofCollection(Arrays.asList("One", "Two", "Three")) ②
        .filteringByPrefix(v -> v)) ③
    .build();
```

① Create a `String` type `SingleSelect`

② Use a `DataProvider` as items data source

③ Set the filtering mode, in this case checking whether the item value starts with the query value

When a `Datastore` based data source is used (see Use the `Datastore` API as items data source), a function can be configured to transform the text typed by the user into the `QueryFilter` to add to the Datastore query to filter the returned items.

The filter converter function can be configured either using:

1. The `dataSource` builder method version which accepts the filter converter function:

```
SingleSelect<Long> singleSelect = Input.singleSelect(ID) ①
    .dataSource(getDatastore(), TARGET, ②
        text -> NAME.contains(text), ③
        SUBJECT ④
    ).build();
```

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

② Use a `Datastore` as items data source, providing the `DataTarget` to use

③ Provide the function to convert the user typed text value to a `QueryFilter`

④ Use the `SUBJECT` property set as items property set

1. The `filterConverter` builder method:

```
SingleSelect<Long> singleSelect = Input.singleSelect(ID) ①
    .dataSource(getDatastore(), TARGET, SUBJECT) ②
    .filterConverter(text -> NAME.contains(text)) ③
    .build();
```

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

② Use a `Datastore` as items data source, providing the `DataTarget` to use and `SUBJECT` as items property set

③ Provide the function to convert the user typed text value to a `QueryFilter`

Use select inputs with enumerations

The `Input` API and the `Components.input` API provides convenience methods to create **enumeration** based select inputs, automatically adding all the available enumeration values to the selection items set.

A `enum` type select can be created both as a `SingleSelect` and as `MultiSelect`.

For example, given the following `enum` class:

```
enum MyEnum {

  FIRST, SECOND, THIRD;

}
```

A select type input for the enumeration values can be created as follows:

```
SingleSelect<MyEnum> singleSelect = Input.enumSelect(MyEnum.class).build(); ①
singleSelect = Input.enumOptionSelect(MyEnum.class).build(); ②

MultiSelect<MyEnum> multiSelect = Input.enumMultiSelect(MyEnum.class).build(); ③
```

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

② Create a `SingleSelect` for the `MyEnum` enumeration using the *options* rendering mode

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

The Caption annotation can be used on the enumeration class values to configure the select items *caption*, with localizable messages support.

```
enum MyEnum {

  @Caption(value = "The first", messageCode = "first.message.code")
  FIRST,

  @Caption(value = "The second", messageCode = "second.message.code")
  SECOND,

  @Caption(value = "The third", messageCode = "third.message.code")
  THIRD;

}
```

See the next section to learn how to configure the select items caption using the select input builder API.

Selection items caption

By default, the selectable items are displayed in UI **using the `toString()` representation of the item itself**.

To change this behaviour, the item *captions* can be configured either setting them explicitly using the builder API (through the `itemCaption` method), or using a `ItemCaptionGenerator` function to provide the caption for each selection item.

Example using explicit item captions:

```
SingleSelect<Integer> singleSelect = Input.singleSelect(Integer.class) ①
    .items(1, 2) //
    .itemCaption(1, "One") ②
    .itemCaption(2, "One", "two.message.code") ③
    .build();
```

① Create a `Integer` type `SingleSelect`

② Set the `1` item caption

③ Set the `2` item caption using a localizable message

Example using a `ItemCaptionGenerator`:

```
SingleSelect<Long> singleSelect = Input.singleSelect(ID) ①
    .dataSource(getDatastore(), TARGET, SUBJECT) ②
    .itemCaptionGenerator(item -> item.getValue(NAME)) ③
    .build();
```

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

② Use `SUBJECT` as items property set, which includes the `NAME` property

③ Set the `ItemCaptionGenerator`: the `NAME` property value is used as item caption

For `Property` bound selectable inputs, a convenience `itemCaptionProperty` builder method is available to provide the property to use to obtain the item value which will be used as items caption.

```
SingleSelect<Long> singleSelect = Input.singleSelect(ID) ①
    .dataSource(getDatastore(), TARGET, SUBJECT) ②
    .itemCaptionProperty(NAME) ③
    .build();
```

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

② Use `SUBJECT` as items property set, which includes the `NAME` property

③ The `NAME` property value is used as item caption

4.9.5. Extend the Input API using adapters

The `Input` API can be extended using custom *adapters*, i.e. a class/interface to provide additional Input attributes and/or functionalities.

An adapter can be registered at `Input` build time using the `withAdapter` method, providing:

- The adapter `Class`
- A `Function` to provide the adapter instance when requested, with the `Input` instance as function argument.

An adapter instance can be obtained using the `Input` API method `<A> Optional<A> as(Class<A> type)`. If an adapter of the given `type` is available (i.e. was registered as described above), the adapter function will be invoked to provide the adapter instance.

```
Input<String> input = Input.string().withAdapter(MyExtension.class, i -> new
MyExtension()) ①
    .build();

Optional<MyExtension> extension = input.as(MyExtension.class); ②
```

① Register and adapter for the `MyExtension` type

② Get the `MyExtension` type instance, if available

4.9.6. Validatable inputs

The value of any `Input` component can be **validated** using the standard Holon Platform Validator API, through the ValidatableInput API.

An `Input` can be made *validatable*, adding support for `Validator` registration, using:

1. The `validatable` builder method:

```
ValidatableInput<String> validatableInput = Input.string() //
    .validatable() ①
    .withValidator(Validator.max(10)) ②
    .validateOnValueChange(true) ③
    .build();
```

① Make the input validatable

② Add a input value `Validator`

③ Set whether to automatically validate the input value when it changes

2. The `ValidatableInput` adapter:

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

ValidatableInput<String> validatableInput = ValidatableInput.builder(input) ①
    .withValidator(Validator.max(10)) ②
    .build();

validatableInput = ValidatableInput.from(input); ③
validatableInput.addValidator(Validator.max(10)); ④
```

① Use the `ValidatableInput` builder method to create and configure a validatable input from a standard `Input`

② Add a input value `Validator`

③ Use the `ValidatableInput` *from* method to directly create a validatable input from a standard `Input`

④ Add a input value `Validator` using the `addValidator` method

A `ValidatableInput` extends the Validatable API, which provides method to **validate the input value** using the registered validators.

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

validatableInput.isValid(); ②

try {
  validatableInput.validate(); ③
} catch (ValidationException e) {
  /* value is not valid */
}

Optional<String> value = validatableInput.getValueIfValid(); ④
```

① Create a `ValidatableInput`

② Check whether the input value is valid

③ Validate the input value, throwing a `ValidationException` if the validation fails

④ Get the input value only if it is valid

By default, the **validation errors** are notified to the user using the UI component error message, if supported by the concrete input component.

The validation errors notification strategy can be customized using a `ValidationStatusHandler` function. The validation event provides the validation status (valid, invalid or unresolved) and the current validation errors, if any.

```
ValidatableInput<String> validatableInput = Input.string().validatable() //
    .validationStatusHandler(event -> { ①
      Status validationStatus = event.getStatus();
      String error = event.getErrorMessage();
      /* notify the validation errors */
    }).build();
```

① Set the `ValidationStatusHandler` to use for the `ValidatableInput`

the Validatable API provides a set of `adapt` methods to use a Vaadin `com.vaadin.flow.data.binder.Validator` as a Holon Platform `Validator`.

4.9.7. Organize `Input` components in groups and forms

A set of `Input` components can be grouped and organized using a PropertyInputGroup, using the Holon Platform Property model to bind each `Input` to a property definition.

The property definitions will be used as **references** to get and configure the `Input` instances managed by the group and to **get and set the group values** using a `PropertyBox` type.

The `PropertyInputGroup` API provides method to inspect the available properties and the `Input` instances bound to each property. Through the `ValueHolder` API, the group value can be setted and managed (see The `ValueHolder` interface).

When a `PropertyBox` type value is setted for the group, each `Input` value bound to a property is setted according to the property value provided by the `PropertyBox` instance, if available.

When a the group value is obtained using the `getValue()` method, the current `Input` values are reflected into the `PropertyBox` instance, using the property to which each `Input` is bound.

To create a `PropertyInputGroup`, the **property set** to use must be provided at build time. Both the `PropertyInputGroup` API and the `Components` API (through the `Components.input` sub interface) can be used to obtain a `PropertyInputGroup` builder.

By default, the property rendering architecture is used to **automatically generate** the `Input` components for each property of the group property set (see `Input` property renderer).

For example, given the following property model definition:

```
static final NumericProperty<Long> ID = NumericProperty.longType("id");
static final StringProperty NAME = StringProperty.create("name");
static final PropertySet<?> SUBJECT = PropertySet.of(ID, NAME);
```

A `PropertyInputGroup` can be created in the following way:

```
PropertyInputGroup group = PropertyInputGroup.builder(SUBJECT).build(); ①
group = Components.input.propertyGroup(SUBJECT).build(); ②

Collection<Property<?>> properties = group.getProperties(); ③
Stream<Input<?>> components = group.getElements(); ④
group.getBindings().forEach(binding -> { ⑤
  Property<?> property = binding.getProperty();
  Input<?> component = binding.getElement();
});

Optional<Input<?>> element = group.getElement(NAME); ⑥
Input<?> component = group.requireElement(NAME); ⑦

PropertyBox value = PropertyBox.builder(SUBJECT).set(ID, 1L).set(NAME, "One").build();

group.setValue(value); ⑧
value = group.getValue(); ⑨

group.clear(); ⑩
group.isEmpty(); ⑪
```

① Create a `PropertyInputGroup` using the `SUBJECT` property set. The `Input` for each property will be automatically generated using the property renderers

② Create a `PropertyInputGroup` from the `Components` API

③ Get the group property set

④ Get the group `Input` elements

⑤ Get the group bindings, i.e. the `Input` bound to each property

⑥ Get the `Input` bound to the `NAME` property, if available

⑦ Require the `Input` bound to the `NAME` property

⑧ Set the group value using a `PropertyBox`: each property value available from the `PropertyBox` instance will be setted into the corresponding `Input`, if available

⑨ Get the current group value: the current `Input` values (which in the meanwhile could have been changed by the user) are updated into the `PropertyBox` instance, according to the property to which each `Input` is bound

⑩ Clear the group value

⑪ Check whether the group is empty, i.e. has no value

To customize this behaviour and provide the `Input` component to use for one or more property, the `PropertyInputGroup` builder API provides a set of `bind` methods, which can be use to explicitly bind a `Input` instance to a property.

```
PropertyInputGroup group = PropertyInputGroup.builder(SUBJECT) ①
    .bind(NAME, Input.string().build()) ②
    .bind(NAME, property -> Input.stringArea().build()) ③
    .build();
```

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

② Bind a `Input` instance to the `NAME` property

③ Bind a `Input` instance to the `NAME` property using a function

Input group configuration

The `PropertyInputGroup` builder API provides method to configure the group, allowing to:

- Hide one or more property: the hidden property won't be bound to a `Input`
- Add a *post processor* to configure the `Input` instances just before the binding occurs
- Provide a **default value** for one or more `Input` components
- Add group value change listeners

```
PropertyInputGroup group = PropertyInputGroup.builder(SUBJECT) ①
    .hidden(ID) ②
    .defaultValue(NAME, () -> "(no name)") ③
    .withPostProcessor((property, component) -> { ④
      /* Inputs configuration */
      component.hasStyle().ifPresent(s -> s.addClassName("my-style"));
    }) //
    .withValueChangeListener(event -> { ⑤
      /* handle value change */
      PropertyBox value = event.getValue();
    }).build();
```

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

② Set the `ID` property as hidden

③ Set the default value supplier for the `NAME` property

④ Add a binding post processor

⑤ Add a value change listener

Using a `PropertyInputForm` to make the group a UI component

The `PropertyInputForm` API handles a group of `Input` components bound to a property set, but it is not a UI component itself. The PropertyInputForm API can be used to provide a UI component for the group and use it as a *form* in application UI.

A `PropertyInputForm` extends the `HasComponent` interface to provide the actual group UI component (see The `HasComponent` interface) and the Composable API to declare the UI component to use as group elements container and the strategy to apply to organize the `Input` group elements into the group content area.

The `Composer` API can be used to implement the composition logic, i.e. to add the available `ViewComponent` elements to the UI component declared as group content. The `Composer` function provides:

- The group UI content element.
- A reference to the group API to obtain the `Input` elements (and the corresponding property bindings) to add to the group UI content element.

A default `Composer` for a `com.vaadin.flow.component.HasComponents` type UI element can be obtain using the `componentContainerComposer()` method of the `Composable` API. This composer adds the group elements sequentially to the `HasComponents` group content element, in the order they are provided from the group property set.

```
PropertyInputForm form = PropertyInputForm.builder(new VerticalLayout(), SUBJECT) ①
    .composer(Composable.componentContainerComposer()) ②
    .build();

myLayout.add(form.getComponent()); ③

form.setValue(PropertyBox.builder(SUBJECT).set(ID, 1L).set(NAME, "One").build()); ④
```

① Create a `PropertyInputForm` using the `SUBJECT` property set, a `VerticalLayout` as group UI element

② The default component container composer is used

③ The `getComponent()` method can be used to add the `PropertyInputForm` to a layout

④ Set a form value

A set of convenience builder methods are available for the base Vaadin layouts:

```
PropertyInputForm form = PropertyInputForm.verticalLayout(SUBJECT).build(); ①
form = PropertyInputForm.horizontalLayout(SUBJECT).build(); ②
form = PropertyInputForm.formLayout(SUBJECT).build(); ③

// Using the Components API:
form = Components.input.formVertical(SUBJECT).build();
form = Components.input.formHorizontal(SUBJECT).build();
form = Components.input.form(SUBJECT).build();
```

① Create a `PropertyInputForm` using the `SUBJECT` property set and a `VerticalLayout` as group UI element

② Create a `PropertyInputForm` using the `SUBJECT` property set and a `HorizontalLayout` as group UI element

③ Create a `PropertyInputForm` using the `SUBJECT` property set and a `FormLayout` as group UI element

A custom `Composer` can be used to control and customize the group element composition strategy:

```
PropertyInputForm form = PropertyInputForm.verticalLayout(SUBJECT) //
    .composer((content, group) -> { ①
      group.getBindings().forEach(binding -> {
        content.add(binding.getElement().getComponent());
      });
    }).build();
```

① Provide a custom composition logic

Form configuration

A `PropertyInputForm` can be configured using the builder API in order to:

- Hide one or more property: the hidden property won't be bound to a `Input`

- Add an *initializer* to configure the form UI content
- Add a *post processor* to configure the `Input` instances just before the binding occurs
- Add form value change listeners
- Configure the property *captions*, i.e. to set the label to use for the `Input` bound to a property

```
PropertyInputForm.formLayout(SUBJECT) ①
    .hidden(ID) ②
    .initializer(content -> { ③
      /* content configuration */
      FormLayout fl = content;
    }).withPostProcessor((property, component) -> { ④
      /* Inputs configuration */
      component.hasStyle().ifPresent(s -> s.addClassName("my-style"));
    }) //
    .withValueChangeListener(event -> { ⑤
      /* handle value change */
      PropertyBox value = event.getValue();
    }) //
    .hidePropertyCaption(ID) ⑥
    .propertyCaption(NAME, "My name", "name.message.code") ⑦
    .build();
```

① Create a `PropertyInputForm` using the `SUBJECT` property set and a `FormLayout`

② Set the `ID` property as hidden

③ Set the form content initializer

④ Set the binding post processor

⑤ Add a value change listener

⑥ Hide the `ID` property caption

⑦ Set the localizable caption for the `NAME` property

Input group and form validation

The `PropertyInputGroup` and the `PropertyInputForm` APIs provides `Input` validation support using the standard Holon Platform Validator API. The value validation is supported:

- At **property level**, to validate the single `Input` value for each property.
- At **group level**, to validate the overall `PropertyBox` type group value.

The validators can be added using the `PropertyInputGroup` and the `PropertyInputForm` builder APIs.

```
PropertyInputForm.formLayout(SUBJECT) ①
    .withValidator(NAME, Validator.max(10)) ②
    .withValidator(propertyBox -> { ③
      /* group value validation */
    }).validateOnValueChange(true) ④
    .build();
```

① Create a `PropertyInputForm` using the `SUBJECT` property set

② Add a validator for the `Input` component bound to the `NAME` property

③ Add a group value validator

④ Set whether to validate the `Input` values when each `Input` value changes

The `PropertyInputGroup` and the `PropertyInputForm` APIs provides a set of methods to perform the group/form validation, a part of which is inherited from the Validatable API.

```
PropertyInputForm form = PropertyInputForm.formLayout(SUBJECT).build();

form.isValid(); ①
try {
  form.validate(); ②
} catch (ValidationException e) {
  /* value is not valid */
}

Optional<PropertyBox> valueIfValid = form.getValueIfValid(); ③
PropertyBox value = form.getValue(); ④
value = form.getValue(false); ⑤
```

① Check whether the form is valid

② Validate the form, firing both property level and group level validators

③ Get the form value, returning an empty `Optional` if validation fails

④ By default, when the form value is requested, the form validation is performed

⑤ Get the form value skipping validation

Input groups validation status handler

By default, for the `PropertyInputGroup` and the `PropertyInputForm` components, the validation errors originated by the registered validators are notified to the user in the following way:

- For **property level** validators, if the `Input` component supports error messages notification (using the Vaadin `HasValidation` interface), the validation errors are setted as component errors.
- For **group level** validators, a `Dialog` is used to notify the validation errors.

The validation status errors notification can be customized using the ValidationStatusHandler interface, both at property and group level.

The `ValidationStatusHandler` validation event provides the validation status (valid, invalid or unresolved) and the current validation errors, if any.

To customize both the **property level** and the **group level** validation status notification, the `validationStatusHandler` builder method can be used.

```
PropertyInputForm form = PropertyInputForm.formLayout(SUBJECT) //
    .validationStatusHandler(event -> { ①
      if (event.isInvalid()) {
        Notification.show("Validation falied: " + event.getErrorMessage());
      }
    }).validationStatusHandler(NAME, event -> { ②
      /* omitted */
    }).build();
```

① Set a custom group level validation status handler

② Set a custom property level validation status handler for the `NAME` property

Furthermore, the GroupValidationStatusHandler interface can be used to override all the validation status notification, customizing both the property level and the group level validation status notification using a single handler.

```
PropertyInputForm form = PropertyInputForm.formLayout(SUBJECT) //
    .groupValidationStatusHandler(event -> { ①
      event.getGroupStatus(); ②
      event.getInputsValidationStatus(); ③
      event.getGroupErrorMessages(); ④
      event.getInputsValidationStatus().forEach(s -> s.getErrorMessages()); ⑤
    }).build();
```

① Set the `GroupValidationStatusHandler` for the form

② Get the group level validation status

③ Get the single `Input` validation status

④ Get the group level validation errors

⑤ Get the single `Input` validation errors

5. Vaadin session scope

When the `holon-vaadin-flow` artifact is in classpath, a Holon Platform Context *scope* bound to the current **Vaadin session** is automatically registered and available to manage context resources.

The VaadinSessionScope API can be used to obtain the **scope name**. The `VaadinSessionScope` API also provides convenience methods to access the Vaadin session scope.

The Vaadin session **attributes** are used as context resource registry and the a current `VaadinSession` must be available in order for the scope to work.

```
Optional<ContextScope> scope = Context.get().scope(VaadinSessionScope.NAME); ①
scope = VaadinSessionScope.get(); ②
ContextScope sessionScope = VaadinSessionScope.require(); ③
```

① Get the session scope using the scope name

② Get the session scope using the `VaadinSessionScope` API

③ Require the session scope using the `VaadinSessionScope` API

For example, if a `LocalizationContext` is bound to the current Vaadin session using a session attribute, it can be retrieved using the standard Holon Platform `Context` API.

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

Optional<LocalizationContext> localizationContext = Context.get().resource
(LocalizationContext.class); ②

localizationContext = LocalizationContext.getCurrent(); ③
```

① Bind a `LocalizationContext` instance to the current Vaadin session attribute

② The `LocalizationContext` instance is now available as a context resource

③ The `getCurrent()` method will return the `LocalizationContext` instance

6. Device information

Since version 5.2.7, the DeviceInfo API is deprecated: use `com.vaadin.flow.server.WebBrowser` instead (it can be obtained from `VaadinSession`).

7. Routing and navigation

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin-flow-navigator</artifactId>
<version>5.5.0</version>
```

The `holon-vaadin-flow-navigator` artifact provides URL query parameters marshalling and unmarshalling between URL representation and Java types using the `@QueryParameter` annotation and a `Navigator` API which can be used to handle the application UI *views* routing

Furthermore, a complete support for view *authentication* and *authorization* is provided, using the Holon Platform core authentication APIs.

7.1. Navigation parameters handling

When the `holon-vaadin-flow-navigator` artifact is in classpath, the URL query parameters marshalling and unmarshalling for application *routes* is automatically enabled and setted up at application bootstrap.

The URL query parameters value handling can thus be implemented using the QueryParameter annotation on *routing target classes* (i.e. UI component classes annotated with `@Route`) class **fields**.

For information about the Vaadin Flow routing architecture, including the `@Route` annotation and route layouts, see the Vaadin Flow documentation.

When a routing target class field is annotated with `@QueryParameter`, **its value will be automatically setted using the corresponding URL query parameter value**, if available, just before the component is rendered in application UI.

The `String` type URL query parameter value will be **converted to the required Java type** declared by the field type. See Built in query parameter types for the Java types supported by default and Adding query parameter type support to extend the supported Java types.

By default, the URL query parameter name bound to a `@QueryParameter` field is equal to the **field name**.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter ①
  private Integer parameter;

}
```

① The bound URL query parameter name is the field name, i.e. `parameter`. For example, when the route `some/path?parameter=1` is used, the `parameter` field value will be setted to the `1` Integer value.

The `value()` annotation attribute can be used to **declare the URL query parameter name** to which the field is bound, if you don't want to use the field name.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter("myparam") ①
  private Integer parameter;

}
```

① The URL query parameter name bound to the `parameter` field is `myparam`. For example, when the route `some/path?myparam=1` is used, the `parameter` field value will be setted to the `1` Integer value.

The routing target class field declaration (even `private`) it's enough to enable the the query parameter value handling, but the standard Java Beans convention can be also used, providing a **setter method** to set the field value.

If a field setter method is available in the routing target class, it will be used to set the query parameter field value. This allows, for example, to perform some operations one the query parameter value before it is setted as field value.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter
  private Integer parameter;

  public void setParameter(Integer parameter) { ①
    this.parameter = parameter;
  }

}
```

① The setter method will be used to set the `parameter` query parameter field value

7.1.1. Query parameter URL encoding and decoding

By default, all the navigation related APIs automatically handle the query parameter names and values **encoding**, when serialized to a URL, and **decoding**, when deserialized from a URL.

The query parameter names and values encoding and decoding is performed using the `application/x-www-form-urlencoded` MIME type and the default `UTF-8` charset.

Each API which handles URL query parameters provides method to skip the query parameter names and values encoding and decoding, if required.

See the specific API documentation below for further details.

7.1.2. Using the injected parameter values

The URL query paramter values are automatically injected in the routing target class `@QueryParameter` annotated fields at the **after navigation** routing lifecycle event, i.e. when the routing target component instance is added to the UI.

For this reason, consistent parameter values are only available from this phase on. To handle and use the `@QueryParameter` annotated fields value, you have to ensure to be at least in the *after navigation* lifecycle phase of the routing component.

For this purpose, the `com.vaadin.flow.router.AfterNavigationObserver` interface can be used, implementing it by the routing target class.

```
@Route("some/path")
public class View extends Div implements AfterNavigationObserver { ①

  @QueryParameter
  private Integer parameter;

  @Override
  public void afterNavigation(AfterNavigationEvent event) { ②
    /* handle the parameters value */
    Notification.show("Parameter value: " + this.parameter);
  }

}
```

① Implement the `AfterNavigationObserver` interface to manage *after navigation* events

② When the after navigation event occurs, all the `@QueryParameter` fields values are consistent and already injected using the URL query parameters

Since it is bound to the same *after navigation* lifecycle phase, a `@OnShow` annotated method can also be used.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter
  private Integer parameter;

  @OnShow ①
  public void processParameters() {
    /* handle the parameters value */
    Notification.show("Parameter value: " + this.parameter);
  }

}
```

① When a `@OnShow` annotated method in invoked, all the `@QueryParameter` fields values are consistent and already injected using the URL query parameters

See Using `@OnShow` on route target classes for details.

7.1.3. Direct query parameter values deserialization

The NavigationParameters API can be used to directly deserialize URL query parameter values into the supported Java types.

The `NavigationParameters` API provides a set of builder methods to create a `NavigationParameters` handler instance from a `Map` of query parameter name and values, from a `com.vaadin.flow.router.QueryParameters` instance or from a `com.vaadin.flow.router.Location`

reference.

```
Map<String, List<String>> queryParameters = getQueryParameters();
NavigationParameters navigationParameters = NavigationParameters.create
(queryParameters); ①

navigationParameters = NavigationParameters
    .create(QueryParameters.simple(Collections.singletonMap("test", "value"))); ②

navigationParameters = NavigationParameters.create(new Location("host.com/?test=value
")); ③
```

① Create a `NavigationParameters` from a `Map` of query parameter name and values

② Create a `NavigationParameters` from a `com.vaadin.flow.router.QueryParameters` instance

③ Create a `NavigationParameters` from a `com.vaadin.flow.router.Location` reference

By default, the query parameter names and values are **decoded from the URL representation**, using the `application/x-www-form-urlencoded` MIME type and the default `UTF-8` charset.

To skip query parameter names and values decoding, the `NavigationParameters` API builder methods provides a `decode` parameter which can be set to `false`.

```
NavigationParameters navigationParameters = NavigationParameters.create
(getQueryParameters(), false); ①
```

① Set the `decode` parameter to `false` to skip the query parameter names and values decoding

The `NavigationParameters` API provides a set of methods to obtain the query parameter values, **deserialized to the required Java type**, if supported.

See Built in query parameter types for the Java types supported by default and Adding query parameter type support to extend the supported Java types.

```
NavigationParameters navigationParameters = NavigationParameters.create
(getQueryParameters()); ①

boolean hasParameterAndValue = navigationParameters.hasQueryParameter("myparam"); ②

List<Integer> values = navigationParameters.getQueryParameterValues("myparam",
Integer.class); ③

Optional<Integer> value = navigationParameters.getQueryParameterValue("myparam",
Integer.class); ④

Integer valueOrDefault = navigationParameters.getQueryParameterValue("myparam",
Integer.class, 0); ⑤
```

① Create a `NavigationParameters` handler

② Check if a parameter named `myparam` is present and has a value

③ Get the values of the parameter named `myparam`, deserialized using the `Integer` type

④ Get the single value of the parameter named `myparam`, deserialized using the `Integer` type, if available

⑤ Get the single value of the parameter named `myparam`, deserialized using the `Integer` type, or the default `0` value

7.1.4. Built in query parameter types

By default, the following Java types are supported for query parameter values conversion from the URL query string:

Type	Sub type	Format
String	[NONE]	Any text
Number	`Integer`, `Long`, `Short`, `Byte`, `Float`, `Double`, `BigInteger`, `BigDecimal`	For decimal numbers, the **dot** (`.`) character must be used as decimal positions separator. The `-` character can be used for negative numbers. Hexadecimal representation is supported using the `0x` or `0X` prefix.
Boolean	[NONE]	`true` or `false`
Enums	[NONE]	The enumeration value **name**
LocalDate	[NONE]	ISO date format, for example `2019-01-11`
LocalTime	[NONE]	ISO time format, for example `10:15` or `10:15:30`
LocalDateTime	[NONE]	ISO date and time format, for example `2019-01-11T10:15:30`
OffsetTime	[NONE]	ISO time format with offset support, for example `10:15:30+01:00`
OffsetDateTime	[NONE]	ISO date and time format with offset support, for example `2019-01-11T10:15:30+01:00` or `2019-01-11T10:15:30+01:00[Europe/Paris]`
java.util.Date	[NONE]	ISO date and/or time format, for example `2019-01-11` or `2019-01-11T10:15:30`

7.1.5. Optional query parameter values

The `java.util.Optional` class is supported for `@QueryParameter` field type declarations.

Using `Optional`, the field value will never be `null`, and when the parameter value is not available a `Optional.empty()` value is setted as field value.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter
  private Optional<String> parameter; ①

}
```

① `Optional` query parameter `String` type declaration

7.1.6. Multiple query parameter values

Multiple query parameter values are supported either using the `java.util.Set` or `java.util.List` class.

For example, given the following routing target class:

```
@Route("some/path")
public class View extends Div {

  @QueryParameter
  private Set<String> parameter; ①

}
```

① Query parameter with multiple value support declaration using a `java.util.Set`

For the routing URL `some/path?parameter=a&parameter=b`, the `parameter` field value will be a `java.util.Set` instance containing the `a` and `b` String values.

When the URL query parameter value is not available, a empty `java.util.Set` or `java.util.List` is setted as field value.

7.1.7. Adding query parameter type support

The `@QueryParameter` annotated fields supported Java types set can be **extended**, adding support for new Java types marshalling and unmarshalling, using the NavigationParameterTypeMapper API.

To add support for a new query parameter type, a `NavigationParameterTypeMapper` instance bound to the required type can be created and automatically registered using the Java service extension architecture, i.e. providing a `com.holonplatform.vaadin.flow.navigator.NavigationParameterTypeMapper` named file in the `META-`

`INF/services` folder of a jar, containing the `NavigationParameterTypeMapper` concrete class names to register.

For example, suppose we need to handle the following `MyType` parameter type:

```
public class MyType {

  private final int value;

  public MyType(int value) {
    this.value = value;
  }

  public int getValue() {
    return value;
  }

}
```

We can create a new `NavigationParameterTypeMapper` class, generalized on the `MyType` type, as follows:

```
public class MyTypeParameterMapper implements NavigationParameterTypeMapper<MyType> {

  @Override
  public Class<MyType> getParameterType() { ①
    return MyType.class;
  }

  @Override
  public String serialize(MyType value) throws InvalidNavigationParameterException {
②
    if (value != null) {
      return String.valueOf(value.getValue());
    }
    return null;
  }

  @Override
  public MyType deserialize(String value) throws InvalidNavigationParameterException {
③
    if (value != null) {
      return new MyType(Integer.valueOf(value));
    }
    return null;
  }

}
```

① Handled parameter type declaration

② Parameter value unmarshalling to the `String` type URL query parameter value

③ Parameter value marshalling from the `String` type URL query parameter value

An then register the `MyTypeParameterMapper` class using the Java service extension architecture, creating a suitable file under the `META-INF/services` folder:

com.holonplatform.vaadin.flow.navigator.NavigationParameterTypeMapper file

```
my.package.MyTypeParameterMapper
```

This way, the `MyType` parameter type will be automatically handled using the `@QueryParameter` annotation.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter
  private MyType parameter; ①

}
```

① The `MyType` type parameter is handled using the `MyTypeParameterMapper` class

7.1.8. Required query parameters

The `required()` attribute of the `@QueryParameter` annotation can be used to declare **required** query parameter values.

When a required query parameter value is not available from the URL query string, a navigation error is fired before the navigation to the routing target occurs.

The InvalidNavigationParameterException is used to notify the error and interrupt the navigation.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter(required = true) ①
  private Integer parameter;

}
```

① The query parameter is required

See Default query parameter values to provide a default value for a parameter, which can be used to provide a default value for a required parameter when it is not available from the URL query string.

7.1.9. Default query parameter values

The `defaultValue()` attribute of the `@QueryParameter` annotation can be used to declare a **default value** for a query parameter. This value will be used and injected into the parameter field when the parameter value is not available from the URL query string.

The `defaultValue()` attribute value is a `String`, and the format conventions described in Built in query parameter types should be used as `String` format for the supported Java types.

```
@Route("some/path")
public class View extends Div {

  @QueryParameter(defaultValue = "0") ①
  private Integer parameter;

}
```

① Declare the parameter default value

7.2. Using `@OnShow` on route target classes

The OnShow annotation can be used un **route target class methods** to handle the **after navigation** routing lifecycle event, i.e. to execute code when the routing target component instance is added to the UI.

A method annotation with `@OnShow`:

- Must be `public`.
- Can declare either no parameters or a single parameter of type `com.vaadin.flow.router.AfterNavigationEvent`, to obtain the event which triggered the method invocation.

Any valid `@OnShow` annotated method (including super classes) in invoked at the *after navigation* phase of the routing target component. No invocation order is guaranteed.

```
@Route("some/path")
public class View extends Div {

  @OnShow ①
  public void afterNavigation1() {
    /* ... */
  }

  @OnShow ②
  public void afterNavigation2(AfterNavigationEvent event) {
    /* ... */
  }

}
```

① Declare the method to be invoked just after the navigation to this routing target

② Declare the method to be invoked just after the navigation to this routing target, providing the `AfterNavigationEvent` type event

7.3. The `Navigator` API

The `Navigator` API can be used to handle the application routing, i.e. the navigation between the UI components which represent the application *views*.

Each `Navigator` is bound to a UI and the `Navigator` API for the current UI can be obtained using the `get()` static method. This method checks if a `Navigator` instance is available as a *context* resource and if not uses an internal registry to obtain the instance bound to the current UI, creating a new instance if none was bound to the UI yet.

See the Holon Platform Context documentation for information about the context resources management.

```
Navigator navigator = Navigator.get(); ①
```

① Get the `Navigator` API for the current UI

A `Navigator` instance can also be directly created for a specific UI using the `create` static method.

```
Navigator navigator = Navigator.create(myUI); ①
```

① Create a `Navigator` for the given UI

The `Navigator` API makes available a set of methods to handle the routing between the application's declared route targets, ensuring a **consistent query parameters** management with Java types marshalling support (see Navigation parameters handling) and providing suitable **routing URL builders** using the supported Java types to provide the query parameters value.

See Built in query parameter types to learn about the default supported Java types for query parameters declaration and Adding query parameter type support to learn how to add support for additional Java types.

For information about the Vaadin Flow routing architecture, including the `@Route` annotation and route layouts, see the Vaadin Flow documentation.

The `Navigator` API provides a set of `navigateTo(…)` methods to perform navigation towards a routing target. The routing target URL can be declared either using:

- The **route path**, as declared using the `@Route` annotation.
- The **routing target class** reference.
- A full URL **location**, which includes the route path and the optional query part with the query parameters declaration.

Each `Navigator` API `navigateTo(…)` method provides multiple versions in order to:

- Specify the query parameter values, using a map of query parameter name and the corresponding value, provided as one of the **supported Java type object**. The `Navigator` API will take care of parameter value serialization to be used in the URL query part.
- Specify one or more **path parameter**, if the routing target class support them through the `com.vaadin.flow.router.HasUrlParameter` interface.

For example, supposing the following routing target class is registered in the application:

```
@Route("some/path")
public class View extends Div {

  @QueryParameter("myparam")
  private Integer parameter; ①

}
```

① Declares a single `Integer` type query parameter named `myparam`

The `Navigator` API can be used as follows to navigate to the routing target:

```
Navigator navigator = Navigator.get(); ①

navigator.navigateTo("some/path"); ②
navigator.navigateTo("some/path", Collections.singletonMap("myparam", new Integer(1))
); ③

navigator.navigateTo(View.class); ④
navigator.navigateTo(View.class, Collections.singletonMap("myparam", new Integer(1)));
⑤

navigator.navigateToLocation("some/path?myparam=1"); ⑥
```

① Get the `Navigator` for the current UI

② Navigate to the `View` component using the route path

③ Navigate to the `View` component using the route path and specifying the `myparam` query parameter value

④ Navigate to the `View` component using the component class

⑤ Navigate to the `View` component using the component class and specifying the `myparam` query parameter value

⑥ Navigate to the `View` component using the full location URL, including the route path and the query parameters

Supposing the `View` component declares a *path parameter* using the `com.vaadin.flow.router.HasUrlParameter` interface:

```
@Route("some/path")
public class View extends Div implements HasUrlParameter<String> {

  @QueryParameter("myparam")
  private Integer parameter;

  @Override
  public void setParameter(BeforeEvent event, String parameter) {
    /* handle the path parameter value */
  }

}
```

The `Navigator` API can be used as follows:

```
Navigator navigator = Navigator.get(); ①

navigator.navigateTo(View.class, "value"); ②
navigator.navigateTo(View.class, "value", Collections.singletonMap("myparam", new
Integer(1))); ③
```

① Get the `Navigator` for the current UI

② Navigate to the `View` component using the component class and providing the path parameter `value`

③ Navigate to the `View` component using the component class and providing both the path parameter `value` and the `myparam` query parameter value

Furthermore, the `Navigator` API provides provides two convenience methods:

- `navigateToDefault()`: to navigate to the default route, i.e. the empty (`""`) route, if available.
- `navigateBack()`: to navigate back in the browser navigation history.

```
navigator.navigateToDefault(); ①
navigator.navigateBack(); ②
```

① Navigate to the default route

② Navigate back in the browser navigation history

7.3.1. The navigation builder API

The `Navigator` API provides a **navigation URL builder API**, available through the `navigation(…)` methods, which allows to declare a routing target URL and its parameters (both *query* and *path* parameter types) using a *fluent* style builder and then either perform the actual navigation or obtain the complete location URL.

The navigation builder API can be obtained either specifying the **route path** or the **routing target class** and provides methods to set the *path* or *query* parameters to use.

The query parameter values are provided using one of the **supported Java types**, the `Navigator` API will take care of parameter value serialization to be used in the URL query part.

The navigation is performed using the `navigate()` method.

To obtain the complete URL location and do not perform the actual navigation, the `asLocation()` or `asLocationURL()` methods can be used, which provide the navigation URL as a `com.vaadin.flow.router.Location` object or a `String` respectively.

```
Navigator navigator = Navigator.get(); ①

navigator.navigation("some/path") ②
    .withQueryParameter("myparam", new Integer(1)) ③
    .withQueryParameter("multi", "a", "b", "c") ④
    .navigate(); ⑤

navigator.navigation(View.class) ⑥
    .withPathParameter("value") ⑦
    .navigate(); ⑧

Location location = navigator.navigation("some/path") //
    .withQueryParameter("myparam", new Integer(1)) //
    .asLocation(); ⑨

String url = navigator.navigation(View.class) //
    .withQueryParameter("myparam", new Integer(1)) //
    .asLocationURL(); ⑩
```

① Get the `Navigator` for the current UI

② Build a navigation using a route path

③ Add a query parameter value

④ Add multiple query parameter values

⑤ Perform the navigation action

⑥ Build a navigation using a route target class

⑦ Set the path parameter value

⑧ Perform the navigation action

⑨ Get the complete navigation URL as a `com.vaadin.flow.router.Location` object

⑩ Get the complete navigation URL as a `String`

By default, the query parameter names and values are **encoded to the URL representation**, using the `application/x-www-form-urlencoded` MIME type and the default `UTF-8` charset.

To skip the query parameter names and values encoding, the `encodeQueryParameters` builder method can be used:

```
Navigator.get().navigation("some/path") ①
    .withQueryParameter("myparam", new Integer(1)) ②
    .withQueryParameter("multi", "a", "b", "c") ③
    .encodeQueryParameters(false) ④
    .navigate(); ⑤
```

① Build a navigation using a route path

② Add a query parameter value

③ Add multiple query parameter values

④ Disable the query parameters URL encoding

⑤ Perform the navigation action

7.3.2. Get the URL of a navigation target

The `Navigator` API provides a set of `getUrl(⋯)` convenince methods to obtain the registered URL for a navigation target class.

```
String url = navigator.getUrl(View.class); ①
url = navigator.getUrl(View.class, "path_param_value"); ②
url = navigator.getUrl(View.class, "path_param_value1", "path_param_value2"); ③
```

① Get the `View` routing target class base URL

② Get the `View` routing target class URL with the provided path parameter value

③ Get the `View` routing target class URL with the provided path parameter values

7.3.3. Listening to navigation changes

The NavigationChangeListener listener interface can be use to listen to navigation target changes.

The navigation change event provides:

- The changed URL **location**.
- The changed **navigation target component** instance.

A `NavigationChangeListener` can be registered using the `Navigator` API `addNavigationChangeListener` method and it is invoked when a routing action is performed, i.e. when the current navigation target changes.

```
Navigator navigator = Navigator.get(); ①

navigator.addNavigationChangeListener(event -> { ②
  Location location = event.getLocation(); ③
  HasElement target = event.getNavigationTarget(); ④
});
```

① Get the `Navigator` for the current UI

② Register a `NavigationChangeListener`

③ The new navigation location

④ The new navigation target instance

7.3.4. Navigation links

The NavigationLink API represents a **navigation link component**, which handles navigation

internally instead of loading a new page in the browser.

The `NavigationLink` API provides a builder API to configure the navigation target URL, either specifying a route path or a routing target class, and providing any *path* or *query* parameter value.

The query parameter values are provided using one of the **supported Java types**, the link implementation will take care of parameter value serialization to be used in the URL query part.

The `NavigationLink` API extends `HasComponent`, allowing to use it as a UI component through the `getComponent()` method. See The `HasComponent` interface.

```
NavigationLink link = NavigationLink.builder("some/path") ①
    .withQueryParameter("myparam", 123) ②
    .text("Link text") ③
    .build();

link = NavigationLink.builder(View.class) ④
    .withQueryParameter("myparam", 123) ⑤
    .withPathParameter("value") ⑥
    .text("Link text") ⑦
    .build();

myLayout.add(link.getComponent()); ⑧
```

① Build a `NavigationLink` using a route path

② Add a query parameter value

③ Set the link text

④ Build a `NavigationLink` using a routing target class

⑤ Add a query parameter value

⑥ Add a path parameter value

⑦ Set the link text

⑧ Add the link component to a layout

By default, the query parameter names and values are **encoded to the URL representation**, using the `application/x-www-form-urlencoded` MIME type and the default `UTF-8` charset.

To skip the query parameter names and values encoding, the `encodeQueryParameters` builder method can be used:

```
NavigationLink link = NavigationLink.builder("some/path") ①
    .withQueryParameter("myparam", 123) ②
    .encodeQueryParameters(false) ③
    .text("Link text") ④
    .build();
```

① Build a `NavigationLink` using a route path

② Add a query parameter value

③ Disable the query parameters URL encoding

④ Set the link text

7.4. Authentication support for UI routes

When the `holon-vaadin-flow-navigator` artifact is in classpath, the Authenticate annotation can be used on routing target classes to provide **authentication support** to the routing infrastructure.

A routing target class annotated with `@Authenticate` is under authentication control: only authenticated users can access the corresponding route.

An AuthContext definition is required in order to enable the authentication support and the `AuthContext` to use must be available as a *context* resource, for example bound to the Vaddin session.

See the Holon Platform AuthContext documentation for information about the authentication context and the Realm documentation for information about the authentication configuration.

The `redirectURI()` attribute of the `@Authenticate` annotation can be used to specify a **redirection URL** to use when a not authenticated request is performed against an authentication protected route.

When a redirect URI is not provided and the authentication is not available for an authentication protected route, a UnauthorizedNavigationException is used to notify the error and interrupt the navigation.

```
@Authenticate ①
@Route("some/path")
public class View1 extends Div {

  /* content omitted */

}

@Authenticate(redirectURI = "login") ②
@Route("some/path")
public class View2 extends Div {

  /* content omitted */

}
```

① The `View1` routing target class, and so the `some/path` route path, can be accessed only by an authenticated user

② The `View2` routing target class, and so the `some/path` route path, can be accessed only by an

authenticated user and a `login` redirect URI is specified and will be used to redirect the user navigation when the authentication is not available

The `@Authenticate` annotation can be also used on a **routing layout class**: any routing target class child of the layout will inherit the `@Authenticate` annotation definition.

```
@Authenticate ①
public class MainLayout extends Div implements RouterLayout {

}

@Route(value = "some/path", layout = MainLayout.class) ②
public class View extends Div {

  /* content omitted */

}
```

① The `@Authenticate` is declared on a `RouterLayout`: any child routing target class will inherit the authentication control declaration

② The `View` routing target class declares `MainLayout` as parent layout, so it will be under authentication control

7.5. Authorization support for UI routes

When the `holon-vaadin-flow-navigator` artifact is in classpath, the `javax.annotation.security.RolesAllowed` and `javax.annotation.security.PermitAll` annotations can be used on routing target classes to perform a **role based authorization access control** on routing targets.

An AuthContext definition is required in order to enable the authorization support and the `AuthContext` to use must be available as a *context* resource, for example bound to the Vaddin session.

See the Holon Platform AuthContext documentation for information about the authentication context and the Realm documentation for information about the authentication configuration.

The user roles check is performed using the current `Authentication` permissions, provided by the `AuthContext`.

See the Holon Platform Authorization documentation for information about the permissions configuration and control.

When the `javax.annotation.security.RolesAllowed` annotation is used, **any of the specified role name** must be granted to the current authenticated user in order to allow the route access.

When the authorization control fails, a ForbiddenNavigationException is used to notify the error

and interrupt the navigation.

```
@RolesAllowed({ "ROLE1", "ROLE2" }) ①
@Route("some/path")
public class View extends Div {

  /* content omitted */

}
```

① The `View` routing target class, and so the `some/path` route path, is under authorization control and can be accessed only if either the `ROLE1` or `ROLE2` role name is granted to the authenticated user

8. Spring integration

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

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-vaadin-flow-spring</artifactId>
<version>5.5.0</version>
```

The `holon-vaadin-flow-spring` artifact includes the standard Vaadin Flow Spring integration support, declaring the `vaadin-spring` artifact as dependency. See the Vaadin Spring documentation for information about the standard integration features and setup.

8.1. Provide the `Navigator` API as a Spring bean

The EnableNavigator annotation can be used on a Spring configuration class to made available a UI-scoped `Navigator` instance as a Spring bean.

The Vaadin Spring **UI scope** must be enabled to use this configuration annotation, for example using the `com.vaadin.flow.spring.annotation.EnableVaadin` annotation or the Vaadin Spring Boot integration auto-configuration facilities.

```
@Configuration
@EnableVaadin
@EnableNavigator ①
class Config {

}

@Autowired
Navigator navigator; ②
```

① Enable the `Navigator` API UI-scoped Spring bean configuration

② Obtain the `Navigator` API instance for current UI using injection

8.2. Using Spring Security for authorization control

The EnableSecuredRoute annotation can be used on a Spring configuration class to enable the Spring Security `org.springframework.security.access.annotation.Secured` annotation on routing target classes to perform a role based route authorization access control.

The Spring Security dependency must be explicitly provided by the project and must be available on classpath.

In order to use the `@Secured` annotation based authorization control, a `SecurityContext` must be available and properly configured.

When the `javax.annotation.security.RolesAllowed` annotation is used, **any of the specified role name** must be granted to the current authenticated user in order to allow the route access.

When the authorization control fails, a ForbiddenNavigationException is used to notify the error and interrupt the navigation.

9. Spring Boot integration

The `holon-vaadin-flow-spring-boot` artifact provides integration with Spring Boot, dealing with `Navigator` API and `LocalizationContext` integration auto configuration.

To enable the 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-flow-spring-boot</artifactId>
<version>5.5.0</version>
```

The `holon-vaadin-flow-spring-boot` artifact includes the standard Vaadin Flow Spring Boot integration, declaring the `vaadin-spring` artifact as dependency. See the Vaadin Spring documentation for information about the standard integration features and setup.

9.1. `Navigator` API auto-configuration

The `Navigator` API auto configuration provides automatic setup of a **UI-scoped** `Navigator` bean, if a `Navigator` type bean is not already available in application context. See Provide the `Navigator` API as a Spring bean.

9.2. `LocalizationContext` integration auto configuration

The `LocalizationContext` integration auto-configuration provides the following features:

1. Localization context localization synchronization:

If a `LocalizationContext` type bean is available in context, and its scope is either **UI** (`vaadin-ui`) or **Session** (`session`) or **Vaadin session** (`vaadin-session`), when the `LocalizationContext` localization changes, the changed `Locale` is reflected to the current Vaadin UI or Vaadin session, according to bean scope.

This feature can be disabled setting the application configuration property `holon.vaadin.localization-context.reflect-locale` to `false`.

2. Localization context initialization:

If a `LocalizationContext` type bean is available in context, and its scope is either **UI** (`vaadin-ui`) or **Session** (`session`) or **Vaadin session** (`vaadin-session`), the initial `Locale` detected at session/UI initialization (according to bean scope) is setted to the `LocalizationContext` instance.

This feature can be disabled setting the application configuration property `holon.vaadin.localization-context.auto-init` to `false`.

3. Localization context `I18NProvider` integration:

If a `LocalizationContext` type bean is available in context, and a `I18NProvider` type bean is not available, the detected `LocalizationContext` is used as `I18NProvider` and registered as a Spring bean using the `LocalizationContextI18NProvider` API.

See Use a LocalizationContext as I18NProvider.

This feature can be disabled setting the application configuration property `holon.vaadin.localization-context.i18nprovider` to `false`.

9.3. Spring Boot starters

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

1. The **Vaadin Flow starter** provides the dependencies to the Vaadin Flow Spring Boot integration artifact `holon-vaadin-flow-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, which includes the embedded **Tomcat** auto-configuration.

See the Spring Boot starters documentation for details on Spring Boot *starters*.

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-starter-vaadin-flow</artifactId>
<version>5.5.0</version>
```

2. The **Vaadin Flow starter using Undertow** provides the same dependencies as the previous starter, but uses **Undertow** instead of Tomcat as embedded servlet container.

Maven coordinates:

```
<groupId>com.holon-platform.vaadin</groupId>
<artifactId>holon-starter-vaadin-flow-undertow</artifactId>
<version>5.5.0</version>
```

10. 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`.

11. System requirements

11.1. Java

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

11.2. Vaadin

The Holon Platform Vaadin module requires the Vaadin Flow platform version **12** or higher.