Annotations

Table of Contents

1. Annotations 1
   1.1. Other Guides 1
   1.2. Examples 1
2. Summary 3
   2.1. Core annotations 3
   2.2. Other Isis Annotations 4
   2.3. JDO Annotations 4
   2.4. Java EE Annotations 6
   2.5. Deprecated Annotations 6
   2.6. Incomplete/partial support 12
3. `@Action` 15
   3.1. `associateWith()` 17
   3.2. Command Persistence and Processing 18
   3.3. domainEvent() 23
   3.4. `hidden()` 27
   3.5. `invokeOn()` 27
   3.6. `publishing()` 28
   3.7. `restrictTo()` 29
   3.8. `semantics()` 30
   3.9. `typeOf()` 32
4. `@ActionLayout` 33
   4.1. `bookmarking()` 34
   4.2. `contributedAs()` 36
   4.3. `cssClass()` 37
   4.4. `cssClassFa()` 37
   4.5. `describedAs()` 38
   4.6. `hidden()` 39
   4.7. `named()` 40
   4.8. `position()` 40
5. `@Collection` 43
   5.1. `domainEvent()` 44
   5.2. `editing()` 47
   5.3. `hidden()` 48
   5.4. `notPersisted()` 49
   5.5. `typeOf()` 50
6. `@CollectionLayout` 51
   6.1. `cssClass()` 52
   6.2. `defaultView()` 53

6.3. describedAs() . . . 53
6.4. hidden() . . . 54
6.5. named() . . . 55
6.6. paged() . . . 56
6.7. render() . . . 57
6.8. sortedBy() . . . 57
7. @Column (javax.jdo) . . . 60
7.1. Nullability . . . 60
7.2. Length for Strings . . . 60
7.3. Length/scale for BigDecimals . . . 61
7.4. Hints and Tips . . . 61
7.5. Mapping Blobs and Clobs . . . 62
8. @Digits (javax) . . . 64
9. @Discriminator (javax.jdo) . . . 65
9.1. Examples . . . 65
9.2. Precedence . . . 65
10. @DomainObject . . . 67
10.1. auditing() . . . 69
10.2. autoCompleteRepository() . . . 69
10.3. bounded() . . . 71
10.4. createdLifecycleEvent() . . . 72
10.5. editing() . . . 73
10.6. loadedLifecycleEvent() . . . 74
10.7. mixinMethod() . . . 76
10.8. nature() . . . 77
10.9. persistedLifecycleEvent() . . . 79
10.10. persistingLifecycleEvent() . . . 80
10.11. objectType() . . . 82
10.12. publishing() . . . 83
10.13. removingLifecycleEvent() . . . 84
10.14. updatingLifecycleEvent() . . . 86
10.15. updatedLifecycleEvent() . . . 88
11. @DomainObjectLayout . . . 91
11.1. bookmarking() . . . 92
11.2. cssClass() . . . 94
11.3. cssClassFa() . . . 94
11.4. cssClassUiEvent() . . . 95
11.5. describedAs() . . . 97
11.6. iconUiEvent() . . . 97
11.7. named() . . . 99
11.8. paged() . . . 100

11.9. plural() 100
11.10. titleUiEvent() 101
12. @DomainService 104
12.1. nature() 104
12.2. objectType() 106
12.3. repositoryFor() 107
13. @DomainServiceLayout 108
13.1. menuBar() 108
13.2. menuOrder() 111
13.3. named() 112
14. @Facets 114
15. @HomePage 115
16. @Inject (javax) 117
16.1. Alternative syntaxes 117
16.2. Injecting collection of services 118
16.3. Manually injecting services 118
17. @MemberGroupLayout 120
18. @MemberOrder 121
19. @Mixin 122
19.1. method() 122
20. @Nullable (javax) 124
21. @NotPersistent (javax.jdo) 125
22. @MinLength 126
23. @Parameter 127
23.1. fileAccept() 128
23.2. maxLength() 129
23.3. mustSatisfy() 129
23.4. optionality() 130
23.5. regexPattern() 131
24. @ParameterLayout 133
24.1. cssClass() 134
24.2. describedAs() 134
24.3. labelPosition() 135
24.4. multiLine() 136
24.5. named() 136
24.6. renderedAsDayBefore() 137
24.7. typicalLength() 138
25. @PersistenceCapable (javax.jdo) 139
25.1. Examples 139
25.2. Precedence 140
26. @PostConstruct (javax) 141

27. @PreDestroy (javax) . . . 143
28. @PrimaryKey (javax.jdo) . . . 144
29. @Programmatic . . . 145
30. @Property . . . 146
30.1. Command Persistence and Processing . . . 148
30.2. domainEvent() . . . 152
30.3. editing() . . . 155
30.4. fileAccept() . . . 155
30.5. hidden() . . . 156
30.6. maxLength() . . . 157
30.7. mustSatisfy() . . . 158
30.8. notPersisted() . . . 159
30.9. optionality() . . . 160
30.10. regexPattern() . . . 163
31. @PropertyLayout . . . 164
31.1. cssClass() . . . 165
31.2. describedAs() . . . 166
31.3. labelPosition() . . . 166
31.4. multiLine() . . . 168
31.5. named() . . . 169
31.6. promptStyle() . . . 170
31.7. renderedAsDayBefore() . . . 170
31.8. typicalLength() . . . 171
31.9. unchanging() . . . 172
32. @RequestScoped (javax) . . . 173
33. @Title . . . 174
33.1. Lombok support . . . 174
34. @ViewModel . . . 176
35. @ViewModelLayout . . . 178
35.1. cssClass() . . . 179
35.2. cssClassFa() . . . 179
35.3. describedAs() . . . 180
35.4. named() . . . 180
35.5. paged() . . . 181
35.6. plural() . . . 181
36. @XmlJavaTypeAdapter (jaxb) . . . 182
37. @XmlRootElement (jaxb) . . . 183
37.1. Example . . . 183
37.2. See also . . . 184

Chapter 1. Annotations

This guide describes the various annotations used by Apache Isis to provide additional metadata about the domain objects. Most of these are defined by Isis itself, but some are from other libraries. It also identifies a number of annotations that are now deprecated, and indicates their replacement.

1.1. Other Guides

Apache Isis documentation is broken out into a number of user, reference and "supporting procedures" guides.

The user guides available are:

- Fundamentals
- Wicket viewer
- Restful Objects viewer
- DataNucleus object store
- Security
- Testing
- Beyond the Basics

The reference guides are:

- Annotations (this guide)
- Domain Services
- Configuration Properties
- Classes, Methods and Schema
- Apache Isis Maven plugin
- Framework Internal Services

The remaining guides are:

- Developers' Guide (how to set up a development environment for Apache Isis and contribute back to the project)
- Committers' Guide (release procedures and related practices)

1.2. Examples

To give just a few examples of annotations supported by Apache Isis:

- if a property is read-only, then this can be annotated with `@Property(editing=EditingDISABLED)`.
- if a class has a small fixed set of instances (eg a picklist), then it can be annotated using

`@DomainObject(bounded=true)`

- if a class is a domain service and should be automatically instantiated as a singleton, then it can be annotated using `@DomainService`
- if an action is idempotent, then it can be annotated using `@Action(semantics=SemanticsOf.IDEMPOTENT)`.
- if an action parameter is optional, it can be annotated using `@Parameter(optionality=Optionality.OPTIONAL)`

Some annotations act as UI hints, for example:

- if a collection should be rendered "open" rather than collapsed, it can be annotated using `@CollectionLayout(render=RenderType.EAGERLY)`
- if an action has a tooltip, it can be annotated using `@ActionLayout(describedAs=…)`
- if a domain object is bookmarkable, it can be annotated using `@DomainObjectLayout(bookmarking=BookmarkPolicy.AS_ROOT`).

Chapter 2. Summary

This section summarizes the various annotations supported by Apache Isis. They break out into five categories.

2.1. Core annotations

In Apache Isis every domain object is either a domain entity, a view model or a domain service. And each of these are made up of properties, collections and actions (domain services only have actions).

For each of these domain types and members there are two annotations. One covers the semantics intrinsic to the domain (eg whether an action parameter is optional or not), then other (suffix `…Layout`) captures semantics relating to the UI/presentation layer.

Most UI semantics can also be specified using dynamic object layout.

The table below summarizes these most commonly used annotations in Apache Isis.

Table 1. Core annotations for domain objects, services and members

Annotation	Purpose	Layer	File-based layout?
`@Action`	Domain semantics for actions	Domain	
`@ActionLayout`	User interface hints for actions	UI	Yes
`@Collection`	Domain semantics for collections	Domain	
`@CollectionLayout`	User interface hints for collections	UI	Yes
`@DomainObject`	Domain semantics for domain object (entities and optionally view models, see also `@ViewModel`)	Domain	
`@DomainObjectLayout`	User interface hints for domain object (entities and optionally view models, see also `@ViewModelLayout`)	UI	Yes
`@DomainService`	Class is a domain service (rather than an entity or view model)	Domain	
`@DomainServiceLayout`	User interface hints for domain services	UI	
`@Parameter`	Domain semantics for action parameters	Domain	
`@ParameterLayout`	Layout hints for an action parameter (currently: its label position either to top or the left).	UI	Yes
`@Property`	Domain semantics for properties	Domain	
`@PropertyLayout`	Layout hints for a property	UI	Yes

Annotation	Purpose	Layer	File-based layout?
`@ViewModel`	Specify that a class is a view model (as opposed to an entity or domain service); equivalent to `@DomainObject(nature`=VIEW_MODEL).	Domain, Persistence	
`@ViewModelLayout`	User interface hints for view models. For use with `@ViewModel`. If specifying view models using `@DomainObject(nature=VIEW_MODEL)` then use `@DomainObjectLayout`)	UI	Yes

2.2. Other Isis Annotations

These annotations are also commonly used, but relate *not* to objects or object members but instead to other aspects of the Apache Isis metamodel.

Table 2. Other Isis Annotations

Annotation	Purpose	Layer	File-based layout?
`@Facets`	Install arbitrary facets within the Apache Isis metamodel.	(any)	
`@HomePage`	Query-only action (on domain service) to be invoked, result of which is rendered as the user's home page.	UI	
`@MemberOrder`	Ordering of properties, collections and actions, and also associating actions with either a property or a collection.	UI	Yes
`@MinLength`	Minimum number of characters required for an auto-complete search argument.	UI	
`@Programmatic`	Ignore a public method, excluded from the Apache Isis metamodel.	Domain	

2.3. JDO Annotations

Apache Isis uses JDO/DataNucleus as its ORM, and infers some of its own metadata from the JDO annotations.

Isis (currently) builds up metadata by parsing the JDO annotations from source, *not* by querying the JDO metamodel. The upshot is that, for the annotations documented here at least, your domain entities must use JDO annotations rather than XML.

Furthermore, note that although JDO (the property-related) annotations to be placed on either the field or on the getter, Apache Isis requires that annotations are placed on the getter.

The table below lists the JDO annotations currently recognized by Apache Isis.

Table 3. JDO Annotations

Annotation	Purpose	Layer	Applies to
`@javax.jdo.annotations.Column`	Used to determine whether a property is mandatory or optional. For `String` and `BigDecimal` properties, used to determine length/precision/scale.	Domain / persistence	Property
`@javax.jdo.annotations.Discriminator`	Override for the object type, as used in `Bookmark`s, URLs for `RestfulObjects viewer and elsewhere. Note that the discriminator overrides the object type that may otherwise be inferred from the @PersistenceCapable` annotation.	Domain / persistence	Class
`@javax.jdo.annotations.NotPersistent`	Used to determine whether to enforce or skip some metamodel validation for `@Column` versus equivalent Isis annotations.	Domain / persistence	Property
`@javax.jdo.annotations.PersistenceCapable`	Used to build Apache Isis' own internal identifier for objects. If the <code>schema()</code> attribute is specified (and if <code>@Discriminator</code> _hasn’t been specified), is also used to derive the object type, as used in `Bookmark`s, URLs for RestfulObjects viewer and elsewhere.	Domain / persistence	Class
`@javax.jdo.annotations.PrimaryKey`	Used to ensure Apache Isis does not overwrite application-defined primary keys, and to ensure is read-only in the UI.	Domain / persistence	Property

Isis also parses the following JDO annotations, but the metadata is currently unused.

Table 4. JDO Annotations (unused within Apache Isis)

Annotation	Purpose	Layer	Applies to
`@javax.jdo.annotations.DataStoreIdentity`	Unused	Persistence	Class
`@javax.jdo.annotations.EmbeddedOnly`	Unused	Persistence	Class
`@javax.jdo.annotations.Query`	Unused	Persistence	Class

2.4. Java EE Annotations

While Apache Isis does, as of today, define a good number of its own annotations, the policy is to reuse standard Java/JEE annotations wherever they exist or are added to the Java platform.

The table below lists the JEE annotations currently recognized. Expect to see more added in future releases of Apache Isis.

Table 5. Java EE Annotations

Annotation	Purpose	Layer	File-based layout?
`@javax.validation.constraints.Digits`	Precision/scale for BigDecimal values.	Domain	
`@javax.inject.Inject`	Inject domain service into a domain object (entity or view model) or another domain service.	Domain	
`@javax.annotation.Nullable`	Specify that a property/parameter is optional.	Domain	
`@javax.annotation.PostConstruct`	Callback for domain services (either singleton or request-scoped) to initialize themselves once instantiated.	Domain	
`@javax.annotation.PreDestroy`	Callback for domain services (either singleton or request-scoped) to clean up resources prior to destruction.	Domain	
`@javax.enterprise.context.RequestScoped`	Specify that a domain service has request-scope (rather than a singleton).	Domain	
`javax.xml.bind.annotation XmlRootElement`	JAXB annotation indicating the XML root element when serialized to XML; also used by the framework for view models (whose memento is the XML), often also acting as a DTO.	Application	
`javax.xml.bind.annotation XmlJavaTypeAdapter`	JAXB annotation defining how to serialize an entity. Used in conjunction with the (framework provided) `PersistentEntityAdapter` class to serialize persistent entities into a canonical OID (equivalent to the `Bookmark` provided by the `BookmarkService`).	Domain	

2.5. Deprecated Annotations

As Apache Isis has evolved and grown, we found ourselves adding more and more annotations; but most of these related to either an object type (entity, view model, service) or an object member (property, collection, action). Over time it became harder and harder for end programmers to discover these new features.

Accordingly, (in v1.8.0) we decided to unify the semantics into the main (core) annotations listed above.

The annotations listed in the table below are still supported by Apache Isis, but will be retired in Apache Isis v2.0.

Table 6. Deprecated Annotations

Annotation	Purpose	Use instead	Layer	File-based layout?
`@ActionOrder`	Order of buttons and menu items representing actions.	`@MemberOrder`	UI	Yes
`@ActionInteraction`	Enable subscribers on the Event Bus Service to either veto, validate or take further steps before/after an action has been invoked.	`@Action#domainEvent()`	Domain	
`@ActionSemantics`	Query-only, idempotent or non-idempotent.	`@Action#semantics()`	Domain	
`@Audited`	Audit changes to an object.	`@DomainObject#auditing()`	Domain	
`@AutoComplete`	Repository method to search for entities	`@DomainObject#autoCompleteRepository()`	UI/Domain	
`@Bookmarkable`	Whether (and how) to create a bookmark for visited object.	`@DomainObjectLayout#bookmarking()`	UI	
`@Bounded`	Bounded (and limited) number of instances of an entity type, translates into a drop-down for any property of that type.	`@DomainObject#bounded()`	Domain	
`@Bulk`	Indicates an action is a bulk action, can be applied to multiple instances.	`@Action#invokeOn()`	UI, Domain	
`@CollectionInteraction`	Enable subscribers on the Event Bus Service to either veto, validate or take further steps before/after a collection has been added to or removed from.	`@Collection#domainEvent()`	Domain	
`@Command`	Action invocation should be reified as a command object, optionally persistable for profiling and enhanced auditing, and background/async support.	`@Action#command()`	Domain	

Annotation	Purpose	Use instead	Layer	File-based layout?
`@CssClass`	Allow visual representation of individual objects or object members layout to be customized by application-specific CSS.	`#cssClass()` attribute for: `@DomainObjectLayout`, `@PropertyLayout`, `@CollectionLayout`, `@ActionLayout` and `@ParameterLayout`	UI	Yes
`@CssClassFa`	So that font awesome icons can be applied to action buttons/menu items and optionally as an object icon.	`cssClassFa()` attribute for: `@ActionLayout`, `DomainObjectLayout` and `ViewModelLayout`	UI	Yes
`@Debug`	Action only invokable in debug mode.	Not supported by either the Wicket viewer or the RestfulObjects viewer; use prototype mode instead (`@Action#restrictTo()`)	UI	
`@DescribedAs`	Provide a longer description/tool-tip of an object or object member.	`#describedAs()` attribute for `@DomainObjectLayout`, `@PropertyLayout`, `@CollectionLayout`, `@ActionLayout` and `@ParameterLayout`	UI	Yes
`@Disabled`	Object property cannot be edited, an object collection cannot be added to/removed from, or an object action cannot be invoked.	`#editing()` attribute for `@Property`, `@Collection` and `@DomainObject`	UI, Domain	Yes
`@Exploration`	Action available in special 'exploration' mode.	Not supported by either the Wicket viewer or the RestfulObjects viewer; use prototype mode instead (`@Action#restrictTo()`)	UI	
`@FieldOrder`	Order of properties and collections.	`@MemberOrder`	UI	Yes
`@Hidden`	Object member is not visible, or on domain service (to indicate that none of its actions are visible).	For domain object members, use `#hidden()` attribute of `Action`, `Property` or `Collection`. For domain service, use `@DomainService( nature=DOMAIN)`	UI, Domain	Yes
`@Idempotent`	Whether an action is idempotent (can be invoked multiple times with same post-condition).	`@Action#semantics`	Domain	

Annotation	Purpose	Use instead	Layer	File-based layout?
`@Ignore`	Exclude this method from the metamodel.	`@Programmatic`. `@Ignore` was deprecated because it can easily clash with `@org.junit.Ignore`.	Domain	
`@Immutable`	An object's state cannot be changed (properties cannot be edited, collections cannot be added to or removed from). Actions can still be invoked.	`@DomainObject#editing()`	Domain	
`@Mask`	How to parse/render values (never properly supported)	(None)	UI/domain	
`@MaxLength`	Maximum length of a property value (strings).	`#maxLength()` attribute for `@Property` or `@Parameter`	Domain	
`@MemberGroups`	Layout of properties and collections of a domain object or view model object.	dynamic `.layout.xml` files	UI	Yes
@MemberGroup Layout `	Grouping of properties into groups, and organizing of properties, collections into columns.	dynamic `.layout.xml` files	UI	Yes
`@MultiLine`	Render string property over multiple lines (a textarea rather than a textbox).	`#multiLine()` attribute for `@Property` or `@Parameter`	UI	Yes
`@MustSatisfy`	Specify arbitrary specification constraints on a property or action parameter.	`#mustSatisfy()` attribute for `@Property` or `@Parameter`	Domain	
`@Named`	Override name inferred from class. Required for parameter names (prior to Java8).	`#named()` attribute for `@DomainServiceLayout`, `@DomainObjectLayout`, `@PropertyLayout`, `@CollectionLayout`, `@ActionLayout` and `@ParameterLayout`	UI	Yes

Annotation	Purpose	Use instead	Layer	File-based layout?
`@NotContributed`	Indicates that a domain service action is not rendered as an action on the (entity) types of its parameters. For 1-arg query-only actions, controls whether the domain service action is rendered as a property or collection on the entity type of its parameter.	Use `@DomainService#nature()` to specify whether any of the actions in a domain service should appear in the menu bars (applies at type level, not action level). For individual actions, use `@ActionLayout#contributedAs()` to specify whether any individual action should be contributed only as an action or as an association (property or collection).	UI	
`@NotInServiceMenu`	Indicates that a domain service should not be rendered in the application menu (at top of page in Wicket viewer).	`@DomainService#nature()` to signify that none of the actions in a domain service should appear in the menu bars	UI	
`@NotPersisted`	Indicates that an object property is not persisted (meaning it is excluded from view model mementos, and should not be audited).	`#notPersisted()` attribute of `@Property` and `@Collection`	Domain, Persistence	
`@ObjectType`	For constructing the external identifier (URI) of an entity instance (part of its URL in both Wicket viewer and Restful Objects viewer). Also part of the toString representation of bookmarks, if using the Bookmark Service	`@DomainObject#objectType()`	Domain	
`@Optional`	Specifies that a property or action parameter is not mandatory.	`#optionality()` attribute for `@Property` or `@Parameter`	Domain	
`@Paged`	Number of instances to display in tables representing (standalone or parented) collections.	`#paged()` attribute for `@DomainObjectLayout` or `@CollectionLayout`	UI	Yes
`@Plural`	For the irregular plural form of an entity type.	`@DomainObjectLayout#plural()`	UI	

Annotation	Purpose	Use instead	Layer	File-based layout?
`@PostsActionInvokedEvent`	Post a domain event to the Event Bus Service indicating that an action has been invoked.	`@Action#domainEvent()`	Domain	
`@PostsCollectionAddedToEvent`	Post a domain event to the Event Bus Service indicating that an element has been added to a collection.	`@Collection#domainEvent()`	Domain	
`@PostsCollectionRemovedFromEvent`	Post a domain event to the Event Bus Service indicating that an element has been removed from a collection.	`@Collection#domainEvent()`	Domain	
`@PostsPropertyChangedEvent`	Post a domain event to the Event Bus Service indicating that the value of a property has changed.	`@Property#domainEvent()`	Domain	
`@PropertyInteraction`	Enable subscribers on the Event Bus Service to either veto, validate or take further steps before/after a property has been modified or cleared.	`@Property#domainEvent()`	Domain	
`@Prototype`	Indicates that an action should only be visible in 'prototype' mode.	`@Action#restrictTo()`	UI	Yes
`@PublishedAction`	Action invocation should be serialized and published by configured PublishingService (if any), eg to other systems.	`@Action#publishing()`	Domain	
`@PublishedObject`	Change to object should be serialized and published by configured PublishingService (if any), eg to other systems.	`@DomainObject#publishing()`	Domain	
`@QueryOnly`	Whether an action is query-only (has no side-effects).	`@Action#semantics()`	Domain	
`@RegEx`	Validate change to value of string property.	`#regexPattern()` for `@Property` or `@Parameter`.	Domain	
`@Render`	Eagerly (or lazily) render the contents of a collection.	`@CollectionLayout#render()`	UI	Yes
`@RenderedAsDayBefore`	Render dates as the day before; ie store [a,b) internally but render [a,b-1]) to end-user.	`#renderedAsDayBefore()` attribute for `@PropertyLayout` and `@ParameterLayout`.	UI	

Annotation	Purpose	Use instead	Layer	File-based layout?
`@Resolve`	Eagerly (or lazily) render the contents of a collection (same as `@Render`)	`@CollectionLayout #render()`	UI	Yes
`@SortedBy`	Display instances in collections in the order determined by the provided Comparator.	`@CollectionLayout #sortedBy()`	UI	Yes
`@TypeOf`	The type of entity stored within a collection, or as the result of invoking an action, if cannot be otherwise inferred, eg from generics.	`#typeOf()` attribute for `@Collection` and `@Action`	Domain	
`@TypicalLength`	The typical length of a string property, eg to determine a sensible length for a textbox.	`#typicalLength()` attribute for `@PropertyLayout` and `@ParameterLayout`	UI	Yes

2.6. Incomplete/partial support

These annotations have only incomplete/partial support, primarily relating to the management of value types. We recommend that you do not use them for now. Future versions of Apache Isis may either formally deprecate/retire them, or we may go the other way and properly support them. This will depend in part on the interactions between the Apache Isis runtime, its two viewer implementations, and DataNucleus persistence.

Table 7. Annotations with incomplete/partial support

Annotation	Purpose	Layer
`@Aggregated`	Indicates that the object is aggregated, or wholly owned, by a root object. This information could in theory provide useful semantics for some object store implementations, eg to store the aggregated objects "inline". Currently neither the JDO ObjectStore nor any of the viewers exploit this metadata.	Domain, Persistence

Annotation	Purpose	Layer
`@Defaulted`	Indicates that a (value) class has a default value. The concept of "defaulted" means being able to provide a default value for the type by way of the `o.a.i.applib.adapters.DefaultsProvider` interface. Generally this only applies to value types, where the `@Value` annotation implies encodability through the `ValueSemanticsProvider` interface. For these reasons the `@Defaulted` annotation is generally never applied directly, but can be thought of as a placeholder for future enhancements whereby non-value types might also have a default value provided for them.	Domain
`@Encodable`	Indicates that a (value) class can be serialized/encoded. Encodability means the ability to convert an object to-and-from a string, by way of the `o.a.i.applib.adapters.EncoderDecoder` interface. Generally this only applies to value types, where the `@Value` annotation implies encodability through the `ValueSemanticsProvider` interface. For these reasons the `@Encodable` annotation is generally never applied directly, but can be thought of as a placeholder for future enhancements whereby non-value types might also be directly encoded. Currently neither the Wicket viewer nor the RO viewer use this API. The Wicket viewer uses Wicket APIs, while RO viewer has its own mechanisms (parsing data from input JSON representations, etc.)	Persistence
`@NotPersistable`	Indicates that a domain object may not be programmatically persisted. + This annotation indicates that transient instances of this class may be created but may not be persisted. The framework will not provide the user with an option to 'save' the object, and attempting to persist such an object programmatically would be an error. For example: [source,java] ---- @NotPersistable(By.USER) public class InputForm { ... } ---- By default the annotated object is effectively transient (ie default to `By.USER_OR_PROGRAM`). This annotation is not supported by: Wicket viewer (which does not support transient objects). See also ISIS-743 contemplating the removal of this annotation.	Domain, Persistence

Annotation	Purpose	Layer
`@Parseable`	Indicates that a (value) class can be reconstructed from a string. Parseability means being able to parse a string representation into an object by way of the `o.a.i.applib.adapters.Parser` interface. Generally this only applies to value types, where the `@Value` annotation implies encodability through the `ValueSemanticsProvider` interface. For these reasons the `@Parser` annotation is generally never applied directly, but can be thought of as a placeholder for future enhancements whereby non-value types might also have be able to be parsed. Note that the Wicket viewer uses Apache Wicket's Converter API instead.	UI, Domain
`@Value`	Specify that a class has value-semantics. The `@Value` annotation indicates that a class should be treated as a value type rather than as a reference (or entity) type. It does this providing an implementation of a `o.a.i.applib.adapters.ValueSemanticsProvider`. For example: [source,java] ---- @Value(semanticsProviderClass= ComplexNumberValueSemanticsProvider.class) public class ComplexNumber { … } ---- The `ValueSemanticsProvider` allows the framework to interact with the value, parsing strings and displaying as text, and encoding/decoding (for serialization).	Domain

Chapter 3. `@Action`

The `@Action` annotation groups together all domain-specific metadata for an invokable action on a domain object or domain service.

The table below summarizes the annotation's attributes.

Table 8. `@Action` attributes

Attribute	Values (default)	Description
`associateWith()`	*memberId* ("")	associates an action with another property or collection of the action.
`associateWith-Sequence()`	*memberId* ("")	associates an action with another property or collection of the action.
`command()`	`AS_CONFIGURED`, `ENABLED`, `DISABLED` (`AS_CONFIGURED`)	whether the action invocation should be reified into a `o.a.i.applib.services.command.Command` object through the `CommandContext` service.
`commandExecuteIn()`	`FOREGROUND`,`BACKGROUND` (`FOREGROUND`)	whether to execute the command immediately, or to persist it (assuming that an appropriate implementation of `CommandService` has been configured) such that a background scheduler can execute the command asynchronously
`commandPersistence()`	`PERSISTED`, `NOT_PERSISTED`, `IF_HINTED` (`PERSISTED`)	whether the reified `Command` (as provided by the `CommandContext` domain service) should actually be persisted (assuming an appropriate implementation of `CommandService` has been configured).
`commandDtoProcessor()`	Implementation of `CommandDtoProcessor` interface (null)	If the `Command` also implements `CommandWithDto` (meaning that it can return a `CommandDto`, in other words be converted into an XML memento), then optionally specifies a processor that can refine this XML.
`domainEvent()`	subtype of `ActionDomainEvent` (`ActionDomainEvent.Default`)	the event type to be posted to the `EventBusService` to broadcast the action's business rule checking (hide, disable, validate) and its invocation (pre-execute and post-execute).
`hidden()`	`EVERYWHERE`, `NOWHERE` (`NOWHERE`)	indicates where (in the UI) the action should be hidden from the user.
`invokeOn()`	`OBJECT_ONLY`, `COLLECTION_ONLY`, `OBJECT_AND_COLLECTION` (`OBJECT_ONLY`)	(deprecated - use view models and associated actions instead). whether an action can be invoked on a single object and/or on many objects in a collection. Currently this is only supported for no-arg actions.

Attribute	Values (default)	Description
publishing()	AS_CONFIGURED, ENABLED, DISABLED (AS_CONFIGURED)	whether the action invocation should be published to the registered PublishingService.
publishing-PayloadFactory()	subtype of PublishingPayloadFactory- ForAction (none)	(deprecated). specifies that a custom implementation of PublishingPayloadFactoryForAction be used to create the (payload of the) published event representing the action invocation
restrictTo()	NO_RESTRICTIONS,PROTOTYPING (NO_RESTRICTIONS)	whether the action is only available in prototyping mode, or whether it is available also in production mode.
semantics()	SAFE_AND_REQUEST_CACHEABLE, SAFE, IDEMPOTENT, IDEMPOTENT_ARE_YOU_SURE NON_IDEMPOTENT, NON_IDEMPOTENT_ARE_YOU_SURE + (NON_IDEMPOTENT)	the action's semantics (ie whether objects are modified as the result of invoking this action, and if so whether reinvoking the action would result in no further change; if not whether the results can be cached for the remainder of the request). The ···ARE_YOU_SURE variants cause a confirmation dialog to be displayed in the Wicket viewer.
typeOf()	(none)	if the action returns a collection, hints as to the run-time type of the objects within that collection (as a fallback)

For example:

```
public class ToDoItem {
    public static class CompletedEvent extends ActionDomainEvent<ToDoItem> { }
    @Action(
        command=CommandReification.ENABLED,
        commandExecuteIn=CommandExecuteIn.FOREGROUND,          ①
        commandPersistence=CommandPersistence.NOT_PERSISTED,   ②
        domainEvent=CompletedEvent.class,
        hidden = Where.NOWHERE,                                ③
        invokeOn = InvokeOn.OBJECT_ONLY,                       ④
        publishing = Publishing.ENABLED,
        semantics = SemanticsOf.IDEMPOTENT
    )
    public ToDoItem completed() { ... }
}
```

① default value, so could be omitted

② default value, so could be omitted

③ default value, so could be omitted

④ default value, so could be omitted

3.1. associateWith()

The associateWith attribute allows an action to be associated with other properties or collections of the same domain object. The optional associateWithSequence attribute specifies the order of the action in the UI.

For example, an Order could have a collection of OrderItems, and might provide actions to add and remove items:

```
public class Order {

    @Collection
    SortedSet<OrderItem> getItems() { ... }

    @Action(associateWith="items", associateWithSequence="1")
    public Order addItem(Product p, int quantity) { ... }

    @Action(associateWith="items", associateWithSequence="2")
    public Order removeItem(OrderItem item) { ... }
    ...
}
```

These actions - addItem() and removeItem() can be thought of as associated with with the items collection because that is the state that they primarily affect.

In the user interface associated actions are rendered close to the member to which they relate.

The same effect can be accomplished using @MemberOrder or with the .layout.xml file.

3.1.1. Inferred Defaults and Choices

If an action is associated with a collection, then any scalar or collection parameter of the action that is the same type as that collection will automatically have a list of choices provided for it, being the items of the associated collection.

This is only done provided that there isn't already an explicit choicesNXxx() or autoCompleteNXxx() supporting method. However, this list of choices *does* take priority over any choices that are inferred from the parameter type itself (as per either an @DomainObject(autoCompleteRepository=⋯) or @DomainObject(bounded=⋯)).

In addition, if the action has a collection parameter of the same type as the associated collection, then the Wicket viewer will render the collection with checkboxes. The user can use these checkboxes can be used to select the items of the action parameter.

For example, suppose we have a "removeItems(...)" action:

```
public class Order {

    @Collection
    SortedSet<OrderItem> getItems() { ... }

    ...

    @Action(associateWith="items", associateWithSequence="2")
    public Order removeItems(SortedSet<OrderItem> items) { ... }
}
```

The Wicket viewer will then render the "items" collection with checkboxes, and any selected items will be used as the pre-selected set of items if the action is invoked.

3.2. Command Persistence and Processing

Every action invocation (and property edit for that matter) is automatically reified into a concrete `Command` object. The `@Action(command=…, commandXxx=…)` attributes provide hints for the persistence of that `Command` object, and the subsequent processing of that persisted command. The primary use cases for this are to support the deferring the execution of the action such that it can be invoked in the background, and to replay commands in a master/slave configuration.

Note that for a `Command` to actually be persisted requires an appropriate implementation of `CommandService` SPI. The framework does *not* provide an implementation of this SPI "out-of-the-box". However, the (non-ASF) Incode Platform's command module) *does* provide such an implementation.

3.2.1. Design

The annotation works with (and is influenced by the behaviour of) a number of domain services:

- `CommandContext`
- `CommandService`
- `BackgroundService` and
- `BackgroundCommandService`

Each action invocation is automatically reified by the `CommandContext` service into a `Command` object, capturing details of the target object, the action, the parameter arguments, the user, a timestamp and so on.

If an appropriate `CommandService` is configured (for example using (non-ASF) Incode Platform's command module), then the `Command` itself is persisted.

By default, actions are invoked in directly in the thread of the invocation. If there is an implementation of `BackgroundCommandService` (as the (non-ASF) Incode Platform's command module does provide), then this means in turn that the `BackgroundService` can be used by the domain object code to programmatically create background `Command`s.

If background `Command`s are used, then an external scheduler, using headless access, must also be configured.

3.2.2. `command()` and `commandPersistence()`

The `command()` and `commandPersistence() attributes work together to determine whether a command will actually be persisted. There inter-relationship is somewhat complex, so is probably best explained by way of examples:

`command()`	`isis.services.command.actions` **config property**	**action's declared** `semantics()`	`commandPersistence()`	**action dirties objects?**	**is command persisted?**
`ENABLED`	(any)	(any)	`PERSISTED`	(either)	yes
`ENABLED`	(any)	(any)	`IF_HINTED`	no	no
`ENABLED`	(any)	(any)	`IF_HINTED`	yes	yes
`ENABLED`	(any)	(any)	`NOT_PERSISTED`	(any)	no
`AS_CONFIGURED`	`all`	(any)	`PERSISTED`	no	yes
`AS_CONFIGURED`	`all`	(any)	`IF_HINTED`	no	no
`AS_CONFIGURED`	`all`	(any)	`IF_HINTED`	yes	yes
`AS_CONFIGURED`	`all`	(any)	`NOT_PERSISTED`	(any)	no
`AS_CONFIGURED`	`ignoreSafe` or `ignoreQueryOnly`	`SAFE`	`PERSISTED`	no	no (!)
`AS_CONFIGURED`	`ignoreSafe` or `ignoreQueryOnly`	`SAFE`	`IF_HINTED` or `NOT_PERSISTED`	no	no
`AS_CONFIGURED`	`ignoreSafe` or `ignoreQueryOnly`	`SAFE`	`PERSISTED` or `IF_HINTED`	yes	yes
`AS_CONFIGURED`	`ignoreSafe` or `ignoreQueryOnly`	`SAFE`	`NOT_PERSISTED`	yes	yes (!)
`AS_CONFIGURED`	`ignoreSafe` or `ignoreQueryOnly`	`IDEMPOTENT` or `NON_IDEMPOTENT`	`PERSISTED`	(any)	yes
`AS_CONFIGURED`	`ignoreSafe` or `ignoreQueryOnly`	`IDEMPOTENT` or `NON_IDEMPOTENT`	`IF_HINTED`	no	no

command()	isis.services.command.actions config property	action's declared semantics()	commandPersistence()	action dirties objects?	is command persisted?
AS_CONFIGURED	ignoreSafe or ignoreQueryOnly	IDEMPOTENT or NON_IDEMPOTENT	IF_HINTED	yes	yes
AS_CONFIGURED	ignoreSafe or ignoreQueryOnly	IDEMPOTENT or NON_IDEMPOTENT	NOT_PERSISTED	(any)	no
AS_CONFIGURED	none	(any)	PERSISTED	no	no (!)
AS_CONFIGURED	none	(any)	PERSISTED	yes	yes
AS_CONFIGURED	none	(any)	IF_HINTED	no	no
AS_CONFIGURED	none	(any)	IF_HINTED	yes	yes
AS_CONFIGURED	none	(any)	NOT_PERSISTED	no	no
AS_CONFIGURED	none	(any)	NOT_PERSISTED	yes	yes (!)
DISABLED	(any)	(any)	PERSISTED	no	no (!)
DISABLED	(any)	(any)	PERSISTED	yes	yes
DISABLED	(any)	(any)	IF_HINTED	no	no
DISABLED	(any)	(any)	IF_HINTED	yes	yes
DISABLED	(any)	(any)	NOT_PERSISTED	no	no
DISABLED	(any)	(any)	NOT_PERSISTED	yes	yes (!)

For example:

```
public class Order {
    @Action(
        command=CommandReification.ENABLED,
        commandPersistence=CommandPersistence.PERSISTED
    )
    public Invoice generateInvoice(...) { ... }
}
```

As can be seen, whether a command is actually persisted does not always follow the value of the commandPersistence() attribute. This is because the command() attribute actually determines whether any command metadata for the action is captured within the framework's internal metamodel. If command is DISABLED or does not otherwise apply due to the action's declared semantics, then the

framework decides to persist a command based solely on whether the action dirtied any objects (as if `commandPersistence()` was set to `IF_HINTED`).

3.2.3. `commandExecuteIn()`

For persisted commands, the `commandExecuteIn()` attribute determines whether the `Command` should be executed in the foreground (the default) or executed in the background.

Background execution means that the command is not executed immediately, but is available for a configured `BackgroundCommandService` to execute, eg by way of an in-memory scheduler such as Quartz. See here for further information on this topic.

For example:

```
public class Order {
    @Action(
        command=CommandReification.ENABLED,
        commandExecuteIn=CommandExecuteIn.BACKGROUND
    )
    public Invoice generateInvoice(...) { ... }
}
```

will result in the `Command` being persisted but its execution deferred to a background execution mechanism. The returned object from this action invocation is the persisted `Command` itself.

3.2.4. `commandDtoProcessor()`

The `commandDtoProcessor()` attribute allows an implementation of `CommandDtoProcessor` to be specified. This interface has the following API:

```
public interface CommandDtoProcessor {
    CommandDto process(             ①
            Command command,        ②
            CommandDto dto);        ③
}
```

① The returned `CommandDto`. This will typically be the `CommandDto` passed in, but supplemented in some way.

② The `Command` being processed

③ The `CommandDto` (XML) obtained already from the `Command` (by virtue of it also implementing `CommandWithDto`, see discussion below).

This interface is used by the framework-provided implementations of `ContentMappingService` for the REST API, allowing `Command`s implementations that also implement `CommandWithDto` to be further customised as they are serialized out. The primary use case for this capability is in support of master/slave replication.

- on the master, `Command`s are serialized to XML. This includes the identity of the target object and the argument values of all parameters.

However, any `Blob`s and `Clob`s are deliberately excluded from this XML (they are instead stored as references). This is to prevent the storage requirements for `Command` from becoming excessive. A `CommandDtoProcessor` can be provided to re-attach blob information if required.

- replaying `Command`s requires this missing parameter information to be reinstated. The `CommandDtoProcessor` therefore offers a hook to dynamically re-attach the missing `Blob` or `Clob` argument.

As a special case, returning `null` means that the command's DTO is effectively excluded when retrieving the list of commands. If replicating from master to slave, this effectively allows certain commands to be ignored. The `CommandDtoProcessor.Null` class provides a convenience implementation for this requirement.

If `commandDtoProcessor()` is specified, then `command()` is assumed to be ENABLED.

Example implementation

Consider the following method:

```
@Action(
    domainEvent = IncomingDocumentRepository.UploadDomainEvent.class,
    commandDtoProcessor = DeriveBlobArg0FromReturnedDocument.class
)
public Document upload(final Blob blob) {
    final String name = blob.getName();
    final DocumentType type = DocumentTypeData.INCOMING.findUsing
(documentTypeRepository);
    final ApplicationUser me = meService.me();
    String atPath = me != null ? me.getAtPath() : null;
    if (atPath == null) {
        atPath = "/";
    }
    return incomingDocumentRepository.upsertAndArchive(type, atPath, name, blob);
}
```

The `Blob` argument will not be persisted in the memento of the `Command`, but the information is implicitly available in the `Document` that is returned by the action. The `DeriveBlobArg0FromReturnedDocument` processor retrieves this information and dynamically adds:

```
public class DeriveBlobArg0FromReturnedDocument
        extends CommandDtoProcessorForActionAbstract {

    @Override
    public CommandDto process(Command command, CommandDto commandDto) {
        final Bookmark result = commandWithDto.getResult();
        if(result == null) {
            return commandDto;
        }
        try {
            final Document document = bookmarkService.lookup(result, Document.class);
            if (document != null) {
                ParamDto paramDto = getParamDto(commandDto, 0);
                CommonDtoUtils.setValueOn(paramDto, ValueType.BLOB, document.getBlob(
), bookmarkService);
            }
        } catch(Exception ex) {
            return commandDto;
        }
        return commandDto;
    }
    @Inject
    BookmarkService bookmarkService;
}
```

Null implementation

The null implementation can be used to simply indicate that no DTO should be returned for a `Command`. The effect is to ignore it for replay purposes:

```
pubc interface CommandDtoProcessor {
    ...
    class Null implements CommandDtoProcessor {
        public CommandDto process(Command command, CommandDto commandDto) {
            return null;
        }
    }
}
```

3.3. domainEvent()

Whenever a domain object (or list of domain objects) is to be rendered, the framework fires off multiple domain events for every property, collection and action of the domain object. In the cases of the domain object's actions, the events that are fired are:

- hide phase: to check that the action is visible (has not been hidden)
- disable phase: to check that the action is usable (has not been disabled)

- validate phase: to check that the action's arguments are valid
- pre-execute phase: before the invocation of the action
- post-execute: after the invocation of the action

Subscribers subscribe through the `EventBusService` using either Guava or Axon Framework annotations and can influence each of these phases.

By default the event raised is `ActionDomainEvent.Default`. For example:

```
public class ToDoItem {
    @Action()
    public ToDoItem completed() { ... }
    ...
}
```

The `domainEvent()` attribute allows a custom subclass to be emitted allowing more precise subscriptions (to those subclasses) to be defined instead. This attribute is also supported for collections and properties.

For example:

```
public class ToDoItem {
    public static class CompletedEvent extends ActionDomainEvent<ToDoItem> { }  ①
    @Action(domainEvent=CompletedEvent.class)
    public ToDoItem completed() { ... }
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

The framework provides no-arg constructor and will initialize the domain event using (non-API) setters rather than through the constructor. This substantially reduces the boilerplate required in subclasses because no explicit constructor is required.

3.3.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ActionDomainEvent ev) {
        ...
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.CompletedEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

The subscriber's method is called (up to) 5 times:

- whether to veto visibility (hide)
- whether to veto usability (disable)
- whether to veto execution (validate)
- steps to perform prior to the action being invoked.
- steps to perform after the action has been invoked.

The subscriber can distinguish these by calling `ev.getEventPhase()`. Thus the general form is:

```
@Programmatic
@com.google.common.eventbus.Subscribe
public void on(ActionDomainEvent ev) {
    switch(ev.getEventPhase()) {
        case HIDE:
            // call ev.hide() or ev.veto("") to hide the action
            break;
        case DISABLE:
            // call ev.disable("...") or ev.veto("...") to disable the action
            break;
        case VALIDATE:
            // call ev.invalidate("...") or ev.veto("...")
            // if action arguments are invalid
            break;
        case EXECUTING:
            break;
        case EXECUTED:
            break;
    }
}
```

It is also possible to abort the transaction during the executing or executed phases by throwing an exception. If the exception is a subtype of `RecoverableException` then the exception will be rendered as a user-friendly warning (eg Growl/toast) rather than an error.

3.3.2. Default, Doop and Noop events

If the `domainEvent` attribute is not explicitly specified (is left as its default value, `ActionDomainEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.actionAnnotation.domainEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `domainEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ActionDomainEvent.Doop` as such a subclass, so setting the `domainEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ActionDomainEvent.Noop`; if `domainEvent` attribute is set to this class, then no event will be posted.

3.3.3. Raising events programmatically

Normally events are only raised for interactions through the UI. However, events can be raised programmatically either by calling the `EventBusService` API directly, or by emulating the UI by wrapping the target object using the `WrapperFactory` domain service.

3.4. hidden()

Actions can be hidden at the domain-level, indicating that they are not visible to the end-user. This attribute can also be applied to properties and collections.

It is also possible to use `@ActionLayout#hidden()` or file-based layouts such that the action can be hidden at the view layer. Both options are provided with a view that in the future the view-layer semantics may be under the control of (expert) users, whereas domain-layer semantics should never be overridden or modified by the user.

For example:

```
public class Customer {
    @Action(hidden=Where.EVERYWHERE)
    public void updateStatus() { ... }
    ...
}
```

The acceptable values for the `where` parameter are:

- `Where.EVERYWHERE` or `Where.ANYWHERE`

  The action should be hidden at all times.

- `Where.NOWHERE`

  The action should not be hidden.

The other values of the `Where` enum have no meaning for a collection.

For actions of domain services the visibility is dependent upon its `@DomainService#nature()` and also on whether it is contributed (as per `@ActionLayout#contributedAs()`).

3.5. invokeOn()

The `invokeOn()` attribute indicates whether the an action can be invoked on a single object (the default) and/or on many objects in a collection.

For example:

```
public class ToDoItem {
    @Action(invokeOn=InvokeOn.OBJECT_AND_COLLECTION)
    public void markAsCompleted() {
        setCompleted(true);
    }
    ...
}
```

Actions to be invoked on collection (currently) have a number of constraints. It:

- must take no arguments
- cannot be hidden (any annotations or supporting methods to that effect will be ignored)
- cannot be disabled (any annotations or supporting methods to that effect will be ignored).

The example given above is probably ok, because `setCompleted()` is most likely idempotent. However, if the action also called some other method, then we should add a guard.

For example, for this non-idempotent action:

```
@Action(invokeOn=InvokeOn.OBJECT_AND_COLLECTION)
public void markAsCompleted() {
    setCompleted(true);
    todoTotalizer.incrementNumberCompleted();
}
```

we should instead write it as:

```
@Action(invokeOn=InvokeOn.OBJECT_AND_COLLECTION)
public void markAsCompleted() {
    if(isCompleted()) {
        return;
    }
    setCompleted(true);
    todoTotalizer.incrementNumberCompleted();
}
```

This attribute has no meaning if annotated on an action of a domain service.

3.6. `publishing()`

The `publishing()` attribute determines whether and how an action invocation is published via the registered implementation of a `PublishingService`) or `PublisherService`. This attribute is also supported for domain objects, where it controls whether changed objects are published as events, and for `@Property#publishing()`, where it controls whether property edits are published as events.

A common use case is to notify external "downstream" systems of changes in the state of the Isis application. The default value for the attribute is `AS_CONFIGURED`, meaning that the configuration property `isis.services.publish.actions` is used to determine the whether the action is published:

- `all`

  all action invocations are published

- `ignoreSafe` (or `ignoreQueryOnly`)

  invocations of actions with safe (read-only) semantics are ignored, but actions which may modify data are not ignored

- `none`

  no action invocations are published

If there is no configuration property in `isis.properties` then publishing is automatically enabled.

This default can be overridden on an action-by-action basis; if `publishing()` is set to `ENABLED` then the action invocation is published irrespective of the configured value; if set to `DISABLED` then the action invocation is *not* published, again irrespective of the configured value.

For example:

```
public class Order {
    @Action(publishing=Publishing.ENABLED)        ①
    public Invoice generateInvoice(...) { ... }
}
```

① because set to enabled, will be published irrespective of the configured value.

3.6.1. `publishingPayloadFactory()`

The (optional) related `publishingPayloadFactory()` specifies the class to use to create the (payload of the) event to be published by the publishing factory.

Rather than simply broadcast that the action was invoked, the payload factory allows a "fatter" payload to be instantiated that can eagerly push commonly-required information to all subscribers. For at least some subscribers this should avoid the necessity to query back for additional information.

Be aware that this attribute is only honoured by the (deprecated) `PublishingService`, so should itself be considered as deprecated. It is ignored by the replacement `PublisherService`,

3.7. `restrictTo()`

By default actions are available irrespective of the deployment mode. The `restrictTo()` attribute

specifies whether the action should instead be restricted to only available in prototyping mode.

For example:

```
public class Customer {
    public Order placeNewOrder() { ... }
    public List<Order> listRecentOrders() { ... }

    @Action(restrictTo=RestrictTo.PROTOTYPING)
    public List<Order> listAllOrders() { ... }
    ...
}
```

In this case the listing of all orders (in the `listAllOrders()` action) probably doesn't make sense for production; there could be thousands or millions. However, it would be useful to disaply how for a test or demo system where there are only a handful of orders.

3.8. `semantics()`

The `semantics()` attribute describes whether the invocation modifies state of the system, and if so whether it does so idempotently. If the action invocation does *not* modify the state of the system, in other words is safe, then it also can beused to specify whether the results of the action can be cached automatically for the remainder of the request.

The attribute was originally introduced for the RestfulObjects viewer in order that action invocations could be using the appropriate `HTTP` verb (`GET`, `PUT` and `POST`).

The table below summarizes the semantics:

Semantic	Changes state	Effect of multiple calls	HTTP verb (Restful Objects)
`SAFE_AND_REQUEST_CACHEABLE`	No	Will always return the same result each time invoked (within a given request scope)	`GET`
`SAFE`	No	Might result in different results each invocation	`GET`
`IDEMPOTENT` `IDEMPOTENT_ARE_YOU_SURE`	Yes	Will make no further changes if called multiple times (eg sets a property or adds to a `Set`). The "are you sure" variant requires that the user must explicitly confirm the action.	`PUT`

Semantic	Changes state	Effect of multiple calls	HTTP verb (Restful Objects)
`NON_IDEMPOTENT` `NON_IDEMPOTENT_ARE_YOU_SURE`	Yes	Might change the state of the system each time called (eg increments a counter or adds to a `List`). The "are you sure" variant requires that the user must explicitly confirm the action.	`POST`

The actions' semantics are also used by the core runtime as part of the in-built concurrency checkng; invocation of a safe action (which includes request-cacheable) does *not* perform a concurrency check, whereas non-safe actions *do* perform a concurrency check.

For example:

```
public class Customer {
    @Action(semantics=SemanticsOf.SAFE_AND_REQUEST_CACHEABLE)
    public CreditRating checkCredit() { ... }

    @Action(semantics=SemanticsOf.IDEMPOTENT)
    public void changeOfAddress(Address address) { ... }

    @Action(semantics=SemanticsOf.NON_IDEMPOTENT)
    public Order placeNewOrder() { ... }
    ...
}
```

Actions that are safe and request-cacheable automatically use the `QueryResultsCache` service to cache the result of the method. Note though that the results of this caching will only be apparent if the action is invoked from another method using the `WrapperFactory` service.

Continuing the example above, imagine code that loops over a set of `Order`s where each `Order` has an associated `Customer`. We want to check the credit rating of each `Customer` (a potentially expensive operation) but we don't want to do it more than once per `Customer`. Invoking through the `WrapperFactory` will allow us to accomplish this by exploiting the semantics of `checkCredit()` action:

```
public void dispatchToCreditWorthyCustomers(final List<Order> orders) {
    for(Order order: orders) {
        Customer customer = order.getCustomer();
        CreditRating creditRating = wrapperFactory.wrapSkipRules(customer).
checkCredit();  ①
        if(creditRating.isWorthy()) {
            order.dispatch();
        }
    }
}
@Inject
WrapperFactory wrapperFactory;
```

① wrap the customer to dispatch.

In the above example we've used `wrapSkipRules(…)` but if we wanted to enforce any business rules associated with the `checkCredit()` method, we would have used `wrap(…)`.

3.9. `typeOf()`

The `typeOf()` attribute specifies the expected type of an element returned by the action (returning a collection), when for whatever reason the type cannot be inferred from the generic type, or to provide a hint about the actual run-time (as opposed to compile-time) type. This attribute can also be specified for collections.

For example:

```
public void AccountService {
    @Action(typeOf=Customer.class)
    public List errantAccounts() {
        return customers.allNewCustomers();
    }
    ...

    @Inject
    CustomerRepository customers;
}
```

In general we recommend that you use generics instead, eg `List<Customer>`.

Chapter 4. `@ActionLayout`

The `@ActionLayout` annotation applies to actions, collecting together all UI hints within a single annotation.

The table below summarizes the annotation's attributes.

Table 9. `@ActionLayout` attributes

Attribute	Values (default)	Description
`bookmarking()`	`AS_ROOT`, `NEVER` (`NEVER`)	indicates if an action (with safe action semantics) is automatically bookmarked.
`contributedAs()`	`AS_BOTH`, `AS_ACTION`, `AS_ASSOCIATION`, `AS_NEITHER` (`AS_BOTH`)	for a domain service action that can be contributed, whether to contribute as an action or as an association (ie a property or collection). For a domain service action to be contributed, the domain services must have a nature nature of either `VIEW` or `VIEW_CONTRIBUTIONS_ONLY`, and the action must have safe action semantics, and takes a single argument, namely the contributee domain object.
`cssClass()`	Any string valid as a CSS class	an additional CSS class around the HTML that represents for the action, to allow targetted styling in `application.css`. Supported by the Wicket viewer but currently ignored by the RestfulObjects viewer.
`cssClassFa()`	Any valid Font awesome icon name	specify a font awesome icon for the action's menu link or icon.
`cssClassFaPosition()`	`LEFT`, `RIGHT` (`LEFT`)	Positioning of the icon on the button/menu item.
`describedAs()`	String.	provides a short description of the action, eg for rendering as a 'tool tip'.
`hidden()`	`EVERYWHERE`, `NOWHERE` (`NOWHERE`)	indicates where (in the UI) the action should be hidden from the user.
`named()`	String.	to override the name inferred from the action's name in code. A typical use case is if the desired name is a reserved Java keyword, such as `default` or `package`.
`position()`	`BELOW`, `RIGHT`, `PANEL`, `PANEL_DROPDOWN` (`BELOW`)	for actions associated (using `@MemberOrder#named()`) with properties, the positioning of the action's button with respect to the property

For example:

```
public class ToDoItems {
    @Action(semantics=SemanticsOf.SAFE)             ①
    @ActionLayout(
        bookmarking=BookmarkPolicy.AS_ROOT,
        cssClass="x-key",
        cssClassFa="fa-checkbox",
        describedAs="Mark the todo item as not complete after all",
        hidden=Where.NOWHERE                        ②
    )
    @MemberOrder(sequence = "1")
    public List<ToDoItem> notYetComplete() {
        ...
    }
}
```

① required for bookmarkable actions

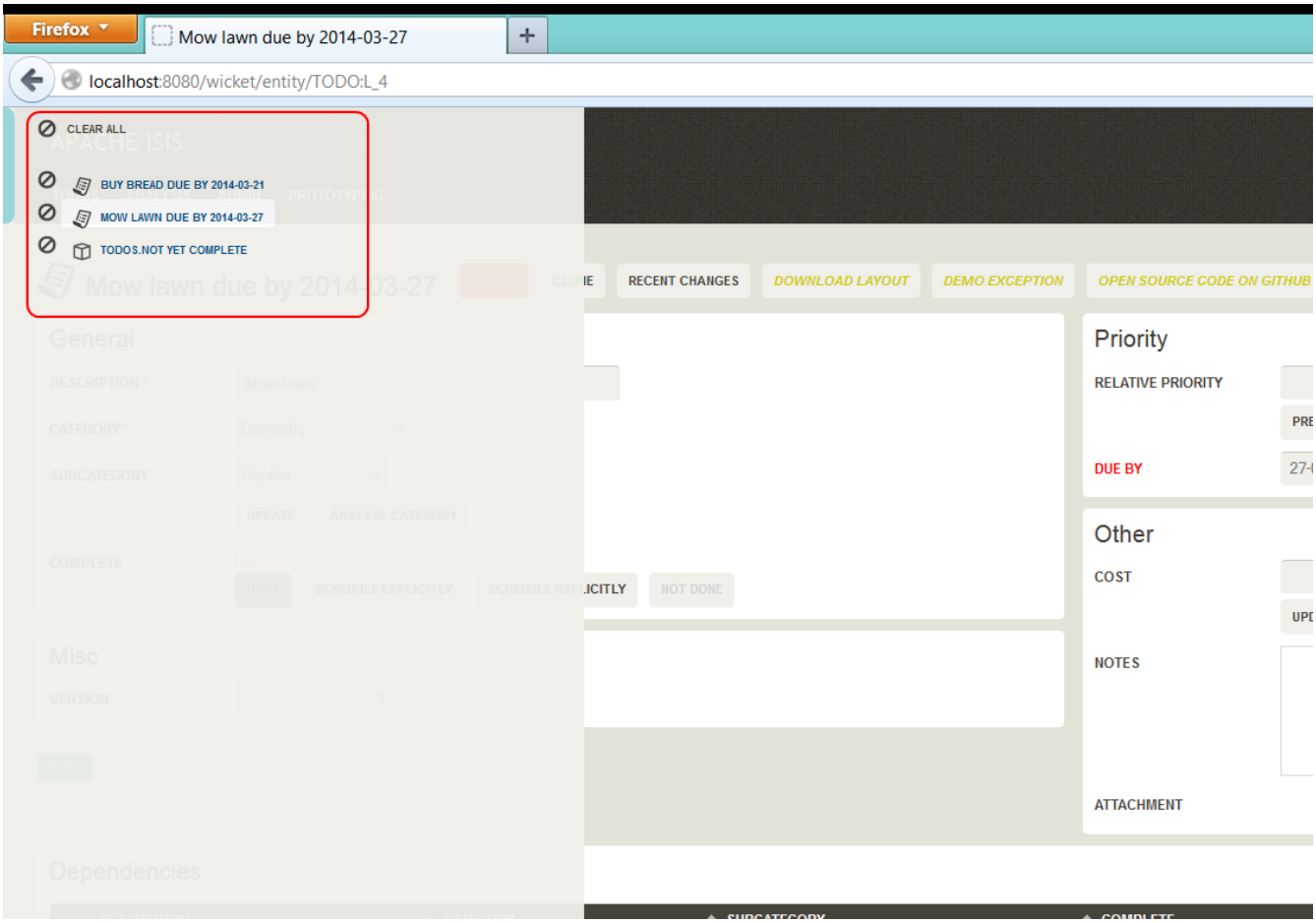
② default value, so could be omitted

As an alternative to using the `@ActionLayout` annotation, a file-based layout can be used (and is generally to be preferred since it is more flexible/powerful).

4.1. bookmarking()

The `bookmarking()` attribute indicates if an action (with safe action semantics) is automatically bookmarked. This attribute is also supported for domain objects.

In the Wicket viewer, a link to a bookmarked object is shown in the bookmarks panel:



Note that this screenshot shows an earlier version of the Wicket viewer UI (specifically, pre 1.8.0).

The Wicket viewer supports `alt-[` as a shortcut for opening the bookmark panel. `Esc` will close.

For example:

```
public class ToDoItems {
    @Action(semantics=SemanticsOf.SAFE)
    @ActionLayout(bookmarking=BookmarkPolicy.AS_ROOT)
    @MemberOrder(sequence = "1")
    public List<ToDoItem> notYetComplete() {
        ...
    }
}
```

indicates that the `notYetComplete()` action is bookmarkable.

The enum value `AS_CHILD` has no meaning for actions; it relates only to bookmarked domain objects.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"notYetComplete": {
    "actionLayout": { "bookmarking": "AS_ROOT" }
}
```

4.2. contributedAs()

For a domain service action that *can* be contributed, the `contributedAs()` attribute determines how it is contributed: as an action or as an association (ie a property or collection).

The distinction between property or collection is automatic: if the action returns a `java.util.Collection` (or subtype) then the action is contributed as a collection; otherwise it is contributed as a property.

For a domain service action to be contributed, the domain services must have a nature nature of either `VIEW` or `VIEW_CONTRIBUTIONS_ONLY`, and the action must have safe action semantics, and takes a single argument, namely the contributee domain object.

For example:

```
@DomainService(nature=NatureOfService.VIEW_CONTRIBUTIONS_ONLY)
public class CustomerContributions {
    @Action(semantics=SemanticsOf.SAFE)
    @ActionLayout(contributedAs=Contributed.AS_ASSOCIATION)
    public List<Order> mostRecentOrders(Customer customer) { ... }
    ...
}
```

The `@ActionLayout` is not required if the action does not have safe semantics, or if the action takes more than one argument; in these cases the action can only be contributed *as* an action.

It's also possible to use the attribute to suppress the action completely:

```
@DomainService(nature=NatureOfService.VIEW)
public class OrderContributions {
    @ActionLayout(contributedAs=Contributed.AS_NEITHER)
    public void cancel(Order order);
    ...
}
```

In such cases, though, it would probably make more sense to annotate the action as either hidden or indeed `@Programmatic`.

Unlike other `@ActionLayout` attributes, this attribute *cannot* be specified using a file-based layout because it relates to the contributor domain service, not the contributee domain object.

4.3. `cssClass()`

The `cssClass()` attribute can be used to render additional CSS classes in the HTML (a wrapping `<div>`) that represents the action. Application-specific CSS can then be used to target and adjust the UI representation of that particular element.

This attribute can also be applied to domain objects, view models, properties, collections and parameters.

For example:

```
public class ToDoItem {
    @ActionLayout(cssClass="x-key")
    public ToDoItem postpone(LocalDate until) { ... }
    ...
}
```

The similar `@ActionLayout#cssClassFa()` annotation attribute is also used as a hint to apply CSS, specifically to add Font Awesome icons on action menu items or buttons.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"postpone": {
    "actionLayout": { "cssClass": "x-key" }
}
```

4.4. `cssClassFa()`

The `cssClassFa()` attribute is used to specify the name of a Font Awesome icon name, to be rendered on the action's representation as a button or menu item. The related `cssClassFaPosition()` attribute specifies the positioning of the icon, to the left or the right of the text.

These attributes can also be applied to domain objects and to view models to specify the object's icon.

For example:

```
public class ToDoItem {
    @ActionLayout(
        cssClassFa="fa-step-backward"
    )
    public ToDoItem previous() { ... }

    @ActionLayout(
        cssClassFa="fa-step-forward",
        cssClassFaPosition=ActionLayout.CssClassFaPosition.RIGHT
    )
    public ToDoItem next() { ... }
}
```

There can be multiple "fa-" classes, eg to mirror or rotate the icon. There is no need to include the mandatory `fa` "marker" CSS class; it will be automatically added to the list. The `fa-` prefix can also be omitted from the class names; it will be prepended to each if required.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"previous": {
    "actionLayout": {
        "cssClassFa": "fa-step-backward",
        "cssClassFaPosition": "LEFT"
    }
},
"next": {
    "actionLayout": {
        "cssClassFa": "fa-step-forward",
        "cssClassFaPosition": "RIGHT"
    }
}
```

The similar `@ActionLayout#cssClass()` annotation attribute is also used as a hint to apply CSS, but for wrapping the representation of an object or object member so that it can be styled in an application-specific way.

4.5. `describedAs()`

The `describedAs()` attribute is used to provide a short description of the action to the user. In the Wicket viewer it is displayed as a 'tool tip'.

This attribute can also be specified for collections, properties, parameters, domain objects and view models.

For example:

```
public class Customer {
    @ActionLayout(describedAs="Place a repeat order of the last (most recently placed)
order")
    public Order placeRepeatOrder(...) { ... }
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"postpone": {
    "actionLayout": { "describedAs": "Place a repeat order of the last (most recently
placed) order" }
}
```

4.6. hidden()

The `hidden()` attribute indicates where (in the UI) the action should be hidden from the user. This attribute can also be applied to properties and collections.

It is also possible to use `@Action#hidden()` to hide an action at the domain layer. Both options are provided with a view that in the future the view-layer semantics may be under the control of (expert) users, whereas domain-layer semantics should never be overridden or modified by the user.

For example:

```
public class Customer {
    @ActionLayout(hidden=Where.EVERYWHERE)
    public void updateStatus() { ... }
    ...
}
```

The acceptable values for the `where` parameter are:

- `Where.EVERYWHERE` or `Where.ANYWHERE`

  The action should be hidden at all times.

- `Where.NOWHERE`

  The action should not be hidden.

The other values of the `Where` enum have no meaning for a collection.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"updateStatus": {
    "actionLayout": { "hidden": "EVERYWHERE" }
}
```

For actions of domain services the visibility is dependent upon its `@DomainService#nature()` and also on whether it is contributed (as per `@ActionLayout#contributedAs()`).

4.7. named()

The `named()` attribute explicitly specifies the action's name, overriding the name that would normally be inferred from the Java source code. This attribute can also be specified for collections, properties, parameters, domain objects, view models and domain services.

Following the don't repeat yourself principle, we recommend that you only use this attribute when the desired name cannot be used in Java source code. Examples of that include a name that would be a reserved Java keyword (eg "package"), or a name that has punctuation, eg apostrophes.

For example:

```
public class Customer {
    @ActionLayout(named="Get credit rating")
    public CreditRating obtainCreditRating() { ... }
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"obtainCreditRating": {
    "actionLayout": { "named": "Get credit rating" }
}
```

The framework also provides a separate, powerful mechanism for internationalization.

4.8. position()

The `position()` attribute pertains only to actions that have been associated with properties using `@MemberOrder#named()`. For these actions, it specifies the positioning of the action's button with

respect to the field representing the object property.

The attribute can take one of four values: `BELOW`, `RIGHT`, `PANEL` or `PANEL_DROPDOWN`.

For example:

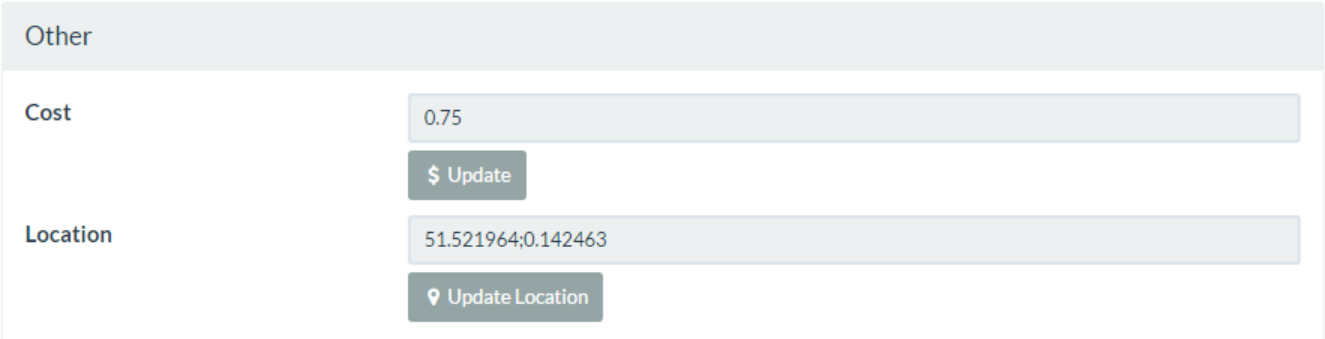
```
public class Customer {

    @Property(
        editing=Editing.DISABLED                    ①
    )
    public CustomerStatus getStatus() { ... }
    public void setStatus(CustomerStatus customerStatus) { ... }

    @MemberOrder(
        named="status",                             ②
        sequence="1"
    )
    @ActionLayout(
        named="Update",                             ③
        position=Position.BELOW
    )
    public CreditRating updateStatus(Customer ) { ... }
}
```

① indicate the property as read-only, such that it can only be updated using an action

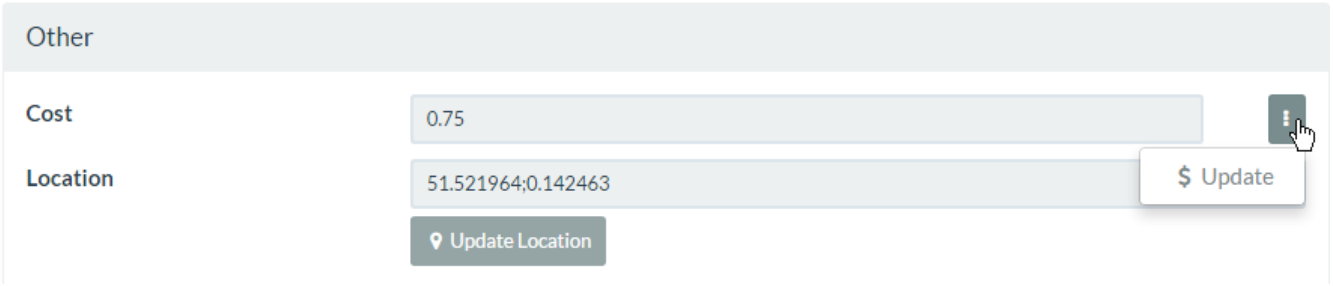
② associate the "updateStatus" action with the "status" property

③ give the action an abbreviated name, because the fact that the "status" property is to be updated is implied by its positioning

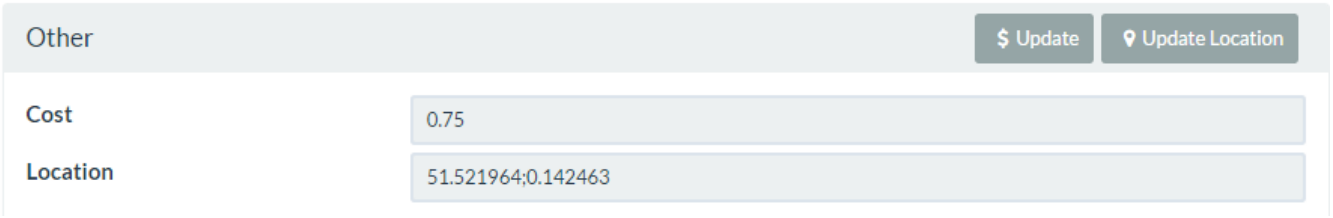
The default is `BELOW`, which is rendered (by the Wicket viewer) as shown below:



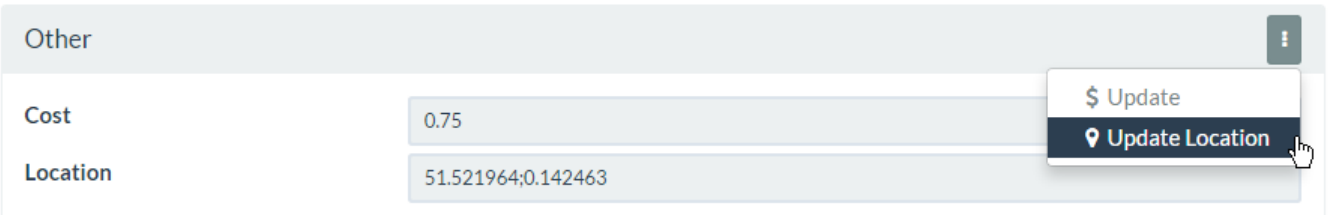
If the action is positioned as `RIGHT`, then the action's button is rendered to the right of the property's field, in a compact drop-down. This is ideal if there are many actions associated with a property:



If the action is positioned as `PANEL`, then the action's button is rendered on the header of the panel that contains the property:



And finally, if the action is positioned as `PANEL_DROPDOWN`, then the action's button is again rendered on the panel header, but as a drop-down:



If there are multiple actions associated with a single property then the positioning can be mix'ed-and-match'ed as required. If the `PANEL` or `PANEL_DROPDOWN` are used, then (as the screenshots above show) the actions from potentially multiple properties grouped by that panel will be shown together.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"obtainCreditRating": {
    "actionLayout": { "named": "Get credit rating" }
}
```

The fact that the layout is dynamic (does not require a rebuild/restart) is particularly useful in that the look-n-feel can be easily experimented with and adjusted.

Chapter 5. `@Collection`

The `@Collection` annotation applies to collections collecting together all domain semantics within a single annotation.

The table below summarizes the annotation's attributes.

Table 10. `@Collection` attributes

Attribute	Values (default)	Description
`domainEvent()`	subtype of `CollectionDomainEvent` (`CollectionDomainEvent.Default`)	the event type to be posted to the `EventBusService` to broadcast the collection's business rule checking (hide, disable, validate) and its modification (before and after).
`editing()`	`ENABLED`, `DISABLED`, `AS_CONFIGURED` (`AS_CONFIGURED`)	whether a collection can be added to or removed from within the UI
`editingDisabledReason()`	String value	if `editing()` is `DISABLED`, provides a reason as to why.
`hidden()`	`EVERYWHERE`, `OBJECT_FORMS`, `NOWHERE` (`NOWHERE`)	indicates where (in the UI) the collection should be hidden from the user.
`notPersisted()`	`true`, `false` (`false`)	whether to exclude from snapshots. [WARNING] ==== Collection must also be annotated with `@javax.jdo.annotations.NotPersistent` in order to not be persisted. ====
`typeOf()`		hints as to the run-time type of the objects within that collection (as a fallback)

For example:

```
public class ToDoItem {
    public static class DependenciesChangedEvent
            extends CollectionDomainEvent<ToDoItem, ToDoItem> { } ①
    @Collection(
        domainEvent=DependenciesChangedEvent.class,
        editing = Editing.ENABLED,
        hidden = Where.NOWHERE,                                   ②
        notPersisted = false,                                     ③
        typeOf = ToDoItem.class                                   ④
    )
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

① can use no-arg constructor.

② default value, so could be omitted

③ default value, so could be omitted

④ default value, so could be omitted

The annotation is one of a handful (others including `@CollectionLayout`, `@Property` and `@PropertyLayout`) that can also be applied to the field, rather than the getter method. This is specifically so that boilerplate-busting tools such as Project Lombok can be used.

5.1. `domainEvent()`

Whenever a domain object (or list of domain objects) is to be rendered, the framework fires off multiple domain events for every property, collection and action of the domain object. In the cases of the domain object's collections, the events that are fired are:

- hide phase: to check that the collection is visible (has not been hidden)
- disable phase: to check that the collection is usable (has not been disabled)
- validate phase: to check that the collection's arguments are valid (to add or remove an element)
- pre-execute phase: before the modification of the collection
- post-execute: after the modification of the collection

Subscribers subscribe through the `EventBusService` using either Guava or Axon Framework annotations and can influence each of these phases.

The Wicket viewer does *not* currently support the modification of collections; they are rendered read-only. However, domain events are still relevant to determine if such collections should be hidden.

The workaround is to create add/remove actions and use UI hints to render them close to the collection.

By default the event raised is `CollectionDomainEvent.Default`. For example:

```
public class ToDoItem {
    @Collection()
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

The `domainEvent()` attribute allows a custom subclass to be emitted allowing more precise subscriptions (to those subclasses) to be defined instead. This attribute is also supported for actions and properties.

For example:

```
public class ToDoItem {
    public static class DependenciesChangedEvent
            extends CollectionDomainEvent<ToDoItem, ToDoItem> { } ①
    @Collection(
        domainEvent=DependenciesChangedEvent.class
    )
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

① inherit from `CollectionDomainEvent<T,E>` where `T` is the type of the domain object being interacted with, and `E` is the type of the element in the collection (both `ToDoItem` in this example)

The benefit is that subscribers can be more targetted as to the events that they subscribe to.

The framework provides a no-arg constructor and will initialize the domain event using (non-API) setters rather than through the constructor. This substantially reduces the boilerplate in the subclasses because no explicit constructor is required..

5.1.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(CollectionDomainEvent ev) {
        ...
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.DependenciesChangedEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

The subscriber's method is called (up to) 5 times:

- whether to veto visibility (hide)
- whether to veto usability (disable)
- whether to veto execution (validate) the element being added to/removed from the collection
- steps to perform prior to the collection being added to/removed from
- steps to perform after the collection has been added to/removed from.

The subscriber can distinguish these by calling `ev.getEventPhase()`. Thus the general form is:

```
@Programmatic
@com.google.common.eventbus.Subscribe
public void on(CollectionDomainEvent ev) {
    switch(ev.getEventPhase()) {
        case HIDE:
            // call ev.hide() or ev.veto("") to hide the collection
            break;
        case DISABLE:
            // call ev.disable("...") or ev.veto("...") to disable the collection
            break;
        case VALIDATE:
            // call ev.invalidate("...") or ev.veto("...")
            // if object being added/removed to collection is invalid
            break;
        case EXECUTING:
            break;
        case EXECUTED:
            break;
    }
}
```

It is also possible to abort the transaction during the executing or executed phases by throwing an exception. If the exception is a subtype of `RecoverableException` then the exception will be rendered as a user-friendly warning (eg Growl/toast) rather than an error.

5.1.2. Default, Doop and Noop events

If the `domainEvent` attribute is not explicitly specified (is left as its default value, `CollectionDomainEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.collectionAnnotation.domainEvent.postForDefault` configuration collection can be set to "false"; this will disable posting.

On the other hand, if the `domainEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `CollectionDomainEvent.Doop` as such a subclass, so setting the `domainEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration collection setting.

And, conversely, the framework also provides `CollectionDomainEvent.Noop`; if `domainEvent` attribute is set to this class, then no event will be posted.

5.1.3. Raising events programmatically

Normally events are only raised for interactions through the UI. However, events can be raised programmatically either by calling the `EventBusService` API directly, or by emulating the UI by wrapping the target object using the `WrapperFactory` domain service.

5.2. `editing()`

The `editing()` annotation indicates whether a collection can be added to or removed from within the UI. This attribute can also be specified for properties, and can also be specified for the domain object

The related `editingDisabledReason()` attribute specifies the a hard-coded reason why the collection cannot be modified directly.

The Wicket viewer does **not** currently support the modification of collections; they are rendered read-only.

The workaround is to create add/remove actions and use UI hints to render them close to the collection.

Whether a collection is enabled or disabled depends upon these factors:

- whether the domain object has been configured as immutable through the `@DomainObject#editing()` attribute
- else (that is, if the domain object's editability is specified as being `AS_CONFIGURED`), then the value of the configuration property `isis.objects.editing`. If set to `false`, then the object's collections (and properties) are **not** editable
- else, then the value of the `@Collection(editing=…)` attribute itself.
- else, the result of invoking any supporting `disable…()` supporting methods

Thus, to make a collection read-only even if the object would otherwise be editable, use:

```
public class ToDoItem {
    @Collection(
        editing=Editing.DISABLED,
        editingDisabledReason="Use the add and remove actions to modify"
    )
    public SortedSet<ToDoItem> getDependencies() { ... }
}
```

To reiterate, it is *not* possible to enable editing for a collection if editing has been disabled at the object-level.

5.3. hidden()

Collections can be hidden at the domain-level, indicating that they are not visible to the end-user. This attribute can also be applied to actions and properties.

It is also possible to use `@CollectionLayout#hidden()` or using file-based layout such that the collection can be hidden at the view layer. Both options are provided with a view that in the future the view-layer semantics may be under the control of (expert) users, whereas domain-layer semantics should never be overridden or modified by the user.

For example:

```
public class Customer {
    @Collection(where=Where.EVERYWHERE)
    public SortedSet<Address> getAddresses() { ... }
}
```

The acceptable values for the `where` parameter are:

- `Where.EVERYWHERE` or `Where.ANYWHERE`

  The collection should be hidden everywhere.

- `Where.ANYWHERE`

  Synonym for everywhere.

- `Where.OBJECT_FORMS`

  The collection should be hidden when displayed within an object form.

- `Where.NOWHERE`

  The collection should not be hidden.

The other values of the `Where` enum have no meaning for a collection.

The Wicket viewer suppresses collections when displaying lists of objects.

The RestfulObjects viewer by default suppress collections when rendering a domain object.

5.4. `notPersisted()`

The (somewhat misnamed) `notPersisted()` attribute indicates that the collection should be excluded from any snapshots generated by the `XmlSnapshotService`. This attribute is also supported for properties.

This annotation does *not* specify that a collection is not persisted in the JDO/DataNucleus objectstore. See below for details as to how to additionally annotate the collection for this.

For example:

```
public class Customer {
    @Collection(notPersisted=true)
    public SortedSet<Order> getPreviousOrders() {...}
    public void setPreviousOrder(SortedSet<Order> previousOrders) {...}
    ...
}
```

Historically this annotation also hinted as to whether the collection's contents should be persisted in the object store. However, the JDO/DataNucleus objectstore does not recognize this annotation. Thus, to ensure that a collection is actually not persisted, it should **also** be annotated with `@javax.jdo.annotations.NotPersistent`.

For example:

```
public class Customer {
    @Collection(notPersisted=true)              ①
    @javax.jdo.annotations.NotPersistent        ②
    public SortedSet<Order> getPreviousOrders() {...}
    public void setPreviousOrder(SortedSet<Order> previousOrders) {...}
    ...
}
```

① ignored by Apache Isis

② ignored by JDO/DataNucleus

Alternatively, if the collection is derived, then providing only a "getter" will also work:

```
public class Customer {
    public SortedSet<Order> getPreviousOrders() {...}
    ...
}
```

5.5. typeOf()

The `typeOf()` attribute specifies the expected type of an element contained within a collection when for whatever reason the type cannot be inferred from the generic type, or to provide a hint about the actual run-time (as opposed to compile-time) type. This attribute can also be specified for actions.

For example:

```
public void Customer {
    @TypeOf(Order.class)
    public SortedSet getOutstandingOrders() { ... }
    ...
}
```

In general we recommend that you use generics instead, eg `SortedSet<Order>`.

Chapter 6. `@CollectionLayout`

The `@CollectionLayout` annotation applies to collections, collecting together all UI hints within a single annotation. It is also possible to apply the annotation to actions of domain services that are acting as contributed collections.

The table below summarizes the annotation's attributes.

Table 11. `@CollectionLayout` attributes

Attribute	Values (default)	Description
`cssClass()`	Any string valid as a CSS class	the css class that a collection should have, to allow more targetted styling in `application.css`
`defaultView()`	`table`, `excel`, `calendar`, `map`, ...	Which view is selected by default, if multiple views are available. See the (non-ASF) Incode Platform for further Wicket components providing views.
`describedAs()`	String.	description of this collection, eg to be rendered in a tooltip.
`hidden()`	`EVERYWHERE`, `OBJECT_FORMS`, `NOWHERE` (`NOWHERE`)	indicates where (in the UI) the collection should be hidden from the user.
named()	String.	to override the name inferred from the collection's name in code. A typical use case is if the desired name is a reserved Java keyword, such as `default` or `package`.
namedEscaped()	`true`,`false` (`true`)	whether to HTML escape the name of this property.
`paged()`	Positive integer	the page size for instances of this class when rendered within a table.
`render()`	`EAGERLY`, `LAZILY` (`LAZILY`)	whether the collection should be (eagerly) rendered open or (lazily) rendered closed
`sortedBy()`	Subclass of `java.util.Comparator` for element type	indicates that the elements in the `java.util.SortedSet` collection should be sorted according to a specified `Comparator` rather than their natural sort order.

For example:

```
public class ToDoItem {
    @CollectionLayout(
        cssClass="x-key",
        named="Todo items that are <i>dependencies</i> of this item.",
        namedEscaped=false,
        describedAs="Other todo items that must be completed before this one",
        labelPosition=LabelPosition.LEFT,
        render=EAGERLY)
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

As an alternative to using the `@CollectionLayout` annotation, a file-based layout can be used (and is generally to be preferred since it is more flexible/powerful).

The annotation is one of a handful (others including `@Collection`, `@Property` and `@PropertyLayout`) that can also be applied to the field, rather than the getter method. This is specifically so that boilerplate-busting tools such as Project Lombok can be used.

6.1. cssClass()

The `cssClass()` attribute can be used to render additional CSS classes in the HTML (a wrapping `<div>`) that represents the collection. Application-specific CSS can then be used to target and adjust the UI representation of that particular element.

This attribute can also be applied to domain objects, view models, actions, properties and parameters.

For example:

```
public class ToDoItem {
    @CollectionLayout(
        cssClass="x-important"
    )
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"dependencies": {
    "collectionLayout": { "cssClass": "x-important" }
}
```

6.2. defaultView()

The Wicket viewer allows additional views to be configured to render collections of objects; at the time of writing thesee include the excel, fullcalendar2 and gmap3 provided by the (non-ASF) Incode Platform. If the objects to be rendered have the correct "shape", then the appropriate view will be made available. For example, objects with a date can be rendered using `calendar`; objects with locations can be rendered using `map`.

The `defaultView()` attribute is used to select which of these views should be used by default for a given collection.

For example:

```
public class BusRoute {
    @CollectionLayout(
        defaultView="map"
    )
    public SortedSet<BusStop> getStops() { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"dependencies": {
    "collectionLayout": {
        "defaultView": "map"
    }
}
```

This attribute takes precedence over any value for the `@CollectionLayout#render()` attribute. For example, if the `defaultView` attribute is defined to "table", then the table will be show even if `render` is set to `LAZILY`.

6.3. describedAs()

The `describedAs()` attribute is used to provide a short description of the collection to the user. In the Wicket viewer it is displayed as a 'tool tip'.

The `describedAs()` attribute can also be specified for properties, actions, parameters, domain

objects and view models.

For example:

```
public class ToDoItem {
    @CollectionLayout(
        describedAs="Other todo items that must be completed before this one"
    )
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"dependencies": {
    "collectionLayout": {
        "describedAs": "Other todo items that must be completed before this one"
    }
}
```

6.4. hidden()

The `hidden()` attribute indicates where (in the UI) the collection should be hidden from the user. This attribute can also be applied to actions and properties.

It is also possible to use `@Collection#hidden()` to hide an action at the domain layer. Both options are provided with a view that in the future the view-layer semantics may be under the control of (expert) users, whereas domain-layer semantics should never be overridden or modified by the user.

For example:

```
public class ToDoItem {
    @CollectionLayout(
        hidden=Where.EVERYWHERE
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

The acceptable values for the `where` parameter are:

- `Where.EVERYWHERE` or `Where.ANYWHERE`

  The collection should be hidden everywhere.

- `Where.ANYWHERE`

  Synonym for everywhere.

- `Where.OBJECT_FORMS`

  The collection should be hidden when displayed within an object form.

- `Where.NOWHERE`

  The collection should not be hidden.

The other values of the `Where` enum have no meaning for a collection.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"dependencies": {
    "collectionLayout": { "hidden": "EVERYWHERE" }
}
```

6.5. `named()`

The `named()` attribute explicitly specifies the collection's name, overriding the name that would normally be inferred from the Java source code. This attribute can also be specified for actions, properties, parameters, domain objects, view models and domain services.

Following the don't repeat yourself principle, we recommend that you only use this attribute when the desired name cannot be used in Java source code. Examples of that include a name that would be a reserved Java keyword (eg "package"), or a name that has punctuation, eg apostrophes.

By default the name is HTML escaped. To allow HTML markup, set the related `namedEscaped()` attribute to `false`.

For example:

```
public class ToDoItem {
    @CollectionLayout(
        named="Todo items that are <i>dependencies</i> of this item",
        namedEscaped=false
    )
    public SortedSet<ToDoItem getDependencies() { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"dependencies": {
    "collectionLayout": {
        "named": "Todo items that are <i>dependencies</i> of this item",
        "namedEscaped": false,
    }
}
```

The framework also provides a separate, powerful mechanism for internationalization.

6.6. paged()

The `paged()` attribute specifies the number of rows to display in a (parented) collection. This attribute can also be applied to domain objects and view models.

The RestfulObjects viewer currently does not support paging. The Wicket viewer *does* support paging, but note that the paging is performed client-side rather than server-side.

We therefore recommend that large collections should instead be modelled as actions (to allow filtering to be applied to limit the number of rows).

For example:

```
public class Order {
    @CollectionLayout(paged=15)
    public SortedSet<OrderLine> getDetails() {...}
}
```

It is also possible to specify a global default for the page size of standalone collections, using the configuration property `isis.viewer.paged.parented`.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"details": {
    "collectionLayout": {
        "paged": 15
    }
}
```

6.7. `render()`

The `render()` attribute specifies that the collection be rendered either "eagerly" (shown open, displaying its contents) or "lazily" (shown closed, hiding its contents). The terminology here is based on the similar concept of lazy loading of collections in the domain/persistence layer boundary (except that the rendering relates to the presentation/domain layer boundary).

For example:

```
public class Order {
    @CollectionLayout(render=RenderType.EAGERLY)
    public SortedSet<LineItem> getDetails() { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"details": {
    "collectionLayout": {
        "render": "EAGERLY"
    }
}
```

Note that contributed collections (which, under the covers are just action invocations against a domain service) are always rendered eagerly.

Also, if a `@CollectionLayout#defaultView()` attribute has been specified then that will take precedence over the value of the `render()` attribute.

6.8. `sortedBy()`

The `sortedBy()` attribute specifies that the collection be ordered using the specified comparator, rather than the natural ordering of the entity (as would usually be the case).

For example:

```
public class ToDoItem implements Comparable<ToDoItem> {          ①
    public static class DependenciesComparator                   ②
            implements Comparator<ToDoItem> {
        @Override
        public int compare(ToDoItem p, ToDoItem q) {
            return ORDERING_BY_DESCRIPTION                       ③
                    .compound(Ordering.<ToDoItem>natural())
                    .compare(p, q);
        }
    }
    @CollectionLayout(sortedBy=DependenciesComparator.class) ④
    public SortedSet<ToDoItem> getDependencies() { ... }
    ...
}
```

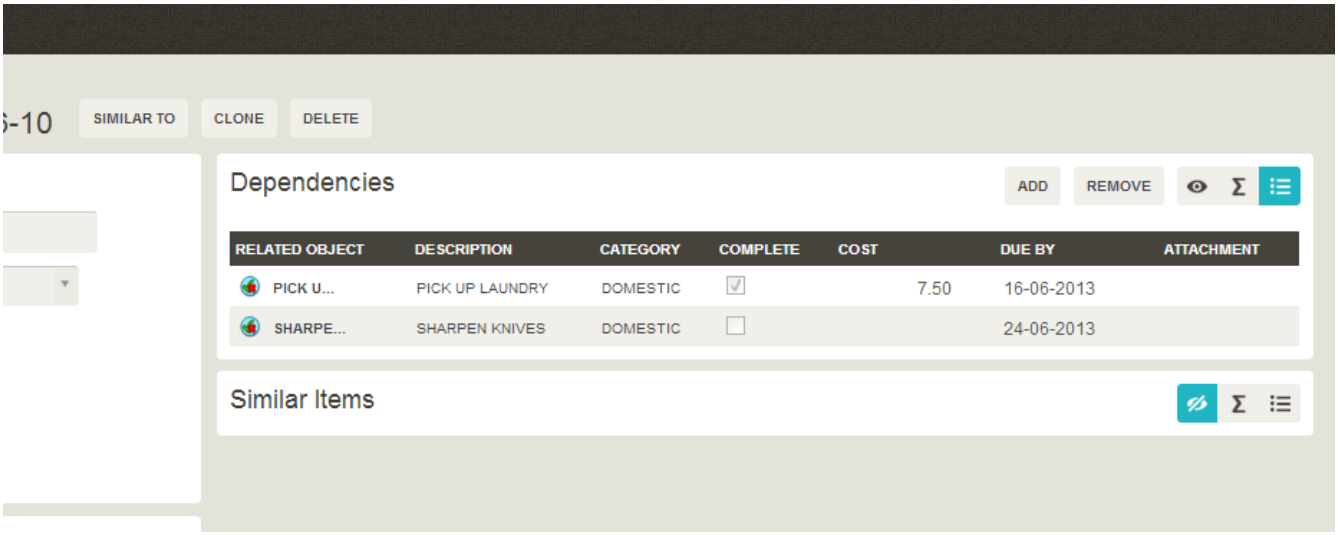
① the class has a natural ordering (implementation not shown)

② declaration of the comparator class

③ ordering defined as being by the object's `description` property (not shown), and then by the natural ordering of the class

④ specify the comparator to use

When the `dependencies` collection is rendered, the elements are sorted by the `description` property first:



Note that this screenshot shows an earlier version of the Wicket viewer UI (specifically, pre 1.8.0).

Without this annotation, the order would have been inverted (because the natural ordering places items not completed before those items that have been completed.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"dependencies": {
    "collectionLayout": {
        "sortedBy": "com.mycompany.myapp.dom.ToDoItem.DependenciesComparator"
    }
}
```

Chapter 7. `@Column` (`javax.jdo`)

The JDO `@javax.jdo.annotation.Column` provides metadata describing how JDO/DataNucleus should persist the property to a database RDBMS table column (or equivalent concept for other persistence stores).

Apache Isis also parses and interprets this annotation in order to build up aspects of its metamodel.

Isis parses the `@Column` annotation from the Java source code; it does not query the JDO metamodel. This means that it the `@Column` annotation must be used rather than the equivalent `<column>` XML metadata.

Moreover, while JDO/DataNucleus will recognize annotations on either the field or the getter method, Apache Isis (currently) only inspects the getter method. Therefore ensure that the annotation is placed there.

This section identifies which attributes of `@Column` are recognized and used by Apache Isis.

7.1. Nullability

The `allowsNull()` attribute is used to specify if a property is mandatory or is optional.

For example:

```
public class Customer {
    @javax.jdo.annotations.Column(allowsNull="true")
    public String getMiddleInitial() { ... }
    public void setMiddleInitial(String middleInitial) { ... }
```

Isis also provides `@Property#optionality()` attribute. If both are specified, Apache Isis will check when it initializes for any contradictions, and will fail-fast with an appropriate error message in the log if there are.

You should also be aware that in the lack of either the `@Column#allowsNull()` or the `@Property#optionality()` attributes, that the JDO and Apache Isis defaults differ. Apache Isis rule is straight-forward: properties are assumed to be required. JDO on the other hand specifies that only primitive types are mandatory; everything else is assumed to be optional. Therefore a lack of either annotation can also trigger the fail-fast validation check.

In the vast majority of cases you should be fine just to add the `@Column#allowsNull()` attribute to the getter. But see the documentation for `@Property#optionality()` attribute for discussion on one or two minor edge cases.

7.2. Length for `String`s

The `length()` attribute is used to specify the length of `java.lang.String` property types as they map to `varchar(n)` columns.

For example:

```
public class Customer {
    @javax.jdo.annotations.Column(length=20)
    public String getFirstName() { ... }
    public void setFirstName(String firstName) { ... }
    @javax.jdo.annotations.Column(allowsNull="true", length=1)
    public String getMiddleInitial() { ... }
    public void setMiddleInitial(String middleInitial) { ... }
    @javax.jdo.annotations.Column(length=30)
    public String getLastName() { ... }
    public void setLastName(String lastName) { ... }
```

Isis also provides `@Property#maxLength()` attribute. If both are specified, Apache Isis will check when it initializes for any contradictions, and will fail-fast with an appropriate error message in the log if there are.

7.3. Length/scale for `BigDecimal`s

The `length()` and `scale()` attributes are used to infer the precision/scale of `java.math.BigDecimal` property types as they map to `decimal(n,p)` columns.

For example:

```
public class Customer {
    @javax.jdo.annotations.Column(length=10, scale=2)
    public BigDecimal getTotalOrdersToDate() { ... }
    public void setTotalOrdersToDate(BigDecimal totalOrdersToDate) { ... }
```

For <code>BigDecimal</code>s it is also possible to specify the <code>@Digits</code> annotation, whose form is <code>@Digits(integer, fraction)</code>. There is a subtle difference here: while <code>@Column#scale()</code> corresponds to <code>@Digits#fraction()</code>, the value of <code>@Column#length()</code> (ie the precision) is actually the _sum of the <code>@Digits’ `integer()</code> and <code>fraction()</code> parts.

If both are specified, Apache Isis will check when it initializes for any contradictions, and will fail-fast with an appropriate error message in the log if there are.

7.4. Hints and Tips

This seems to be a good place to describe some additional common mappings that use `@Column`. Unlike the sections above, the attributes specified in these hints and tips aren't actually part of Apache Isis metamodel.

7.4.1. Mapping foreign keys

The `name()` attribute can be used to override the name of the column. References to other objects are generally mapped as foreign key columns. If there are multiple references to a given type, then you will want to override the name that JDO/DataNucleus would otherwise default.

For example (taken from estatio app):

```
public class PartyRelationship {
    @Column(name = "fromPartyId", allowsNull = "false")
    public Party getFrom() { ... }
    public void setFrom(Party from) { ... }
    @Column(name = "toPartyId", allowsNull = "false")
    public Party getTo() { ... }
    public void setTo(Party to) { ... }
    ...
}
```

7.5. Mapping `Blob`s and `Clob`s

Isis provides custom value types for `Blob`s and `Clob`s. These value types have multiple internal fields, meaning that they corresponding to multiple columns in the database. Mapping this correctly requires using `@Column` within JDO's `@Persistent` annotation.

For example, here's how to map a `Blob` (taken from (non-ASF) Isis addons' todoapp):

```
private Blob attachment;
@javax.jdo.annotations.Persistent(defaultFetchGroup="false", columns = {
        @javax.jdo.annotations.Column(name = "attachment_name"),
        @javax.jdo.annotations.Column(name = "attachment_mimetype"),
        @javax.jdo.annotations.Column(name = "attachment_bytes", jdbcType = "BLOB",
sqlType = "LONGVARBINARY")
})
@Property(
        domainEvent = AttachmentDomainEvent.class,
        optionality = Optionality.OPTIONAL
)
public Blob getAttachment() { ... }
public void setAttachment(Blob attachment) { ... }
```

And here's how to map a `Clob` (also taken from the todoapp):

```
private Clob doc;
@javax.jdo.annotations.Persistent(defaultFetchGroup="false", columns = {
        @javax.jdo.annotations.Column(name = "doc_name"),
        @javax.jdo.annotations.Column(name = "doc_mimetype"),
        @javax.jdo.annotations.Column(name = "doc_chars", jdbcType = "CLOB", sqlType =
"LONGVARCHAR")
})
@Property(
        optionality = Optionality.OPTIONAL
)
public Clob getDoc() { ... }
public void setDoc(final Clob doc) { ... }
```

Chapter 8. `@Digits` (`javax`)

The `@javax.validation.constraints.Digits` annotation is recognized by Apache Isis as a means to specify the precision for properties and action parameters of type `java.math.BigDecimal`.

For example (taken from the (non-ASF) Isis addons' todoapp):

```
@javax.jdo.annotations.Column(
    scale=2                                                 ①
)
@javax.validation.constraints.Digits(
    integer=10,
    fraction=2                                              ②
)
public BigDecimal getCost() {
    return cost;
}
public void setCost(final BigDecimal cost) {
    this.cost = cost!=null
        ? cost.setScale(2, BigDecimal.ROUND_HALF_EVEN)      ③
        :null;
}
```

① the `@Column#scale()` attribute must be ...

② ... consistent with `@Digits#fraction()`

③ the correct idiom when setting a new value is to normalized to the correct scale

Chapter 9. `@Discriminator` (`javax.jdo`)

The `@javax.jdo.annotation.Discriminator` is used by JDO/DataNucleus to specify how to discriminate between subclasses of an inheritance hierarchy.

It is valid to add a `@Discriminator` for any class, even those not part of an explicitly mapped inheritance hierarchy. Apache Isis also checks for this annotation, and if present will use the `@Discriminator#value()` as the object type, a unique alias for the object's class name.

Isis parses the `@Discriminator` annotation from the Java source code; it does not query the JDO metamodel. This means that it the `@Discriminator` annotation must be used rather than the equivalent `<discriminator>` XML metadata.

Moreover, while JDO/DataNucleus will recognize annotations on either the field or the getter method, Apache Isis (currently) only inspects the getter method. Therefore ensure that the annotation is placed there.

This value is used internally to generate a string representation of an objects identity (the `Oid`). This can appear in several contexts, including:

- as the value of `Bookmark#getObjectType()` and in the `toString()` value of `Bookmark` (see `BookmarkService`)
  - and thus in the "table-of-two-halves" pattern, as per the (non-ASF) Incode Platform's poly module
- in the serialization of `OidDto` in the command and interaction schemas
- in the URLs of the RestfulObjects viewer
- in the URLs of the Wicket viewer (in general and in particular if copying URLs)
- in XML snapshots generated by the `XmlSnapshotService`

9.1. Examples

For example:

```
@javax.jdo.annotations.Discriminator(value="custmgmt.Customer")
public class Customer {
    ...
}
```

has an object type of `custmgmt.Customer`.

9.2. Precedence

The rules of precedence for determining a domain object's object type are:

1. `@Discriminator`
2. `@DomainObject#objectType`
3. `@PersistenceCapable`, if at least the `schema` attribute is defined.

   If both `schema` and `table` are defined, then the value is “schema.table”. If only `schema` is defined, then the value is “schema.className”.

4. Fully qualified class name of the entity.

This might be obvious, but to make explicit: we recommend that you always specify an object type for your domain objects.

Otherwise, if you refactor your code (change class name or move package), then any externally held references to the OID of the object will break. At best this will require a data migration in the database; at worst it could cause external clients accessing data through the Restful Objects viewer to break.

If the object type is not unique across all domain classes then the framework will fail-fast and fail to boot. An error message will be printed in the log to help you determine which classes have duplicate object tyoes.

Chapter 10. `@DomainObject`

The `@DomainObject` annotation applies to domain objects, collecting together all domain semantics within a single annotation.

The table below summarizes the annotation's attributes.

Table 12. `@DomainObject` attributes

Attribute	Values (default)	Description
`auditing()`	`AS_CONFIGURED`, `ENABLED`, `DISABLED` (`AS_CONFIGURED`)	indicates whether each of the changed properties of an object should be submitted to the registered `AuditingService` (deprecated) or (its replacement) `AuditerService`
`autoCompleteRepository()`	Domain service class	nominate a method on a domain service to be used for looking up instances of the domain object
`autoCompleteAction()`	Method name (`autoComplete()`)	override the method name to use on the auto-complete repository
`bounded()`	`true`, `false` (`false`)	Whether the number of instances of this domain class is relatively small (a "bounded" set), such that instances could be selected from a drop-down list box or similar.
`created-LifecycleEvent()`	subtype of `ObjectCreatedEvent` (`ObjectCreatedEvent.Default`)	the event type to be posted to the `EventBusService` whenever an instance is created
`editing()`	`AS_CONFIGURED`, `ENABLED`, `DISABLED` (`AS_CONFIGURED`)	whether the object's properties and collections can be edited or not (ie whether the instance should be considered to be immutable)
`mixinMethod()`	Method name within the mixin	How to recognize the "reserved" method name, meaning that the mixin's own name will be inferred from the mixin type. Typical examples are "exec", "execute", "invoke", "apply" and so on. The default "reserved" method name is `$$`.
`nature()`	`NOT_SPECIFIED`, `JDO_ENTITY`, `EXTERNAL_ENTITY`, `INMEMORY_ENTITY`, `MIXIN`, `VIEW_MODEL` (`NOT_SPECIFIED`)	whether the domain object logically is an entity (part of the domain layer) or is a view model (part of the application layer); or is a mixin. If an entity, indicates how its persistence is managed.
`objectType()`	(none, which implies fully qualified class name)	specify an alias for the domain class used to uniquely identify the object both within the Apache Isis runtime and externally

Attribute	Values (default)	Description
persisted-LifecycleEvent()	subtype of ObjectPersistedEvent (ObjectPersistedEvent.Default)	the event type to be posted to the EventBusService whenever an instance has just been persisted
persisting-LifecycleEvent()	subtype of ObjectPersistingEvent (ObjectPersistingEvent.Default)	the event type to be posted to the EventBusService whenever an instance is about to be persisted
publishing()	AS_CONFIGURED, ENABLED, DISABLED (AS_CONFIGURED)	whether changes to the object should be published to the registered PublishingService.
publishing-PayloadFactory()	subtype of PublishingPayloadFactory- ForObject (none)	specifies that a custom implementation of PublishingPayloadFactoryForObject be used to create the (payload of the) published event representing the change to the object
removing-LifecycleEvent()	subtype of ObjectRemovingEvent (ObjectRemovingEvent.Default)	the event type to be posted to the EventBusService whenever an instance is about to be deleted
updated-LifecycleEvent()	subtype of ObjectUpdatedEvent (ObjectUpdatedEvent.Default)	the event type to be posted to the EventBusService whenever an instance has just been updated
updating-LifecycleEvent()	subtype of ObjectUpdatingEvent (ObjectUpdatingEvent.Default)	the event type to be posted to the EventBusService whenever an instance is about to be updated

For example:

```
@DomainObject(
    auditing=Auditing.ENABLED,
    autoCompleteRepository=CustomerRepository.class
    editing=Editing.ENABLED,                                ①
    updatedLifecycleEvent=Customer.UpdatedEvent.class

)
public class Customer {
    ...
}
```

① default value, so could be omitted

10.1. auditing()

The auditing() attribute indicates that if the object is modified, then each of its changed properties should be submitted to the AuditingService (if one has been configured), or to any

The default value for the attribute is AS_CONFIGURED, meaning that the configuration property isis.services.audit.objects is used to determine the whether the action is audited:

- all

  all changed properties of objects are audited

- none

  no changed properties of objects are audited

If there is no configuration property in isis.properties then auditing is automatically enabled for domain objects.

This default can be overridden on an object-by-object basis; if auditing() is set to ENABLED then changed properties of instances of the domain class are audited irrespective of the configured value; if set to DISABLED then the changed properties of instances are *not* audited, again irrespective of the configured value.

For example:

```
@DomainObject(
    auditing=Auditing.ENABLED  ①
)
public class Customer {
    ...
}
```

① because set to enabled, will be audited irrespective of the configured value.

10.2. autoCompleteRepository()

The autoCompleteRepository() attribute nominates a single method on a domain service as the fallback means for looking up instances of the domain object using a simple string.

For example, this might search for a customer by their name or number. Or it could search for a country based on its ISO-3 code or user-friendly name.

If you require additional control - for example restricting the returned results based on the object being interacted with - then use the autoComplete···() supporting method instead.

For example:

```
@DomainObject(
    autoCompleteRepository=CustomerRepository.class
)
public class Customer {
   ....
}
```

where:

```
@DomainService
public class CustomerRepository {
    List<Customer> autoComplete(String search);  ①
    ...
}
```

① is assumed to be called "autoComplete", and accepts a single string

10.2.1. `autoCompleteAction()`

As noted above, by default the method invoked on the repository is assumed to be called "autoComplete". The optional `autoCompleteAction()` attribute allows the method on the repository to be overridden.

For example:

```
@DomainObject(
    autoCompleteRepository=Customers.class,
    autoCompleteAction="findByName"
)
public class Customer {
   ....
}
```

where in this case `findByName` might be an existing action already defined:

```
@DomainService(natureOfService=VIEW_MENU_ONLY)
public class Customers {
    @Action(semantics=SemanticsOf.SAFE)
    public List<Customer> findByName(
        @Parameter(minLength=3)              ①
        @ParameterLayout(named="name")
        String name);
    ...
}
```

① end-user must enter minimum number of characters to trigger the query

The autocomplete action can also be a regular method, annotated using `@Programmatic`:

```
@DomainService(natureOfService=VIEW_MENU_ONLY)
public class Customers {
    @Programmatic
    public List<Customer> findByName(
        @Parameter(minLength=3)
        String name);
    ...
}
```

The method specified must be an action, that is, part of the Isis metamodel. Said another way: it must not be annotated with `@Programmatic`. However, it **can** be hidden or placed on a domain service with nature of `DOMAIN`, such that the action would not be rendered otherwise in the UI. Also, the action cannot be restricted to prototyping only.

10.3. bounded()

Some domain classes are immutable to the user, and moreover have only a fixed number of instances. Often these are "reference" ("standing") data, or lookup data/pick lists. Typical examples could include categories, countries, states, and tax or interest rate tables.

Where the number of instances is relatively small, ie bounded, then the `bounded()` attribute can be used as a hint. For such domain objects the framework will automatically allow instances to be selected; Wicket viewer displays these as a drop-down list.

For example:

```
@DomainObject(
    bounded=true,
    editing=Editing.DISABLED  ①
)
public class Currency {
    ...
}
```

① This attribute is commonly combined with `editing=DISABLED` to enforce the fact that reference data is immutable

There is nothing to prevent you from using this attribute for regular mutable entities, and indeed this is sometimes worth doing during early prototyping. However, if there is no realistic upper bound to the number of instances of an entity that might be created, generally you should use `autoComplete…()` supporting method or the `@DomainObject#autoCompleteRepository()` attribute instead.

10.4. createdLifecycleEvent()

Whenever a domain object is instantiated or otherwise becomes known to the framework, a "created" lifecycle event is fired. This is typically when the `DomainObjectContainer`'s `newTransientInstance()` is called; it will also happen if the object is simply instantiated with `new(…)`, and then the container's `injectServicesInto(…)` method is called.

Subscribers subscribe through the `EventBusService` and can use the event to obtain a reference to the object just created. The subscriber could then, for example, update the object, eg looking up state from some external datastore.

By default the event raised is `ObjectCreatedEvent.Default`. For example:

```
@DomainObject
public class ToDoItemDto {
    ...
}
```

The purpose of the `createdLifecycleEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for other lifecycle events.

For example:

```
@DomainObjectLayout(
    createdLifecycleEvent=ToDoItem.CreatedEvent.class
)
public class ToDoItem {
    public static class CreatedEvent
        extends org.apache.isis.applib.services.eventbus.ObjectCreatedEvent<ToDoItem>
{ }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

10.4.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ObjectCreatedEvent ev) {
        if(ev.getSource() instanceof ToDoItem) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.ObjectCreatedEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

10.4.2. Default, Doop and Noop events

If the `createdLifecycleEvent` attribute is not explicitly specified (is left as its default value, `ObjectCreatedEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectAnnotation.createdLifecycleEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `createdLifecycleEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ObjectCreatedEvent.Doop` as such a subclass, so setting the `createdLifecycleEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ObjectCreatedEvent.Noop`; if `createdLifecycleEvent` attribute is set to this class, then no event will be posted.

10.5. `editing()`

The `editing()` attribute determines whether a domain object's properties and collections are not editable (are read-only).

The default is `AS_CONFIGURED`, meaning that the configuration property `isis.objects.editing` is used to determine the whether the object is modifiable:

- `true`

  the object's properties and collections are modifiable.

- `false`

  the object's properties and collections are read-only, ie *not* modifiable.

If there is no configuration property in `isis.properties` then object are assumed to be modifiable.

In other words, editing can be disabled globally for an application by setting:

```
isis.objects.editing=false
```

We recommend enabling this feature; it will help drive out the underlying business operations (processes and procedures) that require objects to change; these can then be captured as business actions.

The related `editingDisabledReason()` attribute specifies the a hard-coded reason why the object's properties and collections cannot be modified directly.

This default can be overridden on an object-by-object basis; if `editing()` is set to `ENABLED` then the object's properties and collections are editable irrespective of the configured value; if set to `DISABLED` then the object's properties and collections are not editable irrespective of the configured value.

For example:

```
@DomainObject(
    editing=Editing.DISABLED,
    editingDisabledReason="Reference data, so cannot be modified"
)
public class Country {
    ...
}
```

Another interesting example of immutable reference data is to define an entity to represent individual dates; after all, for a system with an expected lifetime of 20 years that equates to only 7,300 days, a comparatively tiny number of rows to hold in a database.

10.6. loadedLifecycleEvent()

Whenever a persistent domain object is loaded from the database, a "loaded" lifecycle event is fired.

Subscribers subscribe through the `EventBusService` and can use the event to obtain a reference to

the domain object just loaded. The subscriber could then, for example, update or default values on the object (eg to support on-the-fly migration scenarios).

By default the event raised is `ObjectLoadedEvent.Default`. For example:

```
@DomainObject
public class ToDoItemDto {
    ...
}
```

The purpose of the `loadedLifecycleEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for other lifecycle events.

For example:

```
@DomainObjectLayout(
    loadedLifecycleEvent=ToDoItem.LoadedEvent.class
)
public class ToDoItem {
    public static class LoadedEvent
        extends org.apache.isis.applib.services.eventbus.ObjectLoadedEvent<ToDoItem> {
}
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

10.6.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ObjectLoadedEvent ev) {
        if(ev.getSource() instanceof ToDoItem) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.ObjectLoadedEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

10.6.2. Default, Doop and Noop events

If the `loadedLifecycleEvent` attribute is not explicitly specified (is left as its default value, `ObjectLoadedEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectAnnotation.loadedLifecycleEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `loadedLifecycleEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ObjectLoadedEvent.Doop` as such a subclass, so setting the `loadedLifecycleEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ObjectLoadedEvent.Noop`; if `loadedLifecycleEvent` attribute is set to this class, then no event will be posted.

10.7. `mixinMethod()`

The `mixinMethod()` attribute specifies the name of the method to be treated as a "reserved" method name, meaning that the mixin's name should instead be inferred from the mixin's type.

For example:

```
@DomainObject
public class Customer {

    @DomainObject(nature=Nature.MIXIN, mixinMethod="execute")
    public static class placeOrder {

        Customer customer;
        public placeOrder(Customer customer) { this.customer = customer; }

        public Customer execute(Product p, int quantity) { ... }
        public String disableExecute() { ... }
        public String validate0Execute() { ... }
    }
    ...
)
```

This allows all mixins to follow a similar convention, with the name of the mixin inferred entirely from its type ("placeOrder").

When invoked programmatically, the code reads:

```
mixin(Customer.placeOrder.class, someCustomer).execute(someProduct, 3);
```

10.8. `nature()`

The `nature()` attribute is used to characterize the domain object as either an entity (part of the domain layer) or as a view model (part of the application layer). If the domain object should be thought of as an entity, it also captures how the persistence of that entity is managed.

For example:

```
@DomainObject(nature=Nature.VIEW_MODEL)
public class PieChartAnalysis {
    ...
}
```

Specifically, the nature must be one of:

- `NOT_SPECIFIED,`

  (the default); specifies no paricular semantics for the domain class.

- `JDO_ENTITY`

  indicates that the domain object is an entity whose persistence is managed internally by Apache Isis, using the JDO/DataNucleus objectstore.

- `EXTERNAL_ENTITY`

  indicates that the domain objecct is a wrapper/proxy/stub (choose your term) to an entity that is managed by some related external system. For example, the domain object may hold just the URI to a RESTful resource of some third party REST service, or the id of some system accessible over SOAP.

  The identity of an external entity is determined solely by the state of entity's properties. The framework will automatically recreate the domain object each time it is interacted with.

- `INMEMORY_ENTITY`

  indicates that the domain object is a wrapper/proxy/stub to a "synthetic" entity, for example one that is constructed from some sort of internal memory data structure.

  The identity of an inmemory entity is determined solely by the state of entity's properties. The framework will automatically recreate the domain object each time it is interacted with.

- `MIXIN`

  indicates that the domain object is part of the domain layer, and is contributing behaviour to objects of some other type as a mixin (also known as a trait).

  Equivalent to annotating with `@Mixin`. For further discussion on using mixins, see mixins in the user guide.

- `VIEW_MODEL`

  indicates that the domain object is conceptually part of the application layer, and exists to surfaces behaviour and/or state that is aggregate of one or more domain entities.

Those natures that indicate the domain object is an entity (of some sort or another) mean then that the domain object is considered to be part of the domain model layer. As such the domain object's class cannot be annotated with `@ViewModel` or implement the `ViewModel` interface.

Under the covers Apache Isis' support for `VIEW_MODEL`, `EXTERNAL_ENTITY` and `INMEMORY_ENTITY` domain objects is identical; the state of the object is encoded into its internal OID (represented ultimately as its URL), and is recreated directly from that URL.

Because this particular implementation was originally added to Apache Isis in support of view models, the term was also used for the logically different external entities and inmemory entities.

The benefit of `nature()` is that it allows the developer to properly characterize the layer (domain vs application) that an entity lives, thus avoiding confusion as "view model" (the implementation technique) and "view model" (the application layer concept).

On the other hand, view models defined in this way do have some limitations; see `@ViewModel` for further discussion.

These limitations do *not* apply to JAXB view models. If you are using view models heavily, you may wish to restrict yourself to just the JAXB flavour.

10.9. persistedLifecycleEvent()

Whenever a (just created, still transient) domain object has been saved (INSERTed in)to the database, a "persisted" lifecycle event is fired.

Subscribers subscribe through the `EventBusService` and can use the event to obtain a reference to the domain object. The subscriber could then, for example, maintain an external datastore.

The object should *not* be modified during the persisted callback.

By default the event raised is `ObjectPersistedEvent.Default`. For example:

```
@DomainObject
public class ToDoItemDto {
    ...
}
```

The purpose of the `persistedLifecycleEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for other lifecycle events.

For example:

```
@DomainObjectLayout(
    persistedLifecycleEvent=ToDoItem.PersistedEvent.class
)
public class ToDoItem {
    public static class PersistedEvent
        extends org.apache.isis.applib.services.eventbus.ObjectPersistedEvent<
ToDoItem> { }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

10.9.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ObjectPersistedEvent ev) {
        if(ev.getSource() instanceof ToDoItem) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.ObjectPersistedEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

10.9.2. Default, Doop and Noop events

If the `persistedLifecycleEvent` attribute is not explicitly specified (is left as its default value, `ObjectPersistedEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectAnnotation.persistedLifecycleEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `persistedLifecycleEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ObjectPersistedEvent.Doop` as such a subclass, so setting the `persistedLifecycleEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ObjectPersistedEvent.Noop`; if `persistedLifecycleEvent` attribute is set to this class, then no event will be posted.

10.10. persistingLifecycleEvent()

Whenever a (just created, still transient) domain object is about to be saved (INSERTed in)to the database, a "persisting" lifecycle event is fired.

Subscribers subscribe through the `EventBusService` and can use the event to obtain a reference to the domain object. The subscriber could then, for example, update the object, or it could use it maintain an external datastore. One possible application is to maintain a full-text search database

using Apache Lucene or similar.

Another use case is to maintain "last updated by"/"last updated at" properties. While you can roll your own, note that the framework provides built-in support for this use case through the `Timestampable` role interface.

By default the event raised is `ObjectPersistingEvent.Default`. For example:

```
@DomainObject
public class ToDoItemDto {
    ...
}
```

The purpose of the `persistingLifecycleEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for other lifecycle events.

For example:

```
@DomainObjectLayout(
    persistingLifecycleEvent=ToDoItem.PersistingEvent.class
)
public class ToDoItem {
    public static class PersistingEvent
        extends org.apache.isis.applib.services.eventbus.ObjectPersistingEvent
<ToDoItem> { }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

10.10.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ObjectPersistingEvent ev) {
        if(ev.getSource() instanceof ToDoItem) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.ObjectPersistingEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

10.10.2. Default, Doop and Noop events

If the `persistingLifecycleEvent` attribute is not explicitly specified (is left as its default value, `ObjectPersistingEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectAnnotation.persistingLifecycleEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `persistingLifecycleEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ObjectPersistingEvent.Doop` as such a subclass, so setting the `persistingLifecycleEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ObjectPersistingEvent.Noop`; if `persistingLifecycleEvent` attribute is set to this class, then no event will be posted.

10.11. `objectType()`

The `objectType()` attribute is used to provide a unique alias for the object's class name.

This value is used internally to generate a string representation of an objects identity (the `Oid`). This can appear in several contexts, including:

- as the value of `Bookmark#getObjectType()` and in the `toString()` value of `Bookmark` (see `BookmarkService`)
    - and thus in the "table-of-two-halves" pattern, as per the (non-ASF) Incode Platform's poly module
- in the serialization of `OidDto` in the command and interaction schemas
- in the URLs of the RestfulObjects viewer
- in the URLs of the Wicket viewer (in general and in particular if copying URLs)
- in XML snapshots generated by the `XmlSnapshotService`

10.11.1. Examples

For example:

```
@DomainObject(
    objectType="orders.Order"
)
public class Order {
    ...
}
```

10.11.2. Precedence

The rules of precedence are:

1. `@Discriminator`
2. `@DomainObject#objectType`, or `@ObjectType` (deprecated)
3. `@PersistenceCapable`, if at least the `schema` attribute is defined.

   If both `schema` and `table` are defined, then the value is "schema.table". If only `schema` is defined, then the value is "schema.className".

4. Fully qualified class name of the entity.

This might be obvious, but to make explicit: we recommend that you always specify an object type for your domain objects.

Otherwise, if you refactor your code (change class name or move package), then any externally held references to the OID of the object will break. At best this will require a data migration in the database; at worst it could cause external clients accessing data through the Restful Objects viewer to break.

If the object type is not unique across all domain classes then the framework will fail-fast and fail to boot. An error message will be printed in the log to help you determine which classes have duplicate object tyoes.

10.12. `publishing()`

The `publishing()` attribute determines whether and how a modified object instance is published via the registered implementation of a `PublishingService`) or `PublisherService`. This attribute is also supported for actions, where it controls whether action invocations are published as events, and for `@Property#publishing()`, where it controls whether property edits are published as events.

A common use case is to notify external "downstream" systems of changes in the state of the Isis application.

The default value for the attribute is `AS_CONFIGURED`, meaning that the configuration property

`isis.services.publish.objects` is used to determine the whether the action is published:

- `all`

  all changed objects are published

- `none`

  no changed objects are published

If there is no configuration property in `isis.properties` then publishing is automatically enabled for domain objects.

This default can be overridden on an object-by-object basis; if `publishing()` is set to `ENABLED` then changed instances of the domain class are published irrespective of the configured value; if set to `DISABLED` then the changed instances are *not* published, again irrespective of the configured value.

For example:

```
@DomainObject(
    publishing=Publishing.ENABLED  ①
)
public class InterestRate {
    ...
}
```

① because set to enabled, will be published irrespective of the configured value.

10.12.1. `publishingPayloadFactory()`

The (optional) related `publishingPayloadFactory()` specifies the class to use to create the (payload of the) event to be published by the publishing factory.

Rather than simply broadcast that the object was changed, the payload factory allows a "fatter" payload to be instantiated that can eagerly push commonly-required information to all subscribers. For at least some subscribers this should avoid the necessity to query back for additional information.

Be aware that this attribute is only honoured by the (deprecated) `PublishingService`, so should itself be considered as deprecated. It is ignored by the replacement `PublisherService`,

10.13. removingLifecycleEvent()

Whenever a (persistent) domain object is about to be removed (DELETEd) from the database, a "removing" lifecycle event is fired.

Subscribers subscribe through the `EventBusService` and can use the event to obtain a reference to the domain object. The subscriber could then, for example, could use it maintain an external

datastore. One possible application is to maintain a full-text search database using Apache Lucene or similar.

Another use case is to maintain "last updated by"/"last updated at" properties. While you can roll your own, note that the framework provides built-in support for this use case through the `Timestampable` role interface.

By default the event raised is `ObjectRemovingEvent.Default`. For example:

```
@DomainObject
public class ToDoItemDto {
    ...
}
```

The purpose of the `removingLifecycleEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for other lifecycle events.

For example:

```
@DomainObjectLayout(
    removingLifecycleEvent=ToDoItem.RemovingEvent.class
)
public class ToDoItem {
    public static class RemovingEvent
        extends org.apache.isis.applib.services.eventbus.ObjectRemovingEvent<ToDoItem>
{ }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

10.13.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ObjectRemovingEvent ev) {
        if(ev.getSource() instanceof ToDoItem) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.ObjectRemovingEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

10.13.2. Default, Doop and Noop events

If the `removingLifecycleEvent` attribute is not explicitly specified (is left as its default value, `ObjectRemovingEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectAnnotation.removingLifecycleEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `removingLifecycleEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ObjectRemovingEvent.Doop` as such a subclass, so setting the `removingLifecycleEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ObjectRemovingEvent.Noop`; if `removingLifecycleEvent` attribute is set to this class, then no event will be posted.

10.14. updatingLifecycleEvent()

Whenever a (persistent) domain object has been modified and is about to be updated to the database, an "updating" lifecycle event is fired.

Subscribers subscribe through the `EventBusService` and can use the event to obtain a reference to the domain object. The subscriber could then, for example, update the object, or it could use it maintain an external datastore. One possible application is to maintain a full-text search database using Apache Lucene or similar.

Another use case is to maintain "last updated by"/"last updated at" properties. While you can roll your own, note that the framework provides built-in support for this use case through the `Timestampable` role interface.

By default the event raised is `ObjectUpdatingEvent.Default`. For example:

```
@DomainObject
public class ToDoItemDto {
    ...
}
```

The purpose of the `updatingLifecycleEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for other lifecycle events.

For example:

```
@DomainObjectLayout(
    updatingLifecycleEvent=ToDoItem.UpdatingEvent.class
)
public class ToDoItem {
    public static class UpdatingEvent
        extends org.apache.isis.applib.services.eventbus.ObjectUpdatingEvent<ToDoItem>
{ }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

10.14.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ObjectUpdatingEvent ev) {
        if(ev.getSource() instanceof ToDoItem) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.ObjectUpdatingEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

10.14.2. Default, Doop and Noop events

If the `updatingLifecycleEvent` attribute is not explicitly specified (is left as its default value, `ObjectUpdatingEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectAnnotation.updatingLifecycleEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `updatingLifecycleEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ObjectUpdatingEvent.Doop` as such a subclass, so setting the `updatingLifecycleEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ObjectUpdatingEvent.Noop`; if `updatingLifecycleEvent` attribute is set to this class, then no event will be posted.

10.15. updatedLifecycleEvent()

Whenever a (persistent) domain object has been modified and has been updated in the database, an "updated" lifecycle event is fired.

Subscribers subscribe through the `EventBusService` and can use the event to obtain a reference to the domain object.

The object should *not* be modified during the updated callback.

By default the event raised is `ObjectUpdatedEvent.Default`. For example:

```
@DomainObject
public class ToDoItemDto {
    ...
}
```

The purpose of the `updatedLifecycleEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for other lifecycle events.

For example:

```
@DomainObjectLayout(
    updatedLifecycleEvent=ToDoItem.UpdatedEvent.class
)
public class ToDoItem {
    public static class UpdatedEvent
        extends org.apache.isis.applib.services.eventbus.ObjectUpdatedEvent<ToDoItem>
{ }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

10.15.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ObjectUpdatedEvent ev) {
        if(ev.getSource() instanceof ToDoItem) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.ObjectUpdatedEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

10.15.2. Default, Doop and Noop events

If the `updatedLifecycleEvent` attribute is not explicitly specified (is left as its default value, `ObjectUpdatedEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectAnnotation.updatedLifecycleEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `updatedLifecycleEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `ObjectUpdatedEvent.Doop` as such a subclass, so setting the `updatedLifecycleEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `ObjectUpdatedEvent.Noop`; if `updatedLifecycleEvent` attribute is set to this class, then no event will be posted.

Chapter 11. `@DomainObjectLayout`

The `@DomainObjectLayout` annotation applies to domain classes, collecting together all UI hints within a single annotation.

For view models that have been annotated with `@ViewModel` the equivalent `@ViewModelLayout` can be used.

The table below summarizes the annotation's attributes.

Table 13. `@DomainObjectLayout` attributes

Attribute	Values (default)	Description
`bookmarking()`	`AS_ROOT`, `AS_CHILD`, `NEVER` (`NEVER`)	whether (and how) this domain object should be automatically bookmarked
`cssClass()`	Any string valid as a CSS class	the css class that a domain class (type) should have, to allow more targetted styling in `application.css`
`cssClassFa()`	Any valid Font awesome icon name	specify a font awesome icon for the action's menu link or icon.
`cssClassFaPosition()`	`LEFT`, `RIGHT` (`LEFT`)	Currently unused.
`cssClassUiEvent()`	subtype of `CssClassUiEvent` (`CssClassUiEvent.Default`)	the event type to be posted to the `EventBusService` to obtain a CSS class for the domain object.
`describedAs()`	String.	description of this class, eg to be rendered in a tooltip.
`iconUiEvent()`	subtype of `IconUiEvent` (`IconUiEvent.Default`)	the event type to be posted to the `EventBusService` to obtain the icon (name) for the domain object.
`named()`	String.	to override the name inferred from the action's name in code. A typical use case is if the desired name is a reserved Java keyword, such as `default` or `package`.
`paged()`	Positive integer	the page size for instances of this class when rendered within a table (as returned from an action invocation)
`plural()`	String.	the plural name of the class
`titleUiEvent()`	subtype of `TitleUiEvent` (`TitleUiEvent.Default`)	the event type to be posted to the `EventBusService` to obtain the title for the domain object.

For example:

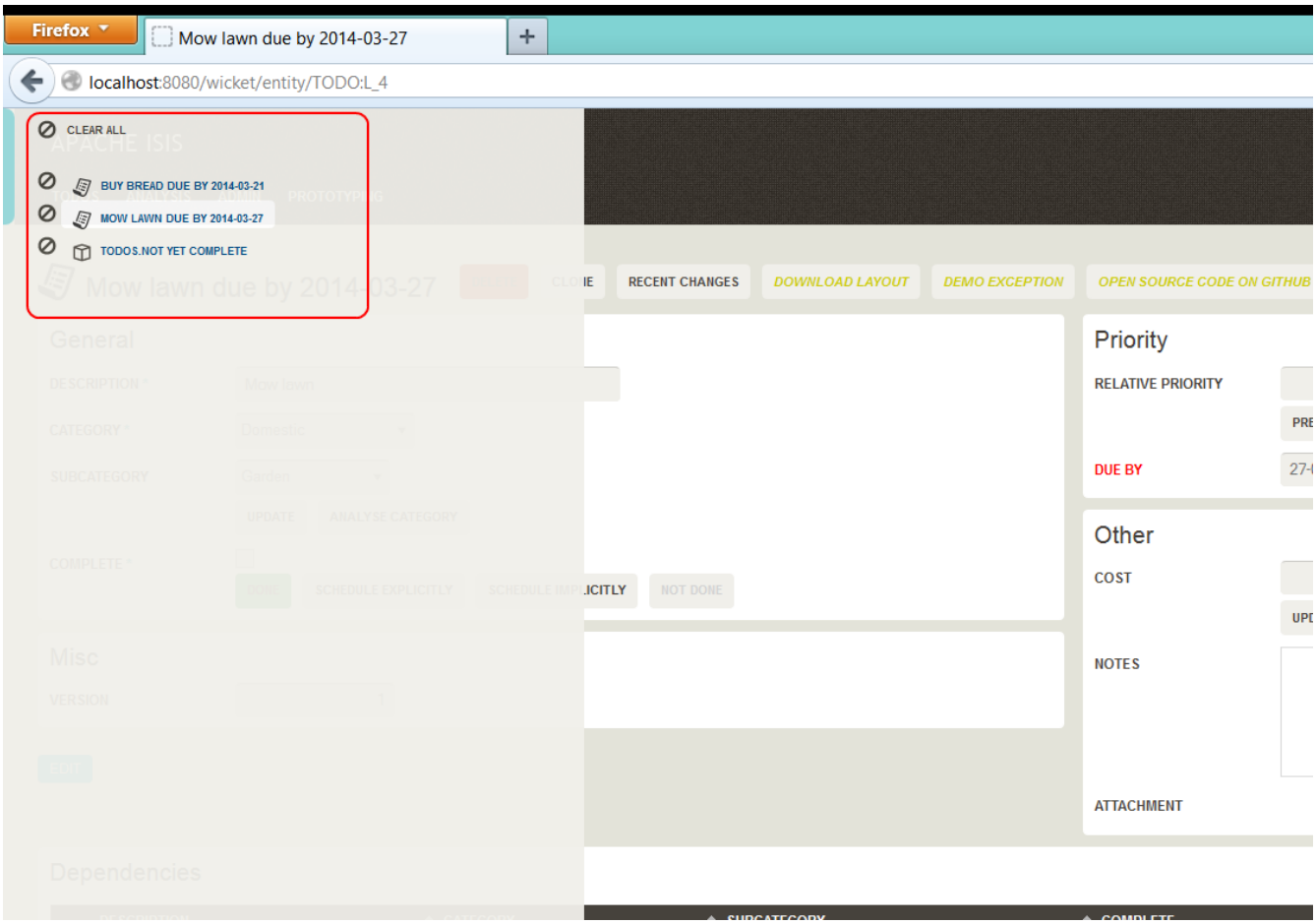
```
@DomainObjectLayout(
    cssClass="x-key",
    cssClassFa="fa-checklist",
    describedAs="Capture a task that you need to do",
    named="ToDo",
    paged=30,
    plural="ToDo List")
)
public class ToDoItem {
    ...
}
```

Note that there is (currently) no support for specifying UI hints for domain objects through the dynamic `.layout.json` file (only for properties, collections and actions are supported).

11.1. `bookmarking()`

The `bookmarking()` attribute indicates that an entity is automatically bookmarked. This attribute is also supported for domain objects.

(In the Wicket viewer), a link to a bookmarked object is shown in the bookmarks panel:



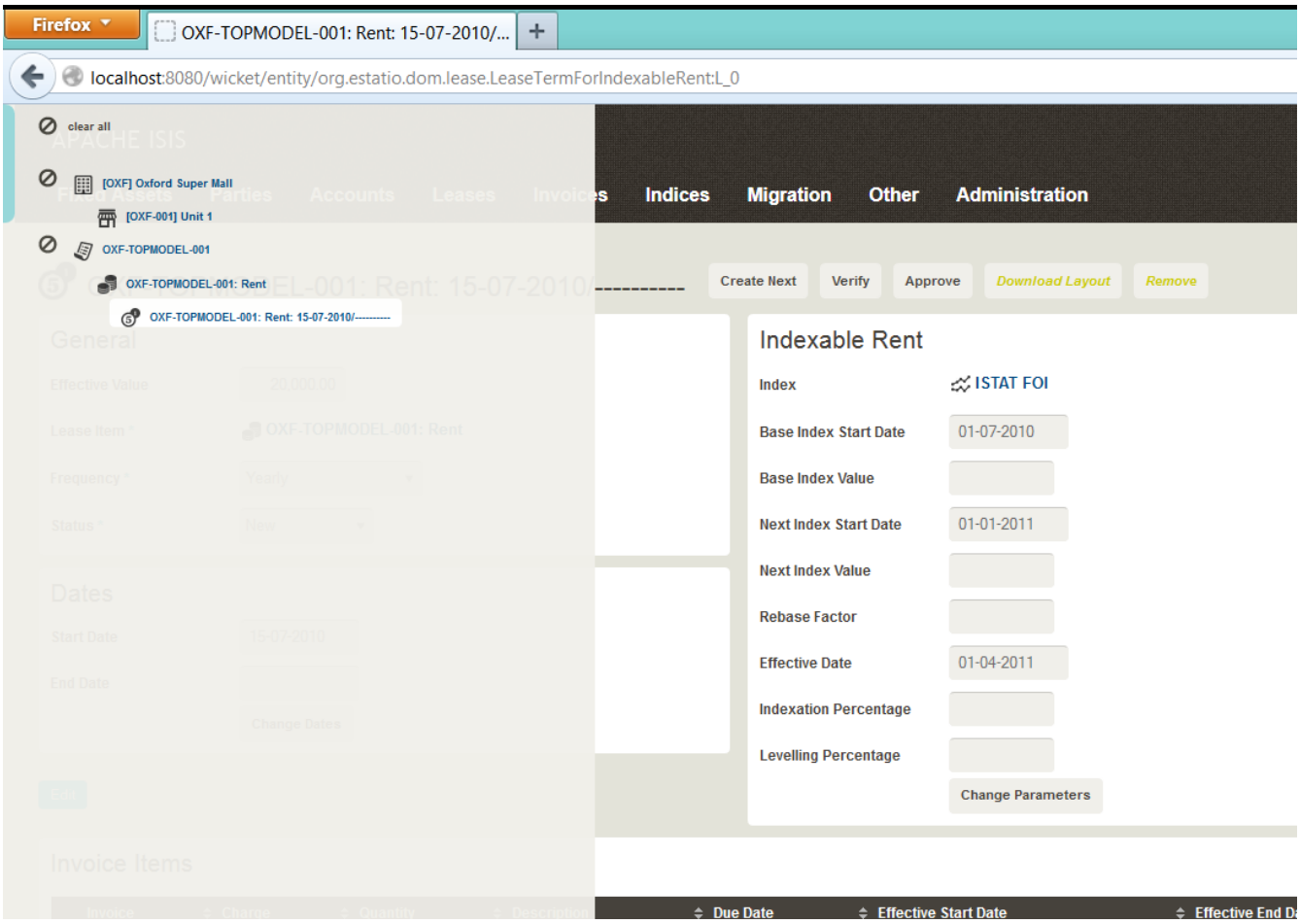
Note that this screenshot shows an earlier version of the Wicket viewer UI (specifically, pre 1.8.0).

For example:

```
@DomainObject(bookmarking=BookmarkPolicy.AS_ROOT)
public class ToDoItem ... {
    ...
}
```

indicates that the `ToDoItem` class is bookmarkable:

It is also possible to nest bookmarkable entities. For example, this screenshot is taken from Estatio:



Note that this screenshot shows an earlier version of the Wicket viewer UI (specifically, pre 1.8.0).

For example, the `Property` entity "[OXF] Oxford Super Mall" is a root bookmark, but the `Unit` child entity "[OXF-001] Unit 1" only appears as a bookmark *but only if* its parent `Property` has already been bookmarked.

This is accomplished with the following annotations:

```
@DomainObject(bookmarking=BookmarkPolicy.AS_ROOT)
public class Property { ... }
```

and

```
@DomainObject(bookmarking=BookmarkPolicy.AS_CHILD)
public abstract class Unit { ... }
```

The nesting can be done to any level; the Estatio screenshot also shows a bookmark nesting `Lease` > `LeaseItem` > `LeaseTerm` (3 levels deep).

11.2. `cssClass()`

The `cssClass()` attribute can be used to render additional CSS classes in the HTML (a wrapping `<div>`) that represents the domain object. Application-specific CSS can then be used to target and adjust the UI representation of that particular element.

This attribute can also be applied to domain objects, view models, actions properties, collections and parameters.

For example:

```
@DomainObject(
    cssClass="x-core-entity"
)
public class ToDoItem { ... }
```

The similar `@DomainObjectLayout#cssClassFa()` annotation attribute is also used as a hint to apply CSS, but in particular to allow Font Awesome icons to be rendered as the icon for classes.

11.3. `cssClassFa()`

The `cssClassFa()` attribute is used to specify the name of a Font Awesome icon name, to be rendered as the domain object's icon.

These attributes can also be applied to view models to specify the object's icon, and to actions to specify an icon for the action's representation as a button or menu item.

If necessary the icon specified can be overridden by a particular object instance using the `iconName()` method.

For example:

```
@DomainObjectLayout(
    cssClassFa="fa-check-circle"
)
public class ToDoItem { ... }
```

There can be multiple "fa-" classes, eg to mirror or rotate the icon. There is no need to include the mandatory `fa` "marker" CSS class; it will be automatically added to the list. The `fa-` prefix can also be omitted from the class names; it will be prepended to each if required.

The related `cssClassFaPosition()` attribute is currently unused for domain objects; the icon is always rendered to the left.

The similar `@DomainObjectLayout#cssClass()` annotation attribute is also used as a hint to apply CSS, but for wrapping the representation of an object or object member so that it can be styled in an application-specific way.

11.4. cssClassUiEvent()

Whenever a domain object is to be rendered, the framework fires off an CSS class UI event to obtain a CSS class to use in any wrapping `<div>`s and `<span>`s that render the domain object. This is as an alternative to implementing `cssClass()` reserved method. (If `cssClass()` is present, then it will take precedence).

Subscribers subscribe through the `EventBusService` and can use obtain a reference to the domain object from the event. From this they can, if they wish, specify a CSS class for the domain object using the event's API.

The feature was originally introduced so that `@XmlRootElement`-annotated view models could be kept as minimal as possible, just defining the data. UI events allow subscribers to provide UI hints, while mixins can be used to provide the behaviour.

By default the event raised is `CssClassUiEvent.Default`. For example:

```
@DomainObjectLayout
public class ToDoItemDto {
    ...
}
```

The purpose of the `cssClassUiEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for titles and icons.

For example:

```
@DomainObjectLayout(
    iconUiEvent=ToDoItemDto.CssClassUiEvent.class
)
public class ToDoItemDto {
    public static class CssClassUiEvent
        extends org.apache.isis.applib.services.eventbus.CssClassUiEvent<ToDoItemDto>
{ }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

11.4.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(CssClassUiEvent ev) {
        if(ev.getSource() instanceof ToDoItemDto) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItemDto.CssClassUiEvent ev) {
        ...
    }
}
```

The subscriber should then use `CssClassUiEvent#setCssClass(…)` to actually specify the CSS class to be used.

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

11.4.2. Default, Doop and Noop events

If the `cssClassUiEvent` attribute is not explicitly specified (is left as its default value, `CssClassUiEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectLayoutAnnotation.cssClassUiEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `cssClassUiEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `CssClassUiEvent.Doop` as such a subclass, so setting the `cssClassUiEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `CssClassUiEvent.Noop`; if `cssClassUiEvent` attribute is set to this class, then no event will be posted.

11.4.3. Raising events programmatically

Normally events are only raised for interactions through the UI. However, events can be raised programmatically either by calling the `EventBusService` API directly, or as a result of calling the `DomainObjectContainer`'s `cssClassOf(…)` method.

11.5. `describedAs()`

The `describedAs()` attribute is used to provide a short description of the domain object to the user. In the Wicket viewer it is displayed as a 'tool tip'. The attribute can also be specified for collections, properties, actions, parameters and view models.

For example:

```
@DescribedAs("A customer who may have originally become known to us via " +
             "the marketing system or who may have contacted us directly.")
public class ProspectiveSale {
   ...
}
```

11.6. iconUiEvent()

Whenever a domain object is to be rendered, the framework fires off an icon UI event to obtain an icon (name) for the object (if possible). This is as an alternative to implementing `iconName()` reserved method. (If `iconName()` is present, then it will take precedence).

Subscribers subscribe through the `EventBusService` and can use obtain a reference to the domain object from the event. From this they can, if they wish, specify an icon name for the domain object using the event's API.

The feature was originally introduced so that `@XmlRootElement`-annotated view models could be kept as minimal as possible, just defining the data. UI events allow subscribers to provide UI hints, while mixins can be used to provide the behaviour.

By default the event raised is `IconUiEvent.Default`. For example:

```
@DomainObjectLayout
public class ToDoItemDto {
    ...
}
```

The purpose of the `iconUiEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for titles and CSS classes.

For example:

```
@DomainObjectLayout(
    iconUiEvent=ToDoItemDto.IconUiEvent.class
)
public class ToDoItemDto {
    public static class IconUiEvent
        extends org.apache.isis.applib.services.eventbus.IconUiEvent<ToDoItemDto> { }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

11.6.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(IconUiEvent ev) {
        if(ev.getSource() instanceof ToDoItemDto) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItemDto.IconUiEvent ev) {
        ...
    }
}
```

The subscriber should then use `IconUiEvent#setIconName(…)` to actually specify the icon name to be used.

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

11.6.2. Default, Doop and Noop events

If the `iconUiEvent` attribute is not explicitly specified (is left as its default value, `IconUiEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectLayoutAnnotation.iconUiEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `iconUiEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `IconUiEvent.Doop` as such a subclass, so setting the `iconUiEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `IconUiEvent.Noop`; if `iconUiEvent` attribute is set to this class, then no event will be posted.

11.6.3. Raising events programmatically

Normally events are only raised for interactions through the UI. However, events can be raised programmatically either by calling the `EventBusService` API directly, or as a result of calling the `DomainObjectContainer`'s `iconNameOf(…)` method.

11.7. `named()`

The `named()` attribute explicitly specifies the domain object's name, overriding the name that would normally be inferred from the Java source code. The attribute can also be specified for actions, collections, properties, parameters, view models and domain services.

Following the don't repeat yourself principle, we recommend that you only use this attribute when the desired name cannot be used in Java source code. Examples of that include a name that would be a reserved Java keyword (eg "package"), or a name that has punctuation, eg apostrophes.

For example:

```
@DomainObjectLayout(
   named="Customer"
)
public class CustomerImpl implements Customer{
   ...
}
```

It's also possible to specify a plural form of the name, used by the framework when rendering a standalone collection of the domain object.

The framework also provides a separate, powerful mechanism for internationalization.

11.8. paged()

The `paged()` attribute specifies the number of rows to display in a standalone collection, as returned from an action invocation. This attribute can also be applied to collections and view models.

The RestfulObjects viewer currently does not support paging. The Wicket viewer *does* support paging, but note that the paging is performed client-side rather than server-side.

We therefore recommend that large collections should instead be modelled as actions (to allow filtering to be applied to limit the number of rows).

For example:

```
@DomainObjectLayout(paged=15)
public class Order {
    ...
}
```

It is also possible to specify a global default for the page size of standalone collections, using the configuration property `isis.viewer.paged.standalone`.

11.9. plural()

When Apache Isis displays a standalone collection of several objects, it will label the collection

using the plural form of the object type.

By default the plural name will be derived from the end of the singular name, with support for some basic English language defaults (eg using "ies" for names ending with a "y").

The `plural()` attribute allows the plural form of the class name to be specified explicitly. This attribute is also supported for view models.

For example:

```
@DomainObjectLayout(plural="Children")
public class Child {
    ...
}
```

11.10. titleUiEvent()

Whenever a domain object is to be rendered, the framework fires off a title UI event to obtain a title for the object. This is as an alternative to implementing `title()` reserved method, or using the `@Title` annotation, within the class itself. (If either `title()` or `@Title` are present, then they will take precedence).

Subscribers subscribe through the `EventBusService` and can use obtain a reference to the domain object from the event. From this they can, if they wish, specify a title for the domain object using the event's API.

The feature was originally introduced so that `@XmlRootElement`-annotated view models could be kept as minimal as possible, just defining the data. UI events allow subscribers to provide UI hints, while mixins can be used to provide the behaviour.

By default the event raised is `TitleUiEvent.Default`. For example:

```
@DomainObjectLayout
public class ToDoItemDto {
    ...
}
```

The purpose of the `titleUiEvent()` attribute is to allows a custom subclass to be emitted instead. A similar attribute is available for icon names and CSS classes.

For example:

```
@DomainObjectLayout(
    titleUiEvent=ToDoItemDto.TitleUiEvent.class
)
public class ToDoItemDto {
    public static class TitleUiEvent
        extends org.apache.isis.applib.services.eventbus.TitleUiEvent<ToDoItemDto> { }
    ...
}
```

The benefit is that subscribers can be more targeted as to the events that they subscribe to.

11.10.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(TitleUiEvent ev) {
        if(ev.getSource() instanceof ToDoItemDto) { ... }
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItemDto.TitleUiEvent ev) {
        ...
    }
}
```

The subscriber should then use either `TitleUiEvent#setTranslatableTitle(…)` or `TitleUiEvent#setTitle(…)` to actually specify the title to be used.

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

11.10.2. Default, Doop and Noop events

If the `titleUiEvent` attribute is not explicitly specified (is left as its default value, `TitleUiEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.domainObjectLayoutAnnotation.titleUiEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `titleUiEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `TitleUiEvent.Doop` as such a subclass, so setting the `titleUiEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `TitleUiEvent.Noop`; if `titleUiEvent` attribute is set to this class, thn no event will be posted.

11.10.3. Raising events programmatically

Normally events are only raised for interactions through the UI. However, events can be raised programmatically either by calling the `EventBusService` API directly, or as a result of calling the `DomainObjectContainer`'s `titleOf(…)` method.

Chapter 12. @DomainService

The `@DomainService` annotation indicates that the (concrete) class should be automatically instantiated as a domain service.

Domain services with this annotation do NOT need to be registered explicitly in `isis.properties`; they will be discovered automatically on the CLASSPATH.

The table below summarizes the annotation's attributes.

Table 14. `@DomainService` attributes

Attribute	Values (default)	Description
`nature()`	`VIEW`, `VIEW_MENU_ONLY`, `VIEW_CONTRIBUTIONS_ONLY`, `VIEW_REST_ONLY`, `DOMAIN` (`VIEW`)	the nature of this service: providing actions for menus, or as contributed actions, or for the RestfulObjects REST API, or neither
`objectType()`		equivalent to `@DomainObject#objectType()`, specifies the objectType of the service. The instanceId for services is always "1".
`repositoryFor()`		if this domain service acts as a repository for an entity type, specify that entity type. This is used to determine an icon to use for the service (eg as shown in action prompts).
`menuOrder()`		Deprecated in 1.8.0; use instead `@DomainServiceLayout#menuOrder()`

For example:

```
@DomainService(
    nature=NatureOfService.DOMAIN,
    repositoryFor=Loan.class
)
public class LoanRepository {
    @Programmatic
    public List<Loan> findLoansFor(Borrower borrower) { ... }
}
```

12.1. nature()

By default, a domain service's actions will be rendered in the application menu bar *and* be contributed *and* appear in the REST API *and* (of course) be available to invoke programmatically wherever that domain service is injected. This is great for initial prototyping, but later on you may prefer to add a little more structure. This is the purpose of the `nature()` attribute: to indicates the intent of (all of) the actions defined within the domain service.

The values of the enum are:

- `VIEW`

  The default; the service's actions appear on menu bars, can be contributed, appear in the REST API

- `VIEW_MENU_ONLY`

  The service's actions appear on menus and in the REST API, but are not contributed to domain objects or view models

- `VIEW_CONTRIBUTIONS_ONLY`

  The service's actions are intended only to be used as contributed actions/associations to domain objects and view models.

  The related `@ActionLayout#contributedAs()` determines whether any given (1-arg) action is contributed as an association rather than an action.

- `VIEW_REST_ONLY`

  The service's actions are intended only to be listed in the REST API exposed by the RestfulObjects viewer.

- `DOMAIN`

  The service and its actions are only intended to be invoked programmatically; they are a domain layer responsibility.

The actual class name of the domain service is only rendered for the `VIEW`, `VIEW_MENU_ONLY` and `VIEW_REST_ONLY` natures. Thus, you might also want to adopt naming conventions for your domain classes so you can infer the nature from the class. For example, the naming convention adopted (by and large) by the (non-ASF) Incode Platform is `ProgrammaticServices` or `Repository` as a suffix for `DOMAIN` services, and `Contributions` as a suffix for `VIEW_CONTRIBUTIONS_ONLY` services.

For example:

```
@DomainService(
    nature=NatureOfService.VIEW_CONTRIBUTIONS_ONLY
)
public class LoanContributions {                                    ①
    @Action(semantics=SemanticsOf.SAFE)                             ②
    @ActionLayout(contributed=Contributed.AS_ASSOCIATION )
    public List<Loan> currentLoans(Borrower borrower) { ... }
    public Borrower newLoan(Borrower borrower, Book book) { ... }
}
```

① `Contributions` as a suffix for a domain service that contributes a number of actions to `Borrower`s. Note that `Borrower` could be a (marker) interface, so this functionality is "mixed in" merely by the class (eg `LibraryMember`) implementing this interface

② actions contibuted as associations (a collection in this case) must have safe semantics

Another example:

```
@DomainService(
    nature=NatureOfService.DOMAIN
)
public class LoanRepository {                                    ①
    @Programmatic                                                ②
    public List<Loan> findLoansFor(Borrower borrower) { ... }
}
```

① `Repository` as a suffix for a domain-layer service

② methods on `DOMAIN` services are often `@Programmatic`; they will never be exposed in the UI, so there's little point in including them in Apache Isis' metamodel

A final example:

```
@DomainService(
    nature=NatureOfService.VIEW_MENU_ONLY
)
public class Loans {                                 ①
    @Action(semantics=SemanticsOf.SAFE)
    public List<Loan> findOverdueLoans() { ... }
    @Inject
    LoanRepository loanRepository;                   ②
}
```

① name is intended to be rendered in the UI

② it's common for domain-layer domain services to be injected into presentation layer services (such as `VIEW_MENU_ONLY` and `VIEW_CONTRIBUTIONS_ONLY`).

12.2. `objectType()`

The `objectType()` attribute is used to provide a unique alias for the domain service's class name.

This value is used internally to generate a string representation of an service identity (the `Oid`). This can appear in several contexts, including:

- as the value of `Bookmark#getObjectType()` and in the `toString()` value of `Bookmark` (see `BookmarkService`)
- in the serialization of `OidDto` in the command and interaction schemas
- in the URLs of the RestfulObjects viewer
- in the URLs of the Wicket viewer (specifically, for bookmarked actions)

12.2.1. Example

For example:

```
@DomainService(
    objectType="orders.OrderMenu"
)
public class OrderMenu {
    ...
}
```

12.2.2. Precedence

The rules of precedence are:

1. `@DomainService#objectType`
2. `getId()`
3. The fully qualified class name.

This might be obvious, but to make explicit: we recommend that you always specify an object type for your domain services.

Otherwise, if you refactor your code (change class name or move package), then any externally held references to the OID of the service will break. At best this will require a data migration in the database; at worst it could cause external clients accessing data through the Restful Objects viewer to break.

If the object type is not unique across all domain classes then the framework will fail-fast and fail to boot. An error message will be printed in the log to help you determine which classes have duplicate object tyoes.

12.3. `repositoryFor()`

The `repositoryFor()` attribute is intended for domain services (probably with a `nature=DOMAIN`) that are intended to act as repositories for domain entities.

For example:

```
@DomainService(
    nature=NatureOfService.DOMAIN,
    repositoryFor=Loan.class
)
public class LoanRepository {
    @Programmatic
    public List<Loan> findLoansFor(Borrower borrower) { ... }
}
```

Currently the metadata is unused; one planned use is to infer the icon for the domain service from the icon of the nominated entity.

Chapter 13. `@DomainServiceLayout`

The `@DomainServiceLayout` annotation applies to domain services, collecting together all view layout semantics within a single annotation.

You will also find some additional material in the object layout chapter.

The table below summarizes the annotation's attributes.

Table 15. `@DomainServiceLayout` attributes

Attribute	Values (default)	Description
`menuBar()`	`PRIMARY`, `SECONDARY`, `TERTIARY` (`PRIMARY`).	the menubar in which the menu that holds this service's actions should reside.
`menuOrder()`		the order of the service's menu with respect to other service's.
`named()`	string, eg "Customers"	name of this class (overriding the name derived from its name in code)

For example:

```
@DomainService
@DomainServiceLayout(
    menuBar=MenuBar.PRIMARY,
    menuOrder="100",
    named="ToDos"
)
public class ToDoItems {
    ...
}
```

Note that there is (currently) no support for specifying UI hints for domain services through the dynamic `.layout.json` file (only for properties, collections and actions are supported).

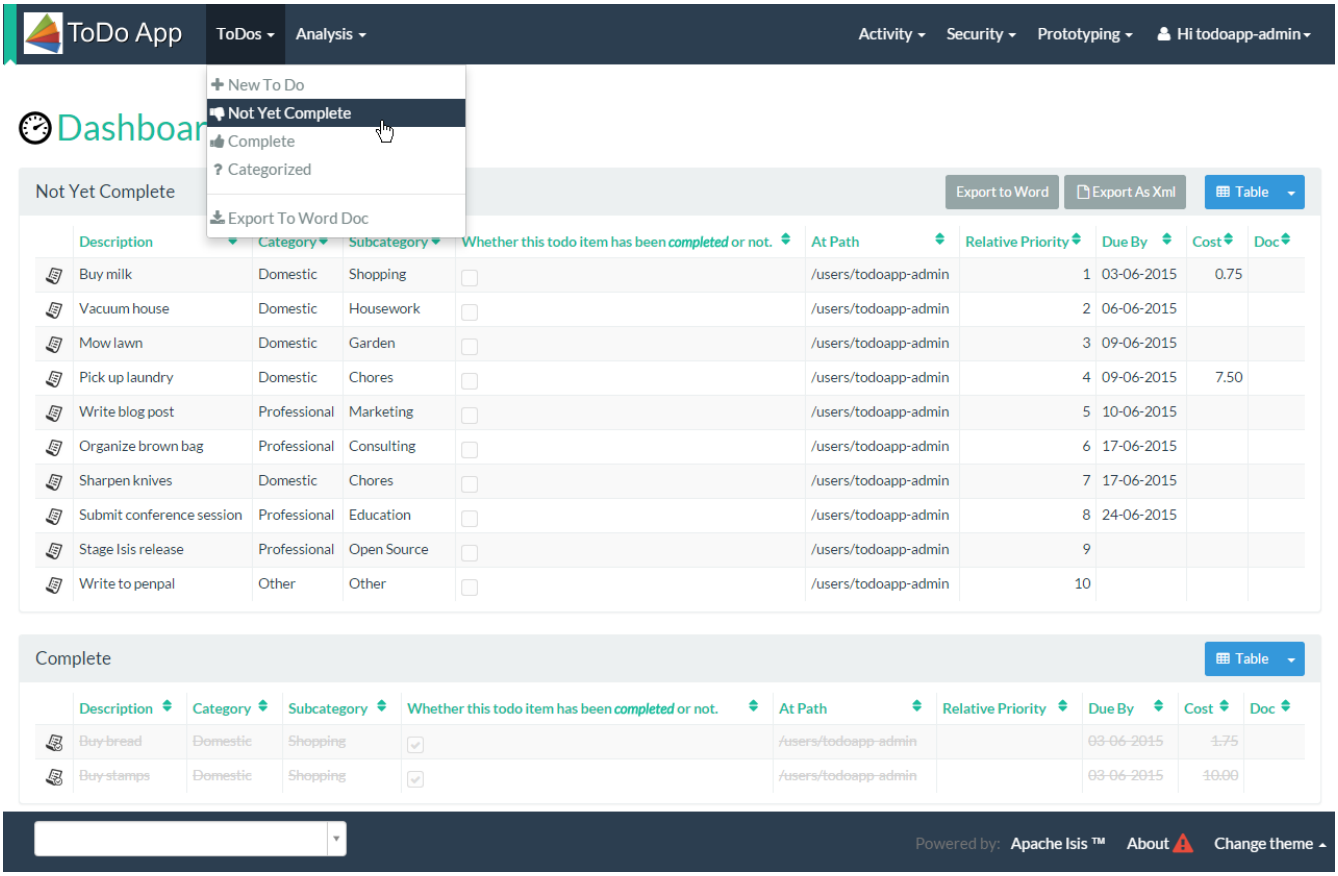
13.1. `menuBar()`

The `menuBar()` attribute is a hint to specify where on the application menu a domain service's actions should be rendered.

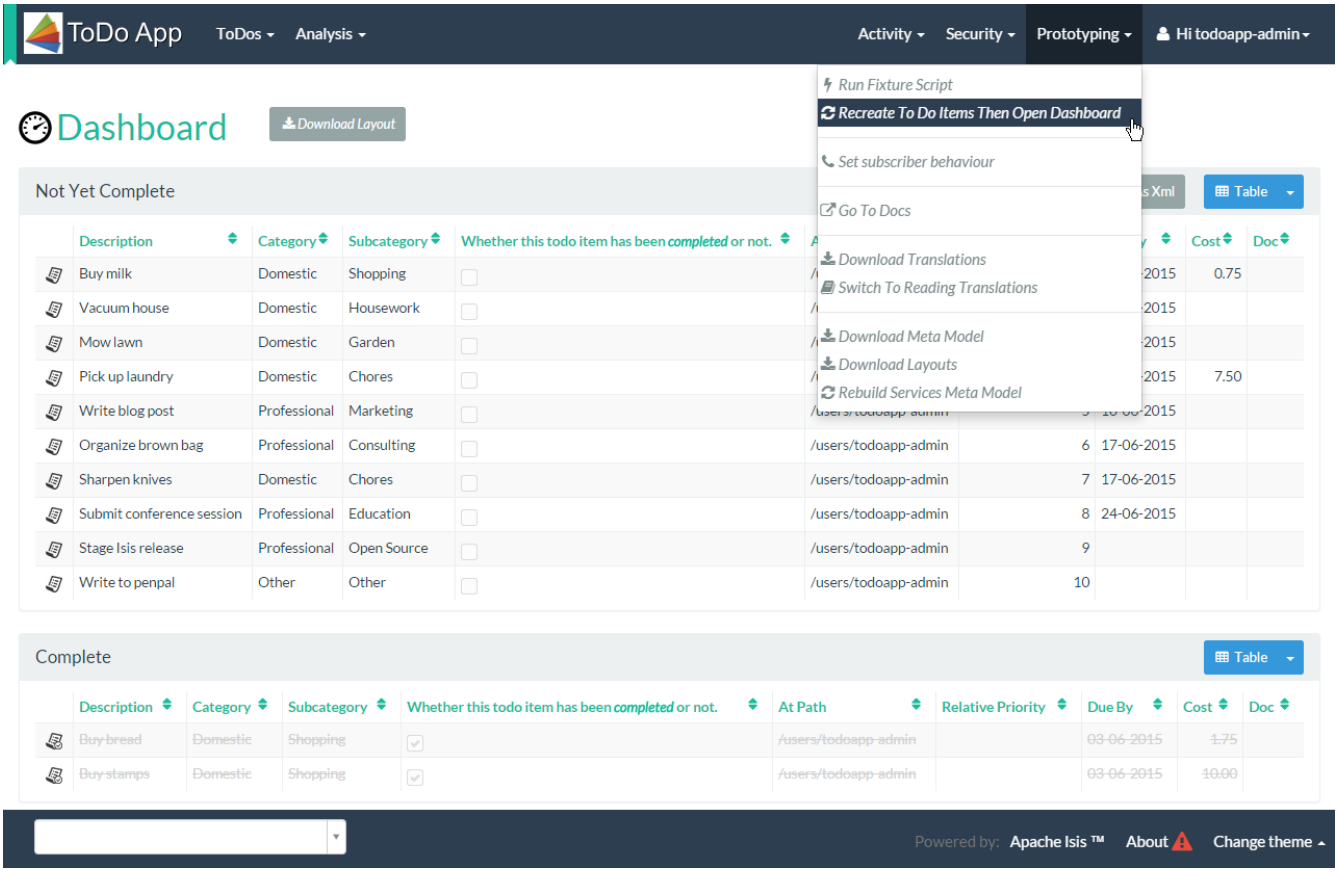
For example:

```
@DomainService
@DomainServiceLayout(menuBar=MenuBar.PRIMARY)
public class ToDoItems {
    ...
}
```

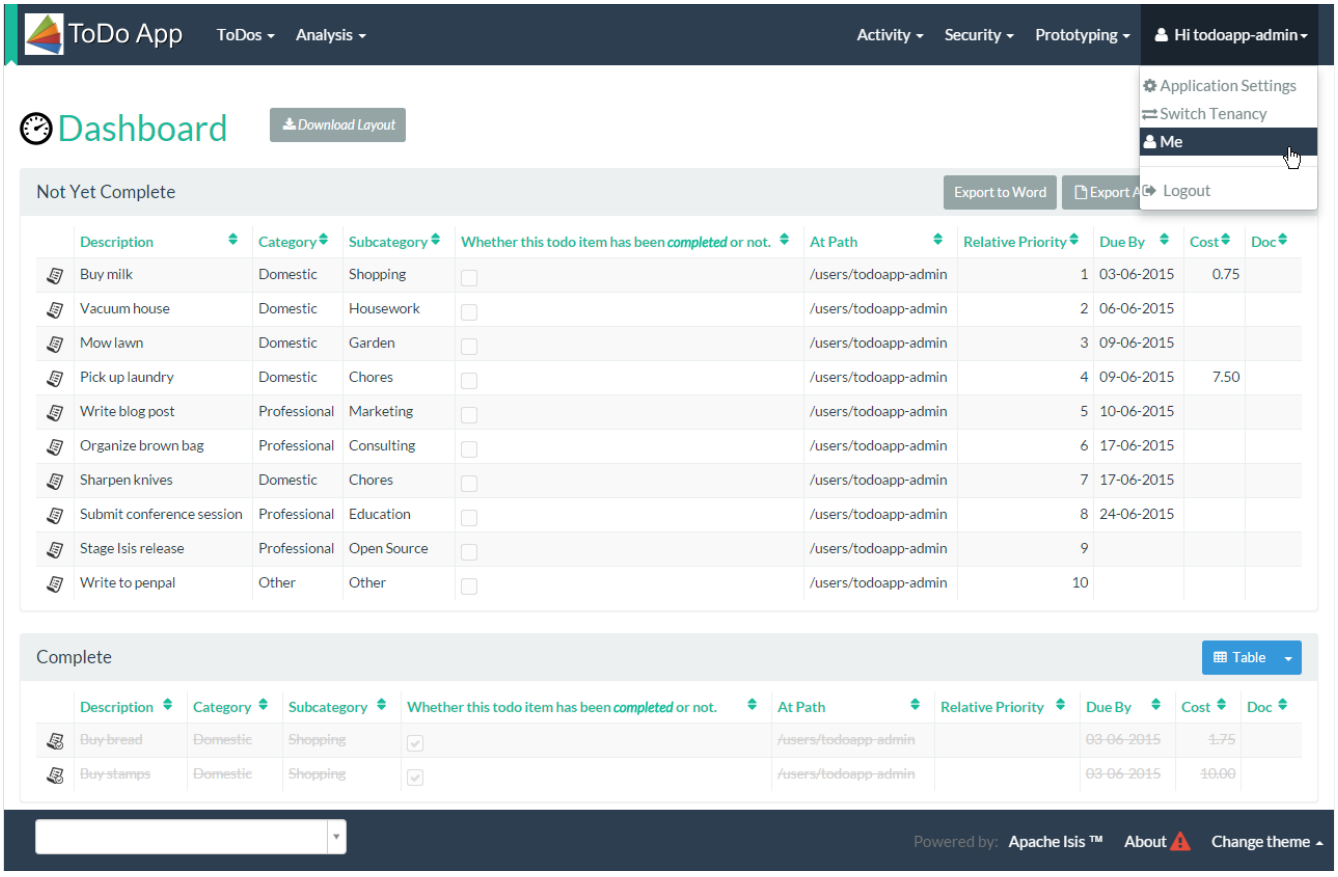
In the Wicket viewer, domain services placed on the `PRIMARY` menu bar appears to the left:



Domain services placed on the `SECONDARY` menu bar appear to the right:



Domain services placed on the `TERTIARY` appear in the menu bar associated with the user's name (far top-right)



The grouping of multiple domain services actions within a single drop-down is managed by the `@DomainServiceLayout#menuOrder()` attribute.

The RestfulObjects viewer does not support this attribute.

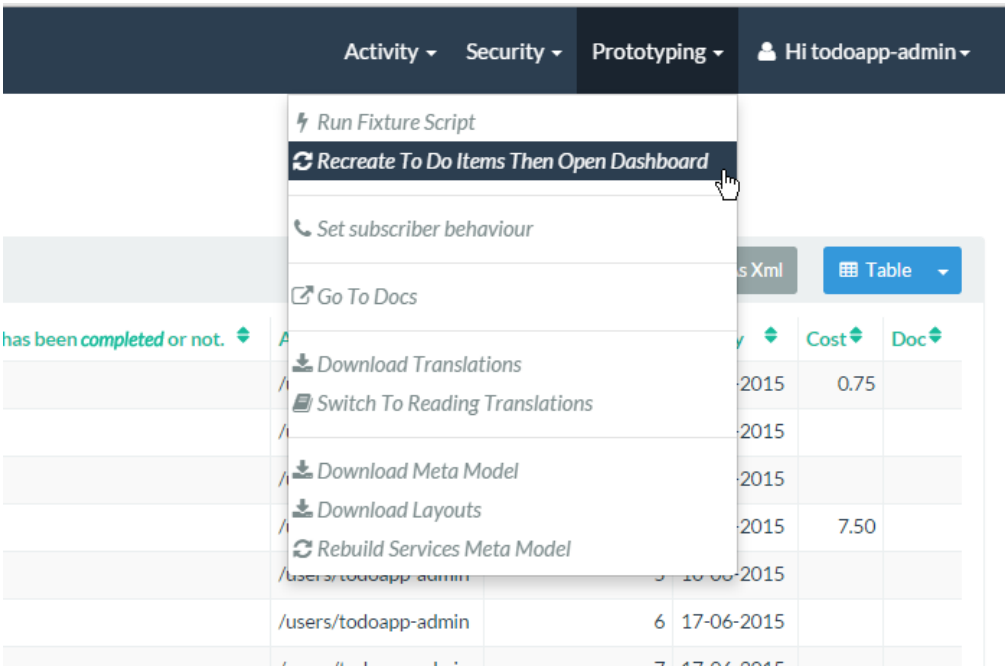
13.2. `menuOrder()`

The `menuOrder()` attribute determines the ordering of a domain service's actions as menu items within a specified menu bar and top-level menu.

The algorithm works as follows:

- first, the `menuBar()` determines which of the three menu bars the service's actions should be rendered
- then, the domain service's top-level name (typically explicitly specified using `named()`) is used to determine the top-level menu item to be rendered on the menu bar
- finally, if there is more than domain service that has the same name, then the `menuOrder` attribute is used to order those actions on the menu item drop-down.

For example, the screenshot below shows the "prototyping" menu from the (non-ASF) Isis addons' todoapp:



The Wicket viewer automatically places separators between actions from different domain services. From this we can infer that there are actually five different domain services that are all rendered on the "prototyping" top-level menu.

One of these is the todoapp's `DemoDomainEventSubscriptions` service:

```
@DomainService(
        nature = NatureOfService.VIEW_MENU_ONLY
)
@DomainServiceLayout(
        menuBar = MenuBar.SECONDARY,
        named = "Prototyping",                  ①
        menuOrder = "500.20")                   ②
public class DemoDomainEventSubscriptions {
    @ActionLayout(named="Set subscriber behaviour")
    @MemberOrder(sequence = "500.20.1")         ③
    public void subscriberBehaviour(...) { ... }
    ...
}
```

① render on the "Prototyping" menu

② positioning relative to other service's on the "Prototyping" menu

③ by convention (nothing more) the `@MemberOrder#sequence()` attribute continues the same Dewey decimal sequence format (a simple string "1" could in fact have been used instead)

while others come from services provided by Apache Isis itself, eg:

```
@DomainServiceLayout(
        named = "Prototyping",              ①
        menuBar = MenuBar.SECONDARY,
        menuOrder = "500.500"               ②
)
public class MetaModelServicesMenu {
    @MemberOrder(sequence="500.500.1")      ③
    public Clob downloadMetaModel( ... ) { ... }
    ...
}
```

① render on the "Prototyping" menu

② positioning relative to other service's on the "Prototyping" menu; this appears after the `DemoDomainEventSubscriptions` service shown above

③ by convention (nothing more) the `@MemberOrder#sequence()` attribute continues the same Dewey decimal sequence format (a simple string "1", "2", "3", ... could in fact have been used instead)

13.3. `named()`

The `named()` attribute explicitly specifies the domain service's name, overriding the name that would normally be inferred from the Java source code. This attribute can also be specified for actions, collections, properties, parameters, domain objects and view models.

The value of this attribute also has an important role to play in the positioning of the domain service's actions relative to the actions of other domain services. See `menuOrder()` for a full discussion with examples.

For example:

```
@DomainService
@DomainServiceLayout(
    named="Customers"
)
public class CustomerRepository {
   ...
}
```

Chapter 14. `@Facets`

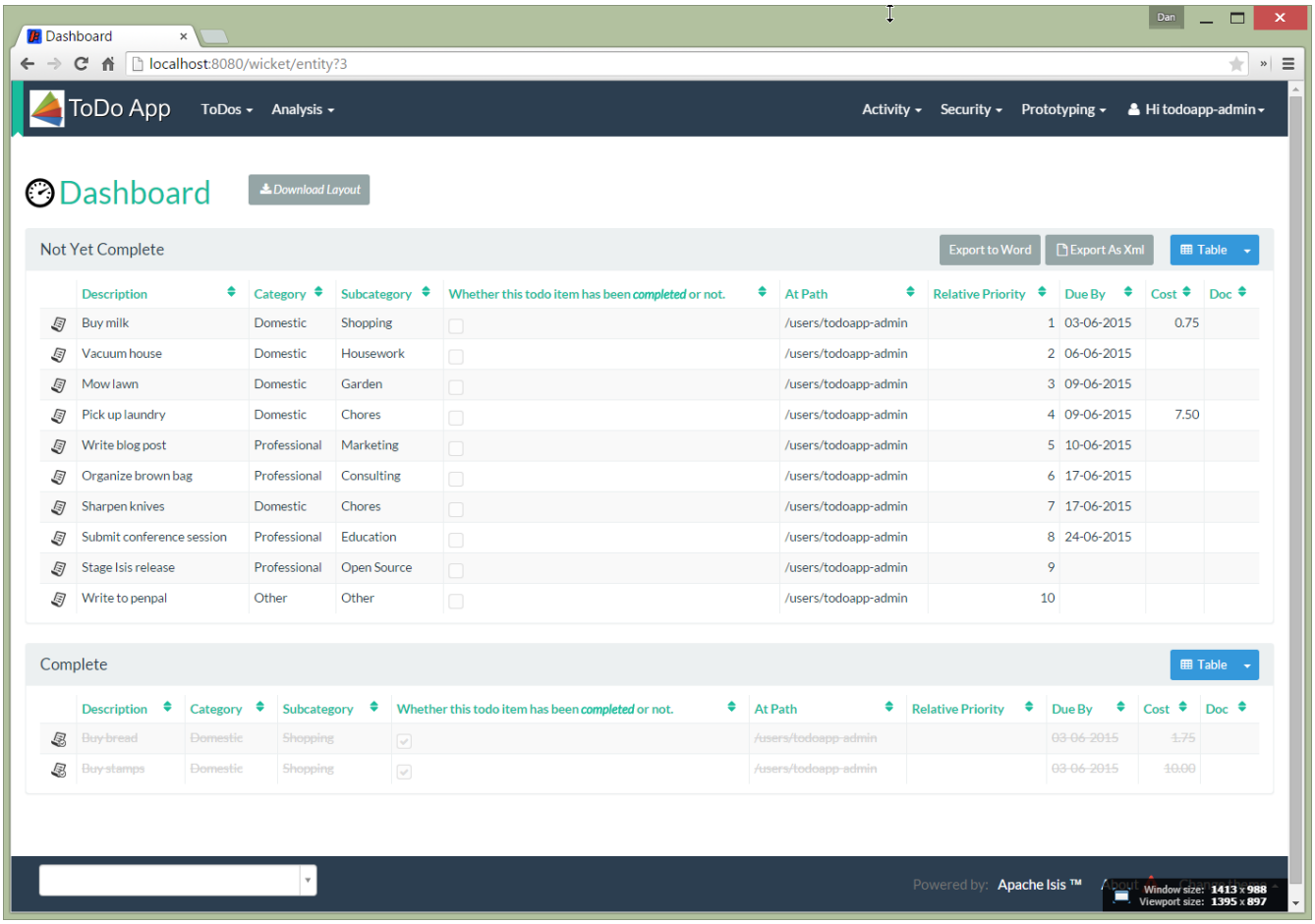
The `@Facets` annotation allows `FacetFactory` implementations and so can be used to run install arbitrary `Facet`s for a type. Generally this is not needed, but can be useful for overriding a custom programming model where a `FacetFactory` is not typically included.

`FacetFactory` is an important internal API that is used by Apache Isis to

Chapter 15. @HomePage

The `@HomePage` annotation allows a *single* (no-arg, query-only) action on a *single* domain service to be nominated as the action to invoke for the default home page. This often returns a view model that acts as some sort of dashboard, presenting key information and makeing the most commonly used actions easy to invoke.

For example, the (non-ASF) Isis addons' todoapp uses `@HomePage` to return a dashboard of todo items to complete:



The corresponding code is:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class ToDoAppDashboardService  {
    @Action(
            semantics = SemanticsOf.SAFE
    )
    @HomePage
    public ToDoAppDashboard lookup() {
        return container.injectServicesInto(new ToDoAppDashboard());
    }
    @Inject
    private DomainObjectContainer container;
}
```

where `ToDoAppDashboard` is:

```
@DomainObject(nature = Nature.VIEW_MODEL)
public class ToDoAppDashboard {
    public String title() { return "Dashboard"; }

    public List<ToDoItem> getNotYetComplete() { ... }
    public List<ToDoItem> getComplete() { ... }

    public Blob exportToWordDoc() { ... }  ①
}
```

① associated using file-based layout with the `notYetComplete` collection.

The other two actions shown in the above screenshot — `exportAsXml` and `downloadLayout` — are actually contributed to the `ToDoAppDashboard` through various domain services, as is the `downloadLayout` action.

Chapter 16. `@Inject` (`javax`)

Apache Isis automatically injects domain services into other domain services and also into domain objects and view models. In fact, it also injects domain services into integration tests and fixture scripts.

One omission: Apache Isis (currently) does not inject services into `o.a.i.applib.spec.Specification` instances (as used by `@Property#mustSatisfy()` and `@Parameter#mustSatisfy()` annotations.

Isis supports several syntaxes for injecting domain services. The simplest uses the `@javax.inject.Inject` annotation on the field, as defined in JSR-330.

For example:

```
public class Customer {
    public List<Order> findRecentOrders() {    ①
        return orders.recentOrdersFor(this);
    }
    @javax.inject.Inject
    OrderRepository orders;                    ②
}
```

① an alternative implementation would be to implement `findRecentOrders()` as a contributed action.

② we recommend default (rather than `private`) visibility so that unit tests can easily mock out the service

16.1. Alternative syntaxes

Isis also supports setter-based injection:

```
public class Customer {
    ...
    public void setOrderRepository(OrderRepository orderRepository) { ... }
}
```

and also supports an additional syntax of using `inject···` as the prefix:

```
public class Customer {
    ...
    public void injectOrderRepository(OrderRepository orderRepository) { ... }
}
```

Generally we recommend using `@javax.inject.Inject`; it involves less code, and is more

immediately familiar to most Java developers.

16.2. Injecting collection of services

It can sometimes be useful to have declared multiple implementations of a particular domain service. For example, you may have a module that defines an SPI service, where multiple other modules might provide implementations of that SPI (akin to the chain of responsibility pattern). To support these scenarios, it is possible to annotate a `List` or `Collection`.

For example, suppose that we provide an SPI service to veto the placing of `Order`s for certain `Customer`s:

```
public interface CustomerOrderAdvisorService {
    @Programmatic
    String vetoPlaceOrder(Customer c);
```

We could then inject a collection of these services:

```
public class Customer {
    public Order placeOrder(Product p, int quantity) { ... }
    public String disablePlaceOrder(Product p, int quantity) {
        for(CustomerOrderAdvisorService advisor: advisors) {
            String reason = advisor.vetoPlaceOrder(this);
            if(reason != null) { return reason; }
        }
        return null;
    }
    @Inject
    Collection<CustomerOrderAdvisorService> advisors;       ①
}
```

① inject a collection of the services.

An alternative and almost equivalent design would be to publish an event using the `EventBusService` and implement the domain services as subscribers to the event. This alternative design is used in the (non-ASF) Incode Platform's poly module, for example.

16.3. Manually injecting services

Isis performs dependency injection when domain entities are recreated. It will also perform dependency injection if an object is created through the `DomainObjectContainer`.

For example, to create a new (transient) domain object, the idiom is:

```
Customer cust = container.newTransientInstance(Customer.class);  ①
// initialize state of "cust"
container.persist(cust);
```

① where `container` is an instance of `DomainObjectContainer`.

View models are created similarly:

```
ToDoAppDashboard dashboard = container.newViewModelInstance(ToDoAppDashboard.class);
```

If you prefer, though, you can simply instantiate domain objects using "new" and then inject domain services manually:

```
Customer cust = new Customer();
container.injectServicesInto(cust);
// initialize state of "cust"
container.persist(cust);
```

or if you prefer:

```
Customer cust = container.injectServicesInto(new Customer());
// initialize state of "cust"
container.persist(cust);
```

There is one subtle difference between using `DomainObjectContainer#newTransientInstance(…)` and `DomainObjectContainer#injectServicesInto(…)`, in that with the former Apache Isis will automatically initialize all fields to their default values.

This isn't a particular useful feature (and indeed can sometimes be rather confusing) so you may well wish to standardize on using `injectServicesInto(…)` throughout.

Chapter 17. `@MemberGroupLayout`

The `@MemberGroupLayout` annotation specifies how an object's properties and collections are grouped together into columns, also specifying the relative positioning of those columns. It works in conjunction with the `@MemberOrder` annotation.

The `@MemberOrder` annotation is used to specify the relative order of domain object members, that is: properties, collections and actions. It works in conjunction with the `@MemberGroupLayout` annotation.

The annotation defines two attributes, `name()` and `sequence()`. Broadly speaking the `name()` attribute is used to group or associate members together, while the `sequence()` attribute orders members once they have been grouped.

As this is an important topic, there is a separate chapter that discussed object layout in full.

Chapter 18. `@MemberOrder`

The `@MemberOrder` annotation is used to specify the relative order of domain object members, that is: properties, collections and actions. It works in conjunction with the `@MemberGroupLayout` annotation.

The annotation defines four attributes:

- `columnSpans()` — of type `int[]` — which specifies the relative column sizes of the three columns that render properties as well as a fourth column that renders only collections
- `left()` — of type `String[]` - that specifies the order of the property groups (inferred from `@MemberOrder#name()`) as applied to properties) in the left-most column
- `middle()` — of type `String[]` - that specifies the order of the property groups (if any) as applied to properties) in the middle column
- `right()` — of type `String[]` - that specifies the order of the property groups (if any) as applied to properties) in the right-most column

Collections are always rendered in the "last" column. This can appear either below the columns holding properties (if their column spans = 12), or can be rendered to the right of the property columns (if the spans of the property columns come to <12 leaving enough room for the span of the collection column).

As this is an important topic, there is a separate chapter that discussed object layout in full.

The annotation is one of a handful (others including `@Collection`, `@CollectionLayout`, `@Property`) and `@PropertyLayout` that can also be applied to the field, rather than the getter method. This is specifically so that boilerplate-busting tools such as Project Lombok can be used.

Chapter 19. `@Mixin`

The `@Mixin` annotation indicates that the class acts as a mixin, contributing behaviour - actions, (derived) properties and (derived) collections - to another domain object.

Mixins were originally introduced as a means of allowing contributions from one module to the types of another module; in such cases the mixin type is often an interface type (eg `DocumentHolder`) that might be implemented by numerous different concrete types. However, mixins are also a convenient mechanism for grouping functionality even for a concrete type.

For further discussion on using mixins, see mixins in the user guide.

The table below summarizes the annotation's attributes.

Table 16. `@Mixin` attributes

Attribute	Values (default)	Description
`method()`	Method name within the mixin	How to recognize the "reserved" method name, meaning that the mixin's own name will be inferred from the mixin type. Typical examples are "exec", "execute", "invoke", "apply" and so on. The default "reserved" method name is `$$`.

An alternative and equivalent approach is to use the `@DomainObject#nature()` annotation with a nature of `MIXIN`.

19.1. `method()`

The `method()` attribute specifies the name of the method to be treated as a "reserved" method name, meaning that the mixin's name should instead be inferred from the mixin's type.

For example:

```
@DomainObject
public class Customer {

    @Mixin(method="execute")
    public static class placeOrder {

        Customer customer;
        public placeOrder(Customer customer) { this.customer = customer; }

        public Customer execute(Product p, int quantity) { ... }
        public String disableExecute() { ... }
        public String validate0Execute() { ... }
    }
    ...
)
```

This allows all mixins to follow a similar convention, with the name of the mixin inferred entirely from its type ("placeOrder").

When invoked programmatically, the code reads:

```
mixin(Customer.placeOrder.class, someCustomer).execute(someProduct, 3);
```

Chapter 20. `@Nullable` (`javax`)

Apache Isis' defaults for properties and parameters is that they are mandatory unless otherwise stated. The `@javax.annotation.Nullable` annotation is recognized by Apache Isis for both properties and parameters as means to indicate that the property/parameter is not mandatory.

For example:

```
@javax.annotation.Nullable
public String getName() {
    return name;
}
public void setName(final String name) {
    this.name = name;
}
```

or:

```
public Customer updateName(@javax.annotation.Nullable final String name) {
    setName(name);
    return this;
}
```

Apache Isis does provide several other ways to specify optionality: using the `@Property#optionality()` / `@Parameter#optionality()` annotation. For properties, the optionality can also be inferred from the `@Column#allowsNull()` attribute.

See the `@Property#optionality()` documentation for a much fuller discussion on the relationship between using the Apache Isis annotations vs `@Column#allowsNull()`.

If more than one method is specified then the framework will validate that there are no incompatibilities (and fail to boot otherwise). This can also be verified using the validate goal of the Apache Isis Maven plugin.

Chapter 21. `@NotPersistent` (`javax.jdo`)

The `@javax.jdo.annotation.NotPersistent` annotation is used by JDO/DataNucleus to indicate that a property should not be persisted to the database.

Apache Isis also uses this annotation, though (currently) only in the very minimal way to suppress checking of inconsistent metadata between JDO and Isis annotations (eg `@Column#allowsNull()` vs `@Property#optionality()`, or `@Column#length()` and `@Property#maxLength()`).

Isis parses the `@NotPersistent` annotation from the Java source code; it does not query the JDO metamodel. This means that it the `@NotPersistent` annotation must be used rather than the equivalent `<field>` XML metadata.

Moreover, while JDO/DataNucleus will recognize annotations on either the field or the getter method, Apache Isis (currently) only inspects the getter method. Therefore ensure that the annotation is placed there.

Chapter 22. `@MinLength`

The `@MinLength` annotation is used to specify the minimum number of characters in a search of an `autoComplete…()` supporting method.

For example:

```
public ToDoItem add(
        @TypicalLength(20)
        final ToDoItem toDoItem) {
    getDependencies().add(toDoItem);
    return this;
}
public List<ToDoItem> autoComplete0Add(
        final @MinLength(2)
        String search) {
    final List<ToDoItem> list = toDoItems.autoComplete(search);
    list.removeAll(getDependencies());
    list.remove(this);
    return list;
}
```

The `@Parameter(minLength=…)`e can also be used (even though strictly speaking the search argument is not a parameter of the action).

Chapter 23. `@Parameter`

The `@Parameter` annotation applies to action parameters collecting together all domain semantics within a single annotation.

The table below summarizes the annotation's attributes.

Table 17. `@Paramter` attributes

Attribute	Values (default)	Description
`fileAccept()`	Media type or file extension	Hints the file type to be uploaded for `Blob` or `Clob`. Note that this does not prevent the user from uploading some other file type; rather it merely defaults the file type in the file open dialog.
`maxLength()`	Positive integer	maximum number of characters for string parameters; ignored otherwise
`minLength()`	Positive integer	Deprecated; use <code>@MinLength</code> instead. Can be used to specify the minimum length for <code>autoComplete…()</code> supporting method; but because this _is a supporting method rather than the action method itself, we now feel it is misleading to use the <code>@Parameter</code> annotation in this situation.
`mustSatisfy()`	implementation of `o.a.i.applib.spec.Specification`	allows arbitrary validation to be applied
`optionality()`	`MANDATORY`, `OPTIONAL` (`MANDATORY`)	specifies a parameter is optional rather than mandatory
regexPattern()	regular expression	validates the contents of a string parameter against the regular expression pattern
`regexPatternFlags()`	value of flags as normally passed to `java.util.regex.Pattern#compile(…)`	modifies the compilation of the regular expression
`regexPatternReplacement()`		Unused.

For example:

```
public class Customer {
    public static class EmailSpecification extends AbstractSpecification<String> {
        public String satisfiesSafely(String proposed) {
            return EmailUtil.ensureValidEmail(proposed);    ①
        }
    }
    @Action(semantics=SemanticsOf.IDEMPOTENT)
    public Customer updateEmail(
        @Parameter(
            maxLength=30,
            mustSatisfy=EmailSpecification.class,
            optionality=Optionality.OPTIONAL,
            regexPattern = "(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+",
            regexPatternFlags=Pattern.CASE_INSENSITIVE
        )
        @ParameterLayout(named="New Email Address")
        final String newEmailAddress
        ...
    }
}
```

① the (fictitious) `EmailUtil.ensureValid(…)` (omitted for brevity) returns a string explaining if an email is invalid

23.1. `fileAccept()`

The `fileAccept()` attribute applies only to `Blob` or `Clob` parameters, indicating the type of file to accept when uploading a new value. The attribute is also supported on properties.

For example:

```
public class Scanner {
    public ScannedDocument newScan(
        @Parameter(fileAccept="image/*")            ①
        @ParameterLayout(named="Scanned image")     ②
        final Blob scannedImage) {
        ...
    }
}
```

① as per reference docs, either a media type (such as `image/*`) or a file type extension (such as `.png`).

② the `@ParameterLayout(named=…)` attribute is required for Java 7; for Java 8 it can be omitted if the (non-ASF) Incode Platform's paraname8 metamodel extension is used.

23.2. maxLength()

The `maxLength()` attribute applies only to `String` parameters, indicating the maximum number of characters that the user may enter (for example in a text field in the UI). It is ignored if applied to parameters of any other type. This attribute can also be applied to properties.

For example:

```
public class CustomerRepository {
    public Customer newCustomer(
        @Parameter(maxLength=30)
        @ParameterLayout(named="First Name")  ①
        final String firstName,
        @Parameter(maxLength=50)
        @ParameterLayout(named="Last Name")
        final String lastName) {
        ...
    }
}
```

① the `@ParameterLayout(named=…)` attribute is required for Java 7; for Java 8 it can be omitted if the (non-ASF) Incode Platform's paraname8 metamodel extension is used.

23.3. mustSatisfy()

The `mustSatisfy()` attribute allows arbitrary validation to be applied to parameters using an (implementation of a) `org.apache.isis.applib.spec.Specification` object. The attribute is also supported on properties.

The specification implementations can (of course) be reused between parameters and properties.

The `Specification` is consulted during validation, being passed the proposed value. If the proposed value fails, then the value returned is the used as the invalidity reason.

For example:

```
public class StartWithCapitalLetterSpecification
        extends AbstractSpecification<String> {           ①
    public String satisfiesSafely(String proposed) {
        return "".equals(proposed)
            ? "Empty string"
            : !Character.isUpperCase(proposed.charAt(0))
                ? "Does not start with a capital letter"
                : null;
    }
}
public class CustomerRepository {
    public Customer newCustomer(
            @Parameter(
                mustSatisfy=StartWithCapitalLetterSpecification.class
            )
            @ParameterLayout(named="First Name")
            final String firstName,
            @Parameter(
                mustSatisfy=StartWithCapitalLetterSpecification.class
            )
            @ParameterLayout(named="Last Name")
            final String lastName) {
        ...
    }
    ...
}
```

① the `AbstractSpecification` class conveniently handles type-safety and dealing with null values. The applib also provides `SpecificationAnd` and `SpecificationOr` to allow specifications to be combined "algebraically".

It is also possible to provide translatable reasons. Rather than implement `Specification`, instead implement `Specification2` which defines the API:

```
public interface Specification2 extends Specification {
    public TranslatableString satisfiesTranslatable(Object obj); ①
}
```

① Return `null` if specification satisfied, otherwise the reason as a translatable string

With `Specification2` there is no need to implement the inherited `satifies(Object)`; that method will never be called.

23.4. `optionality()`

By default, Apache Isis assumes that all parameters of an action are required (mandatory). The `optionality()` attribute allows this to be relaxed. The attribute is also supported for properties.

The attribute has no meaning for a primitive type such as `int`: primitives will always have a default value (e.g. zero). If optionality is required, then use the corresponding wrapper class (e.g. `java.lang.Integer`) and annotate with `Parameter#optionality()` as required.

The values for the attribute are simply `OPTIONAL` or `MANDATORY`.

For example:

```
public class Customer {
    public Order placeOrder(
            final Product product,
            @ParameterLayout(named = "Quantity")
            final int quantity,
            @Parameter(optionality = Optionality.OPTIONAL)
            @ParameterLayout(named = "Special Instructions")
            final String instr) {
        ...
    }
    ...
}
```

It is also possible to specify optionality using `@Nullable` annotation.

23.5. `regexPattern()`

There are three attributes related to enforcing regular expressions:

- The `regexPattern()` attribute validates the contents of any string parameter with respect to a regular expression pattern. It is ignored if applied to parameters of any other type. This attribute can also be specified for properties.
- The `regexPatternFlags()` attribute specifies flags that modify the handling of the pattern. The values are those that would normally be passed to `java.util.regex.Pattern#compile(String,int)`.
- The related `regexPatternReplacement()` attribute specifies the error message to show if the provided argument does not match the regex pattern.

For example:

```
public class Customer {
    public void updateEmail(
            @Parameter(
                regexPattern = "(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+",
                regexPatternFlags = Pattern.CASE_INSENSITIVE,
                regexPatternReplacement = "Must be valid email address (containing a
'@') symbol"   ①
            )
            @ParameterLayout(named = "Email")
            final String email) {
        ...
    }
)
```

① Note that there is currently no i18n support for this phrase.

Chapter 24. `@ParameterLayout`

The `@ParameterLayout` annotation applies to action parameters, collecting together all UI hints within a single annotation.

The table below summarizes the annotation's attributes.

Table 18. `@ParameterLayout` attributes

Attribute	Values (default)	Description
`cssClass()`	Any string valid as a CSS class	the css class that a parameter should have, to allow more targetted styling in `application.css`
`describedAs()`	String	description of this parameter, eg to be rendered in a tooltip.
`labelPosition()`	`LEFT`, `TOP`, `RIGHT`, `NONE` (`LEFT`)	in forms, the positioning of the label relative to the property value. Default is `LEFT`, unless `multiLine` in which case `TOP`. The value `RIGHT` is only supported for boolean parameters.
`multiLine()`	Positive integer	for string parameters, render as a text area over multiple lines. If set > 1, then then `labelPosition` defaults to `TOP`.
`named()`	String	the name of this parameter. For Java 7 this is generally required. For Java 8, the name can often be inferred from the code so this attribute allows the name to be overridden. A typical use case is if the desired name is a reserved Java keyword, such as `default` or `package`.
`namedEscaped()`	`true`, `false` (`true`)	whether to HTML escape the name of this parameter.
`renderedAsDayBefore()`		for date parameters only, render the date as one day prior to the actually stored date (eg the end date of an open interval into a closed interval)
`typicalLength()`		the typical entry length of a field, use to determine the optimum width for display

For example:

```
public class ToDoItem {
    public ToDoItem updateDescription(
                        @ParameterLayout(
                            cssClass="x-key",
                            describedAs="What needs to be done",
                            labelPosition=LabelPosition.LEFT,
                            named="Description of this <i>item</i>",
                            namedEscaped=false,
                            typicalLength=80)
                        final String description) {
        setDescription(description);
        return this;
    }
    ...
}
```

Note that there is (currently) no support for specifying UI hints for domain services through the dynamic `.layout.json` file (only for properties, collections and actions are supported).

24.1. cssClass()

The `cssClass()` attribute can be used to render additional CSS classes in the HTML (a wrapping `<div>`) that represents the action parameter. Application-specific CSS can then be used to target and adjust the UI representation of that particular element.

This attribute can also be applied to domain objects, view models, actions properties, collections and parameters.

For example:

```
public class ToDoItem {
    public ToDoItem postpone(
            @ParameterLayout(
                named="until",
                cssClass="x-key"
            )
            LocalDate until
        ) { ... }
    ...
}
```

24.2. describedAs()

The `describedAs()` attribute is used to provide a short description of the action parameter to the user. In the Wicket viewer it is displayed as a 'tool tip'. The `describedAs()` attribute can also be

specified for collections, properties, actions, domain objects and view models.

Descriptions may be provided for objects, members (properties, collections and actions), and for individual parameters within an action method. `@DescribedAs` therefore works in a very similar manner to `@Named` <!--(see ?)-→.

To provide a description for an individual action parameter, use the `@DescribedAs` annotation in-line i.e. immediately before the parameter declaration.

For example:

```
public class Customer {
    public Order placeOrder(
                        Product product,
                        @ParameterLayout(
                            named="Quantity",
                            describedAs="The quantity of the product being ordered"
                        )
                        int quantity) {
        ...
    }
    ...
}
```

24.3. `labelPosition()`

The `labelPosition()` attribute determines the positioning of labels for parameters. This attribute can also be specified for properties.

The positioning of labels is typically `LEFT`, but can be positioned to the `TOP`. The one exception is `multiLine()` string parameters, where the label defaults to `TOP` automatically (to provide as much real-estate for the multiline text field as possible).

For boolean parameters a positioning of `RIGHT` is also allowed; this is ignored for all other types.

It is also possible to suppress the label altogether, using `NONE`.

For example:

```
public class Order {
    public Order changeStatus(
                    OrderStatus newStatus
                    @Parameter(
                        optionality=Optionality.OPTIONAL
                    )
                    @ParameterLayout(
                        named="Reason",
                        labelPosition=LabelPosition.TOP
                    )
                    String reason) {
        ...
    }
    ...
}
```

To get an idea of how these are rendered (in the Wicket viewer), see PropertyLayout#labelPosition().

24.4. multiLine()

The `multiLine()` attribute specifies that the text field for a string parameter should span multiple lines. It is ignored for other parameter types. The attribute is also supported for properties.

For example:

```
public class BugReport {
    public BugReport updateStepsToReproduce(
                        @Parameter(named="Steps")
                        @ParameterLayout(
                            numberOfLines=10
                        )
                        final String steps) {
        ...
    }
    ...
}
```

If set > 1 (as would normally be the case), then the default `labelPosition` defaults to `TOP` (rather than `LEFT`, as would normally be the case).

24.5. named()

The `named()` attribute explicitly specifies the action parameter's name. This attribute can also be specified for actions, collections, properties, domain objects, view models and domain services.

Unlike most other aspects of the Apache Isis metamodel, the name of method parameters cannot

(prior to Java 8, at least) be inferred from the Java source code. Without other information, Apache Isis uses the object's type (`int`, `String` etc) as the name instead. This can be sufficient for application-specific reference types (eg `ToDoItem`) but is generally not sufficient for primitives and other value types.

The `named()` attribute (or the deprecated `@Named` annotation) is therefore often required to specify the parameter name.

As of Java 8, the Java reflection API has been extended. The (non-ASF) Incode Platform's paraname8 metamodel extension provides support for this. Note that your application must (obviously) be running on Java 8, and be compiled with the `-parameters` compile flag for javac.

By default the name is HTML escaped. To allow HTML markup, set the related `namedEscaped()` attribute to `false`.

For example:

```
public class Customer {
    public Order placeOrder(
            final Product product
           ,@ParameterLayout(named="Quantity")
            final int quantity) {
        Order order = newTransientInstance(Order.class);
        order.modifyCustomer(this);
        order.modifyProduct(product);
        order.setQuantity(quantity);
        return order;
    }
    ...
}
```

The framework also provides a separate, powerful mechanism for internationalization.

24.6. `renderedAsDayBefore()`

The `renderedAsDayBefore()` attribute applies only to date parameters whereby the date will be rendered as the day before the value actually held in the domain object. It is ignored for parameters of other types. This attribute is also supported for properties.

This behaviour might at first glance appear odd, but the rationale is to support the use case of a sequence of instances that represent adjacent intervals of time. In such cases there would typically be `startDate` and `endDate` properties, eg for all of Q2. Storing this as a half-closed interval — eg `[1-Apr-2015, 1-July-2015)` — can substantially simplify internal algorithms; the `endDate` of one interval will correspond to the `startDate` of the next.

However, from an end-user perspective the requirement may be to render the interval as a fully closed interval; eg the end date should be shown as `30-Jun-2015`.

This attribute therefore bridges the gap; it presents the information in a way that makes sense to an end-user, but also stores the domain object in a way that is easy work with internally.

For example:

```
public class Tenancy {
    public void changeDates(
        @ParameterLayout(named="Start Date")
        LocalDate startDate,
        @ParameterLayout(
            named="End Date",
            renderedAsDayBefore=true
        )
        LocalDate endDate) {
        ...
    }
}
```

24.7. typicalLength()

The `typicalLength()` attribute indicates the typical length of a string parameter. It is ignored for parameters of other types. The attribute is also supported for properties.

The information is intended as a hint to the UI to determine the space that should be given to render a particular string parameter. That said, note that the Wicket viewer uses the maximum space available for all fields, so in effect ignores this attribute.

For example:

```
public class Customer {
    public Customer updateName(
                @Parameter(maxLength=30)
                @ParameterLayout(
                    named="First name",
                    typicalLength=20
                )
                final String firstName,
                @Parameter(maxLength=30)
                @ParameterLayout(
                    named="Last name",
                    typicalLength=20
                )
                final String lastName) {
        ...
    }
    ...
}
```

Chapter 25. `@PersistenceCapable` (`javax.jdo`)

The `@javax.jdo.annotation.PersistenceCapable` is used by JDO/DataNucleus to indicate that a class is a domain entity to be persisted to the database.

Apache Isis also checks for this annotation, and if the `@PersistenceCapable#schema()` attribute is present will use it to form the object type.

Isis parses the `@PersistenceCapable` annotation from the Java source code; it does not query the JDO metamodel. This means that it the `@PersistenceCapable` annotation must be used rather than the equivalent `<class>` XML metadata.

Moreover, while JDO/DataNucleus will recognize annotations on either the field or the getter method, Apache Isis (currently) only inspects the getter method. Therefore ensure that the annotation is placed there.

This value is used internally to generate a string representation of an objects identity (the `Oid`). This can appear in several contexts, including:

- as the value of `Bookmark#getObjectType()` and in the `toString()` value of `Bookmark` (see `BookmarkService`)
  - and thus in the "table-of-two-halves" pattern, as per the (non-ASF) Incode Platform's poly module
- in the serialization of `OidDto` in the command and interaction schemas
- in the URLs of the RestfulObjects viewer
- in the URLs of the Wicket viewer (in general and in particular if copying URLs)
- in XML snapshots generated by the `XmlSnapshotService`

The actual format of the object type used by Apache Isis for the concatenation of `schema()` and `@PersistenceCapable#table()`. If the `table()` is not present, then the class' simple name is used instead.

25.1. Examples

For example:

```
@javax.jdo.annotations.PersistenceCapable(schema="custmgmt")
public class Customer {
    ...
}
```

has an object type of `custmgmt.Customer`, while:

```
@javax.jdo.annotations.PersistenceCapable(schema="custmgmt", table="Address")
public class AddressImpl {
    ...
}
```

has an object type of `custmgmt.Address`.

On the other hand:

```
@javax.jdo.annotations.PersistenceCapable(table="Address")
public class AddressImpl {
    ...
}
```

does *not* correspond to an object type, because the `schema()` attribute is missing.

25.2. Precedence

The rules of precedence for determining a domain object's object type are:

1. `@Discriminator`
2. `@DomainObject#objectType`
3. `@PersistenceCapable`, if at least the `schema` attribute is defined.

   If both `schema` and `table` are defined, then the value is "schema.table". If only `schema` is defined, then the value is "schema.className".

4. Fully qualified class name of the entity.

This might be obvious, but to make explicit: we recommend that you always specify an object type for your domain objects.

Otherwise, if you refactor your code (change class name or move package), then any externally held references to the OID of the object will break. At best this will require a data migration in the database; at worst it could cause external clients accessing data through the Restful Objects viewer to break.

If the object type is not unique across all domain classes then the framework will fail-fast and fail to boot. An error message will be printed in the log to help you determine which classes have duplicate object tyoes.

Chapter 26. `@PostConstruct` (`javax`)

The `@javax.annotation.PostConstruct` annotation, as defined in JSR-250, is recognized by Apache Isis as a callback method on domain services to be called just after they have been constructed, in order that they initialize themselves.

It is also recognized for view models (eg annotated with `@ViewModel`).

For the default application-scoped (singleton) domain services, this means that the method, if present, is called during the bootstrapping of the application. For `@RequestScoped` domain services, the method is called at the beginning of the request.

The signature of the method is:

```
@PostConstruct                  ①
public void init() { ... }      ②
```

① It is not necessary to annotate the method with `@Programmatic`; it will be automatically excluded from the Apache Isis metamodel.

② the method can have any name, but must have `public` visibility.

In the form shown above the method accepts no arguments. Alternatively - for domain services only, not view models - the method can accept a parameter of type `Map<String,String>`:

```
@PostConstruct
@Programmatic
public void init(Map<String,String> properties) { ... }
```

Isis uses argument to pass in the configuration properties read from all configuration files:

Alternatively, you could inject `DomainObjectContainer` into the service and read configuration properties using `DomainObjectContainer#getProperty(…)` and related methods. Note that when using this latter API only those configuration properties prefixes `application.` key are provided.

A common use case is for domain services that interact with the `EventBusService`. For example:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class MySubscribingService {
    @PostConstruct
    public void postConstruct() {
        eventBusService.register(this);
    }
    @PreDestroy
    public void preDestroy() {
        eventBusService.unregister(this);
    }
    ...
    @javax.inject.Inject
    EventBusService eventBusService;
}
```

In this particular use case, it is generally simpler to just subclass from `AbstractSubscriber`.

Other use cases include obtaining connections to external datasources, eg subscribing to an ActiveMQ router, say, or initializing/cleaning up a background scheduler such as Quartz.

See also `@PreDestroy`

Chapter 27. `@PreDestroy` (`javax`)

The `@javax.annotation.PreDestroy` annotation, as defined in JSR-250, recognized by Apache Isis as a callback method on domain services to be called just as they go out of scope.

For the default application-scoped (singleton) domain services, this means that the method, if present, is called just prior to the termination of the application. For `@RequestScoped` domain services, the method is called at the end of the request.

The signature of the method is:

```
@PreDestroy                         ①
public void deinit() { ... }    ②
```

① It is not necessary to annotate the method with `@Programmatic`; it will be automatically excluded from the Apache Isis metamodel.

② the method can have any name, but must have `public` visibility, and accept no arguments.

A common use case is for domain services that interact with the `EventBusService`. For example:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class MySubscribingService {
    @PostConstruct
    public void init() {
        eventBusService.register(this);
    }
    @PreDestroy
    public void deinit() {
        eventBusService.unregister(this);
    }
    ...
    @javax.inject.Inject
    EventBusService eventBusService;
}
```

In this particular use case, it is generally simpler to just subclass from `AbstractSubscriber`.

Other use cases include obtaining connections to external datasources, eg subscribing to an ActiveMQ router, say, or initializing/cleaning up a background scheduler such as Quartz.

See also `@PostConstruct`

Chapter 28. `@PrimaryKey` (`javax.jdo`)

The `@javax.jdo.annotation.PrimaryKey` annotation is used by JDO/DataNucleus to indicate that a property is used as the primary key for an entity with application-managed identity.

Apache Isis also uses this annotation in a very minimal way: to ensure that the framework's own logic to initialize newly instantiated objects (eg using `DomainObjectContainer#newTransientInstance(…)` does not touch the primary key, and also to ensure that the primary key property is always disabled (read-only).

Isis parses the `@NotPersistent` annotation from the Java source code; it does not query the JDO metamodel. This means that it the `@NotPersistent` annotation must be used rather than the equivalent `<field>` XML metadata.

Moreover, while JDO/DataNucleus will recognize annotations on either the field or the getter method, Apache Isis (currently) only inspects the getter method. Therefore ensure that the annotation is placed there.

Chapter 29. @Programmatic

The `@Programmatic` annotation causes the method to be excluded completely from the Apache Isis metamodel. This means it won't appear in any UI, and it won't appear in any mementos or snapshots.

A common use-case is to ignore implementation-level artifacts. For example:

```
public class Customer implements Comparable<Customer> {
    ...
    @Programmatic
    public int compareTo(Customer c) {
        return getSalary() - c.getSalary();
    }
    ...
}
```

Note that `@Programmatic` does not simply imply `@Hidden`; it actually means that the class member will not be part of the Apache Isis metamodel.

Chapter 30. `@Property`

The `@Property` annotation applies to properties collecting together all domain semantics within a single annotation.

It is also possible to apply the annotation to actions of domain services that are acting as contributed properties.

Table 19. `@Property` attributes

Attribute	Values (default)	Description
`command()`	`AS_CONFIGURED`, `ENABLED`, `DISABLED` (`AS_CONFIGURED`)	whether the property edit should be reified into a `o.a.i.applib.services.command.Command` object through the `CommandContext` service.
`commandExecuteIn()`	`FOREGROUND`,`BACKGROUND` (`FOREGROUND`)	whether to execute the command immediately, or to persist it (assuming that an appropriate implementation of `CommandService` has been configured) such that a background scheduler can execute the command asynchronously
`commandPersistence()`	`PERSISTED`, `NOT_PERSISTED`, `IF_HINTED` (`PERSISTED`)	whether the reified `Command` (as provided by the `CommandContext` domain service) should actually be persisted (assuming an appropriate implementation of `CommandService` has been configured).
`commandDtoProcessor()`	Implementation of `CommandDtoProcessor` interface (null)	If the `Command` also implements `CommandWithDto` (meaning that it can return a `CommandDto`, in other words be converted into an XML memento), then optionally specifies a processor that can refine this XML.
`domainEvent()`	subtype of `PropertyDomainEvent` (`PropertyDomainEvent.Default`)	the event type to be posted to the `EventBusService` to broadcast the property's business rule checking (hide, disable, validate) and its modification (before and after).
`editing()`	`ENABLED`, `DISABLED`, `AS_CONFIGURED` (`AS_CONFIGURED`)	whether a property can be modified or cleared from within the UI
`fileAccept()`	Media type or file extension	Hints the files to be uploaded to a `Blob` or `Clob`. Note that this does not prevent the user from uploading some other file type; rather it merely defaults the file type in the file open dialog.

Attribute	Values (default)	Description
`hidden()`	`EVERYWHERE`, `OBJECT_FORMS`, `PARENTED_TABLES`, `STANDALONE_TABLES`, `ALL_TABLES`, `NOWHERE` (`NOWHERE`)	indicates where (in the UI) the property should be hidden from the user.
`maxLength()`		maximum number of characters for string parameters; ignored otherwise In many/most cases you should however use `@Column#length()`
`mustSatisfy()`	implementation of `o.a.i.applib.spec.Specification`	allows arbitrary validation to be applied
`notPersisted()`	`true`, `false` (`false`)	whether to exclude from snapshots. [WARNING] ==== Property must also be annotated with `@javax.jdo.annotations.NotPersistent` in order to not be persisted. ====
`optionality()`		specifies a property is optional rather than mandatory In many/most cases you should however use `@Column#allowsNull()`
`regexPattern()`	regular expression	validates the contents of a string parameter against the regular expression pattern
`regexPatternFlags()`	value of flags as normally passed to `java.util.regex.Pattern#compile(…)`	modifies the compilation of the regular expression

For example:

```
@DomainObject
public class Customer {
    public static class EmailSpecification extends AbstractSpecification<String> {
        public String satisfiesSafely(String proposed) {
            return EmailUtil.ensureValidEmail(proposed);    ①
        }
    }
    @javax.jdo.annotations.Column(allowsNull="true")                ②
    @Property(
        maxLength=30,
        minLength=5,
        mustSatisfy=EmailSpecification.class,
        regexPattern = "(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+",
        regexPatternFlags=Pattern.CASE_INSENSITIVE
    )
    public String getEmailAddress() { ... }
    public void setEmailAddress(String emailAddress) { ... }
    ...
}
```

① the (fictitious) `EmailUtil.ensureValid(…)` (omitted for brevity) returns a string explaining if an email is invalid

② generally use instead of the `@Property#optionality()` attribute

The annotation is one of a handful (others including `@Collection`, `@CollectionLayout` and `@PropertyLayout`) that can also be applied to the field, rather than the getter method. This is specifically so that boilerplate-busting tools such as Project Lombok can be used.

30.1. Command Persistence and Processing

Every property edit (and action invocation for that matter) is automatically reified into a concrete `Command` object. The `@Property(command=…, commandXxx=…)` attributes provide hints for the persistence of that `Command` object, and the subsequent processing of that persisted command. The primary use cases for this are to support the deferring the execution of the action such that it can be invoked in the background, and to replay commands in a master/slave configuration.

30.1.1. Design

The annotation works with (and is influenced by the behaviour of) a number of domain services:

- `CommandContext`
- `CommandService`
- `BackgroundService` and
- `BackgroundCommandService`

Each property edit is automatically reified by the `CommandContext` service into a `Command` object,

capturing details of the target object, the property, the proposed new value fo the property, the user, a timestamp and so on.

If an appropriate `CommandService` is configured (for example using (non-ASF) Incode Platform's command module), then the `Command` itself is persisted.

By default, actions are invoked in directly in the thread of the invocation. If there is an implementation of `BackgroundCommandService` (as the (non-ASF) Incode Platform's command module does provide), then this means in turn that the `BackgroundService` can be used by the domain object code to programmatically create background `Command`s.

If background `Command`s are used, then an external scheduler, using headless access, must also be configured.

30.1.2. `command()` and `commandPersistence()`

The `command()` and `commandPersistence() attributes work together to determine whether a command will actually be persisted. There inter-relationship is somewhat complex, so is probably best explained by way of examples:

`command()`	`isis.services.command.properties` **config property**	`commandPersistence()`	**action dirties objects?**	**is command persisted?**
`ENABLED`	(any)	`PERSISTED`	(either)	yes
`ENABLED`	(any)	`IF_HINTED`	no	no
`ENABLED`	(any)	`IF_HINTED`	yes	yes
`ENABLED`	(any)	`NOT_PERSISTED`	(any)	no
`AS_CONFIGURED`	`all`	`PERSISTED`	no	yes
`AS_CONFIGURED`	`all`	`IF_HINTED`	no	no
`AS_CONFIGURED`	`all`	`IF_HINTED`	yes	yes
`AS_CONFIGURED`	`all`	`NOT_PERSISTED`	(any)	no
`AS_CONFIGURED`	`none`	`PERSISTED`	no	no (!)
`AS_CONFIGURED`	`none`	`PERSISTED`	yes	yes
`AS_CONFIGURED`	`none`	`IF_HINTED`	no	no
`AS_CONFIGURED`	`none`	`IF_HINTED`	yes	yes
`AS_CONFIGURED`	`none`	`NOT_PERSISTED`	no	no
`AS_CONFIGURED`	`none`	`NOT_PERSISTED`	yes	yes (!)
`DISABLED`	(any)	`PERSISTED`	no	no (!)
`DISABLED`	(any)	`PERSISTED`	yes	yes

command()	isis.services.command.properties config property	commandPersistence()	action dirties objects?	is command persisted?
DISABLED	(any)	IF_HINTED	no	no
DISABLED	(any)	IF_HINTED	yes	yes
DISABLED	(any)	NOT_PERSISTED	no	no
DISABLED	(any)	NOT_PERSISTED	yes	yes (!)

For example:

```
public class Order {
    @Property(
        command=CommandReification.ENABLED,
        commandPersistence=CommandPersistence.PERSISTED
    )
    public Product getProduct() { ... }
    public void setProduct(Product p) { ... }
}
```

As can be seen, whether a command is actually persisted does not always follow the value of the commandPersistence() attribute. This is because the command() attribute actually determines whether any command metadata for the action is captured within the framework's internal metamodel. If command is DISABLED or does not otherwise apply due to the action's declared semantics, then the framework decides to persist a command based solely on whether the action dirtied any objects (as if commandPersistence() was set to IF_HINTED).

30.1.3. commandExecuteIn()

For persisted commands, the commandExecuteIn() attribute determines whether the Command should be executed in the foreground (the default) or executed in the background.

Background execution means that the command is not executed immediately, but is available for a configured BackgroundCommandService to execute, eg by way of an in-memory scheduler such as Quartz. See here for further information on this topic.

For example:

```
public class Order {
    @Property(
        command=CommandReification.ENABLED,
        commandExecuteIn=CommandExecuteIn.BACKGROUND
    )
    public Product getProduct() { ... }
    public void setProduct(Product p) { ... }
}
```

will result in the `Command` being persisted but its execution deferred to a background execution mechanism. The returned object from this property edit is the persisted `Command` itself.

30.1.4. `commandDtoProcessor()`

The `commandDtoProcessor()` attribute allows an implementation of `CommandDtoProcessor` to be specified. This interface has the following API:

```
public interface CommandDtoProcessor {
    CommandDto process(             ①
            Command command,        ②
            CommandDto dto);        ③
}
```

① The returned `CommandDto`. This will typically be the `CommandDto` passed in, but supplemented in some way.

② The `Command` being processed

③ The `CommandDto` (XML) obtained already from the `Command` (by virtue of it also implementing `CommandWithDto`, see discussion below).

This interface is used by the framework-provided implementations of `ContentMappingService` for the REST API, allowing `Command`s implementations that also implement `CommandWithDto` to be further customised as they are serialized out. The primary use case for this capability is in support of master/slave replication.

- on the master, `Command`s are serialized to XML. This includes the identity of the target object and the intended new value of the property.

However, any `Blob`s and `Clob`s are deliberately excluded from this XML (they are instead stored as references). This is to prevent the storage requirements for `Command` from becoming excessive. A `CommandDtoProcessor` can be provided to re-attach blob information if required.

- replaying `Command`s requires this missing parameter information to be reinstated. The `CommandDtoProcessor` therefore offers a hook to dynamically re-attach the missing `Blob` or `Clob` argument.

As a special case, returning `null` means that the command's DTO is effectively excluded when retrieving the list of commands. If replicating from master to slave, this effectively allows certain commands to be ignored. The `CommandDtoProcessor.Null` class provides a convenience implementation for this requirement.

If `commandDtoProcessor()` is specified, then `command()` is assumed to be ENABLED.

For an example application, see `Action#command()`.

30.2. `domainEvent()`

Whenever a domain object (or list of domain objects) is to be rendered, the framework fires off multiple domain events for every property, collection and action of the domain object. In the cases of the domain object's properties, the events that are fired are:

- hide phase: to check that the property is visible (has not been hidden)
- disable phase: to check that the property is usable (has not been disabled)
- validate phase: to check that the property's arguments are valid (to modify/clear its value)
- pre-execute phase: before the modification of the property
- post-execute: after the modification of the property

Subscribers subscribe through the `EventBusService` using either Guava or Axon Framework annotations and can influence each of these phases.

By default the event raised is `PropertyDomainEvent.Default`. For example:

```
public class ToDoItem {
    @Property()
    public LocalDate getDueBy() { ... }
    ...
}
```

The `domainEvent()` attribute allows a custom subclass to be emitted allowing more precise subscriptions (to those subclasses) to be defined instead. This attribute is also supported for actions and properties.

For example:

```
public class ToDoItem {
    public static class DueByChangedEvent extends PropertyDomainEvent<ToDoItem,
LocalDate> { }  ①
    @Property(domainEvent=ToDoItem.DueByChangedEvent)
    public LocalDate getDueBy() { ... }
    ...
}
```

① inherit from `PropertyDomainEvent<T,P>` where `T` is the type of the domain object being interacted with, and `P` is the type of the property (`LocalDate` in this example)

The benefit is that subscribers can be more targetted as to the events that they subscribe to.

The framework provides a no-arg constructor and will initialize the domain event using (non-API) setters rather than through the constructor. This substantially reduces the boilerplate in the subclasses because no explicit constructor is required..

30.2.1. Subscribers

Subscribers (which must be domain services) subscribe using either the Guava API or (if the `EventBusService` has been appropriately configured) using the Axon Framework API. The examples below use the Guava API.

Subscribers can be either coarse-grained (if they subscribe to the top-level event type):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(PropertyDomainEvent ev) {
        ...
    }
}
```

or can be fine-grained (by subscribing to specific event subtypes):

```
@DomainService(nature=NatureOfService.DOMAIN)
public class SomeSubscriber extends AbstractSubscriber {
    @com.google.common.eventbus.Subscribe
    public void on(ToDoItem.DueByChangedEvent ev) {
        ...
    }
}
```

If the AxonFramework is being used, replace `@com.google.common.eventbus.Subscribe` with `@org.axonframework.eventhandling.annotation.EventHandler`.

The subscriber's method is called (up to) 5 times:

- whether to veto visibility (hide)
- whether to veto usability (disable)
- whether to veto execution (validate)
- steps to perform prior to the property being modified
- steps to perform after the property has been modified.

The subscriber can distinguish these by calling `ev.getEventPhase()`. Thus the general form is:

```
@Programmatic
@com.google.common.eventbus.Subscribe
public void on(PropertyDomainEvent ev) {
    switch(ev.getEventPhase()) {
        case HIDE:
            // call ev.hide() or ev.veto("") to hide the property
            break;
        case DISABLE:
            // call ev.disable("...") or ev.veto("...") to disable the property
            break;
        case VALIDATE:
            // call ev.invalidate("...") or ev.veto("...")
            // if proposed property value is invalid
            break;
        case EXECUTING:
            break;
        case EXECUTED:
            break;
    }
}
```

It is also possible to abort the transaction during the executing or executed phases by throwing an exception. If the exception is a subtype of `RecoverableException` then the exception will be rendered as a user-friendly warning (eg Growl/toast) rather than an error.

30.2.2. Default, Doop and Noop events

If the `domainEvent` attribute is not explicitly specified (is left as its default value, `PropertyDomainEvent.Default`), then the framework will, by default, post an event.

If this is not required, then the `isis.reflector.facet.propertyAnnotation.domainEvent.postForDefault` configuration property can be set to "false"; this will disable posting.

On the other hand, if the `domainEvent` has been explicitly specified to some subclass, then an event will be posted. The framework provides `PropertyDomainEvent.Doop` as such a subclass, so setting the `domainEvent` attribute to this class will ensure that the event to be posted, irrespective of the configuration property setting.

And, conversely, the framework also provides `PropertyDomainEvent.Noop`; if `domainEvent` attribute is set to this class, then no event will be posted.

30.2.3. Raising events programmatically

Normally events are only raised for interactions through the UI. However, events can be raised programmatically by wrapping the target object using the `WrapperFactory` service.

30.3. editing()

The editing() attribute can be used to prevent a property from being modified or cleared, ie to make it read-only. This attribute can also be specified for collections, and can also be specified for the domain object.

The related editingDisabledReason() attribute specifies the a hard-coded reason why the property cannot be modified directly.

Whether a property is enabled or disabled depends upon these factors:

- whether the domain object has been configured as immutable through the @DomainObject#editing() attribute
- else (that is, if the domain object’s editability is specified as being <code>AS_CONFIGURED</code>), then the value of the configuration property <code>isis.objects.editing</code>. If set to <code>false</code>, then the object’s properties (and collections) are _not editable
- else, then the value of the @Property(editing=···) attribute itself
- else, the result of invoking any supporting disable···() supporting methods

Thus, to make a property read-only even if the object would otherwise be editable, use:

```
public class Customer {
    @Property(
        editing=Editing.DISABLED,
        editingDisabledReason="The credit rating is derived from multiple factors"
    )
    public int getInitialCreditRating(){ ... }
    public void setInitialCreditRating(int initialCreditRating) { ... }
}
```

To reiterate, it is *not* possible to enable editing for a property if editing has been disabled at the object-level.

30.4. fileAccept()

The fileAccept() attribute applies only to Blob or Clob parameters, indicating the type of file to accept when uploading a new value. The attribute is also supported on parameters.

For example:

```
public class ScannedDocument {

    @Property(fileAccept="image/*")                ①
    private Blob scannedImage;
    // getters and setters omitted

}
```

① as per reference docs, either a media type (such as `image/*`) or a file type extension (such as `.png`).

30.5. `hidden()`

Properties can be hidden at the domain-level, indicating that they are not visible to the end-user. This attribute can also be applied to actions and collections.

It is also possible to use `@Property#hidden()` to hide an action at the domain layer. Both options are provided with a view that in the future the view-layer semantics may be under the control of (expert) users, whereas domain-layer semantics should never be overridden or modified by the user.

For example:

```
public class Customer {
    @Property(hidden=Where.EVERYWHERE)
    public int getInternalId() { ... }
    @Property(hidden=Where.ALL_TABLES)
    public void updateStatus() { ... }
    ...
}
```

The acceptable values for the `where` parameter are:

- `Where.EVERYWHERE` or `Where.ANYWHERE`

  The property should be hidden everywhere.

- `Where.ANYWHERE`

  Synonym for everywhere.

- `Where.OBJECT_FORMS`

  The property should be hidden when displayed within an object form.

- `Where.PARENTED_TABLES`

  The property should be hidden when displayed as a column of a table within a parent object's

collection.

- `Where.STANDALONE_TABLES`

  The property should be hidden when displayed as a column of a table showing a standalone list of objects, for example as returned by a repository query.

- `Where.ALL_TABLES`

  The property should be hidden when displayed as a column of a table, either an object's * collection or a standalone list. This combines `PARENTED_TABLES` and `STANDALONE_TABLES`.

- `Where.NOWHERE`

  The property should not be hidden, overriding any other metadata/conventions that would normally cause the property to be hidden.

For example, if a property is annotated with `@Title`, then normally this should be hidden from all tables. Annotating with `@Property(where=Where.NOWHERE)` overrides this.

The RestfulObjects viewer has only partial support for these `Where` enums.

30.6. `maxLength()`

The `maxLength()` attribute applies only to `String` properties, indicating the maximum number of characters that the user may enter (for example in a text field in the UI). The attribute It is ignored if applied to properties of any other type. This attribute can also be applied to parameters.

That said, properties are most commonly defined on persistent domain objects (entities), in which case the JDO `@Column` will in any case need to be specified. Apache Isis can infer the `maxLength` semantic directly from the equivalent `@Column#length()` annotation/attribute.

For example:

```
public class Customer {
    @javax.jdo.annotations.Column(length=30)
    public String getFirstName() { ... }
    public void setFirstName(String firstName) { ... }
    ...
}
```

In this case there is therefore no need for the `@Property#maxLength()` attribute.

30.6.1. Non-persistent properties

Of course, not every property is persistent (it could instead be derived), and neither is every domain object an entity (it could be a view model). For these non persistable properties the `maxLength()` attribute is still required.

For example:

```
public class Customer {
    @javax.jdo.annotation.NotPersistent                    ①
    @Property(maxLength=100)
    public String getFullName() { ... }                    ②
    public void setFullName(String fullName) { ... }       ③
    ...
}
```

① a non persisted (derived) property

② implementation would most likely derive full name from constituent parts (eg first name, middle initial, last name)

③ implementation would most likely parse the input and update the constituent parts

30.7. `mustSatisfy()`

The `mustSatisfy()` attribute allows arbitrary validation to be applied to properties using an (implementation of a) `org.apache.isis.applib.spec.Specification` object. The attribute is also supported on parameters.

The specification implementations can (of course) be reused between properties and parameters.

The `Specification` is consulted during validation, being passed the proposed value. If the proposed value fails, then the value returned is the used as the invalidity reason.

For example:

```
public class StartWithCapitalLetterSpecification
        extends AbstractSpecification<String> {            ①
    public String satisfiesSafely(String proposed) {
        return "".equals(proposed)
            ? "Empty string"
            : !Character.isUpperCase(proposed.charAt(0))
                ? "Does not start with a capital letter"
                : null;
    }
}
public class Customer {
    @Property(mustSatisfy=StartWithCapitalLetterSpecification.class)
    public String getFirstName() { ... }
    ...
}
```

① the `AbstractSpecification` class conveniently handles type-safety and dealing with null values. The applib also provides `SpecificationAnd` and `SpecificationOr` to allow specifications to be

combined "algebraically".

It is also possible to provide translatable reasons. Rather than implement `Specification`, instead implement `Specification2` which defines the API:

```
public interface Specification2 extends Specification {
    public TranslatableString satisfiesTranslatable(Object obj); ①
}
```

① Return `null` if specification satisfied, otherwise the reason as a translatable string

With `Specification2` there is no need to implement the inherited `satifies(Object)`; that method will never be called.

30.8. `notPersisted()`

The (somewhat misnamed) `notPersisted()` attribute indicates that the collection should be excluded from any snapshots generated by the `XmlSnapshotService`. This attribute is also supported for collections.

This annotation does *not* specify that a property is not persisted in the JDO/DataNucleus objectstore. See below for details as to how to additionally annotate the property for this.

For example:

```
public class Order {
    @Property(notPersisted=true)
    public Order getPreviousOrder() {...}
    public void setPreviousOrder(Order previousOrder) {...}
    ...
}
```

Historically this annotation also hinted as to whether the property's value contents should be persisted in the object store. However, the JDO/DataNucleus objectstore does not recognize this annotation. Thus, to ensure that a property is actually not persisted, it should **also** be annotated with `@javax.jdo.annotations.NotPersistent`.

For example:

```
public class Order {
    @Property(notPersisted=true)                ①
    @javax.jdo.annotations.NotPersistent        ②
    public Order getPreviousOrder() {...}
    public void setPreviousOrder(Order previousOrder) {...}
    ...
}
```

① ignored by Apache Isis

② ignored by JDO/DataNucleus

Alternatively, if the property is derived, then providing only a "getter" will also work:

```
public class Order {
    public Order getPreviousOrder() {...}
    ...
}
```

30.9. `optionality()`

By default, Apache Isis assumes that all properties of an domain object or view model are required (mandatory). The `optionality()` attribute allows this to be relaxed. The attribute is also supported for parameters.

That said, properties are most commonly defined on persistent domain objects (entities), in which case the JDO `@Column` should be specified. Apache Isis can infer the maxLength directly from the equivalent @Column#length() annotation.

That said, properties are most commonly defined on persistent domain objects (entities), in which case the JDO `@Column` will in any case need to be specified. Apache Isis can infer the `optionality` semantic directly from the equivalent `@Column#allowsNull()` annotation/attribute.

For example:

```
public class Customer {
    @javax.jdo.annotations.Column(allowsNull="true")
    public String getMiddleInitial() { ... }
    public void setMiddleInitial(String middleInitial) { ... }
    ...
}
```

In this case there is no need for the `@Property#optionality()` attribute.

30.9.1. Mismatched defaults

If the `@Column#allowsNull()` attribute is omitted and the `@Property#optionality() attribute is also

omitted, then note that Isis' defaults and JDO's defaults differ. Specifically, Isis always assumes properties are mandatory, whereas JDO specifies that primitives are mandatory, but all reference types are optional.

When Apache Isis initializes it checks for these mismatches during its metamodel validation phase, and will fail to boot ("fail-fast") if there is a mismatch. The fix is usually to add the `@Column#allowsNull()` annotation/attribute.

30.9.2. Superclass inheritance type

There is one case (at least) it may be necessary to annotate the property with both `@Column#allowsNull` and also `@Property#optionality()`. If the property is logically mandatory and is in a subclass, but the mapping of the class hierarchy is to store both the superclass and subclass(es) into a single table (ie a "roll-up" mapping using `javax.jdo.annotations.InheritanceStrategy#SUPERCLASS_TABLE`), then JDO requires that the property is annotated as `@Column#allowsNull="true"`: its value will be not defined for other subclasses.

In this case we therefore require both annotations.

```
@javax.jdo.annotations.PersistenceCapable
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
public abstract class PaymentMethod {
    ...
}
@javax.jdo.annotations.PersistenceCapable
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.SUPERCLASS_TABLE)
public class CreditCardPaymentMethod extends PaymentMethod {
    private String cardNumber;
    @javax.jdo.annotations.Column(allowsNull="true")
    @Property(optionality=Optionality.MANDATORY)
    public String getCardNumber() { return this.cardNumber; }
    public void setCardNumber(String cardNumber) { this.cardNumber = cardNumber; }
    ...
}
```

Alternatively, you could rely on the fact that Apache Isis never looks at fields (whereas JDO does) and move the JDO annotation to the field:

```
@javax.jdo.annotations.PersistenceCapable
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.SUPERCLASS_TABLE)
public class CreditCardPaymentMethod extends PaymentMethod {
    @javax.jdo.annotations.Column(allowsNull="true")
    private String cardNumber;
    public String getCardNumber() { return this.cardNumber; }
    public void setCardNumber(String cardNumber) { this.cardNumber = cardNumber; }
    ...
}
```

However this at first glance this might be read as eing that the property is optional whereas Isis' default (required) applies. Also, in the future Apache Isis may be extended to support reading annotations from fields.

30.9.3. Non-persistent properties

Of course, not every property is persistent (it could instead be derived), and neither is every domain object an entity (it could be a view model). For these non persistable properties the `optionality()` attribute is still required.

For example:

```
public class Customer {
    @javax.jdo.annotation.NotPersistent                    ①
    @Property(optionality=Optionality.OPTIONAL)
    public String getFullName() { ... }                    ②
    public void setFullName(String fullName) { ... }       ③
    ...
}
```

① a non persisted (derived) property

② implementation would most likely derive full name from constituent parts (eg first name, middle initial, last name)

③ implementation would most likely parse the input and update the constituent parts

The attribute has no meaning for a primitive type such as `int`: primitives will always have a default value (e.g. zero). If optionality is required, then use the corresponding wrapper class (e.g. `java.lang.Integer`) and annotate with `Parameter#optionality()` as required.

The values for the attribute are simply `OPTIONAL` or `MANDATORY`.

For example:

```
public class Customer {
    public Order placeOrder(
            final Product product,
            @ParameterLayout(named = "Quantity")
            final int quantity,
            @Parameter(optionality = Optionality.OPTIONAL)
            @ParameterLayout(named = "Special Instructions")
            final String instr) {
        ...
    }
    ...
}
```

It is also possible to specify optionality using `@Nullable` annotation.

30.10. `regexPattern()`

There are three attributes related to enforcing regular expressions:

- The `regexPattern()` attribute validates the contents of any string property with respect to a regular expression pattern. It is ignored if applied to properties of any other type. This attribute can also be specified for parameters.
- The `regexPatternFlags()` attribute specifies flags that modify the handling of the pattern. The values are those that would normally be passed to `java.util.regex.Pattern#compile(String,int)`.
- The related `regexPatternReplacement()` attribute specifies the error message to show if the provided argument does not match the regex pattern.

For example:

```
public class Customer {
    @Property(
        regexPattern = "(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+",
        regexPatternFlags=Pattern.CASE_INSENSITIVE,
        regexPatternReplacement = "Must be valid email address (containing a '@')
symbol"   ①
    )
    public String getEmail() { ... }
}
```

① Note that there is currently no i18n support for this phrase.

Chapter 31. `@PropertyLayout`

The `@PropertyLayout` annotation applies to properties collecting together all UI hints within a single annotation.

The table below summarizes the annotation's attributes.

Table 20. `@PropertyLayout` attributes

Attribute	Values (default)	Description
`cssClass()`	Any string valid as a CSS class	the css class that a property should have, to allow more targetted styling in `application.css`
`describedAs()`	String	description of this property, eg to be rendered in a tooltip.
`hidden()`	`EVERYWHERE`, `OBJECT_FORMS`, `PARENTED_TABLES`, `STANDALONE_TABLES`, `ALL_TABLES`, `NOWHERE` (`NOWHERE`)	indicates where (in the UI) the property should be hidden from the user.
`labelPosition()`	`LEFT`, `TOP`, `RIGHT`, `NONE` (`LEFT`)	in forms, the positioning of the label relative to the property value. Defaults is `LEFT`, unless `multiLine` in which case `TOP`. The value `RIGHT` is only supported for boolean properties. It is also possible to change the default through a configuration property
`multiLine()`	Positive integer	for string properties, render as a text area over multiple lines. If set > 1, then `labelPosition` defaults to TOP.
`named()`	String	to override the name inferred from the collection's name in code. A typical use case is if the desired name is a reserved Java keyword, such as `default` or `package`.
`namedEscaped()`	`true`, `false` (`true`)	whether to HTML escape the name of this property.
`promptStyle()`	`DIALOG`, `INLINE`, `AS_CONFIGURED` (`AS_CONFIGURED`)	how a property prompt should be displayed within the UI
`renderedAsDayBefore()`	`true`, `false` (`false`)	for date properties only, render the date as one day prior to the actually stored date.
`typicalLength()`	Positive integer.	the typical entry length of a field, use to determine the optimum width for display

Attribute	Values (default)	Description
`unchanging()`	`false`, `true` (`false`)	indicates that the value held by the property never changes over time (even if other properties of the object do change). Used as a hint to the viewer not to redraw the property if possible after an AJAX update.

For example:

```
public class ToDoItem {
    @PropertyLayout(
        cssClass="x-key",
        named="Description of this <i>item</i>",
        namedEscaped=false,
        describedAs="What needs to be done",
        labelPosition=LabelPosition.LEFT,
        typicalLength=80
    )
    public String getDescription() { ... }
    ...
}
```

It is also possible to apply the annotation to actions of domain services that are acting as contributed properties.

As an alternative to using the `@PropertyLayout` annotation, a file-based layout can be used (and is generally to be preferred since it is more flexible/powerful).

The annotation is one of a handful (others including `@Collection`, `@CollectionLayout` and `@Property`) that can also be applied to the field, rather than the getter method. This is specifically so that boilerplate-busting tools such as Project Lombok can be used.

31.1. `cssClass()`

The `cssClass()` attribute can be used to render additional CSS classes in the HTML (a wrapping `<div>`) that represents the property. Application-specific CSS can then be used to target and adjust the UI representation of that particular element.

This attribute can also be applied to domain objects, view models, actions collections and parameters.

For example:

```
public class ToDoItem {
    @PropertyLayout(cssClass="x-key")
    public LocalDate getDueBy() { ... }
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"dueBy": {
    "propertyLayout": { "cssClass": "x-key" }
}
```

31.2. describedAs()

The `describedAs()` attribute is used to provide a short description of the property to the user. In the Wicket viewer it is displayed as a 'tool tip'. The attribute can also be specified for collections, actions, parameters, domain objects and view models.

For example:

```
public class Customer {
    @DescribedAs("The name that the customer has indicated that they wish to be " +
                 "addressed as (e.g. Johnny rather than Jonathan)")
    public String getFirstName() { ... }
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"firstName:" {
    "propertyLayout": {
        "describedAs": "The name that the customer has indicated that they wish to be
addressed as (e.g. Johnny rather than Jonathan)"
     }
}
```

31.3. labelPosition()

The `labelPosition()` attribute determines the positioning of labels for properties. This attribute can also be specified for parameters.

The positioning of labels is typically `LEFT`, but can be positioned to the `TOP`. The one exception is

`multiLine()` string properties, where the label defaults to `TOP` automatically (to provide as much real-estate for the multiline text field as possible).

For boolean properties a positioning of `RIGHT` is also allowed; this is ignored for all other types.

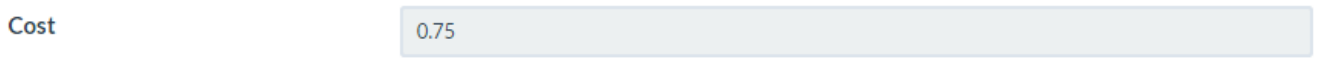
It is also possible to suppress the label altogether, using `NONE`.

For example:

```
public class ToDoItem {
    @PropertyLayout(
        labelPosition=LabelPosition.TOP
    )
    public String getDescription() { ... }
    public void setDescription(String description) { ... }
    ...
}
```

To get an idea of how these are rendered (in the Wicket viewer), we can look at the (non-ASF) Isis addons' todoapp that happens to have examples of most of these various label positions.

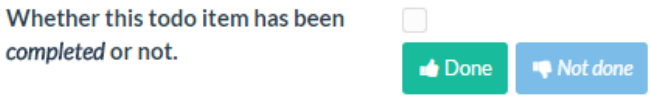
The default `LEFT` label positioning is used by the `cost` property:



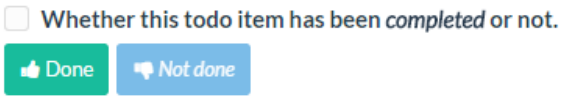
The `TOP` label positioning is used by the `category` property:

Labels are suppressed, using `NONE`, for the `subcategory` property:

The todoapp's `complete` (boolean) property renders the label to the LEFT (the default):



Moving the label to the `RIGHT` looks like:



As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"description": {
    "propertyLayout": {
        "labelPosition": "TOP"
    }
}
```

Specifying a default setting for label positions

If you want a consistent look-n-feel throughout the app, eg all property labels to the top, then it'd be rather frustrating to have to annotate every property.

Instead, a default can be specified using a configuration property in `isis.properties`:

```
isis.viewers.propertyLayout.labelPosition=TOP
```

or

```
isis.viewers.propertyLayout.labelPosition=LEFT
```

If these are not present then Apache Isis will render according to internal defaults. At the time of writing, this means labels are to the left for all datatypes except multiline strings.

31.4. `multiLine()`

The `multiLine()` attribute specifies that the text field for a string property should span multiple lines. It is ignored for other property types. The attribute is also supported for parameters.

For example:

```
public class BugReport {
    @PropertyLayout(
        numberOfLines=10
    )
    public String getStepsToReproduce() { ... }
    public void setStepsToReproduce(String stepsToReproduce) { ... }
    ...
}
```

Here the `stepsToReproduce` will be displayed in a text area of 10 rows.

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"stepsToReproduce": {
    "propertyLayout": {
        "numberOfLines": 10
    }
}
```

If set > 1 (as would normally be the case), then the default `labelPosition` defaults to `TOP` (rather than `LEFT`, as would normally be the case).

31.5. named()

The `named()` attribute explicitly specifies the property's name, overriding the name that would normally be inferred from the Java source code. This attribute can also be specified for actions, collections, parameters, domain objects, view models and domain services.

Following the don't repeat yourself principle, we recommend that you only use this attribute when the desired name cannot be used in Java source code. Examples of that include a name that would be a reserved Java keyword (eg "package"), or a name that has punctuation, eg apostrophes.

By default the name is HTML escaped. To allow HTML markup, set the related `namedEscaped()` attribute to `false`.

For example:

```
public class ToDoItem {
    @PropertyLayout(
        named="Description of this <i>item</i>",
        namedEscaped=false
    )
    public String getDescription() { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"description": {
    "propertyLayout": {
        "named": "Description of this <i>item</i>",
        "namedEscaped": false
    }
}
```

The framework also provides a separate, powerful mechanism for internationalization.

31.6. promptStyle()

The `promptStyle()` attribute is used to specify whether, when editing a domain object property, the new value for the property is prompted by way of a dialog box, or is prompted using an inline panel (replacing the property on the page).

If the attribute is not set, then the value of the configuration property `isis.viewer.wicket.promptStyle` is used. If this is itself not set, then an inline prompt is used.

For example:

```
public class Customer {
    @PropertyLayout(
        promptStyle=PromptStyle.INLINE          ①
    )
    public int getNotes(){ ... }
    public void setNotes(String notes) { ... }
}
```

① prompt for the new value for the property using an inline panel Note that the value `INLINE_AS_IF_EDIT` does not make sense for properties; if specified then it will be interpreted as just `INLINE`.

Alternatively, the `promptStyle()` can be specified using file-based layouts.

FIXME - provide an example here

31.7. renderedAsDayBefore()

The `renderedAsDayBefore()` attribute applies only to date properties whereby the date will be rendered as the day before the value actually held in the domain object. It is ignored for properties of other types. This attribute is also supported for parameters.

This behaviour might at first glance appear odd, but the rationale is to support the use case of a sequence of instances that represent adjacent intervals of time. In such cases there would typically be `startDate` and `endDate` properties, eg for all of Q2. Storing this as a half-closed interval — eg `[1-`

`Apr-2015, 1-July-2015)` — can substantially simplify internal algorithms; the `endDate` of one interval will correspond to the `startDate` of the next.

However, from an end-user perspective the requirement may be to render the interval as a fully closed interval; eg the end date should be shown as `30-Jun-2015`.

This attribute therefore bridges the gap; it presents the information in a way that makes sense to an end-user, but also stores the domain object in a way that is easy work with internally.

For example:

```
public class Tenancy {
    public LocalDate getStartDate() { ... }
    public void setStartDate(LocalDate startDate) { ... }
    @PropertyLayout(
        renderedAsDayBefore=true
    )
    public LocalDate getEndDate() { ... }
    public void setEndDate(LocalDate EndDate) { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - change to .layout.xml syntax instead.

```
"endDate": {
    "propertyLayout": {
        "renderedAsDayBefore": true
    }
}
```

31.8. `typicalLength()`

The `typicalLength()` attribute indicates the typical length of a string property. It is ignored for properties of other types. The attribute is also supported for parameters.

The information is intended as a hint to the UI to determine the space that should be given to render a particular string property. That said, note that the Wicket viewer uses the maximum space available for all fields, so in effect ignores this attribute.

For example:

```
public class Customer {
    @javax.jdo.annotations.Column(length=30)
    @ParameterLayout(typicalLength=20)
    public String getFirstName() { ... }
    public void setFirstName(String firstName) { ... }
    ...
}
```

As an alternative to using the annotation, the dynamic file-based layout can be used instead, eg:

FIXME - provide a .layout.xml example here.

31.9. unchanging()

The `unchanging()` attribute is used to indicate that the value held by the property never changes over time, even when other properties of the object do change.

Setting this attribute to `true` is used as a hint to the viewer to not redraw the property after an AJAX update of some other property/ies of the object have changed. This is primarily for performance, eg can improve the user experience when rendering PDFs/blobs.

Note that for this to work, the viewer will also ensure that none of the property's parent component (such as a tab group panel) are re-rendered.

Design note: we considered implementing this an "immutable" flag on the `@Property` annotation (because this flag is typically appropriate for immutable/unchanging properties of a domain object). However, we decided not to do that, on the basis that it might be interpreted as having a deeper impact within the framework than simply a hint for rendering.

For example:

```
public class Document {
    @PropertyLayout(
        unchanging=true
    )
    public Blob getBlob(){ ... }
    public void setBlob(Blob blob) { ... }
}
```

Chapter 32. `@RequestScoped` (`javax`)

The `@javax.enterprise.context.RequestScoped` JSR-299 CDI annotation is used to specify that a domain service should be request-scoped rather than a singleton.

Although Apache Isis does not (currently) leverage CDI, the semantics are the same as request-scoped service; a new instance is created for each HTTP request, reserved for the exclusive use of all objects interacted with during that request.

One of the built-in domain services that uses this annotation is `Scratchpad`, intended to allow the arbitrary sharing of data between objects. Here is the full source code of this service is:

```
@DomainService(
        nature = NatureOfService.DOMAIN
)
@RequestScoped
public class Scratchpad {
    private final Map<Object, Object> userData = Maps.newHashMap();    ①
    @Programmatic
    public Object get(Object key) {
        return userData.get(key);                                      ②
    }
    @Programmatic
    public void put(Object key, Object value) {
        userData.put(key, value);                                      ③
    }
    @Programmatic
    public void clear() {
        userData.clear();                                              ④
    }
}
```

① Provides a mechanism for each object being acted upon to pass data to the next object.

② Obtain user-data, as set by a previous object being acted upon.

③ Set user-data, for the use of a subsequent object being acted upon.

④ Clear any user data.

The vast majority of domain services in Apache Isis tend to be singletons (which requires no special annotation); but as you can see setting up request-scoped services is very straightforward.

Behind the covers Apache Isis creates a (singleton) wrapper for the domain service; the individual request-scoped instances are held in a thread-local of this wrapper. One consequence of this implementation is that request-scoped methods should not be marked as `final`.

Chapter 33. `@Title`

The `@Title` annotation is used to indicate which property or properties make up the object title. If more than one property is used, the order can be specified (using the same Dewey-decimal notation as used by `@MemberOrder`) and the string to use between the components can also be specified.

For example:

```
public void Customer {
    @Title(sequence="1.0")
    public String getLastName() { ... }     ①
    ...
    @Title(sequence="1.5", prepend=", ")
    public String getFirstName() { ... }
    ...
    @Title(sequence="1.7", append=".")
    public String getMidInitial() { ... }
    ...
}
```

① backing field and setters omitted

could be used to create names of the style "Bloggs, Joe K."

It is also possible to annotate reference properties; in this case the title will return the title of the referenced object (rather than, say, its string representation).

An additional convention for `@Title` properties is that they are hidden in tables (in other words, it implies `@Hidden(where=Where.ALL_TABLES)`. For viewers that support this annotation (for example, the Wicket viewer), this convention excludes any properties whose value is already present in the title column. This convention can be overridden using `@Hidden(where=Where.NOWHERE)`.

33.1. Lombok support

If Project Lombok is being used, then `@Title` can be specified on the backing field.

For example:

```
public void Customer {
    @Title(sequence="1.0")
    @Getter @Setter
    private String name;

    @Title(sequence="1.5", prepend=", ")
    @Getter @Setter
    private String firstName;

    @Title(sequence="1.7", append=".")
    @Getter @Setter
    private String midInitial;
}
```

Chapter 34. `@ViewModel`

The `@ViewModel` annotation, applied to a class, indicates that the class is a view model. It's a synonym for using `@DomainObject(nature=VIEW_MODEL)`.

View models are not persisted to the database, instead their state is encoded within their identity (ultimately represented in the URL).

For example:

```
@ViewModel
public class CustomerViewModel {
    public CustomerViewModel() {}
    public CustomerViewModel(String firstName, int lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    ...
}
```

Although there are several ways to instantiate a view model, we recommend that they are instantiated using an N-arg constructor that initializes all relevant state. The `ServiceRegistry` can then be used to inject dependencies into the view model. For example:

```
Customer cust = ...
CustomerViewModel vm = new CustomerViewModel(cust.getFirstName(),cust.getLastName());
serviceRegistry.injectServicesInto(vm);
```

See this tip for further discussion about instantiating view models.

View models must have a no-arg constructor; this is used internally by the framework for subsequent "recreation".

The view model’s memento will be derived from the value of the view model object’s properties. Any <code>@Property#notPersisted()</code> properties will be excluded from the memento, as will any <code>@Programmatic</code> properties. Properties that are merely hidden _are included in the memento.

Only properties supported by the configured `MementoService` can be used. The default implementation supports all the value types and persisted entities.

View models, as defined by `@ViewModel` (or `@DomainObject(nature=VIEW_MODEL)` for that matter) have some limitations:

- view models cannot hold collections other view models (simple properties *are* supported,

though)

- collections (of either view models or entities) are ignored.
- not every data type is supported,

However, these limitations do *not* apply to JAXB view models. If you are using view models heavily, you may wish to restrict yourself to just the JAXB flavour.

The `@ViewModel` does not allow the objectType to be specified, meaning that it is incompatible with the metamodel validation check ennabled by the `explicitObjectType` configuration property.

Instead, use `@DomainObject#nature()` with `Nature.VIEW_MODEL`, and specify `@DomainObject#objectType()`.

Chapter 35. `@ViewModelLayout`

The `@ViewModelLayout` annotation is identical to the `@DomainObjectLayout`, but is provided for symmetry with domain objects that have been annotated using `@ViewModel` (rather than `@DomainObject(nature=VIEW_MODEL)`).

The table below summarizes the annotation's attributes.

Table 21. `@ViewModel` attributes

Attribute	Values (default)	Description
`cssClass()`	Any string valid as a CSS class	the css class that a domain class (type) should have, to allow more targetted styling in `application.css`
`cssClassFa()`	Any valid Font awesome icon name	specify a font awesome icon for the action's menu link or icon.
`cssClassFaPosition()`	`LEFT`, `RIGHT` (`LEFT`)	Currently unused.
`describedAs()`	String.	description of this class, eg to be rendered in a tooltip.
`named()`	String.	to override the name inferred from the action's name in code. A typical use case is if the desired name is a reserved Java keyword, such as `default` or `package`.
`paged()`	Positive integer	the page size for instances of this class when rendered within a table (as returned from an action invocation)
`plural()`	String.	the plural name of the class

For example:

```
@ViewModel                                  ①
@ViewModelLayout(
    cssClass="x-analysis",
    cssClassFa="fa-piechart",
    describedAs="Analysis of todo items by category"
)
public class CategoryPieChart { ... }
```

① this annotation is intended for use with `@ViewModel`. If a view model has been specified using the equivalent `@DomainObject(nature=Nature.VIEW_MODEL)`, then we recommend you use `@DomainObjectLayout` instead.

Note that there is (currently) no support for specifying UI hints for view models through the dynamic `.layout.json` file (only for properties, collections and actions are supported).

35.1. `cssClass()`

The `cssClass()` attribute can be used to render additional CSS classes in the HTML (a wrapping `<div>`) that represents the view model. Application-specific CSS can then be used to target and adjust the UI representation of that particular element.

This attribute can also be applied to domain objects, actions properties, collections and parameters.

For example:

```
@ViewModel
@ViewModelLayout(cssClass="x-analysis")
public class CategoryPieChart { ... }
```

The similar `@ViewModelLayout#cssClassFa()` annotation attribute is also used as a hint to apply CSS, but in particular to allow Font Awesome icons to be rendered as the icon for classes.

35.2. `cssClassFa()`

The `cssClassFa()` attribute is used to specify the name of a Font Awesome icon name, to be rendered as the domain object's icon.

These attribute can also be applied to domain objects to specify the object's icon, and to actions to specify an icon for the action's representation as a button or menu item.

If necessary the icon specified can be overridden by a particular object instance using the `iconName()` method.

For example:

```
@ViewModel
@ViewModelLayout(
    cssClassFa="fa-piechart"
)
public class CategoryPieChart { ... }
```

There can be multiple "fa-" classes, eg to mirror or rotate the icon. There is no need to include the mandatory `fa` "marker" CSS class; it will be automatically added to the list. The `fa-` prefix can also be omitted from the class names; it will be prepended to each if required.

The related `cssClassFaPosition()` attribute is currently unused for domain objects; the icon is

always rendered to the left.

The similar `@ViewModelLayout#cssClass()` annotation attribute is also used as a hint to apply CSS, but for wrapping the representation of an object or object member so that it can be styled in an application-specific way.

35.3. `describedAs()`

The `describedAs()` attribute is used to provide a short description of the view model to the user. In the Wicket viewer it is displayed as a 'tool tip'. The `describedAs()` attribute can also be specified for collections, properties, actions, parameters and domain objects.

For example:

```
@ViewModel
@ViewModelLayout(
    cssClass="x-analysis",
    cssClassFa="fa-piechart",
    describedAs="Analysis of todo items by category"
)
public class CategoryPieChart { ... }
```

35.4. `named()`

The `named()` attribute explicitly specifies the view model's name, overriding the name that would normally be inferred from the Java source code. This attribute can also be specified for actions, collections, properties, parameters, domain objects and domain services.

Following the don't repeat yourself principle, we recommend that you only use this attribute when the desired name cannot be used in Java source code. Examples of that include a name that would be a reserved Java keyword (eg "package"), or a name that has punctuation, eg apostrophes.

For example:

```
@ViewModel
@ViewModelLayout(
    named="PieChartAnalysis"
)
public class PieChartAnalysisViewModel {
   ...
}
```

The framework also provides a separate, powerful mechanism for internationalization.

35.5. paged()

The `paged()` attribute specifies the number of rows to display in a standalone collection, as returned from an action invocation. This attribute can also be applied to collections and domain objects.

The RestfulObjects viewer currently does not support paging. The Wicket viewer *does* support paging, but note that the paging is performed client-side rather than server-side.

We therefore recommend that large collections should instead be modelled as actions (to allow filtering to be applied to limit the number of rows).

For example:

```
@ViewModel
@ViewModelLayout(paged=15)
public class OrderAnalysis {
    ...
}
```

It is also possible to specify a global default for the page size of standalone collections, using the configuration property `isis.viewer.paged.standalone`.

35.6. plural()

When Apache Isis displays a standalone collection of several objects, it will label the collection using the plural form of the object type.

By default the plural name will be derived from the end of the singular name, with support for some basic English language defaults (eg using "ies" for names ending with a "y").

The `plural()` attribute allows the plural form of the class name to be specified explicitly. This attribute is also supported for domain objects.

For example:

```
@ViewModel
@ViewModelLayout(plural="Children")
public class Child {
    ...
}
```

Chapter 36. `@XmlJavaTypeAdapter` (`jaxb`)

The JAXB `@XmlJavaTypeAdapter` annotation is used with the framework-provided `PersistentEntityAdapter` to instruct JAXB to serialize references to persistent entities using the canonical `OidDto` complex type: the object's type and its identifier. This is the formal XML equivalent to the `Bookmark` provided by the `BookmarkService`.

For example:

```
@XmlJavaTypeAdapter(PersistentEntityAdapter.class)
public class ToDoItem ...  {
    ...
}
```

This annotation therefore allows view models/DTOs to have references to persistent entities; a common idiom.

For a more complete discussion of writing JAXB view models/DTOs, see this topic in the user guide.

Chapter 37. `@XmlRootElement` (`jaxb`)

The `@XmlRootElement` annotation provides an alternative way to define a view model, in particular one intended to act as a DTO for use within RestfulObjects viewer, or which contains arbitrarily complex state.

A view model is a non-persisted domain object whose state is converted to/from a string memento. In the case of a JAXB-annotated object this memento is its XML representation. JAXB generally requires that the root element of the XML representation is annotated with `@XmlRootElement`. Apache Isis makes this a mandatory requirement.

In comparison to using either the `ViewModel` interface or the `@ViewModel` annotation, using `@XmlRootElement` has a couple of significant advantages:

- the view model can be used as a "canonical" DTO, for example when accessing data using the RestfulObjects viewer in combination with the `ContentMappingService`.

  This provides a stable and versioned API to access data in XML format using whatever client-side technology may be appropriate.

- the XML graph can be as deep as required; in particular it can contain collections of other objects.

  In contrast, if the `@ViewModel` annotation is used then only the state of the properties (not collections) is captured. If using `ViewModel` interface then arbitrary state (including that of collections), however the programmer must write all the code by hand

The main disadvantages of using JAXB-annotated view models is that any referenced persistent entity must be annotated with the `@XmlJavaTypeAdapter`, using the framework-provided `PersistentEntityAdapter`. This adapter converts any references to such domain entities using the `oidDto` complex type (as defined by the Apache Isis common schema): the object's type and its identifier.

The memento string for view models is converted into a form compatible with use within a URL. This is performed by the `UrlEncodingService`, the default implementation of which simply encodes to base 64. If the view model XML graph is too large to be serialized to a string, then an alternative implementation (eg which maps XML strings to a GUID, say) can be configured using the technique described in here in the user guide.

37.1. Example

This example is taken from the (non-ASF) Isis addons' todoapp:

```
@XmlRootElement(name = "toDoItemDto")           ①
public class ToDoItemDto implements Dto {
    @Getter @Setter                             ②
    protected String description;
    @Getter @Setter
    protected String category;
    @Getter @Setter
    protected String subcategory;
    @Getter @Setter
    protected BigDecimal cost;
}
```

① identifies this class as a view model and defines the root element for JAXB serialization

② using Project Lombok for getters and setters

37.2. See also

Although (like any other viewmodel) a JAXB-annotated can have behaviour (actions) and UI hints, you may wish to keep the DTO "clean", just focused on specifying the data contract.

Behaviour can therefore be provided using mixins (annotated with `@Mixin`), while UI events can be used to obtain title, icons and so on.

For a more complete discussion of writing JAXB view models/DTOs, see this topic in the user guide.