

# Classes, Methods and Schema

# Table of Contents

1. Classes, Methods and Schema .....	1
1.1. Other Guides .....	1
2. Methods .....	2
2.1. Supporting Method Prefixes .....	2
2.2. Reserved Methods .....	21
2.3. Lifecycle Methods .....	28
3. Classes and Interfaces .....	32
3.1. <b>AppManifest</b> (bootstrapping) .....	32
3.2. Superclasses .....	37
3.3. Domain Event Classes .....	40
3.4. UI Event Classes .....	43
3.5. Lifecycle Events .....	47
3.6. Value Types .....	52
3.7. Applib Utility Classes .....	54
3.8. Specification pattern .....	57
3.9. i18n support .....	59
3.10. Contributee .....	59
3.11. Roles .....	60
3.12. Mixins .....	62
3.13. Layout .....	65
4. Schema .....	68
4.1. Common Schema .....	68
4.2. Action Invocation Memento .....	70

# Chapter 1. Classes, Methods and Schema

This reference guide lists and describes various elements of the the Apache Isis Programming Model, specifically reserved and prefix [methods](#) (such as `title()` and `validate...()`) and various utility and supporting [classes](#).

It also describes the [XSD schema](#) defined by Apache Isis. One use case is for the JAXB serialization of view models.

## 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](#)
- [Security](#)
- [Testing](#)
- [Beyond the Basics](#)

The reference guides are:

- [Annotations](#)
- [Domain Services](#)
- [Configuration Properties](#)
- [Classes, Methods and Schema](#) (this guide)
- [Apache Isis Maven plugin](#)

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)

# Chapter 2. Methods

The Apache Isis metamodel is built up from declaratively (ie, [annotations](#)) and imperatively, from "supporting" methods and other reserved methods.

This chapter documents the supporting methods and the reserved methods. It also documents (separately) the reserved methods that act as callback hooks into the persistence lifecycle of domain entities.

## 2.1. Supporting Method Prefixes

Supporting methods are those that are associated with properties, collections and actions, providing additional imperative business rule checking and behaviour to be performed when the user interacts with those object members.

This association is performed by name matching. Thus, a property called "firstName", derived from a method `getFirstName()` may have supporting methods `hideFirstName()`, `disableFirstName()` and `validateFirstName()`. Supporting methods are, therefore, each characterized by their own particular prefix.



Using name matching to associate supporting methods generally works very well, but of course if an object member's method is renamed, there's always the risk that the developer forgets to rename the supporting method; the supporting methods become "orphaned".

Apache Isis checks for this automatically, and will fail-fast (fail to boot) if any orphaned methods are located. A suitable error message is logged so that the issue can be easily diagnosed.

The table below lists the method prefixes that are recognized as part of Apache Isis' default programming model.

Table 1. Recognized Method Prefixes

Prefix	Object	Property	Collection	Action	Action Param	Description
<code>addTo…()</code>			Y			add object to a collection [NOTE] ==== Directly mutable collections are not currently supported by the <a href="#">Wicket viewer</a> . ==== See also <code>removeFrom…()</code>
<code>autoComplete…()</code>		Y			Y	Return a list of matching elements for a property or an action parameter. Alternatively, can specify for a class using <code>@DomainObject #autoCompleteRepository</code> See also <code>choices…()</code>

Prefix	Object	Property	Collection	Action	Action Param	Description
<code>choices…()</code>		Y			Y	Provide list of choices for a property or action parameter. See also <code>autoComplete…()</code> .
<code>clear…()</code>		Y				Clear a property (set it to null). Allows business logic to be placed apart from the setter. See also <code>modify…()</code>
<code>default…()</code>		Y			Y	Default value for a property or an action parameter.
<code>disable…()</code>	Y	Y	Y	Y		Disables (makes read-only) a property, a collection or an action.
<code>get…()</code>		Y	Y			Access the value of a property or collection. See also <code>set…()</code> .
<code>hide…()</code>		Y	Y	Y		Hides a property, a collection or an action.
<code>modify…()</code>		Y				Modify a property (set it to a non-null) value. Allows business logic to be placed apart from the setter. See also <code>clear…()</code> .
<code>removeFrom…()</code>			Y			remove object from a collection. [NOTE] ==== Directly mutable collections are not currently supported by the <a href="#">Wicket viewer</a> . ==== See also <code>addTo…()</code>
<code>set…()</code>		Y	Y			Sets the value of a property or a collection.
<code>validate…()</code>	Y			Y	Y	Check that a proposed value of a property or a set of action parameters or a single action parameter is valid. See also <code>validateAddTo…()</code> and <code>validateRemoveFrom…()</code> to validate modifications to collections.
<code>validateAddTo…()</code>		Y				Check that a proposed object to add to a collection is valid. [NOTE] ==== Directly mutable collections are not currently supported by the <a href="#">Wicket viewer</a> . ==== See also <code>validateRemoveFrom…()</code> , and <code>validate…()</code> for properties and actions.

Prefix	Object	Property	Collection	Action	Action Param	Description
<code>validateRemoveFrom…()</code>		Y				Check that a proposed object to remove from a collection is valid. [NOTE] ==== Directly mutable collections are not currently supported by the <a href="#">Wicket viewer</a> . ==== See also <code>validateAddTo…()</code> , and <code>validate…()</code> for properties and actions.

### 2.1.1. `addTo…()`

The `addTo…()` supporting method is called whenever an object is added to a collection. Its purpose is to allow additional business logic to be performed.



Directly mutable collections are not currently supported by the [Wicket viewer](#). The suggested workaround is to simply define an action.

For example:

```
public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public void addToBorrowed(Book book) {
        borrowed.add(book); ①
        reminderService.addReminder(this, book, clock.today().plusDays(21)); ②
    }
    public void removeFromBorrowed(Book book) { ... }
    ...
}
```

① update the collection

② perform some additional business logic

See also `removeFrom…()`

### 2.1.2. `autoComplete…()`

The `autoComplete…()` supporting method is called for action parameters and for properties to find objects from a drop-down list box. The use case is when the number of candidate objects is expected to be large, so the user is required to enter some characters to narrow the search down.



If the number of candidate objects is comparatively small, then use `choices…()` supporting method instead.

The signature of the supporting method depends on whether it is for a parameter or a property.

## Parameters

For an action parameter in (0-based) position  $N$ , and of type  $T$ , the signature is:

```
public List<T> autoCompleteNxx(String search) { ... }
```

It is also valid to return  $T[]$ , a  $Set<T>$  or a  $Collection<T>$ .

For example:

```
public class ShoppingCartItem {
    @Property(editing=Editing.DISABLED)
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    @Property(editing=Editing.DISABLED)
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

    @Action(semantic=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final int quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Collection<Product> autoCompleteUpdateProduct( ①
        @MinLength(3) String search ②
    ) {
        ...
    }
    ...
}
```

① `product` is the 0th argument of the action.

② the `@MinLength` annotation specifies the minimum number of characters that must be entered before a search is performed for matching objects

## Properties

For a property of type  $T$ , the signature is:

```
public List<T> autoCompleteXxx(String search) { ... }
```

(As for action parameters) it is also valid to return  $T[]$ , a  $Set<T>$  or a  $Collection<T>$ .

For example:

```

public class ShoppingCartItem {
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    public Collection<Product> autoCompleteProduct(
        @MinLength(3) String search           ①
    ) {
        ...
    }
    ...
}

```

① the `@MinLength` annotation specifies the minimum number of characters that must be entered before a search is performed for matching objects

### 2.1.3. `choices…()`

The `choices…()` supporting method is called for both action parameters and for properties, to find objects from a drop-down list box. Unlike `autoComplete…()`, the use case is when the number of objects is comparatively small and can be selected from a drop-down without any additional filtering.

The signature of the supporting method depends on whether it is for an action parameter or a property.

#### Parameters

For an action parameter in (0-based) position  $N$ , and of type  $T$ , the signature is:

```

public Collection<T> choicesNXxx() { ... }

```

For example:



```

public class ShoppingCartItem {
    @Property(editing=Editing.DISABLED)
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    @Property(editing=Editing.DISABLED)
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

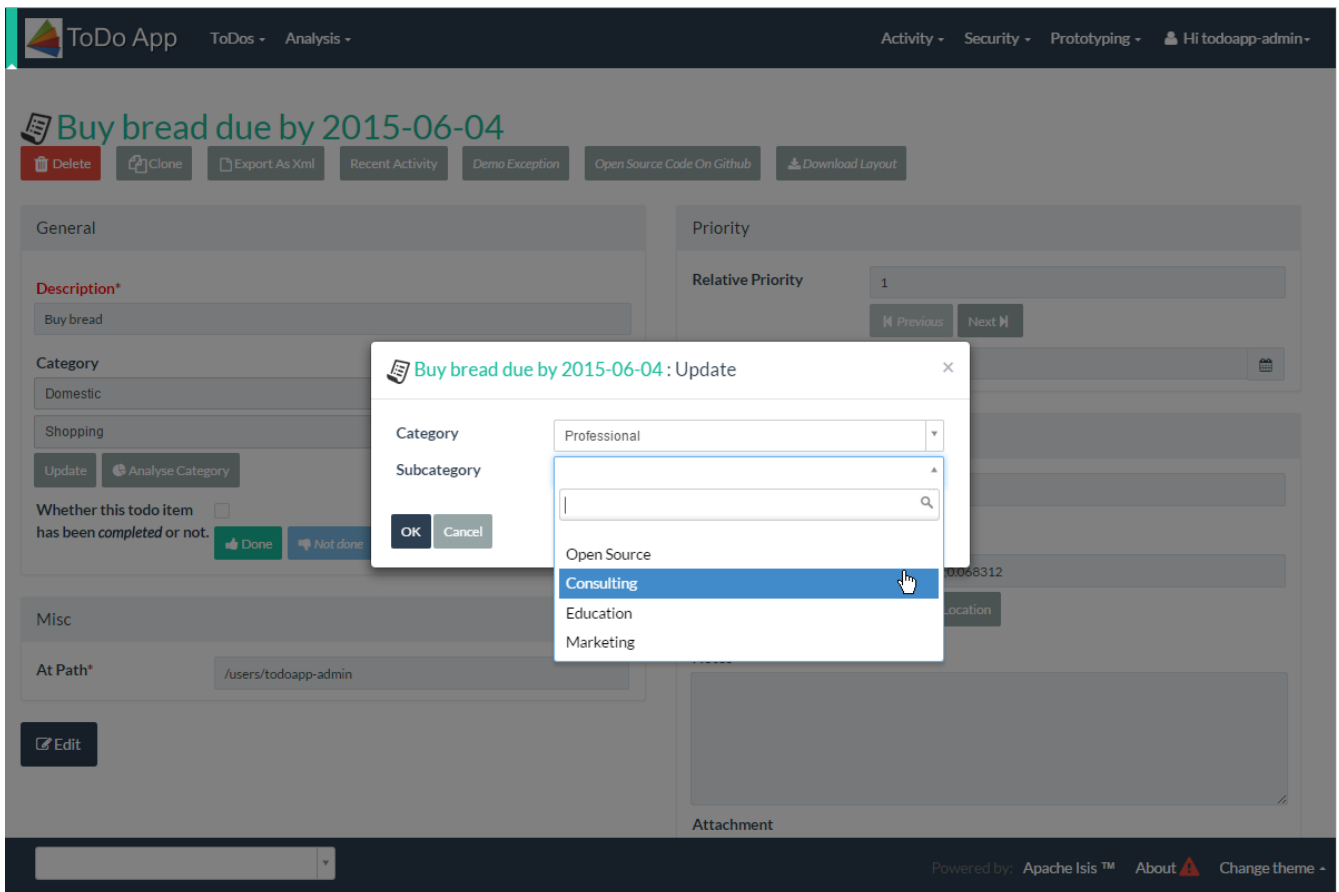
    @Action(semantic=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final Integer quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Collection<Integer> choices1UpdateProduct() {
        return Arrays.asList(1,2,3,5,10,25,50,100);
    }
    ...
}

```

### Dependent Choices

Action parameters also support the notion of dependent choices, whereby the list of choices is dependent upon the value of some other argument.

An example can be found in the (non-ASF) [Isis addons' todoapp](#), whereby `ToDoItem`s are categorized and then can also be subcategorized:



This functionality is actually implemented as a [contributed action](#), so the code for this is:

```

@DomainService(nature = NatureOfService.VIEW_CONTRIBUTIONS_ONLY)
public class UpdateCategoryContributions ... {
    @ActionLayout(
        describedAs = "Update category and subcategory"
    )
    @Action(semantic = SemanticsOf.IDEMPOTENT)
    public Categorized updateCategory(
        final Categorized item,
        final Category category,
        @Parameter(optionality = Optionality.OPTIONAL)
        final Subcategory subcategory) {
        item.setCategory(category);
        item.setSubcategory(subcategory);
        return item;
    }
    public List<Subcategory> choices2UpdateCategory(
        final Categorized item,
        final Category category) {
        return Subcategory.listFor(category);
    }
    ...
}

```

① `ToDoItem` implements `Categorized`

- ② subcategory is the 2-th argument (0-based)
- ③ the item contributed to
- ④ the category selected

Dependent choices are not restricted to enums, however. Going back to the shopping cart example shown above, the choices for the `quantity` parameter could be dependent upon the selected `Product`:

```
public class ShoppingCartItem {
    ...
    @Action(semantic=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final Integer quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Collection<Integer> choices1UpdateProduct(Product product) {
        return productService.quantityChoicesFor(product);
    }
    ...
}
```

- ① `productService` is a (fictitious) injected service that knows what the quantity choices should be for any given product

## Properties

For a property of type `T`, the signature is:

```
public Collection<T> choicesXxx() { ... }
```

For example:

```
public class ShoppingCartItem {
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    public Collection<Product> choicesProduct() {
        ...
    }
}
```

### 2.1.4. `clear…()`

The `clear…()` supporting method is called—instead of the setter—whenever an (optional) property is to be set to `null`. Its purpose is to allow additional business logic to be performed.

For example:

```
public class LibraryMember {
    public Title getFavoriteTitle() { ... }
    public void setFavoriteTitle(Title title) { ... }
    public void modifyFavoriteTitle(Title title) { ... }
    public void clearFavoriteTitle() {
        if(getTitle() == null) { return; }
        setFavoriteTitle(null);           ①
        titleFavoritesService.decrement(title); ②
    }
    ...
}
```

① update the property

② perform some additional business logic

See also `modify…()`

### 2.1.5. `default…()`

The `default…()` supporting method is called for action parameters to return the initial argument value. This may be some sensible default (eg today's date, or 0 or 1), or— for an action that is modifying the state of an object — might default to the current value of a corresponding property.

The method is *also* called for properties in the case when an object is newly instantiated using `DomainObjectContainer#newTransientInstance(…)`. This is a much less common use case. If a default is not specified then properties are initialized to a default based on their type (eg 0 or `false`).

The signature of the supporting method depends on whether it is for an action parameter or a property.

#### Parameters

For an action parameter in (0-based position *n*), and of type *T*, the signature is:

```
public T defaultNXxx() { ... }
```

For example:

```

public class ShoppingCartItem {
    @Property(editing=Editing.DISABLED)
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    @Property(editing=Editing.DISABLED)
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

    @Action(semantic=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final Integer quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Product default0UpdateProduct() {      ①
        return getProduct();
    }
    public int default1UpdateProduct() {        ②
        return getQuantity();
    }
    ...
}

```

① default the 0-th parameter using the current value of the `product` property

② default the 1-th parameter using the current value of the `quantity` property

Defaults are also supported (of course) for [contributed actions](#). For example, here is a contributed action for updating category/subcategory of the (non-ASF) [Isis addons' todoapp](#):

```

@DomainService(nature = NatureOfService.VIEW_CONTRIBUTIONS_ONLY)
public class UpdateCategoryContributions ... {
    @ActionLayout(
        describedAs = "Update category and subcategory"
    )
    @Action(semantic = SemanticsOf.IDEMPOTENT)
    public Categorized updateCategory(
        final Categorized item,           ①
        final Category category,
        @Parameter(optional = Optionality.OPTIONAL)
        final Subcategory subcategory) {
        item.setCategory(category);
        item.setSubcategory(subcategory);
        return item;
    }
    public Category default1UpdateCategory(
        final Categorized item) {           ②
        return item != null? item.getCategory(): null;
    }
    public Subcategory default2UpdateCategory(
        final Categorized item) {           ③
        return item != null? item.getSubcategory(): null;
    }
}

```

- ① `ToDoItem` implements `Categorized`
- ② defaults the 1-th parameter using the item's `category` property
- ③ defaults the 2-th parameter using the item's `subcategory` property

## Properties

For a property of type `T`, the signature is:

```
public T defaultXxx() { ... }
```

For example:

```

public class ShoppingCartItem {
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

    public int defaultProduct() {
        return 1;
    }
}

```

## Alternatives

There are, in fact, two other ways to set properties of a newly instantiated object to default values.

The first is to use the `created()` callback, called by the framework when `DomainObjectContainer#newTransientInstance(...)` is called. This method is called after any dependencies have been injected into the service.

The second is more straightforward: simply initialize properties in the constructor. However, this cannot use any injected services as they will not have been initialized.

### 2.1.6. `disable...`()

The `disable...`() supporting method is called for properties, collections and actions. It allows the modification of the property/collection to be vetoed (ie made read-only) and to prevent the invocation of the action ("grey it out").



Directly mutable collections are not currently supported by the [Wicket viewer](#); they are always implicitly disabled.

Typically modification/invocation is vetoed based on the state of the domain object being interacted with, though it could be any reason at all (eg the current date/time of the interaction, or the state of some other related data such as stock levels, or the identity of the calling user).

The reason for vetoing a modification/invocation is normally returned as a string. However, Apache Isis' [i18n support](#) extends this so that reasons can be internationalized.

## Actions

For an action the signature of the supporting method is:

```
public String disableXxx(...) { ... }
```

where the returned string is the reason the action invocation is vetoed (or `null` if not vetoed), and the supporting method takes the same parameter types as the action itself.

For example:

```

public class Customer {
    public boolean isBlacklisted() { ... }

    public Order placeOrder(
        final Product product,
        @ParameterLayout(named="Quantity")
        final int quantity) {
        ...
    }
    public String disablePlaceOrder(
        final Product product,
        final int quantity
    ) {
        return isBlacklisted()
            ? "Blacklisted customers cannot place orders"
            : null;
    }
    ...
}

```

## Properties and Collections

For both properties and collections the signature of the supporting method is:

```

public String disableXxx() { ... }

```

where the returned string is the reason the modification is vetoed (or `null` if not vetoed).

For example:

```

public class Customer {
    public boolean isBlacklisted() { ... }

    public BigDecimal getCreditLimit() { ... }
    public void setCreditLimit(BigDecimal creditLimit) { ... }
    public String disableCreditLimit() {
        return isBlacklisted()
            ? "Cannot change credit limit for blacklisted customers"
            : null;
    }
    ...
}

```

### 2.1.7. `get...`()

The `get...`() prefix is simply the normal JavaBean getter prefix that denotes properties or collections.



When Apache Isis builds its metamodel, it first searches for the getter methods, characterizing them as either properties or collections based on the return type. It then refines the metamodel based on the presence of annotations and supporting methods.

All remaining `public` methods (that do not use one of the Apache Isis prefixes) are interpreted as actions.

Any methods "left over" that *do* use one of the Apache Isis prefixes, are interpreted to be orphaned. Apache Isis "fails-fast" and will not boot, instead printing an error message to the log so that the issue can be easily diagnosed.

See also `set...()`.

### 2.1.8. `hide...()`

The `hide...()` supporting method is called for properties, collections and actions. It allows the property/collection to be completely hidden from view.

It's comparatively rare for properties or collections to be imperatively hidden from view, but actions are sometimes hidden or shown visible (as opposed to being just `disabled`, ie greyed out).

#### Actions

For an action the signature of the supporting method is either:

```
public bool hideXxx(...) { ... }
```

where the supporting method takes the same parameter types as the action itself, or more simply:

```
public bool hideXxx() { ... }
```

with no parameters at all. Returning `true` will hide the action, returning `false` leaves it visible.

For example:

```

public class Customer {
    public boolean isBlacklisted() { ... }

    public Order placeOrder(
        final Product product,
        @ParameterLayout(named="Quantity")
        final int quantity) {
        ...
    }
    public boolean hidePlaceOrder() {
        return isBlacklisted();
    }
    ...
}

```

## Properties and Collections

For both properties and collections the signature of the supporting method is:

```

public boolean hideXxx() { ... }

```

where returning `true` will hide the property/collection, returning `false` leaves it visible.

For example:

```

public class Customer {
    public boolean isBlacklisted() { ... }

    public BigDecimal getCreditLimit() { ... }
    public void setCreditLimit(BigDecimal creditLimit) { ... }
    public boolean hideCreditLimit() {
        return isBlacklisted();
    }
    ...
}

```

### 2.1.9. `modify...()`

The `modify...()` supporting method is called — instead of the setter — whenever a property has been set to be set to a new value. Its purpose is to allow additional business logic to be performed.

For example:

```

public class LibraryMember {
    public Title getFavoriteTitle() { ... }
    public void setFavoriteTitle(Title title) { ... }
    public void modifyFavoriteTitle(Title title) {
        if(getTitle() != null) {
            titleFavoritesService.decrement(getTitle()); ①
        }
        setFavoriteTitle(title); ②
        titleFavoritesService.decrement(title); ③
    }
    public void clearFavoriteTitle() { ... }
    ...
}

```

- ① perform some additional business logic
- ② update the property
- ③ perform some additional business logic

See also `clear…()``

### 2.1.10. `removeFrom…()`

The `removeFrom…()` supporting method is called whenever an object is removed from a collection. Its purpose is to allow additional business logic to be performed.



Directly mutable collections are not currently supported by the [Wicket viewer](#). The suggested workaround is to simply define an action.

For example:

```

public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public void addToBorrowed(Book book) { ... }
    public void removeFromBorrowed(Book book) {
        getBorrowed().remove(book); ①
        reminderService.removeReminder(this, book); ②
    }
    ...
}

```

- ① update the collection
- ② perform some additional business logic

See also `addTo…()``

### 2.1.11. `set…()`

The `set…()` prefix is simply the normal JavaBean setter prefix that denotes writeable properties or collections.

See also `get…()`.

### 2.1.12. `validate…()`

The `validate…()` supporting method is called for properties, actions and action parameters. It allows the proposed new value for a property to be rejected, or the proposed argument of an action parameter to be rejected, or to reject a whole set of action arguments for an action invocation.

The reason for vetoing a modification/invocation is normally returned as a string. However, Apache Isis' [i18n support](#) extends this so that reasons can be internationalized if required.

#### Action Parameter

For an action parameter in (0-based) position  $N$ , and of type  $T$ , the signature is:

```
public String validateNxxx(T proposed) { ... }
```

where the returned string is the reason why the argument is rejected (or `null` if not vetoed).

For example:

```
public class Customer {
    public Order placeOrder(
        final Product product,
        @ParameterLayout(named="Quantity")
        final int quantity) {
        ...
    }
    public String validate0PlaceOrder(
        final Product product) {
        return product.isDiscontinued()
            ? "Product has been discontinued"
            : null;
    }
    ...
}
```

#### Action Parameter Set

In addition to validating a single single action argument, it is also possible to validate a complete set of action arguments. The signature is:

```
public String validateXxx(...) { ... }
```

where the returned string is the reason why the argument is rejected (or `null` if not vetoed), and the supporting method takes the same parameter types as the action itself.

For example:

```
public class Customer {
    public Order placeOrder(
        final Product product,
        @ParameterLayout(named="Quantity")
        final int quantity) {
        ...
    }
    public String validatePlaceOrder(
        final Product product,
        final int quantity) {
        return quantity > product.getOrderLimit()
            ? "May not order more than " + product.getOrderLimit() + " items
for this product"
            : null;
    }
    ...
}
```

## Properties

For properties of type `T` the signature of the supporting method is:

```
public String validateXxx(T proposed) { ... }
```

where the returned string is the reason the modification is vetoed (or `null` if not vetoed).

For example:

```
public class Customer {
    public BigDecimal getCreditLimit() { ... }
    public void setCreditLimit(BigDecimal creditLimit) { ... }
    public validateCreditLimit(BigDecimal creditLimit) {
        return creditLimit.compareTo(BigDecimal.ZERO) < 0
            ? "Credit limit cannot be negative"
            : null;
    }
    ...
}
```

### 2.1.13. `validateAddTo…()`

The `validateAddTo…()` supporting method is called whenever an object is to be added to a collection. Its purpose is to validate the proposed object and possibly veto the change.



Directly mutable collections are not currently supported by the [Wicket viewer](#). The suggested workaround is to simply define an action.

The signature of the supporting method for a collection with element type `E` is:

```
public String validateAddToXxx(E element) { ... }
```

where the returned string is the reason the collection modification invocation is vetoed (or `null` if not vetoed). Apache Isis' [i18n support](#) extends this so that reasons can be internationalized if required.

For example:

```
public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public String validateAddToBorrowed(Book book) {
        return book.isReference()? "Reference books cannot be borrowed": null;
    }
    public void validateRemoveFromBorrowed(Book book) { ... }
    ...
}
```

See also `addTo…()` and `validateRemoveFrom…()`

### 2.1.14. `validateRemoveFrom…()`

The `validateRemoveFrom…()` supporting method is called whenever an object is to be removed from a collection. Its purpose is to validate the proposed object removal and possibly veto the change.



Directly mutable collections are not currently supported by the [Wicket viewer](#). The suggested workaround is to simply define an action.

The signature of the supporting method for a collection with element type `E` is:

```
public String validateRemoveFromXxx(E element) { ... }
```

where the returned string is the reason the collection modification invocation is vetoed (or `null` if not vetoed). Apache Isis' [i18n support](#) extends this so that reasons can be internationalized if required.

For example:

```
public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public String validateAddToBorrowed(Book book) { ... }
    public void validateRemoveFromBorrowed(Book book) {
        return !book.hasBeenReadBy(this)? "You didn't read this book yet": null;
    }
    ...
}
```

See also `removeFrom...`() and `validateAddTo...`()`

## 2.2. Reserved Methods

The table below lists the reserved methods that are recognized as part of Apache Isis' default programming model.

Table 2. Reserved Methods

Method	Description
<code>cssClass()</code>	Provides a CSS class for this object instance. In conjunction with <code>application.css</code> , can therefore provide custom styling of an object instance wherever it is rendered. See also <code>title()</code> and <code>iconName()</code> .
<code>disable(...)</code>	Disable all or some of an object's properties
<code>getId()</code>	Provides an optional unique identifier of a service. If not provided, the service's fully-qualified class name is used.
<code>hide(...)</code>	Hide all or some of an object's properties
<code>iconName()</code>	Provides the name of the image to render, usually alongside the title, to represent the object. If not provided, then the class name is used to locate an image. See also <code>title()</code> and <code>cssClass()</code>
<code>title()</code>	Provides a title for the object. See also <code>iconName()</code> and <code>cssClass()</code>
<code>validate()</code>	Validate the object's state prior to persisting.

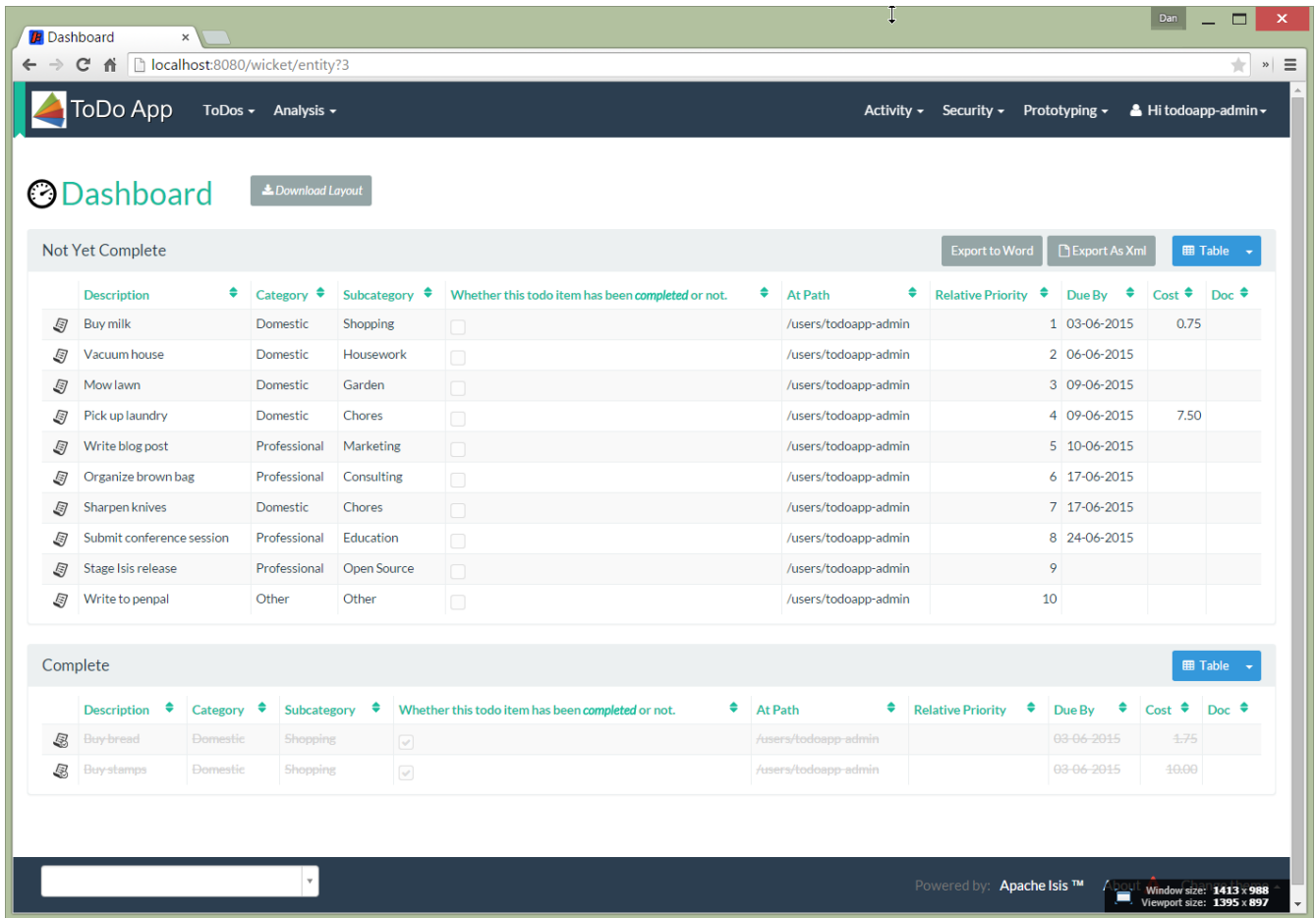
### 2.2.1. `cssClass()`

The `cssClass()` returns a CSS class for a particular object instance.

The [Wicket viewer](#) wraps the object's representation in a containing `<div>` with the class added. This is done both for rendering the object either in a table or when rendering the object on its own page.

In conjunction with `application.css`, can therefore provide custom styling of an object instance wherever it is rendered.

For example, the (non-ASF) [Isis addons' todoapp](#) uses this technique to add a strikethrough for completed todo items. This is shown on the home page:



The code to accomplish this is straightforward:

```
public class ToDoItem ... {
    public String cssClass() {
        return !isComplete() ? "todo" : "done";
    }
    ...
}
```

In the `application.css`, the following styles were then added:

```
tr.todo {
}
tr.done {
    text-decoration: line-through;
    color: #d3d3d3;
}
```



See also `title()` and `iconName()`.

### 2.2.2. `disable()`

One use case that Apache Isis supports is that of a domain object with a lifecycle whereby at some stage it should become immutable: all its properties/collections should be disabled, and/or its actions become not invocable.

It would be painful to have to write a separate `disable…()` method for each and every member, so instead Isis allows a single `disable…(…)` method to be implemented that is applied to all members.

The signature of the method is:

```
public String disable(Identifier.Type identifierType) { ... }
```

where `Identifier.Type` is part of the Isis applib (nested static class of `o.a.i.applib.Identifier`) to distinguish between an interaction with an action, a property or an action.

Note that Apache Isis' [i18n support](#) extends this so that the returned reason can also be internationalized.

For example:

```
public String disable(Identifier.Type identifierType) {
    return !calendarService.isOfficeHours(clock.today())
        ? "Cannot modify objects outside of office hours"
        : null;
}
```

See also the similar methods to `hide()` object members en-masse.

### Alternatives

An alternative design—and one that could be easily argued is actually more flexible—is to leverage domain events with vetoing subscribers.

With this approach we define, for a given domain class, a base `PropertyDomainEvent`, `CollectionDomainEvent` and `ActionDomainEvent`. A good pattern is to make these nested static classes. For example:

```
public class ToDoItem ... {
    public static abstract class PropertyDomainEvent<T>
        extends ToDoAppDomainModule.PropertyDomainEvent<ToDoItem, T> {
        ...
    }
    ...
}
```

where in turn:

```
public final class ToDoAppDomainModule {
    private ToDoAppDomainModule(){}
    public abstract static class PropertyDomainEvent<S,T>
        extends org.apache.isis.applib.services.eventbus.PropertyDomainEvent<S,T>
    {
        ...
    }
    ...
}
```

Then, each property/collection/action emits either these base domain events or their own subclass:

```
public class ToDoItem ... {
    public static class DescriptionDomainEvent
        extends PropertyDomainEvent<String> {
        ...
    }
    @Property(
        domainEvent = DescriptionDomainEvent.class
    )
    public String getDescription() { ... }
    ...
}
```

A vetoing subscriber can then subscribe to the domain events and veto access, eg:

```
@DomainObject
public class VetoOutOfOfficeHours {
    @Subscribe
    public void on(ToDoItem.PropertyDomainEvent ev) {
        if(!calendarService.isOfficeHours(clock.today()) {
            ev.veto("Cannot modify objects outside of office hours");
        }
    }
    ...
}
```

Obviously there's an awful lot more boilerplate here, but there's also a lot more flexibility.

### 2.2.3. getId()

The `getId()` method applies only to domain services, and allows a unique identifier to be provided for that service.

This identifier corresponds in many ways to the `objectType()` attribute for domain objects; it is used

as an internal identifier but also appears in URLs within the [RestfulObjects viewer](#)'s REST API.

If the identifier is omitted, the services fully qualified class name is used.



Unlike domain objects, where the use of an object type is strongly encouraged (eg using `@PersistenceCapable`), it matters much less if an id is specified for domain services. The principle benefit is shorter URLs in the REST API.

#### 2.2.4. `hide()`

One use case that Apache Isis supports is that of a domain object with a lifecycle whereby at some stage some number of the object's members should be hidden. For example, for an object that at some stage is logically immutable, we might want to make all its properties/collections unmodifiable and hide all its actions.

While we could write a separate `hide...()` method for each and every action, this could become painful. So instead Isis allows a single `hide...(...)` method to be implemented that is applied to all members.

The signature of the method is:

```
public boolean hide(Identifier.Type identifierType) { ... }
```

where `Identifier.Type` is part of the Isis applib (nested static class of `o.a.i.applib.Identifier`) to distinguish between an interaction with an action, a property or an action.

For example:

```
public boolean hide(Identifier.Type identifierType) {  
    return identifierType == Identifier.Type.ACTION && isFrozen();  
}
```

See also the similar method to `disable()` object members en-masse.

#### Alternatives

An alternative design—and one that could be easily argued is actually more flexible—is to leverage domain events with vetoing subscribers.

There is further discussion on this approach in [here](#).

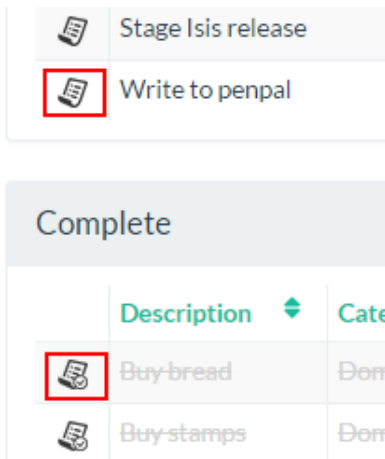
#### 2.2.5. `iconName()`

Every object is represented by an icon; this is based on the domain object's simple name. The [Wicket viewer](#) searches for the image in the same package as the `.class` file for the domain object.

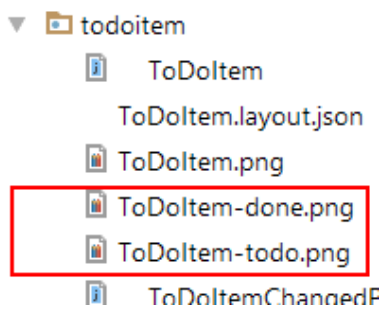
The `iconName()` allows the icon that to be used to change for individual object instances. These are usually quite subtle, for example to reflect the particular status of an object. The value returned by

the `iconName()` method is added as a suffix to the base icon name.

For example, the (non-ASF) [Isis addons' todoapp](#) uses this technique to add an overlay for todo items that have been completed:



The screenshot below shows the location of these png icon files:



The code to accomplish this is straightforward:

```
public class ToDoItem ... {
    public String iconName() {
        return !isComplete() ? "todo" : "done";
    }
    ...
}
```

See also `title()` and `cssClass()`

## 2.2.6. `title()`

Every object is represented by a title. This appears both as a main header for the object when viewed as well as being used as a hyperlink within properties and collections. It therefore must contain enough information for the end-user to distinguish the object from any others.

This is most commonly done by including some unique key within the title, for example a customer's SSN, or an order number, and so forth. However note that Apache Isis itself does *not* require the title to be unique; it is merely recommended in most cases.

An object's title can be constructed in various ways, but the most flexible is to use the `title()`

method. The signature of this method is usually:

```
public String title() { ... }
```

Note that Apache Isis' [i18n support](#) extends this so that titles can also be internationalized.

For example, the (non-ASF) [Isis addons' todoapp](#) uses this technique to add an overlay for todo items that have been completed:

```
public String title() {  
    final TitleBuffer buf = new TitleBuffer();           ①  
    buf.append(getDescription());  
    if (isComplete()) {                                 ②  
        buf.append("- Completed!");  
    } else {  
        try {  
            final LocalDate dueBy = wrapperFactory.wrap(this).getDueBy();  ③  
            if (dueBy != null) {  
                buf.append(" due by", dueBy);  
            }  
        } catch (final HiddenException ignored) {      ④  
        }  
    }  
    return buf.toString();  
}
```

- ① simple [utility class](#) to help construct the title string
- ② imperative conditional logic
- ③ using the [WrapperFactory](#) to determine if the `dueBy` field is visible for this user ...
- ④ ... but ignore if not

As the example above shows, the implementation can be as complex as you like.

In many cases, though, you may be able to use the `@Title` annotation.

See also `iconName()` and `cssClass()`

### 2.2.7. `validate()`

The `validate()` method is used to specify that invariants pertaining to an object's state are enforced.



(As of 1.8.0) there are known limitations with this functionality. Invariants are enforced when an object is initially created and when it is edited, however invariants are currently *not* enforced if an action is invoked.

The signature of the method is:

```
public String validate() { ... }
```

where the returned string is the reason that the invocation is vetoed.

Note that Apache Isis' [i18n support](#) extends this so that the returned reason can also be internationalized.

## 2.3. Lifecycle Methods

The lifecycle callback methods notify a domain entity about its interaction within the persistence lifecycle. For example, the entity is notified immediately prior to being persisted, or when it is about to be updated.



Note that these callbacks are fired by Apache Isis rather than JDO. In the future we may deprecate them because there are better mechanisms available using listeners/subscribers:

- in Isis 1.9.0 and earlier, you may therefore want to consider using the JDO API directly to set up a lifecycle listener; see [here](#) for further discussion.
- alternatively, you can use a subscriber for the [lifecycle events](#) fired in Isis.

The lifecycle callback methods supported by Isis are:

*Table 3. Lifecycle methods (partial support)*

Method	Description
<code>created()</code>	called when an object has just been created using <code>newTransientInstance()</code>
<code>loaded()</code>	called when a (persistent) object has just been loaded from the object store.
<code>persisted()</code>	called when object has just been persisted from the object store.
<code>persisting()</code>	called when a (not-yet-persistent) object is just about to be persisted from the object store
<code>removed()</code>	called when a (persistent) object has just been deleted from the object store
<code>removing()</code>	called when a (persistent) object is just about to be deleted from the object store
<code>updated()</code>	called when a (persistent) object has just been updated in the object store
<code>updating()</code>	called when a (persistent) object is just about to be updated in the object store

Some lifecycle methods have been deprecated:

*Table 4. Deprecated lifecycle methods*

Method	Notes
<code>deleted()</code>	Replaced by <code>removed()</code>

Method	Notes
<code>deleting()</code>	Replaced by <code>removing()</code>
<code>loading()</code>	callback for when the (persistent) object is just about to be loaded from the object store. [WARNING] ==== This method is never called. ====
<code>saved()</code>	Replaced by <code>persisted()</code>
<code>saving()</code>	Replaced by <code>persisting()</code>

### 2.3.1. `created()`

The `created()` lifecycle callback method is called when an object has just been created using `newTransientInstance()`



Alternatively, consider using a [event bus subscriber](#) on the `ObjectCreatedEvent`.

### 2.3.2. `loaded()`

The `loaded()` lifecycle callback method is called when a (persistent) object has just been loaded from the object store.



Alternatively, consider using a [event bus subscriber](#) on the `ObjectLoadedEvent`.

### 2.3.3. `persisted()`

The `persisted()` lifecycle callback method is called when object has just been persisted from the object store.

See also `persisting()`.



Alternatively, consider using a [event bus subscriber](#) on the `ObjectPersistedEvent`.

### 2.3.4. `persisting()`

The `persisting()` lifecycle callback method is called when a (not-yet-persistent) object is just about to be persisted from the object store

See also `persisted()`.



Alternatively, consider using a [event bus subscriber](#) on the `ObjectPersistingEvent`.

### 2.3.5. `removed()`

The `removed()` lifecycle callback method is called when a (persistent) object has just been deleted from the object store

See also `removing()`.



Alternatively, consider using a [event bus subscriber](#) on the `ObjectRemovedEvent`.

### 2.3.6. `removing()`

The `removing()` lifecycle callback method is called when a (persistent) object is just about to be deleted from the object store

See also `removed()`.



Alternatively, consider using a [event bus subscriber](#) on the `ObjectRemovingEvent`.

### 2.3.7. `updated()`

The `updated()` lifecycle callback method is called when a (persistent) object has just been updated in the object store

See also `updating()`.



Alternatively, consider using a [event bus subscriber](#) on the `ObjectUpdatedEvent`.

### 2.3.8. `updating()`

The `updating()` lifecycle callback method is called when a (persistent) object is just about to be updated in the object store

See also `updated()`.



Alternatively, consider using a [event bus subscriber](#) on the `ObjectUpdatingEvent`.

### 2.3.9. Using the JDO API

As an alternative to relying on Apache Isis to call lifecycle callback methods, you could instead use the JDO [lifecycle listener](#) API directly.



We may decide to deprecate the Apache Isis callbacks in the future because they merely duplicate this functionality already available in JDO.

You can gain access to the relevant JDO API using the `IsisJdoSupport` domain service.

For example:



```

@RequestScoped ①
@DomainService(nature=NatureOfService.DOMAIN)
public class ObjectChangedListenerService
    implements javax.jdo.listener.StoreLifecycleListener { ②
    @Programmatic
    @PostConstruct
    public void init() {
        getPmFactory().addInstanceLifecycleListener(this);
    }
    @Programmatic
    @PreDestroy
    public void tidyUp() {
        getPmFactory().removeInstanceLifecycleListener(this);
    }
    private PersistenceManager getPersistenceManager() {
        return jdoSupport.getPersistenceManager(); ③
    }
    @Programmatic
    public void preStore (InstanceLifecycleEvent event) { ... }
    @Programmatic
    public void postStore (InstanceLifecycleEvent event) { ... }
    @Inject
    IsisJdoSupport jdoSupport;
}

```

- ① must be `@RequestScoped` because we register on the `PersistenceManager`, which is different for each request.
- ② implement whichever callback lifecycle listeners are of interest
- ③ use the injected `IsisJdoSupport` service to obtain the `PersistenceManager`.

Note that it isn't possible to register on the `PersistenceManagerFactory` because listeners cannot be attached once a persistence session has been created (which it will have been when the service's `@PostConstruct` method is called).

# Chapter 3. Classes and Interfaces

This chapter describes the usage of various classes and interfaces that are not otherwise associated with [domain services](#), [object layout](#) or [configuration](#).

## 3.1. AppManifest (bootstrapping)

This section describes how to implement the `AppManifest` interface to bootstrap both an Apache Isis web application, and also its integration tests.

### 3.1.1. API

The `AppManifest` interface allows the constituent parts of an application to be defined programmatically, most specifically the packages that contain domain services and/or persistent entities. Its API is defined as:

```
public interface AppManifest {
    public List<Class<?>> getModules();           ①
    public List<Class<?>> getAdditionalServices(); ②
    public String getAuthenticationMechanism();  ③
    public String getAuthorizationMechanism();  ④
    public List<Class<? extends FixtureScript>> getFixtures(); ⑤
    public Map<String,String> getConfigurationProperties(); ⑥
}
```

- ① Must return a non-null list of classes, each of which representing the root of one of the modules containing services and possibly entities, which together makes up the running application.
- ② If non-null, overrides the value of `isis.services` configuration property to specify a list of additional classes to be instantiated as domain services (over and above the domain services defined via `getModules()` method).
- ③ If non-null, overrides the value of `isis.authentication` configuration property to specify the authentication mechanism.
- ④ If non-null, overrides the value of `isis.authorization` configuration property to specify the authorization mechanism.
- ⑤ If non-null, overrides the value of `isis.fixtures` configuration property to specify a fixture script to be installed.
- ⑥ Overrides for any other configuration properties.

The following sections describe each of these methods in a little more detail.

#### `getModules()`

The most significant method (the only one which must return a non-null value) is the `getModules()` method. Each module is identified by a class; the framework simply uses that class' package as the root to search for domain services (annotated with `@DomainService`) and entities (annotated with `@PersistenceCapable`). Generally there is one such module class per Maven module.

A module class for a domain module might for example be defined as:

```
package com.mycompany.myapp.dom;
public final class MyAppDomainModule {
    private MyAppDomainModule() {}
}
```

This tells the framework that the package and subpackages under `com.mycompany.myapp.dom` should be searched for domain services and entities.

As is perhaps apparent, the `getModules()` method replaces and overrides both the `isis.services.ServicesInstallerFromAnnotation.packagePrefix` key (usually found in the `isis.properties` file) and also the `isis.persistor.datanucleus.RegisterEntities.packagePrefix`` key (usually found in the `persistor_datanucleus.properties` file). The value of the `isis.services-installer` configuration property is also ignored.

For example, the (non-ASF) [Isis addons' todoapp](#) defines the following:

```
@Override
public List<Class<?>> getModules() {
    return Arrays.asList(
        ToDoAppDomainModule.class,
        ToDoAppFixtureModule.class,
        ToDoAppAppModule.class,
        org.isisaddons.module.audit.AuditModule.class,
        org.isisaddons.module.command.CommandModule.class,
        org.isisaddons.module.devutils.DevUtilsModule.class,
        org.isisaddons.module.docx.DocxModule.class,
        org.isisaddons.module.publishing.PublishingModule.class,
        org.isisaddons.module.sessionlogger.SessionLoggerModule.class,
        org.isisaddons.module.settings.SettingsModule.class,
        org.isisaddons.wicket.gmap3.cpt.service.Gmap3ServiceModule.class
    );
}
```

As can be seen, the various (non-ASF) [Isis Addons](#) modules also each provide a module class that can be easily referenced.

### `getAdditionalServices()`

We normally we recommend that services are defined exclusively through `getModules()`, and that this method should therefore return an empty list. However, there are certain use cases where the a service must be explicitly specified either because the service required does not (for whatever reason) have a `@DomainService` annotation.

For example, the (non-ASF) [Isis addons' security](#) module (v1.9.0) allows the policy to evaluate conflicting permissions to be specified by explicitly registering either the `PermissionsEvaluationServiceAllowBeatsVeto` domain service or the

`PermissionsEvaluationServiceVetoBeatsAllow` domain service:

```
@Override
public List<Class<?>> getAdditionalServices() {
    return Arrays.asList(
        org.isisaddons.module.security.dom.permission
        .PermissionsEvaluationServiceVetoBeatsAllow.class
    );
}
```

If this method returns a non-`null` value, then it overrides the value of `isis.services` configuration property.

#### `getAuthenticationMechanism()`

If non-`null`, this method specifies the authentication mechanism to use. The valid values are currently `"shiro"` or `"bypass"`. If `null` is returned then the value of the `isis.authentication` configuration property (in `isis.properties` file) is used instead.

See the [security guide](#) for further details on configuring shiro or bypass security.



This property is ignored for integration tests (which always uses the `"bypass"` mechanism).

#### `getAuthorizationMechanism()`

If non-`null`, this method specifies the authorization mechanism to use. The valid values are currently `"shiro"` or `"bypass"`. If `null` is returned then the value of the `isis.authorization` configuration property (in `isis.properties` file) is used instead.

See the [security guide](#) for further details on configuring shiro or bypass security.



This property is ignored for integration tests (which always uses the `"bypass"` mechanism).

#### `getFixtures()`

If non-`null`, this method specifies the fixture script(s) to be run on startup. This is particularly useful when developing or demoing while using an in-memory database.

For example:

```
@Override
public List<Class<? extends FixtureScript>> getFixtures() {
    return Lists.newArrayList(todoapp.fixture.demo.DemoFixture.class);
}
```

Note that in order for fixtures to be installed it is also necessary to set the

`isis.persistor.datanucleus.install-fixtures` key to `true`. This can most easily be done using the `getConfigurationProperties()` method, discussed below.

`getConfigurationProperties()`

This method allow arbitrary other configuration properties to be overridden. One common use case is in conjunction with the `getFixtures()` method, discussed above:

```
@Override
public Map<String, String> getConfigurationProperties() {
    Map<String, String> props = Maps.newHashMap();
    props.put("isis.persistor.datanucleus.install-fixtures", "true");
    return props;
}
```

### 3.1.2. Bootstrapping

One of the primary goals of the `AppManifest` is to unify the bootstrapping of both integration tests and the webapp. This requires that the integration tests and webapp can both reference the implementation.

We strongly recommend using a `myapp-app` Maven module to hold the implementation of the `AppManifest`. This Maven module can then also hold dependencies which are common to both integration tests and the webapp, specifically the `org.apache.isis.core:isis-core-runtime` and the `org.apache.isis.core:isis-core-wrapper` modules.

We also strongly recommend that any application-layer domain services and view models (code that references persistent domain entities but that is not referenced back) is moved to this `myapp-app` module. This will allow the architectural layering of the overall application to be enforced by Maven.

What then remains is to update the bootstrapping code itself.

#### Integration Tests

Bootstrapping integration tests is still performed using the `IsisSystemForTest.Builder`, but very often the bootstrapping code can be substantially simplified (compared to not using an `AppManifest`, that is).

For example, this is the bootstrapping code for the `SimpleApp archetype` looks something like:

```

public class DomainAppSystemInitializer {
    public static void initIsft() {
        IsisSystemForTest isft = IsisSystemForTest.getElseNull();
        if(isft == null) {
            isft = new IsisSystemForTest.Builder()
                .withLoggingAt(org.apache.log4j.Level.INFO)
                .with(new MyAppAppManifest())
                .with(new IsisConfigurationForJdoIntegTests())
                .build()
                .setUpSystem();
            IsisSystemForTest.set(isft);
        }
    }
}

```

## Webapps

To bootstrap an Apache Isis webapp (using the [Wicket viewer](#)), there are two choices:

- either specify the `AppManifest` by overriding the `IsisWicketApplication#newWicketModule()`, eg:

```

@Override
protected Module newIsisWicketModule() {
    final Module isisDefaults = super.newIsisWicketModule();
    ...
    final Module overrides = new AbstractModule() {
        @Override
        protected void configure() {
            ...
            bind(AppManifest.class).toInstance(new MyAppAppManifest());
        }
    };
    return Modules.override(isisDefaults).with(overrides);
}

```

- alternatively update `isis.properties`, using the `isis.appManifest` key to specify the `AppManifest` implementation, eg:

```

isis.appManifest=domainapp.app.MyAppAppManifest

```

The first (programmatic) approach takes precedence over the second approach (configuration properties).



If you use the `org.apache.isis.WebServer` class to launch your application from the **command line**, then note that you can specify the `AppManifest` using the `-m` (or `--manifest`) flag:

```
java org.apache.isis.WebServer -m
com.mycompany.myapp.MyAppAppManifestWithFixtures
```

### 3.1.3. Subsidiary Goals

There are a number of subsidiary goals of the `AppManifest` class (though as of v1.9.0 these have not yet implemented):

- Allow different integration tests to run with different manifests.
  - Normally the running application is shared (on a thread-local) between integration tests. What the framework could do is to be intelligent enough to keep track of the manifest in use for each integration test and tear down the shared state if the "next" test uses a different manifest
- Speed up bootstrapping by only scanning for classes annotated by `@DomainService` and `@PersistenceCapable` once.
- Provide a programmatic way to contribute elements of `web.xml`.
- Provide a programmatic way to configure Shiro security.
- Anticipate the module changes forthcoming in Java 9.
  - Eventually we see that the `AppManifest` class acting as an "aggregator", with the list of modules will become Java 9 modules each advertising the types that they export.
  - It might even be possible for `AppManifests` to be switched on and off dynamically (eg if Java9 is compatible with OSGi, being one of the design goals).

## 3.2. Superclasses

This section catalogues the various convenience (non event) superclasses defined by Apache Isis. These are listed in the table below.

Table 5. Convenience Superclasses

API	Maven Module Impl'n (g: a:)	Implemen tation	Notes
<code>o.a.i.applib. AbstractContainedObject</code>	<code>o.a.i.core isis-core-applib</code>	(abstract class)	
<code>o.a.i.applib. AbstractDomainObject</code>	<code>o.a.i.core isis-core-applib</code>	(abstract class)	

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. AbstractFactoryAndRepository	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. AbstractService	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. AbstractSubscriber	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. AbstractViewModel	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. fixturescript FixtureScript	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. fixturescripts FixtureScripts	o.a.i.core isis-core-applib	(abstract class). FixtureScriptsDefault is a default implementation that is used when the alternative FixtureScriptsSpecificationProvider is provided (and no other implementation of FixtureScripts was found).	depends on: ClassDiscoveryService

### 3.2.1. AbstractContainedObject

This class is a convenience superclass for domain objects and services, providing general purpose methods for interacting with the framework. These include:

- `allMatches(Query)` - search for all objects matching the specified `Query`.
  - + Note that this, and other similar methods (eg `firstMatch(...)`, `uniqueMatch(...)`) will automatically flush the current transaction.
- `newTransientInstance(Class)` - to create a new instance of an object, with any services injected



into it

- `persistIfNotAlready(Object)` - to persist an object

In fact, the object is queued up to be persisted, and is only actually persisted either when the transaction commits, or when the transaction is flushed (typically when a query is performed).

- `warnUser(String)` - generate a warning to the user
- `getContainer()` - which returns the `DomainObjectContainer`

Each of these methods simply delegates to an equivalent method in `DomainObjectContainer`.



In practice we find that there's little to gain from subclassing; it's easier/less obscure to simply inject `DomainObjectContainer` into a simple pojo class.

### 3.2.2. AbstractDomainObject

This class extends `AbstractContainedObject`, adding in convenience methods for managing the persistence lifecycle of the object instance.

Each of these methods, eg `isPersistent(...)`, delegates to an equivalent method in `DomainObjectContainer`.

### 3.2.3. AbstractFactoryAndRepository

This class extends `AbstractContainedObject`. Its intent was to be a convenience subclass for services acting as either a repository or a factory, however note that all of the methods that it defines are now deprecated.

Instead, indicate that a service is repository using the `@DomainService#repositoryFor()` attribute.

### 3.2.4. AbstractService

This class extends `AbstractContainedObject`, adding in an implementation of `getId()` based upon the classes name.

In practice there is little to gain from subclassing; simply inject `DomainObjectContainer` for broadly equivalent functionality.

### 3.2.5. AbstractSubscriber

This is a convenience superclass for creating subscriber domain services on the `EventBusService`. It uses `@PostConstruct` and `@PreDestroy` callbacks to automatically register/unregister itself with the `EventBusService`.

It's important that subscribers register before any domain services that might emit events on the `EventBusService`. For example, the (non-ASF) `Isis addons' security` module provides a domain service that automatically seeds certain domain entities; these will generate `lifecycle events` and so any subscribers must be registered before such seed services. The easiest way to do this is to use the `@DomainServiceLayout#menuOrder()` attribute.

As a convenience, the `AbstractSubscriber` specifies this attribute.

### 3.2.6. `AbstractViewModel`

This class extends `AbstractContainedObject`, also implementing the `ViewModel` interface. In and of itself it provides no new behaviour.



As an alternative, consider simply annotating the view model class with `{@link org.apache.isis.applib.annotation.ViewModel}`.

### 3.2.7. `FixtureScript`

The `FixtureScript` class is an abstract class defining an API to set up data within the object store, either for integration tests or while demoing/prototyping.

The primary method that subclasses must implement is:

```
protected abstract void execute(final ExecutionContext executionContext);
```

In this method the fixture script can in theory do anything, but in practice it is recommended that it uses injected domain services to set up data. The provided `ExecutionContext` is used to invoke child fixture scripts, and also can be used to store references to any created objects (so that the calling test can access these objects/so that they are rendered in the view model).

See the see the [user guide's testing chapter](#) for further discussion on the use of fixture scripts, in particular [fixture scripts' API and usage](#).

### 3.2.8. `FixtureScripts`

This abstract class is intended to allow a domain service that can execute `FixtureScripts` to be easily written.

However, it has now been deprecated; instead we recommend that the `FixtureScriptsSpecificationProvider` service is implemented instead. The framework will then automatically use `FixtureScriptsDefault` as a fallback implementation of this class.

See the see the [user guide's testing chapter](#) for further discussion on the use of fixture scripts, in particular [fixture scripts' API and usage](#).

## 3.3. Domain Event Classes

This section catalogues the various domain event classes defined by Apache Isis.

These events are broadcast on the `EventBusService`. The domain events are broadcast as a result of being specified in the `@Action#domainEvent()`, `@Property#domainEvent()` or `@Collection#domainEvent()` attributes.

They are listed in the table below.

Table 6. Domain Event Classes

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. AbstractDomainEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class)	Superclass of the other domain events, listed below in this table.
o.a.i.applib. ActionDomainEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ActionDomainEvent.Default is the concrete implementation used if no @Action#domainEvent attribute is specified	Broadcast whenever there is an interaction (hide/disable/validate/pre-execute/post-execute) with an object's action.
o.a.i.applib. CollectionDomainEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). CollectionDomainEvent.Default is the concrete implementation used if no @Collection#domainEvent attribute is specified.	Broadcast whenever there is an interaction (hide/disable/validate/access) with an object's collection.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. PropertyDomainEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). PropertyDomainEvent. Default is the concrete implementation used if no @PropertyDomainEvent attribute is specified	Broadcast whenever there is an interaction (hide/disable/validate/access) with an object's property.

### 3.3.1. AbstractDomainEvent

This class is the superclass for all domain events that are raised by the framework when interacting with actions, properties or collections.

Its immediate subclasses are:

- ActionDomainEvent
- PropertyDomainEvent
- CollectionDomainEvent

This class defines the `Phase` enum that is common to all domain events (hide/disable/validate/pre-execute/post-execute), and provides methods such as `veto(...)` by which subscribers can influence the interaction (eg hide a collection, disable a property, validate action arguments).

It class also provides `get(...)` and `put(...)` methods that allow adhoc sharing of user data between different phases. One event instance is used for both the hide and disable phases, and a different event instance is shared between validate/pre-execute/post-execute.

### 3.3.2. ActionDomainEvent

Subclass of `AbstractDomainEvent` for actions.

The class has a number of responsibilities (in addition to those it inherits):

- capture the target object being interacted with
- capture the arguments for each of the action's parameters
- provide selected metadata about the action parameters from the metamodel (names, types)
- link back to the `CommandContext` service's `Command` object

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.

### 3.3.3. `CollectionDomainEvent`

Subclass of `AbstractDomainEvent` for collections.

The class has a couple of responsibilities (in addition to those it inherits):

- capture the target object being interacted with
- indicate whether the interaction is to add or remove an object from the collection (or simply to indicate that the collection is being accessed/read)
- capture the object reference being added or removed

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's collection.

### 3.3.4. `PropertyDomainEvent`

Subclass of `AbstractDomainEvent` for properties.

The class has a couple of responsibilities (in addition to those it inherits):

- capture the target object being interacted with
- capture the old and new values of the property

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's property.

## 3.4. UI Event Classes

This section catalogues the various UI event classes defined by Apache Isis.

These events are broadcast on the `EventBusService`. The domain events are broadcast as a result of being specified in the `@DomainObjectLayout#titleUiEvent()`, `@DomainObjectLayout#iconUiEvent()` or `@DomainObjectLayout#cssClassUiEvent()` attributes.

They are listed in the table below.

*Table 7. UI Event Classes*

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. TitleUiEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). TitleUiEvent.Default is the concrete implementation used if no @DomainObjectLayout#titleUiEvent attribute is specified	Broadcast whenever there is a requirement to obtain a title for a domain object. Note that if the domain object defines its own title() supporting method, or has @Title annotation (s) on its properties, then these will take precedence.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. IconUiEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). IconUiEvent.Default is the concrete implementation used if no @DomainObjectLayout#iconUiEvent attribute is specified	Broadcast whenever there is a requirement to obtain an icon (or rather, the name of an icon) for a domain object. Note that if the domain object defines its own iconName() supporting method, or if it has the @DomainObjectLayout#cssClassFactory() attribute, then these will take precedence.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. CssClassUiEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). CssClassUiEvent.Default is the concrete implementation used if no @DomainObjectLayout#cssClassUiEvent attribute is specified	Broadcast whenever there is a requirement to obtain a CSS class hint for a domain object. Note that if the domain object defines its own cssClass() supporting method then this will take precedence.

### 3.4.1. TitleUiEvent

This event class represents a request to obtain the title of a domain object. The class has a number of responsibilities:

- capture the target object being interacted with
- capture the title, if any, as specified to one of the subscribers

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.



If the domain object defines its own `title()` supporting method, or has `@Title` annotation(s) on its properties, then these will take precedence.

### 3.4.2. IconUiEvent

This event class represents a request to obtain the icon (or rather, name of icon) of a domain object. The class has a number of responsibilities:

- capture the target object being interacted with
- capture the icon (name), if any, as specified to one of the subscribers

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.





If the domain object defines its own `iconName()` supporting method, or if it has the `@DomainObjectLayout#cssClassFa()` attribute, then these will take precedence.

### 3.4.3. `CssClassUiEvent`

This event class represents a request to obtain the a CSS class hint of a domain object. The class has a number of responsibilities:

- capture the target object being interacted with
- capture the CSS class, if any, as specified to one of the subscribers

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.



if the domain object defines its own `cssClass()` supporting method then this will take precedence.

## 3.5. Lifecycle Events

This section catalogues the various lifecycle event classes defined by Apache Isis. These events are fired automatically when a domain object is loaded, created, updated and so forth.

The lifecycle event classes are listed in the table below:

Table 8. Lifecycle Event Classes

API	Maven Module Impl'n (g: a:)	Implementation	Notes
<code>o.a.i.applib. AbstractLifecycleEvent</code>	<code>o.a.i.core services.eventbus isis-core-applib</code>	(abstract class)	Superclass of the other lifecycle events, listed below in this table.
<code>o.a.i.applib. ObjectCreatedEvent</code>	<code>o.a.i.core services.eventbus isis-core-applib</code>	(abstract class). <code>ObjectCreatedEvent.Default</code> is the concrete implementation that is used.	Broadcast when an object is first instantiated using the <code>DomainObjectContainer's #newTransientInstance(...)</code> method.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. ObjectLoadedEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectLoadedEvent.Default is the concrete implementation that is used.	Broadcast when an object is retrieved from the database.
o.a.i.applib. ObjectPersistedEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectPersistedEvent.Default is the concrete implementation that is used.	Broadcast when an object is first saved (inserted) into the database using the DomainObjectContainer's #persist(...) method.
o.a.i.applib. ObjectPersistingEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectPersistingEvent.Default is the concrete implementation that is used.	Broadcast when an object is about to be saved (inserted) into the database using the DomainObjectContainer's #persist(...) method.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. ObjectRemovingEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectRemovingEvent. Default is the concrete implementation that is used.	Broadcast when an object is about to be deleted from the database using the DomainObjectContainer's #remove(...) method.
o.a.i.applib. ObjectUpdatedEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectUpdatedEvent. Default is the concrete implementation that is used.	Broadcast when an object has just been updated in the database. This is done either explicitly when the current transaction is flushed using the DomainObjectContainer's #flush(...) method, else is done implicitly when the transaction commits at the end of the user request.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. ObjectUpdatingEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectUpdatingEvent. Default is the concrete implementation that is used.	Broadcast when an object is about to be updated in the database. This is done either explicitly when the current transaction is flushed using the DomainObjectContainer's #flush(...) method, else is done implicitly when the transaction commits at the end of the user request.

### 3.5.1. AbstractLifecycleEvent

This class is the superclass for all lifecycle events that are raised by the framework when loading, saving, updating or deleting objects from the database.

Its immediate subclasses are:

- ObjectCreatedEvent
- ObjectLoadedEvent
- ObjectPersistedEvent
- ObjectPersistingEvent
- ObjectRemovingEvent
- ObjectUpdatedEvent
- ObjectUpdatingEvent

### 3.5.2. ObjectCreatedEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is first instantiated using the `DomainObjectContainer`'s `#newTransientInstance(...)` method.

`ObjectCreatedEvent.Default` is the concrete implementation that is used.



In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

### 3.5.3. ObjectLoadedEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is retrieved from the database.

`ObjectLoadedEvent.Default` is the concrete implementation that is used.



In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

### 3.5.4. ObjectPersistedEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is first saved (inserted) into the database using the `DomainObjectContainer`'s `#persist(...)` method.

`ObjectPersistedEvent.Default` is the concrete implementation that is used.



In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

### 3.5.5. ObjectPersistingEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is about to be saved (inserted) into the database using the `DomainObjectContainer`'s `#persist(...)` method.

`ObjectPersistingEvent.Default` is the concrete implementation that is used.



In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

### 3.5.6. ObjectRemovingEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is about to be deleted from the database using the `DomainObjectContainer`'s `#remove(...)` method.

`ObjectRemovingEvent.Default` is the concrete implementation that is used.



In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

### 3.5.7. ObjectUpdatedEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object has just been updated in the database. This is done either explicitly when the current transaction is flushed using the `DomainObjectContainer`'s `#flush(...)` method, else is done implicitly when the transaction commits at the end of the user request.

`ObjectUpdatedEvent.Default` is the concrete implementation that is used.



In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

### 3.5.8. ObjectUpdatingEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is about to be updated in the database. This is done either explicitly when the current transaction is flushed using the `DomainObjectContainer`'s `#flush(...)` method, else is done implicitly when the transaction commits at the end of the user request.

`ObjectUpdatingEvent.Default` is the concrete implementation that is used.



In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

## 3.6. Value Types



TODO

Table 9. JDK Classes

Type	Description
<code>java.lang.Boolean</code>	
<code>java.lang.Character</code>	
<code>java.lang.Double</code>	
<code>java.lang.Float</code>	
<code>java.lang.Integer</code>	
<code>java.lang.Long</code>	
<code>java.lang.Short</code>	
<code>java.lang.String</code>	
<code>java.math.BigDecimal</code>	
<code>java.math.BigInteger</code>	
<code>java.sql.Date</code>	
<code>java.sql.Time</code>	
<code>java.sql.Timestamp</code>	

Type	Description
<code>java.util.Date</code>	

Table 10. JodaTime classes

Type	Description
<code>org.joda.time.DateTime</code>	
<code>org.joda.time.LocalDateTime</code>	
<code>org.joda.time.LocalDate</code>	

Table 11. Isis Applib classes

Type	Description
<code>o.a.i.applib.value.Blob</code>	
<code>o.a.i.applib.value.Clob</code>	
<code>o.a.i.applib.value.Color</code>	
<code>o.a.i.applib.value.Money</code>	
<code>o.a.i.applib.value.Password</code>	

### 3.6.1. Blob



TODO - the `org.apache.isis.applib.value.Blob` class.

### 3.6.2. Clob



TODO - the `org.apache.isis.applib.value.Clob` class.

### 3.6.3. Color



TODO - the `org.apache.isis.applib.value.Color` class.

### 3.6.4. Money



TODO - the `org.apache.isis.applib.value.Money` class.

### 3.6.5. Password



TODO

## 3.7. Applib Utility Classes

The `org.apache.isis.applib.util` package has a number of simple utility classes designed to simplify the coding of some common tasks.

### 3.7.1. Enums



TODO

```
public final class Enums {
    public static String getFriendlyNameOf(Enum<?> anEnum) { ... }
    public static String getFriendlyNameOf(String anEnumName) { ... }
    public static String getEnumNameFromFriendly(String anEnumFriendlyName) { ... }
    public static String enumToHTTPHeader(final Enum<?> anEnum) { ... }
    public static String enumNameToHTTPHeader(final String name) { ... }
    public static String enumToCamelCase(final Enum<?> anEnum) { ... }
}
```

### 3.7.2. ObjectContracts

The `ObjectContracts` test provides a series of methods to make it easy for your domain objects to:

- implement `Comparable` (eg so can be stored in `java.util.SortedSets`)
- implement `toString()`
- implement `equals()`
- implement `hashCode()`

For example:

```
public class ToDoItem implements Comparable<ToDoItem> {

    public boolean isComplete() { ... }
    public LocalDate getDueBy() { ... }
    public String getDescription() { ... }
    public String getOwnedBy() { ... }

    public int compareTo(final ToDoItem other) {
        return ObjectContracts.compare(this, other, "complete", "dueBy", "description");
    }

    public String toString() {
        return ObjectContracts.toString(this, "description", "complete", "dueBy",
        "ownedBy");
    }
}
```



Note that `ObjectContracts` makes heavy use of Java Reflection. While it's great to get going quickly in prototyping, we recommend you use your IDE to code generate implementations of these methods for production code.

Moreover (and perhaps even more importantly) `ObjectContracts` implementation can cause DataNucleus to recursively rehydrate a larger number of associated entities (More detail below).



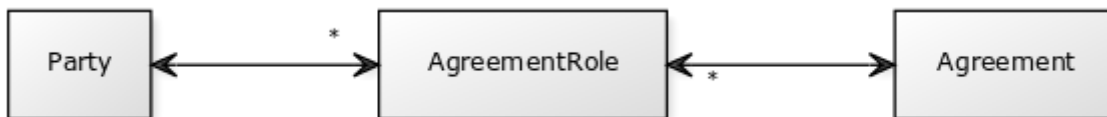
We therefore recommend that you disable persistence-by-reachability by adding:

`persistor_datanucleus.properties`

```
isis.persistor.datanucleus.impl.datanucleus.persistenceByReachabilityAtCommit=false
```

### The issue in more detail

Consider the `entities`:



In the course of a transaction, the `Agreement` entity is loaded into memory (not necessarily modified), and then new `AgreementRoles` are associated to it.

All these entities implement `Comparable` using `ObjectContracts`, so that the implementation of `AgreementRole`'s (simplified) is:

```
public class AgreementRole {
    ...
    public int compareTo(AgreementRole other) {
        return ObjectContracts.compareTo(this, other, "agreement", "startDate",
"party");
    }
    ...
}
```

while `Agreement`'s is implemented as:

```
public class Agreement {
    ...
    public int compareTo(Agreement other) {
        return ObjectContracts.compareTo(this, other, "reference");
    }
    ...
}
```

and `Party`'s is similarly implemented as:

```
public class Party {
    ...
    public int compareTo(Party other) {
        return ObjectContracts.compareTo(this, other, "reference");
    }
    ...
}
```

DataNucleus's persistence-by-reachability algorithm adds the `AgreementRoles` into a `SortedSet`, which causes `AgreementRole#compareTo()` to fire:

- the evaluation of the "agreement" property delegates back to the `Agreement`, whose own `Agreement#compareTo()` uses the scalar `reference` property. As the `Agreement` is already in-memory, this does not trigger any further database queries
- the evaluation of the "startDate" property is just a scalar property of the `AgreementRole`, so will already in-memory
- the evaluation of the "party" property delegates back to the `Party`, whose own `Party#compareTo()` requires the uses the scalar `reference` property. However, since the `Party` is not yet in-memory, using the `reference` property triggers a database query to "rehydrate" the `Party` instance.

In other words, figuring out whether `AgreementRole` is comparable requires the persistence-by-reachability algorithm to run, causing the adjacent associated entity `Party` to also be retrieved.

### 3.7.3. Reasons

There are two different classes provided to help build reasons returned by `disableXxx()` and `validateXxx()` methods:

- the `org.apache.isis.applib.util.ReasonBuffer` helper class
- the `org.apache.isis.applib.util.Reasons` helper class

For example:

```
public class Customer {
    ...
    public String validatePlaceOrder(Product p, int quantity) {
        return Reasons.coalesce(
            whetherCustomerBlacklisted(this),
            whetherProductOutOfStock(p)
        );
    }
}
```

Which you use (if any) is up to you.

### 3.7.4. TitleBuffer

The `TitleBuffer` utility class is intended to make it easy to construct title strings (returned from the `title()` method).

For example, it has overloaded versions of methods called `append()` and `concat()`.

## 3.8. Specification pattern

The interfaces and classes listed in this chapter provide support for the `Specification` pattern, as described in Eric Evans' book *Domain Driven Design*, p224.

Apache Isis will automatically apply such specifications as validation rules on properties (as per `@Property#mustSatisfy()`) and on action parameters (as per `@Parameter#mustSatisfy()`).

### 3.8.1. Specification

The heart of the support for this pattern is the `Specification` interface:

```
public interface Specification {  
    public String satisfies(Object obj); ①  
}
```

① if returns `null`, then the constraint is satisfied; otherwise returns the reason why the constraint has not been satisfied.

For example:

```
public class StartWithCapitalLetterSpecification implements Specification {  
    public String satisfies(Object proposedObj) {  
        String proposed = (String)proposedObj; ①  
        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 this ugly cast can be avoided using some of the other classes available; see below.

### 3.8.2. Specification2

The `Specification2` interface extends the `Specification` API to add support for i18n. This is done by defining an additional method that returns a [translatable string](#):

```
public interface Specification2 extends Specification {
    public TranslatableString satisfiesTranslatable(Object obj); ①
}
```

① if returns `null`, then the constraint is satisfied; otherwise returns the reason why the constraint has not been satisfied.

Note that if implementing `Specification2` then there is no need to also provide an implementation of the inherited `satisfies(Object)` method; this will never be called by the framework for `Specification2` instances.

### 3.8.3. Adapter classes

The `AbstractSpecification` and `AbstractSpecification2` adapter classes provide a partial implementation of the respective interfaces, providing type-safety. (Their design is modelled on the `TypesafeMatcher` class within [Hamcrest](#)).

For example:

```
public class StartWithCapitalLetterSpecification extends AbstractSpecification<String>
{
    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() { ... }
    ...
}
```

The `AbstractSpecification2` class is almost identical; its type-safe method is `satisfiesTranslatableSafely(T)` instead.

### 3.8.4. Combining specifications

There are also adapter classes that can be inherited from to combine specifications:

- `SpecificationAnd` - all provided specifications' constraints must be met

- `SpecificationOr` - at least one provided specifications' constraints must be met
- `SpecificationNot` - its constraints are met if-and-only-if the provided specification's constraint was *not* met.

Note that these adapter classes inherit `Specification` but do not inherit `Specification2`; in other words they do not support i18n.

## 3.9. i18n support

The `org.apache.isis.applib.services.i18n` package contains a single class to support i18n.

### 3.9.1. TranslatableString



TODO - see [user guide, i18n](#).

The `TranslatableString` utility class ...

## 3.10. Contributee

The interfaces listed in this chapter act as contributees; they allow domain services to contribute actions/properties/collections to any domain objects that implement these interfaces.

### 3.10.1. HasTransactionId

The `HasTransactionId` interface is a mix-in for any domain objects that reference a transaction id, such as auditing entries or commands.

The interface is defined is:

```
public interface HasTransactionId {
    public UUID getTransactionId();
    public void setTransactionId(final UUID transactionId);
}
```

① unique identifier (a GUID) of the transaction in which this interaction occurred.

Modules that either have domain entity that implement and/or services that contribute this interface are:

- (non-ASF) `Isis addons' audit` module (`AuditEntry` entity, `AuditingServiceContributions` service)
- (non-ASF) `Isis addons' command` module (`CommandJdo` entity, `CommandServiceJdoContributions` service)
- (non-ASF) `Isis addons' publishing` module (`PublishedEvent` entity, `PublishingServiceContributions`)

### 3.10.2. HasUsername

The `HasUsername` interface is a mix-in for domain objects to be associated with a username. Other services and modules can then contribute actions/collections to render such additional information relating to the activities of the user.

The interface is defined is:

```
public interface HasUsername {
    public String getUsername();
}
```

Modules that either have domain entity that implement and/or services that contribute this interface are:

- (non-ASF) `Isis addons' security` module ( `ApplicationUser` entity, `HasUsernameContributions` service)
- (non-ASF) `Isis addons' audit` module (`AuditEntry` entity,
- (non-ASF) `Isis addons' command` module's `CommandJdo` entity, `HasUsernameContributions` service)
- (non-ASF) `Isis addons' publishing` module (`PublishedEvent` entity)
- (non-ASF) `Isis addons' sessionlogger` module (`SessionLogEntry` entity, `HasUsernameContributions` service)
- (non-ASF) `Isis addons' settings` module (`UserSettingJdo` entity)

## 3.11. Roles

The interfaces listed in this chapter are role interfaces; they define a contract for the framework to interact with those domain objects that implement these interfaces.

### 3.11.1. HoldsUpdatedAt

The `HoldsUpdatedAt` role interface allows the (framework-provided) `TimestampService` to update each object with the current timestamp whenever it is modified in a transaction.

The interface is defined as:

```
public interface HoldsUpdatedAt {
    void setUpdatedAt(java.sql.Timestamp updatedAt);
}
```

The current time is obtained from the `ClockService`.

Entities that implement this interface often also implement `HoldsUpdatedBy` role interface; as a convenience the `Timestampable` interface combines the two roles.

## Alternative approaches

An alternative way to maintain a timestamp is to use JDO's `@Version` annotation. With this approach, it is the JDO/DataNucleus that maintains the version, rather than the framework's `TimestampService`.

For example:

```
@javax.jdo.annotations.Version(  
    strategy=VersionStrategy.DATE_TIME,  
    column="version")  
public class Customer {  
    ...  
    public java.sql.Timestamp getVersionSequence() {  
        return (java.sql.Timestamp) JDOHelper.getVersion(this);  
    }  
}
```

### 3.11.2. HoldsUpdatedBy

The `HoldsUpdatedBy` role interface ...

```
public interface HoldsUpdatedBy {  
    void setUpdatedBy(String updatedBy);  
}
```

Entities that implement this interface often also implement `HoldsUpdatedAt` role interface; as a convenience the `Timestampable` interface combines the two roles.

### 3.11.3. Timestampable

The `Timestampable` role interface is a convenience that combines the `HoldsUpdatedAt` and `HoldsUpdatedBy` interfaces. It is defined as:

```
public interface Timestampable  
    extends HoldsUpdatedAt, HoldsUpdatedBy {  
}
```

The interface no additional methods of its own.

## Alternatives

An alternative way to maintain a timestamp is to use JDO's `@Version` annotation. With this approach, it is the JDO/DataNucleus that maintains the version, rather than the framework's `TimestampService`. See `HoldsUpdatedBy` for further details.

## 3.12. Mixins

The interfaces listed in this chapter are role interfaces; they define a contract for the framework to interact with those domain objects that implement these interfaces.

### 3.12.1. Object

The framework provides a single mixin that contributes to simply `java.lang.Object`. It provides the ability to download the layout XML for any domain object (in practical terms: entities and view models).

#### `clearHints()`

When a domain object is rendered the end-user can select different tabs, and for collections can sort the columns, navigate to second pages, or select different views of collections. If the user revisits that object, the [Wicket viewer](#) will remember these hints and render the domain object in the same state. These rendering hints are also included if the user copies the URL using the anchor link (to right hand of the object's title).

The `Object_clearHints` mixin provides the ability for the end-user to discard these hints so that the object is rendered in its initial state:

```
public void clearHints() {  
    ...  
}
```

#### Appearance in the UI

This mixin actions are all associated with the "Metadata" fieldset. If there is no such field set, then the action will be rendered as a top-level action).

#### Related Services

This mixin uses the `HintStore` service to store and retrieve UI hints for each rendered object, per user.

#### `downloadLayoutXml()`

The `Object_downloadLayoutXml` mixin provides an action to download the [layout XML](#) for the current domain object. It has the following signature:

```
public Object downloadLayoutXml(  
    @ParameterLayout(named = "File name")  
    final String fileName,  
    final LayoutService.Style style) {           ①  
    ...  
}
```

① either current, complete, normalized or minimal.



See the documentation on [layout XML](#) and also the [LayoutService](#) for more information on these styles

#### Appearance in the UI

This mixin actions are all associated with the "Metadata" fieldset.

A number of other [mixins](#) also contribute properties and actions to the "Metadata" fieldset.

#### Related Services

This mixin calls [LayoutService](#) to obtain the layout XML.

#### `rebuildMetamodel()`

The `Object_rebuildMetamodel` mixin provides the ability to discard the current internal metamodel data (an instance of `ObjectSpecification`) for the domain class of the rendered object, and recreate from code and other sources (most notably, layout XML data). It has the following signature:

```
public void rebuildMetamodel() {  
    ...  
}
```

#### Appearance in the UI

This mixin actions are all associated with the "Metadata" fieldset.

A number of other [mixins](#) also contribute properties and actions to the "Metadata" fieldset.

#### Related Services

This mixin calls [MetaModelService](#) and the [GridService](#) to invalidate their caches.

### 3.12.2. Dto

The `Dto` role interface is intended to be implemented by JAXB-annotated view models, that is, annotated using `@XmlRootElement`. It enables the ability to download the XML and XSD schema of those objects using two [mixins](#), `Dto_downloadXml` and `Dto_downloadXsd`.

The interface is just a marker interface (with no members), and is defined as:

```
public interface Dto { }
```

The `Dto_downloadXml` mixin defines the following action:

```

@Mixin
public class Dto_downloadXml {
    public Dto_downloadXml(final Dto dto) { ... } ①
    public Object downloadXml(final String fileName) { ... } ②
    ...
}

```

- ① provided as an action to any class that (trivially) implements the `Dto` interface
- ② actually this method is called '\$\$' in the code, a "special case" that means to use the derive the action name from the class name.

This will return the XML text wrapped up in a `Clob`.

The `Dto_downloadXsd` mixin is similar:

```

@Mixin
public class Dto_downloadXsd {
    public Dto_downloadXsd(final Dto dto) { ... }
    ① public Object downloadXsd(final String fileName, final IsisSchemes isisSchemes) {
    ... } ②
}

```

- ① provided as an action to any class that (trivially) implements the `Dto` interface
- ② actually this is '\$\$' in the code, a "special case" that means to use the derive the action name from the class name.

If the domain object's JAXB annotations reference only a single XSD schema then this will return that XML text as a `Clob` of that XSD. If there are multiple XSD schemas referenced then the action will return a zip of those schemas, wrapped up in a `Blob`. The `IsisSchemes` parameter to the action can be used to optionally ignore the common [Apache Isis schemas](#) (useful if there is only one other XSD schema referenced by the DTO).

## Related Services

The `Dto_downloadXml` and `Dto_downloadXsd` delegate to the `JaxbService` to actually generate the XML/XSD.

### 3.12.3. Persistable

All domain entities automatically implement the DataNucleus `Persistable` role interface as a result of the enhancer process (the fully qualified class name is `org.datanucleus.enhancement.Persistable`). So as a developer you do not need to write any code to obtain the mixins that contribute to this interface.

`downloadJdoMetadata()`

The `Persistable_downloadJdoMetadata` mixin provides an action which allows the JDO [class metadata](#)

to be downloaded as XML. It has the following signature:

```
public Clob downloadJdoMetadata( ①
    @ParameterLayout(named = ".jdo file name")
    final String fileName) {
    ...
}
```

① returns the XML text wrapped up in a `Clob`.

### Appearance in the UI

This mixin action is associated with the "Metadata" fieldset, and will appear as a panel drop-down action.

These mixin properties are all associated with the "Metadata" fieldset. The [Object mixin](#) also contribute an action to the "Metadata" fieldset.

### Related Services

The mixin delegates to the `IsisJdoSupport` service to obtain a reference to the `JDO PersistenceManagerFactory`.

### `datanucleusXxx`

The framework provides a number of mixins that expose the datanucleus Id and version of a persistable domain entity. Several implementations are provided to support different datatypes:

- `Persistable_datanucleusIdLong` will expose the entity's id, assuming that the id is or can be cast to `java.lang.Long`. Otherwise the property will be hidden.
- `Persistable_datanucleusVersionTimestamp` will expose the entity's version, assuming that the version is or can be cast to `java.sql.Timestamp`. Otherwise the property will be hidden.
- `Persistable_datanucleusVersionLong` will expose the entity's version, assuming that the version is or can be cast to `java.lang.Long`. Otherwise the property will be hidden.

### Appearance in the UI

These mixin properties are all associated with the "Metadata" fieldset. The [Object mixin](#) also contribute an action to the "Metadata" fieldset.

## 3.13. Layout

The `org.apache.isis.applib.layout` package defines a number of classes that allow the layout of domain objects (entities and view models) to be customized. These classes fall into two main categories:

- grid classes, that define a grid structure of rows, columns, tab groups and tabs, and;
- common component classes, that capture the layout metadata for an object's properties, collections and actions. These are bound (or associated) to the regions of the grid

The framework provides an implementation of the grid classes modelled closely on [Bootstrap 3](#), along with [Wicket viewer](#) components capable of rendering that grid system. In principle it is also possible to extend the layout architecture for other grid systems. The component classes, though, are intended to be reusable across all grid systems.

The component classes, meanwhile, are broadly equivalent to the "layout" annotations ([@PropertyLayout](#), [@CollectionLayout](#), [@ActionLayout](#) and [@DomainObjectLayout](#)

All of the classes in this package are JAXB-annotated, meaning that they can be serialized to/from XML (the `component` classes in the <http://isis.apache.org/applib/layout/component> XSD namespace, the bootstrap 3 grid classes in the <http://isis.apache.org/applib/layout/grid/bootstrap3> XSD namespace). This ability to serialize to/from XML is used by the [GridLoaderService](#), the default implementation of which reads the grid layout for a domain class from a `.layout.xml` file on the classpath.

### 3.13.1. Component

The component classes reside in the `org.apache.isis.applib.layout.component` package, and consist of:

- [FieldSet](#)

A fieldset (previously also called a property group or member group) of a number of the domain object's properties (along with any associated actions of those properties).

- layout data classes, which correspond to the similarly named annotations:
  - [PropertyLayoutData](#), corresponding to the [@PropertyLayout](#) annotation;
  - [CollectionLayoutData](#), corresponding to the [@CollectionLayout](#) annotation;
  - [ActionLayoutData](#), corresponding to the [@ActionLayout](#) annotation;
  - [DomainObjectLayoutData](#), corresponding to the [@DomainObjectLayout](#) annotation.

In addition, the component package includes [Grid](#), representing the top level container for a custom layout for a domain object. [Grid](#) itself is merely an interface, but it also defines the visitor pattern to make it easy for validate and normalize the grid layouts. The [GridAbstract](#) convenience superclass provides a partial implementation of this visitor pattern.

### 3.13.2. Bootstrap3 Grid

As noted above, the default bootstrap3 grid classes are modelled closely on [Bootstrap 3](#). Bootstrap's [grid system](#) divides the page width equally into 12 columns, and so each column spans 1 or more of these widths. Thus, a column with a span of 12 is the full width, one with a span of 6 is half the width, one with a span of 4 is a third of the width, and so on.

When specifying the span of a column, Bootstrap also allows a size to be specified ([XS](#), [SM](#), [MD](#), [LG](#)). The size determines the rules for responsive design. Apache Isis defaults to [MD](#) but this can be overridden. It is also possible to specify multiple size/spans for a given column.

The grid classes provided by Apache Isis reside in the

`org.apache.isis.applib.layout.grid.bootstrap3` package, and consist of:

- `BS3Grid`

Consists of a number of `BS3Rows`.

This class is the concrete implementation of `Grid` interface, discussed previously. As such, it extends the `Grid.Visitor` to iterate over all of the `Rows` of the grid.

- `BS3Row`

A container of `BS3Cols`. This element is rendered as `<div class="row">`.

- `BS3Col`

A container of almost everything else. A column most commonly contains properties (grouped into `FieldSets`, described above) or collections (specified by `CollectionLayoutData`, also above). However, a `Col` might instead contain a `BS3TabGroup` (described below) in order that the object members is arranged into tabs.

It is also possible for a `Col` to contain the object's title/icon (using `DomainObjectLayoutData`) or indeed arbitrary actions (using `ActionLayoutData`).

Finally, a `BS3Col` can also contain other `BS3Rows`, allowing arbitrarily deep hierarchies of containers as required.

This element is rendered as, for example, `<div class="col-md-4">` (for a size `MD`, span of 4).

- `BS3TabGroup`

A container of `BS3Tabs`.

- `BS3Tab`

A container of `BS3Rows`, which will in turn contain `BS3Cols` and thence ultimately the object's members.

There are also two close cousins of `Col`, namely `ClearFixVisible` and `ClearFixHidden`. These map to Bootstrap's [responsive utility classes](#), and provide greater control for responsive designs.

As you can probably guess, the `BS3Grid` is the top-level object (that is, it is JAXB `@XmlElement`); this is the object that is serialized to/from XML.

All of these classes also allow custom CSS to be specified; these are added to the CSS classes for the corresponding `<div>` in the rendered page. The `application.css` file can then be used for application-specific CSS, allowing arbitrary fine-tuning of the layout of the page.

# Chapter 4. Schema

Most applications need to integrate with other apps in the enterprise. To facilitate such integration scenarios, Apache Isis defines two standard XSD schemas:

- the [common schema](#), which defines the `oidDto` complex type: the object's type and its identifier.

It also defines some two further XSD types, which are building blocks used in...

- the [action memento invocation](#) schema, which allows action invocations to be captured and reified.

The (non-ASF) [Isis addons'](#) `publishmq` module uses this XML format to publish messages onto an [Apache ActiveMQ](#) message queue.

The sections below discuss these schemas in more detail.

## 4.1. Common Schema

The main purpose of the common schema is to define the `oidDto` complex type: the object's type and its identifier. This is basically a formal XML equivalent to the `Bookmark` object obtained from the `BookmarkService`.

Although simple, this is an enormously powerful concept, in that it represents a URI to any domain object managed by a given Apache Isis application. With it, we have the ability to lookup any arbitrary object. Further discussion and examples can be found [here](#).

```

<xs:schema targetNamespace="http://isis.apache.org/schema/common"
①
    elementFormDefault="qualified"
    xmlns="http://isis.apache.org/schema/common"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:complexType name="oidDto">
②
        <xs:sequence>
            <xs:element name="objectType" type="xs:string"/>
③
            <xs:element name="objectIdentifier" type="xs:string"/>
④
        </xs:sequence>
        <xs:attribute name="objectState" type="bookmarkObjectState"
default="persistent"/>
        </xs:complexType>

        <xs:simpleType name="bookmarkObjectState">
⑤
            <xs:restriction base="xs:string">
                <xs:enumeration value="persistent"/>
                <xs:enumeration value="transient"/>
                <xs:enumeration value="viewModel"/>
            </xs:restriction>
        </xs:simpleType>
        ...
    </xs:schema>

```

- ① the common schema has a namespace URI of "http://isis.apache.org/schema/common". Although URIs are not the same as URLs, you will find that the schemas are also downloadable from this location.
- ② the `oidDto` complex type defines the unique identifier for any domain object: its type, and an identifier. The `objectState` attribute is of type `bookmarkObjectState` and will typically be "transient" or "viewModel"
- ③ the object type, corresponding to either the `@DomainObject#objectType()` attribute, or to the (JDO) `@PersistenceCapable` annotation (`schema` and/or `table` attributes), or to the (JDO) `@Discriminator` annotation. If none is specified, then the fully qualified class name will be used.
- ④ the object identifier (aka primary key), converted to string form.
- ⑤ the `bookmarkObjectState` enumerates the possible persistence states of the referenced object. Although "transient" is shown, in practice the only types of objects will be "persistent" (entities) or "viewModel".

The `oidDto` complex type is used in two specific places by the framework:

- first, as a means of serializing JAXB view model/DTOs (annotated with `@XmlElement`), that reference domain entities.

These references are serialized instead into OIDs

- second, as references to the target of action invocations, as described by the [action invocation memento](#) schema.

They are also used to represent references to any action arguments that take domain object entities.

The common schema also defines two further types: the `valueDto` complex type and the `valueType` simple type:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/common"
  elementFormDefault="qualified"
  xmlns="http://isis.apache.org/schema/common"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
  <xs:complexType name="valueDto"> ①
    <xs:choice minOccurs="0" maxOccurs="1">
      <xs:element name="string" type="xs:string"/>
      <xs:element name="byte" type="xs:byte"/>
      <xs:element name="short" type="xs:short"/>
      ...
      <xs:element name="localTime" type="xs:time"/>
      <xs:element name="reference" type="oidDto"/>
    </xs:choice>
  </xs:complexType>

  <xs:simpleType name="valueType"> ②
    <xs:restriction base="xs:string">
      <xs:enumeration value="string"/>
      <xs:enumeration value="byte"/>
      <xs:enumeration value="short"/>
      ...
      <xs:enumeration value="isisApplibClob"/>
      <xs:enumeration value="reference"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

These two type definitions are just building blocks, also used within the [action invocation memento](#) schema. The first, `valueDto` is intended to hold any valid value, eg of an argument to an action. The second, `valueType`, enumerates the different types of vales, eg of a formal parameter to an action.

## 4.2. Action Invocation Memento

The "aim" schema defines the serialized form (or memento) of an action invocation.

Action invocations are captured (in memory rather than in serialized form) when the end-user



invokes the action "through" the UI, by way of the [CommandContext](#) service. Using the [ActionInvocationMementoDtoUtils](#) utility class, a service can instantiate [ActionInvocationMementoDto](#) which can then be serialized to/from using the same [ActionInvocationMementoDtoUtils](#) class.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://isis.apache.org/schema/aim"
①
    elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://isis.apache.org/schema/aim"
    xmlns:common="http://isis.apache.org/schema/common">

    <xs:import namespace="http://isis.apache.org/schema/common"
②
        schemaLocation="http://isis.apache.org/schema/common/common-1.0.xsd"/>

    <xs:element name="actionInvocationMementoDto">
③
        <xs:complexType>
            <xs:sequence>
                <xs:element name="metadata">
                    <xs:complexType>
                        <xs:sequence>
④
                            <xs:element name="transactionId" type="xs:string"/>
⑤
                            <xs:element name="sequence" type="xs:int"/>
⑥
                            <xs:element name="timestamp" type="xs:dateTime"/>
⑦
                            <xs:element name="target" type="common:oidDto"/>
⑧
                            <xs:element name="targetClass" type="xs:string"/>
⑨
                            <xs:element name="targetAction" type="xs:string"/>
⑩
                            <xs:element name="actionIdentifier" type="xs:string"/>
⑪
                            <xs:element name="user" type="xs:string"/>
⑫
                            <xs:element name="title" type="xs:string"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="payload">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="parameters">
⑬
                                <xs:complexType>
```

```

        <xs:sequence maxOccurs="unbounded">
            <xs:element name="param" type="paramDto"/>
        </xs:sequence>
        <xs:attribute name="num" use="required"
type="xs:int"/>
    </xs:complexType>
</xs:element>
<xs:element name="return" type="common:valueDto"
    minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:complexType name="paramDto">
    <xs:sequence>
        <xs:element name="value" type="common:valueDto"/>
    </xs:sequence>
    <xs:attribute name="parameterName" use="required" type="xs:string"/>
    <xs:attribute name="parameterType" use="required" type="common:valueType"/>
    <xs:attribute name="null" use="optional" type="xs:boolean"/>
</xs:complexType>
</xs:schema>

```

- ① the aim schema has a namespace URI of "http://isis.apache.org/schema/aim". Although URIs are not the same as URLs, you will find that the schemas are also downloadable from this location.
- ② reuses the `common` schema
- ③ definition of the `actionInvocationMementoDto` complex type. This consists of metadata (the transaction identifier, the target object, the initiating user) and the payload (the action parameter/arguments, the return value if known).
- ④ the unique transaction Id (a guid) allocated by the framework for each and every transaction
- ⑤ a sequence number within the transaction. It is possible for there to be more than one action invocation to be
- ⑥ when the action was invoked
- ⑦ target object, as an OID (using `oidDto` from the `common` schema)
- ⑧ fully qualified class name of the target object, for information only
- ⑨ name of the action, for information only
- ⑩ Javadoc style unique identifier for the action.
- ⑪ User that invoked the action

- ⑫ title of the target object, for information only
- ⑬ Collection of parameter/arguments, defined in terms of the `paramDto` complex type (discussed just below)
- ⑭ The return value of the action, if known (and not void)
- ⑮ The `paramDto` defines both an action parameter and its corresponding argument values
- ⑯ The value of the parameter, in other words an argument value
- ⑰ Metadata about the parameter itself: its name, type, optionality.



As of `1.11.0` this schema is not used directly by the framework; in particular `Command#setMemento(...)` sets a similar but less formal XML structure. This may change in the future.