



Holon

PLATFORM

Holon Platform Core

Version 5.0.2

Table of Contents

| | |
|---|----|
| 1. Introduction | 1 |
| 2. Obtaining the artifacts | 1 |
| 2.1. Using the Platform BOM | 2 |
| 3. Core API, services and components | 2 |
| 3.1. Introduction | 2 |
| 3.2. Context | 2 |
| 3.2.1. Scopes | 2 |
| Default Scopes | 3 |
| Context extension: adding Scopes | 4 |
| 3.2.2. Obtaining Context resources | 6 |
| 3.2.3. Default ClassLoader | 6 |
| 3.3. Configuration and parameters | 6 |
| 3.3.1. Configuration | 6 |
| Configuration property | 7 |
| Configuration property provider | 7 |
| Configuration property set | 8 |
| 3.3.2. ParameterSet | 8 |
| 3.4. Data validation | 9 |
| 3.4.1. Validator | 9 |
| 3.4.2. Bultin validators | 9 |
| 3.4.3. Validatable and ValidatorSupport | 10 |
| 3.5. StringValuePresenter | 11 |
| 3.5.1. Default StringValuePresenter | 11 |
| 3.6. Internationalization | 13 |
| 3.6.1. Localizable messages | 13 |
| 3.6.2. @Caption annotation | 14 |
| 3.6.3. Message providers | 14 |
| 3.6.4. LocalizationContext | 15 |
| 3.6.5. Building a LocalizationContext | 15 |
| 3.6.6. Obtaining a LocalizationContext | 16 |
| 3.6.7. Localizing a LocalizationContext | 16 |
| 3.6.8. Using the LocalizationContext | 17 |
| 3.7. Properties | 18 |
| 3.7.1. Configuration | 18 |
| 3.7.2. Converters | 19 |
| 3.7.3. Localization (caption) | 21 |
| 3.7.4. Validation | 21 |
| 3.8. Path | 21 |

| | |
|--|----|
| 3.9. PathProperty | 22 |
| 3.10. VirtualProperty | 22 |
| 3.11. PropertySet | 23 |
| 3.12. PropertyBox | 24 |
| 3.13. Property value presentation | 26 |
| 3.13.1. PropertyValuePresenter registration | 27 |
| 3.14. Property rendering | 27 |
| 3.14.1. PropertyRenderer registration | 29 |
| 3.15. Java Beans and the Property architecture | 29 |
| 3.15.1. BeanIntrospector | 29 |
| BeanPropertyPostProcessor | 30 |
| Builtin bean property processing annotations | 31 |
| 3.15.2. BeanPropertySet | 31 |
| 3.15.3. BeanIntrospector cache | 34 |
| 3.16. Datastore | 34 |
| 3.16.1. Expressions and resolvers | 34 |
| 3.16.2. Property configuration and converters | 35 |
| 3.16.3. DataTarget | 36 |
| 3.16.4. Data manipulation | 36 |
| 3.17. Query | 39 |
| 3.17.1. Query definition | 39 |
| QueryFilter | 40 |
| QuerySort | 42 |
| Aggregations | 43 |
| Paging results | 45 |
| Query results projection | 45 |
| 3.17.2. Configuration | 49 |
| Multiple Datastore configuration | 49 |
| 3.17.3. Common Datastore configuration properties | 50 |
| 3.17.4. Relational Datastores | 50 |
| Sub-query | 50 |
| Alias and Joins | 52 |
| 3.17.5. Datastore extension | 54 |
| Common ExpressionResolver s | 54 |
| DataTargetResolver | 54 |
| Custom QueryFilters | 54 |
| Custom QuerySorts | 57 |
| Datastore <i>commodities</i> definition and registration | 58 |
| 3.17.6. Available Datastores | 59 |
| 3.18. Multi tenancy support | 59 |
| 3.19. Utilities | 59 |

| | |
|--|----|
| 3.19.1. Initializer | 60 |
| 3.19.2. SizedStack | 60 |
| 4. HTTP and REST | 60 |
| 4.1. HTTP messages | 60 |
| 4.1.1. HttpRequest | 61 |
| 4.1.2. HttpResponse | 61 |
| 4.2. RESTful client | 61 |
| 4.2.1. Obtain a RestClient instance | 62 |
| 4.2.2. Configure defaults | 63 |
| 4.2.3. Build a request | 63 |
| 4.2.4. Configure the request | 63 |
| Request URI | 63 |
| URI <i>template</i> variable substitution values | 64 |
| URI <i>query</i> parameters | 64 |
| Request headers | 64 |
| Authorization headers | 65 |
| 4.2.5. Invoke the request and obtain a response | 65 |
| 4.2.6. Request invocation methods | 66 |
| 4.2.7. Request entity | 67 |
| 4.2.8. Response type | 68 |
| 4.2.9. Response entity | 68 |
| 4.2.10. Property and PropertyBox support | 69 |
| 5. Authentication and Authorization | 70 |
| 5.1. Realm | 70 |
| 5.1.1. Authenticator | 71 |
| AuthenticationToken | 72 |
| Builtin AuthenticationTokens | 72 |
| Custom authentication tokens | 73 |
| Authentication | 75 |
| 5.1.2. MessageAuthenticator | 75 |
| AuthenticationTokenResolver | 76 |
| 5.1.3. Authorizer | 77 |
| Permission | 77 |
| Authorization checking | 77 |
| 5.1.4. AuthenticationListener support | 78 |
| 5.2. Credentials | 78 |
| 5.2.1. Create and encode Credentials | 78 |
| 5.2.2. Credentials encoder | 79 |
| 5.2.3. Credentials matching | 80 |
| 5.3. Account | 80 |
| 5.3.1. AccountProvider | 81 |

| | |
|--|----|
| 5.3.2. Authenticator | 81 |
| 5.4. AuthContext | 82 |
| 5.5. @Authenticate annotation | 84 |
| 5.6. JWT support | 84 |
| 5.6.1. Introduction | 84 |
| 5.6.2. Configuration | 84 |
| 5.6.3. Building JWT tokens for an Authentication | 86 |
| 5.6.4. Performing authentication using JWT tokens | 87 |
| 6. Spring framework integration | 88 |
| 6.1. Spring beans as context resources | 89 |
| 6.2. EnvironmentConfigPropertyProvider | 90 |
| 6.3. Spring <i>tenant</i> scope | 90 |
| 6.4. Datastore configuration | 91 |
| 6.4.1. DatastoreResolver | 91 |
| 6.4.2. DatastoreCommodityFactory | 91 |
| 6.4.3. DatastorePostProcessor | 92 |
| 6.5. RestClient implementation using Spring RestTemplate | 92 |
| 6.6. Spring Boot auto-configuration | 93 |
| 6.6.1. Spring Boot starters | 93 |
| 7. Loggers | 94 |
| 8. System requirements | 94 |
| 8.1. Java | 94 |

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

1. Introduction

The Holon Platform **Core** module represents the platform foundation, providing the definition of the overall architecture, base structures and APIs.

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.core</groupId>
<artifactId>holon-bom</artifactId>
<version>5.0.2</version>
```

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

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.core</groupId>
      <artifactId>holon-bom</artifactId>
      <version>5.0.2</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2.1. Using the Platform BOM

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

See [Obtain the platform artifacts](#) for details.

3. Core API, services and components

3.1. Introduction

The **holon-core** artifact is the Holon platform **core** API and implementation asset, defining and providing the main platform architecture concepts and structures. All other platform artifacts derive from this one and declares it as a dependency.

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-core</artifactId>
<version>5.0.2</version>
```

3.2. Context

The entry point of the context API is the [Context](#) interface.

The **Context** behaves as a generic resources registry and provider. A *resource* can be any Java class instance, and it's identified by a String **key**. The **Context** allows resource registration and retrieving in a static way, using a classloader-scoped default *singleton* instance of the registry, which can be obtained using the `get()` method:

```
Context currentContext = Context.get(); ①
```

① Get the current Context instance

3.2.1. Scopes

The **Context** is organized in *scopes*, represented by the [ContextScope](#) interface. Each *scope* act as a sub-registry of resources, with it's own namespace. A scope is identified by a **name** and has an assigned priority level. The priority level is an integer number, following standard priority conventions, where the highest priority corresponds to the lowest value, and vice-versa.

When a resource is requested from **Context**, the lookup process is the following:

- Each registered *scope* is queried to obtain the resource, starting from the scope with the highest priority

- The returned resource instance is the one obtained from the first *scope* which provides a resource instance bound to the requested key, if any.

A registered *scope* can be obtained from Context using:

Get a registered ContextScope

```
Optional<ContextScope> scope = Context.get().scope("scopeName"); ①  
  
scope = Context.get().scope("scopeName", aClassLoader); ②
```

① Get a scope by name using the default ClassLoader

② Get a scope by name using a specific ClassLoader

The `ContextScope` interface can be used to obtain, register and remove a scoped resource. Resource registration and removal are optional operations, so the concrete scope implementation could not support them, throwing a `UnsupportedOperationException`.

Default Scopes

The platform provides two default *scopes*, automatically registered and made available to current `Context`:

1. A **ClassLoader**-bound scope, which handles resources as *singleton* instances for the reference `ClassLoader`, that is at most one resource instance for a given key is present in the scope for a specific `ClassLoader`. This scope is registered with a low precedence order. The scope name is provided by the constant `CLASSLOADER_SCOPE_NAME`;
2. A **Thread**-bound scope, which binds resources to current `Thread` using `ThreadLocal` variables. This scope is registered with a high precedence order. The scope name is provided by the constant `THREAD_SCOPE_NAME`.

The `Context` interface provides some useful methods to access the special **Thread**-bound scope, since it could be highly mutable during an application lifecycle:

Using the current Thread scope

```
Optional<ContextScope> threadScope = Context.get().threadScope(); ①  
  
threadScope = Context.get().threadScope(aClassLoader); ②  
  
Context.get().executeThreadBound("resourceKey", resourceInstance, () -> {  
    // do something ③  
});  
  
Context.get().executeThreadBound("resourceKey", resourceInstance, () -> {  
    // do something ④  
    return null;  
});
```

① Get the Thread scope using the default ClassLoader

- ② Get the Thread scope using a specific ClassLoader
- ③ Execute a *Runnable* operation, binding the resource instance with given *resourceKey* to current Thread before execution and removing the binding after operation execution
- ④ Execute a *Callable* operation, binding the resource instance with given *resourceKey* to current Thread before execution and removing the binding after operation execution

Context extension: adding Scopes

Additional `ContextScope` implementations can be added to the default platform `Context` by using standard Java service extensions.

1. Create a class which implements the `ContextScope` interface:

```

public class MyContextScope implements ContextScope {

    @Override
    public String getName() {
        return "MY_SCOPE_NAME"; ①
    }

    @Override
    public int getOrder() {
        return 100; ②
    }

    @Override
    public <T> Optional<T> get(String resourceKey, Class<T> resourceType) throws
TypeMismatchException {
        return Optional.empty(); ③
    }

    @Override
    public <T> Optional<T> put(String resourceKey, T value) throws
UnsupportedOperationException { ④
        throw new UnsupportedOperationException(); // implement this method to allow
resource registration
    }

    @Override
    public <T> Optional<T> putIfAbsent(String resourceKey, T value) throws
UnsupportedOperationException { ④
        throw new UnsupportedOperationException(); // implement this method to allow
resource registration
    }

    @Override
    public boolean remove(String resourceKey) throws UnsupportedOperationException { ④
        throw new UnsupportedOperationException(); // implement this method to allow
resource removal
    }

}

```

- ① The scope name: must be unique among all registered context scopes
- ② The scope ordering number: the lower is the value, the higher is the scope priority in resources lookup
- ③ Implement a meaningful logic to provide the resource identified by given **resourceKey** and of the given **resourceType** type, if available in scope
- ④ If the scope allows direct resource registration, implement the resource management methods

2. Create a file named *com.holonplatform.core.ContextScope* containing the fully qualified class

name(s) of the scope implementation and put it under the **META-INF/services** folder of your project to register the scope in the default Context.

3.2.2. Obtaining Context resources

Most of the times the **Context** is used to request a resource using a *key*, obtaining the most suitable resource instance according to registered *scopes* and current application environment configuration. To obtain a resource, the **resource(...)** methods are provided:

Get a Context resource

```
Optional<ResourceType> resource = Context.get().resource("resourceKey", ResourceType.class); ①

resource = Context.get().resource("resourceKey", ResourceType.class, aClassLoader); ②

resource = Context.get().resource(ResourceType.class); ③
```

- ① Get the resource identified by the given *resourceKey* and of the specified type, using the default ClassLoader
- ② Get the resource identified by the given *resourceKey* and of the specified type, using the specified ClassLoader
- ③ Get the resource of the specified type using the default ClassLoader. The resource key is assumed to be the fully qualified resource class name



The platform standard interfaces which are candidates to be **Context** resources, provides a convenience **getCurrent()** static method to obtain the current implementation available from **Context**, if present.

3.2.3. Default ClassLoader

When referring to the *default* ClassLoader, it means the ClassLoader obtained by using the following lookup strategy:

- The current Thread context ClassLoader, if available
- The ClassLoader with which the platform core classes are loaded, if available
- The System ClassLoader if no other available

3.3. Configuration and parameters

Holon platform relies on some common structures to define and provide configuration properties and parameters used by platform modules.

3.3.1. Configuration

Configuration properties definition, provision and management is supported by the following three API interfaces:

- **ConfigProperty**: represents a configuration property, identified by a String *key* and with a specified *type*;
- **ConfigPropertyProvider**: provides the values for a set of configuration properties;
- **ConfigPropertySet**: a configuration property set definition, bound to one or more providers to provide the property values.

Configuration property

A configuration property is represented by the **ConfigProperty** interface. A configuration property is identified by a String **key** and is bound to a predefined **type**.

ConfigProperty definition

```
ConfigProperty<String> property = ConfigProperty.create("test", String.class); ①  
  
String key = property.getKey(); ②  
  
Class<String> type = property.getType(); ③
```

① Create a configuration property of String type with given *key*

② Get the configuration property key

③ Get the configuration property type

Configuration property provider

The **ConfigPropertyProvider** interface represents a value provider for a set of configuration properties, allowing to read the configuration properties values from different sources.

Each concrete implementation is able to read configuration properties values from a specific source, for example an in-memory key-value map or a *properties* file. Automatic type conversions from String property source values to required configuration property types are performed when applicable.

The platform provides some useful builtin property values providers as shown below:

Builtin configuration property providers

```
Map<String, Object> values = new HashMap<>();  
ConfigPropertyProvider provider = ConfigPropertyProvider.using(values); ①  
  
Properties properties = new Properties();  
provider = ConfigPropertyProvider.using(properties); ②  
  
provider = ConfigPropertyProvider.using("config.properties", ClassUtils  
.getDefaultClassLoader()); ③  
  
provider = ConfigPropertyProvider.usingSystemProperties(); ④
```

① Provider which uses an in memory key-value map as property values source

- ② Provider which uses a `Properties` instance as property values source
- ③ Provider which uses a `properties` File as property values source
- ④ Provider which uses Java `System` properties as property values source

Configuration property set

The `ConfigPropertySet` interface represents a configuration property set bound to one or more `ConfigPropertyProvider` property source.

Each set is identified by a String **name**, used as a prefix for all the configuration properties of the set. Property name segments are separated by convention by a dot, so, for each property key, the property set will lookup for a property named **set_name*.property_name** in the property source.

Platform elements which support a configuration property set provide a specific `ConfigPropertySet` extension to list all supported configuration properties and, in some cases, helper methods to obtain frequently used configuration properties values.

3.3.2. ParameterSet

A `ParameterSet` is the representation of a generic parameter name and value map.

It provides methods to inspect the parameter set and obtain parameter values of a specific type and completely supports `ConfigProperty` representation as a parameter reference.

ParameterSet examples

```
final ConfigProperty<String> property = ConfigProperty.create("test", String.class);

ParameterSet set = ParameterSet.builder()
    .parameter("testParameter", 1L) ①
    .parameter(property, "testValue") ②
    .build();

boolean present = set.hasParameter("testParameter"); ③
present = set.hasNotNullParameter("testParameter"); ④

set.getParameter(property).ifPresent(p -> p.toString()); ⑤
```

- ① Add a `Long` value parameter using a `String` parameter name
- ② Add a parameter value using a `ConfigProperty`. Since the configuration property is of `String` type, only a `String` type value is admitted
- ③ Check if the parameter set contains a parameter identified by a name
- ④ Check if the parameter set contains a parameter identified by a name and it's value is not null
- ⑤ Check if the parameter set contains a parameter identified by given configuration property: if present, perform an operation

3.4. Data validation

The main entry point to configure and perform data validation in Holon Platform is the [Validator](#) interface.

3.4.1. Validator

A [Validator](#) is an interface implemented by a class which performs the validation of a value. The [Validator](#) is generalized on the value type which the validator is able to validate and can be used as a *functional interface*, providing the single method `validate(T value)`, which is responsible for the actual validation operation and throws a [ValidationException](#) if the given value is not valid.

The [ValidationException](#) is *localizable*, supporting invalid value message localization and can act as a container for multiple validation exceptions.

Example validators

```
Validator<String> validator = v -> { ①
    if (v.length() < 10)
        throw new ValidationException("Value must be at least 10 characters");
};

validator = Validator.create(v -> v.length() >= 10, "Value must be at least 10
characters"); ②

validator = Validator.create(v -> v.length() >= 10, "Value must be at least 10
characters",
    "messageLocalizationCode"); ③
```

- ① Create a Validator for `String` value types which checks if the value is at least 10 characters long
- ② Create the same Validator created using the `Validator.create()` builder method
- ③ Create the same Validator created using the `Validator.create()` builder method and providing an invalid value message localization code

3.4.2. Bultin validators

The platform provides a set of validators for the most common use cases. Each of the builtin validators supports a localizable invalid value message or provides a default invalid value message if a custom one is not specified.

The available builtin validators can be obtained using the corresponding static builder method of the [Validator](#) interface:

- `isNull`: checks that the value is *null*
- `notNull`: checks that the value is not *null*
- `notEmpty`: checks that the value is neither *null* nor empty
- `notBlank`: checks that the value is neither *null* nor empty, trimming spaces

- **max**: checks that the value is lower than or equal to a *max* value (for Strings, arrays and collections the size/length is checked against given *max* value)
- **min**: checks that the value is greater than or equal to a *min* value (for Strings, arrays and collections the size/length is checked against given *min* value)
- **pattern**: checks that the value matches a regular expression
- **in**: checks that the value is one of the values of a specified set
- **notIn**: checks that the value is not one of the values of a specified set
- **notNegative**: checks that a numeric value is not negative
- **digits**: checks that a numeric value is within an accepted range of integral/fractional digits
- **past**: checks that a date type value is in the past
- **future**: checks that a date type value is in the future
- **lessThan**: checks that a value is less than another value
- **lessOrEqual**: checks that a value is less than or equal to another value
- **greaterThan**: checks that a value is greater than another value
- **greaterOrEqual**: checks that a value is greater than or equal to another value
- **email**: checks that the value is a valid e-mail address using RFC822 format rules

Example builtin validators

```
try {
    Validator.notEmpty().validate("mustBeNotEmpty"); ①
    Validator.notEmpty("Value must be not empty", "myLocalizationMessageCode").validate
("mustBeNotEmpty"); ②
} catch (ValidationException e) {
    // invalid value
    System.out.println(e.getLocalizedMessage());
}
```

① Uses the builtin **notEmpty** validator to validate a value, using the default invalid value message

② Uses the builtin **notEmpty** validator to validate a value, using a custom invalid value message and localization message code

3.4.3. Validatable and ValidatorSupport

The **ValidatorSupport** interface is implemented by classes which supports adding and removing validators.

The **Validatable** interface declares the support for value validation, using the **Validator** interface, for a class. The **validate(T value)** checks the validity of the given *value* against every registered validator, and throws a **ValidationException** with a single or multiple validation error message if a given value is not valid.

See the **Properties** section as an example of **Validatable** usage.

3.5. StringValuePresenter

A `StringValuePresenter` deals with `String` representation of a generic Object, according to object type and using the current context for example for messages localization.

Presentation parameters can be used to tune the presentation strategy.

3.5.1. Default StringValuePresenter

The default `StringValuePresenter` can be obtained using the `getDefault()` static method.

The default presentation strategy is organized according to the type of the value to present, with the following rules:

- **CharSequence:** the value `toString()` representation is used
- **Boolean:** boolean values are represented using the *default boolean localization* rules of the current `LocalizationContext`, if available. Otherwise, `String.valueOf(value)` is used.

See [Internationalization](#) for further information on internationalization and `LocalizationContext`

- **Localizable:** the value is localized using current `LocalizationContext`, if available.

See [Internationalization](#) for further information on internationalization and `LocalizationContext`

- **Enum:** The enum value name is used by default. If the enumeration is `Localizable`, the value is localized using current `LocalizationContext`, if available. The `@Caption` annotation is supported on enumeration values for localization.
- **Temporal and Date:** The current `LocalizationContext` is used to format the value, if available. Otherwise, a default short date format is used with the default Locale.
- **Number:** The current `LocalizationContext` is used to format the value, if available. Otherwise, the default number format for the default Locale is used.
- **Any other type:** the value `toString()` representation is used
- **Arrays and collections:** Each element of the array/collection is presented using the rules described above, then the values are joined together in a single String using a separator character.



The default separator character for array/collection presentation is `;`. Use `holon.value-presenter.values-separator` parameter to change the separator character to use.

Presentation parameters:

The default `StringValuePresenter` supports the following parameters to setup and tune the presentation:

Table 1. Default presentation parameters

| Name | Constant | Type | Meaning |
|--|---|--|---|
| <i>holon.value-presenter.</i> values-separator | MULTIPLE_VALUES_SEPARATOR | String | The separator character to use for arrays/collections presentation |
| <i>holon.value-presenter.</i> max-length | MAX_LENGTH | Integer number | Limit the max length of the presented String |
| <i>holon.value-presenter.</i> decimal-positions | DECIMAL_POSITIONS | Integer number | Specify the decimal positions to use to present numeric type values |
| <i>holon.value-presenter.</i> disable-grouping | DISABLE_GROUPING | Boolean (true/false) | Disable the use of grouping symbol for numeric type values |
| <i>holon.value-presenter.</i> hide-zero-decimals | HIDE_DECIMALS_WHEN_ALL_ZERO | Boolean (true/false) | Hide number decimals when all decimal positions (if any) are equal to zero |
| <i>holon.value-presenter.</i> percent-style | PERCENT_STYLE | Boolean (true/false) | Use a percent-style format for numeric decimal values |
| <i>holon.value-presenter.</i> temporal-type | TEMPORAL_TYPE | TemporalType enumeration | Set the temporal time format (Date, time or date and time) to use to present Date and Calendar values |



For [Property value presentation](#), the presentation parameters are read from the property configuration attributes.

```
enum MyEnum {

    @Caption("The value 1")
    VALUE1,

    @Caption(value = "The value 2", messageCode = "message.value2")
    VALUE2

}

public void present() {
    String presented = StringValuePresenter.getDefault().present("stringValue"); ①
    presented = StringValuePresenter.getDefault().present("stringValue",
        ParameterSet.builder().parameter(StringValuePresenter.MAX_LENGTH, 6).build());
    ②
    presented = StringValuePresenter.getDefault().present(MyEnum.VALUE1); ③
    presented = StringValuePresenter.getDefault().present(new MyEnum[] { MyEnum.VALUE1,
        MyEnum.VALUE2 }); ④
}
```

① Return `stringValue`

② Return `string`

③ Return `The value 1`, using the `@Caption` annotation

④ Return `The value 1;The value 2`

3.6. Internationalization

The internationalization architecture of the Holon platform relies upon the [LocalizationContext](#) interface, which is the main entry point for localization of messages, numbers and date/time elements.

3.6.1. Localizable messages

A *localizable* message is represented using the following attributes:

- A **default** message: This is the default message to use if a localization attribute or provider is not available
- A localization **message code**: The symbolic message code to use as identifier to provide message translations
- Optional message **arguments**: A set of arguments to be used to replace a conventional placeholder in the message String with actual values

The [Localizable](#) interface is available to represent a localizable message, providing a convenience fluent builder to create [Localizable](#) instances.

```
Localizable localizable = Localizable.builder().message("defaultMessage").messageCode("message.code").build(); ①

localizable = Localizable.builder().message("message &").messageCode("message.code").messageArguments("test") ②
    .build();
```

① Build a `Localizable` with a `defaultMessage` and a message localization code

② Build a `Localizable` using a localization argument too

3.6.2. @Caption annotation

The `Caption` annotation can be used to provide the localizable message to use as the *caption* (i.e. the short description or explanatory label of an element) of an element.

The relevant annotation attributes are:

- **value**: The *default* message to use as caption
- **messageCode**: The symbolic message code to use as identifier to provide message translations

The support for the `@Caption` annotation must be declared and documented by the classes/elements which support it.



For example, the default `StringValuePresenter`, used also as default `Properties` value presenter supports the `@Caption` annotation for `enum` values presentation.

3.6.3. Message providers

To perform actual messages localization the `MessageProvider` functional interface is used. A `MessageProvider` provides a message translation for a specified message localization identifier and a `Locale` representing the language/country for which the translation is required.

The platform makes available a default `MessageProvider` using *properties* files as message localization containers, which can be created using the static `fromProperties(String... basenames)` method.

Properties files are resolved using a configured basenames as prefix, and `Locale` language, country and variant separated by an underscore as file name. Files must have the *.properties* extension.

The basenames follow the `java.util.ResourceBundle` conventions: essentially, a fully-qualified classpath location. If it doesn't contain a package qualifier, it will be resolved from the classpath root. Note that the JDK's standard `ResourceBundle` treats dots as package separators: this means that `test.messages` is equivalent to `test/messages`.

As an example, suppose to have a `messages` folder under the classpath root containing the following files:

- **messages_en_US_var.properties:** This file will be used for a **Locale** with **en** as language, **US** as country and **var** as variant
- **messages_en_US.properties:** This file will be used for a **Locale** with **en** as language, **US** as country and no variant
- **messages_en.properties:** This file will be used for a **Locale** with **en** as language and a country different from **US**
- **messages_it.properties:** This file will be used for a **Locale** with **it** as language, ignoring country or variant
- **messages.properties:** This is the default file to use as fallback if no other match is found for a **Locale**

A message localization properties file simply contains a list of the localization (translation) of the available message localization identifiers (or message codes), e.g. **test.msg=translation**.

3.6.4. LocalizationContext

The **LocalizationContext** interface is the main entry point for localization of messages, numbers and date/time elements.

3.6.5. Building a LocalizationContext

The simplest way to build a **LocalizationContext** is to use the fluent builder:

```
LocalizationContext localizationContext = LocalizationContext.builder()
    .messageProvider(MessageProvider.fromProperties("messages").build()) ①
    .messageProvider(MessageProvider.fromProperties("messages2").build()) ②
    .messageArgumentsPlaceholder("$") ③
    .withDefaultDateTemporalFormat(TemporalFormat.MEDIUM) ④
    .withDefaultTimeTemporalFormat(TemporalFormat.FULL) ⑤
    .withDefaultBooleanLocalization(Boolean.TRUE, Localizable.builder().messageCode(
"boolean.true").build()) ⑥
    .withDefaultBooleanLocalization(Boolean.FALSE,
        Localizable.builder().messageCode("boolean.false").build()) ⑦
    .withInitialSystemLocale() ⑧
    .withInitialLocale(Locale.US) ⑨
    .build();
```

- ① Add a **MessageProvider** using *properties* files located under the **messages** folder (see [Message providers](#))
- ② Add a **MessageProvider** using *properties* files located under the **messages2** folder (see [Message providers](#))
- ③ Use the **\$** character as message localization arguments placeholder (replacing the default **&** character)
- ④ Use the *medium* format as default date format style
- ⑤ Use the *full* format as default time format style

- ⑥ Use the `boolean.true` message code to localize the `true` boolean values
- ⑦ Use the `boolean.false` message code to localize the `false` boolean values
- ⑧ Initially Localize the `LocalizationContext` using the default system `Locale`
- ⑨ Initially Localize the `LocalizationContext` using the `US Locale`

3.6.6. Obtaining a `LocalizationContext`

If the `LocalizationContext` is registered as a `Context` resource using the default context resource key, i.e. the fully qualified `LocalizationContext` class name, the current `LocalizationContext` can be obtained by using the convenience `getCurrent()` method.

3.6.7. Localizing a `LocalizationContext`

Before using a `LocalizationContext`, you must ensure that it is *localized*, i.e. bound to a specific `Locale`. This will be the `Locale` used for the localization of messages, numbers and date/time elements. To *localize* a `LocalizationContext`, use the `localize(...)` method of the `LocalizationContext` interface, providing a `Locale` instance.

To have more control on context localization, a `Localization` object can be used, which allows to setup:

- A *parent* `Localization`, i.e. the `Localization` to use as fallback when a localization operation cannot be successfully performed using current localization, for example because a message translation is not available. This allows the creation of a hierarchy of `Localization`'s;
- The default decimal positions to use to format a localized numeric decimal value, if decimal positions are not explicitly given;
- The default date format style
- The default time format style

LocalizationContext localization

```
LocalizationContext localizationContext = LocalizationContext.getCurrent()  
    .orElseThrow(() -> new IllegalStateException("Missing LocalizationContext")); ①  
  
localizationContext.localize(Locale.US); ②  
boolean localized = localizationContext.isLocalized(); ③  
  
localizationContext.localize(Localization.builder(Locale.JAPAN).defaultDecimalPosition  
s(2)  
    .defaultDateTemporalFormat(TemporalFormat.FULL).build()); ④
```

- ① Require a `LocalizationContext` to be available as context resource
- ② Localize the `LocalizationContext` using the `US Locale`
- ③ Check the `LocalizationContext` is localized
- ④ Localize the `LocalizationContext` using a `Localization`

3.6.8. Using the LocalizationContext

The `LocalizationContext` provides several methods to perform localizations of messages, dates and numbers.

- For numbers formatting, the `NumberFormatFeature` enumeration can be used to tune the format style
- For date and times formatting, the `TemporalFormat` enumeration can be used to specify the format style

```
LocalizationContext ctx = LocalizationContext.builder()
    .messageProvider(MessageProvider.fromProperties("messages").build())
    .withInitialLocale(Locale.US)
    .build();

ctx.getLocale().ifPresent(l -> System.out.println(l)); ①

String localizedMessage = ctx.getMessage("test.message", "defaultMessage"); ②
localizedMessage = ctx
    .getMessage(Localizable.builder().message("defaultMessage").messageCode(
        "test.message").build()); ③

ctx.format(2.56); ④
ctx.format(0.5, NumberFormatFeature.PERCENT_STYLE); ⑤
ctx.format(5600.678, 2); ⑥

NumberFormat nf = ctx.getNumberFormat(Integer.class); ⑦

ctx.format(new Date(), TemporalType.DATE); ⑧
ctx.format(new Date(), TemporalType.DATE_TIME, TemporalFormat.LONG, TemporalFormat
    .LONG); ⑨

ctx.format(LocalDate.of(2017, Month.MARCH, 15)); ⑩
ctx.format(LocalDate.of(2017, Month.MARCH, 15, 16, 48), TemporalFormat.FULL,
    TemporalFormat.SHORT); ⑪

DateFormat df = ctx.getDateFormat(TemporalType.DATE); ⑫
DateTimeFormatter dtf = ctx.getDateTimeFormatter(TemporalType.DATE_TIME); ⑬
```

- ① Print the current `LocalizationContext` `Locale`
- ② Localize a message providing the message localization code and the default message to use if no translation is available for the current `LocalizationContext` `Locale`
- ③ Localize a message using a `Localizable`
- ④ Format a number using default styles and localization settings
- ⑤ Format a number using the *percent* style
- ⑥ Format the given number using 2 decimal places
- ⑦ Get the `LocalizationContext` `NumberFormat` for `Integer` numbers localization

- ⑧ Format a `Date` considering the date value of `DATE` type (without time)
- ⑨ Format a `Date` considering the date value of `DATE_TIME` type (including time) and using the `LONG` style for both date and time parts
- ⑩ Format a `LocalDate` with default style
- ⑪ Format a `LocalDateTime` using `FULL` style for the date part and `SHORT` style for the time part
- ⑫ Get the `DateFormat` to use to format ``Date``'s without time
- ⑬ Get the `DateTimeFormatter` to use to format temporals with date and time

3.7. Properties

The *properties* architecture is a central concept in the Holon platform. A *property* represent a data attribute in a general and abstract way, allowing to:

- Collect all relevant features and configurations of the data attribute in a single point, to avoid duplications and inconsistency between application layers;
- Abstract the property definition from the concrete data representation and persistence model, to favor loose coupling and independence from underlying data structures;
- Use a common structure for data attributes definition which can be shared by different distributed application layers;
- Provide common operations and functionalities, such as value converters and validators;
- Provide builtin naming and localization features;
- Use the property as an abstract data model reference to build queries and to transport data model values.

A *property* is represented by the `Property` interface. Provides a **type** and it is generalized on such type, which represents the value type handled by the property.

3.7.1. Configuration

A `PropertyConfiguration` class is bound to each property, representing a common place to handle property configuration attributes using a general purpose structure which extends `ParameterSet` to provide operations to inspect and use configuration parameter values. This structure can be used also for application-specific properties configuration and platform extensions parameters collector.

The property configuration is accessible from the `Property.getConfiguration()` method. To set the property configuration attributes, the property builder `configuration(String parameterName, Object value)` methods can be used.

The `PropertyConfiguration` interface provides a special `TemporalType` attribute which can be used to specify the nature (date, time or date and time) of generic java temporal types, such as `Date` and `Calendar`. This attribute can be used from a number of platform services and structures to perform consistent operations on property value, such as presentation, rendering or persistence data manipulation.



The Holon platform fully supports the new **Java 8 Date and Time API**, which represents a big step forward compared to the previous date and time support classes, to address the shortcomings of the older `java.util.Date` and `java.util.Calendar` types. It is strongly recommended to use the new `java.time.*` classes to manage date and times, such as `LocalDate`, `LocalTime`, `LocalDateTime` and so on. This way, in addition to achieving a more robust and consistent code, there is no need to use the `TemporalType` property configuration attribute to ensure consistency in property value manipulation and presentation.

Date type property configuration example

```
PathProperty<Date> dateProperty = PathProperty.create("oldDateTypeProperty", Date.class)
    .temporalType(TemporalType.DATE_TIME) ①
    .configuration("myAttribute", "myValue"); ②

PropertyConfiguration cfg = dateProperty.getConfiguration(); ③
Optional<String> value = cfg.getParameter("myAttribute", String.class); ④
```

- ① Set the property `TemporalType`
- ② Set a custom configuration attribute
- ③ Get the property configuration
- ④ Get the `myAttribute` configuration attribute

3.7.2. Converters

Each `Property` supports a `PropertyValueConverter`, which can be used to perform property value conversions when obtaining the property value from the concrete data model and back. The two conversion methods (from the property value type to the data model value type and vice-versa) should be symmetric, so that chaining these together returns the original result for all inputs.

Example converter from String to Integer and back

```
PropertyValueConverter<Integer, String> converter = new PropertyValueConverter<
Integer, String>() {

    @Override
    public Integer fromModel(String value, Property<Integer> property) throws
PropertyConversionException {
        return (value != null) ? Integer.parseInt(value) : null; ①
    }

    @Override
    public String toModel(Integer value, Property<Integer> property) throws
PropertyConversionException {
        return (value != null) ? String.valueOf(value) : null; ②
    }

    @Override
    public Class<Integer> getPropertyType() {
        return Integer.class;
    }

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

};
```

① Convert a **String** model value into the **Integer** property value type

② Convert the **Integer** property value type into the **String** model value type

Some useful **builtin converters** are provided:

- *Numeric boolean converter*: perform conversions from/to a numeric data model type to a **boolean** property type using the following convention: **null** or **0** numeric values will be converted as **false** boolean values, any other value will be converted as the **true** boolean value;
- *Enum by ordinal converter*: perform conversions from/to a Integer data model type to an **Enum** property type using the the enumeration **ordinal** values;
- *Enum by name converter*: perform conversions from/to a String data model type to an **Enum** property type using the the enumeration **name** values;
- *LocalDate value converter*: perform conversions from/to **Date** type data model values and Java 8 **LocalDate** temporal type property types;
- *LocalDateTime value converter*: perform conversions from/to **Date** type data model values and Java 8 **LocalDateTime** temporal type property types.

```
PropertyValueConverter.numericBoolean(Integer.class); ①  
PropertyValueConverter.localDate(); ②  
PropertyValueConverter.localDateTime(); ③  
PropertyValueConverter.enumByOrdinal(); ④  
PropertyValueConverter.enumByName(); ⑤
```

① Numeric boolean converter using a `Integer` type data model value

② `LocalDate` converter

③ `LocalDateTime` converter

④ `Enum` by ordinal converter

⑤ `Enum` by name converter

3.7.3. Localization (caption)

The `Property` interface extends `Localizable` to allow property *caption* localization. The property *caption* is a kind of property description, which can be used also in UI application layers to provide the property description to the user.

See [Internationalization](#) for additional details about the Holon platform internationalization architecture.

3.7.4. Validation

The `Property` interface extends `Validatable` and supports property value validation using `Validators`. See the [Data validation](#) section for detailed information about validators definition and usage.

3.8. Path

Properties which refer to a concrete data model attribute extend the `Path` interface, which represents a named path to a data model attribute and supports hierarchical structures. Each *path* is identified by a `String` **name**, representing the connection with the concrete data model.

A *path* is typed and the path type represents the type of the value identified by such path in the data model.

Path examples

```
Path<String> stringPath = Path.of("pathName", String.class); ①  
  
String name = stringPath.getName(); ②  
boolean root = stringPath.isRootPath(); ③  
  
Path<String> hierarchicalPath = Path.of("subName", String.class).parent(stringPath);  
④  
String fullName = hierarchicalPath.fullName(); ⑤
```

- ① Create a `String` type path with the name `pathName`
- ② The path name is `pathName`
- ③ The path is a *root* path because it has no parent
- ④ Create a path named `subName` and set `pathName` as parent path
- ⑤ The path full name will be `pathName.subName`

3.9. PathProperty

The central interface to define a `Property` bound to a data model attribute is the `PathProperty` interface, which extends `Path` to provide a binding to the symbolic path of the data model attribute.

The `create(...)` method can be used for the definition of a `PathProperty`, providing property configuration attributes, converters, validators and property localization messages through a fluent builder.

Usually a `PathProperty` is defined as a static constant, immutable during the application runtime.

PathProperty definition

```
public final static PathProperty<Long> ID = PathProperty.create("id", Long.class) ①
    .configuration("test", 1) ②
    .validator(Validator.notNull()) ③
    .message("Identifier") ④
    .messageCode("property.id"); ⑤

public final static PathProperty<Boolean> VALID = PathProperty.create("valid",
Boolean.class) ⑥
    .converter(PropertyValueConverter.numericBoolean(Integer.class)); ⑦
```

- ① Create a `PathProperty` named `id` (this is the path name) bound to a `Long` value type
- ② Add a configuration parameter named `test` with value 1
- ③ Add a validator to check that the property value is not *null*
- ④ Set the property *caption* message
- ⑤ Set the property *caption* localization message code
- ⑥ Create a `PathProperty` named `valid` (this is the path name) bound to a `Boolean` value type but which refers to a `Integer` data model attribute
- ⑦ Set the converter to perform conversion between `Integer` data model values and `Boolean` property values and vice-versa

A `PathProperty` can be used in `Query` clauses and projections, and the `PathProperty` interface provides several methods to easily create query clauses and aggregations.

3.10. VirtualProperty

A `VirtualProperty` is a `Property` which is not directly bound to a data model attribute, but which

instead provides its value through a [PropertyValueProvider](#).

The [PropertyValueProvider](#) is a *functional interface* which implementation must be declared when the virtual property is created and is used to provide a *virtual* or *calculated* property value. The [PropertyValueProvider](#) takes a [PropertyBox](#) as argument, allowing the use of other properties to compose and provide the virtual property value. For this reason, the most suitable use of a [VirtualProperty](#) is within a [PropertyBox](#) context.

See [PropertyBox](#) for further information.

VirtualProperty definition

```
public final static VirtualProperty<Integer> ALWAYS_ONE = VirtualProperty.create
(Integer.class, propertyBox -> 1); ①

public final static PathProperty<String> NAME = PathProperty.create("name", String
.class); ②
public final static PathProperty<String> SURNAME = PathProperty.create("surname",
String.class); ③
public final static VirtualProperty<String> FULL_NAME = VirtualProperty.create(String
.class,
    propertyBox -> propertyBox.getValue(NAME) + " " + propertyBox.getValue(SURNAME));
④
```

- ① Create a [VirtualProperty](#) of Integer type which always returns the value 1
- ② [PathProperty](#) definition representing a person *name* attribute
- ③ [PathProperty](#) definition representing a person *surname* attribute
- ④ Create a [VirtualProperty](#) of String type providing the person *full* name, concatenating the person *name* and *surname* values read from the current [PropertyBox](#)

3.11. PropertySet

The [PropertySet](#) interface represents an immutable set of [Property](#) and provides methods for the definition and inspection of the set of properties. It extends the Java standard [Iterable](#) interface, representing an iterable set of properties.

The most common use case of a property set is to aggregate the properties which refer to a data model *entity*, even combining path properties with virtual properties.

Property sets can be used as [Query](#) projections: the query selection will be the set of the properties defined in the [PropertySet](#) and the query result will be a [PropertyBox](#) with the same properties of the projected property set and the query results bound to each property.

```
final PathProperty<String> NAME = PathProperty.create("name", String.class);
final PathProperty<String> SURNAME = PathProperty.create("surname", String.class);
final PathProperty<Long> ID = PathProperty.create("id", Long.class);

PropertySet<Property> set = PropertySet.of(NAME, SURNAME); ①
set = PropertySet.builder().add(NAME).add(SURNAME).build(); ②

boolean contains = set.contains(NAME); ③
set.forEach(p -> p.toString()); ④
String captions = set.stream().map(p -> p.getMessage()).collect(Collectors.joining());
⑤

PropertySet<Property> newSet = PropertySet.builder().add(set).add(ID).build(); ⑥
int size = newSet.size(); ⑦

newSet = PropertySet.builder().add(set).remove(SURNAME).build(); ⑧
```

- ① Create a PropertySet containing **NAME** and **SURNAME** properties
- ② Create a PropertySet containing **NAME** and **SURNAME** properties using the fluent builder
- ③ Check if PropertySet contains the property **NAME**
- ④ Use the **forEach** operation to invoke the **toString** method for each property of the set
- ⑤ Use a **stream** of the properties of the set to join the property captions in a String
- ⑥ Create a new PropertySet as the conjunction of the properties of the first set and the **ID** property
- ⑦ Obtain the set size, i.e. the number of properties in the set
- ⑧ Create a new PropertySet from the first one but excluding the **SURNAME** property

3.12. PropertyBox

A **PropertyBox** represents a container of **Property** values and it is bound to a specific (and immutable) **PropertySet**.

PropertyBox provides methods set and retrieve the values for the properties of set to which it is bound, handling property value validation and conversions according to **Property** configuration and ensuring a consistent behaviour when using a **VirtualProperty**.

The **PropertyBox** abstraction is the base structure to transport and provide property values, used by the Holon platform every time a set of property values comes into play, for example in **Query** results projections involving more than one property.

Using a **PropertyBox** allows to preserve a strong independence from the underlying (and possibly mutable) concrete data model, representing a data model *entity* (when it can be considered a set of data attributes, i.e. properties, and their values) in a generalized fashion.

PropertyBox examples

```
final PathProperty<Long> ID = PathProperty.create("id", Long.class).validator(Validator.notNull());
final PathProperty<String> NAME = PathProperty.create("name", String.class).validator(Validator.notBlank());

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

PropertyBox propertyBox = PropertyBox.create(ID, NAME); ①
propertyBox = PropertyBox.create(PROPERTIES); ②

propertyBox.setValue(ID, 1L); ③
propertyBox.setValue(NAME, "testName"); ④

propertyBox = PropertyBox.builder(PROPERTIES).invalidAllowed(false).set(ID, 1L).set(NAME, "testName").build(); ⑤

propertyBox.validate(); ⑥

Long id = propertyBox.getValue(ID); ⑦
String name = propertyBox.getValueIfPresent(NAME).orElse("default"); ⑧

boolean containsNotNullId = propertyBox.containsValue(ID); ⑨

PropertyBox ids = propertyBox.cloneBox(ID); ⑩
```

- ① Create an empty **PropertyBox** using **ID** and **NAME** properties as property set
- ② Create an empty **PropertyBox** with the same property set but using **PROPERTIES** property set
- ③ Set the value for the **ID** property: the **setValue** method is generalized on property type, so only consistent value types are accepted (a Long in this case)
- ④ Set the value for the **NAME** property: the **setValue** method is generalized on property type, so only consistent value types are accepted (a String in this case)
- ⑤ Create a **PropertyBox** using the fluent builder using the **PROPERTIES** property set and setting the property values
- ⑥ Validate the value of the properties currently present in the box, invoking configured property Validators
- ⑦ Get the value for the **ID** property: the **getValue** method is generalized on property type, so a consistent value type is returned (Long in this case)
- ⑧ Get the *Optional* **NAME** property value, using the **default** String if a value for that property is not present in **PropertyBox**
- ⑨ Check if a value for the **ID** property is present in **PropertyBox**
- ⑩ Clone the **PropertyBox**, creating a new **PropertyBox** with a property set composed only by the **ID** property and copying the values of the properties which was present in the source **PropertyBox** and the destination property set too

3.13. Property value presentation

The Holon platform provides a standard way to present the value of a **Property** as a String using a **PropertyValuePresenter**.

The **PropertyValuePresenter** is *functional interface* aimed to provide a String representation of the value associated to a **Property**. The property value presenters are organized in a registry (the **PropertyValuePresenterRegistry**), which collects all available presenters and provides the most suitable presenter for a given property, using the **conditions**, expressed as **Predicate**, the presenters were registered with.

A *default* property value presenter is provided by the platform and automatically registered in the registry.



The default property value presenter uses a default implementation of **StringValuePresenter** to convert property values into **String**. See **StringValuePresenter** for further information on the presentation strategy.



The default property value presenter uses the **PropertyConfiguration** attributes as property presentation parameters. So you can set default **StringValuePresenter** presentation parameters as property configuration parameters to use them for property presentation.

This way, the registry uses a *fallback* strategy to obtain the most suitable presenter for a property, searching for the presenter associated to the condition which best matches a given property, or falling back to another presenter consistent with the property, if available. If a specific presenter is not found, the *default* presenter is used.

The registry supports a **priority** indication for a **PropertyValuePresenter**, which can be expressed by using standard **javax.annotation.Priority** annotation on presenter implementation class, where lower values corresponds to higher priority. Priority can be used when a set of **conditions** are not clearly one more restrictive than others in the same registry, so an explicit lookup order has to be defined.

The **Property** interface provides a convenience **present(T value)** method to present given property value using the current **PropertyValuePresenterRegistry**, i.e. the registry available as a **Context** resource, if available, or the default registry associated to the current **ClassLoader** otherwise.

Property value presentation

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

String stringValue = ID.present(1L); ①

stringValue = PropertyValuePresenterRegistry.get().getPresenter(ID)
    .orElseThrow(() -> new IllegalStateException("No presenter available for given
property"))
    .present(ID, 1L); ②
```

- ① Present the 1 value for the ID property using the current Context presenters registry or the default one
- ② The same operation made using the PropertyValuePresenterRegistry directly

3.13.1. PropertyValuePresenter registration

The registration of a new PropertyValuePresenter can be made in two ways:

1. Using the PropertyValuePresenterRegistry:

PropertyValuePresenter registration example

```
PropertyValuePresenter<LocalTime> myPresenter = (p, v) -> v.getHour() + "." + v
.getMinute(); ①

PropertyValuePresenterRegistry.get().register(p -> LocalTime.class.isAssignableFrom(p
.getType()), myPresenter); ②
```

- ① Create a presenter for LocalTime type properties which represents the value as *hours.minutes*
- ② Register the presenter binding it to the Predicate wich corresponds to the condition “the property type is LocalTime”

2. Using the standard Java service extensions:

Create a file named *com.holonplatform.core.property.PropertyValuePresenter* containing the fully qualified class name(s) of the PropertyValuePresenter implementation and put it under the META-INF/services folder of your project to register the presenter in the default PropertyValuePresenterRegistry.



When a PropertyValuePresenter is registered using service extensions an **always true** condition is used, i.e. the presenter is available for any property. Use this method only for general purpose presenters. The `javax.annotation.Priority` annotation can be used on presenter's implementation class to assign a priority order to the presenter.

3.14. Property rendering

A further property handling concept is made available by the Holon platform: the property *renderers*.

PropertyRenderer is responsible to render a Property as a specific rendering class type, declared by the `getRenderType()` method.

The property renderers are organized in a registry (the `PropertyRendererRegistry`), which collects all available renderers and provides the most suitable renderer for a given property and a specific rendering type, using the **conditions**, expressed as `Predicate`s`, the renderers were registered with.

This paradigm can be used to provide property representation or management objects in a standard way, organizing the renderers by rendering type and gathering them together in a common registry, to make them available to application layers.



No default `PropertyRenderer` is provided by the core platform module, because the renderers are very related to the specific application logic or UI technology.

The registry supports a **priority** indication for a `PropertyRenderer`, which can be expressed using standard `javax.annotation.Priority` annotation on renderer implementation class, where lower values corresponds to higher priority. Priority can be used when a set of **conditions** is not clearly more restrictive than another, so an explicit lookup order has to be defined.

The `Property` interface provides two convenience methods to render the property value using the current `PropertyRendererRegistry`, i.e. the registry available as a `Context` resource, if available, or the default registry associated to the current `ClassLoader` otherwise. These methods are:

- `render(Class renderType)`: Renders the property as given `renderType` object type. Throws a `NoSuitableRendererAvailableException` if no `PropertyRenderer` is available for this property and given rendering type
- `renderIfAvailable(Class renderType)`: Renders the property as given `renderType` object type if a suitable `PropertyRenderer` for the required `renderType` is available from the `PropertyRendererRegistry` obtained from current `Context` or from the default one for the current `ClassLoader`. If a suitable renderer is not available, an empty `Optional` is returned.

Property renderers

```
class MyRenderingType { ①

    private final Class<?> propertyType;

    public MyRenderingType(Class<?> propertyType) {
        this.propertyType = propertyType;
    }

}

public void render() {
    PropertyRenderer<MyRenderingType, Object> myRenderer = PropertyRenderer.create
(MyRenderingType.class,
    p -> new MyRenderingType(p.getType())); ②

    PropertyRendererRegistry.get().register(p -> true, myRenderer); ③

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

    MyRenderingType rendered = ID.render(MyRenderingType.class); ④
}
```

① Define a custom rendering class (in a UI layer, this could be for example a field to manage

property value)

- ② Create a renderer for `MyRenderingType`, available for any property type
- ③ Register the renderer binding it to an *always true* condition, so it will be available for any property
- ④ Render the `ID` property as `MyRenderingType` type

3.14.1. PropertyRenderer registration

The registration of a new `PropertyRenderer` can be made in two ways:

1. Using the PropertyRendererRegistry:

Property renderers can be registered to a `PropertyRendererRegistry` using the `register` method and providing a condition to bind the renderer only to a specific kind/set of properties.

2. Using the standard Java service extensions:

Create a file named `com.holonplatform.core.property.PropertyRenderer` containing the fully qualified class name(s) of the `PropertyRenderer` implementation and put it under the `META-INF/services` folder of your project to register the renderer in the default `PropertyRendererRegistry`.



When a `PropertyRenderer` is registered using service extensions an **always true** condition is used, i.e. the renderer is available for any property. The `javax.annotation.Priority` annotation can be used on the renderer's implementation class to assign a priority order to the renderer.

3.15. Java Beans and the Property architecture

The Holon platform offers a wide support to treat standard Java Beans properties as `Properties` abstraction.

The properties of a Java Bean class can be represented as `PathProperty`'s, where the property path name corresponds to the bean property name, and are collected in a bean-related property set represented by the `BeanPropertySet` interface.

A `BeanIntrospector` can be used to inspect the properties of a Java Bean class and obtain the corresponding `BeanPropertySet`.

3.15.1. BeanIntrospector

The `BeanIntrospector` interface provides methods to inspect a Java Bean class and obtain informations about bean properties, representing them as a `PathProperty` where the path name of each property corresponds to the bean property name.



Nested bean classes are supported, keeping the property hierarchy intact, i.e. the parent property of a `PathProperty` obtained from the bean property of a nested class will be the bean property to which the nested class refers to. To access a nested property by path name, the conventional *dot notation* is supported. For example `parentProperty.nestedProperty`.

The detected bean properties are collected and returned as a `BeanPropertySet`.

```
BeanIntrospector introspector = BeanIntrospector.get(); ①  
BeanPropertySet<MyBean> properties = introspector.getPropertySet(MyBean.class); ②
```

- ① Get the current `BeanIntrospector`, i.e. the instance registered as a `Context` resource, or the default instance if not available in context
- ② Introspect given bean class and obtain a `BeanPropertySet` which contains all detected bean properties

BeanPropertyPostProcessor

A `BeanPropertyPostProcessor` can be used to integrate the bean introspection strategy and process the properties which will become part of resulting `BeanPropertySet`, for example to modify or add property configuration attributes, validators, converters and so on.

The `BeanPropertyPostProcessor` *functional interface* method `processBeanProperty` is called for every detected and valid bean property, provided as *builder* to participate in the property building process.

The registration of a `BeanPropertyPostProcessor` can be performed in two ways:

- 1. Registration using `BeanIntrospector`:** The `addBeanPropertyPostProcessor` method can be used to register a `BeanPropertyPostProcessor`.
- 2. Registration using the standard Java service extensions:** `BeanPropertyPostProcessor` registration can be performed also using default Java extension services, providing a `com.holonplatform.core.beans.BeanPropertyPostProcessor` file under the `META-INF/services` folder containing the fully qualified `BeanPropertyPostProcessor` concrete class names to register.

BeanPropertyPostProcessor registration example

```
BeanIntrospector.get().addBeanPropertyPostProcessor((property, cls) -> property  
.configuration("test", "testValue")); ①
```

- ① Register a `BeanPropertyPostProcessor` which adds a `test` property configuration attribute to all processed properties



The `javax.annotation.Priority` annotation can be used on a `BeanPropertyPostProcessor` implementation class to assign a priority order within the registered processors list, where lower values corresponds to higher priority.

Builtin bean property processing annotations

The Holon platform provides a number of annotations, processed by the default set of `BeanPropertyPostProcessor`, automatically registered in the `BeanIntrospector`, to tune the property definitions and add property configuration elements such as validators and localizable captions.

The following annotations can be placed on bean property class *attributes* to be processed during the bean class introspection:

- **@Ignore**: Can be used to skip a bean class attribute during the introspection process, which will not be part of the resulting `BeanPropertySet`
- **@Caption**: The default platform internationalization *caption* annotation is supported to set the property *localizable* caption
- **@Sequence**: Set the property sequence order within the `BeanPropertySet`
- **@Config**: a *repeteable* annotation to specify a configuration key and its value to be setted in the property configuration. Only `String` type configuration values are supported by this annotation, use your own `BeanPropertyPostProcessor` to perform more advanced property configuration setup operations
- **@Converter**: Setup a `Converters` for the property, providing either a *builtin* converter or the `PropertyValueConverter` class to use.
- **Validators**: One or more `Validator` can be added to a bean property using one of the following methods:
 - Using the standard `javax.validation.constraints` bean validation API annotations. The supported annotations are: `@Null`, `@NotNull`, `@Size`, `@Min`, `@Max`, `@DecimalMin`, `@DecimalMax`, `@Digits`, `@Future`, `@Past`, `@Pattern`;
 - Using the additional platform validation annotations: `@NotEmpty` (CharSequence not null and not empty), `@NotBlank` (CharSequence not null and not empty trimming spaces), `@NotNegative` (Number not negative) and `@Email` (String is a valid e-mail address);
 - Using the *repeteable* `@Validator` annotation, specifying the custom validator class to use.



For bean validation API and builtin validation annotations, the `message` attribute is used to obtain the **invalid value message** to associate to the validator and, by convention, if the message is included between braces is considered as a localization message code, otherwise as a simple, not localizable, message. The `@ValidationMessage` annotation can be used instead to provide a different, localizable, invalid value message. If such annotation is present, the `message` attribute is ignored.

3.15.2. BeanPropertySet

The `BeanPropertySet` interface represents a set of `PathProperty`'s which corresponds to the properties of a Java Bean class, where the path *name* of each property of the set corresponds to the bean property name.

For nested bean classes, the parent property of the nested `PathProperty` will be the bean property the nested class refers to, and the full path of the nested property will be the property path

hierarchy separated by a *dot* character, for example `parentProperty.nestedProperty`.

It extends the default `PropertySet` structure and provides additional functionalities to:

- Obtain a property **by name**
- **Read** and **write** single property values to and from an instance of the Java Bean class bound to the set
- **Read** the property values from an instance of the Java Bean class bound to the set and obtain such values as a `PropertyBox`
- **Write** the property values contained in a `PropertyBox` to an instance of the Java Bean class bound to the set

BeanPropertySet examples

```
class MyNestedBean {  
  
    private String nestedName;  
  
    public String getNestedName() {  
        return nestedName;  
    }  
  
    public void setNestedName(String nestedName) {  
        this.nestedName = nestedName;  
    }  
  
}  
  
class MyBean {  
  
    private Long id;  
    private boolean valid;  
    private MyNestedBean nested;  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
  
    public boolean isValid() {  
        return valid;  
    }  
  
    public void setValid(boolean valid) {  
        this.valid = valid;  
    }  
  
}
```

```

public MyNestedBean getNested() {
    return nested;
}

public void setNested(MyNestedBean nested) {
    this.nested = nested;
}

}

public static final BeanPropertySet<MyBean> PROPERTIES = BeanIntrospector.get()
    .getPropertySet(MyBean.class); ①

public void propertySet() {
    Optional<PathProperty<Long>> idProperty = PROPERTIES.<Long>getProperty("id"); ②
    PathProperty<Long> id = PROPERTIES.requireProperty("id", Long.class); ③

    PathProperty<String> nestedName = PROPERTIES.requireProperty("nested.nestedName");
    ④

    // read
    MyBean instance = new MyBean();
    instance.setId(1L);

    Long value = PROPERTIES.read("id", instance); ⑤
    PropertyBox box = PROPERTIES.read(instance); ⑥
    value = box.getValue(PROPERTIES.requireProperty("id")); ⑦

    // write
    instance = new MyBean();
    PROPERTIES.write("nested.nestedName", "test", instance); ⑧

    MyBean written = PROPERTIES
        .write(PropertyBox.builder(PROPERTIES).set(PROPERTIES.requireProperty("id"), 1L
    ).build(), new MyBean()); ⑨
}

```

- ① Get the **BeanPropertySet** of the **MyBean** class
- ② Get the **PathProperty** which corresponds to the **id** bean property name, obtaining an **Optional** which is empty if the property name is not found within the bean property set
- ③ Get the *required* **PathProperty** which corresponds to the **id** bean property name: if not found, an exception is thrown
- ④ Get the *nested* property which corresponds to the full path **nested.nestedName**
- ⑤ Read the value of the property with the **id** path name from given bean instance (1)
- ⑥ Read all the values of the bean property set from given bean instance, obtaining a **PropertyBox** which contains the read values
- ⑦ Read the value of the **id** property from the **PropertyBox** obtained in previous read operation (1)

- ⑧ Write the `test` value to the property with path `nested.nestedName` in given bean instance
- ⑨ Write all the values of given `PropertyBox` to the given bean instance

3.15.3. BeanIntrospector cache

By default, the `BeanIntrospector` uses an internal cache of processed bean class and property sets, to boost introspection operations and obtain better performance.

If memory consumption issues are detected, the internal cache can be disabled setting `holon.beans.introspector-cache-enabled` configuration property to `false`. To set the configuration property, either a `System` property or a default `holon.properties` file can be used.

3.16. Datastore

The `Datastore` interface is the main entry point to manage data persistence in a technology/platform/vendor independent way.

The `Datastore` data management strategy relies on the Holon platform `Properties` architecture to express data model attributes in a generic and implementation-independent way, using `PropertyBox` as carrier for data model attributes and entity values.

A concrete `Datastore` implementation could provide a more specialized interface providing functionalities expressly related to the persistence technology/model to which the `Datastore` refers.

The `Datastore` interface provides the following operations:

- **Refresh:** Refresh the data of a data model *entity*
- **Insert:** Insert a data model *entity* into the persistence store
- **Update:** Update a data model *entity*
- **Save:** Insert or update a data model *entity*, depending on the existence of the *entity*
- **Bulk** operations on data model (`bulkInsert`, `bulkUpdate` and `bulkDelete`) to execute batch persistence operations
- **Query execution** using the `Query` interface for query definition and execution.

Each operation which involves a possible persistence store data modification returns an `OperationResult` object, which provides information about the operation outcome, like the number of the elements affected by the execution of the operation or the *auto-generated* key values, if the concrete persistence store supports this feature.

3.16.1. Expressions and resolvers

Main `Datastore` operations rely on the core `Expression` type to configure and model data access and manipulation statements, which will be converted into concrete commands and requests to the underlying persistence store engine in its specific language.

To take part in the `Datastore` data access and manipulation statement definition processes, the `ExpressionResolver` interface can be used. This interface is designed to *resolve*, i.e. translate in a

more convenient or suitable form, an `Expression` type into another `Expression` type.

An `ExpressionResolver` declares the expression type which is able to process, and the expression type which provides as resolution result. An `ExpressionResolver` can return an empty optional if it is not able to resolve given specific expression: this way, the resolution process must proceed to the next available resolver for given expression and resolution type.

When an `Expression` must be resolved, the `Datastore` detects all registered resolvers which declare to resolve the given expression type and provide a consistent resolution type. The resolvers are invoked sequentially, and the first valid resolved expression is taken (if any), interrupting the resolution process.

To provide a **resolution sequence**, i.e. a resolver invocation sequence, an `ExpressionResolver` class can be annotated with the standard `javax.annotation.Priority` annotation, where lower values corresponds to higher priority.

During the resolution process, a `ResolutionContext` object is provided to each `ExpressionResolver`. The resolution context provides a `resolve` method which allows to trigger a nested expression resolution operation. Each concrete `Datastore` implementation could also provide a more specialized `ResolutionContext` interface.

Typically, an `ExpressionResolver` can be used to:

- Translate a custom expression into a well-known `Datastore` expression (for example, a custom query filter class into a standard `QueryFilter` object);
- Translate an expression attribute into one which can be correctly interpreted by a specific `Datastore` (for example, a symbolic data target name into a concrete persistence model structure name, such as a table name in a RDBMS);
- Implement specific expressions for a concrete `Datastore` implementation.

To **register or unregister** an `ExpressionResolver`, the `addExpressionResolver(...)` and `removeExpressionResolver(...)` `Datastore` methods can be used.

See below for more informations about `ExpressionResolver` usage.

3.16.2. Property configuration and converters

The `Datastore` operations fully support property values conversions using the standard `Converters`. If a property declares a value converter, it will be used to perform conversions from the property type to the data model attribute type and back.

A converter can be used to adapt specific data types, use custom types or to face common conversion needs, such as *enumeration* property types mapped to integer or text data model types.

The `Configuration` structure can be used by a `Datastore` to detect the `TemporalType` associated to a `java.util.Date` or `java.util.Calendar` property type (through the `getTemporalType()` method), and ensure the consistency of the data manipulation and access operations which involve such kind of property.



The Holon platform **Datastore** fully supports the new **Java 8 Date and Time API**, which represents a big step forward compared to the previous date and time support classes, to address the shortcomings of the older `java.util.Date` and `java.util.Calendar` types. It is strongly recommended to use the new `java.time.*` types for date and time properties, such as `LocalDate`, `LocalTime`, `LocalDateTime` and so on. This way, in addition to achieving a more robust and consistent code, there is no need to use the `TemporalType` property configuration attribute to ensure consistency in property value handling.

3.16.3. DataTarget

The **DataTarget** interface is used by **Datastore** and **Query** to refer to an *entity* of the persistence model in an abstract and independent way from the concrete persistence layer. From the **DataTarget** point of view, an *entity* has the meaning of a *collection of data model attributes*, and it is represented by a **Path**, i.e. by a symbolic **name**.

Examples of **DataTarget** representations are:

- The name of a *table* in a RDBMS
- The class of a JPA **Entity**
- The document *collection* name in a document-oriented database

Concrete **Datastore** implementations could provide more specific **DataTarget** object types to identify an *entity* by using specific target types for the concrete persistence model which represent.

The **DataTarget** interface provides static methods to create data targets using the default name representation:

```
DataTarget<String> target = DataTarget.named("test");
```

All the **Datastore** operations involving persistent *entity* structures use a **DataTarget** to identify the target persistent structure and **PropertyBox** to provide and obtain the data model attribute values.

3.16.4. Data manipulation

The **Datastore** interface provides the most common data manipulation operations, listed in the table below.



Each operation throws a **DataAccessException** if an error occurs during operation execution.

Each operation supports configurable write **options**, represented by the `WriteOption` marker interface. Typically, write options are specific of the underlying persistence model and each concrete **Datastore** implementation provides a set of suitable write options. See the specific datastores documentation for further information.

The **DefaultWriteOption** enumeration provides the write options available for any datastore. By

now, a single default write option is defined:

BRING_BACK_GENERATED_IDS: Bring back any auto-generated id value into the `PropertyBox` which was subject of a data manipulation operation, if a corresponding `Property` (using the property name) is available in the box property set.



This option is only meaningful for datastores bound to a persistence engine which provides *id* auto-generation.

| Operation | Purpose | Return |
|--|---|---|
| <code>refresh(DataTarget target, PropertyBox propertyBox)</code> | Refresh the values of the properties of given <code>PropertyBox</code> , reloading them from the persistence store | The refreshed <code>PropertyBox</code> |
| <code>insert(DataTarget target, PropertyBox propertyBox, WriteOption... options)</code> | Insert a new data <i>entity</i> , identified by given target and represented by given <code>PropertyBox</code> , into the persistence store | The <code>OperationResult</code> . If one or more data model attribute was auto-generated by the concrete persistence store, such values are returned by the <code>getInsertedKeys</code> method. |
| <code>update(DataTarget target, PropertyBox propertyBox, WriteOption... options)</code> | Update an existing data <i>entity</i> , identified by given target and represented by given <code>PropertyBox</code> , into the persistence store | The <code>OperationResult</code> . |
| <code>save(DataTarget target, PropertyBox propertyBox, WriteOption... options)</code> | Insert a new data <i>entity</i> (identified by given target and represented by given <code>PropertyBox</code>) into the persistence store if the <i>entity</i> does not exists, or update it if the <i>entity</i> is present in the persistence store. | The <code>OperationResult</code> . |
| <code>delete(DataTarget target, PropertyBox propertyBox, WriteOption... options)</code> | Remove a data <i>entity</i> , identified by given target and represented by given <code>PropertyBox</code> , from the persistence store | The <code>OperationResult</code> . |
| <code>bulkInsert(DataTarget target, PropertySet<?> propertySet, WriteOption... options)</code> | Obtain an interface which can be used to perform a <i>bulk</i> insert of data <i>entities</i> identified by given target , represented by <code>PropertyBox</code> instances. Only the properties contained in given <code>propertySet</code> will be taken into account to perform insert operations. | The <code>BulkInsert</code> interface to provide the <code>PropertyBox</code> instances to insert and execute the bulk operation. |

| Operation | Purpose | Return |
|--|---|---|
| <code>bulkUpdate(DataTarget target, WriteOption... options)</code> | Obtain an interface which can be used to perform a <i>bulk</i> update of data <i>entities</i> identified by given target , to change a set of property values according to a set of restriction predicates to identify the set of data <i>entities</i> to update | The BulkUpdate interface to provide the properties and the value to update, the restriction predicates, and execute the bulk operation. |
| <code>bulkDelete(DataTarget target, WriteOption... options)</code> | Obtain an interface which can be used to perform a <i>bulk</i> delete of data <i>entities</i> identified by given target , providing a set of restriction predicates to identify the set of data <i>entities</i> to remove | The BulkDelete interface to provide the PropertyBox instances, configure the restriction predicates and execute the bulk operation. |

Datastore operations examples

```

final PathProperty<String> A_PROPERTY = PathProperty.create("propertyPath", String
.class);
final DataTarget<String> TARGET = DataTarget.named("test");

final Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
implementation

PropertyBox data = PropertyBox.builder(A_PROPERTY).set(A_PROPERTY, "aValue").build();

OperationResult result = datastore.save(TARGET, data); ①

result = datastore.insert(TARGET, data); ②
result = datastore.update(TARGET, data); ③

PropertyBox refreshed = datastore.refresh(TARGET, data); ④
datastore.delete(TARGET, refreshed); ⑤

// Bulk operations
result = datastore.bulkInsert(TARGET, PropertySet.of(A_PROPERTY))
    .add(PropertyBox.builder(A_PROPERTY).set(A_PROPERTY, "aValue1").build())
    .add(PropertyBox.builder(A_PROPERTY).set(A_PROPERTY, "aValue2").build())
    .add(PropertyBox.builder(A_PROPERTY).set(A_PROPERTY, "aValue3").build()).execute(
); ⑥

result = datastore.bulkUpdate(TARGET).set(A_PROPERTY, "updated").filter(A_PROPERTY
.isNull()).execute(); ⑦

result = datastore.bulkDelete(TARGET).filter(A_PROPERTY.isNull()).execute(); ⑧

```

- ① Save the **PropertyBox** containing given property value using the defined **DataTarget** (insert a new *entity* if not present in the persistence store or update it if exists, using the specific persistence

model strategy to identify entities)

- ② Insert the given **PropertyBox** data into the persistence store using the defined **DataTarget**
- ③ Update the given **PropertyBox** data into the persistence store using the defined **DataTarget**
- ④ Refresh the **PropertyBox** using the defined **DataTarget**, reloading data attribute values
- ⑤ Remove the *entity* which corresponds to given **PropertyBox** (using the specific persistence model strategy to identify entities)
- ⑥ Execute a *bulk* insert operation using the defined **DataTarget**, inserting given **PropertyBox** elements
- ⑦ Execute a *bulk* update operation using the defined **DataTarget**, setting the property value to **updated** when that property value is null
- ⑧ Execute a *bulk* delete operation using the defined **DataTarget**, removing entities for which the given property value is null

3.17. Query

The **Query** interface can be used to execute queries on the persistence data structures, using **Properties** to refer to the data model attributes and **PropertyBoxes** to represent and obtain the query results.

Using the **Query** abstraction, a query can be performed on the data model in a generic and implementation-independent way.

A **Query** supports the following clauses and configuration attributes:

- The **DataTarget** on which the query has to be performed
- The query **restrictions**, expressed as **QueryFilter** clauses
- The query results **sorting**, expressed as **QuerySort** clauses
- The query results **aggregation**, expressed as **Aggregations** clauses
- The query results **paging**, providing a result set *limit* and *offset*
- Generic query configuration **parameters**
- Support for **DataTargetResolver** and **[QueryClauseResolver]** registration

The query clauses which involve the reference to data model attributes (filtering, sorting, aggregation) are expressed using the **Properties** abstraction.

3.17.1. Query definition

A **Query** instance is obtained using the the **Datastore** **query()** method.

The **Query** interface provides methods to configure and define the query configuration and clauses, using a fluent builder pattern. The query configuration setting and clauses definitions are described below.

QueryFilter

The [QueryFilter](#) interface represents a query results restriction.

Most common restriction predicates representations are provided by the core platform classes. The following predicates are available:

- **Is null / is not null:** The value of a property is *null* / not *null*
- **equal / not equal:** The value of a property is equal / not equal to a given value
- **less than / less than or equal:** The value of a property is less than / less than or equal to a given value
- **greater than / greater than or equal:** The value of a property is greater than / greater than or equal to a given value
- **between:** The value of a property is included between a minimum and a maximum value
- **in / not in:** The value of a property is present / not present in a set of values
- **contains:** The value of a [String](#) property contains given text, ignoring case or not
- **startsWith:** The value of a [String](#) property contains starts with text, ignoring case or not
- **endsWith:** The value of a [String](#) property contains ends with text, ignoring case or not

Negation, conjunction and disjunction of predicates:

- **not:** Negation of a predicate
- **and:** Conjunction of predicates (represents the [AND](#) logical operation)
- **or:** Disjunction of predicates (represents the [OR](#) logical operation)

The [QueryFilter](#) predicates can be obtained in two ways:

1. Using the static methods provided by the [QueryFilter](#) interface, specifying the [Properties](#) the predicate refers to when required.

```
final PathProperty<String> PROPERTY = PathProperty.create("test", String.class);
final PathProperty<String> ANOTHER_PROPERTY = PathProperty.create("another", String.class);

QueryFilter restriction = QueryFilter.isNotNull(PROPERTY); // is not null
restriction = QueryFilter.isNull(PROPERTY); // is null
restriction = QueryFilter.eq(PROPERTY, "value"); // equal to a value
restriction = QueryFilter.eq(PROPERTY, ANOTHER_PROPERTY); // equal to a property
restriction = QueryFilter.neq(PROPERTY, "value"); // not equal
restriction = QueryFilter.lt(PROPERTY, "value"); // less than
restriction = QueryFilter.loeq(PROPERTY, "value"); // less than or equal
restriction = QueryFilter.gt(PROPERTY, "value"); // greater than
restriction = QueryFilter.goeq(PROPERTY, "value"); // greater than or equal
restriction = QueryFilter.between(PROPERTY, "value1", "value2"); // between
restriction = QueryFilter.in(PROPERTY, "value1", "value2", "value3"); // in
restriction = QueryFilter.nin(PROPERTY, "value1", "value2", "value3"); // not in
restriction = QueryFilter.startsWith(PROPERTY, "v", false); // starts with 'v'
restriction = QueryFilter.startsWith(PROPERTY, "v", true); // starts with 'v',
ignoring case
restriction = QueryFilter.endsWith(PROPERTY, "v", false); // ends with 'v'
restriction = QueryFilter.contains(PROPERTY, "v", false); // contains 'v'
QueryFilter restriction2 = QueryFilter.contains(PROPERTY, "v", true); // contains 'v',
ignoring case

// negation
QueryFilter negation = QueryFilter.not(restriction);
negation = restriction.not();

// conjunction
QueryFilter conjunction = restriction.and(restriction2);
conjunction = QueryFilter.allOf(restriction, restriction2).orElse(null);

// disjunction
QueryFilter disjunction = restriction.or(restriction2);
disjunction = QueryFilter.anyOf(restriction, restriction2).orElse(null);
```

2. For [Properties](#) related predicates, using directly the static methods provided by the [Property](#) interface.



`QueryFilter` s can be added to a `Query` using the `filter(QueryFilter filter)` query builder method, which adds a filter predicate to the query. If a `QueryFilter` has already been associated to the query, the new filter is added to the query in conjunction with the existing predicates, i.e. using a **AND** logical operation.

Examples using Property

```
final PathProperty<String> PROPERTY = PathProperty.create("test", String.class);
final PathProperty<String> ANOTHER_PROPERTY = PathProperty.create("another", String.class);

QueryFilter restriction = PROPERTY.isNotNull(); // is not null
restriction = PROPERTY.isNull(); // is null
restriction = PROPERTY.eq("value"); // equal to a value
restriction = PROPERTY.eq(ANOTHER_PROPERTY); // equal to a property
restriction = PROPERTY.neq("value"); // not equal
restriction = PROPERTY.lt("value"); // less than
restriction = PROPERTY.loeq("value"); // less than or equal
restriction = PROPERTY.gt("value"); // greater than
restriction = PROPERTY.goeq("value"); // greater than or equal
restriction = PROPERTY.between("value1", "value2"); // between
restriction = PROPERTY.in("value1", "value2", "value3"); // in
restriction = PROPERTY.nin("value1", "value2", "value3"); // not in
restriction = PROPERTY.startsWith("v"); // starts with
restriction = PROPERTY.startsWithIgnoreCase("v"); // starts with ignoring case
restriction = PROPERTY.endsWith("v"); // ends with
restriction = PROPERTY.endsWithIgnoreCase("v"); // ends with ignoring case
restriction = PROPERTY.contains("v"); // contains
QueryFilter restriction2 = PROPERTY.containsIgnoreCase("v"); // contains ignoring case

// negation
QueryFilter negation = PROPERTY.eq("value").not();

// conjunction
QueryFilter conjunction = PROPERTY.isNotNull().and(PROPERTY.eq("value"));

// disjunction
QueryFilter disjunction = PROPERTY.isNull().or(PROPERTY.eq("value"));
```

QuerySort

The [QuerySort](#) interface represents a query results sorting directive, referred to a [Properties](#).

The [SortDirection](#) enumeration is used to declare the sort direction (ascending or descending).

A list of [QuerySort](#) can be composed to create a sort declaration which involves more than one [Property](#), for each declaring the sort direction.

The [QuerySort](#) directives can be obtained in two ways:

1. Using the static methods provided by the [QuerySort](#) interface, specifying the [Properties](#) to which the sort refers.

Examples using QuerySort

```
final PathProperty<String> PROPERTY = PathProperty.create("test", String.class);
final PathProperty<String> ANOTHER_PROPERTY = PathProperty.create("another", String.class);

QuerySort sort = QuerySort.asc(PROPERTY); // sort ASCENDING on given property
sort = QuerySort.desc(PROPERTY); // sort DESCENDING on given property

QuerySort sort2 = QuerySort.of(ANOTHER_PROPERTY, SortDirection.ASCENDING); // sort
ASCENDING on given property
sort2 = QuerySort.of(ANOTHER_PROPERTY, true); // sort ASCENDING on given property

QuerySort.of(sort, sort2); // sort using 'sort' and 'sort2' declarations, in the given
order
```

2. Using directly the static methods provided by the `Property` interface.

Examples using Property

```
final PathProperty<String> PROPERTY = PathProperty.create("test", String.class);
final PathProperty<String> ANOTHER_PROPERTY = PathProperty.create("another", String.class);

QuerySort sort = PROPERTY.asc(); // sort ASCENDING on given property
sort = PROPERTY.desc(); // sort DESCENDING on given property

PROPERTY.asc().and(ANOTHER_PROPERTY.desc()); // sort ASCENDING on PROPERTY, than sort
DESCENDING on
// ANOTHER_PROPERTY
```



`QuerySort` can be added to a `Query` using the `sort(QuerySort sort)` query builder method, which adds a sort declaration to the query. If one or more `QuerySort` has already been associated to the query, the new sort is added at the end of the query sorts list.

Aggregations

Aggregation clause:

A `Query` supports a results aggregation clause, represented by the `QueryAggregation` interface. This interface provides a fluent builder to create aggregation clauses.

The aggregation clause can be used to specify:

- Which `Path`'s must be used to aggregate the query results, i.e. **grouping the results** by the values of the specified paths;
- The **optional restrictions** to apply on the aggregation path values, expressed by using standard `QueryFilter` predicates.



Query results aggregation semantics can be slightly different from one Datastore implementation to another. Each Datastore implementation should ensure a consistent query execution behaviour, but in some situations it may not be possible to perform the aggregation operation for some query configurations. For example, many RDBMS engines do not allow to project a query result which is not part of the query aggregation clause unless an aggregation function is used.

Examples of aggregation

```
final PathProperty<Integer> PROPERTY = PathProperty.create("test", Integer.class);
final PathProperty<String> ANOTHER_PROPERTY = PathProperty.create("another", String.class);

Datastore datastore = getDatastore(); // build or obtain a concrete Datastore implementation

Stream<PropertyBox> results = datastore.query().target(DataTarget.named("testTarget"))
    .aggregate(PROPERTY)
    .stream(PROPERTY, ANOTHER_PROPERTY.max()); ①

results = datastore.query().target(DataTarget.named("testTarget"))
    .aggregate(QueryAggregation.builder().path(PROPERTY).filter(PROPERTY.isNull())
    .build())
    .stream(PROPERTY, ANOTHER_PROPERTY.max()); ②
```

- ① Aggregate the results grouping by **PROPERTY**
- ② Aggregate the results grouping by **PROPERTY** and restrict the aggregation property values to not *null* values

Aggregation functions:

A set of basic results aggregation *functions* are provided by the default **Datastore**. An aggregation function is related to a **Path** and allows to aggregate the query results related to such path with a specific aggregation semantic.

A **Datastore function** is represented by the `../api/holon-core/com/holonplatform/core/query/QueryFunction.html[QueryFunction^]` interface and the following builtin aggregation functions are provided:

- `../api/holon-core/com/holonplatform/core/query/QueryFunction.Count.html[Count^]` : counts the results
- `../api/holon-core/com/holonplatform/core/query/QueryFunction.Min.html[Min^]` : return the smallest value
- `../api/holon-core/com/holonplatform/core/query/QueryFunction.Max.html[Max^]` : return the larger value
- `../api/holon-core/com/holonplatform/core/query/QueryFunction.Avg.html[Avg^]` : return the average value

- `../api/holon-core/com/holonplatform/core/query/QueryFunction.Sum.html[Sum^]` : return the sum of the values

An aggregation function can be used as a query projection or expression, and can be created in two ways:

1. Using the static methods provided by the `FunctionExpression` interface, by specifying the `Path` to apply the function to.
2. Using directly the static methods provided by the `PathProperty` interface.

Examples of aggregation functions

```
final PathProperty<Integer> PROPERTY = PathProperty.create("test", Integer.class);

// Using AggregationProperty
FunctionExpression<Long> expression = FunctionExpression.count(PROPERTY);

// Using the property
FunctionExpressionProperty<Integer> sp = PROPERTY.sum();
```

Paging results

`Query` supports results **pagination**, allowing to set:

- **limit**:: results limit, i.e. the max number of results to provide
- **offset**:: 0-based offset from which to fetch the query results within the total results set

Pagination example

```
final PathProperty<Integer> PROPERTY = PathProperty.create("test", Integer.class);

Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
implementation

Stream<Integer> values = datastore.query().target(DataTarget.named("testTarget"))
    .limit(100).offset(0)
    .stream(PROPERTY); ①
values = datastore.query().target(DataTarget.named("testTarget")).limit(100).offset
(100).stream(PROPERTY); ②
```

- ① Limit the query results to **100** and fetch the results starting from **0** (i.e. fetch the first page where a page is a window of 100 results)
- ② Limit the query results to **100** and fetch the results starting from **100** (i.e. fetch the second page where a page is a window of 100 results)

Query results projection

`Query` results are obtained using a *projection*, represented by the `QueryProjection` interface.

The `Query` interface extends the `QueryResults` interface, which provides methods to execute the

query and obtain the query results.

Two main methods are provided to obtain the query results:

- `stream(QueryProjection<R> projection)`: Obtain the query results as a `Stream`, using given projection to define the type of the objects contained in the stream
- `count()`: Count the results of the query

Two additional convenience methods are made available by the `QueryResults` interface:

- `findOne(QueryProjection<R> projection)`: Get a single expected result from the query, which type matches the projection type. If more than one result is obtained from query execution, a `QueryNonUniqueResultException` is thrown
- `list(QueryProjection<R> projection)`: Convenience method to obtain query results `Stream` as a `List`

Builtin projections:

The platform provides the following query projections:

- `PathProperty` and `AggregationProperty` are query projections, mapping the result type into the property type
- `PropertySetProjection` uses a set of `Property` to map the query results into `PropertyBox` objects containing the values of the properties of the set returned by the query execution.

The `QueryResults` interface provides convenience methods helpful to use the `PropertySetProjection` query projection:

- `stream(Iterable<P> properties)` and `stream(Property... properties)`: accept a set of `Properties` and return the query results as a stream of `PropertyBox`, each containing the values of the properties of the set obtained from query execution
- `list(Iterable<P> properties)`: Same as above, but returning the results as a `List` instead of a `Stream`
- `findOne(Iterable<P> properties)` and `findOne(Property... properties)`: accept a set of `Properties` and return the unique query result as an Optional `PropertyBox` containing the values of the properties of the set obtained from query execution



Any concrete `Datastore` implementation could provide additional projections related to the specific Datastore persistence model.

Query projection examples

```
final PathProperty<Integer> PROPERTY = PathProperty.create("test", Integer.class);
final PathProperty<String> ANOTHER_PROPERTY = PathProperty.create("another", String.class);

Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
implementation

Stream<Integer> values = datastore.query().target(DataTarget.named("testTarget"))
    .stream(PROPERTY); ①
Optional<Integer> value = datastore.query().target(DataTarget.named("testTarget"))
    .findOne(PROPERTY); ②
Stream<PropertyBox> boxes = datastore.query().target(DataTarget.named("testTarget"))
    .stream(PROPERTY,
        ANOTHER_PROPERTY); ③
List<PropertyBox> list = datastore.query().target(DataTarget.named("testTarget"))
    .list(PROPERTY,
        ANOTHER_PROPERTY); ④
Optional<PropertyBox> box = datastore.query().target(DataTarget.named("testTarget"))
    .findOne(PROPERTY,
        ANOTHER_PROPERTY); ⑤
Optional<Integer> sum = datastore.query().target(DataTarget.named("testTarget"))
    .findOne(PROPERTY.sum()); ⑥
```

- ① Stream of values using **PROPERTY** as projection. The Stream type is the same of the property type (Integer)
- ② Optional unique value using **PROPERTY** as projection. The value type is the same of the property type (Integer)
- ③ Stream of **PropertyBox** containing the values of the set of properties composed by **PROPERTY** and **ANOTHER_PROPERTY**
- ④ List of **PropertyBox** containing the values of the set of properties composed by **PROPERTY** and **ANOTHER_PROPERTY**
- ⑤ Optional unique **PropertyBox** containing the values of the set of properties composed by **PROPERTY** and **ANOTHER_PROPERTY**
- ⑥ Unique value of the **sum** aggregation function applied to **PROPERTY**

Bean projection:

The **BeanProjection** interface can be used to obtain the query results as **Java Bean** instances, providing the mapping bean class to be used. The query results are mapped to the bean class properties by matching the query selection **Path** names with the bean property names.

```
class MyBean {

    private Integer code;
    private String text;

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

}

public void beanProjection() {
    Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
    implementation

    Stream<MyBean> results = datastore.query().target(DataTarget.named("testTarget"))
        .stream(BeanProjection.of(MyBean.class)); ①
    Optional<MyBean> result = datastore.query().target(DataTarget.named("testTarget"))
        .findOne(BeanProjection.of(MyBean.class)); ②

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

    results = datastore.query().target(DataTarget.named("testTarget"))
        .stream(BeanProjection.of(MyBean.class, CODE, TEXT)); ③
}
```

- ① Stream of values using **MyBean** as projection, using all the bean property names as query selection
- ② Optional unique value using **MyBean** as projection, using all the bean property names as query selection
- ③ Stream of values using **MyBean** as projection, using provided **CODE** and **TEXT** properties as query selection

3.17.2. Configuration

The `DatastoreConfigProperties` interface represents and provides the available configuration properties which can be used to configure a generic `Datastore`.

The interface extends a default `ConfigPropertySet` bound to the property name prefix **holon.datastore**.

The available configuration properties are listed below:

Table 2. Datastore configuration properties

| Name | Type | Meaning |
|------------------------------------|--|--|
| <code>holon.datastore.trace</code> | Boolean (<code>true</code> / <code>false</code>) | Enable/disable Datastore operations tracing in log (for example, the concrete query definitions) |

The `DatastoreConfigProperties` can be loaded from a number of sources using the default `ConfigPropertySet` builder interface:

```
DatastoreConfigProperties config = DatastoreConfigProperties.builder()
    .withDefaultPropertySources().build(); ①

config = DatastoreConfigProperties.builder().withSystemPropertySource().build(); ②

Properties props = new Properties();
props.put("holon.datastore.trace", "true");
config = DatastoreConfigProperties.builder().withPropertySource(props).build(); ③

config = DatastoreConfigProperties.builder().withPropertySource("datastore.properties")
    .build(); ④
```

- ① Read the configuration properties from *default* property sources (i.e. the `holon.properties` file)
- ② Read the configuration properties from `System` properties
- ③ Read the configuration properties from a `Properties` instance
- ④ Read the configuration properties from the `datastore.properties` file

Multiple Datastore configuration

When multiple `Datastore` configuration is required and properties are read from the same source, a *data context id* can be used to discern one `Datastore` configuration property set from another.

From the property source point of view, the *data context id* is used as a **suffix** after the configuration property set name (`holon.datastore`) and before the specific property name.

For example, let's say we have a configuration property set for two different datastores as follows:

```
holon.datastore.one.trace=true  
  
holon.datastore.two.trace=false
```

In order to provide the configuration for two **Datastore** instances, one bound to the **one** configuration property set and the other bound to the **two** configuration property set, the **DatastoreConfigProperties** can be obtained as follows, specifying the *data context id* when obtaining the builder:

```
DatastoreConfigProperties config1 = DatastoreConfigProperties.builder("one")  
    .withPropertySource("datastore.properties").build();  
  
DatastoreConfigProperties config2 = DatastoreConfigProperties.builder("two")  
    .withPropertySource("datastore.properties").build();
```

3.17.3. Common Datastore configuration properties

The **DatastoreConfigProperties** interface is a **ConfigPropertySet** bound to the property name prefix **holon.datastore**, which provides common **Datastore** configuration properties:

| Name | Type | Meaning |
|--------------------------------|----------------------|--|
| <i>holon.datastore.trace</i> | Boolean (true/false) | Enable or disable Datastore operations tracing (for example, logging the concrete query/operations executed in native persistence store language). |
| <i>holon.datastore.dialect</i> | String | The fully qualified class name of the <i>dialect</i> to be used, if the concrete Datastore supports dialects. |

3.17.4. Relational Datastores

Regarding the *relational* **Datastore** implementations, i.e. Datastores bound to a **RDBMS**, some additional components are provided to express typical *relational* concepts concerning query definition and execution.

Sub-query

The **SubQuery** interface can be used to represent a *sub-query*, which can be used in a query definition to express query restrictions (filters) that involve a sub-query as filter operand.

To create a **SubQuery**, the **create(...)** static method of the **SubQuery** interface can be used. The **SubQuery** definition process (target, restrictions, ordering and so on) is the same of a normal **Query** definition process, sharing the same query builder interface. In addition, a **SubQuery** must provide a **QueryProjection** to define the sub query selection type.

A **SubQuery** is a **QueryExpression**, allowing to use it as a **QueryFilter** operand.



When a **SubQuery** is used in a query, to avoid property/column names ambiguity, it is strongly recommended to provide a **parent DataTarget** for the query properties. The parent **DataTarget** of a **Property** can be setted using the **parent(...)** method of the property builder or directly using the **property(...)** methods provided by the **DataTarget** interface to create a **Property** with the given **DataTarget** as parent.

```
Datastore datastore = getDatastore(); // this is supposed to be a relational Datastore implementation
```

```
final DataTarget TARGET1 = DataTarget.named("testTarget1");  
final PathProperty<Integer> PROPERTY1 = TARGET1.property("test", Integer.class);
```

```
final DataTarget TARGET2 = DataTarget.named("testTarget2");  
final PathProperty<Integer> PROPERTY2 = TARGET2.property("test", Integer.class);
```

```
SubQuery<Integer> subQuery = SubQuery.create(datastore).target(TARGET2).filter  
(PROPERTY1.goe(1))  
    .select(PROPERTY1); ①
```

```
Stream<Integer> results = datastore.query().target(TARGET1).filter(PROPERTY2.in  
(subQuery)).stream(PROPERTY2); ②
```

① Create a **SubQuery**

② Use the **SubQuery** as the right operand of a **IN** query filter

Two convenience methods are provided by the **SubQuery** interface to create **EXISTS** and **NOT EXISTS** filter predicates. In this case, the sub-query selection projection is not required, and by default a **1** literal value will be used as selection.

```

Datastore datastore = getDatastore(); // this is supposed to be a relational Datastore
implementation

final DataTarget TARGET1 = DataTarget.named("testTarget1");
final PathProperty<Integer> PROPERTY1 = TARGET1.property("test", Integer.class);

final DataTarget TARGET2 = DataTarget.named("testTarget2");
final PathProperty<Integer> PROPERTY2 = TARGET2.property("test", Integer.class);

Stream<Integer> results = datastore.query().target(TARGET1)
    .filter(SubQuery.create(datastore).target(TARGET2).filter(PROPERTY2.eq(PROPERTY1))
    .exists())
    .stream(PROPERTY2); ①

results = datastore.query().target(TARGET1)
    .filter(SubQuery.create(datastore).target(TARGET2).filter(PROPERTY2.eq(PROPERTY1))
    .notExists())
    .stream(PROPERTY2); ②

```

- ① A query with a filter using a **EXISTS** *SubQuery* predicate
- ② A query with a filter using a **NOT EXISTS** *SubQuery* predicate

Alias and Joins

The *RelationalTarget* interface can be used to express **alias** and **joins** for a *DataTarget*.

A *RelationalTarget* is a *DataTarget* itself, and provides methods to assign an **alias** name to the query target and to create **joins** with other targets.

```

final DataTarget<String> TARGET = DataTarget.named("testTarget");

RelationalTarget<String> RT = RelationalTarget.of(TARGET); ①
RelationalTarget<String> RT2 = RT.alias("aliasName"); ②

```

- ① Create a *RelationalTarget* using given *TARGET*
- ② Create a new *RelationalTarget* from the previous one, assigning an alias name to it

The following join types are supported:

- **INNER JOIN**: returns all rows when there is at least one match in BOTH tables represented by the source *DataTarget* and the joined *DataTarget*;
- **LEFT JOIN**: returns all rows from the left table (represented by the source *DataTarget*), and the matched rows from the right table (represented by the joined *DataTarget*);
- **RIGHT JOIN**: returns all rows from the right table (represented by the joined *DataTarget*), and the matched rows from the left table (represented by the source *DataTarget*);



Only one level of joins is supported, i.e. only the source **DataTarget** represented by the **RelationalTarget** can be joined with other targets, and no *sub* join is supported.

The **Join** interface represents the join expression, supporting an **alias** name definition and a **ON** clause definition, to express any join restriction/filter predicate.

A **RelationalTarget** is created from a conventional **DataTarget** using the `of(DataTarget target)` static method.



When joins are used in a query, to avoid property/column names ambiguity, it is strongly recommended to provide a **parent DataTarget** for the query properties. The parent **DataTarget** of a **Property** can be setted using the `parent(...)` method of the property builder or directly using the `property(...)` methods provided by the **DataTarget** interface to create a **Property** with the given **DataTarget** as parent.

```
final DataTarget TARGET1 = DataTarget.named("testTarget1");
final PathProperty<Integer> PROPERTY1 = TARGET1.property("test", Integer.class);

final DataTarget TARGET2 = DataTarget.named("testTarget2");
final PathProperty<Integer> PROPERTY2 = TARGET2.property("test", Integer.class);

RelationalTarget<String> RT = RelationalTarget.of(TARGET1) ①
    .join(TARGET2, JoinType.INNER).on(PROPERTY2.eq(PROPERTY1)).add(); ②

RT = RelationalTarget.of(TARGET1).innerJoin(TARGET2).on(PROPERTY2.eq(PROPERTY1)).add(
); ③
RT = RelationalTarget.of(TARGET1).leftJoin(TARGET2).on(PROPERTY2.eq(PROPERTY1)).add();
④
RT = RelationalTarget.of(TARGET1).rightJoin(TARGET2).on(PROPERTY2.eq(PROPERTY1)).add(
); ⑤

Stream<Integer> results = getDatastore().query().target(RT).stream(PROPERTY1); ⑥
```

- ① Create a **RelationalTarget** using **TARGET1**
- ② Join (using a **INNER** join type) the **TARGET1** with the **TARGET2**, using a **ON** clause to express the join condition
- ③ Join (using a **INNER** join type) the **TARGET1** with the **TARGET2**, using a **ON** clause to express the join condition
- ④ Join (using a **LEFT** join type) the **TARGET1** with the **TARGET2**, using a **ON** clause to express the join condition
- ⑤ Join (using a **RIGHT** join type) the **TARGET1** with the **TARGET2**, using a **ON** clause to express the join condition
- ⑥ Use the created **RelationalTarget** as a query target

3.17.5. Datastore extension

Datastores provides two main entry points for extension purposes:

- **Datastore** components and operations extension through [ExpressionResolver](#), which can be used to add custom expressions in order to extend or modify the data manipulation operations and query definition and execution strategies.
- Additional **Datastore** operations and functionalities relying on the [DatastoreCommodity](#) concept, through the registration of a [DatastoreCommodityFactory](#).

Common **ExpressionResolver** s

For the most common query use cases, a set of builtin **ExpressionResolver** are made available to extend the main query clauses, providing custom clause representations to be translated into a standard query clause which the **Datastore** can handle.

An **ExpressionResolver** can be registered or unregistered in a **Datastore** implementation using the `addExpressionResolver(...)` and `removeExpressionResolver(...)` methods.



See concrete **Datastore** implementations documentation for additional information about any other **ExpressionResolver** based extension capabilities and resolvers registration options.

DataTargetResolver

A [DataTargetResolver](#) can be defined and registered in **Datastore** to *resolve* a **DataTarget** unknown to the concrete Datastore implementation, by translating it into one which the Datastore can recognize and handle.

Typically, a **DataTargetResolver** can be defined to resolve a **DataTarget** with a symbolic name into a specific Datastore data target.

DataTargetResolver example

```
ExpressionResolver resolver = DataTargetResolver.create(DataTarget.class,
    (target, context) -> "test".equals(target.getName())
        ? Optional.of(DataTarget.named("wellKnownTargetName")) : Optional.empty()); ①

Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
implementation
datastore.addExpressionResolver(resolver); ②
```

① Create a resolver which translates the symbolic **test** name in another named target with the **wellKnownTargetName** name

② Register the resolver in the Datastore

Custom QueryFilters

You can define and register custom **QueryFilter** implementations, to provide new predicates as a

combination of simple standard predicates, possibly relying on parameters or application specific conditions, or to express specific filtering conditions related to a concrete persistence model to which a `Datastore` is bound.

A custom `QueryFilter` can be registered in a `Datastore` using standard [Expressions and resolvers](#) interface.

A convenience `QueryFilterResolver` interface is provided to facilitate the creation of a `QueryFilter` resolver.

A typical custom `QueryFilter` definition process takes place with the following steps:

1. First of all, you have to define your custom filter representation, providing a class which implements the `QueryFilter` interface (and, optionally, an interface which extends `QueryFilter` and represents your custom filter API);
2. Then create a class which implements `QueryFilterResolver`, generalized on your custom filter class/interface, whose purpose is to resolve the custom filter, transforming it into a `QueryFilter` that can be handled by the concrete `Datastore`. Two examples are: one of the builtin `QueryFilter` as described above, or a persistence model specific filter type recognized by a specific `Datastore` implementation.
3. Finally, register the `QueryFilterResolver` in the `Datastore` instance, using the `addExpressionResolver(...)` method.

When the custom filter is defined and registered, it can be used anywhere in the *query* or *bulk* operations clauses obtained from the `Datastore` in which the resolver is registered.

```
class MyFilter implements QueryFilter { ❶

    final PathProperty<String> property;
    final String value;

    public MyFilter(PathProperty<String> property, String value) {
        this.property = property;
        this.value = value;
    }

    @Override
    public void validate() throws InvalidExpressionException {
        if (value == null)
            throw new InvalidExpressionException("Value must be not null");
    }
}

class MyFilterResolver implements QueryFilterResolver<MyFilter> { ❷

    @Override
    public Class<? extends MyFilter> getExpressionType() {
        return MyFilter.class;
    }

    @Override
    public Optional<QueryFilter> resolve(MyFilter expression, ResolutionContext context)
        throws InvalidExpressionException {
        return Optional
            .of(expression.property.isNotNull().and(expression.property.contains
(expression.value, true))); ❸
    }
}

final static PathProperty<String> PROPERTY = PathProperty.create("testProperty",
String.class);

public void customFilter() {
    Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
implementation
    datastore.addExpressionResolver(new MyFilterResolver()); ❹

    Stream<String> results = datastore.query().target(DataTarget.named("test"))
        .filter(PROPERTY.isNotNull().and(new MyFilter(PROPERTY, "testValue"))).stream
(PROPERTY); ❺
}
```

- ② Custom filter resolver class
- ③ The resolver translates a `MyFilter` into a predicate composed by well-known standard `QueryFilter`
- ④ The resolver is registered in the `Datastore`, enabling the use of the `MyFilter` type filter in query and bulk operations clauses
- ⑤ Use of a `MyFilter` in a query execution

Custom QuerySorts

A custom `QuerySort` can be registered in a `Datastore` using standard [Expressions and resolvers](#) interface.

A convenience `QuerySortResolver` interface is provided to facilitate the creation of a `QuerySort` resolver.

A typical custom `QuerySort` definition process takes place with the following steps:

1. First of all, you have to define your custom sort representation, providing a class which implements the `QuerySort` interface (and, optionally, an interface which extends `QuerySort` and represents your custom filter API);
2. Then create a class which implements `QuerySortResolver`, generalized on your custom sort class/interface, whose purpose is to resolve the custom sort, transforming it into a `QuerySort` declaration which the concrete `Datastore` can handle. For example, one of the builtin `QuerySort` as described above, or a persistence model specific sort type recognized by a specific `Datastore` implementation.
3. Finally, register the `QuerySortResolver` in the `Datastore` instance, using the `addExpressionResolver(...)` method.

When the custom sort is defined and registered, it can be used anywhere in the *query* clauses obtained from the `Datastore` in which the resolver is registered.

```
class MySort implements QuerySort { ❶

    @Override
    public void validate() throws InvalidExpressionException {
    }

}

class MySortResolver implements QuerySortResolver<MySort> { ❷

    final PathProperty<String> P1 = PathProperty.create("testProperty1", String.class);
    final PathProperty<Integer> P2 = PathProperty.create("testProperty2", Integer.class);

    @Override
    public Class<? extends MySort> getExpressionType() {
        return MySort.class;
    }

    @Override
    public Optional<QuerySort> resolve(MySort expression, ResolutionContext context)
        throws InvalidExpressionException {
        return Optional.of(P1.asc().and(P2.desc())); ❸
    }

}

public void customSort() {
    Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
    implementation
    datastore.addExpressionResolver(new MySortResolver()); ❹

    Stream<String> results = datastore.query().target(DataTarget.named("test")).sort(new
    MySort()).stream(PROPERTY); ❺
}
```

- ❶ Custom sort definition, implementing **QuerySort**
- ❷ Custom sort resolver class
- ❸ The resolver translates a **MySort** into a sort composed by well-known standard **QuerySort**
- ❹ The resolver is registered in the **Datastore**, enabling the use of the **MySort** type sort in query clauses
- ❺ Use of a **MySort** in a query execution

Datastore commodities definition and registration

Using the **DatastoreCommodity** representation, a **Datastore** can be extended by adding new operations and functionalities, represented by a class which implements the **DatastoreCommodity**

interface.

A `DatastoreCommodity` must be provided using a `DatastoreCommodityFactory` implementation, which has to be registered in the target `Datastore` through the `registerCommodity(DatastoreCommodityFactory<X, C> commodityFactory)` method.

Each commodity factory is bound to a specific `DatastoreCommodity` type, provided by the `getCommodityType()` factory method, and can use a `DatastoreCommodityContext` to create and configure the commodity instance when requested. Typically, each concrete `Datastore` implementation provides a specific `DatastoreCommodityContext` extension, to provide useful `Datastore` context and configuration references.

A `DatastoreCommodity` can be obtained from `Datastore` using the `create(Class<C> commodityType)` method. A `DatastoreCommodityFactory` bound to the requested commodity type must be available, i.e. previously registered in `Datastore`, in order to obtain the commodity instance.

The `Query` object itself is a `DatastoreCommodity`, obtained by the convenience `Datastore query()` method. Each concrete `Datastore` implementation register a default factory to provide such commodity.



See concrete `Datastore` implementations documentation for additional information about any specific available *commodity* types and registration options.

3.17.6. Available Datastores

By now, the holon platform provides two default `Datastore` implementations:

- **JDBC Datastore:** using the **Java Database Connectivity (JDBC)** specification to access a relational database
- **JPA Datastore:** using the **Java Persistence API** specification to access a relational database

3.18. Multi tenancy support

The core Holon platform module provides the `TenantResolver` interface, which acts as default platform strategy representation to obtain the `String` which identifies the current **tenant** in a *multi-tenant* environment.

The interface provides a `getCurrent()` convenience method to obtain the current `TenantResolver` registered in `Context`, if available.

Other specific platform modules use this interface to provide their *multi-tenancy* related functionalities. See specific modules documentation for further details.

3.19. Utilities

The core Holon platform module provides some utility interfaces/classes which can be used in applications development.

3.19.1. Initializer

The `Initializer` interface can be used to perform a **lazy initialization** of a generic value (with the same type of the generic `Initializer` type) and provides some static methods to create `Initializer` implementations:

```
Initializer<String> intzr = Initializer.using(() -> "test"); ①  
String lazyInited = intzr.get(); ②
```

- ① Create an `Initializer` using a `Supplier` to provide the lazy-initialized value
- ② Only the first time the `get()` method is invoked, the value is initialized using given `Supplier` and then is returned to the caller

3.19.2. SizedStack

The `SizedStack` class is a `java.util.Stack` extension which supports a **max stack size**, given at construction time.

When the stack size exceeds the max size, the eldest element is removed before adding a new one on the top of the stack.

4. HTTP and REST

The `holon-http` artifact provides base **HTTP** protocol support to the Holon platform, dealing with *HTTP messages* and providing support for *RESTful* web services invocation through a *client* API.

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>  
<artifactId>holon-http</artifactId>  
<version>5.0.2</version>
```

4.1. HTTP messages

The Holon platform provides an implementation-independent representation of the **Hypertext Transfer Protocol** request and response messages, used by other platform modules to deal with HTTP-based operations.

The HTTP request and response message representations are based on the `Message` interface, which represent a generic *message* consisting of a map of message **headers** (identified by a textual header name) and a **payload** which represents the content delivered in the message.



For an HTTP message, the message header values are represented as a `List` of `Strings`, since HTTP supports multiple values for each header. The message *payload* is represented as a `String`.

4.1.1. HttpRequest

The [HttpRequest](#) interface represents a HTTP **request** message and provides the following informations and operations:

- The HTTP **method** describing the desired action to be performed
- The fully qualified name of the client host or the last proxy that sent the request
- The request message *path*
- The request URI query parameters, if any
- The request *cookies*, if any
- An [InputStream](#) of the request message body (payload)

4.1.2. HttpResponse

The [HttpResponse](#) interface represents a HTTP **response** message and provides the following informations and operations:

- The HTTP *status code* of the response, also represented with the convenience [HttpStatus](#) enumeration
- A *builder* to create default HttpResponse instances

4.2. RESTful client

The Holon platform provides an implementation-independent representation of a client to deal with a **RESTful** web services API, using the HTTP protocol.

The client provides a fluent *builder* to compose and execute a RESTful service invocation, using *template* variable substitution, supporting base authentication methods, common headers configuration and request entities definition.

The client is represented by the [RestClient](#) interface and its main features are:

- Support for a **default target request base URI**
- Support for **default request headers**
- Support for URI **template variable substitutions**
- Support for request URI **query parameters**
- Convenience methods to setup **common request message headers**, such as
 - Accepted response media types
 - Acceptable languages
 - Acceptable encodings
 - Acceptable charsets
 - Configure a [Cache-Control](#) header
- Convenience method to setup authorization headers([Basic](#) and [Bearer](#) types)

- Perform the request invocation with a specific HTTP *method* providing a request message entity
- Convenience methods to perform most common invocations using one of the **GET**, **POST**, **PUT**, **PATCH**, **DELETE**, **OPTIONS**, **TRACE** or **HEAD** methods

4.2.1. Obtain a **RestClient** instance

Concrete **RestClient** implementations are obtained from a **RestClientFactory**, registered using Java service extensions through a `com.holonplatform.http.rest.RestClientFactory` file under the **META-INF/services** folder.

A **RestClient** instance can be obtained using one of the `create(...)` methods provided by the interface, either specifying the *fully qualified* class name of the **RestClient** implementation to obtain or using the default implementation according to the available **RestClientFactory** within the current **ClassLoader** (a specific **ClassLoader** can be used instead of the current one).



If more than one **RestClientFactory** is bound to the same **RestClient** implementation type, or if more than one **RestClientFactory** is available in the **ClassLoader** when the implementation class is not specified, the **RestClientFactory** to use to build the **RestClient** instance is selected according to the factory priority level, which can be specified using the **Priority** annotation, if available.



The `forTarget(...)` static methods of the **RestClient** interface can be used as shorters to create a **RestClient** using the default implementation and setting a default base **URI** to use for the client requests.

RestClient creation examples

```
// Create a RestClient using the default available implementation for current
// ClassLoader
RestClient client = RestClient.create();

// Create a RestClient using a specific implementation class name
client = RestClient.create("com.holonplatform.jaxrs.client.JaxrsRestClient");

// Create a RestClient using the default available implementation and set a default
// base URI
client = RestClient.forTarget("https://host/api");
```

[[Available implementations]] ===== Available implementations

The **RestClient** implementations provided by the Holon Platform are are:

- A **JAX-RS** based implementation, using a JAX-RS **Client** to perform invocations, available from the `holon-jaxrs.html`[Holon platform JAX-RS module];
- A **Spring** based implementation, using a Spring **RestTemplate** to perform invocations;

4.2.2. Configure defaults

A `RestClient` instance supports some request **default** configuration attributes:

- A **default target**, i.e. the default base URI which will be used for all the requests performed with the `RestClient`, unless overridden using the specific request configuration `target` method.
- A set of **default headers** to be included in all the requests performed with the `RestClient`.

```
RestClient client = RestClient.create();

client.defaultTarget(new URI("https://rest.api.example")); ①

client.withDefaultHeader(HttpHeaders.ACCEPT_LANGUAGE, "en-CA"); ②
client.withDefaultHeader(HttpHeaders.ACCEPT_CHARSET, "utf-8"); ③
```

- ① Set the default target request base URI, which will be used as target URI for every request configured using `request()`, if not overridden using `target(URI)`.
- ② Add a default request header which will be automatically added to every invocation request message
- ③ Add another default request header

4.2.3. Build a request

To build a client request, the `RequestDefinition` is used, which represents both a *fluent* builder to configure the request message and an `Invocation` to perform the actual invocation and obtain a response.

4.2.4. Configure the request

The request can be configured using the `RequestDefinition` builder methods as described below.

Request URI

The request URI can be composed using:

- A request **target**, i.e. the base URI of the request. If a *default* request target was configured for the `RestClient` instance, it will be overridden by the specific request target.
- One or more request **path*s*, which will be appended to the base request target URI, adding *slash* characters to separate them from one another, if necessary.

```
RestClient client = RestClient.create();

RequestDefinition request = client.request().target(URI.create(
    "https://rest.api.example")); ①
request = request.path("apimethod"); ②
request = request.path("subpath"); ③
```

- ① Set the request *target*, i.e. the base request URI
- ② Set the request *path*, which will be appended to the base request URI
- ③ Append one more *path* to the request URI. The actual URI will be: `https://rest.api.example/apimethod/subpath`

URI *template* variable substitution values

The `RestClient` supports URI *template* variables substitution through the `resolve(...)` method.

IMPORTANT: URI templates variables substitution is only supported for the request URI components specified as `path(...)` elements, not for the `target(...)` base URI part.

```
client.request().target("https://rest.api.example").path("/data/{name}/{id}").resolve(
    "name", "test")
    .resolve("id", 123); ①
```

```
Map<String, Object> templates = new HashMap<>(1);
templates.put("id", "testValue");
request = client.request().target("https://rest.api.example").path("/test/{id}")
    .resolve(templates); ②
```

- ① Subsitute two template variables values
- ② Subsitute template variables values using a name-value map

URI *query* parameters

The `RestClient` supports URI *query parameters* specification, single or multi value, through the `queryParameter(...)` methods.

```
client.request().queryParameter("parameter", "value").queryParameter(
    "multiValueParameter", 1, 2, 3);
```

Request headers

HTTP **headers** can be added to the request using the generic `header(String name, String... values)` method (supporting single or multiple header values) or a set of frequently used headers convenience setter methods, such as `accept`, `acceptLanguage` (supporting Java Locale as arguments) and `cacheControl`.



The `HttpHeaders` interface can be used to refer to HTTP **header names** as constants.



The `MediaType` enumeration can be used for the `Accept` header value using the `accept` header setter method.



The `CacheControl` interface provides a fluent builder to build a `Cache-Control` header value and setting it for the request using the `cacheControl` header setter method.

```
client.request().header("Accept", "text/plain"); // Set an header providing name and
value
client.request().header(HttpHeaders.ACCEPT, "text/plain"); // Set an header providing
name using HttpHeaders and
// value
client.request().accept("text/plain", "text/xml"); // Set an Accept header with two
values
client.request().accept(MediaType.APPLICATION_JSON); // Set an Accept header using
convenience method and
// MediaType enumeration
client.request().acceptEncoding("gzip"); // Set an Accept-Encoding header
client.request().acceptCharset("utf-8"); // Set an Accept-Charset header
client.request().acceptCharset(Charset.forName("utf-8")); // Set an Accept-Charset
header
client.request().acceptLanguage("en-CA"); // Set an Accept-Language header
client.request().acceptLanguage(Locale.US, Locale.GERMANY); // Set an Accept-Language
header using Locales
client.request().cacheControl(CacheControl.builder().noCache(true).noStore(true).build
()); // Set a
// Cache-Control
// header
```

Authorization headers

The `RestClient` provides two convenience request builder methods to set a request `Authorization` header using:

- The `Basic` authorization scheme, providing a *username* and a *password*, using the `authorizationBasic` method.
- The `Bearer` authorization scheme, providing a *token*, , using the `authorizationBearer` method.

```
client.request().authorizationBasic("username", "password"); // set an Authorization
header using a Basic scheme
client.request().authorizationBearer("An389fz56xsr7"); // set an Authorization header
using a Bearer scheme
```

4.2.5. Invoke the request and obtain a response

The `RequestDefinition` interface extends `Invocation`, which can be used to perform the actual invocation and obtain a response.

The `Invocation` interface provides a generic invocation method:

```
<T, R> ResponseEntity<T> invoke(HttpMethod method, RequestEntity<R> requestEntity,
ResponseEntity<T> responseType)
```

This method acts as follows:

- Accept the `HttpMethod` to use to perform the request
- Accept a `RequestEntity` to provide an optional request message payload
- Accept a `ResponseEntity` to declare which type of response *payload* is expected, if any
- It returns a `ResponseEntity` object, which represents the response message entity, including the HTTP status code and an optional response payload, which can be unmarshalled as a Java object



For non textual request or response payload types, any marshalling/unmarshalling strategy and implementation must be provided by the concrete `RestClient`. See the specific `RestClient` implementation documentation for additional informations.

4.2.6. Request invocation methods

In most cases, it is easier and faster to use other methods made available by the interface. Each method is relative to a specific request *method* and it is named accordingly. More than one method version is provided for each request method, providing the most suitable parameters and response types for the most common situations.

For each HTTP request method (apart from the `HEAD` method), the `RestClient` interface makes available a set of invocation methods organized as follows:

- A set of methods to optionally provide a **request entity** and to obtain a `ResponseEntity` (which provides the HTTP status code, the response header and the response payload). If the response is expected to contain an *entity* payload which has to be deserialized into a Java object, the **response type** can be specified, either as a simple or parametrized Java class.

Examples using the `POST` method

```
ResponseEntity<Void> response1 = RestClient.forTarget(
    "https://rest.api.example/testpost").request()
    .post(RequestEntity.json(new TestData()));

ResponseEntity<TestData> response2 = RestClient.forTarget(
    "https://rest.api.example/testpost").request()
    .post(RequestEntity.json(new TestData()), TestData.class);

ResponseEntity<List<TestData>> response3 = RestClient.forTarget(
    "https://rest.api.example/testpost").request()
    .post(RequestEntity.json(new TestData()), ResponseType.of(TestData.class, List
    .class));
```

- A set of method to directly obtain the deserialized response *entity*, named with the `ForEntity` suffix. This methods expects a *successful* response (i.e. a response with a `2xx` HTTP status code),

otherwise throwing a `UnsuccessfulResponseException`, which can be inspected to obtain the response status code and the response itself. This kind of methods returns an `Optional`, which will be empty for empty responses.

Examples using the `GET` method

```
try {

    Optional<TestData> data = RestClient.forTarget("https://rest.api.example/testget")
    .request()
    .getForEntity(TestData.class);

    final ResponseType<List<TestData>> responseType = ResponseType.of(TestData.class,
    List.class);
    List<TestData> dataList = RestClient.forTarget("
    https://rest.api.example/testgetList").request()
    .getForEntity(responseType).orElse(Collections.emptyList());

} catch (UnsuccessfulResponseException e) {
    // got a response with a status code different from 2xx
    int httpStatusCode = e.getStatusCode();
    e.getStatus().ifPresent(status -> System.err.println(status.getDescription()));
    ResponseEntity<?> theResponse = e.getResponse();
}
```

A set of convenience methods are provided for frequent needs and situations, for example:

- A `getForStream` method to obtain a response `InputStream`.
- A `getAsList` method, specifying the list elements type, to obtain a response entity content as a `List` of deserialized Java objects.
- A `postForLocation` to *post* a request entity and obtain the `Location` response header value as a `URI`.

4.2.7. Request entity

The `RequestEntity` interface can be used to provide a *request entity* to the `RestClient` invocation methods, i.e. a request message *payload* as a Java object and a *media type* to set as request `Content-Type` header.



Depending on the `RestClient` implementation used, you must ensure the request media type is supported and suitable message body converters are available to deal with the Java object type and the media type of the request entity.

The `RequestEntity` interface provides a set of convenience static methods to build a request entity using the most common media types, such as `text/plain`, `application/json`, `application/xml` and `application/x-www-form-urlencoded` (also providing a fluent form data builder method).

```

RequestEntity<String> request1 = RequestEntity.text("test"); ①

RequestEntity<TestData> request2 = RequestEntity.json(new TestData()); ②

RequestEntity request3 = RequestEntity
    .form(RequestEntity.formBuilder().set("value1", "one").set("value2", "a", "b")
    .build()); ③

```

- ① Build a `text/plain` type request entity
- ② Build a `application/json` type request entity
- ③ Build a `application/x-www-form-urlencoded` type request entity, using the `formBuilder` method to build form data map

4.2.8. Response type

The `ResponseType` interface can be used to provide the expected response entity type to the `RestClient` invocation methods.

In addition to the simple Java class type, a *parametrized* type can be declared to use Java generic types. For example, to declare a `List<TestData>` response type the following code can be used:

```

ResponseType<List<TestData>> responseType = ResponseType.of(TestData.class, List.
class);

```

4.2.9. Response entity

The `ResponseEntity` interface is used by `RestClient` to represent the *response entity* obtained as invocation result.

The `ResponseEntity` interface provides the **HTTP status code**, the **response headers** (with a set of convenience methods to inspect common HTTP headers) and a method to obtain the **response entity** (the message payload) as a Java object of the type which was expected as invocation result.

The `RequestEntity.EMPTY` constant can be used to provide an empty request entity.



Depending on the `RestClient` implementation used, you must ensure the response media type is supported and suitable message body converters are available to deal with the Java object type and the media type of the response entity.

The `ResponseEntity` interface, in addition to the `getPayload()` method to obtain the Java entity object of the expected response entity type, provides methods (`as(..)`) to unmarshal the response body in an arbitrary Java type, if supported by the underlying `RestClient` implementation.

```

ResponseEntity<TestData> response = RestClient.forTarget(
    "https://rest.api.example/testget").request()
    .accept(MediaType.APPLICATION_JSON).get(TestData.class); ①

HttpStatus status = response.getStatus(); ②

Optional<TestData> entity = response.getPayload(); ③

Optional<String> asString = response.as(String.class); ④

String header = response.getHeaderValue(HttpHeaders.LAST_MODIFIED).orElse(null); ⑤

long contentLength = response.getContentLength().orElse(-1L); ⑥

```

- ① Perform a **GET** request, setting the **Accept** header as **application/json** and declaring the **TestData** class as expected response entity Java type
- ② Get the response status
- ③ Get the response entity payload as a Java object of the expected **TestData** type
- ④ Get the response entity payload as a **String**
- ⑤ Get the response **Last-Modified** header value
- ⑥ Get the response **Content-Length** header value as a **long**

4.2.10. Property and PropertyBox support

The REST client **Invocation** interface provides methods to handle request and response messages which involves **PropertyBox** objects as payload.

Typically, a **PropertyBox** instance is marshalled/unmarshalled using the **JSON** format. The **PropertyBox** JSON serialization and deserialization support is provided by the `holon-json.html` [Holon platform JSON module], both for JAX-RS (*Jersey* and *RestEasy*) and Spring *RestTemplate*. The **PropertyBox** JSON support is automatically setted up when the Holon platform JSON module is available in classpath.

The *property set* which is involved in the request-response cycle can be configured in request definition using the `propertySet(...)` methods.

```
static final PathProperty<Integer> CODE = create("code", int.class);
static final PathProperty<String> VALUE = create("value", String.class);

static final PropertySet<?> PROPERTIES = PropertySet.of(CODE, VALUE);

public void propertiesInvocation() {
    RestClient client = RestClient.create();

    PropertyBox box = client.request().target("https://rest.api.example").path(
        "/apimethod").propertySet(PROPERTIES)
        .getForEntity(PropertyBox.class).orElse(null); ①

    List<PropertyBox> boxes = client.request().target("https://rest.api.example").path(
        "/apimethod")
        .propertySet(PROPERTIES).getAsList(PropertyBox.class); ②
}
```

① GET request for a **PropertyBox** response using **PROPERTIES** property set

② GET request for a list of **PropertyBox** response type using **PROPERTIES** property set

5. Authentication and Authorization

The **holon-auth** artifact provides a complete and highly configurable **Authentication** and **Authorization** architecture, integrated with all platform modules. **HTTP** messages authentication is supported, and several APIs are provided to manage *accounts* and their credentials, perform authentication in a **Realm** and check for **permissions**.

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-auth</artifactId>
<version>5.0.2</version>
```

5.1. Realm

The **Realm** interface represents a security abstraction providing operations for the *principals authentication* and **authorization** (permission check against *principal* grants stored in the realm).

A Realm may be identified by **name**, which should be unique within the same application.

The **Realm** interface is a composition of more specific interfaces, each dealing with a security concern and/or providing hooks to interact with the authentication and authorization process:

- **Authenticator**: It performs the *principal* authentication, relying on an **AuthenticationToken** to

obtain the necessary information to validate the authentication request, it checks the provided credentials and applies a consistent strategy to perform actual authentication

- **MessageAuthenticator**: It performs the *principal* authentication using a generic **Message** (for example, an HTTP request message) from which the authentication informations can be obtained
- **Authorizer**: It performs *permission* check against available *principal* grants
- **AuthenticationNotifier**: It allows **AuthenticationListener** registration to be notified when a successful authentication happens



The **Realm** interface provides a convenience **getCurrent()** static method to obtain the current **Realm** instance made available as ``Context`` resource, using default `ClassLoader`. See **Context** section for further information about context resources.

5.1.1. Authenticator

The **Realm** itself does not implement any authentication model or strategy, but delegates the specific authentication strategy to one or more concrete **Authenticator** registered in the Realm, relying on the **AuthenticationToken** type in order to discern which concrete **Authenticator** has to be used to handle the authentication process.

For this reason, at **Realm** configuration time or during the application lifecycle, the available Authenticators must be registered in Realm to provide the authentication capabilities, using the **addAuthenticator(Authenticator<T> authenticator)** method.

Each **Authenticator** declares the **AuthenticationToken** type it is bound to through the **getTokenType()** method.

To check if an **AuthenticationToken** type is supported by a **Realm** (i.e. an **Authenticator** bound to the token type is registered in Realm), the **supportsToken(Class<? extends AuthenticationToken> authenticationTokenType)** method can be used.

The **Realm** authentication process is structured as follows:

1. The caller invokes the Realm **authenticate(AuthenticationToken authenticationToken)** method, providing a concrete authentication token;
2. The Realm checks if a suitable **Authenticator**, which can handle given **AuthenticationToken** type, is registered. If not, an **UnsupportedTokenException** is thrown;
3. The **authenticate(AuthenticationToken authenticationToken)** method is called on the specific **Authenticator**, performing the concrete authentication operation.

If an authentication operation is not successful, an **AuthenticationException** type is thrown. The concrete type of the exception gives more detailed informations on what went wrong.

Table 3. Available AuthenticationExceptions

| Class | Meaning |
|------------------------------------|---|
| InvalidCredentialsException | Provided credentials are not valid or does not match the stored credentials |

| Class | Meaning |
|--|---|
| <code>ExpiredCredentialsException</code> | Provided credentials are expired |
| <code>UnexpectedCredentialsException</code> | An unexpected internal error occurs during credentials match |
| <code>DisabledAccountException</code> | Account is disabled |
| <code>LockedAccountException</code> | Account is locked |
| <code>UnknownAccountException</code> | Unknown account |
| <code>InvalidCredentialsException</code> | Provided credentials are not valid or does not match the stored credentials |
| <code>UnsupportedTokenException</code> | Unsupported authentication token type |
| <code>UnsupportedMessageException</code> | Unsupported authentication message |
| <code>UnexpectedAuthenticationException</code> | Generic authentication process failure |

AuthenticationToken

The `AuthenticationToken` interface represents an authentication request, and provides the following methods:

- `getPrincipal()`: the **principal** this authentication token refers to, i.e. the account identity submitted during the authentication process. The return type is a generic `Object`, since each authentication model could provide the *principal* information in a different way;
- `getCredentials()`: the **credentials** submitted during the authentication process that verifies the submitted *principal* account identity;

Each `AuthenticationToken` type provides specific values as **principal** and **credentials** information, which the `Authenticator` can interpret and handle to perform the actual authentication.

Builtin AuthenticationTokens

1. Account credentials token: The *account credentials* token represents generic account authentication information, where an *account* is identified by a String type **id** (similar to a *username*) and a String type **secret** (similar to a *password*).

This token returns the account **id** from the `getPrincipal()` method, and the account **secret** from the `getCredentials()` method.

An account credentials token can be created by using the static `accountCredentials(...)` method of the `AuthenticationToken` interface:

```
AuthenticationToken token = AuthenticationToken.accountCredentials("username",
"password");
```

2. Bearer token: The *bearer* token represents a String type information which identifies (or it is bound to) a *principal* and can be used to perform the authentication or grant the access to a resource, checking the token validity. This kind of token is used, for example, in *OAuth* or *JWT*

authentication and authorization models.

This token returns *null* from the `getPrincipal()` method, and the **bearer token** from the `getCredentials()` method.

A bearer token can be created using the static `bearer(...)` method of the `AuthenticationToken` interface:

```
AuthenticationToken token = AuthenticationToken.bearer("Agr564FYda78dsff8Trf7");
```

Custom authentication tokens

You can use any custom `AuthenticationToken` to provide authentication request informations. To handle a custom `AuthenticationToken` in a `Realm`, the corresponding `Authenticator` implementation must be provided and registered.

```

class MyAuthenticationToken implements AuthenticationToken { ❶

    private final String principalName;

    public MyAuthenticationToken(String principalName) {
        super();
        this.principalName = principalName;
    }

    @Override
    public Object getPrincipal() {
        return principalName;
    }

    @Override
    public Object getCredentials() {
        return null;
    }
}

class MyAuthenticator implements Authenticator<MyAuthenticationToken> { ❷

    @Override
    public Class<? extends MyAuthenticationToken> getTokenType() {
        return MyAuthenticationToken.class; ❸
    }

    @Override
    public Authentication authenticate(MyAuthenticationToken authenticationToken) throws
AuthenticationException {
        if (!"test".equals(authenticationToken.getPrincipal())) { ❹
            throw new UnknownAccountException();
        }
        return Authentication.builder(authenticationToken.principalName).build();
    }
}

public void authenticate() {
    Realm realm = Realm.builder().authenticator(new MyAuthenticator()).build(); ❺

    try {
        Authentication authc = realm.authenticate(new MyAuthenticationToken("test")); ❻
    } catch (AuthenticationException e) {
        // handle failed authentication
    }
}

```

❶ Custom `AuthenticationToken` implementation

- ② Custom `Authenticator` bound to the `MyAuthenticationToken` token type
- ③ The token type which the authenticator declares to support
- ④ Actual authentication: if something goes wrong, an exception which extends `AuthenticationException` must be thrown. Otherwise, return the `Authentication` object which represents the authenticated principal
- ⑤ Build a `Realm` and register the custom authenticator
- ⑥ Perform the authentication using a `MyAuthenticationToken` instance

Authentication

The result of the authentication operation performed using the `authenticate(AuthenticationToken authenticationToken)` method of an `Authenticator` is represented by the `Authentication` interface.

An `Authentication` object represents the authenticated *principal*, and extends the default `java.security.Principal` interface, inheriting the `getName()` method to obtain the name which identifies the *principal*.

In addition, the `Authentication` interface holds and provides the following informations:

- The, optional, set of `Permission` granted to the authenticated principal
- A `isRoot()` flag, to mark the authenticated *principal* as *root* principal, i.e. for which the permission checking is always skipped, assuming that any permission is granted to this *principal*
- The optional *scheme* information the *principal* was authenticated with
- It extends `ParameterSet`, representing a set of custom name-value parameters which can be used to provide additional, custom information related to the authenticated *principal*

An `Authentication` can be extended to provide more application-specific informations about the authenticated *principal*, if the parameter set support is not enough or too much generic.

Authentication build example

```
Authentication authc = Authentication.builder("userId").permission("VIEW").permission("MANAGE")
    .parameter("name", "John").parameter("surname", "Doe").build();
```

5.1.2. MessageAuthenticator

The `MessageAuthenticator` interface represents an intermediate *authenticator*, specialized for `Message` based authentication.

The authentication request is provided using a `Message`, for example a HTTP request message. Such message is translated into an `AuthenticationToken` using an `AuthenticationTokenResolver` registered in the `MessageAuthenticator`. From now on, the authentication process proceed as usual, using the obtained `AuthenticationToken` and a suitable `Authenticator` to accomplish the actual authentication operation in `Realm`.

Unlike the standard `Authenticator`, the `MessageAuthenticator` provides a specialized method which accepts a `Message` as authentication request representation:

```
authenticate(Message<?, ?> message, String... schemes)
```

AuthenticationTokenResolver

The `AuthenticationTokenResolver` is responsible for the authentication `Message` handling, and must provide a standard `AuthenticationToken` which represents the authentication message to be used to perform actual authentication in `Realm`.

An `AuthenticationTokenResolver` can optionally declare the authentication *scheme* it is bound to, in order to discern different messages authentication information within a set of messages of the same type, and select the most suitable `AuthenticationTokenResolver` relying on the authentication scheme name.

For example, within the set of the HTTP request messages type, different authentication schemes, such as *Basic* or *Bearer*, can be bound to a different `AuthenticationTokenResolver`, producing different `AuthenticationToken`s.

For `HttpRequest` message types, two builtin `AuthenticationTokenResolver` are provided:

- For the **Basic** HTTP authentication scheme, using the `Authorization` header, producing a `[AccountCredentialsToken]`:

```
AuthenticationTokenResolver<HttpRequest> basicResolver = AuthenticationToken  
.httpBasicResolver();
```

- For the **Bearer** HTTP authentication scheme, using the `Authorization` header, producing a `[BearerAuthenticationToken]`:

```
AuthenticationTokenResolver<HttpRequest> bearerResolver = AuthenticationToken  
.httpBearerResolver();
```

An `AuthenticationTokenResolver` must be registered in `Realm` using the `addAuthenticationTokenResolver(...)` method.

```
Realm realm = Realm.builder().resolver(AuthenticationToken.httpBasicResolver())
    .authenticator(new MyAuthenticator()).build(); ①

HttpRequest request = null; // obtain the HttpRequest message in real world code
try {
    Authentication authc = realm.authenticate(request); ②
} catch (AuthenticationException e) {
    // handle failed authentication
}
```

① Register a HTTP **Basic** message resolver in Realm

② Perform authentication using a **HttpRequest** message

5.1.3. Authorizer

The **Authorizer** interface is responsible for authorization control, checking if one or more *permissions* are granted to a *principal*.

An **Authorizer** uses the **Authentication** representation to obtain the **permissions** granted to an authenticated *principal*, and provides several **isPermitted...** methods to perform permission control.

Permission

A **permission** is the representation of the access to a resource and can be expressed in two ways:

- As a **String**: this can be compared to a **Role** name.
- Using the **Permission** interface. This interface can be extended and implemented in different ways to represent more complex permissions than a simple role name. The **Authorizer** relies on the **equals()** and **hashCode()** implementation to perform **Permission** comparison.



A **Permission** object which uses a simple String name can be created using the **Permission.create(String permission)** static method.

Authorization checking

Each **Authorizer** can be bound to a specific **Permission** type, to create specialized authorizers which handle a specific permission type.

To enable **authorization checking** for a **Realm**, the available authorizers must be registered in realm using the **addAuthorizer(...)** method.

The platform provides a **default Authorizer** which can be obtained using the **Authorizer.create()** static method. The default authorizer is bound to a generic **Permission** type and permission checking is performed by comparing the **Authentication** granted permissions with the permissions to be checked, using the **Permission.equals(...)** method to compare a single permission to another.

The default `Authorizer` supports the `Authentication.isRoot()` state, always granting permissions to **root** authentications.

Authorization example

```
final Permission p1 = Permission.create("role1");
final Permission p2 = Permission.create("role2");

// build an Authentication and grant the two permissions to it
Authentication authc = Authentication.builder("test").permission(p1).permission(p2)
    .build();

// Realm with default authorizer
Realm realm = Realm.builder().withDefaultAuthorizer().build();

// permission checking
boolean permitted = realm.isPermitted(authc, p1);
permitted = realm.isPermitted(authc, p1, p2);
permitted = realm.isPermitted(authc, "p1");
permitted = realm.isPermittedAny(authc, p1, p2);
permitted = realm.isPermittedAny(authc, "p1", "p2");
```

5.1.4. AuthenticationListener support

The `Realm` interface allows the registration of listeners to be notified when a successful authentication happens, using the `addAuthenticationListener(AuthenticationListener authenticationListener)` method.

The `AuthenticationListener` method `onAuthentication(Authentication authentication)` is invoked when a principal is successfully authenticated, providing the `Authentication` representation object.

5.2. Credentials

The platform provides a support for authentication *credentials* management, relying on the following structures:

- The `Credentials` interface to represent credentials data, i.e. the **secret** and the encoding informations related to it
- The `CredentialsContainer` interface identifies an object which provides credentials data (for example, the stored account informations related to a principal and the credentials provided by a *principal* during and authentication process)
- The `CredentialsMatcher` deals with credentials checking, and it is able to determine if two credentials data structures match (for example, if provided credentials match the account stored credentials)

5.2.1. Create and encode Credentials

The `Credentials` interface provides a *builder* to create and encode a `Credentials` representation.

The `Credentials` builder provides method to encode a **secret** by using a *hashing* algorithm (specifying also a *salt* and the hash iterations to be performed), specifying an optional *expiry date* and encode the secret using, for example, `Base64`.

Credentials build examples

```
Credentials credentials = Credentials.builder().secret("test").build(); ①

credentials = Credentials.builder().secret("test").hashAlgorithm(Credentials.Encoder
    .HASH_MD5).build(); ②

credentials = Credentials.builder().secret("test").hashAlgorithm(Credentials.Encoder
    .HASH_MD5).hashIterations(7)
    .salt(new byte[] { 1, 2, 3 }).build(); ③

credentials = Credentials.builder().secret("test").hashAlgorithm(Credentials.Encoder
    .HASH_MD5).base64Encoded()
    .build(); ④

credentials = Credentials.builder().secret("test").expireDate(new Date()).build(); ⑤
```

- ① Simple credentials using `test` as secret and no encodings
- ② Credentials using `test` as secret and `MD5` as hashing algorithm
- ③ Credentials using `test` as secret and `MD5` as hashing algorithm, with a *salt* and 7 hashing iterations
- ④ Credentials using `test` as secret and `MD5` as hashing algorithm, encoded using `Base64`
- ⑤ Simple credentials using `test` as secret and no encodings, specifying an *expiry date*

5.2.2. Credentials encoder

To encode credentials data, for example for storing purposes, the `Credentials` interface provides an `Encoder` interface, which can be obtained using the `encoder()` static method.

Credentials encoding examples

```
String encoded = Credentials.encoder().secret("test").buildAndEncodeBase64(); ①

byte[] bytes = Credentials.encoder().secret("test").hashSHA256().build(); ②

encoded = Credentials.encoder().secret("test").hashSHA256().salt(new byte[] { 1, 2, 3
    }).buildAndEncodeBase64(); ③

encoded = Credentials.encoder().secret("test").hashSHA512().charset("UTF-8")
    .buildAndEncodeBase64(); ④
```

- ① Credentials using `test` as secret and `Base64` encoded
- ② Credentials using `test` as secret and `SHA-256` as hashing algorithm, returned as bytes

- ③ Credentials using `test` as secret and `SHA-256` as hashing algorithm, with a `salt` and `Base64` encoded
- ④ Credentials using `test` as secret and `SHA-512` as hashing algorithm, encoded using `Base64` with the UTF-8 charset

5.2.3. Credentials matching

Credentials matching can be performed using a `CredentialsMatcher`.

The platform provides a **default** `CredentialsMatcher` implementation which can be obtained using the `defaultMatcher()` method on the `CredentialsContainer` interface.

The default credentials matcher tries to employ best-practices and common behaviours to perform credentials validation and matching:

- Try to convert generic Object credentials data into a **byte array**:
 - It supports `char[]`, `String`, `File` and `InputStream` for direct bytes conversion
 - It supports `Credentials` type, using the `getSecret()` method to obtain the bytes
- If the provided credentials data are of `Credentials` type and an *expiry date* is provided, it checks the credentials are not expired
- It check if the array of bytes obtained from the two credentials data structures **match**, hashing and/or decoding the credentials data if these informations are available (i.e. the stored credentials are of `Credentials` type)

5.3. Account

The platform provides an abstraction of an *Account* structure, which represents information about a *principal*.

The `Account` interface is used to represent a generic *account*, providing the following information:

- The account **id** (as a `String`)
- The account **credentials** (as a generic Object)
- Whether the account is a **root** account, i.e. has any permission
- Optional map of generic account **details**, identified by a `String` id
- Optional set of **permissions** granted to the account
- Whether the account is **enabled**
- Whether the account is **locked**
- Whether the account is **expired**

A *builder* is available to create `Account` instances:

```
Account.builder("accountId").enabled(true).locked(false).expired(false)
    .credentials(Credentials.builder().secret("pwd").hashAlgorithm(Credentials.
Encoder.HASH_SHA_256)
    .base64Encoded().build()) // credentials
.root(false) // not root (default)
.permission("role1").permission("role2") // permissions
.detail("name", "TheName") // principal name
.detail("surname", "TheSurname") // principal surname
.build();
```

5.3.1. AccountProvider

The `AccountProvider` interface can be used to provide `Account` instances using the **account id**, for example from a data store, by implementing the `loadAccountById(String id)` method.

The loading method returns an `Optional`, if empty means that an account with given id is not available.

5.3.2. Authenticator

A default `Authenticator` is provided to perform the account authentication by using an `AccountProvider` to load account data. The account authenticator is obtained by using the `authenticator(...)` static method of the `Account` interface, choosing to use a custom credentials matcher or the default one.

The supported `AuthenticationToken` is the *account credentials* authentication token, obtained using the static `accountCredentials(...)` method of the `AuthenticationToken` interface.

```
AccountProvider provider = id -> Optional.of(Account.builder(id).enabled(true)
    .credentials(Credentials.builder().secret("pwd").base64Encoded().build())
    .permission("role1").build()); ①

Realm realm = Realm.builder().authenticator(Account.authenticator(provider))
    .withDefaultAuthorizer().build(); ②

try {
    Authentication authc = realm.authenticate(AuthenticationToken.accountCredentials(
"test", "pwd")); ③
} catch (AuthenticationException e) {
    // handle authentication failures
}
```

- ① Create an `AccountProvider` to provide the `Account` instances according to the *account id*
- ② Create a `Realm` with default authorizer and register an account `Authenticator` which uses the previously defined `AccountProvider`
- ③ Perform authentication using an *account credentials* authentication token

5.4. AuthContext

The `AuthContext` interface can be used to represent the current authentication/authorization context, i.e. acts as a holder of the current `Authentication`, if available, and provides methods to perform the actual authentication operations, relying on a specific `Realm` instance.

For its nature, an `AuthContext` is typically bound to an application *session* or to the current request context for a backend services layer.

It provides methods to:

- **Check** if a *principal* is currently authenticated in context and if so **obtain** the corresponding `Authentication`
- Perform **authentication** on the reference `Realm`, either using an `AuthenticationToken` or a `Message`
- **De-authenticate** the context, removing current `Authentication`
- Perform **authorization** control on current `Authentication`, using the reference `Realm` authorizers

See the `Realm` section for detailed information about authenticators and authorizers.



When an `AuthContext` is **de-authenticated**, any `Realm` registered `AuthenticationListener` is notified, invoking `onAuthentication(Authentication authentication)` listeners method with a *null authentication* parameter value.

Authorization example

```
AccountProvider provider = id -> Optional.of(Account.builder(id).enabled(true)
    .credentials(Credentials.builder().secret("pwd").base64Encoded().build())
    .permission("role1").build()); ①

Realm realm = Realm.builder().authenticator(Account.authenticator(provider))
    .withDefaultAuthorizer().build(); ②

AuthContext context = AuthContext.create(realm); ③

context.addAuthenticationListener(
    a -> System.out.println((a != null) ? "Authenticated: " + a.getName() :
    "Unauthenticated")); ④

try {

    context.authenticate(AuthenticationToken.accountCredentials("test", "pwd")); ⑤

    Authentication authc = context.getAuthentication()
        .orElseThrow(() -> new IllegalStateException("Context not authenticated")); ⑥

    boolean permitted = context.isPermitted("role1"); ⑦

    context.unauthenticate(); ⑧

} catch (AuthenticationException e) {
    // handle authentication failures
}
```

- ① Create an **AccountProvider** to provide the **Account** instances according to the *account id*
- ② Create a **Realm** with default authorizer and register an account **Authenticator** which uses the previously defined **AccountProvider**
- ③ Create an **AuthContext** backed by the created realm
- ④ Add an **AuthenticationListener** to the context
- ⑤ Perform authentication by using an *account credentials* authentication token
- ⑥ Get the current context **Authentication**
- ⑦ Check if the current context **Authentication** has the permission named *role1*
- ⑧ Unauthenticate the context, i.e. remove the current **Authentication**



The **AuthContext** interface provides a convenience **getCurrent()** static method allowing to obtain the current **AuthContext** instance which is made available as ``Context`` resource, using the default `ClassLoader`. See **Context** section for further information about context resources.

5.5. @Authenticate annotation

The [Authenticate](#) can be used on classes or methods to require authentication for resource access.

The support for this annotation must be documented and it is available for other modules of the Holon platform.

The annotation supports optional **schemes** specification to provide the allowed authentication schemes to be used to perform principal authentication, and an optional **redirectURI** which can be used to redirect user interaction when the authentication succeeds or fails (the semantic and behaviour associated to the redirect URI is specific for every authentication delegate).

5.6. JWT support

5.6.1. Introduction

The [holon-auth-jwt](#) artifact provides the support for the [JSON Web Token](#) standard, integrating it in the Holon platform authentication and authorization architecture.

The [jjwt](#) library is used for JWT tokens parsing and building.

JSON Web Token (**JWT**) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. The transmitted information can be digitally signed, in order to be verified and trusted by the parties.

When used for authentication, thanks to its very compact data representation and encoding, a JWT token can transport and provide not only the informations to perform authentication, but also the information obtained as a result of an authentication operation, such as *principal's* details and permissions.

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>  
<artifactId>holon-auth-jwt</artifactId>  
<version>5.0.2</version>
```

5.6.2. Configuration

To enable and use JWT authentication structures, some configuration parameters must be defined and provided to the objects which perform concrete operations on the JWT tokens.

The JWT configuration properties are represented by the [JwtConfiguration](#) interface, which can be instantiated as follows:

1. Using the *builder* provided by the [JwtConfiguration](#) interface:

```

JwtConfiguration cfg = JwtConfiguration.builder().issuer("MyIssuer") // set the JWT
token issuer
    .expireTime(10000) // token expire time in milliseconds
    .includeDetails(true) // include the Authentication details in JWT token
generation
    .includePermissions(true) // include the Authentication permissions in JWT token
generation
    .signatureAlgorithm("HS256") // use HS256 as signature algorithm
    .sharedKey(new byte[] { 1, 2, 3 }) // shared key to use with the symmetric signing
algorithm
    .build();

```

2. Using the configuration properties collected and represented by the [JwtConfigProperties](#) property set, extending a default [ConfigPropertySet](#) and bound to the property name prefix **holon.jwt**.

The available configuration properties are listed here below:

Table 4. JWT configuration properties

| Name | Type | Meaning |
|---------------------------------------|----------------|---|
| <i>holon.jwt. issuer</i> | String | The JWT token issuer |
| <i>holon.jwt. signature-algorithm</i> | String | JWT signature algorithm name |
| <i>holon.jwt. sharedkey-base64</i> | String | JWT sign shared key (base64 encoded) for symmetric signing algorithms |
| <i>holon.jwt. publickey-base64</i> | String | JWT sign public key (base64 encoded) for RSA signing algorithms |
| <i>holon.jwt. publickey-file</i> | String | JWT sign public key (file name) for RSA signing algorithms |
| <i>holon.jwt. privatekey-base64</i> | String | JWT sign private key (base64 encoded) for RSA signing algorithms |
| <i>holon.jwt. privatekey-file</i> | String | JWT sign private key (file name) for RSA signing algorithms |
| <i>holon.jwt. expire-ms</i> | Integer number | JWT token expire time in milliseconds |
| <i>holon.jwt. expire-seconds</i> | Integer number | JWT token expire time in seconds |
| <i>holon.jwt. expire-minutes</i> | Integer number | JWT token expire time in minutes |
| <i>holon.jwt. expire-hours</i> | Integer number | JWT token expire time in hours |
| <i>holon.jwt. expire-days</i> | Integer number | JWT token expire time in days |

| Name | Type | Meaning |
|---------------------------------------|----------------------|---|
| <i>holon.jwt. include-details</i> | Booelan (true/false) | Whether to include Authentication details in JWT token as claims |
| <i>holon.jwt. include-permissions</i> | Booelan (true/false) | Whether to include Authentication permissions in JWT token as claims |

JwtConfiguration example using configuration properties

```
JwtConfiguration cfg = JwtConfiguration
    .build(JwtConfigProperties.builder().withPropertySource("jwt.properties").build()
    );
```

5.6.3. Building JWT tokens for an **Authentication**

To build JWT token the **JwtTokenBuilder** class is available, providing static methods to create a **JWT token** from an **Authentication** object, optionally using a **JwtConfiguration** class to provide token configuration attributes.

If enabled, the **permissions** and **parameters** of the **Authentication** will be written in the JWT token with the following behaviour:

- The **Authentication permissions** will be written only for the permissions which provides a **String** representation through the **Permission.getPermission()** method. The serializable permissions are written as String array and associated to the *claim* name which corresponds to the constant **AuthenticationClaims.CLAIM_NAME_PERMISSIONS** value;
- Each **Authentication parameter** will be written by using the parameter name as *claim* name and the parameter value as *claim* value.

JWT token building example using JwtConfiguration

```
JwtConfiguration configuration = JwtConfiguration
    .build(JwtConfigProperties.builder().withPropertySource("jwt.properties").build()
    ); ①

Authentication authc = Authentication.builder("test").permission("role1").parameter(
    "name", "TestName").build(); ②

String jwtToken = JwtTokenBuilder.buildJwtToken(configuration, authc, UUID.randomUUID
    ().toString()); ③
```

① Build a **JwtConfiguration** instance using the **jwt.properties** file

② Build an example **Authentication**

③ Build a **JWT token** by using given configuration and authentication, using a random id as token id

5.6.4. Performing authentication using JWT tokens

The `JwtAuthenticator` interface is provided as `Authenticator` to handle JWT token based authentication, using a `JwtConfiguration` to obtain JWT token configuration attributes.

The authentication request information are provided by using a `BearerAuthenticationToken`, where the **Bearer** value represents the complete JWT token serialization.

From an `HTTPMessage` point of view, the authentication token is obtained from an `Authorization` header using the `Bearer` scheme.

To obtain a `JwtAuthenticator`, the provided *builder* can be used. The authenticator, just like any other `Authenticator`, must be registered in a `Realm` to enable JWT token based authentication.

The `JwtAuthenticator` supports:

- An optional set of allowed **JWT issuers**: If one or more allowed issuer is set, JWT Issuer *claim* (`iss`) will be required and checked during token authentication: if token issuer doesn't match one of the given issuers, authentication will fail;
- Optional **required claims**: If one or more required *claim* is configured, the specified *claim* must exist in JWT token, otherwise authentication will fail.

```
JwtConfiguration configuration = JwtConfiguration.builder().issuer("MyIssuer") // set
the JWT token issuer
    .expireTime(10000) // token expire time in milliseconds
    .includeDetails(true) // include the Authentication details in JWT token
generation
    .includePermissions(true) // include the Authentication permissions in JWT token
generation
    .signatureAlgorithm("HS256") // use HS256 as signature algorithm
    .sharedKey(new byte[] { 1, 2, 3 }) // shared key to use with the symmetric signing
algorithm
    .build();

// Build the Jwt authenticator using the JwtConfiguration
JwtAuthenticator jwtAuthenticator = JwtAuthenticator.builder().configuration
(configuration)
    .issuer("allowedIssuer").requiredClaim("myClaim").build();

// Build a Realm and register the authenticator
Realm realm = Realm.builder().authenticator(jwtAuthenticator).withDefaultAuthorizer()
.build();

try {

    // Authentication using a bearer token
    Authentication authc = realm.authenticate(AuthenticationToken.bearer(
"TheJWTtokenHere..."));

    // Authentication using a message
    HttpRequest request = null; // expected an HttpRequest message with an
'Authorization' header with 'Bearer:
    // JWTtokenValue'
    authc = realm.authenticate(request);

} catch (AuthenticationException e) {
    // handle authentication failures
}
```

6. Spring framework integration

The `holon-spring` artifact provides integration with the `Spring` framework and auto-configuration features using `Spring Boot` for the Holon platform core module.

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-spring</artifactId>
<version>5.0.2</version>
```

6.1. Spring beans as context resources

The `EnableBeanContext` annotation can be used on Spring `Configuration` classes to configure a `Context` scope using the Spring `ApplicationContext` to provide **context resources** instances as Spring beans.

The scope registration priority order is an intermediate value between the default *thread* scope (highest priority) and *classloader* scope (lowest priority).

If the scope is registered, when a context resource is requested, using for example the `Context.resource(String resourceKey, Class<T> resourceType)` method, the Spring scope checks if a *bean* matches the requested resource key and type using the following strategy:

- If a Spring bean with a **name** equal to the requested resource **key** and with the same required **type** is found, this is returned;
- Otherwise, if the `lookupByType()` attribute value of the `@EnableBeanContext` annotation is `true` and a Spring bean of the required type, ignoring the its name, is present and *only one candidate* is available, this instance is returned.

The Spring **scopes** are respected, in the sense that when a resource is requested and a matching *bean* is found, the bean instance lookup is performed using standard Spring `ApplicationContext` methods, involving any registered and active Spring scope.

```
class TestResource {
}

@Configuration
@EnableBeanContext
class SpringConfig {

    @Bean(name = "testResource")
    public TestResource testResource() {
        return new TestResource();
    }

}

public void getContextResource() {
    // lookup using resource name which matches the bean name
    Optional<TestResource> resource = Context.get().resource("testResource",
TestResource.class);
    // lookup by type
    resource = Context.get().resource(TestResource.class);
}
```

6.2. EnvironmentConfigPropertyProvider

The [EnvironmentConfigPropertyProvider](#) is provided to build a [ConfigPropertyProvider](#) which uses the Spring [Environment](#) structure as property source.

EnvironmentConfigPropertyProvider example

```
@Autowired
Environment environment;

public void env() {
    // build a ConfigPropertyProvider using Spring Environment as property source
    ConfigPropertyProvider provider = EnvironmentConfigPropertyProvider.create
(environment);

    String value = provider.getProperty("test.property.name", String.class);
}
```

6.3. Spring tenant scope

The Holon platform provides a Spring **tenant scope** which provides different bean instances depending on the current *tenant id*.

This scope relies on the default [TenantResolver](#) interface to obtain the current tenant id.

A bean of `TenantResolver` type must be configured and available in the current `BeanFactory` (i.e. in the current `ApplicationContext`) to use the scope. That bean will be used to obtain the current **tenant id**, using the `getTenantId()` method.

To enable the tenant scope, use the `@EnableTenantScope` annotation on Spring configuration classes.

The scope name is `tenant`, so Spring beans can be registered with this scope using either:

- The default Spring `@Scope("tenant")` annotation
- The convenience `@ScopeTenant` annotation

Spring tenant scope example

```
@Configuration
@EnableTenantScope
class TenantScopeConfig {

    @Bean
    public TenantResolver tenantResolver() {
        return () -> Optional.of("test"); // provide a meaningful current tenant id
        resolution strategy...
    }

    @Bean
    @ScopeTenant
    public TestResource testResource() {
        // a different instance of the bean will be provided for each tenant id
        return new TestResource();
    }
}
```

6.4. Datastore configuration

The Spring integration module provides a number of methods to extend and configure a `Datastore` using Spring beans when the `Datastore` is registered as a bean in the Spring context.

6.4.1. DatastoreResolver

The `DatastoreResolver` annotation can be used to annotate `ExpressionResolver` type beans to automatically register them into a `Datastore`.

The `datastoreBeanName()` annotation attribute can be used to uniquely identify the `Datastore` bean into which register the resolver, if more than one `Datastore` bean is present in Spring context.

6.4.2. DatastoreCommodityFactory

The `DatastoreCommodityFactory` annotation can be used to annotate `DatastoreCommodityFactory` type beans to automatically register them into a `Datastore`.

The `datastoreBeanName()` annotation attribute can be used to uniquely identify the `Datastore` bean into which register the factory, if more than one `Datastore` bean is present in Spring context.



Each concrete `Datastore` implementation could provide a specific `DatastoreCommodityFactory` base type to be used to register commodity factories in order to provide a specific `DatastoreCommodityContext`. See specific `Datastore` documentation for further information.

6.4.3. DatastorePostProcessor

The `DatastorePostProcessor` interface can be used to configure a `Datastore`, if registered as a Spring bean, right after it is initialized in the Spring context.

A Spring bean class implementing this interface is automatically detected and the method `postProcessDatastore(Datastore, String)` is called at `Datastore` bean initialization time. For example, the post processor can be used to register additional `ExpressionResolver` or `DatastoreCommodityFactory`.

The `DatastorePostProcessor` beans must be registered with a `singleton` scope in context.

6.5. RestClient implementation using Spring RestTemplate

The Spring integration module provides a `RestClient` implementation using Spring `RestTemplate`.

The Spring `RestClient` is represented by the `SpringRestClient` interface, which provides a `create(RestTemplate restTemplate)` method to create a `RestClient` instance using provided Spring `RestTemplate`.

When a `RestTemplate` instance is available in platform `Context` (for example if is available as Spring bean and the Spring application context support as a context scope is enabled - see [Spring beans as context resources](#)), a `RestClientFactory` is automatically registered to provide a `SpringRestClient` implementation using the `RestTemplate`.

```
@Autowired
RestTemplate restTemplate;

public void restclient() throws URISyntaxException {

    // Create a SpringRestClient
    RestClient client = SpringRestClient.create(restTemplate);

    // obtain the RestClient using default creation methods
    client = RestClient.create();

    client = RestClient.create(SpringRestClient.class.getName());
}
```

6.6. Spring Boot auto-configuration

The Holon platform provides **Spring Boot auto-configuration** features for the most of the platform modules, including the core module.

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

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-spring-boot</artifactId>
<version>5.0.2</version>
```

Two auto-configuration features are provided:

1. **Spring beans as context resources** is enabled, registering the corresponding **Context** scope. This has the same effect as using the `@EnableBeanContext` annotation on Spring configuration classes, with the `lookupByType()` attribute set as true by default.

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

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

2. If a **TenantResolver** bean is registered in Spring **ApplicationContext**, the **Spring tenant scope** is automatically registered. This has the same effect as using the `@EnableTenantScope` annotation on Spring configuration classes.

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

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

6.6.1. Spring Boot starters

The following Spring Boot *starter* artifact is available to provide a quick project configuration setup using Maven dependency system:

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-starter</artifactId>
<version>5.0.2</version>
```

The starter provides the dependency to the `holon-spring-boot` artifact in addition to the one to the base Spring Boot starter (`spring-boot-starter`).

7. 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 following logger names are available:

- [com.holonplatform.core](#): the root **core** logger
 - [presentation](#): for logs related to values presentation
 - [i18n](#): for logs related to localization and internationalization
 - [beans](#): for logs related to bean inspection and bean properties
 - [property](#): for logs related to the [Property](#) architecture, including [PropertyBox](#), property presenters and renderers
 - [query](#): for logs related to [Query](#) definition and execution
 - [datastore](#): for logs related to [Datastore](#) configuration and operation execution
- [com.holonplatform.jdbc](#): for logs related to **JDBC** support classes, such as DataSource builders
- [com.holonplatform.http](#): for logs related to **HTTP** support classes, such as [RestClient](#)
- [com.holonplatform.spring](#): for logs related to [Spring](#) integration

8. System requirements

8.1. Java

The Holon Platform core module requires [Java 8](#) or higher.