

Holon Platform Core Module - Reference manual

Table of Contents

1. Introduction	6
1.1. Sources and contributions	7
2. Obtaining the artifacts	7
2.1. Using the Platform BOM	8
3. What's new in version 5.2.x	8
3.1. Migrating from version 5.1.x	9
3.1.1. API changes	9
4. What's new in version 5.1.x	9
4.1. Property model	9
4.2. Datastore	10
4.3. JWT authentication	10
4.4. Spring ecosystem integration	10
4.5. Migrating from version 5.0.x	10
4.5.1. Deprecations	10
4.5.2. Property model	11
5. Core API, services and components	11
5.1. Introduction	11
5.2. Context	12
5.2.1. Obtaining Context resources	12
5.2.2. Scopes	13
5.2.3. Default Scopes	13
5.2.4. Context extension: adding Scopes	14
5.3. Configuration and parameters	16
5.3.1. Configuration properties	16
5.3.2. Configuration property provider	17
5.3.3. Configuration property set	17
5.3.4. ParameterSet	17
5.4. Data validation	18
5.4.1. Validator	18
5.4.2. Bultin validators	19
5.4.3. Validatable and ValidatorSupport	20
5.5. StringValuePresenter	21
5.5.1. Default StringValuePresenter	21
5.6. Internationalization	23

5.6.1. Localizable messages	23
5.6.2. @Caption annotation	24
5.6.3. Message providers	24
5.6.4. LocalizationContext	25
5.6.5. Building a LocalizationContext	25
5.6.6. Obtaining a LocalizationContext	26
5.6.7. Localizing a LocalizationContext	26
5.6.8. Listening to localization changes	27
5.6.9. Using the LocalizationContext	27
5.6.10. MissingMessageLocalizationListener	29
5.7. Properties	29
5.7.1. Property naming and identity	30
5.7.2. Configuration	30
5.7.3. Converters	32
5.7.4. Localization	34
5.7.5. Validation	34
5.7.6. Read only attribute	34
5.8. PathProperty	35
5.8.1. PathProperty sub types	36
5.9. VirtualProperty	37
5.10. Collection properties	38
5.10.1. Collection virtual properties	39
5.10.2. Collection property value converters	40
5.11. Organizing and collecting properties using a PropertySet	41
5.11.1. PropertySet configuration	42
5.11.2. Identifier properties	42
5.11.3. Using a PathPropertySetAdapter	43
Inspect a PropertySet using the Property names	44
5.12. Managing property values using a PropertyBox	44
5.12.1. PropertyBox instances identification	47
5.12.2. Custom PropertyBox instances identification	48
5.12.3. Using a PathPropertyBoxAdapter	48
5.13. Property value presentation	49
5.13.1. PropertyValuePresenter registration	49
5.13.2. Default PropertyValuePresenter	50
5.13.3. Using the Property presenters	50
5.14. Property rendering	51
5.14.1. PropertyRenderer registration	51
5.14.2. Using the Property renderers	52
5.15. Java Beans and the Property model	53
5.15.1. Bean properties	53

5.15.2. Bean property set	53
5.15.3. BeanIntrospector	56
5.15.4. BeanPropertyPostProcessor	57
5.15.5. BeanPropertySetPostProcessor	58
5.15.6. Builtin Bean post processors	58
@Ignore	59
@Caption	59
@Sequence	60
@Config	60
@Converter	61
Validators	62
@DataPath	63
5.15.7. BeanIntrospector cache	63
5.16. Datastore	64
5.16.1. Expressions and resolvers	64
5.16.2. Property data types	65
5.16.3. DataTarget	66
5.16.4. Data manipulation operations	66
5.17. Query	70
5.17.1. QueryFilter	70
5.17.2. QuerySort	74
5.17.3. QueryFunction	75
Aggregation functions	75
String related functions	76
Temporal functions	77
5.17.4. QueryAggregation	78
5.17.5. Query definition	79
5.17.6. Query projection and execution	80
Builtin query projections	82
5.17.7. Distinct query projection results	85
5.17.8. Configuration	85
Multiple Datastores configuration	86
5.17.9. Relational Datastores	87
Sub-query	87
Alias and Joins	88
5.17.10. Transactional Datastores	90
5.17.11. Datastore API extensions	92
Extend the Datastore API using ExpressionResolver	92
DataTargetResolver	93
QueryFilterResolver	94
QuerySortResolver	96

Datastore <i>commodities</i> definition and registration	97
5.17.12. Available Datastores	98
5.18. DataMappable	98
5.18.1. Data mapping declaration	99
5.18.2. Data mapping usage	99
5.19. Multi tenancy support	99
5.20. Utilities	100
5.20.1. Initializer	100
5.20.2. SizedStack	100
6. HTTP messages and RESTful Java client	100
6.1. HTTP messages	100
6.1.1. Headers	101
6.1.2. HttpRequest	101
6.1.3. HttpResponse	102
6.1.4. Servlet API integration	103
6.2. RESTful client API	103
6.2.1. Obtain a RestClient instance	104
Available implementations	105
6.2.2. Configure defaults	105
6.2.3. Build and configure a request	105
Request URI	105
URI <i>template</i> variable substitution values	106
URI <i>query</i> parameters	106
Request headers	107
Authorization headers	107
6.2.4. Invoke the request and obtain a response	108
6.2.5. Request entity	109
6.2.6. Response type	109
6.2.7. Response entity	110
6.2.8. Specific request invocation methods	111
6.2.9. RestClient API invocation methods reference	113
6.2.10. Property and PropertyBox support	118
7. Authentication and Authorization	119
7.1. Realm	119
7.1.1. Realm name	120
7.1.2. Realm authentication	120
7.1.3. AuthenticationToken	122
Account credentials authentication token	122
Bearer authentication token	123
7.1.4. Authenticator	123
Builtin authenticators	124

Authenticator example	124
7.1.5. Authentication exceptions	126
7.1.6. Authentication	126
7.1.7. Authentication listeners	127
7.1.8. MessageAuthenticator	128
Use Realm as a MessageAuthenticator	129
Builtin HTTP message resolvers	129
7.1.9. Realm authorization	131
7.1.10. Authorizer	133
Default Authorizer	133
7.1.11. Permission	134
Default Permission	134
Authorization control example	134
7.1.12. Realm as a Context resource	135
7.2. Authentication credentials	136
7.2.1. Credentials creation	136
7.2.2. Credentials encoding	137
7.2.3. Credentials container	137
7.2.4. Credentials matching	137
7.3. Account	138
7.3.1. AccountProvider	139
7.3.2. Account Authenticator	139
AccountCredentialsToken	140
Account authenticator example	140
7.4. AuthContext	141
7.4.1. Authentication	142
7.4.2. Current Authentication	142
7.4.3. Custom Authentication holder	143
7.4.4. Authentication listeners	144
7.4.5. AuthContext as a Context resource	144
7.5. @Authenticate annotation	145
7.6. JWT support	145
7.6.1. Configuration	146
7.6.2. Supported JWT signing algorithms	151
7.6.3. Building a JWT from an Authentication	151
Authentication permissions claim	152
Authentication details claims	153
7.6.4. Building an Authentication from a JWT	153
7.6.5. Performing authentication using JWT	154
JWT to Authentication	156
8. Spring ecosystem integration	156

8.1. Spring beans as Context resources	156
8.1.1. Context resources lookup strategy	157
8.2. EnvironmentConfigPropertyProvider	158
8.3. Spring <i>tenant</i> scope	158
8.3.1. TenantResolver lookup strategy	159
8.3.2. Tenant scope <i>ScopedProxyMode</i>	159
8.3.3. Tenant scoped beans lifecycle	160
8.4. Datastore configuration	160
8.4.1. DatastoreResolver	161
8.4.2. DatastoreCommodityFactory	161
8.4.3. DatastorePostProcessor	161
8.4.4. Automatic Datastore beans configuration using @EnableDatastoreConfiguration	161
8.5. RestClient implementation using Spring RestTemplate	162
8.6. Spring Boot auto-configuration	163
8.6.1. Spring context scope auto-configuration	163
8.6.2. Spring <i>tenant</i> scope auto-configuration	164
8.6.3. JwtConfiguration auto-configuration	164
8.6.4. Spring Boot starters	166
8.7. Spring Security integration	166
8.7.1. AuthContext API integration	167
8.7.2. Use an AuthenticationManager as a Realm Authenticator	168
8.7.3. Create a fully integrated AuthContext API	170
8.7.4. Use a Holon Authenticator as Spring Security AuthenticationProvider	171
8.7.5. Permissions and authorizations	173
8.7.6. Spring Security <i>starter</i>	173
9. Loggers	173
10. System requirements	174
10.1. Java	174

Copyright © 2016-2018

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

1. Introduction

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

1.1. Sources and contributions

The Holon Platform **Core** module source code is available from the GitHub repository <https://github.com/holon-platform/holon-core>.

See the repository **README** file for information about:

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

2. Obtaining the artifacts

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

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

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

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-bom</artifactId>
<version>5.2.3</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>5.2.3</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2.1. Using the Platform BOM

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

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

3. What's new in version 5.2.x

- **JDK 9+ module support:** Support for JDK 9+ module system using [Automatic-Module-Name](#).
- **AsyncDatastore:** The new [holon-async-datastore](#) artifact was introduced to provide a core [AsyncDatastore](#) API. The [AsyncDatastore](#) API provides asynchronous Datastore API operations, using the standard JVN [CompletionStage](#) type to provide the operations result. See [Available Datastores](#) to check the specific [AsyncDatastore](#) support.
- **AsyncRestClient:** The new [holon-async-http](#) artifact was introduced to provide a core [AsyncRestClient](#) API. The [AsyncRestClient](#) API provides asynchronous REST client API operations, using the standard JVN [CompletionStage](#) type to provide the operations result.
- **Property model:** The new [CollectionProperty](#) interface was introduced to provide specific [Collection](#) type [Property](#) values, specifically using the standard [Set](#) and [List](#) collection types. See [Collection properties](#).
- **QueryConfigurationProvider builder:** A fluent builder is now available for the [QueryConfigurationProvider](#) API. The builder can be obtained using the [builder\(\)](#) static method.
- **Datastore Query distinct support:** The new [distinct\(\)](#) [QueryBuilder](#) method can be used to obtain *distinct* query projection result values. See [Distinct query projection results](#).
- **Datastore Query select all projection:** A new [SelectAllProjection](#) is now available to obtain all the values of a persistent data entity instance using the [Datastore](#) query API. See [Builtin query projections](#).
- **Datastore Query locks:** The new [LockQuery](#) API is a [Datastore](#) query extension to provide database lock support, through the [LockSupport](#). Currently, only the *pessimistic* lock mode is supported. For the query lock support availability, see the specific [Datastore](#) implementations documentation.
- **@EnableDatastoreConfiguration annotation:** The [EnableDatastoreConfiguration](#) annotation can be used on Spring configuration classes to enable a post processor for automatic [Datastore](#) Spring bean types configuration. See [Automatic Datastore beans configuration using @EnableDatastoreConfiguration](#).
- **LocalizationChangeListener:** The new [LocalizationChangeListener](#) interface can be used to listen to [LocalizationContext](#) localization changes. See [Listening to localization changes](#).
- The Holon Platform **Spring** and **Spring Boot** support is now targeted on the version 5.x and 2.1.x respectively.

3.1. Migrating from version 5.1.x

3.1.1. API changes

- **Transaction API:** The [TransactionConfiguration](#) API was made more abstract and implementation independent, using the [TransactionOptions](#) interface to represent transaction configuration options. The [TransactionOptions](#) has to be extended by concrete implementations to represent and provide the supported transaction configuration attributes. For this reason, the [com.holonplatform.core.datastore.transaction.TransactionIsolation](#) enumeration was moved to the **holon-jdbc** module.

Furthermore, the [TransactionException](#) class hierarchy was moved from the [Transaction](#) interface to the new [../api/holon-core/com/holonplatform/core/datastore/transaction/TransactionStatus.html\[TransactionStatus^\]](#) interface, which represents a base transaction definition, shared with the asynchronous Datastore implementations.

4. What's new in version 5.1.x

4.1. Property model

- The [Property](#) instance identification strategy can now be customized using specific *equals* and *hashCode* handlers. See [Property naming and identity](#).
- The [PathProperty](#) interface now provides a set of sub types, to consistently handle property *expressions* which are type specific, for the main Java types (**String**, **Numbers**, **Temporal types** and **Boolean**). See [PathProperty sub types](#).
- The [PropertySet](#) interface now supports a generic **configuration** container, likewise the [Property](#) interface. See [PropertySet configuration](#).
- The [PropertySet](#) interface now supports *identifier* properties declaration, which can be used to provide a virtual primary key to distinguish a [PropertyBox](#) instance from another (using the *identifier* property values), both at Java objects level (*equals* and *hashCode*) and at persistence architecture level. See [Identifier properties](#).
- Just like the [Property](#) instances, the [PropertyBox](#) instances identification strategy can now be customized using specific *equals* and *hashCode* handlers. Furthermore, if the [PropertyBox](#) property set declares one or more *identifier* property, their values will be used by default to implement the [PropertyBox](#) instance identification strategy. See [PropertyBox instances identification](#).
- The new [BeanPropertySetPostProcessor](#) interface can be used to extend the [BeanIntrospector](#) Bean introspection strategy at Bean property set level, allowing for example to customize the Bean property set configuration. See [BeanPropertySetPostProcessor](#).
- The [@DataPath](#) annotation can be used on Bean classes to declare a **data path mapping** name different from the Bean class or property name, when the Bean class is used in a persistence context and it is bound to a data model definition. See [@DataPath](#).

4.2. Datastore

- A deep revision of the **Expression** based architecture, which is the foundation, above all, of the **Datastore** API and the abstract **Query** engine. This lead to a more consistent and extensible architecture, along with considerable performance improvements. See the [Query](#) and [Datastore API extensions](#) sections of the **Datastore** documentation.
- New standard **QueryFunction** implementations was made available for the core **Datastore** API. Besides [String related functions](#), a set of **temporal** data types related functions is now available to obtain current date/time (with `java.time.*` types support) and to extract a temporal part (`year`, `month`, `day`, `hour`). See [Temporal functions](#).
- A new **Transactional** API which can be used when a **Datastore** implementation supports **transactions**, which can be managed at a higher level, in an abstract and implementation-independent way. The **Transactional** API allows to execute **Datastore** operations whithin a transaction, taking care of the transaction lifecycle and providing transaction reference to perform *commit* and *rollback* operations. See [Transactional Datastores](#).
- New **DataMappable** API to represent data model *mappings*. See [DataMappable](#).

4.3. JWT authentication

- The JWT configuration now supports a wider range of key sources, formats (such as the **PKCS#12** key store format) and encodings (such as the **PEM** encoding) for private and public key declarations when an asymmetric signing algorithm is used. See [JWT configuration](#).

4.4. Spring ecosystem integration

- The **TenantScopeManager** API is now available to manage the *tenant* scoped beans lifecycle. See [Tenant scoped beans lifecycle](#).
- The new **holon-spring-security** artifact provides integration between the Holon Platform authentication and authorization architecture and the **Spring Security** one. Furthermore, a **holon-starter-security** Spring Boot starter is now available for quick project setup. See [Spring Security integration](#).

4.5. Migrating from version 5.0.x

4.5.1. Deprecations

- **PropertySet API:** [join](#). Using this method causes the loss of any property set configuration and/or identifier property declaration. Use the default [PropertySet](#) builder instead.
- **BeanPropertySet API:** [create providing a parent path](#). The bean properties *parent path* will always be the **FinalPath** which represents the Bean class, with the bean fully qualified class name as path name. Use the default `create(Class<? extends T> beanClass)` method instead. The same considerations are valid for the `getPropertySet` method of the [BeanIntrospector](#) API.
- **BeanPropertySet/BeanPropertyInspector API:** [requireProperty](#) methods. The methods name are changed in **property** for consistency with the naming conventions of the new

BeanPropertyInspector API.

- **SubQuery API:** creation methods providing a Datastore. The Datastore parameter is no longer required. Use the new create(...) method versions which does not require the Datastore parameter.
- **JwtConfigProperties API:** private and public key source configuration properties. The private and public key sources can now be specified using the publickey.source property and using the file: and classpath: prefixes to declare the source type.
- **JwtTokenBuilder API:** static token creation method. The JwtTokenBuilder is now an interface. The default implementation can be obtained using the get() static method.

4.5.2. Property model

With the introduction of the PathProperty sub types, a PathProperty declaration should be made using the most suitable sub type, if available.

This is also necessary to use the convenience QueryFilter and QueryFunction static builder methods to create an expression using the property itself. The expression builder methods are now **organized by sub type**, so, for example, the contains method is only available for String type properties and the StringProperty sub type should be used in this case.

```
final StringProperty STR = StringProperty.create("name"); ①
```

```
QueryFilter filter = STR.contains("value"); ②
```

① Create a StringProperty path property type with the name path name

② The StringProperty type makes available convenience expression builder methods according to the String property type, for example contains

5. Core API, services and components

5.1. 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>5.2.3</version>
```

5.2. 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` interface 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

5.2.1. Obtaining Context resources

The `Context` API can be used to statically obtain a resource, using a resource *key* to identify the requested resource type.

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.

To organize and provide the resource references in the most versatile and extensible way, the `Context` API is organized in *scopes*. A context *scope* represents a resources registry, available to the `Context` API in order to perform the context resources lookup and provide a consistent resource reference.

Furthermore, the context *scopes* are the entry points for context resources registration.

See [the next section](#) for details.

5.2.2. Scopes

The `Context` API is organized in *scopes*, represented by the `ContextScope` interface.

Each *scope* acts as a sub-registry of resources, with its 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 priority number.

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

- Each registered *scope* is queried to obtain the resource instance through a specified resource key, starting from the scope with the highest priority.
- The actually returned resource instance is the one obtained from the first *scope* which provides a resource instance bound to the requested resource key, if any.

A reference to a registered *scope* can be obtained from the `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`.

Example of resource registration:

Context resource registration

```
Context.get().scope("myscope") ①  
    .ifPresent(scope -> scope.put("myResourceKey", new ResourceType())); ②
```

① Get the scope named `myscope`, if available

② Register a new resource instance using `myResourceKey` as resource key

5.2.3. Default Scopes

The Holon platform provides two default context *scopes*, automatically registered and made available to the `Context` API:

1. A **ClassLoader**-bound scope, which handles resources as *singleton* instances within the reference `ClassLoader`, that is, at most one resource instance for a given resource 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 the 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** API provides methods to directly obtain the default scopes:

```
Optional<ContextScope> scope = Context.get().classLoaderScope(); ①  
scope = Context.get().classLoaderScope(aClassLoader); ②  
  
Optional<ContextScope> threadScope = Context.get().threadScope(); ③  
threadScope = Context.get().threadScope(aClassLoader); ④
```

- ① Get the default `ClassLoader` scope using the default `ClassLoader`
- ② Get the default `ClassLoader` scope using a specific `ClassLoader`
- ③ Get the default `Thread` scope using the default `ClassLoader`
- ④ Get the default `Thread` scope using a specific `ClassLoader`

The **Context** interface provides some other useful methods to access the special **Thread**-bound scope, to perform an operation ensuring that a context resource is bound to the `Thread` scope before the operation begins and removed from the `Thread` scope just after the operation ends:

```
Context.get().executeThreadBound("resourceKey", resourceInstance, () -> {  
    // do something ①  
});  
  
Context.get().executeThreadBound("resourceKey", resourceInstance, () -> {  
    // do something ②  
    return null;  
});
```

- ① Execute a *Runnable* operation, binding the resource instance with given *resourceKey* to the current `Thread` before execution and removing the binding after the operation execution
- ② Execute a *Callable* operation, binding the resource instance with given *resourceKey* to the current `Thread` before execution and removing the binding after the operation execution

5.2.4. Context extension: adding Scopes

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

To create and register a new context scope, the following steps are required:

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 required *resourceType* type, if the resource is currently available from the 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. When a jar with a valid *com.holonplatform.core.ContextScope* file is available from classpath, the context scope is automatically registered and made available from the **Context** API.



Some Holon platform modules provides specific context scope implementations, which are automatically registered when the module is available from classpath. See each module reference documentation to learn about the available additional platform context scopes.



The **core** module itself provides a *Spring bean factory* based context scope, which uses the *Spring framework* bean registry to provide bean references as context resources. See the [Spring ecosystem integration](#) section for further information and the [Spring context scope](#) section for details.

5.3. Configuration and parameters

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

Configuration properties definition, provision and management is supported by the following 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.

5.3.1. Configuration properties

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*

5.3.2. 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 are performed when applicable.

The Holon 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

5.3.3. 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 name using the pattern **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 property values.

5.3.4. ParameterSet

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

It provides methods to inspect the parameter set and obtain the parameter values.

The `ConfigProperty` interface is fully supported and can be used as a *typed* parameter representation, avoiding type cast errors and always exposing the parameter value type.

ParameterSet examples

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

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

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

Optional<String> value = set.getParameter("testParameter", String.class); ⑤
String val = set.getParameter("testParameter", String.class, "default"); ⑥

Optional<String> configPropertyValue = set.getParameter(property); ⑦
String configPropertyVal = set.getParameter(property, "default"); ⑧

boolean matches = set.hasParameterValue("testParameter", "myValue"); ⑨
matches = set.hasParameterValue(property, "myValue"); ⑩
```

- 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 its value is not null
- ⑤ Get a parameter value of `String` type
- ⑥ Get a parameter value of `String` type with default fallback value when the parameter value is not available
- ⑦ Get a parameter value using a `ConfigProperty`
- ⑧ Get a parameter value using a `ConfigProperty`, providing the default fallback value if not available
- ⑨ Checks if a parameter named `testParameter` is present and its value equals to the `myValue` value
- ⑩ Checks if a `ConfigProperty` is present and its value equals to the `myValue` value

5.4. Data validation

The main Holon Platform entry point to configure and perform data validation is the `Validator` interface.

5.4.1. Validator

The `Validator` interface can be implemented by a class which performs the validation of a value. A

Validator is generalized on the value type which the validator is able to validate.

The `validate(T value)` method performs the actual validation of the specified value, throwing a **ValidationException** if the value is not valid.

The **ValidationException** is *localizable*, supporting invalid value message localization. Furthermore, it can act as a container for multiple validation exceptions.

The **Validator** interface provides static builder methods to create a Validator providing a condition predicate and with validation error message localization support.



See the [Internationalization](#) section for information about messages localization.

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 The same **Validator** created using the **Validator.create()** builder method
- ③ The same **Validator** created using the **Validator.create()** builder method and providing an invalid value message localization code

5.4.2. Bultin validators

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

The available builtin validators can be obtained using the **Validator** interface static builder methods:

- **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

Builtin validators example

```
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()); ③
}
```

- 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 The default **getLocalizedMessage()** method of the **ValidationException** class actually returns the localized validation error message, if a message localization code was provided and the platform localization context is setted up. See the [Internationalization](#) section for information about messages localization.

5.4.3. 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)** method 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.

5.5. StringValuePresenter

The `StringValuePresenter` API deals with `String` representation of a generic Object.

Presentation *parameters* can be used to tune the `String` presentation strategy.

5.5.1. 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 `;`. The `holon.value-presenter.values-separator` parameter can be used to change the separator character to use.



See the [Internationalization](#) section for information about messages localization.

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().withParameter(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`

5.6. Internationalization

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

5.6.1. Localizable messages

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

- A **default** message: The default message to use if the localized message is not available or a localization provider is not available.
- A localization **message code**: The symbolic message code to use as identifier to provide message localizations.
- Optional message **arguments**: A set of arguments to be used to replace conventional placeholders in the message String with the actual values at message localization time.

The [Localizable](#) interface is available to represent a localizable message.

```
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

5.6.2. @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 annotation attributes are:

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

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



For example, the default `StringValuePresenter` supports the `@Caption` annotation for `enum` values presentation.

5.6.3. Message providers

To perform actual messages localization the `MessageProvider` API 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 Holon platform makes available a default `MessageProvider`, which uses *properties* files as message localization containers. It can be created using the `fromProperties(String... basenames)` static method:

```
MessageProvider messageProvider = MessageProvider.fromProperties("messages").build(); ①

messageProvider = MessageProvider.fromProperties("i18n/messages").encoding("UTF-8")
    .build(); ②
```

① Build a `MessageProvider` which uses *properties* files with `messages` as base name

② Build a `MessageProvider` which uses *properties* files with `messages` as base name in the `i18n` folder and set `UTF-8` as encoding

Properties file names are resolved using the configured *basenames* as prefix. This prefix can be followed by the *Locale* language, country and variant codes, separated by an underscore (_) character. The files must have the *.properties* extension.

The *basenames* follow the *java.util.ResourceBundle* conventions: essentially, a fully-qualified classpath location. If the base name 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 folder structure.

The *Locale* attributes are used to build a fallback message localization resolution chain, starting from the most qualified *Locale* definition and matched against the *Locale* for which the message localization is requested.

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 the list of the available localizations (translations), organized by message localization code. For example the *test.msg=translation* line declares *translation* as the localization of the *test.msg* message code.

5.6.4. LocalizationContext

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

5.6.5. Building a LocalizationContext

The *LocalizationContext* interface provides a fluent builder to create *LocalizationContext* instances:

```

LocalizationContext localizationContext = LocalizationContext.builder()
    .withMessageProvider(MessageProvider.fromProperties("messages").build()) ①
    .withMessageProvider(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`

5.6.6. 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), it can be obtained by using the convenience `getCurrent()` static method.

The `require()` static method can be used to obtain the current `LocalizationContext` or throwing an exception if it's not available as context resource.

Furthermore, the `requireLocalized()` static method acts the same as the `require()` method, but additionally requires that the current `LocalizationContext` is localized.

5.6.7. 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`, the `localize(...)` method can be used, providing the `Locale` instance.

To fine tune the context localization, a `Localization` object can be used instead of a simple `Locale`.

A `Localization` is bound to a `Locale` and allows to setup:

- A *parent* `Localization`, i.e. the `Localization` to use as fallback when a localization operation cannot be successfully performed using the current localization, for example because a message translation is not available. This allows the creation of a `Localization` hierarchy;
- 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`

5.6.8. Listening to localization changes

The `LocalizationChangeListener` interface can be used to listen to context localization changes.

The `LocalizationContext` localization, and therefore the current context `Locale`, changes when the `localize(...)` method is invoked.

The `addLocalizationChangeListener(LocalizationChangeListener listener)` method, or the corresponding `withLocalizationChangeListener(LocalizationChangeListener listener)` builder method, can be used to be notified of localization changes.

The `LocalizationChangeEvent` provides information about the `LocalizationContext` from which the localization change was triggered and the new `Locale` of the context, if available.

5.6.9. Using the LocalizationContext

The `LocalizationContext` API provides several methods to perform localizations of messages, temporal values 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

The Java 8 `java.time.*` API is fully supported by the `LocalizationContext` API.

```
LocalizationContext ctx = LocalizationContext.builder()
    .withMessageProvider(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

5.6.10. MissingMessageLocalizationListener

One or more `MissingMessageLocalizationListener` can be registered to a `LocalizationContext` to be notified when a message localization is missing from any of the available `MessageProvider`.

The `MissingMessageLocalizationListener` provides the localization message code for which the localization was requested, in addition to the `Locale` and the optionally provided default message.

The `withMissingMessageLocalizationListener(...)` method of the `LocalizationContext` builder can be used to register a `MissingMessageLocalizationListener`.

```
LocalizationContext ctx = LocalizationContext.builder()
    .withMissingMessageLocalizationListener((locale, messageCode, defaultMessage) -> {
        ①    LOGGER.warn("Missing message localization [" + messageCode + "] for locale [" +
        locale + "]");
    }).build();
```

- ① Add a `MissingMessageLocalizationListener` to the `LocalizationContext` which logs missing message localizations

5.7. 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.

5.7.1. Property naming and identity

Each **Property** provides a **name** through the `getName()` method. The property name can have a different meaning for each implementation category. For example, if the **Property** is actually bound to a concrete data model, the name can represent the data model attribute identifier.

Since the property name semantics is highly dependent from the concrete property implementation and use, it is not used by default to identify the property in the Holon platform architecture. From a standard Java point of view, a **Property** is a Java class and it is identified by its address in memory. For this reason, it makes sense to declare the **Property** instances as *static* class (or, better, interface) members.

But if a finest identity logic is required (and more control on **Property** equality semantics is needed), the base **Property** builder allows to provide a specific **Property** identity and equality strategy.

This is achieved through the **HashCodeProvider** and **EqualsHandler** interfaces. This *functional* interfaces can be used to provide a custom **hash code** and **equals** logic, to override the default Java Objects `hashCode` and `equals` implementations. The combination of these two methods leads to a consistent **Property** identity definition within the Java objects model and the Holon Platform architecture.

For example, to use the **Property name** as unique property identifier, the **Property equals** and **hashCode** strategy can be defined as follows:

```
Property.Builder<String, Property<String>, ?> builder = getPropertyBuilder();

builder.hashCodeProvider(property -> Optional.of(property.getName().hashCode())) ①
    .equalsHandler((property, other) -> { ②
        if (other instanceof Property)
            return property.getName().equals(((Property) other).getName());
        return false;
    });
```

① Set the **Property hash code** strategy using the property *name*

② Set the **Property equals** strategy using the property *name*



The example above is not production ready, since the **null** values checking and management are completely absent.

5.7.2. Configuration

The **Property** interface provides a generic container to store and manage property configuration attributes, and it is mainly intended as a custom configuration attributes handler for extension purposes and to better integrate the **Property** representation in specific application architectures.

The property configuration is represented by the **PropertyConfiguration** interface and can be obtained through the `Property.getConfiguration()` method.

The **PropertyConfiguration** interface extends the Holon platform **ParameterSet** API, providing a set

of methods to inspect and obtain the configuration attributes. Since the `ParameterSet` API can be compared to a *name-value* map, it is highly flexible and versatile, allowing to store and retrieve anything that can be represented as a Java object.

Besides the generic configuration attributes, the `PropertyConfiguration` interface explicitly declares a `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 is used by default by a number of platform services to perform consistent operations on the 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.

The property configuration is considered as immutable during the `Property` lifetime, and can be set up at `Property` build time, using the property builder's `configuration(String parameterName, Object value)` methods. Since the `PropertyConfiguration` interface extends the `ParameterSet` API, the `Configuration properties` type is fully supported.

Property configuration example

```
final ConfigProperty<Long> EXAMPLE_CFG = ConfigProperty.create("exampleConfig", Long
.class);

final PathProperty<LocalDate> PROPERTY = PathProperty.create("example", LocalDate
.class)
    .temporalType(TemporalType.DATE_TIME) ①
    .withConfiguration("myAttribute", "myValue") ②
    .withConfiguration(EXAMPLE_CFG, 7L); ③

PropertyConfiguration cfg = PROPERTY.getConfiguration(); ④
Optional<String> value1 = cfg.getParameter("myAttribute", String.class); ⑤
Long value2 = cfg.getParameter(EXAMPLE_CFG, 0L); ⑥
```

- ① Set the property `TemporalType`
- ② Set a custom configuration attribute
- ③ Set a configuration attribute using a `ConfigProperty`
- ④ Get the property configuration
- ⑤ Get the `myAttribute` configuration attribute
- ⑥ Get a property configuration value using a `ConfigProperty`, providing a default fallback value

5.7.3. Converters

Each **Property** supports a **PropertyValueConverter**, which can be used to perform property value conversions from a the actual **Property** type to a different value type and vice-versa.

Typically, the **PropertyValueConverter** API is used to map the data model attribute type to which the **Property** is bound to the actual **Property presentation** type. 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.

Property value converter example

```
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; ❶
    }

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

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

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

};
```

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

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

A **PropertyValueConverter** can be provided at **Property** build time, using the appropriate property builder methods. The base **Property** builder provides also a method to declare the property value conversion logic using standard Java *functions*.

Futhermore, the **Property** interface makes available a convenience **getConvertedValue** method to obtain the **Property model** value using the configured value converter, if available, or returning the actual property value if not.


```
Property.Builder<Integer, Property<Integer>, ?> builder = getPropertyBuilder();

builder.converter(String.class, ①
    v -> (v != null) ? Integer.parseInt(v) : null, ②
    v -> (v != null) ? String.valueOf(v) : null); ③
```

- ① Set the **Property** value converter providing the *model* data type and the value conversion functions
- ② Set the value conversion function to convert a **String** model value into the **Integer** property type
- ③ Set the value conversion function to convert the **Integer** property value type to a **String** type model value

The Holon platform provides some useful **builtin converters** out-of-the-box:

- *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.

The **PropertyValueConverter** interface provides static methods to obtain the listed builtin converters.

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

Property.Builder<Boolean, Property<Boolean>, ?> builder = getPropertyBuilder();

builder.converter(PropertyValueConverter.numericBoolean(Integer.class)); ⑥
```

- 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
- ⑥ Set a *numeric boolean* builtin converter a **Property** build time, to map an **Integer** model value to

a **Boolean** property type.

5.7.4. Localization

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.

Since **Property** is a **Localizable**, the property localization attributes can be used anywhere is meaningful and in a seamless way within an internationalization environment.

This allows to centralize the property display and localization attributes and store it tightly coupled with the property definition.

See **Internationalization** for additional details about the Holon platform internationalization architecture.

5.7.5. Validation

The **Property** interface implements the **Validatable** API to support property value validation using **Validators**.

A set of property value **Validator** can be added to the property definition at property build time and later used to validate a value against the property definition.

This allows to tightly couple the validation logic to the property representation and make it available to any actor which will use the property itself, promoting validation factorization and value consistency across service or application stack layers.

See the **Data validation** section for detailed information about validators definition and usage.

```
Property.Builder<Integer, Property<Integer>, ?> builder = getPropertyBuilder();  
  
builder.withValidator(Validator.notNull()) ①  
       .withValidator(Validator.lessThan(10)); ②
```

① Add a property validator to ensure the value is not **null**

② Add a property validator to ensure the value is less than **10**

5.7.6. Read only attribute

The **Property** interface declares by default a **read-only** attribute, which can be used by property handlers to check if the property represents a binding with a data model attribute in both directions: to read and to write values.

The actual meaning of the read-only attribute can depend from each implementation and data model architecture. The read-only attribute is immutable, and bound to each **Property** API extension, so it cannot be changed at property definition time or during the property lifecycle.

5.8. PathProperty

When a property is bound to a data model attribute, the Holon platform provides the [PathProperty](#) interface to declare a [Property](#) and represent the binding with the data model attribute.

The binding with the data model attribute is represented by the [Path](#) interface, extended by the [PathProperty](#) itself. A [Path](#) is the symbolic [String](#) representation of the data model attribute, and can assume a different meaning for each concrete data persistence context.

Generally speaking, the [Path](#) **name** corresponds to the name of the data model attribute. In the [PathProperty](#) case, the property name is the path name itself.

A [Path](#) can be hierarchical, supporting *parent* path declaration. When a path hierarchy is defined, the full path name is represented by default by the concatenation of the path hierarchy, starting from the first ancestor (the *root* path), using the *dot* (.) character as path hierarchy separator. The default `fullName()` method can be used to obtain the full path name.

Just like a [Property](#), a [Path](#) is **typed**, i.e. declares the Java type of the path segment which represents. Talking about a data model, this corresponds to the type of the data model attribute to which the path segment is bound. For the [PathProperty](#), the path and the property type coincide.

The [Path](#) can also be used independently from a [Property](#) definition. The [Path](#) interface provides builder methods to create path instances:

```
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](#) named "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 the parent path
- ⑤ The path full name will be [pathName.subName](#)

The [PathProperty](#) API combines the [Property](#) and [Path](#) APIs, allowing to **declare a property which is bound to a data model path**.

The [PathProperty](#) builder makes available all the [Property](#) and [Path](#) builder methods, allowing to declare the property path **name**, the property and path **type** and to setup the property configuration attributes, a property value converter, register property value validators and define property localization attributes.

A property builder is obtained using the `create(...)` static methods:

```

public final static PathProperty<Long> ID = PathProperty.create("id", Long.class) ①
    .withConfiguration("test", 1) ②
    .withValidator(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** (the *path* name) of type **Long**
- 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** (the *path* name) of **Boolean** type but which is bound to a **Integer** data model attribute type
- ⑦ Set the converter to perform conversion between the **Integer** data model values and the **Boolean** property type

As a **Path**, the **PathProperty** builder supports property hierarchy definition, allowing to set the parent property path:

```

public final static PathProperty<String> PARENT_PROPERTY = PathProperty.create("
parent", String.class);

public final static PathProperty<String> A_PROPERTY = PathProperty.create("child",
String.class)
    .parent(PARENT_PROPERTY); ①

```

- ① Create a **PathProperty** named **child** and set the **PARENT_PROPERTY** property definition as parent path

5.8.1. PathProperty sub types

When a **PathProperty** is used as a data model *query* expression, the property value type can be a discriminant for the expression usage within a query definition. For example, a query *restriction* expression can be consistent only for specific data type, or a query *function* could only be applicable for a certain data type.

Since the **PathProperty** interface provides useful convenience methods to create *expressions* from the property itself, a set of **PathProperty** sub types are provided to deal with the most common data types.

Each sub type provides one or more specific builder method too, to easily and quickly create the property definition.

Four **PathProperty** sub types are provided out-of-the-box by the Holon platform:

- **StringProperty**: Represents a **String** type **PathProperty** and extends **StringQueryExpression**, providing String related expression builder methods.
- **NumericProperty**: Represents a **Number** type **PathProperty** and extends **NumericQueryExpression**, providing numbers related expression builder methods.
- **TemporalProperty**: Represents a *temporal* type **PathProperty** and extends **TemporalQueryExpression**, providing temporal related expression builder methods.
- **BooleanProperty**: Represents a *boolean* type **PathProperty**.
- **PropertyBoxProperty**: Represents a **PropertyBox** type **PathProperty**, i.e. a property which value is a set of properties with its values. The property set definition must be provided at construction time. See **PropertySet** and **PropertyBox** for details.

Each **PathProperty** sub type interface provide static builder methods to create property instances using each specific type.

```
public final static StringProperty STRING_PROPERTY = StringProperty.create("name"); ①
public final static NumericProperty<Integer> INTEGER_PROPERTY = NumericProperty.
create("name", Integer.class); ②
public final static NumericProperty<Long> LONG_PROPERTY = NumericProperty.longType(
"name"); ③
public final static TemporalProperty<LocalDate> LDATE_PROPERTY = TemporalProperty
.localDate("name"); ④
public final static TemporalProperty<Date> DATE_PROPERTY = TemporalProperty.date("
name")
    .temporalType(TemporalType.DATE); ⑤
public final static BooleanProperty BOOLEAN_PROPERTY = BooleanProperty.create("name");
⑥
```

- ① Create a **StringProperty**
- ② Create a **NumericProperty** of **Integer** type
- ③ Create a **NumericProperty** of **Long** type
- ④ Create a **TemporalProperty** of **LocalDate** type
- ⑤ Create a **TemporalProperty** of **Date** type
- ⑥ Create a **BooleanProperty**

5.9. VirtualProperty

The **VirtualProperty** interface represents a **Property** which is not directly bound to a data model attribute, but which instead provides its value through a **PropertyValueProvider**.

A **VirtualProperty** is declared as *read only* by default.

The property value provision is delegated to the **PropertyValueProvider** at runtime. The provided value must be consistent with the **VirtualProperty** type.

The `PropertyValueProvider` is configured at `VirtualProperty` definition time, and it is immutable during the property lifecycle.

When a `VirtualProperty` is used within objects and structures which explicitly support it, the current data model context can be used to provide the virtual property value. The current data context is represented through a `PropertyBox` and it is provided to the `PropertyValueProvider` value providing method. This way, the value of other properties can be used to calculate the virtual property value.



See [Managing property values using a PropertyBox](#) for further information.

```
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`

5.10. Collection properties

When the value of a `Property` is a `Collection` type, a set of specific `Property` sub types are available to easily represent and manage a collection type `Property`.

All the builtin collection property types inherits from the `CollectionProperty` interface, which provides the **collection elements type** through the `getElementType()` method.

The concrete collection property type made available by the core platform module are:

- `ListPathProperty`: A `PathProperty` sub type which handles `Collection` type values using a `java.util.List` as collection representation.
- `SetPathProperty`: A `PathProperty` sub type which handles `Collection` type values using a `java.util.Set` as collection representation.

The collection type properties listed above can be constructed and used just like any other `PathProperty` type.

```

final ListPathProperty<String> STR = ListPathProperty.create("str", String.class); ①
final SetPathProperty<Integer> ITG = SetPathProperty.create("str", Integer.class); ②

Class<?> elementType = STR.getElementType(); ③

PropertyBox box = PropertyBox.create(STR, ITG);

box.setValue(STR, Collections.singletonList("a")); ④
List<String> listValue = box.getValue(STR); ⑤

box.setValue(ITG, Collections.singleton(1)); ⑥
Set<Integer> setValue = box.getValue(ITG); ⑦

```

- ① Create a `ListPathProperty` with `String` type elements
- ② Create a `SetPathProperty` with `Integer` type elements
- ③ Get the actual collection elements type
- ④ Set a `ListPathProperty` value into a `PropertyBox`
- ⑤ Get a `ListPathProperty` value from a `PropertyBox`
- ⑥ Set a `SetPathProperty` value into a `PropertyBox`
- ⑦ Get a `SetPathProperty` value from a `PropertyBox`

5.10.1. Collection virtual properties

Two convenience `VirtualProperty` types are also provided to handle `Collection` type `VirtualProperty` values:

- `ListVirtualProperty`: A `VirtualProperty` sub type which handles `Collection` type values using a `java.util.List` as collection representation.
- `SetVirtualProperty`: A `VirtualProperty` sub type which handles `Collection` type values using a `java.util.Set` as collection representation.

The virtual collection type properties listed above can be constructed and used just like any other `VirtualProperty` type.

```

static final StringProperty STR = StringProperty.create("test");

static final ListVirtualProperty<String> VIRTUAL_LIST = ListVirtualProperty.create(
    String.class, ①
    pb -> {
        String value = pb.getValue(STR);
        if (value != null) {
            List<String> l = new ArrayList<>();
            for (char c : value.toCharArray()) {
                l.add(String.valueOf(c));
            }
            return l;
        }
        return Collections.emptyList();
    });

static final SetVirtualProperty<String> VIRTUAL_SET = SetVirtualProperty.create(
    String.class, ②
    pb -> {
        String value = pb.getValue(STR);
        if (value != null) {
            Set<String> l = new HashSet<>();
            for (char c : value.toCharArray()) {
                l.add(String.valueOf(c));
            }
            return l;
        }
        return Collections.emptySet();
    });

```

① Create a `ListVirtualProperty` which provides a `List` of the characters of the `STR` property value

② Create a `SetVirtualProperty` which provides a `Set` of the characters of the `STR` property value

5.10.2. Collection property value converters

The `CollectionProperty` builder API provides convenience methods to configure a `PropertyValueConverter` for the collection property type specifying the **collection elements** conversion logic, instead of the whole collection conversion logic.

The `elementConverter` method can be used for this purpose:

```

final ListPathProperty<String> STR = ListPathProperty.create("str", String.class) ①
    .elementConverter(Integer.class, v -> String.valueOf(v), v -> Integer.valueOf(v));
②

```

① Create a `ListPathProperty` with `String` type elements

② Configure a property value converter providing the single element conversion logic: in this example, `Integer` is specified as model element type and the functions to provide the value

conversion in both directions are provided

5.11. Organizing and collecting properties using a `PropertySet`

The `PropertySet` interface represents an immutable set of `Property`, providing methods to inspect and obtain the `Property` elements. It is an `Iterable`, in analogy with the standard Java collections structures.

A `PropertySet` allows to collect and organize a set of properties and use it consistently through the application stack levels.

In reference to a data model, a `PropertySet` can represent a data model *entity* as a collection of data model attributes. In this sense, it can be used for example as a *query* projection, to obtain the values of the properties which belongs to the set as a `PropertyBox` instance.



See the [Query](#) section for information about query definition and execution.

The `PropertySet` API makes available appropriate builder methods to create `PropertySet` definitions.

```
final StringProperty NAME = StringProperty.create("name");
final StringProperty SURNAME = StringProperty.create("surname");
final NumericProperty<Integer> SEQUENCE = NumericProperty.integerType("surname");

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

PropertySet<Property<?>> set2 = PropertySet.builder().add(set).add(SEQUENCE).build();
③
```

- ① Create a `PropertySet` containing `NAME` and `SURNAME` properties
- ② Create a `PropertySet` containing `NAME` and `SURNAME` properties using the fluent builder
- ③ Create a `PropertySet` adding the `SEQUENCE` property to the properties of the previous property set

The `PropertySet` API can be used as a standard `Iterable`. Furthermore, it provides convenience methods to inspect the property set contents.

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

final PropertySet<Property<?>> SET = PropertySet.of(NAME, SURNAME); ①

boolean contains = SET.contains(NAME); ②
SET.forEach(p -> p.toString()); ③
String captions = SET.stream().map(p -> p.getMessage()).collect(Collectors.joining());
④
List<Property<?>> list = SET.asList(); ⑤
```

- ① Create a `PropertySet` which contains the `NAME` and `SURNAME` properties
- ② Check if `PropertySet` contains the property `NAME`
- ③ Use the `forEach` operation to invoke the `toString` method for each property of the set
- ④ Use a `stream` of the properties of the set to join the property captions in a `String`
- ⑤ Obtain the `PropertySet` as a `List` of properties

5.11.1. PropertySet configuration

The `PropertySet` API provides a generic container to store and manage property configuration attributes, and it is mainly intended as a custom configuration attributes handler for extension purposes and to better integrate the `PropertySet` representation in specific application architectures.

The property set configuration can be obtained through the `PropertySet.getConfiguration()` method and is represented by a `ParameterSet`, providing a set of methods to inspect and obtain the configuration attributes. Since the `ParameterSet` API can be compared to a *name-value* map, it is highly flexible and versatile, allowing to store and retrieve anything that can be represented as a Java object.

5.11.2. Identifier properties

The `PropertySet` API supports the declaration of the properties which act as *identifiers* for the property set. Such properties must belong to the property set itself.

The identifier properties are declared at `PropertySet` definition time and can be later inspected using the `PropertySet` API. The `PropertySet` interface provides also `builderOf` static method to directly provide the properties of the set and declare the identifiers immediately after.

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

PropertySet<Property<?>> SET = PropertySet.builder().add(ID).add(NAME).identifier(ID)
    .build(); ①

SET = PropertySet.builderOf(ID, NAME).identifier(ID).build(); ②

Set<Property<?>> ids = SET.getIdentifiers(); ③
Optional<Property<?>> id = SET.getFirstIdentifier(); ④
SET.identifiers().forEach(p -> p.toString()); ⑤
```

- ① Create a `PropertySet` with the `ID` and `NAME` properties, declaring `ID` as the identifier property
- ② The same operation using the convenience `builderOf` method
- ③ Get the identifier properties as a `Set`
- ④ Get the first identifier property, if available
- ⑤ Get the identifier properties as a `Stream`

5.11.3. Using a PathPropertySetAdapter

The `PathPropertySetAdapter` API can be used to inspect a `PropertySet` using `Path` type expressions and allows to:

- Check if a `Property` which corresponds to a given `Path` is available in the property set and obtain it.
- Obtain the `Path` representation of a property set `Property`, if applicable.
- Get the property set identifier properties as `Path` representations.
- Obtain a Stream of all the available `Path` representations of the properties in the property set.

```
final StringProperty STR = StringProperty.create("str");
final NumericProperty<Integer> ITG = NumericProperty.integerType("itg");
final PropertySet<?> SET = PropertySet.of(STR, ITG);

final Path<String> PATH = Path.of("str", String.class);

PathPropertySetAdapter adapter = PathPropertySetAdapter.create(SET); ①

boolean contains = adapter.contains(PATH); ②
Optional<Property<String>> property = adapter.getProperty(PATH); ③
Optional<Path<String>> path = adapter.getPath(STR); ④
Stream<Path<?>> paths = adapter.paths(); ⑤
```

- ① Create a `PathPropertySetAdapter` from given `SET`
- ② Checks if a `Path` is available in the property set
- ③ Get the `Property` of the property set which corresponds to given `Path`, if available
- ④ Get the `Path` which corresponds to given `Property`, if available
- ⑤ Obtain a `Stream` of all the available `Path` representations in the property set

The `PathPropertySetAdapter` behaviour can be customized through its builder API, providing:

- A `PathConverter` implementation to control how the `Path` representation of a `Property` is obtained. By default, the `Property` definition must implement the `Path` interface to be represented as a `Path`, and in this case the `Property` itself is used as `Path` representation.
- A `PathMatcher` implementation to declare the `Path` matching strategy. By default the `Path` *relative name* is used and a standard "equals" comparison is performed to decide if two `Path` definitions match.

```
PathPropertySetAdapter pathPropertySetAdapter = PathPropertySetAdapter.builder(SET) ①
    .pathConverter(new MyPathConverter()) ②
    .pathMatcher(new MyPathMatcher()) ③
    .build();
```

- ① Use the `PathPropertySetAdapter` builder API

- ② Set a custom `PathConverter`
- ③ Set a custom `PathMatcher`

Inspect a `PropertySet` using the `Property` names

The `PathPropertySetAdapter` API can be also used to inspect a `PropertySet` using the `Property` name and allows to:

- Check if a `Property` with a given *name* is available in the property set and obtain it.
- Obtain a `Stream` of all the available property *names* in the property set.

```
final StringProperty STR = StringProperty.create("str");
final NumericProperty<Integer> ITG = NumericProperty.integerType("itg");
final PropertySet<?> SET = PropertySet.of(STR, ITG);

PathPropertySetAdapter adapter = PathPropertySetAdapter.create(SET); ①

boolean contains = adapter.contains("str"); ②
Optional<Property<?>> property = adapter.getProperty("str"); ③
Optional<Property<String>> typedProperty = adapter.getProperty("str", String.class);
④
Stream<String> paths = adapter.names(); ⑤
```

- ① Create a `PathPropertySetAdapter` from given `SET`
- ② Checks if a `Property` with given name is available in the property set
- ③ Get the `Property` with given name, if available
- ④ Get the `Property` with given name, if available, providing the expected property type
- ⑤ Obtain a `Stream` of all the available `Property` names in the property set

5.12. Managing property values using a `PropertyBox`

A `PropertyBox` represents a container of `Property` values and it is bound to a specific (and immutable) `PropertySet`. For each `Property` of the property set, the `PropertyBox` can be used to set or obtain the associated property value.

The `PropertyBox` handles the property values ensuring property and value type consistency, fully supporting the `PropertyValueConverter` to perform value conversions when suitable.

The `PropertyBox` API provides *builders* to create `PropertyBox` instances and setting the property values. The property values can be obtain from the `PropertyBox` and setted in the `PropertyBox` itself during the object lifetime. Several methods are available to inspect the `PropertyBox` definition and contents.

The `PropertyBox` API directly extends the `PropertySet` API, consequently providing its property set inspection methods.

```

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

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).set(ID, 1L).set(NAME, "testName").build
(); ⑤

Long id = propertyBox.getValue(ID); ⑥
String name = propertyBox.getValueIfPresent(NAME).orElse("default"); ⑦

boolean containsNotNullId = propertyBox.containsValue(ID); ⑧

PropertyBox ids = propertyBox.cloneBox(ID); ⑨

```

- ① Create an empty **PropertyBox** using **ID** and **NAME** properties as property set
- ② Create an empty **PropertyBox** using the **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** type 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** type in this case)
- ⑤ Create a **PropertyBox** using the fluent builder, with the **PROPERTIES** property set, setting the property values
- ⑥ 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)
- ⑦ Get the *Optional* **NAME** property value, using the **default** String value if a value for that property is not present in the **PropertyBox**
- ⑧ Check if a value for the **ID** property is present in the **PropertyBox**, i.e. the **ID** property is available in the property set and its value is not null
- ⑨ Clone the **PropertyBox**, creating a new **PropertyBox** with a property set composed only by the **ID** property

The **Property** value **validation** is enabled by default, ensuring a valid property value management by invoking any property value **Validator** when dealing with property value. The **PropertyBox** API handles the property value validation any time a property value is about to be setted in the **PropertyBox**.

This behaviour can be disabled at **PropertyBox** definition time by using the **invalidAllowed(true)** builder method. An explicit **validate()** is available to explicitly perform the validation of all the property values currently available in the **PropertyBox**.

```

final PathProperty<Long> ID = PathProperty.create("id", Long.class).withValidator(
    Validator.notNull()); ①
final StringProperty NAME = StringProperty.create("name").withValidator(Validator
    .notBlank()); ②

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

PropertyBox propertyBox = PropertyBox.create(PROPERTIES);

propertyBox.setValue(ID, null); ③

propertyBox = PropertyBox.builder(PROPERTIES).invalidAllowed(true).build(); ④

propertyBox.validate(); ⑤

```

- ① Add a *not null* validator to the **ID** property
- ② Add a *not empty* validator to the **NAME** property
- ③ Setting the **ID** property value to null will throw a **ValidationException**
- ④ Build a **PropertyBox** with disabled automatic property value validation
- ⑤ Trigger property value validation explicitly

The **PropertyBox** API fully supports **virtual properties**. When the value of a **VirtualProperty** is requested from a **PropertyBox**, the current **PropertyBox** instance is provided to the **PropertyValueProvider** which is used to supply the virtual property value.

```

final StringProperty NAME = StringProperty.create("name");
final StringProperty SURNAME = StringProperty.create("surname");

final VirtualProperty<String> FULL_NAME = VirtualProperty.create(String.class,
    propertyBox -> { ①
        return propertyBox.getValue(NAME) + " " + propertyBox.getValue(SURNAME);
    });

PropertyBox propertyBox = PropertyBox.create(NAME, SURNAME, FULL_NAME); ②

propertyBox.setValue(NAME, "John");
propertyBox.setValue(SURNAME, "Doe");

String fullName = propertyBox.getValue(FULL_NAME); ③

```

- ① Define a **VirtualProperty** to provide a *full name*, chaining the values of the **NAME** and **SURNAME** properties
- ② Add the **FULL_NAME** property to the **PropertyBox** property set
- ③ Get the **FULL_NAME** virtual property value: it will be **John Doe**

The **PropertyBox** abstraction is **the base structure to transport and provide property values** in

the Holon Platform. It is used by the Holon platform every time a set of property values comes into play.

In a data exchange scenario (for example the interaction with a persistent data model or the interchange of data between two independent services), the `PropertyBox` API allows to preserve a strong independence from the underlying concrete data structure.

In a typical data model mapping scenario, a `PropertyBox` is used to represent a persistent data *entity* value, i.e. the values of all the data model *entity* attributes (the *property set*).

5.12.1. `PropertyBox` instances identification

As a standard Java object, a `PropertyBox` instance is identified by its memory address. This makes each `PropertyBox` instance "different" from one another, in the Java objects sense, regardless of the property set and the property values contained in the `PropertyBox`.

This may not always be the desired behavior. In many situations is it appropriate to implement a `PropertyBox` identification strategy relying on its property set and the corresponding property values. According to this vision, two `PropertyBox` with the same property set and the same property values should be considered *equal*.

When dealing with a data model, a more consistent `PropertyBox` identification strategy should consider only the *identifier* property values, to check `PropertyBox` equality within the same data model *entity*.

For this reason, the `PropertySet identifier` properties are used by default to implement the `PropertyBox` identification strategy: this means that the standard `equals` and `hashCode` methods of the `PropertyBox` instance are implemented accordingly to the *identifier* property values, if available from the property set.

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

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

PropertyBox propertyBox1 = PropertyBox.builder(PROPERTIES).set(ID, 1L).set(NAME,
"name1").build();
PropertyBox propertyBox2 = PropertyBox.builder(PROPERTIES).set(ID, 1L).set(NAME,
"name2").build();

boolean isTrue = propertyBox1.equals(propertyBox2); ②
```

① Declare the `ID` property as the `PropertySet` identifier property

② Since the two `PropertyBox` instances contains the same `ID` property value (1), they will be considered equal by default

5.12.2. Custom PropertyBox instances identification

When the default `PropertyBox` identification strategy is not suitable or consistent for your needs, it can be customized just like it can be done with a [Property definition](#).

This is achieved through the `HashCodeProvider` and `EqualsHandler` interfaces. This *functional* interfaces can be used to provide a custom **hash code** and **equals** logic for the `PropertyBox` instances, to override the default Java Objects `hashCode` and `equals` implementations.

The custom `HashCodeProvider` and `EqualsHandler` implementation can be setted at `PropertyBox` definition time using the appropriate builder methods.

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

PropertyBox propertyBox = PropertyBox.builder(ID, NAME)
    .hashCodeProvider(pb -> Optional.ofNullable(pb.getValue(ID))) ①
    .equalsHandler((pb, other) -> (other instanceof PropertyBox) ②
        && ((PropertyBox) other).getValue(ID).equals(pb.getValue(ID)))
    .build();
```

① Custom `HashCodeProvider`

② Custom `EqualsHandler`

5.12.3. Using a PathPropertyBoxAdapter

Similarly to the `PathPropertySetAdapter` API, the `PathPropertyBoxAdapter` API can be used to inspect a `PropertyBox` using `Path` type expressions and allows to:

- Check if a `Property` value is present using its `Path` representation.
- Obtain a `Property` value through its `Path` representation.
- Set a `Property` value through its `Path` representation.

```
final StringProperty STR = StringProperty.create("str");
final NumericProperty<Integer> ITG = NumericProperty.integerType("itg");
final PropertySet<?> SET = PropertySet.of(STR, ITG);

final Path<String> PATH = Path.of("str", String.class);

PropertyBox box = PropertyBox.builder(SET).set(STR, "test1").set(ITG, 1).build();

PathPropertyBoxAdapter adapter = PathPropertyBoxAdapter.create(box); ①

boolean contains = adapter.containsValue(PATH); ②
Optional<String> value = adapter.getValue(PATH); ③
adapter.setValue(PATH, "value"); ④
```

① Create a `PathPropertyBoxAdapter` using given `PropertyBox` instance

- ② Checks if the value of the **Property** which corresponds to given **Path** is available
- ③ Get the value of the **Property** which corresponds to given **Path**
- ④ Set the value of the **Property** which corresponds to given **Path**

The **Path** mathing strategies can be customized in the same way as for the [PathPropertySetAdapter](#) API.

5.13. 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**.

```
final NumericProperty<Integer> ID = NumericProperty.integerType("id");

PropertyValuePresenter<Integer> presenter = getPropertyValuePresenter();

presenter.present(ID, 123); ①
```

- ① Present the value **123** for the **ID** property using a [PropertyValuePresenter](#)

The property value presenters are organized in a registry (the [PropertyValuePresenterRegistry](#)), which collects all the available presenters and provides the most suitable presenter for a given property, using the **conditions**, expressed as **Predicate**, the presenters were registered with.

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.

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.

5.13.1. 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(); ❶

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

- ❶ Create a presenter for `LocalTime` type properties which represents the value as *hours.minutes*
- ❷ Register the presenter binding it to the `Predicate` which 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.

5.13.2. Default PropertyValuePresenter

A *default* property value presenter is provided by the platform and automatically registered in the registry. The default presenter is used when no other suitable presenter is available from 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.

5.13.3. Using the Property presenters

The `Property` interface provides a convenience `present(T value)` method to present the 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.

```
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

5.14. Property rendering

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

The `PropertyRenderer` interface 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 a `Predicate`, 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 made 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.

5.14.1. 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.

5.14.2. Using the Property renderers

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

5.15. Java Beans and the `Property` model

The Holon platform offers a wide support to handle standard `Java Beans` and seamlessly integrate them with the platform `Property model`.

A Java Bean can be seen as a collection of typed *properties*, with *getter* and *setter* methods to read and write the property values. From this perspective, we can introduce the following analogies:

- A Bean property can be represented by a `Property`.
- The Bean definition (the collection of the declared properties) can be represented by a `PropertySet`.
- A Bean instance, which holds the property values, can be represented by a `PropertyBox`.

The Holon platform provides a complete API to manage Java Beans using the **property model**, making available all the essential services to switch from one model to another, i.e. to handle a Bean as a `Property` set and to manage the Bean property values through a `PropertyBox`, both to read the Bean property values and to write them.

5.15.1. Bean properties

A Java Bean property is represented by a `PathProperty`, where the property **path name** 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.

Internally, a Bean property is actually represented by a `PathProperty` extension, which is defined through the `BeanProperty` interface. A `BeanProperty` instance holds the references to the Bean property *getter* and *setter* methods, besides other property configuration attributes, to ensure consistency for the property value read and write operations.

5.15.2. Bean property set

The `BeanPropertySet` interface represents the collection of the Bean definition properties as a `PropertySet`.

The `BeanPropertySet` is obtained from a Bean class and provides the available Bean properties as `PathProperty` references.

Fully supports **nested** bean classes, allowing to access the nested bean class properties by name using the conventional **dot notation**, for example `parentProperty.nestedProperty`.

The `BeanPropertySet` API extends the default `PropertySet` API and additionally provides operations to:

- Obtain a Bean property **by name** as a `PathProperty`.
- **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 a Bean instance and obtain such values as a `PropertyBox`.
- **Write** the property values contained in a `PropertyBox` to a Bean instance.

A `BeanPropertySet` can be simply obtained from a Bean class using the `create(Class beanClass)` method.

```
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 = BeanPropertySet.create(
    MyBean.class); ①

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

    PathProperty<String> nestedName = PROPERTIES.property("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.property("id")); ⑦

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

    MyBean written = PROPERTIES.write(PropertyBox.builder(PROPERTIES).set(PROPERTIES
        .property("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

In addition to the standard **PathProperty** representation, a Bean property can be also obtained as a specific sub type, when type consistency is ensured. The supported sub types are the builtin **PathProperty** sub types made available by the Holon platform.

Each Bean property can be obtained as a specific **PathProperty** type using the appropriate methods of the **BeanPropertySet** API. When type consistency is not respected, an exception is thrown.

```
final BeanPropertySet<MyBean> PROPERTIES = BeanPropertySet.create(MyBean.class);

StringProperty stringProperty = PROPERTIES.propertyString("
aStringTypeBeanPropertyName"); ①
NumericProperty<Integer> numericProperty = PROPERTIES.propertyNumeric(
"aIntegerTypeBeanPropertyName"); ②
TemporalProperty<LocalDate> temporalProperty = PROPERTIES.propertyTemporal(
"aLocalDateTypeBeanPropertyName"); ③
BooleanProperty booleanProperty = PROPERTIES.propertyBoolean(
"aBooleanTypeBeanPropertyName"); ④
```

- ① Get the Bean property named **aStringTypeBeanPropertyName** as a **StringProperty**
- ② Get the Bean property named **aIntegerTypeBeanPropertyName** as a **NumericProperty**
- ③ Get the Bean property named **aLocalDateTypeBeanPropertyName** as a **TemporalProperty**
- ④ Get the Bean property named **aBooleanTypeBeanPropertyName** as a **BooleanProperty**

5.15.3. BeanIntrospector

A **BeanPropertySet** is built using the **BeanIntrospector** API. It provides methods to actually obtain a **BeanPropertySet** from a Bean class, introspecting it to resolve the Bean properties and their configuration.

The **BeanIntrospector** interface provides static methods to obtain a **BeanIntrospector** as a **Context** resource, if available, or retrieve the default implementation, which is always available.

```
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

Furthermore, the **BeanIntrospector** API makes available convenience methods to directly obtain a Bean instance as a **PropertyBox** and vice-versa.


```

MyBean instance = new MyBean();
instance.setId(7L);

PropertyBox value = BeanIntrospector.get().read(instance); ①

final NumericProperty<Long> ID = NumericProperty.longType("id");

BeanIntrospector.get().write(PropertyBox.builder(ID).set(ID, 8L).build(), instance);
②

```

- ① Read the given **MyBean** instance as a **PropertyBox**. The **PropertyBox** property set will be the **MyBean** class bean property set
- ② Write a **PropertyBox** into a **MyBean** instance. The **PropertyBox** property values, matched by name, will be written into the corresponding Bean instance properties

The **BeanIntrospector** API is easily **extensible** regarding the introspection strategy, especially for the Bean properties configuration. The main extension points are represented by the **BeanPropertyPostProcessor** and the **BeanPropertySetPostProcessor** interfaces, as described below.

5.15.4. BeanPropertyPostProcessor

A **BeanPropertyPostProcessor** can be used to extend the Bean introspection strategy at Bean properties level, before they will be returned as part of the final **BeanPropertySet**.

A **BeanPropertyPostProcessor** can be used for example to set **property configuration** attributes, manage **property validators**, configure **property value converters** and so on.

A **BeanPropertyPostProcessor** must be registered in the **BeanIntrospector** and it is called for every detected and valid Bean property at Bean introspection time. The **processBeanProperty** method accepts the current **BeanProperty** builder, which can be used to modify the configuration of the Property that will be part of the final **BeanPropertySet**.

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

- 1. Registration using the **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.

```

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

```

- ① Register a **BeanPropertyPostProcessor** which adds a **test** property configuration attribute to all the 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.

5.15.5. BeanPropertySetPostProcessor

A `BeanPropertySetPostProcessor` can be used to extend the Bean introspection strategy at Bean property set level, before the final `BeanPropertySet` is returned.

A `BeanPropertySetPostProcessor` can be used for example to modify the `BeanPropertySet` configuration.

A `BeanPropertySetPostProcessor` must be registered in the `BeanIntrospector` and it is called for each Bean class at Bean introspection time. The `processBeanPropertySet` method accepts the current `BeanPropertySet` builder, which can be used to modify the configuration of Bean property set.

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

1. Registration using the `BeanIntrospector`: The `addBeanPropertySetPostProcessor` method can be used to register a `BeanPropertySetPostProcessor`.

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

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

① Register a `BeanPropertySetPostProcessor` which adds a `test` property configuration attribute to the Bean property set



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

5.15.6. Builtin Bean post processors

The Holon platform makes available a set of builtin Bean post processors, automatically registered in the default `BeanIntrospector` implementation.

Most of them supports **annotations** on Bean property field which can be used to tune the Bean introspection strategy and to manage Bean properties configuration.

See below for a list of all the available annotations. All the listed annotations are located in the `com.holonplatform.core.beans` package.

@Ignore

The `@Ignore` annotation can be used on Bean property fields to **skip** the Bean property during the introspection process. The ignored property will not be part of the final [Bean property set](#).

The annotation provides a `includeNested()` attribute which can be used to set whether to ignore any *nested* Bean property (if the Bean property type is itself a Bean class) or not. Defaults to `true`, which means that if the ignored property is a Bean class type, also the properties of the nested bean class will be ignored.

```
class Bean1 {  
  
    public static final BeanPropertySet<Bean1> PROPERTIES = BeanPropertySet.create(  
        Bean1.class); ②  
  
    private Long id;  
  
    @Ignore ①  
    private String name;  
  
    // getters and setters omitted  
  
}
```

① Set the `name` property as ignored

② The Bean property set will not contain the `name` property

@Caption

The `@Caption` annotation can be used on Bean property fields to provide the property **localization** attributes, such as the property *caption* and the property *caption localization message code*.

See [the Property localization section](#) for further details.

```
class Bean2 {  
  
    public static final BeanPropertySet<Bean2> PROPERTIES = BeanPropertySet.create(  
        Bean2.class);  
  
    @Caption("Code") ①  
    private Long id;  
  
    @Caption(value = "Name", messageCode = "name.localization.code") ②  
    private String name;  
  
    // getters and setters omitted  
  
}
```

- ① Set the `id` property default caption message to `Code`. The property caption can later be read using the `getMessage()` method of the `Localizable` interface, a super interface of `Property`.
- ② Set the `name` property caption default message and localization message code

@Sequence

The `@Sequence` annotation can be used on Bean property fields to **order** the bean properties within the `Bean property set`. When used as an `Iterable`, the Bean property set will return the Bean properties ordered according to the `sequence` value declared through this annotation.

```
class Bean3 {  
  
    public static final BeanPropertySet<Bean3> PROPERTIES = BeanPropertySet.create(  
        Bean3.class);  
  
    @Sequence(10) ①  
    private Long id;  
  
    @Sequence(20) ②  
    private String name;  
  
    // getters and setters omitted  
  
}
```

- ① Set the `id` property sequence number to `10`
- ② Set the `name` property sequence number to `20`: this property will be always returned after the `id` property when the `BeanPropertySet` is used as an `Iterable`

@Config

The `@Config` annotation can be used on Bean property fields to add a `property configuration` attribute to the annotated property, specifying the **configuration** key and its value.

Only `String` type configuration values are supported by this annotation, use your own `BeanPropertyPostProcessor` to perform more advanced property configuration setup operations.

The `@Config` annotation is a *repeteable* annotation, so it can be repeated on a Bean property to provide more than one configuration attribute.

```

class Bean4 {

    public static final BeanPropertySet<Bean4> PROPERTIES = BeanPropertySet.create(
        Bean4.class);

    private Long id;

    @Config(key = "test1", value = "myValue1") ①
    @Config(key = "test2", value = "myValue2")
    private String name;

    // getters and setters omitted

}

```

① Set two property configuration attributes using the `@Config` annotation on the `name` bean property

@Converter

The `@Converter` annotation can be used on Bean property fields to configure a [property value converter](#) for a bean property.

Besides the `PropertyValueConverter` class to use, the `@Converter` annotation supports the configuration of a **builtin** property value converter through the `builtin()` annotation attribute. The builtin property value converter can be selected among the ones provided by default by the Holon platform.

```

class Bean5 {

    public static final BeanPropertySet<Bean5> PROPERTIES = BeanPropertySet.create(
        Bean5.class);

    @Converter(MyConverter.class) ①
    private Long id;

    @Converter(builtin = BUILTIN.NUMERIC_BOOLEAN) ②
    private Boolean value;

    // getters and setters omitted

}

```

① Set a custom `MyConverter` property value converter for the `id` bean property

② Set a builtin *numeric boolean* converter for the `value` bean property

Validators

The Bean property `validation` can be configured in one of the following ways:

- 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`. A Bean validation API implementation must be available in classpath.
- 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.

```
class Bean6 {

    public static final BeanPropertySet<Bean6> PROPERTIES = BeanPropertySet.create(
        Bean6.class);

    @Min(1) ①
    @Max(value = 100, message = "{my.localizable.message}") ②
    private Long id;

    @NotBlank ③
    @ValidationMessage(message = "Name must be not blank", messageCode =
        "my.message.localization.code")
    private String name;

    @Validator(MyFirstValidator.class) ④
    @Validator(MySecondValidator.class)
    private String value;

    // getters and setters omitted

}
```

- ① Set the *minimum* value for the `id` bean property using the standard `javax.validation.constraints` annotation
- ② Set the *maximum* value for the `id` bean property using the standard `javax.validation.constraints` annotation and providing the invalid value message localization code
- ③ Set the `name` bean property as *not blank* using the additional platform validation annotation. The

`@ValidationMessage` annotation is used to specify the invalid value default message and localization message code

- ④ Set two custom `Validator` implementations for the `value` bean property using the `@Validator` repeatable annotation

`@DataPath`

The `@DataPath` annotation can be used on Bean class and on Bean property fields to declare a **data path mapping** as Bean property or property set configuration attributes, using the default `PATH` configuration property of the `DataMappable` interface.

The *data path* mapping can be used to declare the data model attribute **path** when it is not the same as the Bean class name or Bean property name, and the Bean property set is used for persistence related operations.

The data path mapping must be explicitly supported by the data model handler API which will be used, that could be for example a `Datastore`.

See the specific Datastore implementation documentation to check the data path mapping support and the actual meaning it assumes.

```
@DataPath("myPath") ②
class Bean7 {

    public static final BeanPropertySet<Bean7> PROPERTIES = BeanPropertySet.create(
        Bean7.class);

    @DataPath("code") ①
    private Long id;

    private String name;

    // getters and setters omitted

}
```

- ① Set the data path mapping to `code` for the `id` bean property

- ② Set the Bean class data path mapping to `myPath`

5.15.7. BeanIntrospector cache

The default `BeanIntrospector` implementation uses an internal cache of processed bean class and property sets, to boost introspection operations and obtain better performances.

If memory consumption issues are detected, the internal cache can be disabled setting the `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.

5.16. Datastore

The **Datastore** API is the main entry point to manage data access and persistence in a technology/platform/vendor independent way.

The **Datastore** data management strategy relies on the Holon platform **property model** architecture to represent and manage data model attributes in a generic and implementation-independent way, using the **Managing property values using a PropertyBox** structure as data interchange carrier between the **Datastore** API and the concrete data model.

A concrete **Datastore** implementation could provide a more specialized API, with functionalities expressly related to the specific persistence technology/model.

We'll refer to an *entity* in this documentation as a generic persistence model data container. An *entity* may be for example a *table* in a RDBMS context, a *JPA entity class* in a JPA environment or a *document* in a document based data store.

The **Datastore** API provides the following operations:

- **Refresh:** Refresh the data of a data model *entity*, retrieving the most updated version.
- **Insert:** Insert a new data model *entity* into the persistence store.
- **Update:** Update a data model *entity* already present in the persistence store.
- **Save:** Insert or update a data model *entity*, depending on the existence of the *entity* itself in the persistence store.
- **Bulk** operations definition and execution (**bulkInsert**, **bulkUpdate** and **bulkDelete**): to execute batch operations in the persistence store.
- **Query definition and execution:** to configure and execute *queries* against the persistence store, allowing to declare query results restrictions, aggregations and sorting and to obtain the query results using different *projections*.

Each operation which involves a possible persistence store data modification returns an **OperationResult** type object, which provides information about the operation outcome, such as 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.

5.16.1. Expressions and resolvers

The **Datastore** API architecture is designed on top of the Holon platform **Expression** based architecture.

An **Expression** is a very abstract and generic representation of an element of a language. The **Datastore** API uses *expressions* to "translate" a meta-language into the actual language which can be understood by the concrete persistence context engine to which the **Datastore** is bound.

The **ExpressionResolver** interface is the key element to perform the language manipulation and translation, since it is used to *resolve* 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 expression: this way, the resolution process must proceed to the next available resolver for given expression and resolution type.

A generic `ResolutionContext` object is provided to the `ExpressionResolver` resolution method, to provide information about the current resolution context.

A set of `ExpressionResolver` can be handled using an `ExpressionResolverRegistry`. When an `Expression` must be resolved, all the available resolvers which declare to resolve the given expression type and provide a consistent resolution type will be taken into account. These resolvers are invoked sequentially, returning the first valid resolved expression, if any.

To order the expression resolvers with the same expression types, the `javax.annotation.Priority` annotation can be used on the `ExpressionResolver` implementation class to assign a priority order, where lower values corresponds to higher priority.

The `Datastore` API, extending the `ExpressionResolverSupport` interface, supports the registration of new `ExpressionResolver` instances. This is the recommended way to implement `Datastore extensions` and persistence operations **customization**.

Since a `Datastore` implementation can be bound to very different data models and persistence context, each expression resolution strategy is highly dependent from the concrete `Datastore` implementation, including a possibly specialized version of the `ResolutionContext`. See each concrete `Datastore` implementation documentation to learn about the `ExpressionResolver` support and the available expression types which can be used for extensibility purposes.

Anyway, the core meta-language expressions set is common to any `Datastore` implementation. For this reason, it is possible to provide `Datastore` API extensions in a general and implementation independent way using the `ExpressionResolver` based strategy when the expression resolution is bound to the standard meta-language expressions.

Most of the standard `Datastore` operations (for example the `Query` operation) support `ExpressionResolver` registration, to provide expression resolution manipulation only for a specific operation execution.

See the [Datastore API extensions](#) section for details.

5.16.2. Property data types

The `Datastore` API fully supports **property values conversions** using the standard [property value converter](#) API. 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, to use custom types or to face common value conversion needs, such as *enumeration* property types mapped to integer or text data model types.

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`, and

`LocalDateTime`.

When a `java.util.Date` or `java.util.Calendar` property type is used with `Datastore` API operations, it is recommended to configure the actual **temporal type** which is expected for the property, to ensure data consistency. This can be done through the [property configuration](#).

5.16.3. DataTarget

The `DataTarget` interface is used by the `Datastore` API and the [Query definition API](#) 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 specialized `DataTarget` object types to identify a data model *entity* which is specific of the persistence model to which the `Datastore` is bound.

The `DataTarget` interface provides static methods to create data targets using the default **name** representation, optionally providing the `DataTarget (Path)` type:

```
DataTarget<String> target1 = DataTarget.named("test1"); ①  
DataTarget<MyType> target2 = DataTarget.of("test2", MyType.class); ②
```

① Create a default `String` type `DataTarget` named `test1`

② Create a `MyType` type `DataTarget` named `test2`

All the `Datastore` operations definition APIs involving persistent *entity* structures use a `DataTarget` to identify the data model *entity* to which the operation refers.

5.16.4. Data manipulation operations

The `Datastore` API provides the most common data manipulation operations, listed in the table below.

Each operation throws a [DataAccessException](#) if an error occurs during the operation execution.

Each operation (except for *refresh*) support 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 each specific `Datastore` documentation for further information.

The [DefaultWriteOption](#) enumeration provides write options which can be available for any [Datastore](#) API. 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 [PropertyBox](#) property set.



Check specific [Datastore](#) implementations documentation to learn if this option is actually supported.

Operation	Purpose	Return
refresh (DataTarget target, PropertyBox propertyBox)	Refresh the values of the properties of given PropertyBox , reloading them from the persistence store, using given data target to identify the persistence model <i>entity</i> .	The refreshed PropertyBox
insert (DataTarget target, PropertyBox propertyBox, WriteOption ... options)	Insert a new data model <i>entity</i> , identified by given data target and represented by given PropertyBox , into the persistence store.	The OperationResult . If one or more data model attribute was auto-generated by the concrete persistence store, such values are returned by the getInsertedKeys method.
update (DataTarget target, PropertyBox propertyBox, WriteOption ... options)	Update an existing data model <i>entity</i> , identified by given data target and represented by given PropertyBox , in the persistence store.	The OperationResult .
save (DataTarget target, PropertyBox propertyBox, WriteOption ... options)	Insert a new data model <i>entity</i> (identified by given data target and represented by given PropertyBox) into the persistence store if the <i>entity</i> does not exists, or update it if the <i>entity</i> is already present in the persistence store.	The OperationResult .
delete (DataTarget target, PropertyBox propertyBox, WriteOption ... options)	Remove a data model <i>entity</i> , identified by given data target and represented by given PropertyBox , from the persistence store.	The OperationResult .

Operation	Purpose	Return
<code>bulkInsert(DataTarget target, PropertySet<?> propertySet, WriteOption... options)</code>	Configure and perform a <i>bulk insert</i> of data model <i>entities</i> identified by given data target and represented by <code>PropertyBox</code> instances. Only the properties contained in the given <code>PropertySet</code> will be taken into account to perform the insert operations.	The <code>BulkInsert</code> API to configure and perform the bulk operation.
<code>bulkUpdate(DataTarget target, WriteOption... options)</code>	Configure and perform a <i>bulk update</i> of data model <i>entities</i> identified by given data target , to change a set of property values according to a set of restriction predicates to identify the set of data model <i>entities</i> to update.	The <code>BulkUpdate</code> API to configure and perform the bulk operation.
<code>bulkDelete(DataTarget target, WriteOption... options)</code>	Configure and perform a <i>bulk delete</i> of data model <i>entities</i> identified by given data target , providing a set of restriction predicates to identify the set of data model <i>entities</i> to remove.	The <code>BulkDelete</code> API to configure and perform the bulk operation.

```
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 specified **DataTarget** (insert a new *entity* if not present in the persistence store or update it if exists)
- ② Insert the given **PropertyBox** data into the persistence store using the specified **DataTarget**
- ③ Update the given **PropertyBox** data into the persistence store using the specified **DataTarget**
- ④ Refresh the **PropertyBox** property values using the specified **DataTarget**
- ⑤ Remove the *entity* which corresponds to given **PropertyBox**
- ⑥ Execute a *bulk* insert operation using the specified **DataTarget**, inserting given **PropertyBox** elements
- ⑦ Execute a *bulk* update operation using the specified **DataTarget**, setting the property value to **updated** when the property value is **null**
- ⑧ Execute a *bulk* delete operation using the specified **DataTarget**, removing entities for which the given property value is **null**

5.17. Query

The [Query](#) API can be used to configure and execute *queries* against the persistence data store.

Just like any other [Datastore](#) API operation, the [Query](#) API relies on the Holon platform [property model](#) to represent the data model attributes and to obtain the query results, using a [PropertyBox](#) to provide a set of property values.

This allows the query to be declared and executed in an abstract and implementation-independent way.

The [Query](#) API supports the following clauses and configuration attributes:

- The [DataTarget](#) on which the query has to be performed.
- A set of query **restrictions**, expressed as [QueryFilter](#) clauses.
- The query results **sorting** declarations, expressed as [QuerySort](#) clauses.
- The query results **aggregation**, expressed as [QueryAggregation](#) clauses.
- The query results **paging**, to configure the query result set *limit* and *offset*.
- A set of generic query configuration **parameters**.

Below are described the standard query [Expressions](#) made available by the Holon platform for the definition of a query.

5.17.1. QueryFilter

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

A [QueryFilter](#) acts on other *expressions* which represent the restriction subject and conditions. Such expressions are typically a [TypedExpression](#), i.e an [Expression](#) with explicitly declared type.

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

- **Is null / is not null:** An expression is *null* / not *null*.
- **equal / not equal:** An expression is equal / not equal to another expression.
- **less than / less than or equal:** An expression is less than / less than or equal to another expression.
- **greater than / greater than or equal:** An expression is greater than / greater than or equal to another expression.
- **between:** The value of an expression is included between a minimum and a maximum value.
- **in / not in:** The value of an expression is included / not included in a set of values.

For [String](#) type expressions:

- **contains:** The value of a [String](#) type expression contains a specified text (ignoring case or not).
- **startsWirth:** The value of a [String](#) type expression contains starts with a specified text

(ignoring case or not).

- **endsWith:** The value of a `String` type expression contains ends with a specified text (ignoring case or not).

Furthermore, the `QueryFilter` predicates can be composed using logical operations:

- **not:** Negation of a `QueryFilter` predicate.
- **and:** Conjunction of `QueryFilter` predicates.
- **or:** Disjunction of `QueryFilter` predicates.

The `QueryFilter` predicates can be obtained in two ways:

1. Using the static builder methods provided by the `QueryFilter` interface.

```
final PathProperty<Integer> PROPERTY1 = PathProperty.create("test1", Integer.class);
final PathProperty<Integer> PROPERTY2 = PathProperty.create("test2", Integer.class);
final StringProperty STRING_PROPERTY = StringProperty.create("test3");
```

①

```
QueryFilter restriction = QueryFilter.isNotNull(PROPERTY1); // is not null
restriction = QueryFilter.isNull(PROPERTY1); // is null
restriction = QueryFilter.eq(PROPERTY1, 7); // equal to a value
restriction = QueryFilter.eq(PROPERTY1, PROPERTY2); // equal to another property
expression
restriction = QueryFilter.neq(PROPERTY1, 7); // not equal
restriction = QueryFilter.lt(PROPERTY1, 7); // less than
restriction = QueryFilter.loe(PROPERTY1, 7); // less than or equal
restriction = QueryFilter.gt(PROPERTY1, 7); // greater than
restriction = QueryFilter.goe(PROPERTY1, 7); // greater than or equal
restriction = QueryFilter.between(PROPERTY1, 1, 7); // between
restriction = QueryFilter.in(PROPERTY1, 1, 2, 3); // in
restriction = QueryFilter.nin(PROPERTY1, 1, 2, 3); // not in
```

②

```
restriction = QueryFilter.startsWith(STRING_PROPERTY, "V", false); // starts with 'v'
restriction = QueryFilter.startsWith(STRING_PROPERTY, "v", true); // starts with 'v',
ignoring case
restriction = QueryFilter.endsWith(STRING_PROPERTY, "v", false); // ends with 'v'
restriction = QueryFilter.contains(STRING_PROPERTY, "v", false); // contains 'v'
QueryFilter restriction2 = QueryFilter.contains(STRING_PROPERTY, "v", true); //
contains 'v', ignoring case
```

// negation ③

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

// conjunction ④

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

// disjunction ⑤

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

- ① Common restriction predicates using a **PathProperty** as expression
- ② **String** type expression restriction predicates
- ③ Negation using either the **not()** **QueryFilter** method or the **not(QueryFilter filter)** builder method
- ④ Conjunction (**AND**) using either the **and()** **QueryFilter** method or the **allOf(QueryFilter... filters)** builder method
- ⑤ Disjunction (**OR**) using either the **or()** **QueryFilter** method or the **anyOf(QueryFilter filter)** builder method

2. Using the convenience methods provided by the `QueryExpression` interface. The `QueryExpression` API is implemented, for example, by the `PathProperty` interface.

The `QueryExpression` API makes available a set of methods to create a `QueryFilter` using the expression itself as the subject of the restriction predicate.

For restrictions which refers to a specific data type, specialized `QueryExpression` extensions should be used. For example, `StringQueryExpression` interface makes available methods to obtain String type restrictions, such a *contains*, *startsWith*, *endsWith*. The type specific query expression APIs are implemented by each `PathProperty` sub type: the `StringQueryExpression` API is implemented by the `StringProperty` type and so on.

```
final PathProperty<Integer> PROPERTY1 = PathProperty.create("test1", Integer.class);
final PathProperty<Integer> PROPERTY2 = PathProperty.create("test2", Integer.class);
final StringProperty STRING_PROPERTY = StringProperty.create("test3");
```

①

```
QueryFilter restriction = PROPERTY1.isNotNull(); // is not null
restriction = PROPERTY1.isNull(); // is null
restriction = PROPERTY1.eq(7); // equal to a value
restriction = PROPERTY1.eq(PROPERTY2); // equal to another property
restriction = PROPERTY1.neq(7); // not equal
restriction = PROPERTY1.lt(7); // less than
restriction = PROPERTY1.loe(7); // less than or equal
restriction = PROPERTY1.gt(7); // greater than
restriction = PROPERTY1.goe(7); // greater than or equal
restriction = PROPERTY1.between(1, 7); // between
restriction = PROPERTY1.in(1, 2, 3); // in
restriction = PROPERTY1.nin(1, 2, 3); // not in
```

②

```
restriction = STRING_PROPERTY.startsWith("v"); // starts with
restriction = STRING_PROPERTY.startsWithIgnoreCase("v"); // starts with ignoring case
restriction = STRING_PROPERTY.endsWith("v"); // ends with
restriction = STRING_PROPERTY.endsWithIgnoreCase("v"); // ends with ignoring case
restriction = STRING_PROPERTY.contains("v"); // contains
QueryFilter restriction2 = STRING_PROPERTY.containsIgnoreCase("v"); // contains
ignoring case
```

③

```
QueryFilter negation = PROPERTY1.eq(7).not(); // negation
QueryFilter conjunction = PROPERTY1.isNotNull().and(PROPERTY2.eq(3)); // conjunction
QueryFilter disjunction = PROPERTY1.isNull().or(PROPERTY2.eq(3)); // disjunction
```

- ① Common restriction predicates using the `QueryExpression` API implemented by `PathProperty`
- ② `String` type restriction predicates using the `StringQueryExpression` API implemented by `StringProperty`
- ③ Logical operations

5.17.2. QuerySort

The [QuerySort](#) interface represents a query results sorting directive.

A [QuerySort](#) acts on a generic [Path](#) expression to declare the sorting subject and uses the [SortDirection](#) enumeration to declare the sort direction (ascending or descending). Query sorts can be composed to declare an ordered list of sort declarations.

A [QuerySort](#) declaration can be obtained in two ways:

1. Using the static builder methods provided by the [QuerySort](#) interface.

The [PathExpression](#) API makes available convenience methods to create a [QuerySort](#) using the expression itself (i.e. the [Path](#) represented by the expression) as the subject of the sort declaration.

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

①

```
QuerySort sort = QuerySort.of(PROPERTY, SortDirection.ASCENDING); // sort ASCENDING on
given property path
sort = QuerySort.of(PROPERTY, true); // sort ASCENDING on given property path
sort = QuerySort.asc(PROPERTY); // sort ASCENDING on given property path
QuerySort sort2 = QuerySort.desc(ANOTHER_PROPERTY); // sort DESCENDING on given
property path
```

②

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

① Sort declarations specifying the sort direction

② Query sorts composition

2. Using the convenience methods provided by the [PathExpression](#) interface. The [PathExpression](#) API is implemented, for example, by the [PathProperty](#) interface.

The [PathExpression](#) API makes available convenience methods to create a [QuerySort](#) using the expression itself (i.e. the [Path](#) represented by the expression) as the subject of the sort declaration.

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

①

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

②

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

① Use the `asc()` and `desc()` `PathExpression` methods to create ascending and descending sorts

② The `and(QuerySort sort)` method of the `QuerySort` interface can be used to compose a list of sorts

5.17.3. QueryFunction

The `QueryFunction` interface represents a **function** expression.

A `QueryFunction` may accept a list of *arguments*, expressed as expressions themselves. The `TypedExpression` type is used as query function arguments type, since a function could only be applicable to a specific argument type.

A set of common query functions is provided by the Holon platform, each represented by a specific `QueryFunction` sub type.

Common query functions can be obtained either using the `QueryFunction` static builder methods or using the convenience methods provided by APIs like `QueryExpression`. In this second case, the expression itself is used as the query function argument.

Aggregation functions

- **Count:** Aggregation function to *count* the number of available elements (for example the number of query results). Represented by the `Count` interface. Always returns a `Long` type result.
- **Min:** Aggregation function to obtain the *smallest value* within a set of available elements. Represented by the `Min` interface.
- **Max:** Aggregation function to obtain the *largest value* within a set of available elements. Represented by the `Max` interface.
- **Avg:** Aggregation function to obtain the *average value* within a set of numeric elements. Represented by the `Avg` interface. Always returns a `Double` type result.
- **Sum:** Aggregation function to sum the values of a set of available elements. Represented by the `Sum` interface.

For aggregation functions which refers to a specific data type, specialized `QueryExpression` extensions should be used. For example, the `NumericQueryExpression` interface makes available methods to obtain **numeric** type restrictions, such a *avg* and *sum*. The type specific query

expression APIs are implemented by each [PathProperty sub type](#): the [NumericQueryExpression](#) API is implemented by the [NumericProperty](#) type and so on.

```
final NumericProperty<Integer> PROPERTY = NumericProperty.integerType("test");
```

①

```
Count count = QueryFunction.count(PROPERTY);
Min<Integer> min = QueryFunction.min(PROPERTY);
Max<Integer> max = QueryFunction.max(PROPERTY);
Avg avg = QueryFunction.avg(PROPERTY);
Sum<Integer> sum = QueryFunction.sum(PROPERTY);
```

②

```
count = Count.create(PROPERTY);
min = Min.create(PROPERTY);
max = Max.create(PROPERTY);
avg = Avg.create(PROPERTY);
sum = Sum.create(PROPERTY);
```

③

```
count = PROPERTY.count();
min = PROPERTY.min();
max = PROPERTY.max();
avg = PROPERTY.avg();
sum = PROPERTY.sum();
```

- ① Aggregation functions created using the [QueryFunction](#) builder methods
- ② Aggregation functions created using each function interface creation method
- ③ Aggregation functions obtained using the [QueryExpression](#) convenience methods

String related functions

For [String](#) type expressions, two standard query functions are made available by the Holon platform:

- **Lower:** Function to convert a string to *lowercase*. Represented by the [Lower](#) interface.
- **Upper:** Function to convert a string to *uppercase*. Represented by the [Upper](#) interface.

```
final StringProperty PROPERTY = StringProperty.create("test");
```

①

```
Lower lower = QueryFunction.lower(PROPERTY);  
Upper upper = QueryFunction.upper(PROPERTY);
```

②

```
lower = Lower.create(PROPERTY);  
upper = Upper.create(PROPERTY);
```

③

```
lower = PROPERTY.lower();  
upper = PROPERTY.upper();
```

- ① Lower/upper functions created using the **QueryFunction** builder methods
- ② Aggregation functions created using each function interface creation method
- ③ Lower/upper functions obtained using the **StringQueryExpression** convenience methods

Temporal functions

For temporal data types, a set of common functions are made available by the Holon platform.

Functions to obtain the **current date or timestamp**:

- **CurrentDate**: function to obtain the current date. Represented by the **CurrentDate** interface.
- **CurrentLocalDate**: function to obtain the current date as a **LocalDate**. Represented by the **CurrentLocalDate** interface.
- **CurrentTimestamp**: function to obtain the current timestamp. Represented by the **CurrentTimestamp** interface.
- **CurrentLocalDateTime**: function to obtain the current timestamp as a **LocalDateTime**. Represented by the **CurrentLocalDateTime** interface.

①

```
CurrentDate currentDate = QueryFunction.currentDate();  
CurrentLocalDate currentLocalDate = QueryFunction.currentLocalDate();  
CurrentTimestamp currentTimestamp = QueryFunction.currentTimestamp();  
CurrentLocalDateTime currentLocalDateTime = QueryFunction.currentLocalDateTime();
```

②

```
currentDate = CurrentDate.create();  
currentLocalDate = CurrentLocalDate.create();  
currentTimestamp = CurrentTimestamp.create();  
currentLocalDateTime = CurrentLocalDateTime.create();
```

- ① Current date/time functions created using the **QueryFunction** builder methods
- ② Current date/time functions created using each function interface creation method

Functions to **extract a temporal part**. All the listed functions returns an **Integer** type result:

- **Year**: function extract the *year* part of a temporal data type. Represented by the **Year** interface.
- **Month**: function extract the *month* part of a temporal data type. Represented by the **Month** interface. The month range index is between 1 and 12.
- **Day**: function extract the *day* part of a temporal data type. Represented by the **Day** interface. The day is intended as the day of month and the day range index is between 1 and 31.
- **Hour**: function extract the *hour* part of a temporal data type. Represented by the **Hour** interface. The 24-hour clock is used and the hour range index is between 0 and 23.

```
final TemporalProperty<LocalDateTime> PROPERTY = TemporalProperty.localDateTime("test");
```

①

```
Year year = QueryFunction.year(PROPERTY);
Month month = QueryFunction.month(PROPERTY);
Day day = QueryFunction.day(PROPERTY);
Hour hour = QueryFunction.hour(PROPERTY);
```

②

```
year = Year.create(PROPERTY);
month = Month.create(PROPERTY);
day = Day.create(PROPERTY);
hour = Hour.create(PROPERTY);
```

③

```
year = PROPERTY.year();
month = PROPERTY.month();
day = PROPERTY.day();
hour = PROPERTY.hour();
```

- ① Temporal part extraction functions created using the **QueryFunction** builder methods
- ② Temporal part extraction functions created using each function interface creation method
- ③ Temporal part extraction functions obtained using the **TemporalQueryExpression** convenience methods

5.17.4. QueryAggregation

The **QueryAggregation** interface represents a query results *aggregation* expression.

The **QueryAggregation** API allows to specify:

- The **paths** (using the **Path** type) to be used to aggregate the query results, i.e. to group the results by the values of the specified paths.
- The **optional restrictions** to apply on the aggregation path values, expressed by using **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.

The **QueryAggregation** API provides a buider to create and configure a query aggregation expression.

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

QueryAggregation aggregation = QueryAggregation.builder() ①
    .path(PROPERTY) ②
    .path(ANOTHER_PROPERTY) ③
    .filter(PROPERTY.isNotNull()) ④
    .build();
```

- ① Obtain a **QueryAggregation** builder
- ② Declare an aggregation path using the **PROPERTY** PathProperty
- ③ Add another aggregation path using the **ANOTHER_PROPERTY** PathProperty
- ④ Configure an aggregation restrinction filter

5.17.5. Query definition

A **Query** can be defined and configured using the **QueryBuilder** interface.

The **QueryBuilder** provides methods to configure the query using the query **expressions** listed above. Furthermore, it provides methods to declare:

- The query **target**, which represents the data model *entity* to be queried and is expressed through a **DataTarget**.
- The optional query results **pagination** declaration, which can be declared using:
 - **limit::** the query results limit, i.e. the max number of results to obtain.
 - **offset::** the 0-based offset from which to fetch the query results within the total results set.
- Optional query **parameters**, mainly used to for extension purposes.

The **Query** API extends **QueryBuilder**, and can be obtained from a **Datastore** using the **query()** method.

```

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

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

Query query = datastore.query() ①
    .target(DataTarget.named("testTarget")) ②
    .filter(PROPERTY.gt(10)) ③
    .sort(PROPERTY.asc()) ④
    .aggregate(PROPERTY) ⑤
    .limit(100) ⑥
    .offset(200); ⑦

query = datastore.query(DataTarget.named("testTarget")) ⑧
    .aggregate(QueryAggregation.builder().path(PROPERTY).filter(PROPERTY.gt(10)).
build()) ⑨
    .restrict(100, 200); ⑩

```

- ① Obtain a query builder
- ② Set the query target using `DataTarget`
- ③ Add a query restriction filter using `QueryFilter`
- ④ Add a query sort declaration using `QuerySort`
- ⑤ Declare a query result aggregation path
- ⑥ Set the query results limit to `100`
- ⑦ Set the query results offset to `200`
- ⑧ Obtain a query builder and simultaneously set the query target
- ⑨ Declare a query results aggregation clause using the `QueryAggregation` builder
- ⑩ Set the the query pagination using the convenience `restrict` method, which accepts the query results limit and offset

5.17.6. Query projection and execution

To obtain the `Query` results, a query results *projection* must be declared. A projection is represented by the `QueryProjection` interface and is used to declare which data model attribute values are to be returned and which type has to be used to represent the query results.

The query results projection is provided at query execution time, using the `QueryResults` API, which is implemented by the `Query` interface.

The main `QueryResults` API method to obtain the query results is `stream(QueryProjection<R> projection)`, which can be used to provide a `QueryProjection` and get the query execution result, which will be of the same type of the query projection type.

In addition to the default *stream* method, a set of other convenience methods are provided by the `QueryResults` API for query execution and results retrieval:

- `list(QueryProjection<R> projection)`: To obtain the query results stream as a `List`.

- `findOne(QueryProjection<R> projection)`: To obtain a result which is expected to be **unique**, if it is available. The query result is provided as an **Optional** and if more than one result is obtained from query execution, a **QueryNonUniqueResultException** is thrown.
- `count()`: To count all the query results, returning the number of results as a **long**.

Furthermore, a set of convenience methods are provided to use the Holon platform **properties** abstraction as query projection and to obtain the query results using the **PropertyBox** type:

- `stream(Iterable<P> properties)` and `stream(Property... properties)`: Allows to provide one or more **Property** as query projection and obtain the query results as a stream of **PropertyBox**. The **list** version is also available to obtain the query results as a **List** instead of a **Stream**.
- `findOne(Iterable<P> properties)` and `findOne(Property... properties)`: Allows to provide one or more **Property** as query projection and obtain a query result which is expected to be **unique** as a **PropertyBox**. The query result is provided as an **Optional** and if more than one result is obtained from query execution, a **QueryNonUniqueResultException** is thrown.

```
final NumericProperty<Integer> PROPERTY1 = NumericProperty.integerType("test1");
final StringProperty PROPERTY2 = StringProperty.create("test2");

final PropertySet<?> PROPERTIES = PropertySet.of(PROPERTY1, PROPERTY2);

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

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

long count = datastore.query().target(TARGET).count(); ①

Stream<Integer> values = datastore.query(TARGET).stream(PROPERTY1); ②
Optional<Integer> value = datastore.query(TARGET).findOne(PROPERTY1); ③

Stream<PropertyBox> results = datastore.query(TARGET).stream(PROPERTY1, PROPERTY2); ④
results = datastore.query(TARGET).stream(PROPERTIES); ⑤
List<PropertyBox> list = datastore.query(TARGET).list(PROPERTY1, PROPERTY2); ⑥

Optional<PropertyBox> result = datastore.query(TARGET).findOne(PROPERTY1, PROPERTY2);
⑦
```

- ① Count the query results
- ② Use the **PROPERTY1** as query projection and obtain the query results as a **Stream** of values of the property value type (**Integer**)
- ③ Use the **PROPERTY1** as query projection expecting a unique result, obtaining the query result as an **Optional** value of the property value type (**Integer**)
- ④ When more than one **Property** is provided as query projection, the query results are obtained as a **Stream** of **PropertyBox** instances
- ⑤ A **PropertySet** can be used to provide a multiple **Property** query projection

- ⑥ The same operation can be performed with the `list` method, obtaining the query results as a `List`
- ⑦ `PROPERTY1` and `PROPERTY2` are provided as query projection and a unique result is expected: the query result is obtained as an `Optional PropertyBox` instance

Builtin query projections

The Holon platform core module provides some builtin `QueryProjection` types which can be used for query execution.

As seen in the previous section, a `PathProperty` is a `QueryProjection` itself, and can be directly used as query projection. When the query projection must include more than one property, the `PropertySetProjection` type can be used. Anyway, is easier to use the appropriate `QueryResults` API methods to directly provide a set of properties or a `PropertySet` as query projection.

The other builtin query projections are:

1. QueryFunction:

A `QueryFunction` can be directly used as query projection. The projection result type will be the same as the `QueryFunction` result type.

```
final NumericProperty<Integer> PROPERTY1 = NumericProperty.integerType("test");
final StringProperty PROPERTY2 = StringProperty.create("test2");

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

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

Optional<Integer> sum = datastore.query(TARGET).findOne(PROPERTY1.sum()); ①

Stream<String> results = datastore.query(TARGET).stream(PROPERTY2.upper()); ②
```

- ① Use the `Sum` function on `PROPERTY1` as query projection
- ② Use the `Upper` function on `PROPERTY2` to obtain a stream of `String` values applying the uppercase transformation

2. Constant expression:

The `ConstantExpressionProjection` type can be used to declare a **constant expression value** as query projection.

```
final NumericProperty<Integer> PROPERTY1 = NumericProperty.integerType("test");
final StringProperty PROPERTY2 = StringProperty.create("test2");

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

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

Optional<Integer> sum = datastore.query(TARGET).findOne(PROPERTY1.sum()); ①

Stream<String> results = datastore.query(TARGET).stream(PROPERTY2.upper()); ②
```

① Use the constant **TEST** value as query projection

3. Bean projection:

The [BeanProjection](#) interface can be used to obtain the query results as Java Bean class instances, providing the **bean class** to be used.

Optionally, the projection **Path** names can be specified to control the bean property set which as to be obtained as query result. If not specified, all the bean definition class properties will be used as query projection paths.

```

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

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

    Stream<MyBean> results = datastore.query(TARGET).stream(BeanProjection.of(MyBean
.class)); ①
    Optional<MyBean> result = datastore.query(TARGET).findOne(BeanProjection.of(MyBean
.class)); ②

    final BeanPropertySet<MyBean> PROPERTIES = BeanPropertySet.create(MyBean.class);

    results = datastore.query(TARGET).stream(BeanProjection.of(MyBean.class, PROPERTIES
.property("code"))); ③
}

```

- ① Use **MyBean** class as query projection and obtain a **Stream** of **MyBean** instances as query results
- ② The same operation but expecting a unique result, which will be returned as an **Optional MyBean** instance
- ③ Use **MyBean** class as query projection and specify the query projection paths. In this example, only the **code** bean property is declared as projection paths, so only the **code** property values will be retrieved from query execution and setted in the **MyBean** result instances

3. Select all projection:

The [SelectAllProjection](#) interface can be used to obtain all the values of a persistent data entity instance as a [Map](#) with the entity attribute *names* as keys and the corresponding entity attribute *values* as values.

The concrete result can be highly dependent on the specific [Datastore](#) implementation, both regarding the entity attribute names and the entity attribute values representations.

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

List<Map<String, Object>> values = datastore.query(DataTarget.named("test")).list
(SelectAllProjection.create()); ①
```

- ① Obtain all the values of the data entity represented by the `test` name as a [Map](#) with the entity attribute names and their values

5.17.7. Distinct query projection results

The `distinct()` query builder method can be used to obtain *distinct* query projection result values.

```
final StringProperty PROPERTY = StringProperty.create("test");

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

Stream<String> results = datastore.query(DataTarget.named("test")) //
    .distinct() ①
    .stream(PROPERTY);
```

- ① Configure the query to obtain distinct query results

5.17.8. Configuration

The [DatastoreConfigProperties](#) interface represents and provides the available configuration properties which can be used to configure a generic [Datastore](#) instance.

The interface extends the default [ConfigPropertySet](#) API, 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

Name	Type	Meaning
<i>holon.datastore. dialect</i>	String	The fully qualified class name of the <i>dialect</i> to be used by the Datastore . The <i>dialect</i> semantics and the available implementations are specific for each Datastore implementation. See each specific Datastore implementation documentation for information.

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 Datastores 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();
```

5.17.9. Relational Datastores

When a **Datastore** implementation refers to a **relational** persistence data model, some additional expressions are provided to use 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 methods of the **SubQuery** interface can be used.

Since **SubQuery** extends **QueryBuilder**, a sub query can be configured (setting the query target, restrictions, sorting and so on) the same way as a standard **Query**.

The **SubQuery** projection is provided to the **create(...)** builder methods at sub query definition time.

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().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, since the **1** literal value is used by default as projection.

```

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().target(TARGET2).filter(PROPERTY2.eq(PROPERTY1)).exists(
)).stream(PROPERTY2); ①

results = datastore.query().target(TARGET1)
    .filter(SubQuery.create().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 declare and **alias** and to configure **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**);

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

5.17.10. Transactional Datastores

If a **Datastore** implementation supports **transactions**, the **Transactional** API can be used to manage the transactions at a higher level, in an abstract and implementation-independent way.

The **Transactional** API makes available method to execute a **Datastore** operation within a transaction, taking care of the transaction lifecycle.

The **TransactionalOperation** functional interface has to be used to perform actual operation execution and to handle the current transaction, represented by a **Transaction** reference, for example to perform transaction **commit** or **rollback**.

The **Datastore** API provides a **isTransactional()** method which can be used to check if the concrete **Datastore** implementation supports transaction and to obtain it as a **Transactional** API reference. The **requireTransactional()** method has the same meaning, but throws an exception if the concrete **Datastore** implementation does not support transactions.

```

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

datastore.isTransactional().ifPresent(transactional -> { ①
    OperationResult result = transactional.withTransaction(tx -> { ②
        OperationResult r = datastore.insert(TARGET,
            PropertyBox.builder(A_PROPERTY).set(A_PROPERTY, "test").build()); ③
        tx.commit(); ④
        return r;
    });
});

```

- ① Check if **Datastore** is transactional: if so, obtain the **Transactional** API reference
- ② Execute on operation within a transaction and return a **OperationResult** type result
- ③ Execute the actual operation
- ④ Commit the transaction

When a return value is not needed, the **TransactionalInvocation** interface can be used instead of the standard **TransactionalOperation** one.

```

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

datastore.requireTransactional() ①
    .withTransaction(tx -> { ②
        datastore.insert(TARGET, PropertyBox.builder(A_PROPERTY).set(A_PROPERTY, "test"
).build()); ③
        tx.commit(); ④
    });

```

- ① Require the **Datastore** to be transactional and obtain the **Transactional** API reference
- ② Execute on operation within a transaction which do not returns any result
- ③ Execute the actual operation
- ④ Commit the transaction

The **Transaction** interface allows also to set the transaction as **rollback only**, so that the only possible outcome of the transaction is for the transaction to be rolled back.

```
getDastore().requireTransactional().withTransaction(tx -> {  
    tx.setRollbackOnly(); ❶  
});
```

❶ Set the transaction as rollback only

The [TransactionConfiguration](#) API can be used to **configure the transaction**.

The transaction configuration options are:

- Set the **auto-commit** mode: whether the transaction must be committed when a transactional operation ends and no error occurred. Default is **false**.
- Set the **rollback on error** mode: whether the transaction must be rolled back when an exception is thrown during a transactional operation execution. Default is **true**.
- Configure the **transaction options**, if supported by the concrete **Datastore** implementation

```
getDastore().requireTransactional().withTransaction(tx -> {  
    // ...  
}, TransactionConfiguration.withAutoCommit()); ❶
```

❶ Configure the transaction enabling the auto-commit mode

```
getDastore().requireTransactional().withTransaction(tx -> {  
    // ...  
}, TransactionConfiguration.create(false, false)); ❶
```

❶ Configure the transaction disabling the *rollback on error* behaviour and the auto-commit mode

5.17.11. Datastore API extensions

Datastores provides two main entry points for extension purposes:

- Use the **Expression** based architecture, through the [ExpressionResolver](#) interface, to provide custom expressions and the expression resolution logic which is required to *resolve* such expressions in a form that the **Datastore** is able to understand.
- Provide additional **Datastore** operations and functionalities relying on the [DatastoreCommodity](#) concept, through the registration of a [DatastoreCommodityFactory](#).

Extend the Datastore API using **ExpressionResolver**

The **Datastore** API supports **ExpressionResolver** registration, to add new expression resolution strategies and to handle new **Expression** types.

In a general sense, any new expression type should be resolved in an expression type that the **Datastore** is able to understand.

Each concrete **Datastore** implementation could provide additional expression types and specific

expression resolution capabilities. See each [Datastore](#) implementation documentation to learn about any additional extension capability which could be provide by a specific [Datastore](#) implementation.

```
final static PathProperty<String> SOME_PROPERTY = PathProperty.create("test", String.class);

class MyExpression implements QuerySort { ①

    @Override
    public void validate() throws InvalidExpressionException {
    }

}

public void resolver() {

    ExpressionResolver<MyExpression, QuerySort> resolver = ExpressionResolver.create
(MyExpression.class, ②
    QuerySort.class, (expression, context) -> {
        return Optional.of(QuerySort.asc(SOME_PROPERTY));
    });

    Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
implementation
    datastore.addExpressionResolver(resolver); ③

    datastore.query().target(DataTarget.named("test")).sort(new MyExpression()).stream
(SOME_PROPERTY); ④
}
```

- ① Create a custom [QuerySort](#) expression class
- ② Create an [ExpressionResolver](#) to resolve the [MyExpression](#) type into a standard [QuerySort](#) type
- ③ Register the [ExpressionResolver](#) in the [Datastore](#)
- ④ Use the [MyExpression](#) type as any other [QuerySort](#) expression

For the most common query expressions, convenience [ExpressionResolver](#) types are provided to easily create expression resolution extensions. See below for the available core [ExpressionResolver](#) extensions.

DataTargetResolver

The [DataTargetResolver](#) is a convenience [ExpressionResolver](#) extension to resolve [DataTarget](#) type expressions.

Typically, a [DataTargetResolver](#) can be defined to resolve a [DataTarget](#) with a symbolic name into a specific Datastore data target.

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

- ① Create a resolver which translates the symbolic `test` data target name into another named target with the `wellKnownTargetName` name

QueryFilterResolver

The `QueryFilterResolver` interface is a convenience `ExpressionResolver` extension to resolve `QueryFilter` type expressions.

A typical custom `QueryFilter` expression 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`.
3. Finally, register the `QueryFilterResolver` in the `Datastore` instance, using the `addExpressionResolver(...)` method.

When the resolver is registered, the custom filter can be used as any another `QueryFilter` implementation.

```

class MyFilter implements QueryFilter { ①

    final StringProperty property;
    final String value;

    public MyFilter(StringProperty 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> { ②

    @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))); ③
    }
}

final static StringProperty PROPERTY = StringProperty.create("testProperty");

public void customFilter() {
    Datastore datastore = getDatastore(); // build or obtain a concrete Datastore
implementation
    datastore.addExpressionResolver(new MyFilterResolver()); ④

    Stream<String> results = datastore.query().target(DataTarget.named("test"))
        .filter(PROPERTY.isNotNull().and(new MyFilter(PROPERTY, "testValue"))).stream
(PROPERTY); ⑤
}

```

① Custom filter definition, implementing `QueryFilter`

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

QuerySortResolver

The `QuerySortResolver` interface is a convenience `ExpressionResolver` extension to resolve `QuerySort` type expressions.

A typical custom `QuerySort` expression 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.
3. Finally, register the `QuerySortResolver` in the `Datastore` instance, using the `addExpressionResolver(...)` method.

When the resolver is registered, the custom sort can be used as any another `QuerySort` implementation.


```

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.



Concrete `Datastore` implementations may provide other methods to register a commodity. See each specific `Datastore` implementation documentation for details.

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.



Concrete `Datastore` implementations may offer specific `DatastoreCommodityContext` extensions to provide specific `Datastore` context references and configuration attributes.

A `DatastoreCommodity` can be obtained from a `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.

See each concrete `Datastore` implementations documentation for further details and examples.

5.17.12. 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.
- **MongoDB Datastore**: the **MongoDB Datastore** implementation, providing synchronous, asynchronous and reactive programming models.

5.18. DataMappable

The `DataMappable` interface can be used to define and provide a **data mapping** for a data model related object, declaring the actual **data attribute path** to which it refers.

The **data mapping** declaration can also be used when the *path* represented by a data model related object does not match the actual data model path name, to provide the real data model path name itself.

The `DataMappable` provides the data path mapping, if available, through the method:

```
Optional<String> getDataPath();
```

5.18.1. Data mapping declaration

Some Holon Platform APIs directly extends the `DataMappable` API and provides *builder* methods to set the data path mapping. Two of these are `Path` and `PathProperty`.

```
PathProperty<Integer> property = PathProperty.create("name", Integer.class) //  
    .dataPath("mapping-name"); ①  
  
Optional<String> mapping = property.getDataPath(); ②
```

① Set the actual data path for the property

② Obtain the data path, if available

As a general convention, the data path mapping is configured in the objects which support a *configuration* using the `DataMappable.PATH` configuration property, and this property can be used as an alternative for the `DataMappable` API method when the object does not directly implement that interface.

For example, to set the data path for a `PropertySet` type object the `DataMappable.PATH` configuration property can be used in this way:

```
PropertySet<?> PROPERTIES = PropertySet.builderOf(P1, P2) //  
    .withConfiguration(DataMappable.PATH, "mapping-name") ①  
    .build();  
  
Optional<String> mapping = PROPERTIES.getConfiguration().getParameter(DataMappable  
    .PATH); ②
```

① Set the property set data path using the `DataMappable.PATH` configuration property

② Get the data path, if available, through the `DataMappable.PATH` configuration property

5.18.2. Data mapping usage

The meaning and usage strategy of the data path value is completely dependent from each concrete API or implementation.

See, for example, the [JDBC Datastore](#) documentation for an use case of the data path representation.

5.19. 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* enviroment.

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.

5.20. Utilities

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

5.20.1. 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 lazyInitd = 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 than is returned to the caller

5.20.2. 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.

6. HTTP messages and RESTful Java client

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>5.2.3</version>
```

6.1. 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 core [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.

The [HttpMessage](#) API is the base HTTP message representation.

6.1.1. Headers

For a HTTP message, the header values are represented as a **List** of Strings, since HTTP supports multiple values for each header.

The [HttpMessage](#) API, through the [HttpHeaders](#) interface, provides a set of methods to inspect the HTTP message headers and obtain header values in a more useful and convenient way.

For example, frequently used header values can be obtained using a suitable Java type, if the value is available in the HTTP message headers.

```
HttpMessage<String> message = getMessage();

Optional<String> value = message.getHeaderValue("HEADER_NAME"); ①
Optional<Date> date = message.getDate(); ②
Optional<URI> location = message.getLocation(); ③
Optional<Long> length = message.getContentLength(); ④
Optional<Locale> locale = message.getLocale(); ⑤
List<Locale> locales = message.getLocales(); ⑥

Optional<String[]> basicAuth = message.getAuthorizationBasicCredentials(); ⑦
Optional<String> bearerAuth = message.getAuthorizationBearer(); ⑧
```

- ① Get the value of given header name, if present. If the HTTP header is present more than once then the values are joined together and separated by a **,** character.
- ② Get the value of the HTTP **Date** header, if available, as a Java **Date** instance
- ③ Get the value of the HTTP **Location** header, if available, as a Java **URI** instance
- ④ Get the value of the HTTP **Content-Length** header, if available, as a Java **Long**
- ⑤ Get the first (most *qualified*) **Locale** using the **Accept-Language** header, if present.
- ⑥ Get a list of **Locale** languages using the **Accept-Language** header, if present. If more than one language is specified in the **Accept-Language** header, returned Locales will be ordered relying on *quality* parameter.
- ⑦ Get the *basic* authorization credentials from a **Basic** type HTTP **Authorization** header, if available. The credentials are decoded from Base64 and returned as a username/password array.
- ⑧ Get the *bearer* authorization token from a **Bearer** type HTTP **Authorization** header, if available.

6.1.2. HttpRequest

The [HttpRequest](#) API represents a HTTP **request** message.

Besides the operations made available from the [HttpMessage](#) API, it 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.
- The request message body (payload) as an [InputStream](#).

The [HttpRequest](#) message type is bound to a [String](#) type message payload.

```
HttpRequest message = getRequestMessage();

HttpMethod method = message.getMethod(); ①
String path = message.getRequestPath(); ②
Optional<String> value = message.getRequestParameter("param1"); ③
Optional<List<String>> values = message.getMultiValueRequestParameter("param2"); ④
Optional<Cookie> cookie = message.getRequestCookie("cookie1"); ⑤

Optional<String> body = message.getPayload(); ⑥
InputStream bodyAsStream = message.getBody(); ⑦
```

- ① Get the HTTP request *method* as [HttpMethod](#) enumeration value
- ② Get the path of the HTTP request, relative to the base URI
- ③ Get a request URI parameter value, if available. If the parameter is multi-value, the values are joined together and separated by a [,](#) character
- ④ Get the values of a multi-value request URI parameter, if available
- ⑤ Get a request [Cookie](#) value, if available
- ⑥ Get the message body as a [String](#)
- ⑦ Get the message body as an [InputStream](#)

6.1.3. HttpResponse

The [HttpResponse](#) interface represents a HTTP **response** message.

Besides the operations made available from the [HttpMessage](#) API, it 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.

```
HttpResponse<String> message = getResponseMessage();

int statusCode = message.getStatusCode(); ①
HttpStatus status = message.getStatus(); ②
```

① Get the HTTP response status code

② Get the HTTP response status using the `HttpStatus` enumeration

6.1.4. Servlet API integration

The `ServletHttpRequest` API represents a `HttpRequest` backed by a `javax.servlet.http.HttpServletRequest` instance, and can be used as an adapter to deal with servlet request messages using the Holon Platform `HttpRequest` API.

The `ServletHttpRequest` API provides additional methods to obtain servlet related message information:

- The request *context* path.
- The request *URI*.
- The HTTP *session* id.

To obtain a `HttpRequest` API from a servlet `HttpServletRequest` instance, the `ServletHttpRequest.create(HttpServletRequest request)` method can be used.

```
HttpServletRequest servletRequest = getServletRequest();

HttpRequest request = ServletHttpRequest.create(servletRequest); ①
```

① Create a `HttpRequest` API from a `HttpServletRequest` instance



This way, a servlet request can be used for example with a Holon `MessageAuthenticator` to perform authentication operations directly using the request message. See the `MessageAuthenticator` section for further information.

6.2. RESTful client API

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` API 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
 - **Cache-Control** header configuration
- Convenience methods to setup authorization headers(**Basic** and **Bearer** types).
- Convenience methods to perform most common invocations using one of the **GET**, **POST**, **PUT**, **PATCH**, **DELETE**, **OPTIONS**, **TRACE** or **HEAD** methods.

6.2.1. 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

```
RestClient client = RestClient.create(); ①

client = RestClient.create("com.holonplatform.jaxrs.client.JaxrsRestClient"); ②

client = RestClient.forTarget("https://host/api"); ③
```

- ① Create a **RestClient** API using the default available implementation for current **ClassLoader**
- ② Create a **RestClient** API using a specific implementation class name

- ③ Create a `RestClient` API using the default available implementation and setting the *default* base URI

Available implementations

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

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

6.2.2. Configure defaults

The `RestClient` API supports some **default** configuration attributes, which will be used for each request performed using a `RestClient` instance:

- A **default target**, i.e. the default base URI which will be used for all the requests performed with the `RestClient` API, 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` API.

```
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

6.2.3. Build and configure a request

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

The request can be configured using the `RequestDefinition` API 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` API 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 Substitute two template variables values
- ② Substitute template variables values using a name-value map

URI *query* parameters

The `RestClient` API supports URI *query parameters* specification, with single or multiple values, through the `queryParameter(...)` methods.

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

- ① Set a single value query parameter
- ② Set a multiple values query parameter

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` types 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 values using the `accept(MediaType... mediaTypes)` builder method.



The `CacheControl` API provides a fluent builder to build and set a `Cache-Control` header value for the request, using the `cacheControl(CacheControl cacheControl)` builder method.

```
client.request().header("Accept", "text/plain"); ①
client.request().header(HttpHeaders.ACCEPT, "text/plain"); ②
client.request().accept("text/plain", "text/xml"); ③
client.request().accept(MediaType.APPLICATION_JSON); ④
client.request().acceptEncoding("gzip"); ⑤
client.request().acceptCharset("utf-8"); ⑥
client.request().acceptCharset(Charset.forName("utf-8")); ⑦
client.request().acceptLanguage("en-CA"); ⑧
client.request().acceptLanguage(Locale.US, Locale.GERMANY); ⑨
client.request().cacheControl(CacheControl.builder().noCache(true).noStore(true).build()); ⑩
```

- ① Set a request header, providing its name and its value
- ② Set a request header, providing its name through the `HttpHeaders` enumeration and its value
- ③ Set the request `Accept` header values
- ④ Set the request `Accept` header value using the `MediaType` enumeration
- ⑤ Set the request `Accept-Encoding` header value
- ⑥ Set the request `Accept-Charset` header value
- ⑦ Set the request `Accept-Charset` header value using the Java `Charset` class
- ⑧ Set the request `Accept-Language` header value
- ⑨ Set the request `Accept-Language` header values using the Java `Locale` class
- ⑩ Build a `CacheControl` definition and set it as `Cache-Control` request header value

Authorization headers

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

- The **Basic** authorization scheme, providing a *username* and a *password*, through the `authorizationBasic(String username, String password)` builder method.
- The **Bearer** authorization scheme, providing a *token*, through the `authorizationBearer(String bearerToken)` builder method.

```
client.request().authorizationBasic("username", "password"); ①
client.request().authorizationBearer("An389fz56xsr7"); ②
```

- ① Set the **Authorization** request header value using the **Basic** scheme and providing the credentials. Username and password will be encoded according to the [HTTP specifications](#)
- ② Set the **Authorization** request header value using the **Bearer** scheme and providing the bearer *token* value. See [RFC6750](#)

6.2.4. Invoke the request and obtain a response

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

The **Invocation** API provides a generic invocation method:

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

This method requires the following parameters:

- The HTTP **method** to use to perform the request (**GET**, **POST**, and so on), specified using the **HttpMethod** enumeration.
- An optional **request entity**, i.e. the request message *payload* (body), represented through the **RequestEntity** API.
- The expected **response entity type** using the **ResponseType** class, to declare the Java type of the response *payload* and apply a suitable converter, if available, to obtain the HTTP response body as the expected Java type.

The method returns a **ResponseEntity** type object, a **HttpResponse** extension which can be used to:

- Inspect the response message, for example to obtain the HTTP response **status** code, as a number or represented through the **HttpStatus** enumeration.
- Obtain the HTTP response raw *payload* or get it as a Java object, unmarshalled by a suitable converter which must be available from the concrete **RestClient** API implementation.



For non textual request or response payload types, any marshalling/unmarshalling strategy and implementation must be provided by the concrete **RestClient** API. See the specific **RestClient** [Available implementations](#) documentation for additional information.

See the next sections for details about the invocation parameters and return types.

6.2.5. Request entity

The `RequestEntity` interface can be used to provide a *request entity* to the `RestClient` API invocation methods, i.e. the request message *payload*.

The request *entity* is represented by a Java object and its serialization format is specified using a *media type* declaration (i.e. a **MIME** type definition) through the `Content-Type` request header value.



Depending on the `RestClient` API implementation used, you must ensure the request media type is supported and a suitable request message body converter is 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 instance using the most common media types, such as `text/plain`, `application/json`, `application/xml` and `application/x-www-form-urlencoded` (the latter 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, using `test` as request entity value
- ② Build a `application/json` type request entity, using a `TestData` class instance as request entity value
- ③ Build a `application/x-www-form-urlencoded` type request entity, using the `formBuilder` method to build the *form* data map

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

```
RequestEntity<?> emptyRequest = RequestEntity.EMPTY; ①
```

- ① Build an empty request empty, to provide a request message without a payload

6.2.6. Response type

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

In addition to a simple Java class type, a *parametrized* type can be declared, allowing to use Java *generic* types as response types.

```
ResponseType<TestData> responseType1 = ResponseType.of(TestData.class); ①

ResponseType<List<TestData>> responseType2 = ResponseType.of(TestData.class, List
.class); ②
```

- ① Declares a response type as `TestData` type
- ② Declares a response type as a `List` of `TestData` types

6.2.7. Response entity

The `ResponseEntity` interface is used by `RestClient` API to represent the invocation *response* and to deal with the optional response *entity* obtained as invocation result.

Since it is a `HttpResponse` instance, the `ResponseEntity` API can be used to inspect the response message, for example the HTTP message headers, including the HTTP status code.

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

HttpStatus status = response.getStatus(); ②
int statusCode = response.getStatusCode(); ③
long contentLength = response.getContentLength().orElse(-1L); ④
Optional<String> value = response.getHeaderValue("HEADER_NAME"); ⑤
```

- 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 as `HttpStatus` enumeration value
- ③ Get the response status code
- ④ Get the `Content-Length` header value
- ⑤ Get a generic header value

To obtain the response *entity* value as the expected Java type, the `getPayload()` method can be used. The return `ResponseEntity` object generic type is provided according to the specified `Response type`, so the payload value will be an instance of the expected response Java type.

Furthermore, the `ResponseEntity` API makes available the `as(Class entityType)` type as a different type from the one specified with the `Response type` invocation parameter, if the media type is supported by the concrete `RestClient` API implementation and a suitable converter is available.

When a response has not a payload, i.e. the response *entity* is not available, the `Optional` result of the `getPayload()` and `as(Class entityType)` methods will be empty.

```

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

boolean hasEntity = response.getPayload().isPresent(); ②

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

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

```

- ① Perform a **GET** request, setting the **Accept** header as **application/json** and declaring the **TestData** class as expected response entity Java type
- ② Checks whether a response *entity* payload is available
- ③ Get the response entity value, as a **TestData** class instance
- ④ Get the response entity value as a **String**



Depending on the concrete **RestClient** API implementation, you must ensure the response media type is supported and a suitable message body converter is available to deal with the Java object type and the media type of the response entity.

6.2.8. Specific request invocation methods

In most cases, it is easier and faster to use HTTP *method*-specific invocation methods, made available by the **RestClient** invocation API.

Each invocation method is relative to a specific HTTP request *method* and it is named accordingly. More than one method version is provided for each HTTP request method, providing the most suitable parameters and response types for for the most common situations.

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

1. A set of methods to optionally provide a **Request entity** and to obtain a **Response entity**. If the response is expected to contain a payload which has to be deserialized into a Java object, the **Response type** can be specified, either as a simple or parametrized Java class.

```

final RestClient client = RestClient.forTarget("https://rest.api.example/test");

ResponseEntity<TestData> response = client.request().get(TestData.class); ①
response = client.request().get(ResponseType.of(TestData.class)); ②

response = client.request().put(RequestEntity.json(new TestData()), TestData.class);
③

```

- ① Perform an invocation using the **GET** method and obtain a **ResponseEntity** expecting the **TestData** class as response entity type

- ② The same invocation using the `ResponseType` API to specify the expected response entity type
- ③ Perform an invocation using the `PUT` method and providing an `application/json` type request entity, expecting a `TestData` response entity type

When a response entity is not expected, this category of invocation methods return a `Void` type `ResponseEntity`.

```
ResponseEntity<Void> response2 = client.request().post(RequestEntity.json(new  
    TestData())); ①  
HttpStatus status = response2.getStatus(); ②
```

- ① Perform an invocation using the `POST` method and providing an `application/json` type request entity, but no response entity is expected
- ② Get the response HTTP status

2. A set of method to directly obtain the deserialized response *entity* value, named with the `ForEntity` suffix. This methods expects a *successful* response (i.e. a response with a `2xx` HTTP status code), otherwise an `UnsuccessfulResponseException` is thrown. The exception which can be inspected to obtain the response status code and the response itself. This kind of methods returns an `Optional` value, which will be empty for empty responses.

```
Optional<TestData> value = client.request().getForEntity(TestData.class); ①  
Optional<List<TestData>> values = client.request().getForEntity(ResponseType.of  
(TestData.class, List.class)); ②
```

- ① Perform an invocation using the `GET` method and directly obtain the `TestData` type response entity value, if available
- ② Perform an invocation using the `GET` method and directly obtain a `List` of `TestData` type response entity values, if available

The `UnsuccessfulResponseException` type, which is thrown by the `xxxForEntity` invocation methods when the response status code do not belongs to the `2xx` family, provides some information about the invocation failure:

- The actual response **status** code.
- A reference to the actual `ResponseEntity` instance.

```
try {  
    client.request().getForEntity(TestData.class);  
} catch (UnsuccessfulResponseException e) {  
    // got a response with a status code different from 2xx  
    int httpStatusCode = e.getStatusCode(); ①  
    Optional<HttpStatus> sts = e.getStatus(); ②  
    ResponseEntity<?> theResponse = e.getResponse(); ③  
}
```


- ① Get the actual response status code
- ② Get the response status code as a `HttpStatus`
- ③ Get the `ResponseEntity` instance

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

- A `getForStream` method to perform a request using the HTTP `GET` method and obtain the response *entity* as an `InputStream`. This can be useful, for example, for API invocations which result is a stream of byte or characters.

```
InputStream responseEntityStream = client.request().getForStream();
```

- A `getAsList` method, to perform a request using the HTTP `GET` method and obtain the response entity contents as a `List` of deserialized Java objects in a specified expected response type. For empty response entities, an empty List is returned.

```
List<TestData> collectionOfValues = client.request().getAsList(TestData.class);
```

- A `postForLocation` to perform a request using the HTTP `POST` and directly obtain the `Location` response header value as a Java `URI` instance, if available.

```
Optional<URI> locationHeaderURI = client.request().postForLocation(RequestEntity.json  
(new TestData()));
```

6.2.9. `RestClient` API invocation methods reference

Below a reference list of the `RestClient Invocation` API, available from the request definition API:

```
RestClient restClient = RestClient.forTarget("http://api.example"); // Obtain a  
RestClient  
restClient.request(); // Request definition
```

Generic invocations:

Operation	Description	Parameters	Returns	Response status handling
<code>invoke</code>	Invoke the request and receive a response back.	<ol style="list-style-type: none"> 1. HTTP method 2. Optional <code>RequestEntity</code> 3. Expected response entity type (<code>Void</code> for none) 	A <code>ResponseEntity</code> instance with expected response entity payload type	<i>None</i>
<code>invokeForSuccess</code>	Invoke the request and receive a response back only if the response has a <i>success (2xx)</i> status code.	<ol style="list-style-type: none"> 1. HTTP method 2. Optional <code>RequestEntity</code> 3. Expected response entity type (<code>Void</code> for none) 	The <code>ResponseEntity</code> with the provided response type payload type	If the response status code is not <i>2xx</i> , an <code>UnsuccessfulResponseException</code> is thrown
<code>invokeForEntity</code>	Invoke the request and receive back the response content entity, already deserialized in the expected response type.	<ol style="list-style-type: none"> 1. HTTP method 2. Optional <code>RequestEntity</code> 3. Expected response entity type 	Optional response <i>entity</i> value, already deserialized in the expected response entity type	If the response status code is not <i>2xx</i> , an <code>UnsuccessfulResponseException</code> is thrown

By method invocations:

1. `GET`:

Operation	Parameters	Returns	Response status handling
<code>get</code>	Expected response entity type, either using a <code>Class<T></code> or a <code>ResponseType<T></code> to handle generic types	A <code>ResponseEntity<T></code> instance, with expected response entity payload type	<i>None</i>
<code>getForEntity</code>	Expected response entity type, either using a <code>Class<T></code> or a <code>ResponseType<T></code> to handle generic types	Optional response <i>entity</i> value (<code>T</code>), already deserialized in the expected response entity type	If the response status code is not <i>2xx</i> , an <code>UnsuccessfulResponseException</code> is thrown

Operation	Parameters	Returns	Response status handling
getForStream	<i>None</i>	The response payload stream as an InputStream , or an empty stream for empty responses	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown
getAsList	Expected response entity type (Class<T>)	A List of the deserialized response entities using the provided response entity type, or an empty list for empty responses	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown

2. POST:

Operation	First parameter	Second parameter	Returns	Response status handling
post	The request <i>entity</i> represented as RequestEntity instance	<i>Optional</i> expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	A ResponseEntity<T> instance, with expected response entity payload type. If the second parameter is not specified, a Void type ResponseEntity is returned	<i>None</i>
postForEntity	The request <i>entity</i> represented as RequestEntity instance	Expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	Optional response <i>entity</i> value (T), already deserialized in the expected response entity type	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown
postForLocation	The request <i>entity</i> represented as RequestEntity instance	<i>None</i>	Optional Location response header value	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown

3. PUT:

Operation	First parameter	Second parameter	Returns	Response status handling
put	The request <i>entity</i> represented as RequestEntity instance	<i>Optional</i> expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	A ResponseEntity<T> instance, with expected response entity payload type. If the second parameter is not specified, a Void type ResponseEntity is returned	<i>None</i>
putForEntity	The request <i>entity</i> represented as RequestEntity instance	Expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	Optional response <i>entity</i> value (T), already deserialized in the expected response entity type	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown

4. PATCH:

Operation	First parameter	Second parameter	Returns	Response status handling
patch	The request <i>entity</i> represented as RequestEntity instance	<i>Optional</i> expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	A ResponseEntity<T> instance, with expected response entity payload type. If the second parameter is not specified, a Void type ResponseEntity is returned	<i>None</i>
patchForEntity	The request <i>entity</i> represented as RequestEntity instance	Expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	Optional response <i>entity</i> value (T), already deserialized in the expected response entity type	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown

5. DELETE:

Operation	Parameter	Returns	Response status handling
delete	<i>Optional</i> expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	A ResponseEntity<T> instance, with expected response entity payload type. If the second parameter is not specified, a Void type ResponseEntity is returned	<i>None</i>
deleteOrFail	<i>None</i>	<i>Nothing</i>	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown
deleteForEntity	Expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	Optional response entity value (T), already deserialized in the expected response entity type	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown

6. OPTIONS:

Operation	Parameter	Returns	Response status handling
options	<i>Optional</i> expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	A ResponseEntity<T> instance, with expected response entity payload type. If the second parameter is not specified, a Void type ResponseEntity is returned	<i>None</i>
optionsForEntity	Expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	Optional response entity value (T), already deserialized in the expected response entity type	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown

7. TRACE:

Operation	Parameter	Returns	Response status handling
trace	<i>Optional</i> expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	A ResponseEntity<T> instance, with expected response entity payload type. If the second parameter is not specified, a Void type ResponseEntity is returned	<i>None</i>
traceForEntity	Expected response entity type, either using a Class<T> or a ResponseType<T> to handle generic types	Optional response <i>entity</i> value (T), already deserialized in the expected response entity type	If the response status code is not 2xx , an UnsuccessfulResponseException is thrown

8. HEAD:

Operation	Returns	Response status handling
head	A Void type ResponseEntity	<i>None</i>

6.2.10. Property and PropertyBox support

The **RestClient** API fully supports the Holon Platform **Property** model when used along with the **PropertyBox** data type as a request/response *entity* in RESTful API calls.

Regarding the **JSON** media type, the **PropertyBox** type marshalling and unmarshalling support is provided by the **Holon Platform JSON module**. For the **builtin RestClient API implementations**, the **PropertyBox** type JSON support is automatically set up when the suitable Holon platform JSON module artifacts are available in classpath.

When a response entity value has to be deserialized into a **PropertyBox** object type, the **property set** to be used must be specified along with the response entity type, in order to instruct the JSON module unmarshallers about the property set with which to build the response **PropertyBox** instances.

For this purpose, the **RestClient** invocation API **propertySet(...)** methods can be used to specify the **property set** with which to obtain a **PropertyBox** type response entity value.

```

final PathProperty<Integer> CODE = create("code", int.class);
final PathProperty<String> VALUE = create("value", String.class);
final PropertySet<?> PROPERTIES = PropertySet.of(CODE, VALUE);

RestClient client = RestClient.create();

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

Optional<PropertyBox> box2 = client.request().target("https://rest.api.example").path(
    "/apimethod")
    .propertySet(CODE, VALUE).getForEntity(PropertyBox.class); ②

List<PropertyBox> boxes = client.request().target("https://rest.api.example").path(
    "/apimethod")
    .propertySet(PROPERTIES).getAsList(PropertyBox.class); ③

```

- ① GET request for a `PropertyBox` type response, using `PROPERTIES` as property set
- ② Response `PropertyBox` property set specification using directly an array of properties
- ③ GET request for a list of `PropertyBox` type response, using `PROPERTIES` as property set

7. Authentication and Authorization

The `holon-auth` artifact provides a complete and highly configurable **authentication** and **authorization** architecture, integrated with all the platform modules.

Maven coordinates:

```

<groupId>com.holon-platform.core</groupId>
<artifactId>holon-auth</artifactId>
<version>5.2.3</version>

```

7.1. Realm

The `Realm` API represents a security abstraction providing operations for *principals* **authentication** and **authorization**.

The `Realm` API is the main entry point to deal with the Holon Platform authentication and authorization architecture: it holds the configuration of the authentication and authorization context and provides operations to perform *principals* authentication and authorization controls.

The `Realm` **authentication strategy** is defined using a set of *authenticators*, represented by the `Authenticator` interface, each bound to a specific `AuthenticationToken`, which represents the principal's credentials.

In a mirrored way, the **Realm authorization strategy** is defined using a set of *authorizers*, represented by the **Authorizer** interface, each bound to a specific **Permission** type, and used by the **Realm** API to perform authorization controls against the principal's granted permissions.

The *authenticators* and *authorizers* bound to a specific **Realm** instance define the authentication and authorization strategy of such **Realm**, so they are registered at **Realm** configuration time.

The **Realm** API provides a *fluent* builder to build and configure a **Realm** instance, with *authenticators* and *authorizers* registration methods.

```
Realm realm = Realm.builder() ①
    .withAuthenticator(AUTHENTICATOR1) ②
    .withAuthenticator(AUTHENTICATOR2) ③
    .withAuthorizer(AUTHORIZER1) ④
    .withAuthorizer(AUTHORIZER2) ⑤
    .build();
```

- ① Obtain a **Realm** builder
- ② Register an **Authenticator**
- ③ Register another **Authenticator**
- ④ Register an **Authorizer**
- ⑤ Register another **Authorizer**

The **Authenticator** and **Authorizer** sections describe these API definitions in detail.

7.1.1. Realm name

A **Realm** instance can be identified by a **name**, which can be used to identify a specific **Realm** instance when more than one is available.

```
Realm realm = Realm.builder().name("nyname").build(); ①

Optional<String> name = realm.getName(); ②
```

- ① Set the **Realm** name using the default builder
- ② Get the **Realm** name, if available

7.1.2. Realm authentication

Authentication requests are made available through the **Authenticator** API, which is implemented by the **Realm** API.

The **Authenticator** API provides the following method to perform authentication requests:


```
Authentication authenticate(AuthenticationToken authenticationToken) throws
AuthenticationException;
```

The authentication request is represented by an `AuthenticationToken` and the authenticated principal, if the authentication request is successful, is returned as an `Authentication` representation.

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

The authentication flow is structured as follows:

1. A concrete `AuthenticationToken`, which represents the authentication request (for example, the *principal's* credentials), is provided to the `authenticate` method;
2. The `Realm` checks if a suitable `Authenticator` is registered, i.e. an `Authenticator` which can handle the given `AuthenticationToken` type. If not, an `UnsupportedTokenException` is thrown;
3. The `authenticate(AuthenticationToken authenticationToken)` method is called on the specific `Authenticator` API, performing the concrete authentication operation.
4. If the authentication operation is successful, the authenticated principal is returned using `Authentication` representation.
5. Otherwise, an `AuthenticationException` type is thrown. The concrete type of the exception gives more detailed informations on what went wrong.

Each `Authenticator` declares the `AuthenticationToken` type to which is bound through the `getTokenType()` method. When a new `Authenticator` is registered, the `Realm` instance will support the `AuthenticationToken` type which is bound to the registered `Authenticator`, and such `Authenticator` will be used to perform the authentication operation when a matching `AuthenticationToken` type is provided.

A concrete `Authenticator` can be registered in a `Realm` instance in two ways:

- Using the `Realm` API *builder*.
- Using the `addAuthenticator` method of the `Realm` API.

```
Realm realm = Realm.builder().withAuthenticator(AUTHENTICATOR1).build(); ①

realm.addAuthenticator(AUTHENTICATOR2); ②
```

① Add a `Realm` `Authenticator` using the *builder* API

② Add a `Realm` `Authenticator` using the `Realm` API method

To check if a `Realm` instance supports a specific `AuthenticationToken` type, the `supportsToken` API method can be used.

```
Realm realm = getRealm();
```

```
boolean supported = realm.supportsToken(MyAuthenticationToken.class); ①
```

① Checks whether given **Realm** supports the **MyAuthenticationToken** authentication token type

So the **Realm** API is itself an **Authenticator**, bound to a generic **AuthenticationToken** type. The **authenticate** method is the entry point to perform any authentication request, providing a suitable **AuthenticationToken** type implementation.

```
Realm realm = getRealm();
```

```
try {  
    Authentication authc = realm.authenticate(new MyAuthenticationToken("test")); ①  
} catch (AuthenticationException e) {  
    // handle failed authentication  
}
```

① Perform an authentication request using the **MyAuthenticationToken** authentication token type

See the next sections for details about authenticators, authentication tokens and the authenticated principal representation.

7.1.3. 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. The return type is a generic **Object**, since each authentication model could represent the *principal* credentials in a different way.

Each **AuthenticationToken** sub-type is bound to an **Authenticator**, which is able to interpret the *principal* and *credentials* representations and to perform the actual authentication process using the information provided through the **AuthenticationToken** instance.

Some builtin **AuthenticationToken** representations are provided by the core Holon Platform module:

Account credentials authentication 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"); ①
```

- ① Create an *account credentials* authentication token type using `username` as account id (the principal's name) and `password` as account secret (the account credentials)

Bearer authentication 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"); ①
```

- ① Create a *bearer* authentication token type, providing the token value

7.1.4. Authenticator

As stated in the [Realm authentication](#) flow description, the `Realm` API relies on the registered `Authenticator` instances to perform the actual authentication process, according to the provided `AuthenticationToken` type.

The `Authenticator` API represents a concrete authentication strategy, using a specific `AuthenticationToken` type to represent the authentication request. The `AuthenticationToken` type to which an `Authenticator` is bound is provided by the `getTokenType()` method.

The `authenticate(AuthenticationToken authenticationToken)` method of the `Authenticator` API is used to perform the actual authentication operation, checking the principal's credentials provided through the `AuthenticationToken` instance and returning an `Authentication` representation of the authenticated principal if the process was successful.

When an authentication request is not successful, an `AuthenticationException` type is thrown. The concrete type of the exception gives more detailed informations on what went wrong. See [Authentication exceptions](#) for a list of the authentication exceptions available by default.

```

Authenticator<MyAuthenticationToken> authenticator = Authenticator.create
(MyAuthenticationToken.class, ①
    token -> {
        // check authentication token information
        token.getPrincipal();
        token.getCredentials();
        boolean valid = true; // ...
        // if not valid, throw an exception
        if (!valid) {
            throw new InvalidCredentialsException();
        }
        // otherwise, return the authenticated principal representation
        return Authentication.builder("thePrincipalName").build();
    });

try {
    Authentication authc = authenticator.authenticate(new MyAuthenticationToken("test")
); ②
} catch (AuthenticationException e) {
    ③
}

```

- ① Create an **Authenticator** bound to the **MyAuthenticationToken** authentication token type
- ② Perform an authentication request on the **Authenticator**, obtaining the authenticated principal representation if successful.
- ③ If the authentication request is not successful, an **AuthenticationException** type is thrown



See the [Authentication](#) section for information about the authenticated principal representation.

Builtin authenticators

- See the [Account](#) section to learn about the builtin *account credentials* type authenticator.
- See the [JWT support](#) section learn about the builtin *JSON Web Token* type authenticator.

Authenticator example

Below is provided a simple example on how to create a custom **Authenticator**, bound to a specific **AuthenticationToken**, register it into a **Realm** and use it to handle authentication requests.

```

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;
    }

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

public void authenticate() {
    Realm realm = Realm.builder().withAuthenticator(new MyAuthenticator()).build(); ❹

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

```

❶ Create an `AuthenticationToken` implementation, which returns the principal name as a String

from the `getPrincipal()` method

- ② Create a custom `Authenticator` bound to the `MyAuthenticationToken` type
- ③ This authenticator only accepts `test` named principals, building a simple `Authentication` instance with the provided principal name. Otherwise, an `UnknownAccountException` is thrown.
- ④ Create a `Realm` and register the custom authenticator
- ⑤ Perform an authentication request using a `MyAuthenticationToken` instance

7.1.5. Authentication exceptions

Below a list of the default authentication exceptions.

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

7.1.6. Authentication

The result of an `Authenticator` successful authentication request is represented by the `Authentication` API.

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:

- An optional set of `Permission` granted to the authenticated principal.
- A `isRoot()` flag, to mark the authenticated *principal* as a *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, to identify the authentication scheme with which the *principal* was authenticated. See `MessageAuthenticator` for details about authentication

schemes.

- It extends the Holon Platform [ParameterSet](#) API, which represents a set of custom name-value attributes and 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.

The [Authentication](#) API provides a *builder* to create new [Authentication](#) instances.

```
Authentication authc = Authentication.builder("userId") ①
    .withPermission("VIEW") ②
    .withPermission(new MyPermission()) ③
    .withParameter("name", "John") ④
    .withParameter("surname", "Doe") ⑤
    .scheme("myscheme") ⑥
    .build();
```

- ① Obtain an [Authentication](#) *builder* and set `userId` as principal name
- ② Add a [VIEW](#) String type granted permission (using the *role* name convention)
- ③ Add a custom [MyPermission](#) type granted permission
- ④ Add `name` named parameter
- ⑤ Add `surname` named parameter
- ⑥ Set `myscheme` as authentication scheme

7.1.7. Authentication listeners

The [AuthenticationListener](#) interface can be used to be notified when a successfull authentication is performed. The authenticated principal, represented as an [Authentication](#) instance, is provided to the listener method.

The [AuthenticationNotifier](#) API allows to add and remove an [AuthenticationListener](#). The [AuthenticationNotifier](#) is implemented by the [Realm](#) API, so an [AuthenticationListener](#) can be registered in a [Realm](#) instance to be notified when a successfull authentication request is performed.

```
Realm realm = getRealm();

realm.addAuthenticationListener(authentication -> { ①
    // do something ...
    authentication.getName();
});
```

- ① Add an [AuthenticationListener](#) to given [Realm](#) instance

7.1.8. MessageAuthenticator

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

The `MessageAuthenticator` API relies on the Holon Platform `Message` API as a generic *message* representation.



See the [HTTP messages](#) section to learn about the HTTP implementations of the `Message` API.

The `MessageAuthenticator` API allows to perform authentication requests **directly using a message** to provide the authentication request information.

The aim of a `MessageAuthenticator` is to *translate* a *message* representation into a standard `AuthenticationToken` representation, through the `AuthenticationTokenResolver` interface, and then use the token in order to perform a conventional authentication flow.

An `AuthenticationTokenResolver` is bound to a specific `Message` type and provides an additional message partitioning level, the authentication **scheme**. This way, for the same type of `Message`, different resolvers can be provided for different authentication *schemes*.

For example, taken two resolvers bound to the same `HttpMessage` type, one could deal with *basic* authentication scheme and the other with *bearer* authentication scheme.

For example, supposing to use:

- A custom `Message` implementation class, that we'll call `MyMessage`.
- A custom `AuthenticationToken` type, called `MyMessageAuthenticationToken`.

We want to create an `AuthenticationTokenResolver` that processes a `MyMessage` instance, looks for a `MY_HEADER` message header value and, if available, resolves the message in a `MyMessageAuthenticationToken` instance setting the `MY_HEADER` header value as token *principal* id:

```
AuthenticationTokenResolver<MyMessage> myResolver = AuthenticationTokenResolver.  
create(MyMessage.class, ①  
    msg -> msg.getHeader("MY_HEADER").map(value -> new MyMessageAuthenticationToken  
(value)) ②  
);
```

① Create an `AuthenticationTokenResolver` bound to the `MyMessage` message type

② If the `MY_HEADER` message header value is present, return a `MyMessageAuthenticationToken` token type, setting the header value as principal

The `MessageAuthenticator` API supports more than one *message* type, through a set of registered `AuthenticationTokenResolver`. The `supportsMessage` and `getResolversForMessageType` methods of the `MessageAuthenticator` API allow to check if a message type is supported (i.e. one or more `AuthenticationTokenResolver` is available for given message type) and to obtain the resolvers for a specific message type, respectively.

The `MessageAuthenticator` API makes available a specialized method which accepts a `Message` as authentication request representation:

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

This method can be used to **perform an authentication request directly using a `Message` representation**.

Use `Realm` as a `MessageAuthenticator`

The `Realm` API extends the `MessageAuthenticator` API, so a `Realm` instance can be used to process authentication requests using messages.

The default `Realm` builder provides methods to register one ore more `AuthenticationTokenResolver`.

The message based authentication flow is structured as follows:

1. Lookup for suitable `AuthenticationTokenResolver` to obtain an `AuthenticationToken` from the request message, i.e. a resolver bound to given message type;
2. If one or more authentication **scheme** is specified, only the `AuthenticationTokenResolver` bound to the provided scheme names will be taken into account.
3. If a not empty set of `AuthenticationTokenResolver` is obtained using the strategy described above, all the suitable resolvers are invoked to obtain an `AuthenticationToken`: the `AuthenticationToken` to be used will be the one obtained from the first `AuthenticationTokenResolver` which provides a not null token (or better, the first resolver which returns a not empty `Optional`, according to the `AuthenticationTokenResolver` API). The resolvers are invoked in the order they were registered and according to provided authentication *schemes* order, if any.
4. If an `AuthenticationToken` was successfully obtained, it will be used to trigger a conventional authentication request, as described in the `Realm authentication` section.
5. Otherwise, an `UnsupportedMessageException` is thrown.

```
Realm realm = Realm.builder().withResolver(myResolver) ①
    .withAuthenticator(new MyAuthenticator()) ②
    .build();

MyMessage message = new MyMessage();

Authentication authc = realm.authenticate(message); ③
```

① Build an configure a `Realm` registering an `AuthenticationTokenResolver`

② Add an `Authenticator` to handle actual `AuthenticationToken` based authentication requests

③ Perform the authentication request using a `MyMessage` message instance

Builtin HTTP message resolvers

For the `HttpRequest` message type, two builtin `AuthenticationTokenResolver` are provided: one to

deal with the HTTP **Basic** scheme and the other to handle the **Bearer** scheme.

Basic HTTP message resolver:

This **AuthenticationTokenResolver** can be obtained using the **httpBasicResolver()** method of the **AuthenticationToken** interface:

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

The resolver inspect the message **Authorization** header, if available, checking if it is declared using the **Basic** authorization scheme. If so, extracts and decodes the basic credentials (username and password) and uses them to create an **Account credentials authentication token** authentication token type.

This resolver can be used, for example, to implement a message based account authentication strategy. See **Account** section to learn how to use the Holon Platform APIs for this purpose.

```
AccountProvider accountProvider = getAccountProvider();

Realm realm = Realm.builder().withResolver(AuthenticationToken.httpBasicResolver()) ①
    .withAuthenticator(Account.authenticator(accountProvider)) ②
    .build();

// obtain the HttpRequest message, for example using a servlet request
HttpRequest request = getHttpRequest();
try {
    Authentication authc = realm.authenticate(request); ③
} catch (AuthenticationException e) {
    // handle authentication failures
}
```

- ① Register a **Basic** authorization HTTP message resolver
- ② Register an **Authenticator** to handle account credentials based authentication
- ③ Perform the authentication request using the HTTP request message

Bearer HTTP message resolver:

This **AuthenticationTokenResolver** can be obtained using the **httpBearerResolver()** method of the **AuthenticationToken** interface:

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

The resolver inspect the message **Authorization** header, if available, checking if it is declared using the **Bearer** authorization scheme. If so, extracts the bearer *token* value and uses it to create an **Bearer authentication token** authentication token type.

This resolver can be used, for example, to implement a *JWT (JSON Web Token)* message based authentication. See [JWT support](#) section to learn how to use the Holon Platform APIs for this purpose.

7.1.9. Realm authorization

Authorization controls operations are provided by the [Authorizer](#) API, which is implemented by the [Realm](#) API itself.

The [Authorizer](#) API provides a set of `isPermitted(...)` methods to perform authorization controls, using the [Permission](#) representation to validate a set of permissions against an [Authentication](#) (which represents an authenticated principal) granted authorities.

A [Permission](#) represents a granted authority and its actual meaning and validation strategy depends on the concrete [Authorizer](#) implementation, which can use specific [Permission](#) sub types to represent any kind of granted authority.

The [Realm](#) API itself does not implement any concrete authorization control strategy, but delegates the specific authorization control strategy to one or more concrete [Authorizer](#), relying on the [Permission](#) type in order to discern which [Authorizer](#) has to be used to handle a specific permission type.

A concrete [Authorizer](#) can be registered in a [Realm](#) instance in two ways:

- Using the [Realm](#) API *builder*.
- Using the `addAuthorizer` method of the [Realm](#) API.

```
Realm realm = Realm.builder().withAuthorizer(AUTHORIZER1).build(); ①  
realm.addAuthorizer(AUTHORIZER2); ②
```

① Add a [Realm](#) [Authorizer](#) using the *builder* API

② Add a [Realm](#) [Authorizer](#) using the [Realm](#) API method

Each [Authorizer](#) declares the [Permission](#) type to which is bound through the `getPermissionType()` method. To check if a [Realm](#) instance supports a specific [Permission](#) type, the `supportsPermission` API method can be used.

```
Realm realm = getRealm();  
  
boolean supported = realm.supportsPermission(MyPermission.class); ①
```

① Checks whether given [Realm](#) supports the [MyPermission](#) permission type

So the [Realm](#) API is itself an [Authorizer](#), bound to a generic [Permission](#) type. The [Authorizer](#) API provides a set of `isPermitted(...)` methods, which accept an [Authentication](#) instance and a set of permissions to be validated according to an authenticated principal's granted authorities, obtained through the `getPermissions()` method of the [Authentication](#) API.

The permissions check methods provided by the `Authorizer` API can be categorized as follows:

Operation	Parameters type	Returns	Description
<code>isPermitted(Authentication authentication, Set of permissions)</code>	<code>Permission</code>	<code>true</code> or <code>false</code>	Check if the given authentication has all the specified permissions.
<code>isPermittedAny(Authentication authentication, Set of permissions)</code>	<code>Permission</code>	<code>true</code> or <code>false</code>	Check if the given authentication has any of the specified permission.
<code>isPermitted(Authentication authentication, String permission representations)</code>	<code>String</code>	<code>true</code> or <code>false</code>	Check if the given authentication has all the specified permissions, using the <code>String</code> permission representation, which can be obtained using the <code>Permission.getPermission()</code> method.
<code>isPermittedAny(Authentication authentication, String permission representations)</code>	<code>String</code>	<code>true</code> or <code>false</code>	Check if the given authentication has any of the specified permission, using the <code>String</code> permission representation, which can be obtained using the <code>Permission.getPermission()</code> method.

For standard **role based** authorization controls, the `Authorizer` API provides a set of methods which accept the **String representation** of a permission, which can be obtained using the `Permission.getPermission()` method.

```
Realm realm = getRealm();
Authentication principal = getAuthentication();

boolean permitted = realm.isPermitted(principal, new MyPermission()); ①
permitted = realm.isPermittedAny(principal, new MyPermission(), new AnotherPermission()); ②
permitted = realm.isPermitted(principal, "role1"); ③
permitted = realm.isPermittedAny(principal, "role1", "role2"); ④
```

① Checks if the `MyPermission` instance is granted to given principal

- ② Checks if any of `MyPermission` or `AnotherPermission` is granted to given principal
- ③ Checks if the `role1` role name (i.e. the String permission representation) is granted to given principal
- ④ Checks if the `role1` or the `role2` role name (i.e. the String permission representation) are granted to given principal

See the next sections for details about authorizers and permission representations.

7.1.10. Authorizer

As stated in the [Realm authorization](#) strategy description, the `Realm` API relies on the registered `Authorizer` instances to perform the actual authorization controls, according to a specific `Permission` type.

The `Authorizer` API represents a concrete authorizations control strategy, using a specific `Permission` type to represent a granted authority. The `Permission` type to which an `Authorizer` is bound is provided by the `getPermissionType()` method.

Default Authorizer

A default `Authorizer` implementation is provided and can be obtained using the `Authorizer.create()` static method. Alternatively, the default `Authorizer` can be created and registered in a `Realm` instance at the same time using the `withDefaultAuthorizer()` Realm builder method.

```
Authorizer<Permission> defaultAuthorizer = Authorizer.create(); ①

Realm realm = Realm.builder().withDefaultAuthorizer() ②
    .build();
```

- ① Create the default `Authorizer` implementation
- ② Create a default `Authorizer` and register it in the `Realm` instance

The default `Authorizer`, since it is bound to the generic `Permission` type, is able to handle any kind of permission and uses the following authorization control strategy:

1. Checks if the `Authentication.isRoot()` method returns `true`: if so, the authorization control is skipped and `true` is always returned from the `isPermitted(...)` methods.
2. Get the `Authentication` granted permissions using the `getPermissions()` method.
3. Compares the `Authentication` granted permissions (if any) with the provided permissions to check, using the standard `equals(...)` method to compare a single permission to another.



To use a custom `Permission` type consistently with the default `Authorizer`, the `equals()/hashCode()` comparison logic of the concrete permission type must be provided accordingly.



See [Authentication](#) for more information about the authenticated principal's granted permissions.

7.1.11. Permission

The [Permission](#) API is the representation of a granted authority.

The actual meaning of a specific [Permission](#) type is implementation dependent and it is bound to the concrete [Authorizer](#) which implements the authorization control strategy.

The [Permission](#) API makes available a [getPermission\(\)](#) method, which returns the *Optional* String representation of the permission, if the permission type can be represented as a String retaining a sufficient precision to be relied upon for authorization control decisions.

Default Permission

A default [Permission](#) implementation is available and can be obtained through the [Permission.create\(String permission\)](#) static method.

The default [Permission](#) implementation can be used as a **role** name representation. The role name is returned from the [getPermission\(\)](#) method and it is used as permission *identity* in the [equals\(\)](#) / [hashCode\(\)](#) based comparison logic.

```
Permission permission = Permission.create("myrole"); ①  
  
Optional<String> roleName = permission.getPermission(); ②
```

① Create a default [Permission](#) using the [myrole](#) role name

② The [getPermission\(\)](#) will return the [myrole](#) String

Authorization control example

Below is provide a simple example on how to use the default [Authorizer](#) and the default [Permission](#) implementation to perform authorization controls:

```

final Permission p1 = Permission.create("role1"); ①
final Permission p2 = Permission.create("role2"); ②

Authentication authc = Authentication.builder("test").withPermission(p1)
    .withPermission(p2).build(); ③

// Realm with default authorizer
Realm realm = Realm.builder().withDefaultAuthorizer().build(); ④

// permission checking
boolean permitted = realm.isPermitted(authc, p1); ⑤
permitted = realm.isPermitted(authc, "role1"); ⑥
permitted = realm.isPermittedAny(authc, p1, p2); ⑦
permitted = realm.isPermittedAny(authc, "role1", "role2"); ⑧

boolean notPermitted = realm.isPermitted(authc, "other_role"); ⑨

```

- ① Create a default **Permission** using the **role1** role name
- ② Create a default **Permission** using the **role2** role name
- ③ Build an **Authentication** and grant the two previously defined permission to it
- ④ Create a **Realm** and register the default **Authorizer**
- ⑤ Verify the **p1** permission is permitted
- ⑥ Verify the **role1** role name is permitted
- ⑦ Verify the **p1** or the **p2** permission is permitted
- ⑧ Verify the **role1** or **role2** role name is permitted
- ⑨ The **other_role** role name is not permitted

7.1.12. **Realm** as a **Context** resource

When a **Realm** is managed using the Holon Platform **Context** architecture, and the default name, i.e. the **Realm** class name, is used as context resource key, the **getCurrent()** and **require()** methods can be used to obtain the current **Realm** instance from context.

```

final Realm realm = Realm.builder().build();

Context.get().classLoaderScope().map(scope -> scope.put(Realm.CONTEXT_KEY, realm)); ①

Optional<Realm> currentRealm = Realm.getCurrent(); ②
Realm requiredRealm = Realm.require(); ③

```

- ① Register a **Realm** instance in context using the *classLoader* default scope
- ② Obtain the **Realm** context resource
- ③ Require the **Realm** context resource, throwing an exception if not available

7.2. Authentication credentials

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

- The `Credentials` API to represent credentials data, for example a *secret* access token and the encoding informations related to it.
- The `CredentialsContainer` API to be used as a credentials data holder (for example, the stored account informations related to a *principal* and the credentials provided by a *principal* with an authentication request).
- The `CredentialsMatcher` API to deal with credential validation, to check if two *credentials* representations match.

7.2.1. Credentials creation

The `Credentials` interface provides a *builder* to create and encode a `Credentials` representation, basing on the `String` representation of the *secret* access token (for example, a password).

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

```
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*

7.2.2. Credentials encoding

To encode credentials data, for example for storing purposes, the `Credentials` interface provides an `Encoder` interface, which can be obtained through 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

7.2.3. Credentials container

The `CredentialsContainer` API represents a *credentials* holder, providing the credentials data through the `getCredentials()`.

The credentials data is provided using a generic `Object` type, since the specific credentials representation is highly dependent from the concrete implementations.

The `AuthenticationToken` API is an example of `CredentialsContainer`.

7.2.4. Credentials matching

Credentials matching can be performed using the `CredentialsMatcher` API.

The `CredentialsMatcher` API can be used to compare two credentials representations, providing the respective `CredentialsContainer` instances.

The Holon Platform provides a **default** `CredentialsMatcher` implementation which can be obtained through the `defaultMatcher()` method from the `CredentialsContainer` interface.

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

- Obtain the credentials representations using the `getCredentials()` method of the `Credentials container` representation.

- Try to convert generic Object credentials data into a **byte array**:
 - Supports `char[]`, `String`, `File` and `InputStream` for direct bytes conversion.
 - Supports the `Credentials` type, using the `getSecret()` method to obtain the credentials secret representation.
- If the provided credentials data are of `Credentials` type and an *expiry date* is provided, it checks the credentials are not expired.
- It checks 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.

7.3. Account

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

The `Account` API 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.
- An optional map of generic account **details**, using a `Map` of `String` type detail key and generic `Object` type value.
- An optional set of **permissions** granted to the account, using the `Permission` representation.
- Whether the account is **enabled**.
- Whether the account is **locked**.
- Whether the account is **expired**.

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

```
Account.builder("accountId") ①
    .enabled(true) ②
    .locked(false) ③
    .expired(false) ④
    .credentials(Credentials.builder().secret("pwd").hashAlgorithm(Credentials.
Encoder.HASH_SHA_256)
        .base64Encoded().build()) ⑤
    .root(false) ⑥
    .withPermission(new MyPermission()) ⑦
    .withPermission("role1") ⑧
    .withDetail("name", "TheName").withDetail("surname", "TheSurname") ⑨
    .build();
```

① Create an `Account` with `accountId` as account id

② Set the account as enabled

- ③ Set the account as not locked
- ④ Set the account as not expired
- ⑤ Set the account credentials using the `Credentials` API builder: set `pwd` as secret, hashed with `SHA-256` and encoded using `Base64`
- ⑥ The account is not a `root` account
- ⑦ Add a permission using a custom `MyPermission` type
- ⑧ Add a role type default permission, using `role1` as role name
- ⑨ Set two account details

7.3.1. AccountProvider

The `AccountProvider` API can be used to provide `Account` instances using the `account id`, for example from a data store.

The `AccountProvider` method to obtain an `Account` by id is:

```
Optional<Account> loadAccountById(String id);
```

The method returns an `Optional` value: when empty means that an `Account` with given `id` is not available from the account provider.

```
AccountProvider accountProvider = accountId -> { ①
    if ("test".equals(accountId)) {
        return Optional.of(Account.builder(accountId)
            // configure account
            // ...
            .build());
    }
    return Optional.empty();
};
```

- ① An `AccountProvider` which provides only the `Account` bound to the `test` account id

7.3.2. Account Authenticator

A default `Authenticator` is provided to perform `Account` based authentication, using an `AccountProvider` to access accounts data and the `AccountCredentialsToken` type to represent the authentication request.

The account authenticator strategy is defined as follows:

1. Obtain the authentication request `credentials` using the `Account credentials authentication token` type.
2. Check if an `Account` with the `account id` obtained from the token `getPrincipal()` method is available, using the `AccountProvider` provided at authenticator creation time.

3. If so, checks if the account *credentials* obtained from the token `getCredentials()` method matches with the credentials provided by the loaded `Account` instance.

The account authenticator can be obtained by using the `authenticator(AccountProvider accountProvider)` static method of the `Account` interface.

```
AccountProvider accountProvider = getAccountProvider(); // build or obtain the
AccountProvider to use

Authenticator<AccountCredentialsToken> authenticator = Account.authenticator
(accountProvider); ①
```

- ① Obtain an account authenticator using given `AccountProvider`

The authenticator builder method shown above uses the default `Credentials matching` to perform account credentials checks.

To provide a custom `CredentialsMatcher`, the following creation method can be used:

```
Authenticator<AccountCredentialsToken> authenticator = Account.authenticator
(getAccountProvider(),
    new MyCredentialsMatcher() ①
);
```

- ① Set a custom `MyCredentialsMatcher` as authenticator `CredentialsMatcher`

AccountCredentialsToken

The default account authenticator uses the `AccountCredentialsToken` type to represent account authentication requests.

This token type returns:

- The **account id** (i.e. the *principal* name) as a `String` from the `getPrincipal()` method.
- The account **secret** as `byte[]` from the `getCredentials()` method.

The `Account` interface provides `AccountCredentialsToken` creation methods:

```
AuthenticationToken token = Account.accountCredentialsToken("accountId", "secret"); ①

token = Account.accountCredentialsToken("accountId", new byte[] { 1, 2, 3 }); ②
```

- ① Create an account `AuthenticationToken` providing account id and secret
- ② Create an account `AuthenticationToken` providing account id and secret as an array of bytes

Account authenticator example

Below an example on how to use an account authenticator with a `Realm`.

```

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

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

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: this provider always provide an `Account` instance, setting `pwd` as credentials secret
- ② Register an account `Authenticator` which uses the previously defined `AccountProvider` in the `Realm` instance
- ③ Perform authentication using an *account credentials* authentication token type

7.4. AuthContext

The `AuthContext` API can be used to represent the **current authentication and authorization context**.

The `AuthContext` API:

- Acts as a **holder of the current Authentication**, providing methods to check if an `Authentication` is available and obtain it.
- Provides a method to remove the current context `Authentication`.
- Provides methods to **perform authentication operations**.
- Provides methods for **authorization controls using the current Authentication**.
- As an `AuthenticationNotifier`, **supports AuthenticationListener registration** to be notified when a successfull authentication is performed or when an `Authentication` is not available anymore.

The default `AuthContext` implementation relies on a `Realm` instance to perform concrete authentication and authorization operations, which must be provided at `AuthContext` creation time.



See the `Realm` section for detailed information about `Realm` operations and configuration.

7.4.1. Authentication

The [AuthContext](#) API provides two methods to perform authentication requests:

- `authenticate(AuthenticationToken authenticationToken)` to perform an authentication using an `AuthenticationToken` as authentication request representation.
- `authenticate(Message<?, ?> message, String... schemes)` to perform an authentication a `Message` and optional authentication scheme names.

The authentication process is completely delegated to the backing Realm instance: See the [Realm authentication](#) section and the [MessageAuthenticator](#) section for details about the two authentication strategies, respectively.

7.4.2. Current Authentication

When an authentication request made using the `AuthContext` API is successful, the current `Authentication` reference is made available from the `AuthContext` instance and can be inspected and obtained using the methods of the [AuthenticationInspector](#) API, extended by the `AuthContext` API.

Operation	Returns	Description
<code>isAuthenticated()</code>	<code>true</code> or <code>false</code>	Checks whether an <code>Authentication</code> is currently available
<code>getAuthentication()</code>	<code>Optional<Authentication></code>	Get the current <code>Authentication</code> , if available
<code>requireAuthentication()</code>	<code>Authentication</code>	Get the current <code>Authentication</code> , throwing an <code>IllegalStateException</code> if not available

To remove the current `Authentication`, the `unauthenticate()` method can be used. After this method is called, a new successful authentication request has to be made to make available a new context `Authentication`.

Below

```

AccountProvider provider = id -> Optional.of(Account.builder(id).enabled(true)
    .credentials(Credentials.builder().secret("pwd").base64Encoded().build())
    .withPermission("role1")
    .build()); ①
Realm realm = Realm.builder().withAuthenticator(Account.authenticator(provider))
    .withDefaultAuthorizer()
    .build(); ②

AuthContext context = AuthContext.create(realm); ③

boolean notAlreadyAuthenticated = context.isAuthenticated(); ④

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

Authentication authc = context.requireAuthentication(); ⑥

context.unauthenticate(); ⑦

```

- 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 **Realm** instance
- ④ An **Authentication** is not available from the **AuthContext** since no authentication operation was performed yet
- ⑤ Trigger an authentication request by using an *account credentials* authentication token
- ⑥ If the authentication request is successful, the current **Authentication** is available from the **AuthContext**
- ⑦ Unauthenticate the context, i.e. remove the current **Authentication**

7.4.3. Custom Authentication holder

A custom *authentication holder* can be used at **AuthContext** creation time, to customize the current **Authentication** handling.

To provide a custom current **Authentication** handling logic, the **AuthenticationHolder** interface can be implemented and provided at **AuthContext** creation time.

```

class ThreadLocalAuthenticationHolder implements AuthenticationHolder { ①

    static final ThreadLocal<Authentication> CURRENT_AUTHENTICATION = new ThreadLocal<>
    ();

    @Override
    public Optional<Authentication> getAuthentication() {
        return Optional.ofNullable(CURRENT_AUTHENTICATION.get());
    }

    @Override
    public void setAuthentication(Authentication authentication) {
        CURRENT_AUTHENTICATION.set(authentication);
    }

}

public void customAuthenticationHolder() {
    AuthContext.create(getRealm(), new ThreadLocalAuthenticationHolder()); ②
}

```

① Create an `AuthenticationHolder` that uses a `ThreadLocal` variable to handle the current `Authentication` reference

② Set the `ThreadLocalAuthenticationHolder` as `AuthContext` authentication holder

7.4.4. Authentication listeners

The `AuthContext` API, through the `AuthenticationNotifier` API, supports `AuthenticationListener` registration to be notified when a successful authentication is performed or when an `Authentication` is not available anymore.

When the `AuthContext` is *unauthenticated* using the `unauthenticate()` method, i.e. when the current `Authentication` is removed from the `AuthContext`, a `null Authentication` value is provided to the registered authentication listeners.



The `Realm` and `AuthContext` authentication listeners are considered as separate sets, but since the `AuthContext` uses its configured `Realm` to perform authentications, when the authentication is performed from the `AuthContext`, also the `Realm` authentication listeners will be triggered. Vice-versa, when the authentication is performed from the `Realm`, the `AuthContext` authentication listeners will not be triggered.

7.4.5. AuthContext as a Context resource

When an `AuthContext` is managed using the Holon Platform `Context` architecture, and the default name, i.e. the `AuthContext` class name, is used as context resource key, the `getCurrent()` and `require()` methods can be used to obtain the current `AuthContext` instance from context.


```
final AuthContext authContext = AuthContext.create(getRealm());

Context.get().classLoaderScope().map(scope -> scope.put(AuthContext.CONTEXT_KEY,
authContext)); ①

Optional<AuthContext> currentAuthContext = AuthContext.getCurrent(); ②
AuthContext requiredAuthContext = AuthContext.require(); ③
```

- ① Register an **AuthContext** instance in context using the *classLoader* default scope
- ② Obtain the **AuthContext** context resource
- ③ Require the **AuthContext** context resource, throwing an exception if not available

7.5. @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).

See for example the [Holon Platform JAX-RS module](#) or the [Holon Platform Vaadin module](#) documentation to learn about some **@Authenticate** annotation use cases.

7.6. JWT support

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-auth-jwt</artifactId>
<version>5.2.3</version>
```

The **holon-auth-jwt** artifact provides the support for the **JSON Web Token** standard, providing a full integration with 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.

7.6.1. Configuration

The [JwtConfiguration](#) interface represents the **default JWT configuration provider** for the Holon Platform JWT support APIs.

It makes available a set of methods to obtain the JWT configuration attributes to be used to generate and parse a *JSON Web Token*.

A [JwtConfiguration](#) instance can be obtained in two ways:

1. Using the provided *builder*:

```
JwtConfiguration cfg = JwtConfiguration.builder() ①
    .issuer("MyIssuer") ②
    .expireTime(10000) ③
    .includeDetails(true) ④
    .includePermissions(true) ⑤
    .signatureAlgorithm(JwtSignatureAlgorithm.HS256) ⑥
    .sharedKey(new byte[] { 1, 2, 3 }) ⑦
    .build();
```

- ① Obtain a [JwtConfiguration](#) builder
- ② Set the JWT token *issuer*
- ③ Set the token expire time in milliseconds
- ④ Include the [Authentication](#) details in JWT token generation
- ⑤ Include the [Authentication](#) permissions in JWT token generation
- ⑥ Sign the JWT using [HS256](#) (*HMAC using SHA-256*) as signature algorithm
- ⑦ Set the shared key to use with the symmetric signing algorithm

2. Using a configuration property set:

The [JwtConfigProperties](#) property set can be used to provide the JWT configuration attributes using the standard [Configuration property set](#) API.

The JWT configuration property set is bound to the property name prefix **holon.jwt**.

The JWT configuration properties are currently used by the [JwtTokenBuilder](#) API to create a JWT from an [Authentication](#) and by the [JwtAuthenticator](#) to parse a JWT for authentication purposes.

The available configuration properties are listed here below, also indicating which API use a specific property value:

Name	Type	Default	Used by	Meaning
<i>holon.jwt. issuer</i>	String	<i>None</i>	<i>JwtTokenBuilder</i>	The JWT token issuer. Corresponds to the JWT <i>iss</i> claim.
<i>holon.jwt. signature-algorithm</i>	String	<i>None</i>	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT signature algorithm name: must be one of the algorithm names listed in the JwtSignatureAlgorithm enumeration
<i>holon.jwt. sharedkey-base64</i>	String	<i>None</i>	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT sign shared key, <i>Base64</i> encoded, for symmetric signing algorithms
<i>holon.jwt. privatekey.source</i>	String	<i>None</i>	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT sign private key source for asymmetric signing algorithms. The <i>file:</i> and <i>classpath:</i> prefixes are supported to provide the key using a file or a classpath resource, respectively. The key source is parsed according to the <i>privatekey.format</i> and <i>privatekey.encoding</i> property values.

Name	Type	Default	Used by	Meaning
<i>holon.jwt.</i> privatekey.format	String	PKCS#8	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT sign private key format: must be one of the one of the formats listed in the KeyFormat enumeration. For private keys, the supported format values are PKCS8 , PKCS11 , PKCS12 .
<i>holon.jwt.</i> privatekey.encoding	String	Base64	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT sign private key encoding: must be one of the one of the encodings listed in the KeyEncoding enumeration. The supported encodings are BASE64 , PEM or NONE .
<i>holon.jwt.</i> publickey.source	String	<i>None</i>	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT sign public key source for asymmetric signing algorithms. The file: and classpath: prefixes are supported to provide the key using a file or a classpath resource, respectively. The key source is parsed according to the publickey.format and publickey.encoding property values.

Name	Type	Default	Used by	Meaning
<i>holon.jwt.</i> publickey.format	String	X.509	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT sign public key format: must be one of the one of the formats listed in the KeyFormat enumeration. For public keys, the supported format values are <i>X509</i> , <i>PKCS#11</i> , <i>PKCS#12</i> .
<i>holon.jwt.</i> publickey.encoding	String	Base64	<i>JwtTokenBuilder</i> and <i>JwtAuthenticator</i>	JWT sign public key encoding: must be one of the one of the encodings listed in the KeyEncoding enumeration. The supported encodings are <i>BASE64</i> , <i>PEM</i> or <i>NONE</i> .
<i>holon.jwt.</i> not-before-now	Booelan (true/false)	<i>false</i>	<i>JwtTokenBuilder</i>	Whether to automatically set the <i>nbf</i> (not before) JWT claim to the timestamp at which the token is created.
<i>holon.jwt.</i> expire-ms	Integer number	<i>None</i>	<i>JwtTokenBuilder</i>	JWT token expire time in milliseconds.
<i>holon.jwt.</i> expire-seconds	Integer number	<i>None</i>	<i>JwtTokenBuilder</i>	JWT token expire time in seconds.
<i>holon.jwt.</i> expire-minutes	Integer number	<i>None</i>	<i>JwtTokenBuilder</i>	JWT token expire time in minutes.
<i>holon.jwt.</i> expire-hours	Integer number	<i>None</i>	<i>JwtTokenBuilder</i>	JWT token expire time in hours.
<i>holon.jwt.</i> expire-days	Integer number	<i>None</i>	<i>JwtTokenBuilder</i>	JWT token expire time in days.

Name	Type	Default	Used by	Meaning
<i>holon.jwt.</i> include-details	Booelan (true/false)	false	JwtTokenBuilder	Whether to include Authentication details in the JWT token as <i>claims</i> .
<i>holon.jwt.</i> include-permissions	Booelan (true/false)	false	JwtTokenBuilder	Whether to include Authentication permissions which provides a String representation in the JWT token. A default <i>claim</i> name will be used, with an array of the permission String representations as value.

When the private and/or public key for an asymmetric signing algorithm is provided using a **keystore**, for example when the **PKCS#12** format is used, a set of additional configuration properties are available to configure the key store passwords and alias:

Name	Type	Meaning
<i>holon.jwt.</i> privatekey.store.password	String	The key store password to use when the JWT sign private key is provided using a key store format such as PKCS#12
<i>holon.jwt.</i> privatekey.store.alias	String	The key store key alias to use when the JWT sign private key is provided using a key store format such as PKCS#12
<i>holon.jwt.</i> privatekey.store.alias-password	String	The key store key recovering password to use when the JWT sign private key is provided using a key store format such as PKCS#12
<i>holon.jwt.</i> publickey.store.password	String	The key store password to use when the JWT sign public key is provided using a key store format such as PKCS#12

Name	Type	Meaning
<i>holon.jwt.</i> publickey.store.alias	String	The key store key alias to use when the JWT sign public key is provided using a key store format such as PKCS#12
<i>holon.jwt.</i> publickey.store.alias-password	String	The key store key recovering password to use when the JWT sign public key is provided using a key store format such as PKCS#12

This configuration property set API can be used to build a [JwtConfiguration](#) instance using the `build(JwtConfigProperties properties)` method.

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

① Load the JWT configuration property set from a properties file named `jwt.properties`



See [JwtConfiguration auto-configuration](#) to learn how to rely on the Spring Boot auto-configuration features to automatically create a [JwtConfiguration](#) bean using JWT configuration properties.

7.6.2. Supported JWT signing algorithms

The [JwtSignatureAlgorithm](#) enumeration provides a list of the support JWT signing algorithms.



When a signing shared key is provided in JWT configuration and a signature algorithm is not specified, the **HMAC using SHA-256** default signature algorithm is used.

For asymmetric signing algorithms such **RSA**, the key pair to use can be loaded using the JWT configuration properties from different sources and using different key formats (**X.509**, **PKCS#8**, **PKCS#11**, **PKCS#12**) and encodings (**Base64**, **PEM**).

See the previous section for the list of the available JWT configuration properties.

7.6.3. Building a JWT from an Authentication

The [JwtTokenBuilder](#) API can be used to create JSON Web Tokens using an [Authentication](#) as source and a [JwtConfiguration](#) instance to provide the token configuration attributes.

The [Authentication](#) instance will be used to:

- Set the JWT subject (**sub**) value using the [Authentication principal name](#).

- If configured, include the **Authentication details** as JWT **claims**.
- If configured, include the **Authentication permissions** as JWT **claims**.

The JWT **id** (**jit**) can be specified at token built time.

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

Authentication authc = Authentication.builder("test").build(); ②

String jwt = JwtTokenBuilder.get().buildJwt(configuration, authc); ③
jwt = JwtTokenBuilder.get().buildJwt(configuration, authc, UUID.randomUUID().toString
()); ④
```

- ① Build a **JwtConfiguration** instance using the **jwt.properties** file
- ② Build an **Authentication**
- ③ Build a **JWT** using given configuration and authentication
- ④ Build a **JWT** using given configuration and authentication and set a random id as token id

Authentication permissions claim

When the **JwtConfiguration** method **isIncludePermissions()** returns **true**, the **Authentication permissions** will be included in JWT using the **AuthenticationClaims CLAIM_NAME_PERMISSIONS** claim name.



See **Authentication** for details about the **Authentication** permissions representation.

Only the **Permission** which provide a **String** representation through the **Permission.getPermission()** method will be taken into account, using the permission **String** representation as claim value.

The actual JWT **claim** value will be by a **String array** of the authentication permissions **String** representation.



See **Permission** for details about the permission **String** representation.

```
JwtConfiguration configuration = JwtConfiguration.builder().includePermissions(true)
    .build(); ①

Authentication authc = Authentication.builder("test").withPermission("role1")
    .withPermission("role2").build(); ②

String jwt = JwtTokenBuilder.get().buildJwt(configuration, authc); ③
```

- ① Build a **JwtConfiguration** instance and set to **true** the include permissions switch

- ② Build an **Authentication** with a default permission named **role1** (permission **String** representation) and a default permission named **role2**
- ③ Build a **JWT** using given configuration and authentication: the token will include a **ATH\$prms** claim name with the value **['role1','role2']**

Authentication details claims

When the **JwtConfiguration** method **isIncludeDetails()** returns **true**, the **Authentication details** will be included in JWT as *claims*.

The **Authentication** details are obtained through the **ParameterSet** API, extended by the **Authentication** interface, as a map of **String** value detail keys and generic **Object** detail values.

Each not null **Authentication** detail will be written in JWT using the detail **key** as *claim name* and the detail **value** as *claim value*.

```
JwtConfiguration configuration = JwtConfiguration.builder().includeDetails(true).build(); ①

Authentication authc = Authentication.builder("test").withParameter("name", "John").build(); ②

String jwt = JwtTokenBuilder.get().buildJwt(configuration, authc); ③
```

- ① Build a **JwtConfiguration** instance and set to **true** the include details switch
- ② Build an **Authentication** with a parameter named **name** with value **John**
- ③ Build a **JWT** using given configuration and authentication: the token will include a **name** claim name with the value **John**

7.6.4. Building an Authentication from a JWT

The **JwtTokenParser** API can be used to create an **Authentication** instance from a JSON Web Token value, using a **JwtConfiguration** instance to provide the token configuration attributes.

The JWT is validated before building the **Authentication** instance, and an error is thrown if the validation fails, for example if the token is malformed, expired or the signature is not valid.

The **Authentication** instance is created with the following strategy:

- The JWT subject (**sub**) value is used as **Authentication principal name**.
- If configured, any JWT **claim** is included as an **Authentication detail** parameter, using the claim name as parameter key.
- If configured, the default **ATH\$prms** JWT **claim** is parsed to obtain the corresponding **Authentication permissions**.

The **JwtTokenParser** API returns an **Authentication.Builder** instance, allowing to perform additional **Authentication** configuration before obtaining the actual **Authentication** instance.

```
JwtConfiguration configuration = JwtConfiguration.builder().includeDetails(true)
    .includePermissions(true)
    .build(); ①
```

```
String jwt = getJwt();
```

```
Authentication authc = JwtTokenParser.get().parseJwt(configuration, jwt).build(); ②
```

① Build a **JwtConfiguration** instance and set to include the JWT claims as authentication details and to parse the default **ATH\$prms** JWT claim to obtain the **Authentication permissions**

② Build an **Authentication** from given JWT value

7.6.5. Performing authentication using JWT

The **JwtAuthenticator** interface represents an **Authenticator** to handle **JWT based authentication** and can be registered in a **Realm** to enable JWT authentication capabilities.

A JWT authentication request is represented through a **Bearer authentication token** type, where the **Bearer** value must represent the JWT serialization and it is used by the **JwtAuthenticator** to validate the JWT and provide an **Authentication** obtained from the token.

The **JwtAuthenticator** API relies on a **JWTConfiguration** definition as JWT configuration properties source to consistently parse and validate the JWT.

Additionally, a **JwtAuthenticator** can support:

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

To obtain a **JwtAuthenticator**, the provided *builder* method can be used.

JWT authenticator example

```
JwtConfiguration configuration = JwtConfiguration.builder() ①
    .issuer("MyIssuer") // JWT token issuer
    .expireTime(10000) // 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(JwtSignatureAlgorithm.HS256) // use HS256 as signature
algorithm
    .sharedKey(new byte[] { 1, 2, 3 }) // shared key to use with the symmetric signing
algorithm
    .build();

// JWT authenticator
JwtAuthenticator jwtAuthenticator = JwtAuthenticator.builder().configuration
(configuration) ②
    .issuer("allowedIssuer") ③
    .withRequiredClaim("myClaim") ④
    .build();

// Realm
Realm realm = Realm.builder().withAuthenticator(jwtAuthenticator) ⑤
    .withDefaultAuthorizer().build();

Authentication authc = realm.authenticate(AuthenticationToken.bearer(
"TheJWTtokenHere...")); ⑥

realm = Realm.builder().withAuthenticator(jwtAuthenticator) //
    .withResolver(AuthenticationToken.httpBearerResolver()) ⑦
    .withDefaultAuthorizer().build();

HttpRequest request = obtainHttpRequest();

authc = realm.authenticate(request); ⑧
```

- ① Build a `JwtConfiguration`
- ② Build a `JwtAuthenticator` using the configuration
- ③ Set `allowedIssuer` as allowed JWT issuer
- ④ Set the `myClaim` JWT claim as required
- ⑤ Build a `Realm` and register the `JwtAuthenticator`
- ⑥ Perform an authentication request using a `BearerAuthenticationToken` with the JWT value
- ⑦ Build a `Realm` with a `JwtAuthenticator` and a `Bearer` HTTP message resolver
- ⑧ Perform an authentication request using a `HttpRequest` message: the message must provide a `Bearer Authorization` type message header with the JWT value

JWT to Authentication

The `JwtAuthenticator` API parses the JWT to obtain `Authentication` instance from it, if the token is valid and well formed.

The `Authentication` obtained from the JSON Web Token is created with the following rules:

- The *principal name* is obtained from the JWT *subject* (`sub`) claim and it is required: if not available, an `UnknownAccountException` is thrown.
- The authentication scheme is set to `Bearer`.
- If the default `permissions` claim is found (i.e. a claim named `ATH$perms`, see the `CLAIM_NAME_PERMISSIONS` constant of the `ink:../api/holon-core/com/holonplatform/auth/jwt/AuthenticationClaims.html`[`AuthenticationClaims^`] interface), it is expected to be a String array of *role names*. For each role name, a default `Permission` is created with given name and granted to the `Authentication`.
- Any other JWT claim is setted as an `Authentication` parameter.

See [Authentication](#) for more information about the `Authentication` API.

8. Spring ecosystem integration

The `holon-spring` artifact provides integration with the `Spring` framework and auto-configuration features using `Spring Boot`.

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-spring</artifactId>
<version>5.2.3</version>
```

8.1. Spring beans as Context resources

The `EnableBeanContext` annotation can be used on Spring `Configuration` classes to configure a `Context` scope which uses the Spring `ApplicationContext` to provide **Spring bean definitions** as **context resources**.



See [Context](#) for information about the Holon Platform *Context* architecture.

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

This scope is a **read-only** scope, i.e. it does not allow direct resource registration or removal using the `put` and `remove` context scope methods.

8.1.1. Context resources lookup strategy

The Spring context scope, when a context resource is requested through the Holon Platform `Context` API, checks if a Spring *bean* definition 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 requested **type** is available, this is returned;
- Otherwise, if the `lookupByType()` attribute value of the `@EnableBeanContext` annotation is `true` and a Spring bean of the required type, ignoring its name, is present and *only one candidate* is available, this instance is returned.



The `lookupByType()` attribute of the `@EnableBeanContext` annotation is `true` by default.

The *lookup strategy* can be also configured using the `Environment` configuration property: `holon.context.bean-lookup-by-type`. Set it to `true` to enable the *lookup by type* strategy or `false` to disable it.

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

Spring context scope example

```
@Configuration
@EnableBeanContext ①
class SpringConfig {

    @Bean(name = "testResource") ②
    public TestResource testResource() {
        return new TestResource();
    }

}

public void getContextResource() {
    // lookup by name and type
    Optional<TestResource> resource = Context.get().resource("testResource",
TestResource.class); ③
    // lookup by type
    resource = Context.get().resource(TestResource.class); ④
}
```

① The `@EnableBeanContext` is used on a Spring Configuration class to enable the Spring context scope. The `lookupByType()` attribute is `true` by default.

② `TestResource` type bean definition with the `testResource` name

③ Get a `TestResource` type using the `testResource` key: the `TestResource` Spring bean will be returned

- ④ Get a `TestResource` type without specifying the resource key: since the *lookup by type* strategy is active, the same `TestResource` Spring bean will be returned

8.2. EnvironmentConfigPropertyProvider

The `EnvironmentConfigPropertyProvider` API can be used to build a `ConfigPropertyProvider` which uses the Spring `Environment` structure as property source.

Any property available from the Spring `Environment` will be available, respecting names and value types, from the `ConfigPropertyProvider` instance.



See [Configuration properties](#) for information about configuration property providers.

EnvironmentConfigPropertyProvider example

```
org.springframework.core.env.Environment environment = obtainSpringEnvironment(); ①

// build a ConfigPropertyProvider using Spring Environment as property source
ConfigPropertyProvider provider = EnvironmentConfigPropertyProvider.create(
environment); ②

String value = provider.getProperty("test.property.name", String.class); ③
```

- ① Obtain the Spring `Environment`, for example using dependency injection
- ② Create a `EnvironmentConfigPropertyProvider` using the Spring environment
- ③ Any property available from the Spring environment will be available from the property provider too, with the same name and value type

8.3. 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` API to obtain the current tenant id.

In order for the scope to be active and available, a bean of `TenantResolver` type must be configured and available in the current `BeanFactory` (i.e. in the current Spring `ApplicationContext`).

That `TenantResolver` type bean will be used to obtain the **current tenant id**, if available, using the `getTenantId()` method.

To enable the tenant scope, the `EnableTenantScope` annotation can be used on Spring configuration classes.

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

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

- Or the convenience `ScopeTenant` annotation.

Spring tenant scope example

```
@Configuration
@EnableTenantScope ❶
class TenantScopeConfig {

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

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

- ❶ Use the `@EnableTenantScope` annotation on a Spring Configuration class to enable the *tenant* scope
- ❷ A `TenantResolver` type bean must be provided for current tenant id resolution
- ❸ The convenience `@ScopeTenant` annotation can be used to declare a Spring bean as *tenant scoped*

8.3.1. `TenantResolver` lookup strategy

By default, the *tenant* scope tries to detect a `TenantResolver` bean type in current application context, to be used as current tenant id provider.

If a `TenantResolver` bean type is not available, or more than one `TenantResolver` type bean definition is present, the tenant scope setup will fail, throwing an `ApplicationContextException` at Spring application context startup time.

When more than one `TenantResolver` bean type is present, the `TenantResolver` bean definition to be used with the tenant scope can be configured providing the `TenantResolver` bean **name**. This can be done in two ways:

- Using the `tenantResolver` attribute of the `@EnableTenantScope` annotation.
- Using a Spring `Environment` configuration property named `holon.tenant-scope.tenant-resolver`. This configuration method has precedence on the annotation attribute value.

8.3.2. Tenant scope `ScopedProxyMode`

The default `ScopedProxyMode` of the `@ScopeTenant` annotation is `INTERFACES`.

This proxy mode allows to configure a *proxy* for the tenant-scoped Spring component, in order to

inject (autowire) it in other Spring components with a different scope.

The **INTERFACES** mode create a JDK dynamic proxy implementing all **interfaces** exposed by the class of the target object.

The `proxyMode()` attribute of the **ScopeTenant** annotation can be used to change the scoped proxy mode.

8.3.3. Tenant scoped beans lifecycle

A *tenant scoped* bean instance is created the first time the bean is requested with a specific tenant id.

From now on, the bean instance will survive for the whole Spring application context lifecycle, likewise a *singleton* scoped bean. This because the tenant scope handler it has no way of knowing if a tenant id is no more available and when this will happen, since it is highly dependent from the concrete application architecture and tenant resolution strategy.

To avoid memory wastage and to ensure Spring context cleanliness, the **TenantScopeManager** API can be used to manage tenant scoped beans lifecycle.

When a *tenant id* is not valid or available anymore, the `discardTenantBeanStore(String tenantId)` API method can be invoked to destroy the bean store bound to given *tenant id*, i.e. to remove all the tenant scoped bean instances which refer to the *tenant id*, triggering any associated bean destruction callback.

If the `enableTenantScopeManager` attribute of the `@EnableTenantScope` annotation is set to `true` (the default value), a **TenantScopeManager** bean type is automatically created and registered in the Spring application context. This way, it can be simply obtained, for example, using dependency injection.

```
@Autowired
TenantScopeManager tenantScopeManager;

void discardTenantScopedBeans() {
    tenantScopeManager.discardTenantBeanStore("a_tenant_id"); ①
}
```

① Discard the tenant scoped bean instances for the `a_tenant_id` tenant id using the **TenantScopeManager** API

8.4. Datastore configuration

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



See the **Datastore** section for information about the **Datastore** API.

8.4.1. DatastoreResolver

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

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

8.4.2. DatastoreCommodityFactory

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

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



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

8.4.3. DatastorePostProcessor

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

A Spring bean class implementing this interface is automatically detected and the method `postProcessDatastore(Datastore, String)` is called at `Datastore` bean initialization time.

The `Datastore` bean **instance** and the `Datastore` bean **name** are provided as method parameters. When more than one `Datastore` type bean is present in the Spring application context, the `postProcessDatastore` will be called one time for each available `Datastore` bean.

For example, the post processor can be used to register additional `ExpressionResolver` or `DatastoreCommodityFactory`.



The `DatastorePostProcessor` type beans must be registered using the `singleton` scope.

8.4.4. Automatic Datastore beans configuration using @EnableDatastoreConfiguration

The `EnableDatastoreConfiguration` annotation can be used on Spring configuration classes to **automatically detect and configure** the `Datastore` configuration beans listed above.

When the `@EnableDatastoreConfiguration` is present, the following beans will be auto-detected in Spring context and automatically registered/applied to any `Datastore` type Spring bean:

- `DatastoreResolver` annotated beans to automatically register *expression resolvers*.

- [DatastoreCommodityFactory](#) annotated beans to automatically register *commodity factories*.
- [DatastorePostProcessor](#) type beans to configure a [Datastore](#) bean right after it is initialized in the Spring application context.

```
@Configuration
@EnableDatastoreConfiguration ①
class DatastoreConfig {

    @Bean
    public Datastore datastore() {
        return buildDatastore();
    }
}
```

① Use the [@EnableDatastoreConfiguration](#) to automatically register suitable [Datastore](#) configuration beans into the [Datastore](#) bean definition

8.5. [RestClient](#) implementation using Spring [RestTemplate](#)

The Spring integration module provides a [RESTful client API](#) implementation using the Spring [RestTemplate](#) API.



See the [RESTful client API](#) documentation for information about the [RestClient](#) API.

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

```
RestTemplate restTemplate = getRestTemplate(); ①

RestClient client = SpringRestClient.create(restTemplate); ②
```

① Create or obtain a [RestTemplate](#) implementation

② Create a [RestClient](#) using the [RestTemplate](#) implementation

When a [RestTemplate](#) instance is available as a Holon Platform [Context](#) resource, a [RestClientFactory](#) is automatically registered to provide a [SpringRestClient](#) implementation using that [RestTemplate](#) implementation. This way, the default [RestClient.create\(...\)](#) static methods can be used to obtain a [RestClient](#) implementation.



If the [Spring context scope](#) is enabled with the default beans lookup strategy, it is sufficient that a [RestTemplate](#) bean type is registered in the Spring application context to obtain it as a *context resource*.

```

@Configuration
@EnableBeanContext ①
class Config {

    @Bean ②
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

}

void restclient() {
    RestClient client = RestClient.create(); ③

    client = RestClient.create(SpringRestClient.class.getName()); ④
}

```

- ① Use the `@EnableBeanContext` to enable Spring beans context
- ② Provide a `RestTemplate` bean definition
- ③ The `RestClient.create()` method can be used to obtain a `RestClient` implementation backed by the defined `RestTemplate` bean definition
- ④ If more than one `RestClientFactory` is available, the `SpringRestClient` class name can be specified to ensure that a `SpringRestClient` type is obtained as a `RestClient` implementation

8.6. 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 the core Spring Boot auto-configuration capabilities, the following artifact must be included in your project dependencies:

```

<groupId>com.holon-platform.core</groupId>
<artifactId>holon-spring-boot</artifactId>
<version>5.2.3</version>

```

See below for the available auto-configuration features.

8.6.1. Spring context scope auto-configuration

The **Spring context scope** is automatically enabled.

This has the same effect as using the `@EnableBeanContext` annotation on Spring configuration classes, with the `lookupByType()` attribute set as true by default.

The `holon.context.bean-lookup-by-type` configuration property name can be used to configure the *resource lookup strategy*, enabling or not the *lookup by type* mode.



See the [Spring beans as Context resources](#) documentation section for details.

To disable this auto-configuration feature, the [EnableBeanContextAutoConfiguration](#) class can be excluded from the Spring Boot auto-configuration classes:

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

8.6.2. Spring tenant scope auto-configuration

If a [TenantResolver](#) type bean is available in Spring application context and a *single candidate* can be determined, the [Spring tenant scope](#) is automatically registered and enabled.

This has the same effect as using the [@EnableTenantScope](#) annotation on Spring configuration classes.



See the [Spring tenant scope](#) documentation section for details about the *tenant* scope.

To disable this auto-configuration feature, the [TenantScopeAutoConfiguration](#) class can be excluded from the Spring Boot auto-configuration classes:

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

8.6.3. JwtConfiguration auto-configuration

When at least one property of the `holon.jwt.` property set is available in Spring environment, and a [JwtConfiguration](#) type bean is not already present in the Spring application context, a [JwtConfiguration](#) bean is automatically created and setted up according to the `holon.jwt.` configuration property values.

See the [JWT configuration](#) section for details about the available JWT configuration properties.

For example, to setup a [JwtConfiguration](#) bean using a symmetric signing algorithm, the following configuration properties can be provided using the YAML format:

```
holon:
  jwt:
    issuer: example-issuer
    signature-algorithm: HS512
    sharedkey-base64: "eWGZLlCrUjtBZwxgzcLPnA"
    expire-hours: 1
```

Or when an asymmetric signing algorithm is used:

```

holon:
  jwt:
    issuer: example-issuer
    expire-hours: 1
    signature-algorithm: RS256
    privatekey:
      encoding: PEM
      source: >
        -----BEGIN RSA PRIVATE KEY-----

        MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJeAgEAAoGBALv07pB1uFK4fQ3Q1HcRSCofMWovYpYp
        h02Jh31h2hIivC3TbFzWn07pL14d8ec8LIoIYWZAn4L9ZUpEzCPr3nbHpdoPaEcrpqXlgpj0/Jgf8Ysa
        QPWq7ArjWr/ifiORA3vRg20VhEGD309BccYh9peh/I0pt9EfmWioYlId0+S/AgMBAECgYB1zoY8y1w1
        l0bk4sg7fPyDUjvRt100lQV5MQtYPh3F4jmaa3rvEaKWfjevQQufCKtN9QS/Z1/TZWm4TDi7hxp0u6YZ
        gVL9JYHw0vb8opX9Yle9FyLRv4pPdHUKhs7ahz mhPPAf0kSjwKAYlqBmTUzZY5HTRZy/ffpVftPwc150
        mQJBAOJAromanqe6PDpxnL4IGcPPyn0dWQ3VyTV+i1XkZ8d60nBoLUriG80k+ehj4eiEYeK4Ca7GPciM
        EqkZc54XrjsCQQUduq0TRB3V+1mVjJtMixN4I1nb5lo2MVASDjvL/3LCv7LxCZErWLPcJpivMrii+00Ar
        k1VenXV7uTLD/Si9HKdNAKEAlUbn4ZJKq4+MvWLIb/kYRsGKcBI095PeNZVQiVMxxc0bpN6XQ5j7iJII
        8PM10hvGGbgja1UQ3ojMpxVL2zr0kQJBANQt1Ejgsj9L1HfqQnjMBek3Zph5ttus75v6R79k8Bxfqyxq
        N6gdaT0VSEm78PZodG/FXUU6v/4ith2INN8I+XkCQCq77unFpv30ESzhNRa0hjJgAAiwwAqwrWRxLHT
        DijzpQ4PNDfR32bTV/pB9i0nJAPce+9cB7ahx+vpLX2jFuLu
        -----END RSA PRIVATE KEY-----

    publickey:
      encoding: PEM
      source: >
        -----BEGIN RSA PUBLIC KEY-----

        MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC7906QdbhSuH0N0JR3EUgqHzFqL2KWKYTtiYd9YdoS
        Irwt02xc1p906S9eHfHnPCyKCGFmQJ+C/WVKRMwj6952x6XaD2hHK6a15YKYzvyYH/GLGkd1quwK41q/
        4n4jkQN70YNjLYRBg99PQXHGIfaXofyDqbfRH5loqGJYnTvkvwIDAQAB
        -----END RSA PUBLIC KEY-----

```

To disable this auto-configuration feature, the [JwtAutoConfiguration](#) class can be excluded from the Spring Boot auto-configuration classes:

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

8.6.4. 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>5.2.3</version>
```

The starter provides the dependency to the **holon-spring-boot** artifact, enabling the auto-configuration features listed above, in addition to the default **core** Spring Boot starter (**spring-boot-starter**), which provides auto-configuration support, logging and YAML support.

See the [Spring Boot starters documentation](#) for details about the Spring Boot *starters* topic and the core Spring Boot starter features.

8.7. Spring Security integration

Maven coordinates:

```
<groupId>com.holon-platform.core</groupId>
<artifactId>holon-spring-security</artifactId>
<version>5.2.3</version>
```

The **holon-spring-security** artifact provides integration between the Holon Platform authentication and authorization architecture and the Spring Security components and APIs.



See the [Authentication and Authorization](#) for information about the the Holon Platform authentication and authorization architecture.

The main entry point to deal with Spring Security integration is the [SpringSecurity](#) API interface, which provides operations to:

- Create an **AuthContext** backed by a Spring Security [SecurityContext](#) and using a [AuthenticationManager](#) to perform authentication operations.
- Seamlessly use a Spring Security [Authentication](#) as a Holon platform [Authentication](#) representation.
- Perform authentication in Spring Security context using standard Holon platform [AuthenticationToken](#), through the [SpringSecurityAuthenticationToken](#) bridge interface.
- Build a Spring Security [AuthenticationProvider](#) using a standard Holon platform [Authenticator](#).
- Provide adapters and conversion methods from Spring Security [GrantedAuthority](#) and standard Holon platform [Permission](#).

8.7.1. AuthContext API integration

The Holon platform `AuthContext` API can be integrated with the Spring Security context to use the `SecurityContext` as current authentication holder.

The `authContext()` and `authContext(Realm realm)` methods of the `SpringSecurity` integration API allows to create an `AuthContext` which uses the Spring Security `SecurityContext` as authentication holder.

This establishes a link between the Holon platform authentication `Realm` and the Spring security authentication context, using the `AuthContext` API as bridge between the two architectures.

When a successful authentication operation is performed using Spring Security, and accordingly the current authenticated principal is available from the `SecurityContext`, the `AuthContext` API will provide the same authenticated principal, represented as an `Authentication`. The current `Authentication` is an adapter of the concrete Spring Security authenticated principal, providing the principal name, details and authorizations as Holon platform `Permission` representations.

This way, the `AuthContext` API can be seamlessly used to inspect authenticated principal attributes and to perform authorization controls using the Holon Platform conventions.

```
AuthContext authContext = SpringSecurity.authContext(); ①

UsernamePasswordAuthenticationToken tkn = new UsernamePasswordAuthenticationToken(
    "user", "pwd",
    Arrays.asList(new GrantedAuthority[] { new SimpleGrantedAuthority("role1") }));
SecurityContextHolder.getContext().setAuthentication(tkn); ②

Authentication authc = authContext.requireAuthentication(); ③

String name = authc.getName(); ④

boolean permitted = authContext.isPermitted("role1"); ⑤

SecurityContextHolder.getContext().setAuthentication(null); ⑥
boolean notAnyMore = authContext.isAuthenticated();
```

- ① Obtain an `AuthContext` API bound to a default `Realm` and which uses the Spring Security `SecurityContext` as authentication holder
- ② Simulate an authentication operation in the Spring Security context, using a `UsernamePasswordAuthenticationToken` with `user` as principal name and `role1` as granted authority
- ③ The `AuthContext` is now authenticated accordingly, and provided the current authenticated principal as a `com.holonplatform.auth.Authentication`
- ④ The provided `Authentication` is an adapter of the Spring Security one, with matching principal attributes and permissions. The returned principal name will be `user`
- ⑤ The `AuthContext` API can be used to perform authorization controls, using the granted authorities of the Spring Security authenticated principal: since the `role1` authority name was granted to the principal, the `isPermitted("role1")` call will return `true`

- ⑥ When the authenticated principal is removed from the Spring Security context, the bound `AuthContext` won't provide an authentication anymore accordingly

When a `Realm` is provided at `AuthContext` build time, it can be configured with a set of `Authenticator` and `Authorizer` according to the `Realm` API definition.

If an authentication operation is performed using the `AuthContext` API, the backing `Realm` is used for the actual authentication strategy, through the registered authorizers. When an authentication operation is successful, the authentication result is setted back in the Spring Security `SecurityContext` as current authenticated principal, with the same principal attributes and permissions, represented as granted authorities.

```
final Realm realm = Realm.builder().withDefaultAuthorizer().withAuthenticator(Account
    .authenticator(id -> { ①
        if ("usr".equals(id)) {
            return Optional.of(Account.builder(id).credentials(Credentials.builder().secret(
                "pwd").build())
                .withPermission("role1").build());
        }
        return Optional.empty();
    })).build();
```

```
AuthContext authContext = SpringSecurity.authContext(realm); ②
```

```
authContext.authenticate(Account.accountCredentialsToken("usr", "pwd")); ③
```

```
org.springframework.security.core.Authentication authc = SecurityContextHolder
    .getContext().getAuthentication(); ④
```

```
String name = authc.getName(); ⑤
```

```
Collection<? extends GrantedAuthority> authorities = authc.getAuthorities(); ⑥
```

- ① Create a `Realm` with default authorizer and a `Account` authenticator with given accounts provider (in this case, a single `Account` named `usr`, with password `pwd` and `role1` as permission)
- ② Obtain an `AuthContext` API bound to the specified `Realm` and which uses the Spring Security `SecurityContext` as authentication holder
- ③ Perform authentication using the `AuthContext` API, providing an account credentials `AuthenticationToken`
- ④ If the authentication is successful, the authentication result is setted back in the Spring Security `SecurityContext` as current authenticated principal
- ⑤ The principal name will be `usr`
- ⑥ The granted authorities will include a `role1` authority

8.7.2. Use an `AuthenticationManager` as a `Realm` `Authenticator`

The Spring Security `AuthenticationManager` can be used as an `Authenticator` API, which can be registered in a `Realm` to provide authentication capabilities using Spring Security authentication

tokens.

This allows to integrate Spring Security authentication architecture with the Holon Platform authentication architecture, and to use a `Realm` (and accordingly the `AuthContext` API) to perform authentication operations using the Holon Platform conventions and APIs.

An `Authenticator` can be obtained from a Spring Security `AuthenticationManager` using the `SpringSecurity` API:

```
AuthenticationManager authenticationManager = getAuthenticationManager(); ①  
  
Authenticator<SpringSecurityAuthenticationToken> authenticator = SpringSecurity  
    .authenticator(authenticationManager); ②
```

① Obtain the Spring Security `AuthenticationManager` (for example using Spring's dependency injection)

② Create an Holon Platform `Authenticator` using the Spring Security `AuthenticationManager`

The `Authenticator` is bound to a `SpringSecurityAuthenticationToken` authentication token type, which must be used to provided the authentication credentials.

A `SpringSecurityAuthenticationToken` instance can be built from a Spring Security `Authentication` instance, either using the `SpringSecurity` API or the `SpringSecurityAuthenticationToken` interface itself.

```
SpringSecurityAuthenticationToken token = SpringSecurityAuthenticationToken  
    .create(new UsernamePasswordAuthenticationToken("usr", "pwd")); ①  
  
token = SpringSecurity.asAuthenticationToken(new UsernamePasswordAuthenticationToken(  
    "usr", "pwd")); ②
```

① Create a `SpringSecurityAuthenticationToken` from given Spring Security `UsernamePasswordAuthenticationToken`

② The same operation using the `SpringSecurity` API

An `account(String accountId, String secret)` convenience method is provided to create Spring Security `UsernamePasswordAuthenticationToken` token type.

The `Authenticator` created in this way can be registered in a `Realm`, thus providing authentication capabilities using the Spring Security context.

```
AuthenticationManager authenticationManager = getAuthenticationManager(); ①

Realm realm = Realm.builder().withDefaultAuthorizer()
    .withAuthenticator(SpringSecurity.authenticator(authenticationManager)) ②
    .build();

Authentication authc = realm.authenticate(SpringSecurityAuthenticationToken.account(
    "user", "pwd1")); ③
```

- ① Obtain the Spring Security **AuthenticationManager** (for example using Spring's dependency injection)
- ② Create a **Realm** and add an *authenticator* using the Spring Security **AuthenticationManager**
- ③ Perform authentication using a username/password type authentication token

8.7.3. Create a fully integrated **AuthContext** API

According to the two previous sections, the **AuthContext** can be fully integrated with the Spring Security context, that is:

- Use the Spring Security context (through the **SecurityContext** API) as the current authentication holder.
- Use the Spring Security **AuthenticationManager** as authenticator to perform authentication operations using the Spring Security environment.

Of course, you can mix the **AuthenticationManager** based authentication with any other **Authenticator** registered in the **Realm** to which the **AuthContext** is bound.

The **SpringSecurity** API provides methods to easily create **AuthContext** instances which uses the Spring Security context as the current authentication holder and backed by a **Realm** with **AuthenticationManager** based authentication capabilities.

```
AuthenticationManager authenticationManager = getAuthenticationManager(); ①

AuthContext authContext = SpringSecurity.authContext(authenticationManager); ②

authContext = SpringSecurity.authContext(authenticationManager, true); ③
```

- ① Obtain the Spring Security **AuthenticationManager** (for example using Spring's dependency injection)
- ② Create an **AuthContext** instance which uses the Spring Security context as the current authentication holder and backed by a **Realm** with **AuthenticationManager** based *authenticator*
- ③ With this method, is also registered an **Authenticator** for the default **AccountCredentialsToken** which uses the Spring Security **AuthenticationManager** to perform the authentication operations.

8.7.4. Use a Holon Authenticator as Spring Security AuthenticationProvider

A default Holon `Authenticator` can be also adapted to be used as a Spring Security `AuthenticationProvider`, which can be used to implement new authentication strategies in the Spring Security environment.

This way, any Holon platform `Authenticator` (either builtin or custom) can be used as a Spring Security authentication processor.

The `authenticationProvider(Authenticator<T> authenticator, Class<A> authenticationType, Function<A, T> converter)` method of the `SpringSecurity` API can be used for this purpose, providing the *function* to be used to convert a Spring Security `Authentication` into the Holon platform `AuthenticationToken` type which will be used as the `Authenticator` authentication credentials.

```

@Configuration
@EnableGlobalAuthentication
class Config {

    @Bean
    public AccountProvider accountProvider() { ①
        return id -> {
            if ("usr1".equals(id)) {
                return Optional.of(Account.builder(id).credentials(Credentials.builder()
                    .secret("pwd1").build())
                    .withPermission("view").build());
            }
            if ("usr2".equals(id)) {
                return Optional.of(Account.builder(id).credentials(Credentials.builder()
                    .secret("pwd2").build())
                    .withPermission("view").withPermission("manage").build());
            }
            return Optional.empty();
        };
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationManagerBuilder
auth,
        AccountProvider accountProvider) throws Exception { ②
        return auth.authenticationProvider( ③
            SpringSecurity.authenticationProvider(Account.authenticator(accountProvider),
④
                UsernamePasswordAuthenticationToken.class, ⑤
                upt -> AccountCredentialsToken.create(upt.getPrincipal().toString(),
                    upt.getCredentials().toString()) ⑥
            ).build();
        }
    }
}

```

- ① Create an **AccountProvider** bean which provides account infos to be used by the Holon Platform **Account** authenticator
- ② Create an **AuthenticationManager** Spring Security bean, to which the adapted **Authenticator** will be added
- ③ Add a Spring Security **AuthenticationProvider** using a Holon Platform **Authenticator**
- ④ Create an **Account** type **Authenticator**
- ⑤ Use the **UsernamePasswordAuthenticationToken** class as Spring Security authentication token type
- ⑥ Provide the function to be used to convert an **UsernamePasswordAuthenticationToken** into the **AccountCredentialsToken** token type required by the authenticator

8.7.5. Permissions and authorizations

The [SpringSecurity](#) API provides methods to seamlessly use a Holon platform [Permission](#) as a Spring Security [GrantedAuthority](#) and vice-versa.

The [String](#) representation of the two types is guaranteed to be preserved, i.e. the value returned from the [getPermission\(\)](#) method of a [Permission](#) obtained from a [GrantedAuthority](#) will be the same returned by the [getAuthority\(\)](#) method and vice-versa.

```
GrantedAuthority ga = new SimpleGrantedAuthority("role1");
Permission permission = SpringSecurity.asPermission(ga); ①

Permission p = Permission.create("role2");
GrantedAuthority grantedAuthority = SpringSecurity.asAuthority(p); ②
```

① Create a [Permission](#) from a [GrantedAuthority](#)

② Create a [GrantedAuthority](#) from a [Permission](#)

8.7.6. Spring Security starter

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-security</artifactId>
<version>5.2.3</version>
```

This *starter* includes the base Holon Platform starter (see the [Spring Boot auto-configuration](#) section), the standard Spring Security starter ([spring-boot-starter-security](#)) and the [holon-spring-security](#) dependency, to provide the Holon Platform Spring Security integration components and APIs.

9. 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

10. System requirements

10.1. Java

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