Domain Services

Table of Contents

Chapter 1. Domain Services

This guide documents Apache Isis' domain services, both those that act as an API (implemented by the framework for your domain objects to call), and those domain services that act as an SPI (implemented by your domain application and which are called by the framework).

1.1. Other Guides

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

The user guides available are:

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

The reference guides are:

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

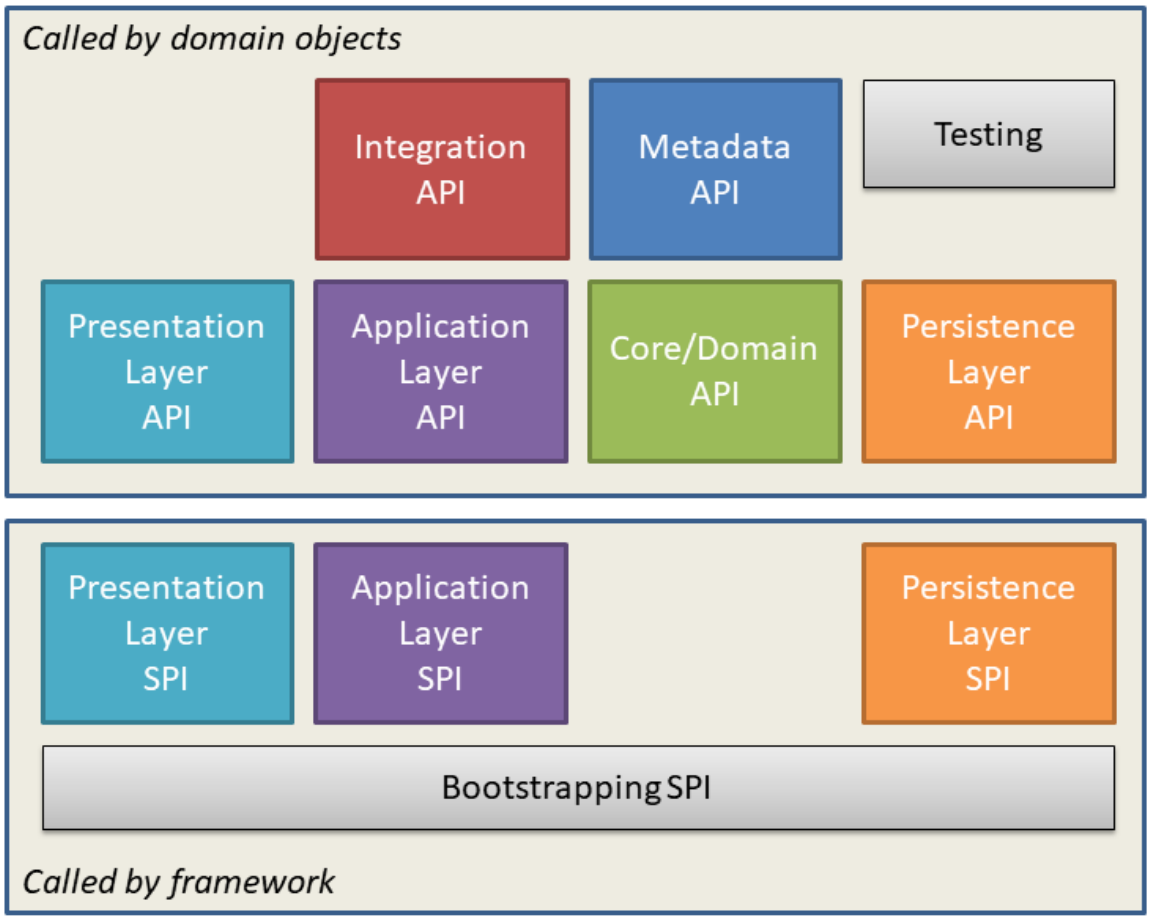
The remaining guides are:

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

Chapter 2. Introduction

2.1. Types of Domain Service

The domain services also group into various broad categories. Many support functionality of the various layers of the system (presentation layer, application layer, core domain, persistence layer); others exist to allow the domain objects to integrate with other bounded contexts, or provide various metadata (eg for development-time tooling). The diagram below shows these categories:



A small number of domain services can be considered both API and SPI; a good example is the `EmailService` that is of direct use for domain objects wishing to send out emails, but is also used by the framework to support the user registration functionality supported by the Wicket viewer. The same is true of the `EventBusService`; this can be used by domain objects to broadcast arbitrary events, but is also used by the framework to automatically emit events for `@Action#domainEvent()` etc.

For these hybrid services we have categorized the service as an "API" service. This chapter therefore contains only the strictly SPI services.

This rest of this guide is broken out into several chapters, one for each of the various types/categories of domain service.

2.2. Public API vs Internal Services

The vast majority of Apache Isis' domain services are defined in Apache Isis' applib (`o.a.i.core:isis-core-applib` module) as stable, public classes. Importantly, this also minimizes the coupling between your code and Apache Isis, allowing you to easily mock out these services in your unit tests.

The framework also defines a number of "internal" services. These are not part of the framework's formal API, in that they use classes that are outside of the applib. These internal framework services should be thought of as part of the internal design of the framework, and are liable to change from release to release. The internal framework services are documented in the Framework Internal Services guide.

2.3. Using the services

Apache Isis includes an extensive number of domain services for your domain objects to use; simply define the service as an annotated field and Apache Isis will inject the service into your object.

For example:

```
public class Customer {

    public void sendEmail( String subject, String body) {
        List<String> cc = Collections.emptyList;
        List<String> bcc = Collections.emptyList;
        emailService.send(getEmailAddress(), cc, bcc, subject, body);
    }
    public boolean hideSendEmail() {
        return !emailService.isConfigured();
    }

    @Inject                                                    ①
    EmailService emailService;
}
```

① Service automatically injected by the framework.

For objects that are already persisted, the service is automatically injected just after the object is rehydrated by JDO/DataNucleus.

For transient objects (instantiated programmatically), the `FactoryService`'s `instantiate()` method (or the deprecated `DomainObjectContainer`'s `newTransientInstance()` method) will automatically inject the services.

Alternatively the object can be instantiated simply using `new`, then services injected using `ServiceRegistry`'s `injectServicesInto(…)` method (or the deprecated `DomainObjectContainer`'s `injectServicesInto(…)` method).

2.4. Overriding the services

The framework provides default implementations for many of the domain services. This is convenient, but sometimes you will want to replace the default implementation with your own service implementation.

The trick is to use the `@DomainServiceLayout#menuOrder()` attribute, specifying a low number (typically `"1"`).

For a small number of domain services, all implementations are used (following the chain-of-responsibility pattern), not just the first one. The services in question are: `ContentMappingService`, `GridSystemService`, and `RoutingService`.

For example, suppose you wanted to provide your own implementation of `LocaleProvider`. Here's how:

```
@DomainService(
        nature = NatureOfService.DOMAIN
)
@DomainServiceLayout(
        menuOrder = "1"                                 ①
)
public class MyLocaleProvider implements LocaleProvider {
    @Override
    public Locale getLocale() {
        return ...
    }
}
```

① takes precedence over the default implementation.

It's also quite common to want to decorate the existing implementation (ie have your own implementation delegate to the default); this is also possible and quite easy (if using `1.10.0` or later). The idea is to have the framework inject all implementations of the service, and then to delegate to the first one that isn't "this" one:

```
@DomainService(nature=NatureOfService.DOMAIN)
@DomainServiceLayout(
        menuOrder = "1"
①
)
public class MyLocaleProvider implements LocaleProvider {
    @Override
    public Locale getLocale() {
        return getDelegateLocaleProvider().getLocale();
②
    }
    private LocaleProvider getDelegateLocaleProvider() {
        return Iterables.tryFind(localeProviders, input -> input != this).orNull();
③
    }
    @Inject
    List<LocaleProvider> localeProviders;
④
}
```

① takes precedence over the default implementation when injected elsewhere.

② this implementation merely delegates to the default implementation

③ find the first implementation that isn't *this* implementation (else infinite loop!)

④ injects all implementations, including this implemenation

The above code could be improved by caching the delegateLocaleProvider once located (rather than searching each time).

2.5. Command and Events

A good number of the domain services manage the execution of action invocations/property edits, along with the state of domain objects that are modified as a result of these. These services capture information which can then be used for various purposes, most notably for auditing or for publishing events, or for deferring execution such that the execution be performed in the background at some later date.

The diagram below shows how these services fit together. The outline boxes are services while the coloured boxes represent data structures - defined in the applib and therefore accessible to domain applications - which hold various information about the executions.

To explain:

- the (request-scoped) `CommandContext` captures the user's intention to invoke an action or edit a property; this is held by the `Command` object.
- if a `CommandService` has been configured, then this will be used to create the `Command` object implementation, generally so that it can then also be persisted.

If the action or property is annotated to be invoked in the background (using `@Action#command…()` or `@Property#command…()`) then no further work is done. But, if the action/property is to be executed in the foreground, then the interaction continues.

- the (request-scoped) `InteractionContext` domain service acts as a factory for the `Interaction` object, which keeps track of the call-graph of executions (`Interaction.Execution`) of either action invocations or property edits. In the majority of cases there is likely to be just a single top-level node of this graph, but for applications that use the `WrapperFactory` extensively each successive call results in a new child execution.
- before and after each action invocation/property edit, a domain event is may be broadcast to all subscribers. Whether this occurs depends on whether the action/property has been annotated (using `@Action#domainEvent()` or `@Property#domainEvent()`).

  (Note that susbcribers will also receive events for vetoing the action/property; this is not shown on the diagram).

- As each execution progresses, and objects that are modified are "enlisted" into the (internal) `ChangedObjectsServiceInternal` domain service. Metrics as to which objects are merely loaded into memory are also captured using the `MetricsService` (not shown on the diagram).
- At the end of each execution, details of that execution are published through the (internal) `PublisherServiceInternal` domain service. This is only done for actions/properties annotated appropriate (with `@Action#publishing()` or `@Property#publishing()`).

  The internal service delegates in turn to any registered `PublishingService` (deprecated) and also to any registered `PublisherService`s (there may be more than one).

- At the end of each transaction, details of all changed objects are published, again through the (internal) `PublisherServiceInternal` to any registered `PublishingService` or `PublisherService` implementations. Only domain objects specified to be published with `@DomainObject#publishing()` are published.

  Note that it's possible for there to be more than one transaction per top-level interaction, by virtue of the `TransactionService`.

- Also at the end of each transaction, details of all changed properties are passed to any registered `AuditerService` or `AuditingService` (the latter deprecated) by way of the (internal) `AuditingServiceInternal` domain service.

Implementations of `CommandService` can use the `Command#getMemento()` method to obtain a XML equivalent of that `Command`, reified using the `cmd.xsd` schema. This can be converted back into a `CommandDto` using the `CommandDtoUtils` utility class (part of the applib).

Similarly, implementations of `PublisherService` can use the `InteractionDtoUtils` utility class to obtain a `InteractionDto` representing the interaction, either just for a single execution or for the entire call-graph. This can be converted into XML in a similar fashion.

Likewise, the `PublishedObjects` class passed to the `PublisherService` at the end of the interaction provides the `PublishedObjects#getDto()` method which returns a `ChangesDto` instance. This can be converted into XML using the `ChangesDtoUtils` utility class.

One final point: multiple `PublisherService` implementations are supported because different implementations may have different responsibilities. For example, the (non-ASF) Incode Platform's publishmq module is responsible for publishing messages onto an ActiveMQ event bus, for inter-system communication. However, the SPI can also be used for profiling; each execution within the call-graph contains metrics of the number of objects loaded or modified as a result of that execution, and thus could be used for application profiling. The framework provides a default `PublisherServiceLogging` implementation that logs this using SLF4J.

Chapter 3. Presentation Layer API

Domain service APIs for the presentation layer allow the domain objects to control aspects of the user interface. The implementations are specific to the particular viewer (Wicket viewer or Restful Objects viewer) so the domain code must guard against them being unavailable in some cases.

The table below summarizes the presentation layer APIs defined by Apache Isis. It also lists their corresponding implementation.

Table 1. Presentation Layer API

API	Description	Implementation	Notes
`o.a.i.applib.services.acceptheader.AcceptHeaderService`	Request-scoped access to HTTP Accept headers.	`AcceptHeaderServiceDefault` `o.a.i.core.isis-core-viewer-restfulobjects-rendering`	Implementation only usable within the Restful Objects viewer.
`o.a.i.applib.services.bookmarkui.BookmarkUiService`	Manage bookmarks and breadcrumbs in the (Wicket) UI.	`BookmarkUiServiceWicket` `o.a.i.core.isis-core-viewer-wicket-impl`	(Implementation only usable within the Wicket viewer.
`o.a.i.applib.services.deeplink.DeepLinkService`	Obtain a URL to a domain object (eg for use within an email or report)	`DeepLinkServiceWicket` `o.a.i.viewer.isis-core-viewer-wicket-impl`	Implementation only usable within the Wicket viewer.
`o.a.i.applib.services.guice.GuiceBeanProvider`	Access to internal framework services initialized using Guice DI.	`GuiceBeanProviderWicket` `o.a.i.core.isis-core-viewer-wicket-impl`	Implementation only usable within the Wicket viewer.

Key:

- `o.a.i` is an abbreviation for `org.apache.isis`
- `o.ia.m` is an abbreviation for `org.isisaddons.module`

- `o.a.i.c.m.s` is an abbreviation for `org.apache.isis.core.metamodel.services`
- `o.a.i.c.r.s` is an abbreviation for `org.apache.isis.core.runtime.services`

3.1. AcceptHeaderService

The `AcceptHeaderService` domain service is a `@RequestScoped` service that simply exposes the HTTP `Accept` header to the domain. Its intended use is to support multiple versions of a REST API, where the responsibility for content negotiation (determining which version of the REST API is to be used) is managed by logic in the domain objects themselves.

As an alternative to performing content negotiation within the domain classes, the `ContentNegotiationService` and `ContentMappingService` SPI domain services allow the framework to perform the content negotiation responsibility.

3.1.1. API & Implementation

The API defined by the service is:

```
@DomainService(nature = NatureOfService.DOMAIN)
@RequestScoped                                          ①
public interface AcceptHeaderService {
    @Programmatic
    List<MediaType> getAcceptableMediaTypes();          ②
}
```

① is `@RequestScoped`, so this domain service instance is scoped to a particular request and is then destroyed

② returns the list of media types found in the HTTP Accept header.

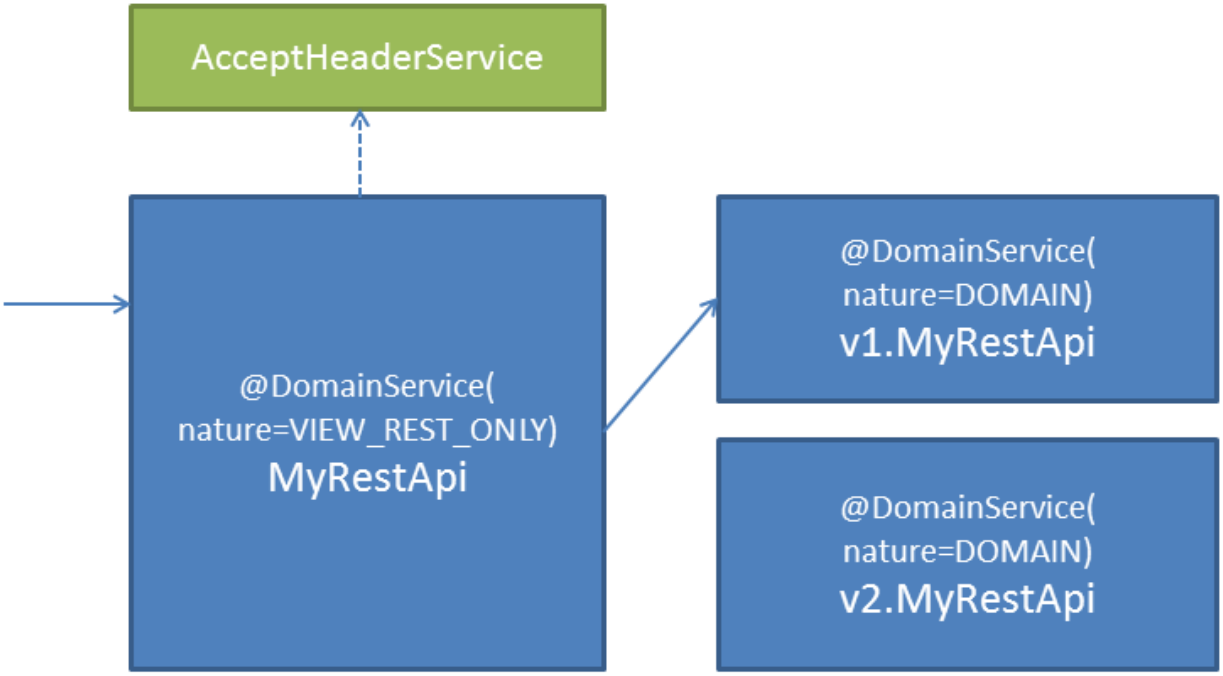
The default implementation is provided by `o.a.i.v.ro.rendering.service.acceptheader.AcceptHeaderServiceForRest`.

Note that the service will only return a list when the request is initiated through the Restful Objects viewer. Otherwise the service will return `null`.

3.1.2. Usage

The intended use of this service is where there are multiple concurrent versions of a REST API, for backward compatibility of existing clients. The `AcceptHeaderService` allows the responsibility for content negotiation (determining which version of the REST API is to be used) to be performed by logic in the domain objects themselves.

The diagram below illustrated this:



The REST request is submitted to a domain service with a nature of `VIEW_REST_ONLY` (`MyRestApi` in the diagram). This uses the `AcceptHeaderService` to obtain the values of the HTTP `Accept` header. Based on this it delegates to the appropriate underlying domain service (with a nature of `DOMAIN` so that they are not exposed in the REST API at all).

The service does not define any conventions as to the format of the media types. The option is to use the media type's type/subtype, eg `application/vnd.myrestapi-v1+json`; an alternative is to use a media type parameter as a hint, eg `application/json;x-my-rest-api-version=1` (where `x-my-rest-api-version` is the media type parameter).

The Restful Objects specification does this something similar with its own `x-ro-domain-type` media type parameter; this is used by the `ContentMappingService` to determine how to map domain objects to view models/DTOs.

3.2. `BookmarkUiService`

The `BookmarkUiService` provides the ability to programmatically interact with bookmarked pages and breadcrumbs, as rendered by the Wicket viewer.

One possible use case is programmatically to support multiple "contexts" and allow the end-user to switch from one context to another.

3.2.1. API & Implementation

The API defined by `BookmarkUiService` is:

```
public interface BookmarkUiService {
    void clear();   ①
}
```

① Simply clears the current list of breadcrumbs and bookmarks.

The Wicket viewer provides an implementation of this service, `o.a.i.viewer.wicket.viewer.services.BookmarkUiServiceWicket`.

3.3. DeepLinkService

The `DeepLinkService` provides the ability to obtain a `java.net.URI` that links to a representation of any (persisted) domain entity or view model.

A typical use case is to generate a clickable link for rendering in an email, PDF, tweet or other communication.

3.3.1. API & Implementation

The API defined by `DeepLinkService` is:

```
public interface DeepLinkService {
    URI deepLinkFor(Object domainObject); ①
}
```

① Creates a URI that can be used to obtain a representation of the provided domain object in one of the Apache Isis viewers.

The Wicket viewer provides an implementation of this service `o.a.i.viewer.wicket.viewer.services.DeepLinkServiceWicket`.

For the RestfulObjects viewer, a URL can be constructed according to the Restful Objects spec in conjunction with a `Bookmark` obtained via the `BookmarkService`.

3.3.2. Usage within the framework

The `EmailNotificationService` uses this service in order to generate emails as part of user registration.

3.4. GuiceBeanProvider

The `GuiceBeanProvider` domain service acts as a bridge between Apache Isis' Wicket viewer internal bootstrapping using Google Guice.

This service operates at a very low-level, and you are unlikely to have a need for it. It is used internally by the framework, in the default implementation of the `DeepLinkService`.

Currently Apache Isis uses a combination of Guice (within the Wicket viewer only) and a home-grown dependency injection framework. In future versions we intended to refactor the framework to use CDI throughout. At that time this service is likely to become redundant because we will allow any of the internal components of Apache Isis to be injected into your domain object code.

3.4.1. API & Implementation

The API defined by this service is:

```
public interface GuiceBeanProvider {
    @Programmatic
    <T> T lookup(Class<T> beanType);
    @Programmatic
    <T> T lookup(Class<T> beanType, final Annotation qualifier);
}
```

The Wicket viewer this provides an implementation of this service.

3.4.2. Usage

Using the Wicket viewer requires subclassing of `IsisWicketApplication`. In the subclass it is commonplace to override `newIsisWicketModule()`, for example:

```
@Override
protected Module newIsisWicketModule() {
    final Module isisDefaults = super.newIsisWicketModule();
    final Module overrides = new AbstractModule() {
        @Override
        protected void configure() {
            bind(String.class).annotatedWith(Names.named("applicationName"))
                              .toInstance("ToDo App");
            bind(String.class).annotatedWith(Names.named("applicationCss"))
                              .toInstance("css/application.css");
            bind(String.class).annotatedWith(Names.named("applicationJs"))
                              .toInstance("scripts/application.js");
            ...
        }
    };
    return Modules.override(isisDefaults).with(overrides);
}
```

This "module" is in fact a Guice module, and so the `GuiceBeanProvider` service can be used to lookup any of the components bound into it.

For example:

```
public class SomeDomainObject {
    private String lookupApplicationName() {
        return guiceBeanProvider.lookup(String.class, Names.named("applicationName"));
    }
    @Inject
    GuiceBeanProvider guiceBeanProvider;
}
```

should return "ToDo App".

Chapter 4. Presentation Layer SPI

Domain service SPIs for the presentation layer influence how the Apache Isis viewers behave.

The table below summarizes the presentation layer SPIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 2. Presentation Layer SPI

SPI	Description	Implementation	Notes
`o.a.i.applb.services.conmap.ContentMappingService`	(Attempt to) map the returned data into the representation required by the client's HTTP `Accept` header.		Replaces (and simplifies) the earlier `ContentMappingService` that defined an SPI using classes internal to the framework. + No default implementation.
`o.a.i.applib.services.userreg.EmailNotificationService`	Notify a user during self-registration of users.	`EmailNotificationService-Default` `o.a.i.core isis-core-runtime`	depends on: a configured `EmailService`
`o.a.i.applib.services.error.ErrorReportingService`	Record details of an error occurring in the system (eg in an external incident recording system such as JIRA), and return a more friendly (jargon-free) message to display to the end user, with optional reference (eg `XXX-1234`).	(none)	

SPI	Description	Implementation	Notes
o.a.i.applib.services.exceprecog.ExceptionRecognizer2	Convert certain exceptions (eg foreign or unique key violation in the database) into a format that can be rendered to the end-user.	ExceptionRecognizer-CompositeFor-JdoObjectStore o.a.i.core isis-core-applib	Extensible using composite pattern if required
o.a.i.applib.services.grid.GridSystemService	Validates and normalizes the grid layout for a domain class (with respect to a particular grid system such as Bootstrap3), also providing a default grid (for those domain classes where there is no grid layout).	GridSystemServiceBS3 o.a.i.core isis-core-metamodel	
o.a.i.applib.services.grid.GridLoaderService	Responsible for loading a grid layout for a domain class, eg from a layout.xml file.	GridLoaderServiceDefault o.a.i.core isis-core-metamodel	
o.a.i.applib.services.grid.GridService	A facade on top of both GridLoaderService and GridSystemService, thus being able to return normalized grids for any domain class.	GridServiceDefault o.a.i.core isis-core-metamodel	
o.a.i.applib.services.hint.HintStore	Stores UI hints on a per-object basis. For example, the viewer remembers which tabs are selected, and for collections which view is selected (eg table or hidden), which page of a table to render, or whether "show all" (rows) is toggled.	HintStoreUsingWicketSession o.a.i.viewer isis-viewer-wicket-impl	
o.a.i.applib.services.i18n.LocaleProvider	Request-scoped service to return the locale of the current user, in support of i18n (ie so that the app's UI, messages and exceptions can be translated to the required locale by the TranslationService.	LocaleProviderWicket o.a.i.viewer isis-viewer-wicket-impl	

SPI	Description	Implementation	Notes
o.a.i.applib.services.routing RoutingService	Return an alternative object than that returned by an action.	RoutingServiceDefault o.a.i.core isis-core-applib	The default implementation will return the home page (per HomePageProviderService) if a void or null is returned. Used by the Wicket viewer only.
o.a.i.applib.services.tablecol TableColumn-OrderService	Allows the columns of a parented or standalone table to be reordered, based upon the parent object, collection id and type of object in the collection..	TableColumn-OrderService.Default o.a.i.core isis-core-applib	
o.a.i.applib.services.i18n TranslationService	Translate an app's UI, messages and exceptions for the current user (as per the locale provided by LocalProvider.	TranslationServicePo o.a.i.core isis-core-runtime	related services: TranslationServicePoMenu depends on: TranslationsResolver, LocaleProvider
o.a.i.applib.services.i18n TranslationsResolver	Obtain translations for a particuar phrase and locale, in support of i18n (ie so that the app's UI, messages and exceptions can be translated to the required locale by the TranslationService	TranslationsResolverWicket o.a.i.viewer isis-viewer-wicket-impl	

SPI	Description	Implementation	Notes
o.a.i.applib.services.urlencoding.UrlEncodingService	Converts strings into a form safe for use within a URL. Used to convert view models mementos into usable URL form.	UrlEncodingServiceUsingBaseEncoding o.a.i.applib isis-core-applib	
o.a.i.applib.services.userprof.UserProfileService	Obtain an alternative (usually enriched/customized) name for the current user, to render in the UI.		

Key:

- o.a.i is an abbreviation for org.apache.isis
- o.ia.m is an abbreviation for org.isisaddons.module
- o.a.i.c.m.s is an abbreviation for org.apache.isis.core.metamodel.services
- o.a.i.c.r.s is an abbreviation for org.apache.isis.core.runtime.services

4.1. ContentMappingService

The ContentMappingService supports the (default implementation of the) ContentNegotiationService enabling the RestfulObjects viewer to represent domain objects in some other format as specified by the HTTP Accept header.

See ContentNegotiationService for further discussion.

Unlike most other domain services, the framework (that is, ContentNegotiationService) will check *all* available implementations of ContentMappingService to convert the domain object to the requested media type, rather than merely the first implementation found; in other words it uses the chain-of-responsibility pattern. Services are checked in the ordering defined by @DomainServiceLayout#menuOrder()). The mapped object used will be the first non-null result returned by an implementation.

4.1.1. SPI

The SPI defined by this service is:

```
public interface ContentMappingService {
    Object map(Object object,                              ①
               List<MediaType> acceptableMediaTypes);      ②
}
```

① typically the input is a domain object (whose structure might change over time), and the output is a DTO (whose structure is guaranteed to be preserved over time)

② as per the caller's HTTP `Accept` header

4.1.2. Implementations

The framework provides two implementations of this service, both in support of the `CommandService` SPI.

By way of background: implementations of the SPI `CommandService` work with custom implementations of the `Command` interface, typically being persisted to a datastore. The `CommandWithDto` interface is a subtype of `Command` for implementations that can be reified into `CommandDto` XML instances. One implementation that does this is the (non-ASF) Incode Platform's command module.

For framework implementations of `ContentMappingService` allow domain service actions that return `CommandDto`s (either singularly or in a list) to be converted into XML documents:

- `o.a.i.applib.conmap.ContentMappingServiceForCommandDto` will map any single instnce of a `CommandWithDto` into a `CommandDto` XML document
- `o.a.i.applib.conmap.ContentMappingServiceForCommandsDto` will map list of `CommandWithDto`s into a `CommandsDto` XML document, and will wrap any single instance of a `CommandWithDto` into a singleton list and thence into a `CommandsDto` XML document.

If the action invocation or property edit represent provides an implementation of a `CommandDtoProcessor` (by way of `@Action#commandDtoProcessor()` or `@Property#commandDtoProcessor()`) then this is also called to post-process the persisted `CommandDto` if required. A typical use case for this is to dynamically add in serialized `Blob`s or `Clob`s, the values of which are not captured by default in `CommandDto`.

To support the writing of custom implementations of this interface, the framework also provides `ContentMappingService.Util` which includes a couple of convenience utilities:

```
public static class Util {
    public static String determineDomainType(
        final List<MediaType> acceptableMediaTypes) { ... }
    public static boolean isSupported(
            final Class<?> clazz,
            final List<MediaType> acceptableMediaTypes) { ... }
}
```

4.1.3. Related Services

This service is a companion to the default implementation of the `ContentNegotiationService`.

The framework implementations of `ContentMappingService` use the `MetaModelService` to lookup any custom implementations of `CommandDtoProcessor`.

4.2. EmailNotificationService

The EmailNotificationService supports the user registration (self sign-up) features of the Wicket viewer whereby a user can sign-up to access an application by providing a valid email address.

The Wicket viewer will check whether an implementation of this service (and also the UserRegistrationService) is available, and if so will (unless configured not to) expose a sign-up page where the user enters their email address. A verification email is sent using this service; the email includes a link back to the running application. The user then completes the registration process (choosing a user name, password and so on) and the Wicket viewer creates an account for them (using the aforementioned UserRegistrationService).

The role of this service in all of this is to format and send out emails for the initial registration, or for password resets.

The default implementation of this service uses the EmailService, which must be configured in order for user registration to be enabled.

4.2.1. SPI

The SPI defined by this service is:

```
public interface EmailNotificationService extends Serializable {
    @Programmatic
    boolean send(EmailRegistrationEvent ev);    ①
    @Programmatic
    boolean send(PasswordResetEvent ev);        ②
    @Programmatic
    boolean isConfigured();                     ③
}
```

① sends an email to verify an email address as part of the initial user registration

② sends an email to reset a password for an already-registered user

③ determines whether the implementation was configured and initialized correctly

If isConfigured() returns false then it is *not* valid to call send(…) (and doing so will result in an IllegalStateException being thrown.

4.2.2. Implementation

The framework provides a default implementation, o.a.i.core.runtime.services.userreg.EmailNotificationServiceDefault that constructs the emails to send.

Alternative Implementations

The text of these email templates is hard-coded as resources, in other words baked into the core jar files. If you need to use different text then you can of course always write and register your own

implementation to be used instead of Isis' default.

If you have configured an alternative email service implementation, it should process the message body as HTML.

If you wish to write an alternative implementation of this service, note that (unlike most Apache Isis domain services) the implementation is also instantiated and injected by Google Guice. This is because `EmailNotificationService` is used as part of the user registration functionality and is used by Wicket pages that are accessed outside of the usual Apache Isis runtime.

This implies a couple of additional constraints:

- first, implementation class should also be annotated with `@com.google.inject.Singleton`
- second, there may not be any Apache Isis session running. (If necessary, one can be created on the fly using `IsisContext.doInSession(⋯)`)

To ensure that your alternative implementation takes the place of the default implementation, register it explicitly in `isis.properties`.

4.2.3. Related Services

As noted elsewhere, the default implementation of this service uses `EmailService`. This service has no specific configuration properties but does require that the `EmailService` has been configured.

Conversely, this service is used by (Isis' default implementation of) `UserRegistrationService`.

4.3. `ErrorReportingService`

The `ErrorReportingService` service is an optional SPI that provides the ability to record any errors/exceptions that might occur in the application into an external incident recording system (such as JIRA or Slack). The service also allows a user-friendly (jargon-free) error message to be returned and rendered to the end-user, along with an optional incident reference (eg a JIRA issue `XXX-1234`). The service can also return a URL for an external image. For example, this could be to a comic strip or (as a bit of fun) a picture of a random kitten.

4.3.1. SPI

The SPI defined by this service is:

```
public interface ErrorReportingService {
    Ticket reportError(ErrorDetails errorDetails);
}
```

where `ErrorDetails` provided to the service is:

```
public class ErrorDetails {
    public String getMainMessage() { ... }          ①
    public boolean isRecognized() { ... }           ②
    public boolean isAuthorizationCause() { ... }   ③
    public List<String> getStackTraceDetailList() { ④
}
```

① the main message to be displayed to the end-user. The service is responsible for translating this into the language of the end-user (it can use `LocaleProvider` if required).

② whether this message has already been recognized by an `ExceptionRecognizer` service. Generally this converts potentially non-recoverable (fatal) exceptions into recoverable exceptions (warnings) as well providing an alternative mechanism for generating user-friendly error messages.

③ whether the cause of the exception was due to a lack of privileges. In such cases the UI restricts the information shown to the end-user, to avoid leaking potentially sensitive information

④ the stack trace of the exception, including the trace of any exceptions in the causal chain. These technical details are hidden from the user and only shown for non-recoverable exceptions.

and `Ticket` (returned by the service) has the constructor:

```
public class Ticket implements Serializable {
    public Ticket(
        final String reference,             ①
        final String userMessage,           ②
        final String details,               ③
        final StackTracePolicy policy,      ④
        final String kittenUrl) { ... }     ⑤
}
```

① is a unique identifier that the end-user can use to track any follow-up from this error. For example, an implementation might automatically log an issue in a bug tracking system such as JIRA, in which case the reference would probably be the JIRA issue number <tt>XXX-1234</tt>.

② a short, jargon-free message to display to the end-user.

③ is optional additional details to show. For example, these might include text on how to recover from the error, or workarounds, or just further details on contacting the help desk if the issue is severe and requires immediate attention.

④ is optional, specifying whether to show a button to view details of the stack trace. By default a stack trace button is not shown if a ticket is returned (but *is* shown if there is no `ErrorReportingService` configured, or if it returns no ticket).

⑤ is optioal, specifying an external URL of an image to display for the end user.

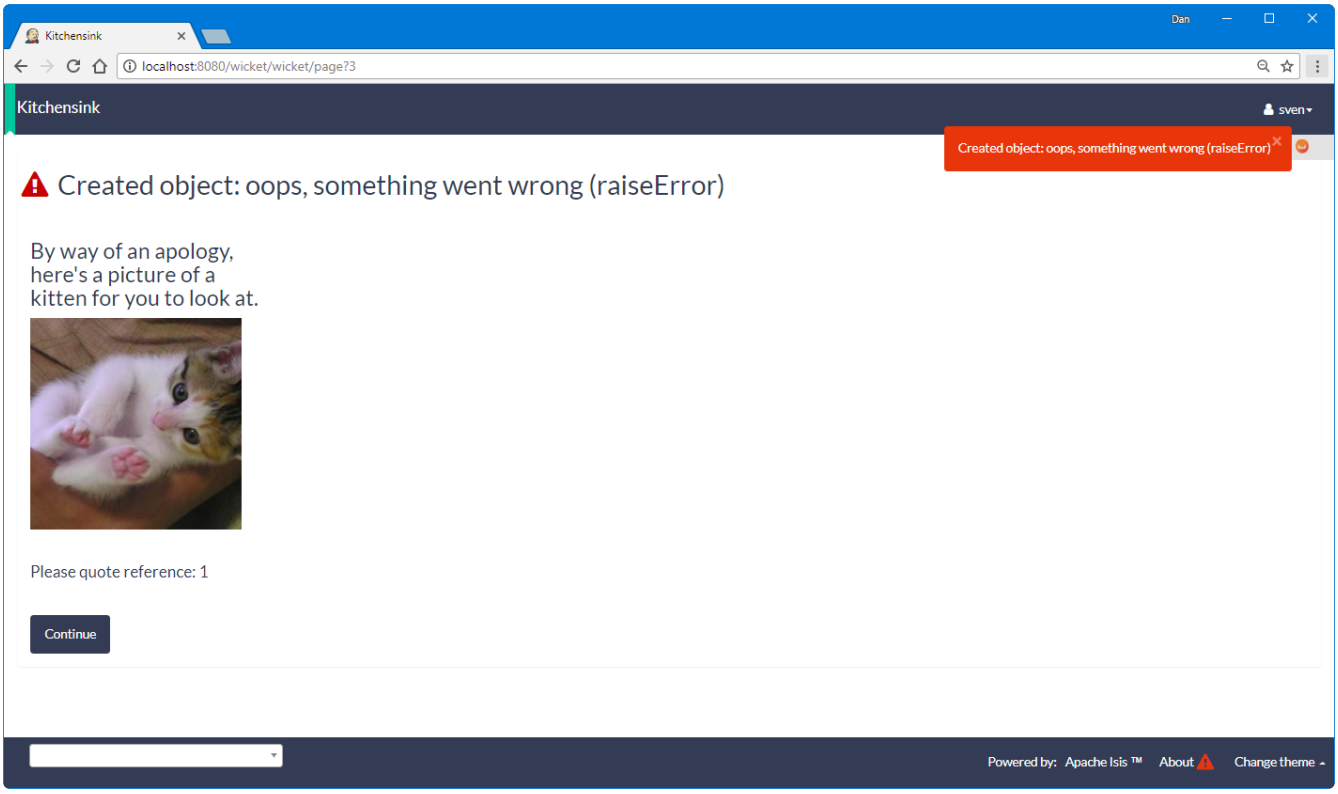
4.3.2. Implementation

There is no default implementation of this service.

The (non-ASF) Isis addons' kitchensink app provides an example implementation:

```
@DomainService( nature = NatureOfService.DOMAIN )
public class KitchensinkErrorReportingService implements ErrorReportingService {
    private int ticketNumber = 1;
    @Override
    public Ticket reportError(final ErrorDetails errorDetails) {
        return new Ticket(
                nextTicketReference(),
                errorDetails.getMainMessage(),
                  "By way of an apology, \n"
                          + "here's a picture of a \n"
                          + "kitten for you to look at.",
                "http://www.randomkittengenerator.com/cats/rotator.php"
                );
    }
    String nextTicketReference() {
        return "" + ticketNumber++;
    }
}
```

which is rendered as:



4.4. ExceptionRecognizer

The `ExceptionRecognizer` service provides the mechanism for both the domain programmer and also for components to be able to recognize and handle certain exceptions that may be thrown by the system. Rather than display an obscure error to the end-user, the application can instead

display a user-friendly message.

For example, the JDO/DataNucleus Objectstore provides a set of recognizers to recognize and handle SQL constraint exceptions such as uniqueness violations. These can then be rendered back to the user as expected errors, rather than fatal stacktraces.

It is also possible to provide additional implementations, registered in `isis.properties`. Unlike other services, where any service registered in `isis.properties` replaces any default implementations, in the case of this service all implementations registered are "consulted" to see if they recognize an exception (the chain-of-responsibility pattern).

4.4.1. SPI

The SPI defined by this service is:

```
public interface ExceptionRecognizer2 ... {
    public enum Category {                          ①
        ...
    }
    public static class Recognition {               ②
        private Category category;
        private String reason;
        ...
    }
    @Programmatic
    public Recognition recognize2(Throwable ex);    ③

    @Deprecated
    @Programmatic
    public String recognize(Throwable ex);          ④

}
```

① an enumeration of varies categories of exceptions that are recognised

② represents the fact that an exception has been recognized as has been converted into a user-friendy message, and has been categorized

③ the main API, to attempt to recognize an exception

④ deprecated API which converted exceptions into strings (reasons), ie without any categorization. This is no longer called.

The categories are:

```
public interface ExceptionRecognizer2 ... {
    public enum Category {
        CONSTRAINT_VIOLATION,           ①
        NOT_FOUND,                      ②
        CONCURRENCY,                    ③
        CLIENT_ERROR,                   ④
        SERVER_ERROR,                   ⑤
        OTHER                           ⑥
    }
    ...
}
```

① a violation of some declarative constraint (eg uniqueness or referential integrity) was detected.

② the object to be acted upon cannot be found (404)

③ a concurrency exception, in other words some other user has changed this object.

④ recognized, but for some other reason… 40x error

⑤ 50x error

⑥ recognized, but uncategorized (typically: a recognizer of the original `ExceptionRecognizer` API).

In essence, if an exception is recognized then it is also categorized. This lets the viewer act accordingly. For example, if an exception is raised from the loading of an individual object, then this is passed by the registered `ExceptionRecognizer`s. If any of these recognize the exception as representing a not-found exception, then an Apache Isis `ObjectNotFoundException` is raised. Both the viewers interprets this correctly (the Wicket viewer as a suitable error page, the Restful Objects viewer as a 404 status return code).

If the implementation recognizes the exception then it returns a user-friendly message to be rendered (by the viewer) back to the user; otherwise it returns `null`. There is no need for the implementation to check for exception causes; the casual chain is unwrapped by Apache Isis core and each exception in the chain will also be passed through to the recognizer (from most specific to least). The recognizer implementation can therefore be as fine-grained or as coarse-grained as it wishes.

4.4.2. Implementation

The framework provides two default implementations:

- `o.a.i.core.metamodel.services.container.DomainObjectContainerDefault` provided by Apache Isis core is itself an `ExceptionRecognizer`, and will handle `ConcurrencyException`s. It will also handle any application exceptions raised by the system (subclasses of `o.a.i.applib.RecoverableException`).
- `o.a.i.objectstore.jdo.applib.service.exceprecog.ExceptionRecognizerCompositeForJdoObjectStore` bundles up a number of more fine-grained implementations:
  - `ExceptionRecognizerForSQLIntegrityConstraintViolationUniqueOrIndexException`
  - `ExceptionRecognizerForJDOObjectNotFoundException`
  - `ExceptionRecognizerForJDODataStoreException`

If you want to recognize and handle additional exceptions (for example to capture error messages specific to the JDBC driver you might be using), then create a fine-grained implementation of `ExceptionRecognizer2` for the particular error message (there are some convenience implementations of the interface that you can subclass from if required) and register in `isis.properties`.

Configuration Properties

The following configuration properties are relevant:

Property	Value (default value)	Description
`isis.services.exceprecog.logRecognizedExceptions`	`true`,`false` (`false`)	whether recognized exceptions should also be logged. Generally a recognized exception is one that is expected (for example a uniqueness constraint violated in the database) and which does not represent an error condition. This property logs the exception anyway, useful for debugging. This is recognised by all implementations that subclass `ExceptionRecognizerAbstract`.
`isis.services.ExceptionRecognizerCompositeForJdoObjectStore.disable`	`true`,`false` (`false`)	whether to disable the default recognizers registered by `ExceptionRecognizerCompositeForJdoObjectStore`. This implementation provides a default set of recognizers to convert RDBMS constraints into user-friendly messages. In the (probably remote) chance that this functionality isn't required, they can be disabled through this flag.

4.5. `GridLoaderService`

The `GridLoaderService` provides the ability to load the XML layout (grid) for a domain class.

4.5.1. SPI

The SPI defined by this service is:

```
public interface GridLoaderService {
    boolean supportsReloading();                ①
    void remove(Class<?> domainClass);          ②
    boolean existsFor(Class<?> domainClass);    ③
    Grid load(final Class<?> domainClass);      ④
}
```

① whether dynamic reloading of layouts is enabled. The default implementation enables reloading for prototyping, disables in production

② support metamodel invalidation/rebuilding of spec, eg as called by this Object mixin action.

③ whether any persisted layout metadata (eg a `.layout.xml` file) exists for this domain class.

④ returns a new instance of a `Grid` for the specified domain class, eg as loaded from a `layout.xml` file. If none exists, will return null (and the calling `GridService` will use `GridSystemService` to obtain a default grid for the domain class).

4.5.2. Implementation

The framework provides a default implementation of this service, namely `GridLoaderServiceDefault`. This implementation loads the grid from its serialized representation as a `.layout.xml` file, loaded from the classpath.

For example, the layout for a domain class `com.mycompany.myapp.Customer` would be loaded from `com/mycompany/myapp/Customer.layout.xml`.

4.5.3. Related Services

This service is used by `GridService`.

4.6. `GridService`

The `GridService` provides the ability to load the XML layout (grid) for a domain class. To do this it delegates:

- to `GridLoaderService` to load a pre-existing layout for the domain class, if possible
- to `GridSystemService` to normalize the grid with respect to Apache Isis' internal metamodel, in other words to ensure that all of the domain objects' properties, collections and actions are associated with regions of the grid.

Once a grid has been loaded for a domain class, this is cached internally by Apache Isis' internal meta model (in the `GridFacet` facet). If running in prototype mode, any subsequent changes to the XML will be detected and the grid rebuilt. This allows for dynamic reloading of layouts, providing a far faster feedback (eg if tweaking the UI while working with end-users). Dynamic reloading is disabled in production mode.

4.6.1. SPI

The SPI defined by this service is:

```
public interface GridService {
    boolean supportsReloading();                ①
    void remove(Class<?> domainClass);          ②
    boolean existsFor(Class<?> domainClass);    ③
    Grid load(final Class<?> domainClass);      ④
    Grid defaultGridFor(Class<?> domainClass);  ⑤
    Grid normalize(final Grid grid);            ⑥
    Grid complete(Grid grid);                   ⑦
    Grid minimal(Grid grid);                    ⑧
}
```

① whether dynamic reloading of layouts is enabled. The default implementation enables reloading for prototyping, disables in production

② support metamodel invalidation/rebuilding of spec, eg as called by this Object mixin action.

③ whether any persisted layout metadata (eg a `.layout.xml` file) exists for this domain class. Just delegates to corresponding method in `GridLoaderService`.

④ returns a new instance of a `Grid` for the specified domain class, eg as loaded from a `layout.xml` file. If none exists, will return null (and the calling `GridService` will use `GridSystemService` to obtain a default grid for the domain class).

⑤ returns a default grid, eg two columns in ratio 4:8. Used when no existing grid layout exists for a domain class.

⑥ validates and normalizes a grid, modifying the grid so that all of the domain object's members (properties, collections, actions) are bound to regions of the grid. This is done using existing metadata, most notably that of the `@MemberOrder` annotation. Such a grid, if persisted as the layout XML file for the domain class, allows the `@MemberOrder` annotation to be removed from the source code of the domain class (but other annotations must be retained).

⑦ Takes a normalized grid and enriches it with additional metadata (taken from Apache Isis' internal metadata) that can be represented in the layout XML. Such a grid, if persisted as the layout XML file for the domain class, allows all layout annotations (`@ActionLayout`, `@PropertyLayout` and `@CollectionLayout`) to be removed from the source code of the domain class.

⑧ Takes a normalized grid and strips out removes all members, leaving only the grid structure. Such a grid, if persisted as the layout XML file for the domain class, requires that the `@MemberOrder` annotation is retained in the source code of said class in order to bind members to the regions of the grid.

The first four methods just delegate to the corresponding methods in `GridSystemService`, while the last four delegate to the corresponding method in `GridSystemService`. The service inspects the `Grid`'s concrete class to determine which actual `GridSystemService` instance to delegate to.

4.6.2. Implementation

The framework provides a default implementation of this service, namely `GridServiceDefault`.

4.6.3. Related Services

This service calls `GridLoaderService` and `GridSystemService`.

This service is called by `LayoutService`, exposed in the UI through `LayoutServiceMenu` (to download the layout XML as a zip file for all domain objects) and the `downloadLayoutXml()` mixin (to download the layout XML for a single domain object).

4.7. `GridSystemService`

The `GridSystemService` encapsulates a single layout grid system which can be used to customize the layout of domain objects. In particular this means being able to return a "normalized" form (validating and associating domain object members into the various regions of the grid) and in providing a default grid if there is no other metadata available.

The framework provides a single such grid implementation, namely for Bootstrap3.

Unlike most other domain services, the framework will check *all* available implementations of `GridSystemService` to obtain available grid systems, rather than merely the first implementation found; in other words it uses the chain-of-responsibility pattern. Services are called in the order defined by `@DomainServiceLayout#menuOrder()`).

Note though that each concrete implementation must also provide corresponding Wicket viewer components capable of interpreting the grid layout.

4.7.1. SPI

The SPI defined by this service is:

```
public interface GridSystemService<G extends Grid> {
    Class<? extends Grid> gridImplementation();                 ①
    String tns();                                               ②
    String schemaLocation();                                    ③
    Grid defaultGrid(Class<?> domainClass);                     ④
    void normalize(G grid, Class<?> domainClass);               ⑤
    void complete(G grid, Class<?> domainClass);                ⑥
    void minimal(G grid, Class<?> domainClass);                 ⑦
}
```

① The concrete subclass of `Grid` supported by this implementation. As noted in the introduction, there can be multiple implementations of this service, but there can only be one implementation per concrete subclass. As is normal practice, the service with the lowest `@DomainServiceLayout#menuOrder()` takes precedence.

② the target namespace for this grid system. This is used when generating the XML. The Bootstrap3 grid system provided by the framework returns the value `http://isis.apache.org/applib/layout/grid/bootstrap3`.

③ the schema location for the XSD. The Bootstrap3 grid system provided by the framework returns the value `http://isis.apache.org/applib/layout/grid/bootstrap3/bootstrap3.xsd`.

④ a default grid, eg two columns in ratio 4:8. Used when no existing grid layout exists for a domain class.

⑤ Validates and normalizes a grid, modifying the grid so that all of the domain object's members (properties, collections, actions) are bound to regions of the grid. This is done using existing metadata, most notably that of the `@MemberOrder` annotation. Such a grid, if persisted as the layout XML file for the domain class, allows the `@MemberOrder` annotation to be removed from the source code of the domain class (but other annotations must be retained).

⑥ Takes a normalized grid and enriches it with additional metadata (taken from Apache Isis' internal metadata) that can be represented in the layout XML. Such a grid, if persisted as the layout XML file for the domain class, allows all layout annotations (`@ActionLayout`, `@PropertyLayout` and `@CollectionLayout`) to be removed from the source code of the domain class.

⑦ Takes a normalized grid and strips out removes all members, leaving only the grid structure. Such a grid, if persisted as the layout XML file for the domain class, requires that the `@MemberOrder` annotation is retained in the source code of said class in order to bind members to the regions of the grid.

4.7.2. Implementation

The framework provides `GridSystemServiceBS3`, an implementation that encodes the bootstrap3 grid system. (The framework also provides Wicket viewer components that are capable of interpreting and rendering this metadata).

4.7.3. Related Services

This service is used by `GridService`.

4.8. `HintStore`

The `HintStore` service defines an SPI for the Wicket viewer to store UI hints on a per-object basis. For example, the viewer remembers which tabs are selected, and for collections which view is selected (eg table or hidden), which page of a table to render, or whether "show all" (rows) is toggled.

The default implementation of this service uses the HTTP session. The service is an SPI because the amount of data stored could potentially be quite large (for large numbers of users who use the app all day). An SPI makes it easy to plug in an alternative implementation that is more sophisticated than the default (eg implementing MRU/LRU queue, or using a NoSQL database, or simply to disabling the functionality altogether).

4.8.1. SPI

The SPI of `HintStore` is:

```
public interface HintStore {
    String get(final Bookmark bookmark, String hintKey);                ①
    void set(final Bookmark bookmark, String hintKey, String value);    ②
    void remove(final Bookmark bookmark, String hintKey);               ③
    void removeAll(Bookmark bookmark);                                  ④
    Set<String> findHintKeys(Bookmark bookmark);                        ⑤
}
```

① obtain a hint (eg which tab to open) for a particular object. Object identity is represented by `Bookmark`, as per the `BookmarkService`, so that alternative implementations can easily serialize this state to a string.

② set the state of a hint. (The value of) all hints are represented as strings.

③ remove a single hint for an object;

④ remove all hints

⑤ obtain all known hints for an object

4.8.2. Implementation

The core framework provides a default implementation of this service (`org.apache.isis.viewer.wicket.viewer.services.HintStoreUsingWicketSession`).

4.8.3. View models

Hints are stored against the `Bookmark` of a domain object, essentially the identifier of the domain object. For a domain entity this identifier is fixed and unchanging but for a view models the identifier changes each time the view model's state changes (the identifier is basically a digest of the object's state). This means that any hints stored against the view model's bookmark are in effect lost as soon as the view model is modified.

To address this issue the `HintStore` provides an optional interface that the view model can implement, the intent of which is to expose the "logical" identity of the view model. This interface is:

```
public interface HintStore {
    interface HintIdProvider {
        String hintId();
    }
}
```

For example, suppose that there's a view model that wraps a `Customer` and its `Order`s For this the `Customer` represents the logical identity. This view model might therefore be implemented as follows:

```
@XmlRootElement("customerAndOrders")
@XmlAccessType(FIELD)
public class CustomerAndOrders implements HintStore.HintIdProvider {

    @Getter @Setter
    private Customer customer;

    ...

    @Programmatic
    public String hintId() {
        bookmarkService.bookmarkFor(getCustomer()).toString();
    }

    @XmlTransient
    @Inject BookmarkService bookmarkService;
}
```

4.8.4. Related Services

The Wicket viewer exposes the "clear hints" mixin action that is for use by end-users of the application to clear any UI hints that have accumulated for a domain object.

4.9. LocaleProvider

The `LocaleProvider` service is one of the services that work together to implement Apache Isis' support for i18n, being used by Isis' default implementation of `TranslationService`.

The role of the service itself is simply to return the `Locale` of the current user.

For the "big picture" and further details on Apache Isis' i18n support, see here.

4.9.1. SPI

The SPI defined by this service is:

```
public interface LocaleProvider {
    @Programmatic
    Locale getLocale();
}
```

This is notionally request-scoped, returning the `Locale` of the current user; *not* that of the server. (Note that the implementation is not required to actually be `@RequestScoped`, however).

4.9.2. Implementation

Isis' Wicket viewer provides an implementation of this service (`LocaleProviderWicket`) which

leverages Apache Wicket APIs.

Currently there is no equivalent implementation for the RestfulObjects viewer.

4.9.3. Related Services

This service works in conjunction with `TranslationService` and `TranslationsResolver` in order to provide i18n support.

4.10. `MenuBarsLoaderService`

The `MenuBarsLoaderService` is used by the default implementation of MenuBarsService to return a a `MenuBars` instance deserialized from the `menubars.layout.xml` file read from the classpath.

4.10.1. SPI and Implementation

The SPI defined by this service is:

```
public interface MenuBarsLoaderService {
    boolean supportsReloading();        ①
    MenuBars menuBars();                ②
}
```

① Whether dynamic reloading of the menu bars layout is enabled. If not, then the MenuBarsService will cache the layout once loaded.

② Returns a new instance of `MenuBars` if possible, otherwise null.

The framework provides a default implementation of this service, namely `o.a.i.core.runtime.services.menu.MenuBarsLayoutServiceDefault`.

This searches for a file resource `menubars.layout.xml`, expected to reside in the same package as the `AppManifest` used to bootstrap the application.

It supports reloading only in prototype mode.

4.11. `MenuBarsService`

The `MenuBarsService` is responsible for returning a `MenuBars` instance, a data structure representing the arrangement of domain service actions across multiple menu bars, menus and sections. This is used by the Wicket viewer to build up the menu, and is also served as the "menuBars" resource by the Restful Objects viewer.

4.11.1. SPI and Implementation

The SPI defined by this service is:

```
public interface MenuBarsService {
    enum Strategy {                          ①
        DEFAULT,
        FALLBACK
    }
    MenuBars menuBars();                     ②
    MenuBars menuBars(Strategy strategy);    ③
}
```

① Select whether to return the "default" `MenuBars` instance - which may be obtained from anywhere, eg read from the classpath, or to "fallback"" and derive from the metamodel facet/annotations.

② Convenience API to return the default `MenuBars` instance

③ Returns an instance of `MenuBars` according the specified strategy.

The framework provides a default implementation of this service, namely `o.a.i.core.runtime.services.menu.MenuBarsServiceDefault`. This uses the MenuBarsLoaderService to load a serialized form of `MenuBars` instance, called `menubars.layout.xml`, from the classpath.

4.12. RoutingService

The `RoutingService` provides the ability to return (and therefore render) an alternative object from an action invocation.

There are two primary use cases:

- if an action returns an aggregate leaf (that is, a child object which has an owning parent), then the parent object can be returned instead.

  For example, an action returning `OrderItem` might instead render the owning `Order` object. It is the responsibility of the implementation to figure out what the "owning" object might be.

- if an action returns `null` or is `void`, then return some other "useful" object.

  For example, return the home page (eg as defined by the `@HomePage` annotation).

Currently the routing service is used only by the Wicket viewer; it is ignored by the Restful Objects viewer.

Unlike most other domain services, the framework will check *all* available implementations of `RoutingService` to return a route, rather than the first implementation found; in other words it uses the chain-of-responsibility pattern. Services are called in the order defined by `@DomainServiceLayout#menuOrder()`). The route used will be the result of the first implementation checked that declares that it can provide a route.

4.12.1. SPI

The SPI defined by this service is:

```
public interface RoutingService {
    @Programmatic
    boolean canRoute(Object original);  ①
    @Programmatic
    Object route(Object original);      ②
}
```

① whether this implementation recognizes and can "route" the object. The `route(…)` method is only called if this method returns `true`.

② the object to use; this may be the same as the original object, some other object, or (indeed) `null`.

4.12.2. Implementation

The framework provides a default implementation - `RoutingServiceDefault` - which will always return the original object provided, or the home page if a `null` or `void` was provided. It uses the `HomePageProviderService`.

There can be multiple implementations of `RoutingService` registered. These are checked in turn (chain of responsibility pattern), ordered according to `@DomainServiceLayout#menuOrder()` (as explained in the introduction to this guide). The route from the first service that returns `true` from its `canRoute(…)` method will be used.

4.12.3. Related Services

The default implementation of this service uses the `HomePageProviderService`.

4.13. `SessionLoggingService`

The `SessionLoggingService` defines an SPI to keep track of (typically: to log) the current sessions that are using the application.

4.13.1. SPI

The SPI defined by this service is:

```
public interface SessionLoggingService {
    public enum Type { LOGIN, LOGOUT }
    public enum CausedBy { USER, SESSION_EXPIRATION, RESTART }

    void log(
        Type type,
        String username,
        Date date,
        CausedBy causedBy,
        String sessionId);  ①
}
```

① an internal identifier (the JVM hashCode of the Wicket session).

4.13.2. Implementations

The framework provides an implementation, `SessionLoggingService.Stderr` that just prints out to standard error. This is not registered by default, but can be easily registered manually using `AppManifestAbstract.Builder#withAdditionalServices(…)`.

The (non-ASF) Incode Platform's sessionlogger module provides an implementation that logs each session as a JDO entity.

4.14. `TableColumnOrderService`

The `TableColumnOrderService` provides the ability to reorder (or suppress) columns in both parented- and standalone tables.

4.14.1. SPI

The SPI defined by this service is:

```
public interface TableColumnOrderService {
    List<String> orderParented(                  ①
            Object parent,
            String collectionId,
            Class<?> collectionType,
            List<String> propertyIds);
    List<String> orderStandalone(                ②
            Class<?> collectionType,
            List<String> propertyIds);
}
```

① for the parent collection owned by the specified parent and collection Id, return the set of property ids in the same or other order.

② for the standalone collection of the specified type, return the set of property ids in the same or other order, else return `null` if provides no reordering.

There can be multiple implementations of `TableColumnOrderService` registered, ordered as per `@DomainServiceLayout#menuOrder()`. The ordering provided by the first such service that returns a non-`null` value will be used. If all provided implementations return `null`, then the framework will fallback to a default implementation.

4.14.2. Implementation

The framework provides a fallback implementation of this service, namely `TableColumnOrderService.Default`.

4.15. `TranslationService`

The `TranslationService` is the cornerstone of Apache Isis' i18n support. Its role is to be able to provide translated versions of the various elements within the Apache Isis metamodel (service and object classes, properties, collections, actions, action parameters) and also to translate business rule (disable/valid) messages, and exceptions. These translations provide for both singular and plural forms.

For the "big picture" and further details on Apache Isis' i18n support, see here.

4.15.1. SPI

The SPI defined by this service is:

```
public interface TranslationService {
    @Programmatic
    String translate(String context, String text);      ①
    @Programmatic
    String translate(String context,                    ②
                     String singularText,
                     String pluralText, int num);

    enum Mode { READ, WRITE;}
    @Programmatic
    Mode getMode();                                     ③
}
```

① translate the text, in the locale of the "current user".

② return a translation of either the singular or the plural text, dependent on the `num` parameter, in the locale of the "current user"

③ whether this implementation is operating in read or in write mode.

If in read mode, then the translations are expected to be present.

If in write mode, then the implementation is saving translation keys, and will always return the untranslated translation.

4.15.2. Implementation

The Apache Isis framework provides a default implementation (`TranslationServicePo`) that uses the GNU `.pot` and `.po` files for translations. It relies on the `LocaleProvider` service (to return the `Locale` of the current user) and also the `TranslationsResolver` service (to read existing translations).

The framework also provides a supporting `TranslationServicePoMenu` provides menu items under the "Prototyping" secondary menu for controlling this service and downloading `.pot` files for translation.

If the menu items are not required then these can be suppressed either using security or by implementing a vetoing subscriber.

For more details on the implementation, see i18n support.

Configuration Properties

The default implementation of this domain service recognises the following configuration properties:

Property	Value (default value)	Description
`isis.services.translation.po.mode`	`read`,`write`	Whether to force the `TranslationService` into either read or write mode. See i18n support to learn more about the translation service.

4.15.3. Related Menus

The `TranslationServicePoMenu` menu exposes the `TranslationServicePo` service's `toPot()` method so that all translations can be downloaded as a single file.

4.15.4. Related Services

This service works in conjunction with `LocaleProvider` and `TranslationsResolver` in order to provide i18n support.

4.16. `TranslationsResolver`

The `TranslationsResolver` service is one of the services that work together to implement Apache Isis' support for i18n, being used by Isis' default implementation of `TranslationService`.

The role of the service itself is locate and return translations.

For the "big picture" and further details on Apache Isis' i18n support, see here.

4.16.1. SPI

The SPI defined by this service is:

```
public interface TranslationsResolver {
    @Programmatic
    List<String> readLines(final String file);
}
```

4.16.2. Implementation

Isis' Wicket viewer provides an implementation of this service (`TranslationsResolverWicket`) which leverages Apache Wicket APIs. This searches for translation files in the standard `WEB-INF/` directory.

Currently there is no equivalent implementation for the RestfulObjects viewer.

4.16.3. Related Services

This service works in conjunction with `LocaleProvider` and `TranslationService` in order to provide i18n support.

4.17. `UrlEncodingService`

The `UrlEncodingService` defines a consistent way to convert strings to/from a form safe for use within a URL. The service is used by the framework to map view model mementos (derived from the state of the view model itself) into a form that can be used as a view model. When the framework needs to recreate the view model (for example to invoke an action on it), this URL is converted back into a view model memento, from which the view model can then be hydrated.

Defining this functionality as an SPI has two use cases:

- first, (though some browsers support longer strings), there is a limit of 2083 characters for URLs. For view model mementos that correspond to large strings (as might occur when serializing a JAXB `@XmlRootElement`-annotated view model), the service provides a hook.

  For example, each memento string could be mapped to a GUID held in some cluster-aware cache.

- the service provides the ability, to encrypt the string in order to avoid leakage of potentially sensitive state within the URL.

The framework provides a default implementation of this service, `UrlEncodingServiceUsingBaseEncoding` (also in the applib) that uses `base-64` encoding to `UTF-8` charset.

4.17.1. SPI

The SPI defined by the service is:

```
public interface UrlEncodingService {
    @Programmatic
    public String encode(final String str);     ①
    @Programmatic
    public String decode(String str);           ②
}
```

① convert the string (eg view model memento) into a string safe for use within an URL

② unconvert the string from its URL form into its original form URL

4.17.2. Implementation

The framework provides a default implementation (`UrlEncodingServiceUsingBaseEncoding`) that simply converts the string using base-64 encoding and UTF-8 character set. As already noted, be aware that the maximum length of a URL should not exceed 2083 characters. For large view models, there's the possibility that this limit could be exceeded; in such cases register an alternative implementation of this service.

To use an alternative implementation, use `@DomainServiceLayout#menuOrder()` (as explained in the introduction to this guide).

4.18. `UserProfileService`

The `UserProfileService` provides the ability for the domain application to return supplementary metadata about the current user. This information is used (by the Wicket viewer) to customize the appearance of the tertiary "Me" menu bar (top right). For example, rather than display the username, instead the user's first and last name could be displayed.

Another use case is to allow the user to switch context in some fashion or other. This might be to emulate a sort of "sudo"-like function, or perhaps to focus on some particular set of data.

4.18.1. SPI

The SPI defined by the service is:

```
public interface UserProfileService {
    @Programmatic
    String userProfileName();       ①
}
```

① is used (in the Wicket viewer) as the menu name of the tertiary "Me" menu bar.

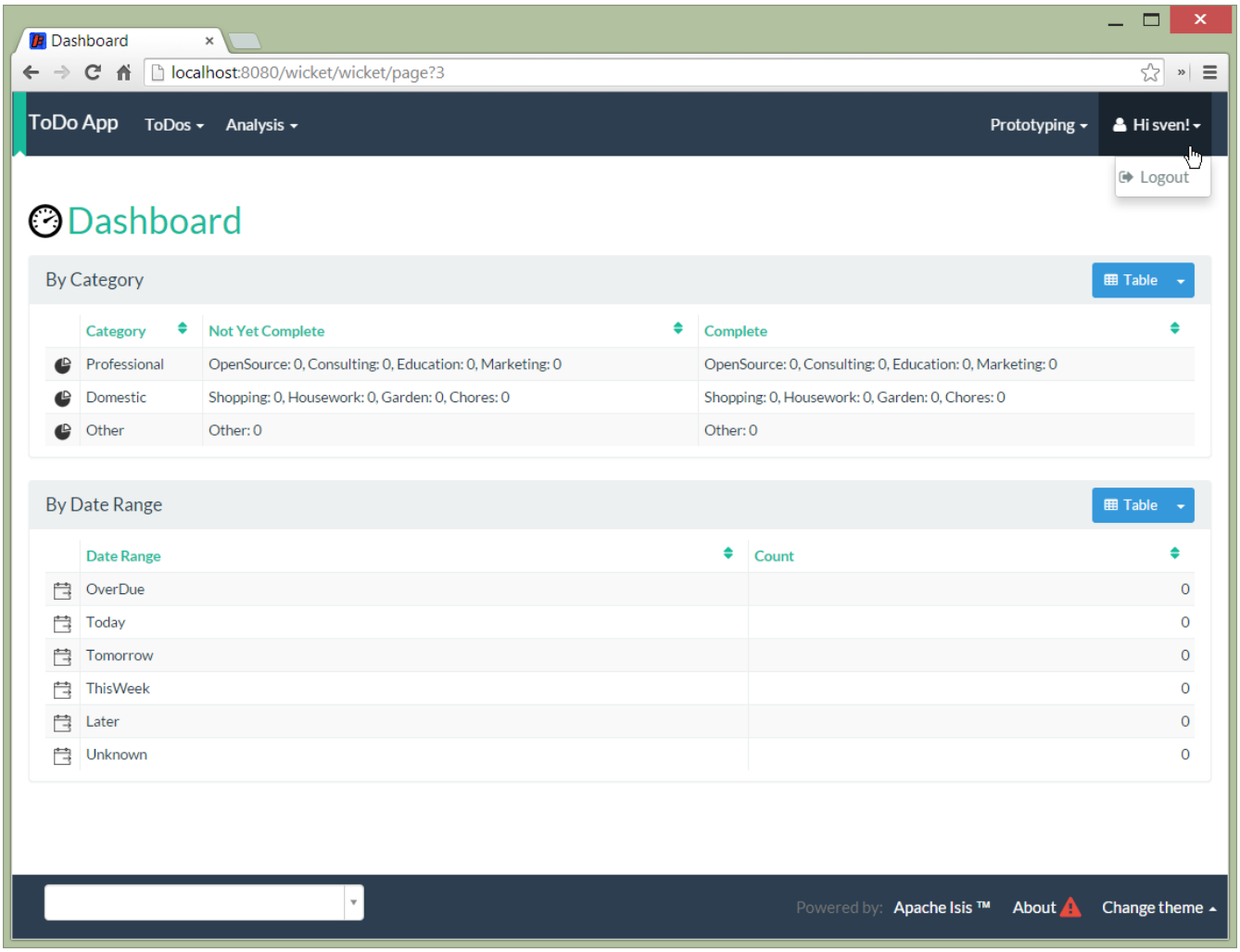
If the method returns `null` or throws an exception then the framework will default to using the current user name.

In the future this API may be expanded; one obvious possibility is to return a profile photo or avatar URL.

4.18.2. Implementation

The framework provides a default implementation of this service, `o.a.i.core.runtime.services.UserProfileServiceDefault`. This simply returns the user's name as the user's profile name.

An example implementation can also be found in the (non-ASF) Isis addons' todoapp:



This feature does not integrate with Apache Isis' authentication mechanisms; the information returned is used purely for presentation purposes.

Chapter 5. Application Layer API

Domain service APIs for the application layer allow the domain objects to control aspects of the application layer, such as sending info messages back to the end-user.

The table below summarizes the application layer APIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 3. Application Layer API

API	Description	Implementation	Notes
`o.a.i.applib.services.actinv.ActionInvocation-Context`	Request-scoped access to whether action is invoked on object and/or on collection of objects	`ActionInvocationContext` `o.a.i.core:isis-core-applib`	API is also concrete class
`o.a.i.applib.services.background.BackgroundService`	Programmatic persistence of commands to be persisted (so can be executed by a background mechanism, eg scheduler)	`BackgroundService-Default` `o.a.i.core:isis-core-runtime`	depends on: `BackgroundCommand-Service`
`o.a.i.applib.services.command.CommandContext`	Request-scoped access to capture the users's *intention* to invoke an action or to edit a property.	`CommandContext` `o.a.i.core:isis-core-applib`	API is also a concrete class. depends on: `CommandService` for persistent `Command`, else in-memory impl. used. The `InteractionContext` manages the actual execution of the command.
`o.a.i.applib.services.command.CommandExecutorService`	Executes the specified `Command`.	`CommandExecutorService- Default` `o.a.i.core:isis-core-runtime`	

API	Description	Implementation	Notes
o.a.i.applib.services.dto.DtoMappingHelper	Maps domain objects internal identifier to an OidDto for use in serialized representations.	DtoMappingHelper o.a.i.core isis-core-applib	API is also a concrete class.
o.a.i.applib.services.iactn.InteractionContext	Request-scoped access to the current member execution (action invocation or property edit), represented as the Interaction context.	InteractionContext o.a.i.core isis-core-applib	API is also a concrete class.
o.a.i.applib.services.message.MessageService	Methods to inform or warn the user, or to raise errors.	FactoryService-Default o.a.i.core isis-core-metamodel	Supercedes methods in DomainObjectContainer.
o.a.i.applib.services.sessmgmt.SessionManagementService	Methods for batching long-running work (eg data migration) into multiple sessions.	SessionManagementService-Default o.a.i.core isis-core-runtime	
o.a.i.applib.services.title.TitleService	Methods to programmatically obtain the title or icon of a domain object.	TitleService-Default o.a.i.core isis-core-metamodel	Supercedes methods in DomainObjectContainer.
o.a.i.applib.services.xactn.TransactionService	Methods for managing transactions.	TransactionService-Default o.a.i.core isis-core-metamodel	Supercedes methods in DomainObjectContainer.
o.a.i.applib.services.wrapper.WrapperFactory	Interact with another domain object "as if" through the UI (enforcing business rules, firing domain events)	WrapperFactoryDefault o.a.i.core isis-core-wrapper	

Key:

- o.a.i is an abbreviation for org.apache.isis
- o.ia.m is an abbreviation for org.isisaddons.module
- o.a.i.c.m.s is an abbreviation for org.apache.isis.core.metamodel.services

- `o.a.i.c.r.s` is an abbreviation for `org.apache.isis.core.runtime.services`

5.1. `ActionInvocationContext` (deprecated)

The `ActionInvocationContext` domain service is a `@RequestScoped` service intended to support the implementation of "bulk" actions annotated with `@Action#invokeOn()`. This allows the user to select multiple objects in a table and then invoke the same action against all of them.

When an action is invoked in this way, this service allows each object instance to "know where it is" in the collection; it acts a little like an iterator. In particular, an object can determine if it is the last object to be called, and so can perform special processing, eg to return a summary calculated result.

Bulk actions are now deprecated, which means that this service is also deprecated.

Instead, the recommended technique is to define a view model to wrap around the collection, and then to use an action on the view model, associated with the collection and with a collection parameter), to act upon the selected items of the collection.

5.1.1. API & Implementation

The API defined by the service is:

```
@DomainService(nature = NatureOfService.DOMAIN)
@RequestScoped                                          ①
public static class ActionInvocationContext {
    public InvokedOn getInvokedOn() { ... }             ②
    public List<Object> getDomainObjects() { ... }      ③
    public int getSize() { ... }
    public int getIndex() { ... }                       ④
    public boolean isFirst() { ... }
    public boolean isLast() { ... }
}
```

① is `@RequestScoped`, so this domain service instance is scoped to a particular request and is then destroyed

② an enum set to either `OBJECT` (if action has been invoked on a single object) or `COLLECTION` (if has been invoked on a collection).

③ returns the list of domain objects which are being acted upon

④ is the 0-based index to the object being acted upon.

5.1.2. Usage

For actions that are void or that return null, Apache Isis will return to the list once executed. But for bulk actions that are non-void, Apache Isis will render the returned object/value from the last object invoked (and simply discards the object/value of all actions except the last).

One idiom is for the domain objects to also use the `Scratchpad` service to share information, for example to aggregate values. The `ActionInvocationContext#isLast()` method can then be used to determine if all the information has been gathered, and then do something with it (eg derive variance across a range of values, render a graph etc).

More prosaically, the `ActionInvocationContext` can be used to ensure that the action behaves appropriately depending on how it has been invoked (on a single object and/or a collection) whether it is called in bulk mode or regular mode. Here's a snippet of code from the bulk action in the Isis addon example todoapp (not ASF):

```
public class ToDoItem ... {
    @Action(invokeOn=InvokeOn.OBJECTS_AND_COLLECTIONS)
    public ToDoItem completed() {
        setComplete(true);
        ...
        return actionInvocationContext.getInvokedOn() == InvokedOn.OBJECT
                ? this  ①
                : null; ②
    }
    @Inject
    ActionInvocationContext actionInvocationContext;
}
```

① if invoked as a regular action, return this object;

② otherwise (if invoked on collection of objects), return null, so that the Wicket viewer will re-render the list of objects

5.1.3. Unit testing support

The `ActionInvocationContext` class also has a couple of static factory methods intended to support unit testing:

```
@DomainService(nature = NatureOfService.DOMAIN)
@RequestScoped
public class ActionInvocationContext {
    public static ActionInvocationContext onObject(final Object domainObject) {
        return new ActionInvocationContext(InvokedOn.OBJECT, Collections.
singletonList(domainObject));
    }
    public static ActionInvocationContext onCollection(final Object... domainObjects)
{
        return onCollection(Arrays.asList(domainObjects));
    }
    public static ActionInvocationContext onCollection(final List<Object>
domainObjects) {
        return new ActionInvocationContext(InvokedOn.COLLECTION, domainObjects);
    }
    ...
}
```

5.2. BackgroundService2

The `BackgroundService2` domain service (and its various supertypes), and also the companion `BackgroundCommandService2` SPI service, enable commands to be persisted such that they may be invoked in the background.

The `BackgroundService2` is responsible for capturing a memento representing the command in a typesafe way, and persisting it rather than executing it directly.

The default `BackgroundServiceDefault` implementation works by using a proxy wrapper around the target so that it can capture the action to invoke and its arguments. This is done using `CommandDtoServiceInternal`.

The persistence delegates the persistence of the memento to an appropriate implementation of the companion `BackgroundCommandService2`. One such implementation of `BackgroundCommandService` is provided by (non-ASF) Isis addons' command module.

The persisting of commands is only half the story; there needs to be a separate process to read the commands and execute them. The `BackgroundCommandExecution` abstract class (discussed below) provides infrastructure to do this; the concrete implementation of this class depends on the configured `BackgroundCommandService` (in order to query for the persisted (background) `Command`s.

5.2.1. API & Implementation

The API is:

```
public interface BackgroundService2 {
    <T> T execute(final T object);                          ①
    <T> T executeMixin(Class<T> mixinClass, Object mixedIn);    ②
}
```

① returns a proxy around the domain object; any methods executed against this proxy will result in a command (to invoke the corresponding action) being persisted by `BackgroundCommandService2`

② Returns a proxy around the mixin; any methods executed against this proxy will result in a command (to invoke the corresponding mixin action) being persisted by `BackgroundCommandService2`.

The default implementation is provided by core (`o.a.i.core.runtime.services.background.BackgroundServiceDefault`).

5.2.2. Usage

Using the service is very straight-forward; wrap the target domain object using `BackgroundService#execute(…)` and invoke the method on the object returned by that method.

For example:

```
public void submitCustomerInvoices() {
    for(Customer customer: customerRepository.findCustomersToInvoice()) {
        backgroundService.execute(customer).submitInvoice();
    }
    messageService.informUser("Calculating...");
}
```

This will create a bunch of background commands executing the `submitInvoice()` action for each of the customers returned from the customer repository.

The action method invoked must be part of the Apache Isis metamodel, which is to say it must be public, accept only scalar arguments, and must not be annotated with `@Programmatic` or `@Ignore`. However, it may be annotated with `@Action#hidden()` or `@ActionLayout#hidden()` and it will still be invoked.

In fact, when invoked by the background service, no business rules (hidden, disabled, validation) are enforced; the action method must take responsibility for performing appropriate validation and error checking.

If you want to check business rules, you can use `@WrapperFactory#wrapNoExecute(…)`.

5.2.3. End-user experience

For the end-user, executing an action that delegates work off to the `BackgroundService` raises the

problem of how does the user know the work is complete?

One option is for the background jobs to take responsibility to notify the user themselves. In the above example, this would be the `submitInvoice()` method called upon each customer. One could imagine more complex designs where only the final command executed notifies the user.

However, an alternative is to rely on the fact that the `BackgroundService` will automatically hint that the `Command` representing the original interaction (to `submitCustomerInvoices()` in the example above) should be persisted. This will be available if the related `CommandContext` and `CommandService` domain services are configured, and the `CommandService` supports persistent commands. Note that (non-ASF) Incode Platform's command module does indeed provide such an implementation of `CommandService` (as well as of the required `BackgroundCommandService`).

Thus, the original action can run a query to obtain it corresponding `Command`, and return this to the user. The upshot is that the child `Command`s created by the `BackgroundService` will then be associated with `Command` for the original action.

We could if we wanted write the above example as follows:

```
public Command submitCustomerInvoices() {
    for(Customer customer: customerRepository.findCustomersToInvoice()) {
        backgroundService.execute(customer).submitInvoice();
    }
    return commandContext.getCommand();
}
@Inject
CommandContext commandContext;  ①
```

① the injected `CommandContext` domain service.

The user would be returned a domain object representing their action invocation.

5.2.4. Related Services

This service is closely related to the `CommandContext` and also that service's supporting `CommandService` service.

The `CommandContext` service is responsible for providing a parent `Command` with which the background `Command`s can then be associated as children, while the `CommandService` is responsible for persisting those parent `Command`s. The latter is analogous to the way in which the `BackgroundCommandService` persists the child background `Command`s.

The implementations of `CommandService` and `BackgroundCommandService` go together; typically both parent `Command`s and child background `Command`s will be persisted in the same way. The (non-ASF) Incode Platform's command module provides implementations of both (see `CommandService` and `BackgroundCommandService`).

The `CommandDtoServiceInternal` is used to obtain a memento of the command such that it can be persisted. (In earlier versions, `MementoService` was used for this purpose).

5.2.5. BackgroundCommandExec'n abstract class

The BackgroundCommandExecution (in isis-core) is an abstract template class provided by isis-core that defines an abstract hook method to obtain background `Command`s to be executed:

```
public abstract class BackgroundCommandExecution
                        extends AbstractIsisSessionTemplate {
    ...
    protected abstract List<? extends Command> findBackgroundCommandsToExecute();
    ...
}
```

The developer is required to implement this hook method in a subclass.

5.2.6. Quartz Scheduler Configuration

The last part of the puzzle is to actually run the (appropriate implementation of) BackgroundCommandExecution). This could be run in a batch job overnight, or run continually by, say, the Quartz scheduler or by Apache Camel. This section looks at configuring Quartz.

If using (non-ASF) Incode Platform's command module, then note that this already provides a suitable concrete implementation, namely org.isisaddons.module.command.dom.BackgroundCommandExecutionFromBackgroundCommandServiceJdo. We therefore just need to schedule this to run as a Quartz job.

First, we need to define a Quartz job, for example:

```
import
org.isisaddons.module.command.dom.BackgroundCommandExecutionFromBackgroundCommandServi
ceJdo;
public class BackgroundCommandExecutionQuartzJob extends AbstractIsisQuartzJob {
    public BackgroundCommandExecutionQuartzJob() {
        super(new BackgroundCommandExecutionFromBackgroundCommandServiceJdo());
    }
}
```

where AbstractIsisQuartzJob is in turn the following boilerplate:

```
package domainapp.webapp.quartz;
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
...
public class AbstractIsisQuartzJob implements Job {
    public static enum ConcurrentInstancesPolicy {
        SINGLE_INSTANCE_ONLY,
        MULTIPLE_INSTANCES
    }
```

```
    private final AbstractIsisSessionTemplate isisRunnable;
    private final ConcurrentInstancesPolicy concurrentInstancesPolicy;
    private boolean executing;

    public AbstractIsisQuartzJob(AbstractIsisSessionTemplate isisRunnable) {
        this(isisRunnable, ConcurrentInstancesPolicy.SINGLE_INSTANCE_ONLY);
    }
    public AbstractIsisQuartzJob(
            AbstractIsisSessionTemplate isisRunnable,
            ConcurrentInstancesPolicy concurrentInstancesPolicy) {
        this.isisRunnable = isisRunnable;
        this.concurrentInstancesPolicy = concurrentInstancesPolicy;
    }

    public void execute(final JobExecutionContext context)
            throws JobExecutionException {
        final AuthenticationSession authSession = newAuthSession(context);
        try {
            if(concurrentInstancesPolicy == ConcurrentInstancesPolicy
.SINGLE_INSTANCE_ONLY &&
               executing) {
                return;
            }
            executing = true;

            isisRunnable.execute(authSession, context);
        } finally {
            executing = false;
        }
    }

    AuthenticationSession newAuthSession(JobExecutionContext context) {
        String user = getKey(context, SchedulerConstants.USER_KEY);
        String rolesStr = getKey(context, SchedulerConstants.ROLES_KEY);
        String[] roles = Iterables.toArray(
                Splitter.on(",").split(rolesStr), String.class);
        return new SimpleSession(user, roles);
    }

    String getKey(JobExecutionContext context, String key) {
        return context.getMergedJobDataMap().getString(key);
    }
}
```

This job can then be configured to run using Quartz' `quartz-config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<job-scheduling-data
    xmlns="http://www.quartz-scheduler.org/xml/JobSchedulingData"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.quartz-scheduler.org/xml/JobSchedulingData
http://www.quartz-scheduler.org/xml/job_scheduling_data_1_8.xsd"
    version="1.8">
    <schedule>
       <job>
         <name>BackgroundCommandExecutionJob</name>
         <group>Isis</group>
         <description>
                Poll and execute any background actions persisted by the
BackgroundActionServiceJdo domain service
            </description>
         <job-class>domainapp.webapp.quartz.BackgroundCommandExecutionQuartzJob</job-
class>
         <job-data-map>
          <entry>
              <key>webapp.scheduler.user</key>
              <value>scheduler_user</value>
          </entry>
          <entry>
              <key>webapp.scheduler.roles</key>
              <value>admin_role</value>
          </entry>
         </job-data-map>
       </job>
       <trigger>
         <cron>
          <name>BackgroundCommandExecutionJobEveryTenSeconds</name>
          <job-name>BackgroundCommandExecutionJob</job-name>
          <job-group>Isis</job-group>
          <cron-expression>0/10 * * * * ?</cron-expression>
         </cron>
       </trigger>
    </schedule>
</job-scheduling-data>
```

The remaining two pieces of configuration are the `quartz.properties` file:

```
org.quartz.scheduler.instanceName = SchedulerQuartzConfigXml
org.quartz.threadPool.threadCount = 1
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
org.quartz.plugin.jobInitializer.class
=org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin
org.quartz.plugin.jobInitializer.fileNames = webapp/scheduler/quartz-config.xml
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
```

and the entry in `web.xml` for the Quartz servlet:

```
<servlet>
     <servlet-name>QuartzInitializer</servlet-name>
     <servlet-class>org.quartz.ee.servlet.QuartzInitializerServlet</servlet-class>
     <init-param>
         <param-name>config-file</param-name>
         <param-value>webapp/scheduler/quartz.properties</param-value>
     </init-param>
     <init-param>
         <param-name>shutdown-on-unload</param-name>
         <param-value>true</param-value>
     </init-param>
     <init-param>
         <param-name>start-scheduler-on-load</param-name>
         <param-value>true</param-value>
     </init-param>
     <load-on-startup>1</load-on-startup>
 </servlet>
```

```
    [[_rgsvc_application-layer-api_CommandContext]]
= `CommandContext`
:Notice: Licensed to the Apache Software Foundation (ASF) under one or more
contributor license agreements. See the NOTICE file distributed with this work for
additional information regarding copyright ownership. The ASF licenses this file to
you under the Apache License, Version 2.0 (the "License"); you may not use this file
except in compliance with the License. You may obtain a copy of the License at.
http://www.apache.org/licenses/LICENSE-2.0 . Unless required by applicable law or
agreed to in writing, software distributed under the License is distributed on an "AS
IS" BASIS, WITHOUT WARRANTIES OR  CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and limitations under
the License.
:_basedir: ../../
:_imagesdir: images/
```

The `CommandContext` service is a request-scoped service that reifies the invocation of an action on a domain object into an object itself. This reified information is encapsulated within the `Command` object.

By default, the `Command` is held in-memory only; once the action invocation has completed, the `Command` object is gone. The optional supporting `CommandService` enables the implementation of `Command` to be pluggable. With an appropriate implementation (eg as provided by the (non-ASF) Incode Platform's command module's `CommandService`) the `Command` may then be persisted.

The primary use case for persistent `Command`s is in support of background commands; they act as a parent to any background commands that can be persisted either explicitly using the `BackgroundService`, or implicitly by way of the `@Action#command()` annotation.

There are a number of related use cases:

- to enable profiling of the running application (which actions are invoked then most often, what is their response time)
- if a `PublisherService` or `PublishingService` (the latter now deprecated) is configured, they provide better traceability as the `Command` is also correlated with any published events, again through the unique `transactionId` GUID
- if a `AuditerService` or `AuditingService` (the latter now deprecated) is configured, they provide better audit information, since the `Command` (the 'cause' of an action) can be correlated to the audit records (the "effect" of the action) through the `transactionId` GUID

However, while persistent `Command`s *can* be used for these use cases, it is recommended instead to use the `InteractionContext` service and persistent implementations of the `Interaction` object, eg as provided by the (non-ASF) Incode Platform's publishmq module.

5.2.7. Screencast

The screencast provides a run-through of the command (profiling) service, auditing service, publishing service. It also shows how commands can be run in the background either explicitly by scheduling through the background service or implicitly by way of a framework annotation.

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

5.2.8. API & Implementation

The `CommandContext` request-scoped service defines the following very simple API:

```
@RequestScoped
public class CommandContext {
    @Programmatic
    public Command getCommand() { ... }
}
```

This class (`o.a.i.applib.services.CommandContext`) is also the default implementation. Under normal circumstances there shouldn't be any need to replace this implementation with another.

The `Command` type referenced above is in fact an interface, defined as:

```
public interface Command extends HasTransactionId {

    public abstract String getUser();                ①
    public abstract Timestamp getTimestamp();        ②

    public abstract Bookmark getTarget();            ③
    public abstract String getMemberIdentifier();    ④
    public abstract String getTargetClass();         ⑤
    public abstract String getTargetAction();        ⑥
    public String getArguments();                    ⑦
    public String getMemento();                      ⑧

    public ExecuteIn getExecuteIn();                 ⑨
    public Executor getExecutor();                   ⑩
    public Persistence getPersistence();             ⑪
    public boolean isPersistHint();                  ⑫

    public abstract Timestamp getStartedAt();        ⑬
    public abstract Timestamp getCompletedAt();      ⑭
    public Command getParent();                      ⑮

    public Bookmark getResult();                     ⑯
    public String getException();                    ⑰

    @Deprecated
    int next(final String sequenceAbbr);            ⑱
}
```

① `getUser()` - is the user that initiated the action.

② `getTimestamp()` - the date/time at which this action was created.

③ `getTarget()` - bookmark of the target object (entity or service) on which this action was performed

④ `getMemberIdentifier()` - holds a string representation of the invoked action

⑤ `getTargetClass()` - a human-friendly description of the class of the target object

⑥ `getTargetAction()` - a human-friendly name of the action invoked on the target object

⑦ `getArguments()` - a human-friendly description of the arguments with which the action was invoked

⑧ `getMemento()` - a formal (XML or similar) specification of the action to invoke/being invoked

⑨ `getExecuteIn()` - whether this command is executed in the foreground or background

⑩ `getExecutor()` - the (current) executor of this command, either user, or background service, or other (eg redirect after post).

⑪ `getPersistence()`- the policy controlling whether this command should ultimately be persisted (either "persisted", "if hinted", or "not persisted")

⑫ `isPersistHint()` - whether that the command should be persisted, if persistence policy is "if

hinted".

⑬ `getStartedAt()` - the date/time at which this action started (same as `timestamp` property for foreground commands)

⑭ `getCompletedAt()` - the date/time at which this action completed.

⑮ `getParent()` - for actions created through the `BackgroundService`, captures the parent action

⑯ `getResult()` - bookmark to object returned by action, if any

⑰ `getException()` - exception stack trace if action threw exception

⑱ No longer used by the framework; see instead `InteractionContext` and `Interaction#next()`.

5.2.9. Usage

The typical way to indicate that an action should be treated as a command is to annotate it with the `@Action#command()` annotation.

For example:

```
public class ToDoItem ... {
    @Action(command=CommandReification.ENABLED)
    public ToDoItem completed() { ... }
}
```

As an alternative to annotating every action with `@Action#command()`, alternatively this can be configured as the default using `isis.services.command.actions` configuration property.

See `@Action#command()` and runtime configuration for further details.

The `@Action#command()` annotation can also be used to specify whether the command should be performed in the background, for example:

```
public class ToDoItem ... {
    @Command(executeIn=ExecuteIn.BACKGROUND)
    public ToDoItem scheduleImplicitly() {
        completeSlowly(3000);
        return this;
    }
}
```

When a background command is invoked, the user is returned the command object itself (to provide a handle to the command being invoked).

This requires that an implementation of `CommandService` that persists the commands (such as the (non-ASF) Incode Platform's command module's `CommandService`) is configured. It also requires that a scheduler is configured to execute the background commands, see `BackgroundCommandService`).

5.2.10. Interacting with the services

Typically domain objects will have little need to interact with the `CommandContext` and `Command` directly; what is more useful is that these are persisted in support of the various use cases identified above.

One case however where a domain object might want to obtain the `Command` is to determine whether it has been invoked in the foreground, or in the background. It can do this using the `getExecutedIn()` method:

Although not often needed, this then allows the domain object to access the `Command` object through the `CommandContext` service. To expand th above example:

```
public class ToDoItem ... {
    @Action(
        command=CommandReification.ENABLED,
        commandExecuteIn=CommandExecuteIn.BACKGROUND
    )
    public ToDoItem completed() {
        ...
        Command currentCommand = commandContext.getCommand();
        ...
    }
    @Inject
    CommandContext commandContext;
}
```

If run in the background, it might then notify the user (eg by email) if all work is done.

This leads us onto a related point, distinguishing the current effective user vs the originating "real" user. When running in the foreground, the current user can be obtained from the `UserService`, using:

```
String user = userService.getUser().getName();
```

If running in the background, however, then the current user will be the credentials of the background process, for example as run by a Quartz scheduler job.

The domain object can still obtain the original ("effective") user that caused the job to be created, using:

```
String user = commandContext.getCommand().getUser();
```

5.2.11. Related Services

The `CommandContext` service is very similar in nature to the `InteactionContext`, in that the `Command` object accessed through it is very similar to the `Interaction` object obtained from the

InteractionContext. The principle distinction is that while Command represents the *intention* to invoke an action or edit a property, the Interaction (and contained Executions) represents the actual execution.

Most of the time a Command will be followed directly by its corresponding Interaction. However, if the Command is annotated to run in the background (using @Action#commandExecuteIn(), or is explicitly created through the BackgroundService, then the actual interaction/execution is deferred until some other mechanism invokes the command (eg as described here). The persistence of background commands requires a configured BackgroundCommandService) to actually persist such commands for execution.

Commands - even if executed in the foreground - can also be persisted by way of the CommandService. Implementations of CommandService and BackgroundCommandService are intended to go together, so that child Commands persistent (to be executed in the background) can be associated with their parent Commands (executed in the foreground, with the background Command created explicitly through the BackgroundService).

5.3. DtoMappingHelper

The DtoMappingHelper converts the domain object's internal identifier into a serializable OidDto for use in the command and interaction schemas.

5.3.1. API and Usage

The API of DtoMappingHelper is:

```
public class DtoMappingHelper {

    public OidDto oidDtoFor(final Object object) { ... }    ①
}
```

① Uses the BookmarkService to convert the domain object's internal identifier into a serializable OidDto.

This class (o.a.i.applib.services.dto.DtoMappingHelper) is also the implementation.

5.4. InteractionContext

The InteractionContext is a request-scoped domain service that is used to obtain the current Interaction.

An Interaction generally consists of a single top-level Execution, either to invoke an action or to edit a property. If that top-level action or property uses WrapperFactory to invoke child actions/properties, then those sub-executions are captured as a call-graph. The Execution is thus a graph structure.

If a bulk action is performed (as per an action annotated using @Action#invokeOn()), then this will result in multiple Interactions, one per selected object (not one Interaction with multiple top-level

Executions).

It is possible for Interaction.Executions to be persisted; this is supported by the (non-ASF) Incode Platform's publishmq module, for example. Persistent Interactions support several use cases:

- they enable profiling of the running application (which actions are invoked then most often, what is their response time)
- if auditing is configured (using either auditing or AuditerService), they provide better audit information, since the Interaction.Execution captures the 'cause' of an interaction and can be correlated to the audit records (the "effect" of the interaction) by way of the transactionId

5.4.1. API & Implementation

The public API of the service consists of several related classes:

- InteractionContext domain service itself:
- Interaction class, obtainable from the InteractionContext
- Execution class, obtainable from the Interaction.

The Execution class itself is abstract; there are two subclasses, ActionInvocation and PropertyEdit.

InteractionContext

The public API of the InteractionContext domain service itself consists of simply:

```
@RequestScoped
public class InteractionContext {
    public Interaction getInteraction();            ①
}
```

① Returns the currently active {@link Interaction} for this thread.

This class is concrete (that is, it is also the implementation).

Interaction

The public API of the Interaction class consists of:

```
public class Interaction {
    public UUID getTransactionId();                 ①
    public Execution getPriorExecution();           ②
    public Execution getCurrentExecution();         ③
    public List<Execution> getExecutions();         ④
    public int next(final String sequenceId);       ⑤
}
```

① The unique identifier of this interaction. This will be the same value as held in Command (obtainable from CommandContext).

② The member `Execution` (action invocation or property edit) that preceded the current one.

③ The current execution.

④ * Returns a (list of) execution}s in the order that they were pushed. Generally there will be just one entry in this list, but additional entries may arise from the use of mixins/contributions when re-rendering a modified object.

⑤ Generates numbers in a named sequence. Used by the framework both to number successive interaction `Execution`s and for events published by the `PublisherService`.

This class is concrete (is also the implementation).

`Interaction.Execution`

The `Interaction.Execution` (static nested) class represents an action invocation/property edit as a node in a call-stack execution graph. Sub-executions can be performed using the `WrapperFactory`.

It has the following public API:

```
public abstract class Execution {
    public Interaction getInteraction();            ①
    public InteractionType getInteractionType();    ②
    public String getMemberIdentifier();            ③
    public Object getTarget();                      ④

    public String getTargetClass();                 ⑤
    public String getTargetMember();

    public Execution getParent();                   ⑥
    public List<Execution> getChildren();

    public AbstractDomainEvent getEvent();          ⑦

    public Timestamp getStartedAt();                ⑧
    public Timestamp getCompletedAt();

    public Object getReturned();                    ⑨
    public Exception getThrew();

    public T getDto();                              ⑩
}
```

① The owning `Interaction`.

② Whether this is an action invocation or a property edit.

③ A string uniquely identifying the action or property (similar to Javadoc syntax).

④ The object on which the action is being invoked or property edited. In the case of a mixin this will be the mixin object itself (rather than the mixed-in object).

⑤ A human-friendly description of the class of the target object, and of the name of the action invoked/property edited on the target object.

⑥ The parent action/property that invoked this action/property edit (if any), and any actions/property edits made in turn via the `WrapperFactory`.

⑦ The domain event fired via the `EventBusService` representing the execution of this action invocation/property edit.

⑧ The date/time at which this execution started/completed.

⑨ The object returned by the action invocation/property edit, or the exception thrown. For `void` methods and for actions returning collections, the value will be `null`.

⑩ A DTO (instance of the "ixn" schema) being a serializable representation of this action invocation/property edit.

Unlike the similar `CommandContext` domain service (discussed below) there is no domain service to different implementations of `Interaction` to be used. That said, the framework simply instantiates the `Interaction` using the `FactoryService`. If a different implementation of `Interaction` was required, then a custom implementation of `FactoryService` could always be supplied.

There are two concrete subclasses of `Execution`.

The first is `ActionInvocation`, representing the execution of an action being invoked:

```
public class ActionInvocation extends Execution {
    public List<Object> getArgs();                  ①
}
```

① The objects passed in as the arguments to the action's parameters. Any of these could be `null`.

The second is `PropertyEdit`, and naturally enough represents the execution of a property being edited:

```
public class PropertyEdit extends Execution {
    public Object getNewValue();                    ①
}
```

① The object used as the new value of the property. Could be `null` if the property is being cleared.

5.4.2. Interacting with the services

Typically domain objects will have little need to interact with the `InteractionContext` and `Interaction` directly. The services are used within the framework however, primarily to support the `PublisherService` SPI, and to emit domain events over the `EventBusService`.

5.4.3. Related Classes

This service is very similar in nature to `CommandContext`, in that the `Interaction` object accessed through it is very similar to the `Command` object obtained from the `CommandContext`. The principle distinction is that while `Command` represents the *intention* to invoke an action or edit a property, the

`Interaction` (and contained `Executions`) represents the actual execution.

Most of the time a `Command` will be followed directly by its corresponding `Interaction`. However, if the `Command` is annotated to run in the background (using `@Action#commandExecuteIn()`, or is explicitly created through the `BackgroundService`, then the actual interaction/execution is deferred until some other mechanism invokes the command (eg as described here).

5.5. `MessageService`

The `MessageService` allows domain objects to raise information, warning or error messages. These messages can either be simple strings, or can be translated.

The methods in this service replace similar methods (now deprecated) in `DomainObjectContainer`.

5.5.1. API and Usage

The API of `MessageService` is:

```
public interface MessageService {

    void informUser(String message);
①
    String informUser(TranslatableString message, Class<?> contextClass, String
contextMethod); ②

    void warnUser(String message);
③
    String warnUser(TranslatableString message, Class<?> contextClass, String
contextMethod);   ④

    void raiseError(String message);
⑤
    String raiseError(TranslatableString message, Class<?> contextClass, String
contextMethod); ⑥
    ...
}
```

① display as a transient message to the user (not requiring acknowledgement). In the Wicket viewer this is implemented as a toast that automatically disappears after a period of time.

② ditto, but with translatable string, for i18n support.

③ warn the user about a situation with the specified message. In the Wicket viewer this is implemented as a toast that must be closed by the end-user.

④ ditto, but with translatable string, for i18n support.

⑤ show the user an unexpected application error. In the Wicket viewer this is implemented as a toast (with a different colour) that must be closed by the end-user.

⑥ ditto, but with translatable string, for i18n support.

For example:

```
public Order addItem(Product product, @ParameterLayout(named="Quantity") int quantity)
{
    if(productRepository.stockLevel(product) == 0) {
        messageService.warnUser(
            product.getDescription() + " out of stock; order fulfillment may be
delayed");
    }
    ...
}
```

5.5.2. Implementation

The core framework provides a default implementation of this service, `o.a.i.core.runtime.services.message.MessageServiceDefault`.

5.6. SessionManagementService

The `SessionManagementService` provides the ability to programmatically manage sessions. The primary use case is for fixture scripts or other routines that are invoked from the UI and which create or modify large amounts of data. A classic example is migrating data from one system to another.

5.6.1. API

The API of `SessionManagementService` is:

```
public interface SessionManagementService {
    void nextSession();
}
```

Normally, the framework will automatically start a session and then a transaction before each user interaction (action invocation or property modification), and wil then commit that transaction and close the session after the interaction has completed. If the interaction throws an exception then the transaction is aborted.

The `nextSession()` method allows a domain object to commit the transaction, close the session, then open a new session and start a new transaction.

Any domain objects that were created in the "previous" session are no longer usable, and must not be rendered in the UI.

5.6.2. Implementation

The core framework provides a default implementation of this service (`o.a.i.core.runtime.services.xactn.SessionManagementServiceDefault`).

5.7. `TitleService`

The `TitleService` provides methods to programmatically obtain the title and icon of a domain object.

The methods in this service replace similar methods (now deprecated) in `DomainObjectContainer`.

5.7.1. API

The API of `TitleService` is:

```
public interface PresentationService {
    String titleOf(Object domainObject);                ①
    String iconNameOf(Object domainObject);             ②
}
```

① return the title of the object, as rendered in the UI by the Apache Isis viewers.

② return the icon name of the object, as rendered in the UI by the Apache Isis viewers.

5.7.2. Usage

By way of example, here's some code based on a system for managing government benefits:

```
public class QualifiedAdult {

    private Customer qualifying;

    public String title() {
        return "QA for " + titleService.titleOf(qualifying);
    }

    ...
    @Inject
    TitleService titleService;
}
```

In this example, whatever the title of a `Customer`, it is reused within the title of that customer's `QualifiedAdult` object.

5.7.3. Implementation

The core framework provides a default implementation of this service (`o.a.i.core.metamodel.services.title.TitleServiceDefault`).

5.8. `TransactionService3`

The `TransactionService3` (and its various supertypes) allows domain objects to influence user transactions.

The methods in this service replace similar methods (now deprecated) in `DomainObjectContainer`.

5.8.1. API

The API of `TransactionService3` is:

```
public interface TransactionService3 {
    Transaction2 currentTransaction();      ①
    void nextTransaction();                 ②
    void nextTransaction(Policy policy);    ③
    void flushTransaction();                ④
    TransactionState getTransactionState(); ⑤
}
```

① to obtain a handle on the current `Transaction`, discussed further below

② The framework automatically start a transaction before each user interaction (action invocation or property edit), and will commit that transaction after the interaction has completed. Under certain circumstances (eg actions used to perform data migration, say, or for large fixture scripts), it can be helpful to programmatically complete one transaction and start another one.

③ overload of nextTransaction() that provides more control on the action to be performed if the current transaction has been marked for abort only

④ If the user interaction creates/persists an object or deletes an object (eg using the `RepositoryService`'s `persist()` or `delete()` methods), then the framework actually queues up the work and only performs the persistence command either at the end of the transaction or immediately prior to the next query. Performing a flush will cause any pending calls to be performed immediately.

⑤ the state of the current or most recently completed transaction.

Here `TransactionState` is an enum defined as:

```
public enum TransactionState {
    NONE,           ①
    IN_PROGRESS,    ②
    MUST_ABORT,     ③
    COMMITTED,      ④
    ABORTED;        ⑤
}
```

① No transaction exists.

② Started, still in progress. May flush, commit or abort.

③ Started, but has hit an exception. May not flush or commit, can only abort.

④ Completed, having successfully committed. May not flush or abort or commit.

⑤ Completed, having aborted. Again, may not flush or abort or commit.

As noted above, `nextTransaction()` can be useful for actions used to perform data migration, say, or for large fixture scripts. It is also used by the Wicket viewer's support for bulk actions; each action is invoked in its own transaction. An overload of this method takes a `Policy` enum, defined as:

```
public enum Policy {
    UNLESS_MARKED_FOR_ABORT,
    ALWAYS
}
```

If the current transaction has been marked for abort, then the `Policy.UNLESS_MARKED_FOR_ABORT` will escalate to a runtime exception, that is, will fail fast. Specifying `Policy.ALWAYS` is provided for use by integration tests so that they can continue on with the test teardown even if the test caused an issue.

The `Transaction2` object - as obtained by `currentTransaction()` method, above - is a minimal wrapper around the underlying database transaction. Its API is:

```
public interface Transaction2 {
    UUID getTransactionId();                    ①
    int getSequence();                          ②
    void flush();                               ③
    TransactionState getTransactionState();     ④
    void clearAbortCause();                     ⑤
}
```

① is a unique identifier for the interaction/request, as defined by the `HasTransactionId` mixin.

② there can actually be multiple transactions within such a request/interaction; the sequence is a (0-based) is used to distinguish such.

③ as per `TransactionService#flushTransaction()` described above.

④ The state of this transaction (same as `TransactionService#getTransactionState()`).

⑤ (For framework use only) If the cause has been rendered higher up in the stack, then clear the cause so that it won't be picked up and rendered elsewhere.

One place where `clearAboutCause()` may be useful is for application-level handling of SQL integrity exceptions, eg as described in ISIS-1476:

```
try {
    // do something...
} catch (final JDODataStoreException e) {
    if (Iterables.filter(Throwables.getCausalChain(e),
        SQLIntegrityConstraintViolationException.class) != null) {
        // ignore
        this.transactionService.currentTransaction().clearAbortCause(
);
    } else {
        throw e;
    }
}
```

5.8.2. Implementation

The core framework provides a default implementation of this service, `o.a.i.core.metamodel.services.xactn.TransactionServiceDefault`.

5.9. `WrapperFactory`

The `WrapperFactory` provides the ability to enforce business rules for programmatic interactions between domain objects. If there is a (lack-of-) trust boundary between the caller and callee — eg if they reside in different modules — then the wrapper factory is a useful mechanism to ensure that any business constraints defined by the callee are honoured.

For example, if the calling object attempts to modify an unmodifiable property on the target object, then an exception will be thrown. Said another way: interactions are performed "as if" they are through the viewer.

For a discussion of the use of the `WrapperFactory` within integration tests (the primary or at least original use case for this service) can be found here

This capability goes beyond enforcing the (imperative) constraints within the `hideXxx()`, `disableXxx()` and `validateXxx()` supporting methods; it also enforces (declarative) constraints such as those represented by annotations, eg `@MaxLength` or `@Regex`.

This capability is frequently used within integration tests, but can also be used in production code. (There are analogies that can be drawn here with the way that JEE beans can interact through an EJB local interface).

5.9.1. API

The API provided by the service is:

```
public interface WrapperFactory {
    @Programmatic
    <T> T wrap(T domainObject);                                    ①
    @Programmatic
    <T> T unwrap(T possibleWrappedDomainObject);                   ②
    @Programmatic
    <T> boolean isWrapper(T possibleWrappedDomainObject);          ③

    public static enum ExecutionMode {                             ④
        EXECUTE(true,true),
        SKIP_RULES(false, true),                                   ⑤
        NO_EXECUTE(true, false);                                   ⑥
    }
    @Programmatic
    <T> T wrap(T domainObject, ExecutionMode mode);                ⑦
    @Programmatic
    <T> T wrapNoExecute(T domainObject);                           ⑧
    @Programmatic
    <T> T wrapSkipRules(T domainObject);                           ⑨
    ...
 }
```

① wraps the underlying domain object. If it is already wrapped, returns the object back unchanged.

② Obtains the underlying domain object, if wrapped. If the object is not wrapped, returns back unchanged.

③ whether the supplied object has been wrapped.

④ enumerates how the wrapper interacts with the underlying domain object.

⑤ validate all business rules and then execute.

⑥ skip all business rules and then execute (including creating commands and firing pre- and post-execute domain events).

⑦ validate all business rules (including those from domain events) but do not execute.

⑧ convenience method to invoke `wrap(…)` with `ExecuteMode#NO_EXECUTE` (make this feature more discoverable)

⑨ convenience method to invoke `wrap(…)` with `ExecuteMode#SKIP_RULES` (make this feature more discoverable)

The service works by returning a "wrapper" around a supplied domain object (a javassist proxy), and it is this wrapper that ensures that the hide/disable/validate rules implies by the Apache Isis programming model are enforced. The wrapper can be interacted with as follows:

- a `get…()` method for properties or collections

- a `set…()` method for properties
- an `addTo…()` or `removeFrom…()` method for collections
- any action

Calling any of the above methods may result in a (subclass of) `InteractionException` if the object disallows it. For example, if a property is annotated with `@Hidden` then a `HiddenException` will be thrown. Similarly if an action has a `validateXxx()` method and the supplied arguments are invalid then an `InvalidException` will be thrown.

In addition, the following methods may also be called:

- the `title()` and `toString()` methods
- any `default…()`, `choices…()` or `autoComplete…()` methods

An exception will be thrown if any other methods are thrown.

5.9.2. Usage

The caller will typically obtain the target object (eg from some repository) and then use the injected `WrapperFactory` to wrap it before interacting with it.

For example:

```
public class CustomerAgent {
    @Action
    public void refundOrder(final Order order) {
        final Order wrappedOrder = wrapperFactory.wrap(order);
        try {
            wrappedOrder.refund();
        } catch(InteractionException ex) {          ①
            container.raiseError(ex.getMessage());  ②
            return;
        }
    }
    ...
    @Inject
    WrapperFactory wrapperFactory;
    @Inject
    DomainObjectContainer container;
}
```

① if any constraints on the `Order's `refund()` action would be violated, then …

② … these will be trapped and raised to the user as a warning.

It ought to be possible to implement an `ExceptionRecognizer`s that would allow the above boilerplate to be removed. This recognizer service would recognize the `InteractionException` and convert to a suitable message.

At the time of writing Apache Isis does not provide an out-of-the-box implementation of such an `ExceptionRecognizer`; but it should be simple enough to write one...

5.9.3. Listener API

The `WrapperFactory` also provides a listener API to allow other services to listen in on interactions.

```
public interface WrapperFactory {
    ...
    @Programmatic
    List<InteractionListener> getListeners();                                   ①
    @Programmatic
    public boolean addInteractionListener(InteractionListener listener);        ②
    @Programmatic
    public boolean removeInteractionListener(InteractionListener listener);     ③
    @Programmatic
    public void notifyListeners(InteractionEvent ev);                           ④
}
```

① all `InteractionListener`s that have been registered.

② registers an `InteractionListener`, to be notified of interactions on all wrappers. The listener will be notified of interactions even on wrappers created before the listener was installed. (From an implementation perspective this is because the wrappers delegate back to the container to fire the events).

③ remove an `InteractionListener`, to no longer be notified of interactions on wrappers.

④ used by the framework itself

The original intent of this API was to enable test transcripts to be captured (in a BDD-like fashion) from integration tests. No such feature has yet been implemented however. Also, the capabilities have by and large been superceded by Apache Isis' support for domain events. We may therefore deprecate this API in the future.

Chapter 6. Application Layer SPI

Domain service SPIs influence how the framework handles application layer concerns, for example which home page to render to the end-user.

The table below summarizes the application layer SPIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 4. Application Layer SPI

API	Description	Implementation	Notes
o.a.i.applib.services.background.BackgroundCommandService	Persisted a memento of an action invocation such that it can be executed asynchronously ("in the background") eg by a scheduler.	BackgroundCommandServiceJdo o.ia.m.command isis-module-command	related services: BackgroundCommandService-JdoContributions, BackgroundCommandService-JdoRepository
o.a.i.applib.services.command.spi.CommandService	Service to act as a factory and repository (create and save) of command instances, ie representations of an action invocation. Used for command/auditing and background services.	CommandServiceJdo o.ia.m.command isis-module-command	related services: CommandService-`JdoContributions`, `CommandService-`JdoRepository
o.a.i.applib.services.homepage.HomePageProviderService	Returns the home page object, if any is defined.	HomePageProviderServiceDefault o.a.i.core isis-core-runtime	Used by the default implementation of RoutingService.

Key:

- o.a.i is an abbreviation for org.apache.isis
- o.ia.m is an abbreviation for org.isisaddons.module
- o.a.i.c.m.s is an abbreviation for org.apache.isis.core.metamodel.services
- o.a.i.c.r.s is an abbreviation for org.apache.isis.core.runtime.services

6.1. BackgroundCommandService

The BackgroundCommandService (SPI) service supports the BackgroundService (API) service, persisting action invocations as commands such that they can subsequently be invoked in the background.

The BackgroundService is responsible for capturing a memento representing the action invocation, and then hands off to the BackgroundCommandService BackgroundCommandService to actually persist it.

The persisting of commands is only half the story; there needs to be a separate process to read the commands and execute them. The abstract BackgroundCommandExecution provides a mechanism to execute such commands. This can be considered an API, albeit "internal" because the implementation relies on internals of the framework.

6.1.1. SPI

The SPI of the BackgroundCommandService is:

```
public interface BackgroundCommandService {
    void schedule(
            ActionInvocationMemento aim,        ①
            Command parentCommand,              ②
            String targetClassName,
            String targetActionName,
            String targetArgs);

}
```

① is a wrapper around a MementoService's Memento, capturing the details of the action invocation to be retained (eg persisted to a database) so that it can be executed at a later time

② reference to the parent Command requesting the action be performed as a background command. This allows information such as the initiating user to be obtained.

The API of ActionInvocationMemento in turn is:

```
public class ActionInvocationMemento {
    public String getActionId() { ... }
    public String getTargetClassName() { ... }
    public String getTargetActionName() { ... }
    public Bookmark getTarget() { ... }
    public int getNumArgs() { ... }
    public Class<?> getArgType(int num) throws ClassNotFoundException { ... }
    public <T> T getArg(int num, Class<T> type) { ... }

    public String asMementoString() { ... }     ①
}
```

① lets the BackgroundCommandService implementation convert the action invocation into a simple string.

6.1.2. "Internal" SPI

The BackgroundCommandExecution (in isis-core) is an abstract template class for headless access, that defines an abstract hook method to obtain background `Command`s to be executed:

```
public abstract class BackgroundCommandExecution
                        extends AbstractIsisSessionTemplate {
    ...
    protected abstract List<? extends Command> findBackgroundCommandsToExecute();
    ...
}
```

The developer is required to implement this hook method in a subclass.

6.1.3. Implementation

The (non-ASF) Incode Platform's command module provides an implementation (org.isisaddons.module.command.dom.BackgroundCommandServiceJdo) that persists Commands using the JDO/DataNucleus object store. It further provides a number of supporting services:

- org.isisaddons.module.command.dom.BackgroundCommandServiceJdoRepository is a repository to search for persisted background Commands
- org.isisaddons.module.command.dom.BackgroundCommandServiceJdoContributions contributes actions for searching for persisted child and sibling Commands.

The module also provides a concrete subclass of BackgroundCommandExecution that knows how to query for persisted (background) `Command`s such that they can be executed by a scheduler.

Details of setting up the Quartz scheduler to actually execute these persisted commands can be found on the BackgroundService page.

6.1.4. Usage

Background commands can be created either declaratively or imperatively.

The declarative approach involves annotating an action using @Action#command() with @Action#commandExecuteIn=CommandExecuteIn.BACKGROUND.

The imperative approach involves explicitly calling the BackgroundService from within domain object's action.

6.1.5. Alternative Implementations

The (non-ASF) Incode Platform's command module provides an implementation of this service (BackgroundCommandService), and also provides a number of related domain services (BackgroundCommandServiceJdo, BackgroundCommandJdoRepository and BackgroundCommandServiceJdoContributions). This module also provides service implementations of the CommandService.

If contributions are not required in the UI, these can be suppressed either using security or by implementing a vetoing subscriber.

6.1.6. Related Services

As discussed above, this service supports the `BackgroundService` , persisting `Command`s such that they can be executed in the background.

There is also a tie-up with the `CommandContext` and its supporting `CommandService` domain service. The `CommandContext` service is responsible for providing a parent `Command` with which the background `Command`s can then be associated as children, while the `CommandService` is responsible for persisting those parent `Command`s (analogous to the way in which the `BackgroundCommandService` persists the child background `Command`s). The `BackgroundCommandService` ensures that these background `Command`s are associated with the parent "foreground" `Command`.

What that means is that the implementations of `CommandService` and `BackgroundCommandService` go together, hence both implemented in the (non-ASF) Incode Platform's command module.).

6.2. `CommandService`

The `CommandService` service supports the `CommandContext` service such that `Command` objects (that reify the invocation of an action/edit of a property on a domain object) can be persisted.

The primary use case for persistent `Command`s is in support of background commands; they act as a parent to any background commands that can be persisted either explicitly using the `BackgroundService`, or implicitly by way of the `@Action#command()` annotation.

Persistent `Command`s also support the ability to replicate from a master to a slave instance of an application. One use case for this is for regression testing, allowing a production usages to be replayed against a new release candidate, eg after upgrading that application to a new version of Apache Isis itself (or some other dependency).

There are a number of related use cases:

- they enable profiling of the running application (which actions are invoked then most often, what is their response time)
- if `PublisherService` or `PublishingService` (latter deprecated) is configured, they provide better traceability as the `Command` is also correlated with any published events, again through the unique `transactionId` GUID
- if `AuditerService` or `AuditingService` (latter deprecated) is configured, they provide better audit information, since the `Command` (the 'cause' of an action) can be correlated to the audit records (the "effect" of the action) through the `transactionId` GUID

However, while persistent `Command`s *can* be used for these use cases, it is recommended instead to use the `InteractionContext` service and persistent implementations of the `Interaction` object, eg as provided by the (non-ASF) Incode Platform's publishmq module.

6.2.1. Screencast

The screencast below provides a run-through of the command (profiling) service, auditing service, publishing service. It also shows how commands can be run in the background either explicitly by scheduling through the background service or implicitly by way of a framework annotation.

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

6.2.2. SPI

The `CommandService` service defines the following very simple API:

```
public interface CommandService {
    Command create();                                                   ①
    @Deprecated
    void startTransaction(Command command, UUID transactionId);        ②
    boolean persistIfPossible(Command command);                        ③
    void complete(Command command);                                    ④
}
```

① Instantiate the appropriate instance of the `Command` (as defined by the `CommandContext` service). Its members will be populated automatically by the framework.

② Deprecated and **IS NOT CALLED** by the framework. The framework automatically populates the `Command`'s `timestamp`, `user` and `transactionId` fields, so there is no need for the service implementation to initialize any of these. In particular, the `Command` will already have been initialized with the provided `transactionId` argument.

③ Set the hint that the `Command` should be persisted if possible (when completed, see below).

④ "Complete" the command, typically meaning that the command should be persisted it if its `Command#getPersistence()` flag and persistence hint (`Command#isPersistHint()`) indicate that it should be.

The framework will automatically have set the `completedAt` property of the `Command`.

6.2.3. Implementation

The (non-ASF) Incode Platform's command module provides an implementation (`org.isisaddons.module.command.dom.CommandServiceJdo`) that persists `Command`s using the JDO/DataNucleus object store. It further provides a number of supporting services:

- `org.isisaddons.module.command.dom.CommandServiceJdoRepository` is a repository to search for persisted `Command`s
- `org.isisaddons.module.command.dom.CommandServiceJdoContributions` contributes actions for searching for persisted child and sibling `Command`s.

6.2.4. Usage

The typical way to indicate that an action should be reified into a `Command` is by annotating the action using `@Action#command()`.

6.2.5. Alternative Implementations

The (non-ASF) Incode Platform's command module provides an implementation of this service (`CommandService`), and also provides a number of related domain services (`CommandJdoRepository` and `CommandServiceJdoContributions`). This module also provides service implementations of the `BackgroundCommandService`.

If contributions are not required in the UI, these can be suppressed either using security or by implementing a vetoing subscriber.

6.2.6. Related Services

As discussed above, this service supports the `CommandContext`, providing the ability for `Command` objects to be persisted. This is closely related to the `BackgroundCommandService`that allows the `BackgroundService` to schedule commands for background/asynchronous execution.

The implementations of `CommandService` and `BackgroundCommandService` are intended to go together, so that persistent parent `Command`s can be associated with their child background `Command`s.

The services provided by this module combines very well with the <code>AuditingService</code>. The <code>CommandService</code> captures the _cause of an interaction (an action was invoked, a property was edited), while the <code>AuditingService3</code> captures the effect of that interaction in terms of changed state.

You may also want to configure the `PublishingService`.

All three of these services collaborate implicitly by way of the `HasTransactionId` interface.

6.3. `HomePageProviderService`

This service simply provides access to the home page object (if any) that is returned from the domain service action annotated with `@HomePage`.

It was originally introduced to support the default implementation of `RoutingService`, but was factored out to support alternative implementations of that service (and may be useful for other use cases).

6.3.1. SPI & Implementation

The SPI defined by `HomePageProviderService` is:

```
@DomainService(nature = NatureOfService.DOMAIN)
public interface HomePageProviderService {
    @Programmatic
    Object homePage();
}
```

The default implementation is provided by `o.a.i.core.runtime.services.homepage.HomePageProviderServiceDefault`.

Chapter 7. Core/Domain API

The core/domain APIs provide general-purpose services to the domain objects, for example obtaining the current time or user, or instantiating domain objects.

The table below summarizes the core/domain APIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 5. Core/Domain Layer API

API	Description	Implementation	Notes
`o.a.i.applib.services.clock.ClockService`	Access the current time (and for testing, allow the time to be changed)	`ClockService` `o.a.i.core:isis-core-applib`	API is also a concrete class.
`o.a.i.applib.services.config.ConfigurationService`	Access configuration properties (eg from `isis.properties` file)	`ConfigurationService-Default` `o.a.i.core:isis-core-runtime`	The `ConfigurationServiceMenu` exposes the `allConfigurationProperties` action in the user interface. + Supercedes methods in `DomainObjectContainer`.
`o.a.i.applib.DomainObjectContainer`	Miscellaneous functions, eg obtain title of object.	`DomainObjectContainer-Default` `o.a.i.core:isis-core-metamodel`	
`o.a.i.applib.services.eventbus.EventBusService`	Programmatically post events to the internal event bus. Also used by Apache Isis itself to broadcast domain events: * `Action#domainEvent()` * `Property#domainEvent()` * `Collection#domainEvent()`	`EventBusServiceJdo` `o.a.i.core:isis-core-runtime`	

API	Description	Implementation	Notes
o.a.i.applib.services.factory FactoryService	Methods to instantiate and initialize domain objects	FactoryService-Default o.a.i.core isis-core-metamodel	Supercedes methods in DomainObjectContainer.
o.a.i.applib.services.scratchpad Scratchpad	Request-scoped service for interchanging information between and aggregating over multiple method calls; in particular for use by "bulk" actions (invoking of an action for all elements of a collection)	Scratchpad o.a.i.core isis-core-applib	API is also a concrete class
o.a.i.applib.services.xactn UserService	Methods to access the currently-logged on user.	UserService-Default o.a.i.core isis-core-metamodel	Supercedes methods in DomainObjectContainer.

Key:

- o.a.i is an abbreviation for org.apache.isis
- o.ia.m is an abbreviation for org.isisaddons.module
- o.a.i.c.m.s is an abbreviation for org.apache.isis.core.metamodel.services
- o.a.i.c.r.s is an abbreviation for org.apache.isis.core.runtime.services

There is also a number of deprecated domain services.

Table 6. Deprecated Domain Services

API	Description	Implementation	Notes
o.a.i.applib.annotation Bulk.InteractionContext	Request-scoped access to whether action is invoked on object and/or on collection of objects	Bulk.InteractionContext o.a.i.core isis-core-applib	Replaced by ActionInvocationContext

Key:

Key:

- o.a.i is an abbreviation for org.apache.isis
- o.ia.m is an abbreviation for org.isisaddons.module
- o.a.i.c.m.s is an abbreviation for org.apache.isis.core.metamodel.services
- o.a.i.c.r.s is an abbreviation for org.apache.isis.core.runtime.services

7.1. ClockService

Most applications deal with dates and times in one way or another. For example, if an `Order` is placed, then the `Customer` may have 30 days to pay the `Invoice`, otherwise a penalty may be levied.

However, such date/time related functionality can quickly complicate automated testing: "today+30" will be a different value every time the test is run.

Even disregarding testing, there may be a requirement to ensure that date/times are obtained from an NNTP server (rather than the system PC). While instantiating a `java.util.Date` to current the current time is painless enough, we would not want complex technical logic for querying an NNTP server spread around domain logic code.

Therefore it's common to provide a domain service whose responsibility is to provide the current time. This service can be injected into any domain object (and can be mocked out for unit testing). Apache Isis provides such a facade through the `ClockService`.

7.1.1. API & Implementation

The API defined by `ClockService` is:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class ClockService {
    @Programmatic
    public LocalDate now() { ... }
    @Programmatic
    public LocalDateTime nowAsLocalDateTime() { ... }
    @Programmatic
    public DateTime nowAsDateTime() { ... }
    @Programmatic
    public Timestamp nowAsJavaSqlTimestamp() { ... }
    @Programmatic
    public long nowAsMillis() { ... }
}
```

This class (`o.a.i.applib.services.clock.ClockService`) is also the default implementation. The time provided by this default implementation is based on the system clock.

7.1.2. Testing Support

The default `ClockService` implementation in fact simply delegates to another class defined in the API, namely the `o.a.i.applib.clock.Clock`, an abstract singleton class. It is not recommended that your code use the `Clock` directly, but you should understand how this all works:

- there are two subclasses implementations `Clock`, namely `SystemClock` and `FixtureClock`.
- the first implementation that is instantiated registers itself as the singleton.
- if running in production (server) mode, then (unless another implementation has beaten it to the punch) the framework will instantiate the `SystemClock`. Once instantiated this cannot be

replaced.

- if running in prototype mode, then the framework will instead instantiate <code>FixtureClock</code>. This _can be replaced if required.

The `FixtureClock` will behave as the system clock, unless its is explicitly set using `FixtureClock#setDate(…)` or `FixtureClock#setTime(…)` and so forth.

Alternative Implementations

Suppose you want (as discussed in the introduction to this service) to use a clock that delegates to NNTP. For most domain services this would amount to implementing the appropriate service and registering the owning module so that it is used in preference to any implementations provided by default by the framework.

In the case of the `ClockService`, though, this approach (unfortunately) will not work, because parts of Apache Isis (still) delegate to the `Clock` singleton rather than using the `ClockService` domain service.

The workaround, therefore, is to implement your functionality as a subclass of `Clock`. You can write a domain service that will ensure that your implementation is used ahead of any implementations provided by the framework.

For example:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class NntpClockServiceInitializer  {
    @Programmatic
    @PostConstruct
    public void postConstruct(Map<String,String> properties) {
        new NntpClock(properties);                          ①
    }
    private static class NntpClock extends Clock {
        NntpClock(Map<String,String> properties) { ... } ②
        protected long time() { ... }                       ③
            ... NNTP stuff here ...
        }
    }
}
```

① enough to simply instantiate the `Clock`; it will register itself as singleton

② connect to NNTP service using configuration properties from `isis.properties`

③ call to NNTP service here

7.2. `ConfigurationService`

The `ConfigurationService` allows domain objects to read the configuration properties aggregated from the various configuration files.

Only configuration properties with the prefix "application" are be exposed.

The methods in this service replace similar methods (now deprecated) in `DomainObjectContainer`.

7.2.1. API and Usage

The API of `ConfigurationService` is:

```
public interface ConfigurationService {

    String getProperty(String name);                          ①
    String getProperty(String name, String defaultValue);     ②
    List<String> getPropertyNames();                          ③
    Set<ConfigurationProperty> allProperties();               ④

}
```

① Return the configuration property with the specified name; else return null.

② Return the configuration property with the specified name; if it doesn't exist then return the specified default value.

③ Return the names of all the available properties.

④ Returns all properties, each as an instance of the `ConfigurationProperty` view model.

For example, here's a fictitious service that might wrap Twitter4J. say:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class TweetService {
    @Programmatic
    @PostConstruct
    public void init() {
        this.oauthConsumerKey = configurationService.getProperty(
"application.tweetservice.oauth.consumerKey");
        this.oauthConsumerSecret = configurationService.getProperty(
"application.tweetservice.oauth.consumerSecret");
        this.oauthAccessToken = configurationService.getProperty(
"application.tweetservice.oauth.accessToken");
        this.oauthAccessTokenSecret = configurationService.getProperty(
"application.tweetservice.oauth.accessTokenSecret");
    }
    ...
    @Inject
    ConfigurationService configurationService;
}
```

If you *do* have a domain service that needs to access Isis properties, then an alternative is to define a `@PostConstruct` method and pass in a `Map<String,String>` of properties. This is provided all properties, not just those with the 'application' prefix.

7.2.2. Implementation

The core framework provides a default implementation of this service (`o.a.i.core.runtime.services.config.ConfigurationServiceDefault`).

7.2.3. Related services

The `ConfigurationServiceMenu` exposes the `allConfigurationProperties` action in the user interface.

7.3. DomainObjectContainer

The `DomainObjectContainer` service provides a set of general purpose functionality for domain objects to call. Principal amongst these are a generic APIs for querying objects and creating and persisting objects. In addition, the service provides access to security context (the "current user"), allows information and warning messages to be raised, and various other miscellaneous functions.

(Almost all of) the methods in this service have been moved out into a number of more fine-grained services: `RepositoryService`, `MessageService`, `FactoryService`, `TitleService`, `ConfigurationService`, `UserService` and `ServiceRegistry`.

7.3.1. APIs

The sections below discuss the functions provided by the service, broken out into categories.

Object Creation API

The object creation APIs are used to instantiate new domain objects or view models.

```
public interface DomainObjectContainer {

    <T> T newTransientInstance(final Class<T> ofType);                          ①
    <T> T newViewModelInstance(final Class<T> ofType, final String memento);    ②
    <T> T mixin();                                                              ③
    ...
}
```

① create a new non-persisted domain entity. Any services will be automatically injected into the service.

② create a new view model, with the specified memento (as per ViewModel#viewModelMemento(). In general it is easier to just annotate with `@ViewModel` and let Apache Isis manage the memento automatically.

③ programmatically instantiate a mixin, as annotated with `@Mixin` or `@DomainObject#nature()`.

For example:

```
Customer cust = container.newTransientInstance(Customer.class);
cust.setFirstName("Freddie");
cust.setLastName("Mercury");
container.persist(cust);
```

As an alternative to using `newTransientInstance(…)` or `mixin(…)`, you could also simply `new()` up the object. Doing this will not inject any domain services, but they can be injected manually using #injectServicesInto(...)`.

Calling `new(…)` also this circumvents Apache Isis' `created()` callback, and in addition any default values for properties (either explicitly set by `default…()` or defaulted implicitly according to Apache Isis' own conventions) will not be called either. If you don't intend to use these features, though, the net effect is code that has less coupling to Isis and is arguably easier to understand (has "less magic" happening).

Generic Repository API

The repository API acts as an abstraction over the JDO/DataNucleus objectstore. You can use it during prototyping to write naive queries (find all rows, then filter using the Guava `Predicate` API, or you can use it to call JDO named queries using JDOQL.

As an alternative, you could also use JDO typesafe queries through the `IsisJdoSupport` service.

```
public interface DomainObjectContainer {
    public <T> List<T> allInstances(Class<T> ofType, long... range);
①
    <T> List<T> allMatches(Query<T> query);
②
    <T> List<T> allMatches(Class<T> ofType, Predicate<? super T> predicate, long...
range); ③
    <T> List<T> allMatches(Class<T> ofType, String title, long... range);
④
    <T> List<T> allMatches(Class<T> ofType, T pattern, long... range);
⑤
    ...
}
```

① all persisted instances of specified type. Mostly for prototyping, though can be useful to obtain all instances of domain entities if the number is known to be small. The optional varargs parameters are for paging control; more on this below.

② all persistence instances matching the specified `Query`. Query itself is an Isis abstraction on top of JDO/DataNucleus' Query API. **This is the primary API used for querying**

③ all persistenced instances of specified type matching `Predicate`. Only really intended for prototyping because in effect constitutes a client-side WHERE clause

④ all persisted instances with the specified string as their title. Only very occasionally used

⑤ all persisted instances matching object (query-by-example). Only very occasionally used

There are various implementations of the `Query` API, but these either duplicate functionality of the other overloads of `allMatches(…)` or they are not supported by the JDO/DataNucleus object store. The only significant implementation of `Query` to be aware of is `QueryDefault`, which identifies a named query and a set of parameter/argument tuples.

For example, in the (non-ASF) Isis addons' todoapp the `ToDoItem` is annotated:

```
@javax.jdo.annotations.Queries( {
    @javax.jdo.annotations.Query(
            name = "findByAtPathAndComplete", language = "JDOQL",              ①
            value = "SELECT "
                    + "FROM todoapp.dom.module.todoitem.ToDoItem "
                    + "WHERE atPath.indexOf(:atPath) == 0 "                    ②
                    + "   && complete == :complete"),                          ③
    ...
})
public class ToDoItem ... {
    ...
}
```

① name of the query

② defines the `atPath` parameter

③ defines the `complete` parameter

This JDO query definitions are used in the `ToDoItemRepositoryImplUsingJdoql` service:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class ToDoItemRepositoryImplUsingJdoql implements ToDoItemRepositoryImpl {
    @Programmatic
    public List<ToDoItem> findByAtPathAndCategory(final String atPath, final Category
category) {
        return container.allMatches(
                new QueryDefault<>(ToDoItem.class,
                        "findByAtPathAndCategory",                             ①
                        "atPath", atPath,                                      ②
                        "category", category));                                ③
    }
    ...
    @javax.inject.Inject
    DomainObjectContainer container;
}
```

①

corresponds to the "findByAtPathAndCategory" JDO named query

② provide argument for the `atPath` parameter. The pattern is parameter, argument, parameter, argument, ... and so on.

③ provide argument for the `category` parameter. The pattern is parameter, argument, parameter, argument, ... and so on.

Other JDOQL named queries (not shown) follow the exact same pattern.

With respect to the other query APIs, the varargs parameters are optional, but allow for (client-side and managed) paging. The first parameter is the `start` (0-based, the second is the `count`.

It is also possible to query using DataNucleus' type-safe query API. For more details, see `IsisJdoSupport`.

Object Persistence API

The persistence API is used to persist newly created objects (as per `#newTransientInstance(…)`, above and to delete (remove) objects that are persistent.

Note that there is no API for updating existing objects; the framework (or rather, JDO/DataNucleus) performs object dirty tracking and so any objects that are modified in the course of a request will be automatically updated).

```
public interface DomainObjectContainer {

    boolean isPersistent(Object domainObject);          ①
    boolean isViewModel(Object domainObject);           ②

    void persist(Object domainObject);                  ③
    void persistIfNotAlready(Object domainObject);      ④

    void remove(Object persistentDomainObject);         ⑤
    void removeIfNotAlready(Object domainObject);       ⑥

    boolean flush();                                    ⑦
    ...
}
```

① test whether a particular domain object is persistent or not.

② test whether a particular domain object is a view model or not. Note that this includes any domain objects annotated with `@DomainObject#nature=Nature.EXTERNAL_ENTITY)` or `@DomainObject#nature=Nature.INMEMORY_ENTITY`

③ persist a transient object. Note though that this will throw an exception if the object is already persistent; this can happen if JDO/DataNucleus's persistence-by-reachability is in effect. For this reason it is generally better to use `persistIfNotAlready(…)`. Also note that `persist(…)` has been deprecate. When moving to `RepositoryService#persist()` take into account that its behavior is identical to <4>, being a no-op if the object is persistent, instead of throwing an exception.

④ persist an object but only if know to not have been persistent. But if the object is persistent, is a no-op

⑤ remove (ie DELETE) a persistent object. For similar reasons to the persistence, it is generally better to use:

⑥ remove (ie DELETE) an object only if known to be persistent. But if the object has already been deleted, then is a no-op.

⑦ flushes all pending changes to the objectstore. Explained further below.

For example:

```
Customer cust = container.newTransientInstance(Customer.class);
cust.setFirstName("Freddie");
cust.setLastName("Mercury");
container.persistIfNotAlready(cust);
```

You should be aware that by default Apache Isis queues up calls to `#persist()` and `#remove()`. These are then executed either when the request completes (and the transaction commits), or if the queue is flushed. This can be done either implicitly by the framework, or as the result of a direct call to `#flush()`.

By default the framework itself will cause `#flush()` to be called whenever a query is executed by way of `#allMatches(Query)`, as documented above. However, this behaviour can be disabled using the configuration property `isis.services.container.disableAutoFlush`.

Messages API

The `DomainObjectContainer` allows domain objects to raise information, warning or error messages. These messages can either be simple strings, or can be translated.

```
public interface DomainObjectContainer {

    void informUser(String message);
①
    String informUser(TranslatableString message, Class<?> contextClass, String
contextMethod); ②

    void warnUser(String message);
③
    String warnUser(TranslatableString message, Class<?> contextClass, String
contextMethod);   ④

    void raiseError(String message);
⑤
    String raiseError(TranslatableString message, Class<?> contextClass, String
contextMethod); ⑥
    ...
}
```

① display as a transient message to the user (not requiring acknowledgement). In the Wicket viewer this is implemented as a toast that automatically disappears after a period of time.

② ditto, but with translatable string, for i18n support.

③ warn the user about a situation with the specified message. In the Wicket viewer this is implemented as a toast that must be closed by the end-user.

④ ditto, but with translatable string, for i18n support.

⑤ show the user an unexpected application error. In the Wicket viewer this is implemented as a toast (with a different colour) that must be closed by the end-user.

⑥ ditto, but with translatable string, for i18n support.

For example:

```
public Order addItem(Product product, @ParameterLayout(named="Quantity") int quantity)
{
    if(productRepository.stockLevel(product) == 0) {
        container.warnUser(
            product.getDescription() + " out of stock; order fulfillment may be
delayed");
    }
    ...
}
```

Security API

The security API allows the domain object to obtain the identity of the user interacting with said object.

```
public interface DomainObjectContainer {
    UserMemento getUser();
    ...
}
```

where in turn (the essence of) `UserMemento` is:

```
public final class UserMemento {
    public String getName() { ... }
    public boolean isCurrentUser(final String userName) { ... }

    public List<RoleMemento> getRoles() { ... }
    public boolean hasRole(final RoleMemento role) { ... }
    public boolean hasRole(final String roleName) { ... }
    ...
}
```

and `RoleMemento` is simpler still:

```
public final class RoleMemento {
    public String getName() { ... }
    public String getDescription() { ... }
    ...
}
```

The roles associated with the `UserMemento` will be based on the configured security (typically Shiro).

In addition, when using the Wicket viewer there will be an additional "org.apache.isis.viewer.wicket.roles.USER" role; this is used internally to restrict access to web pages without authenticating.

Presentation API

A responsibility of every domain object is to return a title. This can be done declaratively using the `@Title` annotation on property/ies, or it can be done imperatively by writing a `title()` method.

It's quite common for titles to be built up of the titles of other objects. If using building up the title using `@Title` then Apache Isis will automatically use the title of the objects referenced by the annotated properties. We also need programmatic access to these titles if going the imperative route.

Similarly, it often makes sense if raising messages to use the title of an object in a message rather (than a some other property of the object), because this is how end-users will be used to identifying the object.

The API defined by `DomainObjectContainer` is simply:

```
public interface DomainObjectContainer {
    String titleOf(Object domainObject);                ①
    String iconNameOf(Object domainObject);             ②
    ...
}
```

① return the title of the object, as rendered in the UI by the Apache Isis viewers.

② return the icon name of the object, as rendered in the UI by the Apache Isis viewers.

By way of example, here's some code from the (non-ASF) Isis addons' todoapp showing the use of the API in an message:

```
    public List<ToDoItem> delete() {
        final String title = container.titleOf(this);   ①
        ...
        container.removeIfNotAlready(this);
        container.informUser(
                TranslatableString.tr(
                    "Deleted {title}", "title", title), ②
                    this.getClass(), "delete");
        ...
    }
```

① the title is obtained first, because we're not allowed to reference object after it's been deleted

② use the title in an i18n `TranslatableString`

Properties API

The properties API allows domain objects to read the configuration properties aggregated from the various configuration files.

```
public interface DomainObjectContainer {
    String getProperty(String name);                        ①
    String getProperty(String name, String defaultValue);   ②
    List<String> getPropertyNames();                        ③
}
```

① Return the configuration property with the specified name; else return null.

② Return the configuration property with the specified name; if it doesn't exist then return the specified default value.

③ Return the names of all the available properties.

For example, here's a fictitious service that might wrap Twitter4J. say:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class TweetService {
    @Programmatic
    @PostConstruct
    public void init() {
        this.oauthConsumerKey = container.getProperty("tweetservice.oauth.consumerKey
");
        this.oauthConsumerSecret = container.getProperty(
"tweetservice.oauth.consumerSecret");
        this.oauthAccessToken = container.getProperty("tweetservice.oauth.accessToken
");
        this.oauthAccessTokenSecret = container.getProperty(
"tweetservice.oauth.accessTokenSecret");
    }
    ...
    @Inject
    DomainObjectContainer container;
}
```

If you *do* have a domain service that needs to access properties, then note that an alternative is to define a `@PostConstruct` method and pass in a `Map<String,String>` of properties. The two techniques are almost identical; it's mostly a matter of taste.

Services API

The services API allows your domain objects to programmatically inject services into arbitrary objects, as well as to look up services by type.

The methods are:

```
public interface DomainObjectContainer {
    <T> T injectServicesInto(final T domainObject);     ①
    <T> T lookupService(Class<T> service);              ②
    <T> Iterable<T> lookupServices(Class<T> service);   ③
    ...
}
```

① injects services into domain object; used extensively internally by the framework (eg to inject to other services, or to entities, or integration test instances, or fixture scripts). Service injection is done automatically if objects are created using `#newTransientInstance()`, described above

② returns the first registered service that implements the specified class

③ returns an `Iterable` in order to iterate over all registered services that implement the specified class

The primary use case is to instantiate domain objects using a regular constructor ("new is the new new") rather than using the `#newTransientInstance()` API, and then using the `#injectServicesInto(…`

) API to set up any dependencies.

For example:

```
Customer cust = container.injectServicesInto( new Customer());
cust.setFirstName("Freddie");
cust.setLastName("Mercury");
container.persist(cust);
```

Validation API

The intent of this API is to provide a mechanism where an object can programmatically check the state any class invariants. Specifically, this means the validating the current state of all properties, as well as any object-level validation defined by `validate()`.

These methods have been deprecated; this feature should be considered experimental and your mileage may vary.

The API provided is:

```
public interface DomainObjectContainer {
    boolean isValid(Object domainObject);
    String validate(Object domainObject);
    ...
}
```

7.3.2. Implementation

The core framework provides a default implementation of this service (`o.a.i.core.metamodel.services.container.DomainObjectContainerDefault`).

7.4. EventBusService

The `EventBusService` allows domain objects to emit events to subscribing domain services using an in-memory event bus.

The primary user of the service is the framework itself, which automatically emit events for actions, properties and collections. Multiple events are generated:

- when an object member is to be viewed, an event is fired; subscribers can veto (meaning that the member is hidden)
- when an object member is to be enabled, the same event instance is fired; subscribers can veto (meaning that the member is disabled, ie cannot be edited/invoked)
- when an object member is being validated, then a new event instance is fired; subscribers can veto (meaning that the candidate values/action arguments are rejected)
- when an object member is about to be changed, then the same event instance is fired;

subscribers can perform pre-execution operations

- when an object member has been changed, then the same event instance is fired; subscribers can perform post-execution operations

If a subscriber throws an exception in the first three steps, then the interaction is vetoed. If a subscriber throws an exception in the last two steps, then the transaction is aborted. For more on this topic, see `@Action#domainEvent()`, `@Property#domainEvent()` and `@Collection#domainEvent()`.

It is also possible for domain objects to programmatically generate domain events. However the events are published, the primary use case is to decoupling interactions from one module/package/namespace and another.

Two implementations are available, using either Guava's `EventBus`, or alternatively using the AxonFramework's SimpleEventBus. It is also possible to plug in a custom implementation.

7.4.1. API

The API defined by `EventBusService` is:

```
public abstract class EventBusService {
    @Programmatic
    public void post(Object event) { ... }                          ①
    @Programmatic
    public void register(final Object domainService) { ... }        ②
    @Programmatic
    public void unregister(final Object domainService) { ... }      ③
}
```

① posts the event onto event bus

② allows domain services to register themselves. This should be done in their `@PostConstruct` initialization method (for both singleton and `@RequestScoped` domain services.

③ exists for symmetry, but need never be called (it is in fact deliberately a no-op).

7.4.2. Registering Subscribers

The `register()` method should be called in the `@PostConstruct` lifecycle method. It is valid and probably the least confusing to readers to also "unregister" in the `@PreDestroy` lifecycle method (though as noted above, unregistering is actually a no-op).

For example:

```
@DomainService(nature=NatureOfService.DOMAIN)   ①
@DomainServiceLayout( menuOrder="1")            ②
public class MySubscribingDomainService {
    @PostConstruct
    public void postConstruct() {
        eventBusService.register(this);         ③
    }
    @PreDestroy
    public void preDestroy() {
        eventBusService.unregister(this);       ④
    }
    ...
    @javax.inject.Inject
    EventBusService eventBusService;
}
```

① subscribers are typically not visible in the UI, so specify a `DOMAIN` nature

② It's important that subscribers register before any domain services that might emit events on the event bus service. For example, the (non-ASF) Incode Platform's 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.

③ register with the event bus service during `@PostConstruct` initialization

④ corresponding deregister when shutting down

This works for both singleton (application-scoped) and also `@RequestScoped` domain services.

The `AbstractSubscriber` class automatically performs this registration. As a convenience, it is also annotated with the `@DomainServiceLayout#menuOrder()` attribute.

7.4.3. Annotating Members

As discussed in the introduction, the framework will automatically emit domain events for all of the object members (actions, properties or collections) of an object whenever that object is rendered or (more generally) interacted with.

For example:

```
public class Customer {
    @Action
    public Customer placeOrder(Product product, @ParameterLayout(named="Quantity") int
qty) { ... }
    ...
}
```

will propagate an instance of the default `o.a.i.applib.services.eventbus.ActionDomainEvent.Default` class. If using the Guava event bus this can be subscribed to using:

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

or if using Axonframework, the subscriber uses a different annotation:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class MySubscribingDomainService
    @Programmatic
    @org.axonframework.eventhandling.annotation.EventHandler
    public void on(ActionDomainEvent ev) { ... }
    ...
}
```

More commonly though you will probably want to emit domain events of a specific subtype. As a slightly more interesting example, suppose in a library domain that a `LibraryMember` wants to leave the library. A letter should be sent out detailing any books that they still have out on loan:

In the `LibraryMember` class, we publish the event by way of an annotation:

```
public class LibraryMember {
    @Action(domainEvent=LibraryMemberLeaveEvent.class)  ①
    public void leave() { ... }
    ...
}
```

① `LibraryMemberLeaveEvent` is a subclass of `o.a.i.applib.eventbus.ActionDomainEvent`. The topic of subclassing is discussed in more detail below.

Meanwhile, in the `BookRepository` domain service, we subscribe to the event and act upon it. For example:

```
public class BookRepository {
    @Programmatic
    @com.google.common.eventbus.Subscribe
    public void onLibraryMemberLeaving(LibraryMemberLeaveEvent e) {
        LibraryMember lm = e.getLibraryMember();
        List<Book> lentBooks = findBooksOnLoanFor(lm);
        if(!lentBooks.isEmpty()) {
            sendLetter(lm, lentBooks);
        }
    }
    ...
}
```

This design allows the `libraryMember` module to be decoupled from the `book` module.

7.4.4. Event hierarchy

By creating domain event subtypes we can be more semantically precise and in turn providesmore flexibility for subscribers: they can choose whether to be broadly applicable (by subscribing to a superclass) or to be tightly focussed (by subscribing to a subclass).

We recommend that you define event classes at (up to) four scopes:

- at the top "global" scope is the Apache Isis-defined `o.a.i.applib.event.ActionDomainEvent`
- for the "module" scope, create a static class to represent the module itself, and creating nested classes within
- for each "class" scope, create a nested static event class in the domain object's class for all of the domain object's actions
- for each "action" scope, create a nested static event class for that action, inheriting from the "domain object" class.

To put all that into code; at the module level we can define:

```
package com.mycompany.modules.libmem;
...
public static class LibMemModule {
    private LibMemModule() {}
    public abstract static class ActionDomainEvent<S>
                extends org.apache.isis.applib.event.ActionDomainEvent<S> {}
    ...                                                                          ①
    public abstract static class PropertyDomainEvent<S,T>
                extends org.apache.isis.applib.event.PropertyDomainEvent<S,T> {}
    public abstract static class CollectionDomainEvent<S,E>
                extends org.apache.isis.applib.event.CollectionDomainEvent<S,E> {}
}
```

① similar events for properties and collections should also be defined

For the class-level we can define:

```
public static class LibraryMember {
    public abstract static class ActionDomainEvent
            extends LibMemModule.ActionDomainEvent<LibraryMember> { }
    ...                                                                    ①
}
```

① similar events for properties and collections should also be defined

and finally at the action level we can define:

```
public class LibraryMember {
    public static class LeaveEvent extends LibraryMember.ActionDomainEvent { }
    @Action(domainEvent=LeaveEvent.class)
    public void leave() { ... }
    ...
}
```

The subscriber can subscribe either to the general superclass (as before), or to any of the classes in the hierarchy.

Variation (for contributing services)

A slight variation on this is to not fix the generic parameter at the class level, ie:

```
public static class LibraryMember {
    public abstract static class ActionDomainEvent<S>
            extends LibMemModule.ActionDomainEvent<S> { }
    ...
}
```

and instead parameterize down at the action level:

```
public class LibraryMember {
    public static class LeaveEvent
            extends LibraryMember.ActionDomainEvent<LibraryMember> { } ①
    }
    @Action(domainEvent=LeaveEvent.class)
    public void leave() { ... }
    ...
}
```

This then allows for other classes - in particular domain services contributing members - to also inherit from the class-level domain events.

7.4.5. Programmatic posting

To programmatically post an event, simply call `#post()`.

The `LibraryMember` example described above could for example be rewritten into:

```
public class LibraryMember {
    ...
    public void leave() {
        ...
        eventBusService.post(new LibraryMember.LeaveEvent(...));    ①
    }
    ...
}
```

① `LibraryMember.LeaveEvent` could be *any* class, not just a subclass of `o.a.i.applib.event.ActionDomainEvent`.

In practice we suspect there will be few cases where the programmatic approach is required rather than the declarative approach afforded by `@Action#domainEvent()` et al.

7.4.6. Using `WrapperFactory`

An alternative way to cause events to be posted is through the `WrapperFactory`. This is useful when you wish to enforce a (lack-of-) trust boundary between the caller and the callee.

For example, suppose that `Customer#placeOrder(…)` emits a `PlaceOrderEvent`, which is subscribed to by a `ReserveStockSubscriber`. This subscriber in turn calls `StockManagementService#reserveStock(…)`. Any business rules on `#reserveStock(…)` should be enforced.

In the `ReserveStockSubscriber`, we therefore use the `WrapperFactory`:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class ReserveStockSubscriber {
    @Programmatic
    @Subscribe
    public void on(Customer.PlaceOrderEvent ev) {
        wrapperFactory.wrap(stockManagementService)
                      .reserveStock(ev.getProduct(), ev.getQuantity());
    }
    ...
    @Inject
    StockManagementService stockManagementService;
    @Inject
    WrapperFactory wrapperFactory;
}
```

7.4.7. Implementation

The framework provides a default implementation of the service, `o.a.i.objectstore.jdo.datanucleus.service.eventbus.EventBusServiceJdo`.

Configuration Properties

The default implementation of this service defines the following configuration properties:

Property	Value (default value)	Description
`isis.services.eventbus.implementation`	`guava`, `axon`, FQCN (`guava`)	which implementation to use by the `EventBusService` as the underlying event bus. The implementation of `EventBusService` provided by Apache Isis will by default use Guava's `EventBus` as the underlying in-memory event bus. Alternatively the AxonFramework's SimpleEventBus can be used. [NOTE] .Guava vs Axon, which to use? ==== Guava actually queues up events; they are not guaranteed to be dispatched immediately. This generally is not problem, but can be for cases where the subscriber may in turn want to post its own events (using `WrapperFactory`). The Axon `SimpleEventBus`-based implementation on the other hand is fully synchronous; events are dispatched as soon as they are posted. This works well in all scenarios (that we have tested). ====
`isis.services.eventbus.allowLateRegistration`	`true`,`false` (`false`)	whether a domain service can register with the <code>EventBusService</code> after any events have posted. Late registration refers to the idea that a domain service can register itself with the <code>EventBusService</code> after events have been posted. Since domain services are set up at boot time, this almost certainly constitutes a bug in the code and so by default late registration is _not allowed. Setting the above property to <code>true</code> disables this check. Since this almost certainly constitutes a bug in application code, by default this is disallowed.

SPI

It is also possible to define use some other underlying event bus implementation, by implementing

the EventBusImplementation SPI:

```
public interface EventBusImplementation {
    void register(Object domainService);
    void unregister(Object domainService);
    void post(Object event);
}
```

As is probably obvious, the EventBusService just delegates down to these method calls when its own similarly named methods are called.

If you do provide your own implementation of this SPI, be aware that your subscribers will need to use whatever convention is required (eg different annotations) such that the events are correctly routed through to your subscribers.

If you have written your own implementation of the EventBusServiceImplementation SPI, then specify instead its fully-qualified class name:

```
isis.services.eventbus.implementation=com.mycompany.isis.MyEventBusServiceImplementati
on
```

7.4.8. Related Services

The EventBusService is intended for fine-grained publish/subscribe for object-to-object interactions within an Apache Isis domain object model. The event propagation is strictly in-memory, and there are no restrictions on the object acting as the event (it need not be serializable, for example).

The PublishingService meanwhile is intended for coarse-grained publish/subscribe for system-to-system interactions, from Apache Isis to some other system. Here the only events published are those that action invocations (for actions annotated with `@Action#publishing()`) and of changed objects (for objects annotated with `@DomainObject#publishing()`).

7.5. FactoryService

The FactoryService collects together methods for instantiating domain objects.

The methods in this service replace similar methods (now deprecated) in DomainObjectContainer.

7.5.1. API

The API of FactoryService is:

```
public interface FactoryService {
    <T> T instantiate(final Class<T> ofType);       ①
    <T> T mixin();                                  ②
}
```

① create a new non-persisted domain entity. Any services will be automatically injected into the service.

② programmatically instantiate a mixin, as annotated with `@Mixin` or `@DomainObject#nature()`.

The object is created in memory, but is not persisted. The benefits of using this method (instead of simply using the Java `new` keyword) are:

- any services will be injected into the object immediately (otherwise they will not be injected until the frameworkbecomes aware of the object, typically when it is persisted through the `RepositoryService`
- the default value for any properties (usually as specified by `defaultXxx()` supporting methods) will not be set and the `created()` callback will be called.

The corollary is: if your code never uses `defaultXxx()` or the `created()` callback, then you can just `new` up the object. The `ServiceRegistry` service can be used to inject services into the domain object.

7.5.2. Usage

For example:

```
Customer cust = factoryService.instantiate(Customer.class);
cust.setFirstName("Freddie");
cust.setLastName("Mercury");
repositoryService.persist(cust);
```

7.5.3. Implementation

The core framework provides a default implementation of this service (`o.a.i.core.metamodel.services.factory.FactoryServiceDefault`).

7.5.4. Related Services

The `RepositoryService` is often used in conjunction with the `FactoryService`, to persist domain objects after they have been instantiated and populated.

An alternative to using the factory service is to simply instantiate the object ("new is the new new") and then use the `ServiceRegistry` service to inject other domain services into the instantiated object.

7.6. Scratchpad

The `Scratchpad` (request-scoped) domain service allows objects to exchange information even if they

do not directly call each other.

7.6.1. API & Implementation

The API of `Scratchpad` service is:

```
@RequestScoped
public class Scratchpad {
    @Programmatic
    public Object get(Object key) { ... }
    @Programmatic
    public void put(Object key, Object value) { ... }
    @Programmatic
    public void clear() { ... }
}
```

This class (`o.a.i.applib.services.scratchpad.Scratchpad`) is also the implementation. And, as you can see, the service is just a request-scoped wrapper around a `java.util.Map`.

7.6.2. Usage

The most common use-case is for bulk actions that act upon multiple objects in a list. The (same) `Scratchpad` service is injected into each of these objects, and so they can use pass information.

For example, the Isis addons example todoapp (not ASF) demonstrates how the `Scratchpad` service can be used to calculate the total cost of the selected `ToDoItem`s:

```
@Action(
    semantics=SemanticsOf.SAFE,
    invokeOn=InvokeOn.COLLECTION_ONLY
)
public BigDecimal totalCost() {
    BigDecimal total = (BigDecimal) scratchpad.get("runningTotal");
    if(getCost() != null) {
        total = total != null ? total.add(getCost()) : getCost();
        scratchpad.put("runningTotal", total);
    }
    return total.setScale(2);
}
@Inject
Scratchpad scratchpad;
```

A more complex example could use a view model to enable bulk updates to a set of objects. The view model's job is to gather track of the items to be updated:

```
public class ToDoItemUpdateBulkUpdate extends AbstractViewModel {
    private List<ToDoItem> _items = ...;
    public ToDoItemBulkUpdate add(ToDoItem item) {
        _items.add(item);
        return this;
    }
    ...                 ①
}
```

① not shown - the implementation of `ViewModel` for converting the list of `_items` into a string.

The bulk action in the objects simply adds the selected item to the view model:

```
@Action(
    invokeOn=InvokeOn.COLLECTIONS_ONLY
    semantics=SemanticsOf.SAFE
)
public ToDoItemBulkUpdate bulkUpdate() {
    return lookupBulkUpdateViewModel().add(this);
}
private ToDoItemBulkUpdate lookupBulkUpdateViewModel() {
    ToDoItemBulkUpdate bulkUpdate =
        (ToDoItemBulkUpdate) scratchpad.get("bulkUpdateViewModel");     ①
    if(bulkUpdate == null) {
        bulkUpdate = container.injectServicesInto(new ToDoItemBulkUpdate());
        scratchpad.put("bulkUpdateViewModel", bulkUpdate);              ②
    }
    return bulkUpdate;
}
@Inject
Scratchpad scratchpad;
```

① look for the `ToDoItemBulkUpdate` in the scratchpad...

② ... and add one if there isn't one (ie for the first object returned).

If using the Wicket viewer, the `ToDoItemBulkUpdate` view model returned from the last action invoked will be displayed. Thereafter this view model can be used to perform a bulk update of the "enlisted" items.

7.6.3. Related Services

The `ActionInteractionContext` service allows bulk actions to co-ordinate with each other.

The `QueryResultsCache` is useful for caching the results of expensive method calls.

7.7. `UserService`

The `UserService` allows the domain object to obtain the identity of the user interacting with said

object.

If SudoService has been used to temporarily override the user and/or roles, then this service will report the overridden values instead.

The methods in this service replace similar methods (now deprecated) in DomainObjectContainer.

7.7.1. API and Usage

The API of UserService is:

```
public interface UserService {
    UserMemento getUser();
}
```

where in turn (the essence of) UserMemento is:

```
public final class UserMemento {
    public String getName() { ... }
    public boolean isCurrentUser(final String userName) { ... }

    public List<RoleMemento> getRoles() { ... }
    public boolean hasRole(final RoleMemento role) { ... }
    public boolean hasRole(final String roleName) { ... }
    ...
}
```

and RoleMemento is simpler still:

```
public final class RoleMemento {
    public String getName() { ... }
    public String getDescription() { ... }
    ...
}
```

The roles associated with the UserMemento will be based on the configured security (typically Shiro).

In addition, when using the Wicket viewer there will be an additional "org.apache.isis.viewer.wicket.roles.USER" role; this is used internally to restrict access to web pages without authenticating.

7.7.2. Implementation

The core framework provides a default implementation of this service (o.a.i.core.runtime.services.user.UserServiceDefault).

Chapter 8. Integration API

The integration APIs provide functionality to the domain objects to integrate with other bounded contexts, for example sending an email or serializing an object out to XML.

The table below summarizes the integration APIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 7. Integration API

API	Description	Implementation	Notes
o.a.i.applib.services.bookmark BookmarkService2	Convert object reference to a serializable "bookmark", and vice versa.	BookmarkServiceDefault o.a.i.core isis-core-metamodel	related services: BookmarkHolder-ActionContributions, BookmarkHolder-Association-Contributions
o.a.i.applib.services.email EmailService	Send a HTML email, optionally with attachments.	EmailServiceDefault o.a.i.core isis-core-runtime	
o.a.i.applib.services.jaxb JaxbService	Marshal and unmarshal JAXB-annotated view models to/from XML.	JaxbServiceDefault o.a.i.core isis-core-schema	
o.a.i.applib.services.memento MementoService	Capture a serializable memento of a set of primitives or bookmarks. Primarily used internally, eg in support of commands/auditing.	MementoServiceDefault o.a.i.core isis-core-runtime	
o.a.i.applib.services.xmlsnapshot XmlSnapshotService	Generate an XML representation of an object and optionally a graph of related objects.	XmlSnapshotServiceDefault o.a.i.core isis-core-runtime	

Key:

- o.a.i is an abbreviation for org.apache.isis

- `o.ia.m` is an abbreviation for `org.isisaddons.module`
- `o.a.i.c.m.s` is an abbreviation for `org.apache.isis.core.metamodel.services`
- `o.a.i.c.r.s` is an abbreviation for `org.apache.isis.core.runtime.services`

8.1. BookmarkService2

The `BookmarkService2` domain service (and its various supertypes) provides the ability to obtain a serializable `o.a.i.applib.bookmarks.Bookmark` for any (persisted) domain object, and to lookup domain objects given a `Bookmark`. This can then in turn be converted to and from a string.

For example, a `Customer` object with:

- an object type of "custmgmt.Customer" (as per `DomainObject#objectType()` or equivalent) , and
- an id=123

could correspond to a `Bookmark` with a string representation of `custmgmt.Customer|123`.

A `Bookmark` is little more than an API equivalent of Apache Isis' internal `Oid` (object identifier). Nevertheless, the ability to uniquely address *any* domain object within an Apache Isis system — to in effect provide a URN — is immensely useful.

For example, a `Bookmark` could be converted into a barcode, and then this used for automated scanning of correspondence from a customer.

`Bookmark`s are used by several other domain services as a means of storing a reference to an arbitrary object (a polymorphic relationship). For example, the (non-ASF) Incode Platform's auditing module's implementation of `AuditerService` uses bookmarks to capture the object that is being audited.

One downside of using `Bookmark`s is that there is no way for the JDO/DataNucleus objectstore to enforce any kind of referental integrity. However, the (non-ASF) Incode Platform's poly module describes and supports a design pattern to address this requirement.

8.1.1. API & Implementation

The API defined by `BookmarkService2` is:

```
public interface BookmarkService2 {
    enum FieldResetPolicy {                                                    ①
        RESET,
        DONT_RESET
    }
    Object lookup(BookmarkHolder bookmarkHolder, FieldResetPolicy policy);
    Object lookup(Bookmark bookmark, FieldResetPolicy policy);
    <T> T lookup(Bookmark bookmark, FieldResetPolicy policy, Class<T> cls);    ②
    Bookmark bookmarkFor(Object domainObject);
    Bookmark bookmarkFor(Class<?> cls, String identifier);
}
```

① if the object has already been loaded from the database, then whether to reset its fields. The default it to `RESET`.

② same as `lookup(Bookmark bookmark)`, but downcasts to the specified type.

The core framework provides a default implementation of this API, namely `o.a.i.core.metamodel.services.bookmarks.BookmarkServiceInternalDefault`

8.1.2. BookmarkHolder

The `BookmarkHolder` interface is intended to be implemented by domain objects that use a `Bookmark` to reference a (single) domain object; an example might be a class such as the audit entry, mentioned above. The interface is simply:

```
public interface BookmarkHolder {
    @Programmatic
    Bookmark bookmark();
}
```

There are two services that will contribute to this interface:

- `BookmarkHolderActionContributions` will provide a `lookup(…)` action
- `BookmarkHolderAssociationContributions` provides an `object` property.

Either of these can be suppressed, if required, using a vetoing subscriber. For example, to suppress the `object` property (so that only the `lookup(…)` action is ever shown for implementations of `BookmarkHolder`, define:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class AlwaysHideBookmarkHolderAssociationsObjectProperty {
    @Subscribe
    public void on(BookmarkHolderAssociationContributions.ObjectDomainEvent ev) {
        ev.hide();
    }
}
```

A more sophisticated implementation could look inside the passed `ev` argument and selectively hide or not based on the contributee.

8.1.3. Usage by other services

Bookmarks are used by the (non-ASF) Incode Platform's command module's implementation of `BackgroundCommandService`, which uses a bookmark to capture the target object on which an action will be invoked subsequently.

Bookmarks are also used by the (non-ASF) Incode Platform's auditing module's implementation of `AuditerService`.

8.2. `EmailService`

The `EmailService` provides the ability to send HTML emails, with attachments, to one or more recipients.

Apache Isis provides a default implementation to send emails using an external SMTP provider. Note that this must be configured (using a number of configuration properties) before it can be used. The that sends email as an HTML message, using an external SMTP provider.

8.2.1. API

The API for the service is:

```
public interface EmailService {
    boolean send(                                                   ①
        List<String> to, List<String> cc, List<String> bcc,         ②
        String subject,
        String body,                                                ③
        DataSource... attachments);
    boolean isConfigured();                                         ④
}
```

① is the main API to send the email (and optional attachments). Will return `false` if failed to send

② pass either `null` or `Collections.emptyList()` if not required

③ should be HTML text

④ indicates whether the implementation was configured and initialized correctly. If this returns `false` then any attempt to call `send(…)` will fail.

8.2.2. Implementation

As noted in the introduction, the core framework provides a default implementation, `EmailServiceDefault`. This sends email as an HTML message, using an external SMTP provider.

Configuration Properties

The default implementation defines the following configuration properties:

Property	Value (default value)	Description
`isis.service.email.override.bcc`	email address	intended to simplify testing, if specified then the email's `bcc` address will be that specified (rather than the email address(es) passed in as an argument to `EmailService#send(…)`). NB: note that the key is mis-spelt, (`isis.service.email` rather than `isis.services.email`)
`isis.service.email.override.cc`	email address	intended to simplify testing, if specified then the email's `cc` address will be that specified (rather than the email address(es) passed in as an argument to `EmailService#send(…)`). NB: note that the key is mis-spelt, (`isis.service.email` rather than `isis.services.email`)
`isis.service.email.override.to`	email address	intended to simplify testing, if specified then the email's `to` address will be that specified (rather than the email address(es) passed in as an argument to `EmailService#send(…)`). NB: note that the key is mis-spelt, (`isis.service.email` rather than `isis.services.email`)
`isis.service.email.port`	port number (`587`)	The port number for the SMTP service on the the external SMTP host (used by `EmailService`). NB: note that the key is mis-spelt, (`isis.service.email` rather than `isis.services.email`)
`isis.service.email.sender.address`	email address	The email address to use for sending out email (used by `EmailService`). **Mandatory**. NB: note that the key is mis-spelt, (`isis.service.email` rather than `isis.services.email`)
`isis.service.email.sender.hostname`	host (`smtp.gmail.com`)	The hostname of the external SMTP provider (used by `EmailService`). NB: note that the key is mis-spelt, (`isis.service.email` rather than `isis.services.email`)

Property	Value (default value)	Description
isis.service.email.sender.password	email password	The corresponding password for the email address to use for sending out email (used by EmailService). **Mandatory**. NB: note that the key is mis-spelt, (isis.service.email rather than isis.services.email)
isis.service.email.socketConnectionTimeout	milliseconds (2000)	The socket connection timeout NB: note that the key is mis-spelt, (isis.service.email rather than isis.services.email)
isis.service.email.socketTimeout	milliseconds (2000)	The socket timeout NB: note that the key is mis-spelt, (isis.service.email rather than isis.services.email)
isis.service.email.throwExceptionOnFail	true,false (true)	Whether to throw an exception if there the email cannot be sent (probably because of some misconfiguration). This behaviour is (now) the default; the old behaviour (of just returning false from the send() method) can be re-enabled by setting this property to false. NB: note that the key is mis-spelt, (isis.service.email rather than isis.services.email)
isis.service.email.tls.enabled	true,false (true)	Whether to enable TLS for the email SMTP connection (used by EmailService). NB: note that the key is mis-spelt, (isis.service.email rather than isis.services.email)

Thus, use this service the following properties must be configured:

- isis.service.email.sender.address
- isis.service.email.sender.password

and these properties may optionally be configured (each has a default to use gmail, documented here):

- isis.service.email.sender.hostname
- isis.service.email.port
- isis.service.email.tls.enabled

These configuration properties can be specified either in isis.properties or in an external configuration file, or programmatically using the AppManifest.

If prototyping (that is, running the app using org.apache.isis.WebServer), the configuration properties can also be specified as system properties. For example, if you create a test email account on gmail, you can configure the service using:

```
-Disis.service.email.sender.address=xxx@gmail.com
-Disis.service.email.sender.password=yyy
```

where "xxx" is the gmail user account and "yyy" is its password

In addition the following properties can be set:

- `isis.service.email.sender.username`

  Rather than authenticate using the sender address, instead use the specified username.

- `isis.service.email.throwExceptionOnFail`

  Whether to throw an exception if there the email cannot be sent (probably because of some misconfiguration). This behaviour is (now) the default; the old behaviour (of just returning `false` from the `send()` method) can be re-enabled by setting this property to `false`.

- `isis.service.email.override.to`

  Intended to simplify testing, if specified then the email's `to` address will be that specified (rather than the email address(es) passed in as an argument to `EmailService#send(…)`).

- `isis.service.email.override.cc`

  Similarly, to override the `cc` email address.

- `isis.service.email.override.to`

  Similarly, to override the `bcc` email address.

- `isis.service.email.socketTimeout`

  The socket timeout, defaulting to 2000ms.

- `isis.service.email.socketConnectionTimeout`

  The socket connection timeout, defaulting to 2000ms.

8.2.3. Alternative Implementations

If you wish to write an alternative implementation, be aware that it should process the message body as HTML (as opposed to plain text or any other format).

Also, note that (unlike most Apache Isis domain services) the implementation is also instantiated and injected by Google Guice. This is because `EmailService` is used as part of the user registration functionality and is used by Wicket pages that are accessed outside of the usual Apache Isis runtime. This implies a couple of additional constraints:

- first, implementation class should also be annotated with `@com.google.inject.Singleton`
- second, there may not be any Apache Isis session running. (If necessary, one can be created on

the fly using `IsisContext.doInSession(…)`)

To ensure that your alternative implementation takes the place of the default implementation, register it explicitly in `isis.properties`.

8.2.4. Related Services

The email service is used by the `EmailNotificationService` which is, in turn, used by `UserRegistrationService`.

8.3. JaxbService

The `JaxbService` allows instances of JAXB-annotated classes to be marshalled to XML and unmarshalled from XML back into domain objects.

8.3.1. API & Implementation

The API defined by `JaxbService` is:

```
public interface JaxbService {
    @Programmatic
    <T> T fromXml(Class<T> domainClass, String xml);
①
    @Programmatic
    public String toXml(final Object domainObject);
②
    public enum IsisSchemas {
③
        INCLUDE, IGNORE
    }
    @Programmatic
    public Map<String, String> toXsd(final Object domainObject, final IsisSchemas
isSchemas);} ④
}
```

① unmarshalls the XML into an instance of the class.

② marshalls the domain object into XML

③ whether to include or exclude the Isis schemas in the generated map of XSDs. Discussed further below.

④ generates a map of each of the schemas referenced; the key is the schema namespace, the value is the XML of the schema itself.

With respect to the `IsisSchemas` enum: a JAXB-annotated domain object will live in its own XSD namespace and may reference multiple other XSD schemas. In particular, many JAXB domain objects will reference the common Isis schemas (for example the `OidDto` class that represents a reference to a persistent entity). The enum indicates whether these schemas should be included or excluded from the map.

Isis provides a default implementation of the service, `o.a.i.schema.services.jaxb.JaxbServiceDefault`.

8.3.2. Usage within the framework

This service is provided as a convenience for applications, but is also used internally by the framework to `@XmlRootElement`-annotated view models. The functionality to download XML and XSD schemas is also exposed in the UI through mixins to `Dto` interface.

8.4. `MementoService` (deprecated)

The `MementoService` was originally introduced to simplify the implementation of ViewModels which are required by the framework to return string representation of all of their backing state, moreover which is safe for use within a URL. This usage is deprecated; use JAXB view models instead.

The service can also be used to create a memento of arbitrary objects, however this usage is also deprecated.

This service is deprecated, with replaced by internal domain services (not public API).

8.4.1. API & Implementation

The API defined by `MementoService` is:

```
@Deprecated
public interface MementoService {
    @Deprecated
    public static interface Memento {
        public Memento set(String name, Object value);
        public <T> T get(String name, Class<T> cls);
        public String asString();
        public Set<String> keySet();
    }
    public Memento create();
    public Memento parse(final String str);
    public boolean canSet(Object input);
}
```

The core framework provides a default implementation of this API, namely `o.a.i.c.r.services.memento.MementoServiceDefault`. The string returned (from `Memento#asString()`) is a base-64 URL encoded representation of the underlying format (an XML string).

In fact, the `MementoServiceDefault` implementation does provide a mechanism to disable the URL encoding, but this is not part of the `MementoService` public API. Note also that the encoding method is not pluggable.

The types of objects that are supported by the `MementoService` are implementation-specific, but would typically include all the usual value types as well as Apache Isis' `Bookmark` class (to represent references to arbitrary entities). Nulls can also be set.

In the case of the default implementation provided by the core framework, the types supported are:

- `java.lang.String`
- `java.lang.Boolean`, `boolean`
- `java.lang.Byte`, `byte`
- `java.lang.Short`, `short`
- `java.lang.Integer`, `int`
- `java.lang.Long`, `long`
- `java.lang.Float`, `float`
- `java.lang.Double`, `double`
- `java.lang.Character`, `char`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `org.joda.time.LocalDate`
- `org.apache.isis.applib.services.bookmark.Bookmark`

If using another implementation, the `canSet(…)` method can be used to check if the candidate object's type is supported.

8.4.2. Usage

As noted in the introduction, a common use case for this service is in the implementation of the `ViewModel` interface.

Rather than implementing `ViewModel`, it's usually easier to annotate your view models with `@ViewModel` (or equivalently `@DomainObject#nature=EXTERNAL_ENTITY` or `@DomainObject#nature=INMEMORY_ENTITY`.

For example, suppose you were implementing a view model that represents an external entity in a SOAP web service. To access this service the view model needs to store (say) the hostname, port number and an id to the object.

Using an injected `MementoService` the view model can roundtrip to and from this string, thus implementing the `ViewModel` API:

```
public class ExternalEntity implements ViewModel {
    private String hostname;
    private int port;
    private String id;
    public String viewModelMemento() {                    ①
        return mementoService.create()
                .set("hostname", hostname)
                .set("port", port)
                .set("id", id)
                .asString();
    }
    public void viewModelInit(String mementoStr) {        ②
        Memento memento = mementoService.parse(mementoStr);
        hostname = memento.get("hostname", String.class);
        port = memento.get("port", int.class);
        id = memento.get("id", String.class);
    ...
    @Inject
    MementoService mementoService;
}
```

① part of the `ViewModel` API

② part of the `ViewModel` API

8.5. XmlSnapshotService

The `XmlSnapshotService` provides the capability to generate XML snapshots (and if required corresponding XSD schemas) based on graphs of domain objects.

Typical use cases include creating mementos for business-focused auditing, such that a report could be generated as to which end-user performed a business action (perhaps for legal reasons). For one system that we know of, a digest of this snapshot of data is signed with the public encryption key so as to enforce non-repudiation.

Another use case is to grab raw data such that it could be merged into a report template or communication.

The service offers a basic API to create a snapshot of a single object, and an more flexible API that allows the size of the graph to be customized.

The core framework provides an implementation of this service (`o.a.i.core.runtime.services.xmlsnapshot.XmlSnapshotServiceDefault`).

8.5.1. Standard API

The (basic) API of `XmlSnapshotService` is:

```
public interface XmlSnapshotService {
    public interface Snapshot {
        Document getXmlDocument();
        Document getXsdDocument();
        String getXmlDocumentAsString();
        String getXsdDocumentAsString();
    }
    @Programmatic
    public XmlSnapshotService.Snapshot snapshotFor(Object domainObject);
    ...
}
```

The most straight-forward usage of this service is simply:

```
XmlSnapshot snapshot = xmlsnapshotService.snapshotFor(customer);
Element customerAsXml = snapshot.getXmlElement();
```

This will return an XML (document) element that contains the names and values of each of the customer's value properties, along with the titles of reference properties, and also the number of items in collections.

As well as obtaining the XML snapshot, it is also possible to obtain an XSD schema that the XML snapshot conforms to.

```
XmlSnapshot snapshot = ...;
Element customerAsXml = snapshot.getXmlElement();
Element customerXsd = snapshot.getXsdElement();
```

This can be useful for some tools. For example, Altova Stylevision can use the XML and XSD to transform into reports. Please note that this link does not imply endorsement (nor even a recommendation that this is a good design).

8.5.2. Builder API

The contents of the snapshot can be adjusted by including "paths" to other references or collections. To do this, the builder is used. The API for this is:

```
public interface XmlSnapshotService {
    ...
    public interface Builder {
        void includePath(final String path);
        void includePathAndAnnotation(String path, String annotation);
        XmlSnapshotService.Snapshot build();
    }
    @Programmatic
    public XmlSnapshotService.Builder builderFor(Object domainObject);
}
```

We start by obtaining a builder:

```
XmlSnapshot.Builder builder = xmlsnapshotService.builderFor(customer);
```

Suppose now that we want the snapshot to also include details of the customer's address, where `address` in this case is a reference property to an instance of the `Address` class. We can "walk-the-graph" by including these references within the builder.

```
builder.includePath("address");
```

We could then go further and include details of every order in the customer's `orders` collection, and details of every product of every order:

```
builder.includePath("orders/product");
```

When all paths are included, then the builder can build the snapshot:

```
XmlSnapshot snapshot = builder.build();
Element customerAsXml = snapshot.getXmlElement();
```

All of this can be strung together in a fluent API:

```
Element customerAsXml = xmlsnapshotService.builderFor(customer)
                        .includePath("address")
                        .includePath("orders/product")
                        .build()
                        .getXmlElement();
```

As you might imagine, the resultant XML document can get quite large very quickly with only a few "include"s.

If an XSD schema is beng generated (using `snapshot.getXsdElement()` then note that for the XSD to be correct, the object being snapshotted must have non-null values for the paths that are `include()'d. If this isn't done then the XSD will not be correct reflect for another snapshotted object that does have non-null values.

8.5.3. Automatic inclusions

If the domain object being snapshotted implements the `SnapshottableWithInclusions` interace, then this moves the responsibility for determining what is included within the snapshot from the caller to the snapshottable object itself:

```
public interface SnapshottableWithInclusions extends Snapshottable {
    List<String> snapshotInclusions();
}
```

If necessary, both approaches can be combined.

As an alternative to using `include()`, you might consider building a view model domain object which can reference only the relevant information required for the snapshot. For example, if only the 5 most recent Orders for a Customer were required, a `CustomerAndRecentOrders` view model could hold a collection of just those 5 `Order`s. Typically such view models would implement `SnapshottableWithInclusions`.

One reason for doing this is to provide a stable API between the domain model and whatever it is that might be consuming the XML. With a view model you can refactor the domain entities but still preserve a view model such that the XML is the same.

8.5.4. Convenience API

The `XmlSnapshotService` also provides some API for simply manipulating XML:

```
public interface XmlSnapshotService {
    ...
    @Programmatic
    public Document asDocument(String xmlStr);                          ①
    @Programmatic
    public <T> T getChildElementValue(                                  ②
                    Element el, String tagname, Class<T> expectedCls);
    @Programmatic
    public Element getChildElement(                                     ③
                    Element el, String tagname);
    @Programmatic
    public String getChildTextValue(Element el);                        ④
}
```

① is a convenience method to convert xml string back into a W3C Document

② is a convenience method to extract the value of an XML element, based on its type.

③ is a convenience method to walk XML document.

④ is a convenience method to obtain value of child text node.

8.5.5. Related Services

The `BookmarkService` provides a mechanism for obtaining a string representations of a single domain object.

The `MementoService` also provides a mechanism for generating string representations of domain objects.

The `JaxbService` is a simple wrapper around standard JAXB functionality for generating both XMLs and XSDs from JAXB-annotated classes. Note that there is built-in support for JAXB classes (ie annotated with `@XmlRootElement`) to be used as view models.

Chapter 9. Metadata API

The metadata APIs provide access to the framework's internal metamodel. These are generally of use to support development-time activities, for example creating custom UIs through Swagger.

The table below summarizes the metadata APIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 8. Metadata API

API	Description	Implementation	Notes
`o.a.i.applib.services.appfeat.ApplicationFeatureRepository`	Provides access to string representations of the features (package, class, class members) of the domain classes within the metamodel.	`ApplicationFeatureDefault` `o.a.i.core:isis-core-metamodel`	(not visible in UI)
`o.a.i.applib.services.layout.LayoutService`	Provides the ability to download `Xxx.layout.xml` files, in various styles.	`LayoutServiceDefault` `o.a.i.core:isis-core-metamodel`	Functionality surfaced in the UI through related mixin and menu.
`o.a.i.applib.services.metamodel.MetaModelService`	Access to certain information from the Apache Isis metamodel.	`MetaModelServiceDefault` `o.a.i.core:isis-core-metamodel`	Functionality surfaced in the UI through related menu.
`o.a.i.applib.services.registry.ServiceRegistry`	Methods to access and use other domain services.	`ServiceRegistry-Default` `o.a.i.core:isis-core-metamodel`	Supercedes methods in `DomainObjectContainer`.

API	Description	Implementation	Notes
o.a.i.applib.services.swagger.SwaggerService	Generates Swagger spec files to describe the public and/or private RESTful APIs exposed by the RestfulObjects viewer. These can then be used with the Swagger UI page to explore the REST API, or used to generate client-side stubs using the Swagger codegen tool, eg for use in a custom REST client app.	SwaggerServiceDefault o.a.i.core.isis-core-metamodel	A SwaggerServiceMenu domain service is also provided which enables the swagger spec to be downloaded. Apache Isis' Maven plugin also provides a swagger goal which allows the spec file(s) to be generated at build time (eg so that client-side stubs can then be generated in turn).

Key:

- o.a.i is an abbreviation for org.apache.isis
- o.ia.m is an abbreviation for org.isisaddons.module
- o.a.i.c.m.s is an abbreviation for org.apache.isis.core.metamodel.services
- o.a.i.c.r.s is an abbreviation for org.apache.isis.core.runtime.services

9.1. ApplicationFeatureRepository

The ApplicationFeatureRepository provides the access to string representations of the packages, classes and class members (collectively: "application features") of the domain classes within the Apache Isis' internal metamodel.

This functionality was originally implemented as part of (non-ASF) Incode Platform security module, where the string representations of the various features are used to represent permissions.

9.1.1. API

The API defined by the service is:

```
public interface ApplicationFeatureRepository {
    List<String> packageNames();
    List<String> packageNamesContainingClasses(ApplicationMemberType memberType);
    List<String> classNamesContainedIn(String packageFqn, ApplicationMemberType
memberType);
    List<String> classNamesRecursivelyContainedIn(String packageFqn);
    List<String> memberNamesOf(String packageFqn, String className,
ApplicationMemberType memberType);
}
```

where `ApplicationMemberType` in turn is:

```
public enum ApplicationMemberType {
    PROPERTY,
    COLLECTION,
    ACTION;
}
```

These methods are designed primarily to return lists of strings for use in drop-downs.

9.1.2. Implementation

The default implementation of this service is `ApplicationFeatureRepositoryDefault`.

Configuration Properties

The default implementation of this domain service supports the following configuration properties:

Property	Value (default value)	Description
`isis.services.applicationFeatures.init`	`lazy`, `eager` (`lazy`)	Whether the application features repository (which surfaces the framework's metamodel) should be initialized lazily or eagerly. Lazy initialization can speed up bootstrapping, useful while developing and running tests.

Related Services

The default implementation of this service uses the `ApplicationFeatureFactory` service to instantiate `ApplicationFeature` instances.

9.2. `LayoutService`

The `LayoutService` provides the ability to obtain the XML layout for a single domain object or for all domain objects. This functionality is surfaced through the user interface through a related mixin and menu action.

9.2.1. API & Implementation

The API defined by `LayoutService` is:

```
public interface LayoutService {
    String toXml(Class<?> domainClass, Style style);        ①
    byte[] toZip(Style style);                              ②
}
```

① Returns the serialized XML form of the layout (grid) for the specified domain class, in specified style (discussed below).

② Returns (a byte array) of a zip of the serialized XML of the layouts (grids), for all domain entities and view models.

The `Style` enum is defined as:

```
enum Style {
    CURRENT,
    COMPLETE,
    NORMALIZED,
    MINIMAL
}
```

The `CURRENT` style corresponds to the layout already loaded for the domain class, typically from an already persisted `layout.xml` file. The other three styles allow the developer to choose how much metadata is to be specified in the XML, and how much (if any) will be obtained elsewhere, typically from annotations in the metamodel (but also from `.layout.json` file if present). The table below summarises the choices:

Table 9. Table caption

Style	`@MemberGroupLayout`	`@MemberOrder`	`@ActionLayout`, `@PropertyLayout`, `@CollectionLayout`
`COMPLETE`	serialized as XML	serialized as XML	serialized as XML
`NORMALIZED`	serialized as XML	serialized as XML	not in the XML

Style	`@MemberGroupLayout`	`@MemberOrder`	`@ActionLayout`, `@PropertyLayout`, `@CollectionLayout`
`MINIMAL`	serialized as XML	not in the XML	not in the XML

As a developer, you therefore have a choice as to how you provide the metadata required for customised layouts:

- if you want all layout metadata to be read from the `.layout.xml` file, then download the "complete" version, and copy the file alongside the domain class. You can then remove all `@MemberGroupLayout`, `@MemberOrder`, `@ActionLayout`, `@PropertyLayout` and `@CollectionLayout` annotations from the source code of the domain class.
- if you want to use layout XML file to describe the grid (columns, tabs etc) and specify which object members are associated with those regions of the grid, then download the "normalized" version. You can then remove the `@MemberGroupLayout` and `@MemberOrder` annotations from the source code of the domain class, but retain the `@ActionLayout`, `@PropertyLayout` and `@CollectionLayout` annotations.
- if you want to use layout XML file ONLY to describe the grid, then download the "minimal" version. The grid regions will be empty in this version, and the framework will use the `@MemberOrder` annotation to bind object members to those regions. The only annotation that can be safely removed from the source code with this style is the `@MemberGroupLayout` annotation.

9.2.2. Related Mixins and Menus

The service's functionality is exposed in the UI through a mixin (per object) and a menu action (for all objects):

- the `Object` mixin provides the ability to download the XML layout for any domain object (entity or view model).
- the `LayoutServiceMenu` provides the ability to download all XML layouts as a single ZIP file (in any of the three styles).

The XML can then be copied into the codebase of the application, and annotations in the domain classes removed as desired.

9.2.3. Related Domain Services

The `GridService` is responsible for loading and normalizing layout XML for a domain class. It in turn uses the `GridLoaderService` and `GridSystemService` services.

9.3. `MetaModelService5`

The `MetaModelService5` service (and its various supertypes) provides access to a number of aspects of Apache Isis' internal metamodel.

9.3.1. API

The API defined by the service is:

```
public interface MetaModelService4 {
    Class<?> fromObjectType(String objectType);          ①
    String toObjectType(Class<?> domainType);            ②
    void rebuild(Class<?> domainType);                   ③
    List<DomainMember> export();                         ④

    // introduced in MetaModelService2
    enum Sort {                                          ⑤
        VIEW_MODEL, JDO_ENTITY, DOMAIN_SERVICE,
        MIXIN, VALUE, COLLECTION, UNKNOWN;
    }
    enum Mode {
        STRICT,
        RELAXED
    }
    Sort sortOf(Class<?> domainType);                    ⑥
    Sort sortOf(Bookmark bookmark);

    // introduced in MetaModelService3
    Sort sortOf(Class<?> domainType, Mode mode);
    Sort sortOf(Bookmark bookmark, Mode mode);

    // introduced in MetaModelService4
    AppManifest getAppManifest();                        ⑦
    AppManifest2 getAppManifest2();

    // introduced in MetaModelService5
    CommandDtoProcessor commandDtoProcessorFor(          ⑧
                        String memberIdentifier);

}
```

① reverse lookup of a domain class' object type

② lookup of a domain class' object type

③ invalidate and rebuild the internal metadata (an `ObjectSpecification`) for the specified domain type.

④ returns a list of representations of each of member of each domain class.

⑤ what sort of object a domain type is (or bookmark) represents

⑥ whether to throw an exception or return `Sort.UNKNOWN` if the object type is not recognized. (The overloads with no `Mode` parameter default to strict mode).

⑦ returns the `AppManifest` used to bootstrap the application. If an `AppManifest2` was used (from a `Module`), then this is also returned (else just `null`).

⑧ obtain an implementation of `CommandDtoProcessor` (if any) as per an

`@Action#commandDtoProcessor()` or `@Property#commandDtoProcessor()`.

This is used by the framework-provided implementations of `ContentMappingService`.

9.3.2. Implementation

The framework provides a default implementation of this service, `o.a.i.c.m.services.metamodel.MetaModelServiceDefault`.

9.3.3. Related Services

The `MetaModelServiceMenu` provides a method to download all domain members as a CSV. Internally this calls `MetaModelService#export()`. Under the covers this uses the API provided by the `ApplicationFeatureRepository` domain service.

9.4. `ServiceRegistry2`

The `ServiceRegistry2` domain service (and its various supertypes)collects together methods for injecting or looking up domain services (either provided by the framework or application-specific) currently known to the runtime.

The methods in this service replace similar methods (now deprecated) in `DomainObjectContainer`.

9.4.1. API

The API of `ServiceRegistry2` is:

```
public interface ServiceRegistry2 {
    <T> T injectServicesInto(final T domainObject);     ①
    <T> T lookupService(Class<T> service);              ②
    <T> Iterable<T> lookupServices(Class<T> service);   ③
    List<Object> getRegisteredServices();               ④
}
```

① injects services into domain object; used extensively internally by the framework (eg to inject to other services, or to entities, or integration test instances, or fixture scripts).

② returns the first registered service that implements the specified class

③ returns an `Iterable` in order to iterate over all registered services that implement the specified class

④ returns the list of all domain services that constitute the running application (including internal domain services).

Service injection is done automatically if objects are created using the `FactoryService`.

9.4.2. Usage

The primary use case is to instantiate domain objects using a regular constructor ("new is the new new"), and then using the `#injectServicesInto(…)` API to set up any dependencies.

For example:

```
Customer cust = serviceRegistry.injectServicesInto( new Customer());
cust.setFirstName("Freddie");
cust.setLastName("Mercury");
repositoryService.persist(cust);
```

The alternative is to use the `FactoryService` API which performs both steps in a single factory method.

9.4.3. Implementation

The core framework provides a default implementation of this service (`o.a.i.core.runtime.services.registry.ServiceRegistryDefault`).

9.5. `SwaggerService`

The `SwaggerService` generates Swagger spec files to describe the public and/or private RESTful APIs exposed by the RestfulObjects viewer.

These spec files can then be used with the Swagger UI page to explore the REST API, or used to generate client-side stubs using the Swagger codegen tool, eg for use in a custom REST client app.

Not all of the REST API exposed by the Restful Objects viewer is included in the Swagger schema definition files; the emphasis is those REST resources that are used to develop custom apps: domain objects, domain object collections and action invocations. When combined with Apache Isis' own simplified representations, these are pretty much all that is needed for this use case.

9.5.1. API & Implementation

The API defined by `SwaggerService` is:

```
public interface SwaggerService {
    enum Visibility {
        PUBLIC,                     ①
        PRIVATE,                    ②
        PRIVATE_WITH_PROTOTYPING;   ③
    }
    enum Format {                   ④
        JSON,
        YAML
    }
    String generateSwaggerSpec(final Visibility visibility, final Format format);
}
```

① Generate a Swagger spec for use by third-party clients, ie public use. This specification is restricted only to view models and to domain services with a nature of `VIEW_REST_ONLY`.

② Generate a Swagger spec for use only by internally-managed clients, ie private internal use. This specification includes domain entities and all menu domain services (as well as any view models).

③ Generate a Swagger spec that is the same as private case (above), but also including any prototype actions.

④ Swagger specs can be written either in JSON or YAML format.

Isis provides a default implementation of the service, `o.a.i.core.metamodel.services.swagger.SwaggerServiceDefault`.

9.5.2. Usage within the framework

This service is provided as a convenience for applications, it is not (currently) used by the framework itself.

9.5.3. Related Services

A `SwaggerServiceMenu` domain service provides a prototype action that enables the swagger spec to be downloaded from the Wicket viewer's UI.

Apache Isis' Maven plugin also provides a swagger goal which allows the spec file(s) to be generated at build time. this then allows client-side stubs can then be generated in turn as part of a build pipeline.

Chapter 10. Testing

The testing APIs provide functionality to domain objects for use when testing or demoing an application.

The testing SPIs allow the framework to provide supporting functionality for use when testing or demoing an application.

The table below summarizes the testing APIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 10. Testing API

API	Description	Implementation	Notes
`o.a.i.applib.fixturescripts.ExecutionParametersService`	...	`ExecutionParametersService` `o.a.i.core isis-core-applib`	API is also a concrete class
`o.a.i.applib.fixturescripts FixtureScripts`	Provides the ability to execute fixture scripts.	`FixtureScriptsDefault` `o.a.i.core isis-core-applib`	Default implementation uses `FixtureScripts-SpecificationProvider`.
`o.a.i.applib.services.fixturespec FixtureScripts-SpecificationProvider`	Provides settings for `FixtureScripts` default domain service (`FixtureScriptsDefault`) for executing fixture scripts.		
`o.a.i.applib.services.sudo SudoService`	For use in testing while running fixture scripts, allows a block of code to run as a specified user account.	`SudoServiceDefault` `o.a.i.core isis-core-runtime`	API is also a concrete class
`o.a.i.applib.fixtures.switchuser SwitchUserServiceService`	(deprecated)	`SwitchUserServiceImpl` `o.a.i.core isis-core-runtime`	

The table below summarizes the testing SPIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 11. Testing SPI

SPI	Description	Implementation	Notes

10.1. ExecutionParametersService

The `ExecutionParametersService` is used by the framework simply to instantiate the `ExecutionParameters` object. The `ExecutionParameters` object in turn is responsible for parsing the string parameter passed when executing fixtures through the UI to the `FixtureScripts` domain service.

10.1.1. API & Implementation

The API and implementation of this service is simply:

```
public class ExecutionParametersService {
    public ExecutionParameters newExecutionParameters(final String parameters) {
        return new ExecutionParameters(parameters);
    }
}
```

10.2. FixtureScripts

The `FixtureScripts` service provides the ability to execute fixture scripts .

The default implementation of this service, `FixtureScriptsDefault`, uses the associated `FixtureScriptsSpecificationProvider` to obtain a `FixtureScriptsSpecification`. This configures this service, for example telling it which package to search for `FixtureScript` classes, how to execute those classes, and hints that influence the UI.

10.2.1. API

The API for the service is:

```
public abstract class FixtureScripts ... {
    @Programmatic
    public List<FixtureResult> runFixtureScript(
        FixtureScript fixtureScript,
        String parameters) { ... }
}
```

10.2.2. Implementation

The default implementation is `o.a.i.applib.services.fixturespec.FixtureScriptsDefault`

Configuration Properties

The default implementation of this domain service supports the following configuration properties:

Table 12. Core Configuration Properties for Fixture Events

Property	Value (default value)	Description
`isis.fixtures.fireEvents`	`true`,`false` (`true`)	Whether fixture `FixturesInstallingEvent` and `FixturesInstalledEvent` events should be posted while the system is bootstrapping. Fixture events are fired to indicate the start and end of fixtures are being installed. This are listened to by the `QueryResultsCache` to disable caching during this period.

Related Services

The default implementation of this domain service interacts with `FixtureScriptsSpecificationProvider`.

10.3. `FixtureScriptsSpec'nProvider`

The `FixtureScriptsSpecificationProvider` configures the `FixtureScripts` domain service, providing the location to search for fixture scripts and other settings.

The service is used only by the default implementation of `FixtureScripts`, namely `FixtureScriptsDefault`.

Of the two designs, we encourage you to implement this "provider" SPI rather than subclass `FixtureScripts`. The primary benefit (apart from decoupling responsibilities) is that it ensures that there is always an instance of `FixtureScripts` available for use.

10.3.1. SPI

The SPI defined by the service is:

```
public interface FixtureScriptsSpecificationProvider {
    @Programmatic
    FixtureScriptsSpecification getSpecification();
}
```

where `FixtureScriptsSpecification` exposes these values:

```
public class FixtureScriptsSpecification {
    public String getPackagePrefix() { ... }
    public FixtureScripts.NonPersistedObjectsStrategy getNonPersistedObjectsStrategy()
{ ... }
    public FixtureScripts.MultipleExecutionStrategy getMultipleExecutionStrategy() {
... }
    public Class<? extends FixtureScript> getRunScriptDefaultScriptClass() { ... }
    public DropDownPolicy getRunScriptDropDownPolicy() { ... }
    public Class<? extends FixtureScript> getRecreateScriptClass() { ... }
    ...
}
```

The class is immutable but it has a builder (obtained using `FixturescriptsSpecification.builder(…)`) for a fluent API.

10.3.2. Implementation

The SimpleApp archetype has a simple implementation of this service:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class DomainAppFixturesProvider implements FixtureScriptsSpecificationProvider
{
    @Override
    public FixtureScriptsSpecification getSpecification() {
        return FixtureScriptsSpecification
                .builder(DomainAppFixturesProvider.class)
                .with(FixtureScripts.MultipleExecutionStrategy.EXECUTE)
                .withRunScriptDefault(RecreateSimpleObjects.class)
                .withRunScriptDropDown(FixtureScriptsSpecification.DropDownPolicy
.CHOICES)
                .withRecreate(RecreateSimpleObjects.class)
                .build();
    }
}
```

10.4. SudoService

The `SudoService` allows the current user reported by the `UserService` to be temporarily changed to some other user. This is useful both for integration testing (eg if testing a workflow system whereby objects are moved from one user to another) and while running fixture scripts (eg setting up objects that would normally require several users to have acted upon the objects).

10.4.1. API

The API provided by the service is:

```
public interface SudoService {
    @Programmatic
    void sudo(String username, final Runnable runnable);
    @Programmatic
    <T> T sudo(String username, final Callable<T> callable);
    @Programmatic
    void sudo(String username, List<String> roles, final Runnable runnable);
    @Programmatic
    <T> T sudo(String username, List<String> roles, final Callable<T> callable);
}
```

which will run the provided block of code (a `Runnable` or a `Callable`) in a way such that calls to `UserService#getUser()` will return the specified user (and roles, if specified). (If roles are not specified, then the roles of the current user are preserved).

The current user/role reported by the internal `AuthenticationSessionProvider` will also return the specified user/roles.

Note however that this the "effective user" does not propagate through to the Shiro security mechanism, which will continue to be evaluated according to the permissions of the current user. See the `ACCESS-ALL-ROLE` below for details of how to circumvent this.

10.4.2. Implementation

The core framework provides a default implementation of this service (`o.a.i.core.runtime.services.sudo.SudoServiceDefault`).

10.4.3. Usage

A good example can be found in the (non-ASF) Isis addons' todoapp which uses the `SudoService` in a fixture script to set up `ToDoItem` objects:

```
protected void execute(final ExecutionContext ec) {
    ...
    sudoService.sudo(getUsername(),
            new Runnable() {
                @Override
                public void run() {
                    wrap(toDoItem).completed();
                }
            });
    ...
}
```

ACCESS_ALL_ROLE

When `sudo(…)` is called the "effective user" is reported by both `UserService` and by `AuthenticationSessionProvider`, but does not propagate through to the Shiro security mechanism. These continue to be evaluated according to the permissions of the current user.

This can be a problem in certain use cases. For example if running a fixture script (which uses the `WrapperFactory`) from within an implementation of `UserRegistrationService`, this is likely to result in `HiddenException`s being thrown because there is no effective user.

In such cases, permission checking can simply be disabled by specifying `SudoService.ACCESS_ALL_ROLE` as one of the roles. For example:

```
protected void execute(final ExecutionContext ec) {
    ...
    sudoService.sudo(getUsername(), Arrays.asList(SudoService.ACCESS_ALL_ROLE),
            new Runnable() {
                @Override
                public void run() {
                    wrap(toDoItem).completed();
                }
            });
    ...
}
```

In the future this service may be used more deeply, eg to propagate permissions through to the Shiro security mechanism also.

10.4.4. SPI

The `SudoService.Spi` service allows implementations of `SudoService` to notify other services/components that the effective user and roles are different. The default implementation of `UserService` has been refactored to leverage this SPI.

```
public interface SudoService {
    ...
    interface Spi {
        void runAs(String username, List<String> roles);        ①
        void releaseRunAs();                                    ②
    }
}
```

① Called by `SudoService#sudo(…)`, prior to invoking its `Runnable` or `Callable`.

② Called by `SudoService#sudo(…)`, after its `Runnable` or `Callable` has been invoked.

The names of these methods were chosen based on similar names within Shiro.

10.5. `SwitchUserService` (deprecated)

The `SwitchUserService` domain service provides the ability to install fixtures changing the effective user half-way through. For example, this allows the setup of a test of a workflow system which checks that work is moved between different users of the system.

This service is deprecated; use fixture scripts and the `SudoService` instead.

10.5.1. API

The API of this service:

```
public class SwitchUserService {
    void switchUser(String username, String... roles);      ①
    void switchUser(String username, List<String> roles);   ①
}
```

① Switches the current user with the list of specified roles.

10.5.2. Implementation

The framework provides a default implementation of this service: `SwitchUserServiceImpl` in `isis-core-runtime`

Chapter 11. Persistence Layer API

The persistence layer APIs provide domain objects with tools to manage the interactions with the persistence layer, for example adding on-the-fly caching to queries that are called many times within a loop.

The table below summarizes the persistence layer APIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 13. Persistence Layer API

API	Description	Implementation	Notes
`o.a.i.applib.services.jdosupport.IsisJdoSupport`	Lower level access to the JDO Persistence API.	`IsisJdoSupportImpl` `o.a.i.core:isis-core-runtime`	
`o.a.i.applib.services.metrics.MetricsService`	Gathers and provides metrics on the numbers of objects used within a transaction.	`MetricsServiceDefault` `o.a.i.core:isis-core-runtime`	
`o.a.i.applib.services.queryresultscache.QueryResultsCache`	Request-scoped caching of the results of queries (or any data set generated by a given set of input arguments).	`QueryResultsCache` `o.a.i.core:isis-core-applib`	API is also a concrete class
`o.a.i.applib.services.repository.RepositoryService`	Methods to help implement repositories: query for existing objects, persist new or delete existing objects	`RepositoryService-Default` `o.a.i.core:isis-core-metamodel`	Supercedes methods in `DomainObjectContainer`.

Key:

- `o.a.i` is an abbreviation for `org.apache.isis`
- `o.ia.m` is an abbreviation for `org.isisaddons.module`
- `o.a.i.c.m.s` is an abbreviation for `org.apache.isis.core.metamodel.services`
- `o.a.i.c.r.s` is an abbreviation for `org.apache.isis.core.runtime.services`

11.1. HsqlDbManagerMenu

The `HsqlDbManagerMenu` provides a single menu item to open up the HSQLDB manager. This is only enabled for prototyping, and if HSQLDB is detected in the underlying JDBC URL. The menu appears

under the "Prototyping" menu.

11.1.1. API & Implementation

The API of the service is:

```
public class HsqlDbManagerMenu {
    public void hsqlDbManager() { ... }
}
```

Note that this launches the manager on the same host that the webapp runs, and so is only appropriate to use when running on `localhost`.

11.1.2. Disabling/hiding the menu

The menu can be hidden or disabled by subscribing to its domain event, eg:

```
@DomainService(nature=DOMAIN)
public void HideHsqlDbManagerMenu extends AbstractSubscriber {

    @EventHandler @Subscribe
    public void on(HsqlDbManagerMenu.ActionDomainEvent ev) {
        ev.hide();
    }
}
```

11.2. `IsisJdoSupport`

The `IsisJdoSupport` service provides a number of general purpose methods for working with the JDO/DataNucleus objectstore. In general these act at a lower-level of abstraction than the APIs normally used (specifically, those of `DomainObjectContainer`), but nevertheless deal with some of the most common use cases. For service also provides access to the underlying JDO `PersistenceManager` for full control.

The following sections discuss the functionality provided by the service, broken out into categories.

11.2.1. Executing SQL

You can use the `IsisJdoSupportService` to perform arbitrary SQL SELECTs or UPDATEs:

```
public interface IsisJdoSupport {
    @Programmatic
    List<Map<String, Object>> executeSql(String sql);
    @Programmatic
    Integer executeUpdate(String sql);
    ...
}
```

The `executeSql(…)` method allows arbitrary SQL `SELECT` queries to be submitted:

```
List<Map<String, Object>> results = isisJdoSupport.executeSql("select * from
custMgmt.customers");
```

The result set is automatically converted into a list of maps, where the map key is the column name.

In a similar manner, the `executeUpdate(…)` allows arbitrary SQL `UPDATE`s to be performed.

```
int count = isisJdoSupport.executeUpdate("select count(*) from custMgmt.customers);
```

The returned value is the number of rows updated.

As an alternative, consider using DataNucleus' type-safe JDO query API, discussed below.

11.2.2. Type-safe JDOQL Queries

DataNucleus provides an extension to JDO, so that JDOQL queries can be built up and executed using a set of type-safe classes.

The types in question for type safe queries are not the domain entities, but rather are companion "Q..." query classes. These classes are generated dynamically by an annotation processor as a side-effect of compilation, one "Q..." class for each of the `@PersistenceCapable` domain entity in your application. For example, a `ToDoItem` domain entity will give rise to a `QToDoItem` query class. These "Q..." classes mirror the structure of domain entity, but expose properties that allow predicates to be built up for querying instances, as well as other functions in support of order by, group by and other clauses.

The IntelliJ IDE automatically enables annotation processing by default, as does Maven. Using Eclipse IDE you may need to configure annotation processing manually; see the Developers' Guide. The DataNucleus' documentation offers some guidance on confirming that APT is enabled.

The `IsisJdoSupport` service offers two methods at different levels of abstraction:

```
public interface IsisJdoSupport {
    @Programmatic
    <T> List<T> executeQuery(final Class<T> cls, final BooleanExpression be);
    @Programmatic
    <T> T executeQueryUnique(final Class<T> cls, final BooleanExpression be);
    @Programmatic
    <T> TypesafeQuery<T> newTypesafeQuery(Class<T> cls);
    ...
}
```

The `executeQuery(…)` method supports the common case of obtaining a set of objects that meet some criteria, filtered using the provided `BooleanExpression`. To avoid memory leaks, the returned list is cloned and the underlying query closed.

For example, in the (non-ASF) Isis addons' todoapp there is an implementation of `ToDoItemRepository` using type-safe queries. The following JDOQL:

```
SELECT
FROM todoapp.dom.module.todoitem.ToDoItem
WHERE atPath.indexOf(:atPath) == 0
   && complete == :complete"
```

can be expressed using type-safe queries as follows:

```
public List<ToDoItem> findByAtPathAndCategory(final String atPath, final Category
category) {
    final QToDoItem q = QToDoItem.candidate();
    return isisJdoSupport.executeQuery(ToDoItem.class,
            q.atPath.eq(atPath).and(
            q.category.eq(category)));
}
```

You can find the full example of the JDOQL equivalent in the `DomainObjectContainer`

The `executeUniqueQuery(…)` method (introduced in `1.15.0`) is similar to `executeQuery(…)`, however expects the query to return at most a single object, which it returns (or `null` if none).

The `newTypesafeQuery(…)` method is a lower-level API that allows a type safe query to be instantiated for most sophisticated querying, eg using group by or order by clauses. See the DataNucleus documentation for full details of using this.

One thing to be aware of is that after the query has been executed, it should be closed, using `query.closeAll()`. If calling `query.executeList()` we also recommend cloning the resultant list first. The following utility method does both of these tasks:

```
private static <T> List<T> executeListAndClose(final TypesafeQuery<T> query) {
    final List<T> elements = query.executeList();
    final List<T> list = Lists.newArrayList(elements);
    query.closeAll();
    return list;
}
```

11.2.3. Fixture support

When writing integration tests you'll usually need to tear down some/all mutable transactional data before each test. One way to do that is to use the `executeUpdate(⋯)` method described above.

Alternatively, the `deleteAll(⋯)` method will let your test delete all instances of a class without resorting to SQL:

```
public interface IsisJdoSupport {
    @Programmatic
    void deleteAll(Class<?>... pcClasses);
    ...
}
```

For example:

```
public class TearDownAll extends FixtureScriptAbstract {
    @Override
    protected void execute(final ExecutionContext ec) {
        isisJdoSupport.deleteAll(Order.class);
        isisJdoSupport.deleteAll(CustomerAddress.class);
        isisJdoSupport.deleteAll(Customer.class);
    }
    @Inject
    IsisJdoSupport isisJdoSupport;
}
```

It can occasionally be the case that Apache Isis' internal adapter for the domain object is still in memory. JDO/DataNucleus seems to bump up the version of the object prior to its deletion, which under normal circumstances would cause Apache Isis to throw a concurrency exception. Therefore to prevent this from happening (ie to *force* the deletion of all instances), concurrency checking is temporarily disabled while this method is performed.

11.2.4. Reloading entities

An (intentional) limitation of JDO/DataNucleus is that persisting a child entity (in a 1:n bidirectional relationship) does not cause the parent's collection to be updated.

```
public interface IsisJdoSupport {
    @Programmatic
    <T> T refresh(T domainObject);
    @Programmatic
    void ensureLoaded(Collection<?> collectionOfDomainObjects);
    ...
}
```

The `refresh(T domainObject)` method can be used to reload the parent object (or indeed any object). Under the covers it uses the JDO `PersistenceManager#refresh(…)` API.

For example:

```
@DomainService(nature=NatureOfService.VIEW_CONTRIBUTIONS_ONLY)
public class OrderContributions {
    public Order newOrder(final Customer customer) {
        Order order = newTransientInstance(Order.class);
        order.setCustomer(customer);
        container.persist(customer);
        container.flush();                  ①
        isisJdoSupport.refresh(customer);   ②
        return order;
    }
    @Inject
    DomainObjectContainer container;
    @Inject
    IsisJdoSupport isisJdoSupport;
}
```

① flush to database, ensuring that the database row corresponding to the `Order` exists in its `order` table.

② reload the parent (customer) from the database, so that its collection of `Order`s is accurate.

The particular example that led to this method being added was a 1:m bidirectional relationship, analogous to `Customer 1←→* Order`. Persisting the child `Order` object did not cause the parent `Customer`'s collection of orders to be updated. In fact, JDO does not make any such guarantee to do so. Options are therefore either to maintain the collection in code, or to refresh the parent.

The `ensureLoaded(…)` method allows a collection of domain objects to be loaded from the database in a single hit. This can be valuable as a performance optimization to avoid multiple roundtrips to the database. Under the covers it uses the `PersistenceManager#retrieveAll(…)` API.

11.2.5. JDO `PersistenceManager`

The functionality provided by `IsisJdoSupport` focus only on the most common use cases. If you require more flexibility than this, eg for dynamically constructed queries, then you can use the

service to access the underlying JDO `PersistenceManager` API:

```
public interface IsisJdoSupport {
    @Programmatic
    PersistenceManager getJdoPersistenceManager();
    ...
}
```

For example:

```
public List<Order> findOrders(...) {
    javax.jdo.PersistenceManager pm = isisJdoSupport.getPersistenceManager();

    // knock yourself out...

    return someListOfOrders;
}
```

11.3. `MetricsService`

The `MetricsService` is a request-scoped domain service that hooks into the JDO/DataNucleus ObjectStore to provide a number of counters relating to numbers of object loaded, dirtied etc.

The service is used by the `InteractionContext` domain service (to populate the DTO held by the `Interaction.Execution`) and also by the (internal) `PublishingServiceInternal` domain service (to populate the `PublishedObjects` class.

11.3.1. API & Implementation

The API of the service is:

```
@RequestScoped
public interface MetricsService {
    int numberObjectsLoaded();              ①
    int numberObjectsDirtied();             ②
    int numberObjectPropertiesModified();   ③
}
```

① The number of objects that have, so far in this request, been loaded from the database. Corresponds to the number of times that `javax.jdo.listener.LoadLifecycleListener#postLoad(InstanceLifecycleEvent)` is fired.

② The number of objects that have, so far in this request, been dirtied/will need updating in the database); a good measure of the footprint of the interaction. Corresponds to the number of times that `javax.jdo.listener.DirtyLifecycleListener#preDirty(InstanceLifecycleEvent)` callback is fired.

③

The number of individual properties of objects that were modified; a good measure of the amount of work being done in the interaction. Corresponds to the number of times that the `AuditingService`'s (or `AuditerService`'s) `audit(…)` method will be called as the transaction completes.

The framework provides a default implementation of this API, namely `o.a.i.c.r.s.metrics.MetricsServiceDefault`.

11.3.2. Related Services

The `PublisherService` also captures the metrics gathered by the `MetricsService` and publishes them as part of the `PublishedObjects` class (part of its SPI).

11.4. `QueryResultsCache`

The purpose of the `QueryResultsCache` is to improve response times to the user, by providing a request-scoped cache of the value of some (safe or idempotent) method call. This will typically be as the result of running a query, but could be any expensive operation.

Caching such values is useful for code that loops "naively" through a bunch of stuff, performing an expensive operation each time. If the data is such that the same expensive operation is made many times, then the query cache is a perfect fit.

This service was inspired by similar functionality that exists in relational databases, for example Sybase's subquery results cache and Oracle's result_cache hint.

11.4.1. API & Implementation

The API defined by `QueryResultsCache` is:

```
@RequestScoped
public class QueryResultsCache {
    public static class Key {
        public Key(Class<?> callingClass, String methodName, Object... keys) {...}
        public Class<?> getCallingClass() { ... }
        public String getMethodName() { ... }
        public Object[] getKeys() { ... }
    }
    public static class Value<T> {
        public Value(T result) { ... }
        private T result;
        public T getResult() {
            return result;
        }
    }
    @Programmatic
    public <T> T execute(
        final Callable<T> callable,
        final Class<?> callingClass, final String methodName, final Object... keys) {
... }
    @Programmatic
    public <T> T execute(final Callable<T> callable, final Key cacheKey) { ... }
    @Programmatic
    public <T> Value<T> get(
        final Class<?> callingClass, final String methodName, final Object... keys) {
... }
    @Programmatic
    public <T> Value<T> get(final Key cacheKey) { ... }
    @Programmatic
    public <T> void put(final Key cacheKey, final T result) { ... }
}
```

This class (`o.a.i.applib.services.queryresultscache.QueryResultsCache`) is also the implementation.

11.4.2. Usage

Suppose that there's a `TaxService` that calculates tax on `Taxable` items, with respect to some `TaxType`, and for a given `LocalDate`. To calculate tax it must run a database query and then perform some additional calculations.

Our original implementation is:

```
@DomainService
public class TaxService {
    public BigDecimal calculateTax(
            final Taxable t, final TaxType tt, final LocalDate d) {
        // query against DB using t, tt, d
        // further expensive calculations
    }
}
```

Suppose now that this service is called in a loop, for example iterating over a bunch of orders, where several of those orders are for the same taxable products, say. In this case the result of the calculation would always be the same for any given product.

We can therefore refactor the method to use the query cache as follows:

```
public class TaxService {
    public BigDecimal calculateTax(
            final Taxable t, final TaxType tt, final LocalDate d) {
        return queryResultsCache.execute(
            new Callable<BigDecimal>(){                          ①
                public BigDecimal call() throws Exception {
                    // query against DB using t, tt, d
                    // further expensive calculations
                }
            },
            TaxService.class,                                    ②
            "calculateTax",
            t, tt, d);
        }
}
```

① the `Callable` is the original code

② the remaining parameters in essence uniquely identify the method call.

This refactoring will be worthwhile provided that enough of the orders being processed reference the same taxable products. If however every order is for a different product, then no benefit will be gained from the refactoring.

11.4.3. Related Services

The `Scratchpad` service is also intended for actions that are called many times, allowing arbitrary information to be shared between them. Those methods could be called from some outer loop in domain code, or by the framework itself if the action invoked has the `@Action#invokeOn()` annotation attribute set to `OBJECT_AND_COLLECTION` or `COLLECTION_ONLY`.

11.5. RepositoryService

The `RepositoryService` collects together methods for creating, persisting and searching for entities from the underlying persistence store. It acts as an abstraction over the JDO/DataNucleus objectstore.

You can use it during prototyping to write naive queries (find all rows, then filter using the Guava `Predicate` API, or you can use it to call JDO named queries using JDOQL.

As an alternative, you could also use JDO typesafe queries through the `IsisJdoSupport` service.

The methods in this service replace similar methods (now deprecated) in `DomainObjectContainer`.

11.5.1. API

The API of `RepositoryService` is:

```
public interface RepositoryService {
    <T> T instantiate(final Class<T> ofType);
①

    boolean isPersistent(Object domainObject);
②
    void persist(Object domainObject);
③
    void persistAndFlush(Object domainObject);
④

    void remove(Object persistentDomainObject);
⑤
    void removeAndFlush(Object persistentDomainObject);
⑥

    <T> List<T> allInstances(Class<T> ofType, long... range);
⑦

    <T> List<T> allMatches(Query<T> query);
⑧
    <T> List<T> allMatches(Class<T> ofType, Predicate<? super T> predicate, long...
range); ⑨

    <T> T uniqueMatch(Query<T> query);
⑩
    <T> T uniqueMatch(final Class<T> ofType, final Predicate<T> predicate);
⑪

    @Deprecated
    <T> T firstMatch(Query<T> query);
⑫
    @Deprecated
    <T> T firstMatch(final Class<T> ofType, final Predicate<T> predicate);
⑬
}
```

① create a new non-persisted domain entity. This is identical to `FactoryService`'s `instantiate(…)` method, but is provided in the `RepositoryService`'s API too because instantiating and persisting objects are often done together.

② test whether a particular domain object is persistent or not

③ persist (ie save) an object to the persistent object store (or do nothing if it is already persistent).

④ persist (ie save) and flush; same as `persist()`, but also flushes changes to database and updates managed properties and collections (i.e., 1-1, 1-n, m-n relationships automatically maintained by the DataNucleus persistence mechanism).

⑤ remove (ie delete) an object from the persistent object store (or do nothing if it has already been deleted).

⑥ remove (delete) and flush; same as `remove()`, but also flushes changes to database and updates managed properties and collections (i.e., 1-1, 1-n, m-n relationships automatically maintained by the DataNucleus persistence mechanism).

⑦ return all persisted instances of specified type. Mostly for prototyping, though can be useful to obtain all instances of domain entities if the number is known to be small. The optional varargs parameters are for paging control; more on this below.

⑧ all persistence instances matching the specified `Query`. `Query` itself is an Isis abstraction on top of JDO/DataNucleus' Query API. **This is the primary API used for querying**

⑨ As the previous, but with client-side filtering using a `Predicate`. Only really intended for prototyping.

⑩ Returns the first instance that matches the supplied query. If no instance is found then `null `will be returned, while if there is more that one instances a run-time exception will be thrown. Generally this method is preferred for looking up an object by its (primary or alternate) key.

⑪ As the previous, but with client-side filtering using a `Predicate`. Only really intended for prototyping.

⑫ (Deprecated) returns the first instance that matches the supplied query. If no instance is found then `null `will be returned. No exception is thrown if more than one matches, so this is less strict that `uniqueMatch(…)`.

⑬ (Deprecated) As the previous, but with client-side filtering using a `Predicate`. Only really intended for prototyping.

The `uniqueMatch(…)` methods are the recommended way of querying for (precisely) one instance. The `firstMatch(…)` methods are for less strict querying.

11.5.2. Usage

This section briefly discusses how application code can use (some of) these APIs.

Persist

```
Customer cust = repositoryService.instantiate(Customer.class);
cust.setFirstName("Freddie");
cust.setLastName("Mercury");
repositoryService.persist(cust);
```

You should be aware that by default Apache Isis queues up calls to `#persist()` and `#remove()`. These are then executed either when the request completes (and the transaction commits), or if the queue is flushed. This can be done either implicitly by the framework, or as the result of a direct call to `#flush()`.

By default the framework itself will cause `#flush()` to be called whenever a query is executed by way of `#allMatches(Query)`, as documented above. However, this behaviour can be disabled using the configuration property `isis.services.container.disableAutoFlush`.

`persistAndFlush(…)`, `removeAndFlush(…)`

In some cases, such as when using managed properties and collections for implementing 1-1, 1-n, or m-n relationships, the developer needs to invoke `flush()` to send the changes to the DataNucleus persistence mechanism. These managed properties and collections and then updated.

The `persistAndFlush(…)` and `removeAndFlush(…)` methods save the developer from having to additionally call the `flush(…)` method after calling `persist()` or `remove()`.

For example, the following code requires a flush to occur, so uses these methods:

```
public abstract class Warehouse extends SalesVIPEntity<Marketplace> {

    @Persistent(mappedBy = "marketplace", dependentElement = "true")
    @Getter @Setter                                                          ①
    private SortedSet<MarketplaceExcludedProduct> excludedProducts =
                            new TreeSet<MarketplaceExcludedProduct>();

    @Action(semantics = SemanticsOf.IDEMPOTENT)
    public MarketplaceExcludedProduct addExcludedProduct(final Product product) {
        MarketplaceExcludedProduct marketplaceExcludedProduct = this
.findExcludedProduct(product);
        if (marketplaceExcludedProduct == null) {
            marketplaceExcludedProduct =
                this.factoryService.instantiate(MarketplaceExcludedProduct.class);
        }

        this.wrap(marketplaceExcludedProduct).setMarketplace(this);
        this.wrap(marketplaceExcludedProduct).setProduct(product);

        this.repositoryService.persistAndFlush(marketplaceExcludedProduct);     ②
        return marketplaceExcludedProduct;
    }

    @Action(semantics = SemanticsOf.IDEMPOTENT)
    public void deleteFromExcludedProducts(final Product product) {
        final MarketplaceExcludedProduct marketplaceExcludedProduct =
findExcludedProduct(product);
        if (marketplaceExcludedProduct != null) {
            this.repositoryService.removeAndFlush(marketplaceExcludedProduct);
        }
    }
    ...                                                                      ③
}
```

① using lombok for brevity

② Needed for updating the managed properties and collections.

③ injected services and other methods ommited

On the “addExcludedProduct()” action, if the user didn’t flush, the following test would fail because

the managed collection would not containing the given product:

```
@Test
public void addExcludedProduct() {

    // given
    final AmazonMarketplace amazonMarketplace = this.wrapSkipRules(
        this.marketplaceRepository).findOrCreateAmazonMarketplace(
            AmazonMarketplaceLocation.FRANCE);

    final Product product = this.wrap(this.productRepository)
        .createProduct(UUID.randomUUID().toString(), UUID.randomUUID().toString());

    // when
    this.wrap(amazonMarketplace).addExcludedProduct(product);

    // then
    Assertions.assertThat(
            this.wrapSkipRules(amazonMarketplace).findAllProductsExcluded()
        ).contains(product);                                                    ①
}
```

① this would fail.

Query and `xxxMatches(…)`

There are various implementations of the `Query` API, but these either duplicate functionality of the other overloads of `allMatches(…)` or they are not supported by the JDO/DataNucleus object store. The only significant implementation of `Query` to be aware of is `QueryDefault`, which identifies a named query and a set of parameter/argument tuples.

For example, in the (non-ASF) Isis addons' todoapp the `ToDoItem` is annotated:

```
@javax.jdo.annotations.Queries( {
    @javax.jdo.annotations.Query(
            name = "findByAtPathAndComplete", language = "JDOQL",               ①
            value = "SELECT "
                    + "FROM todoapp.dom.module.todoitem.ToDoItem "
                    + "WHERE atPath.indexOf(:atPath) == 0 "                     ②
                    + "   && complete == :complete"),                           ③
    ...
})
public class ToDoItem ... {
    ...
}
```

① name of the query

② defines the `atPath` parameter

③ defines the `complete` parameter

This JDO query definitions are used in the `ToDoItemRepositoryImplUsingJdoql` service:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class ToDoItemRepositoryImplUsingJdoql implements ToDoItemRepositoryImpl {
    @Programmatic
    public List<ToDoItem> findByAtPathAndCategory(final String atPath, final Category
category) {
        return container.allMatches(
                new QueryDefault<>(ToDoItem.class,
                        "findByAtPathAndCategory",                    ①
                        "atPath", atPath,                             ②
                        "category", category));                       ③
    }
    ...
    @javax.inject.Inject
    DomainObjectContainer container;
}
```

① corresponds to the "findByAtPathAndCategory" JDO named query

② provide argument for the `atPath` parameter. The pattern is parameter, argument, parameter, argument, ... and so on.

③ provide argument for the `category` parameter. The pattern is parameter, argument, parameter, argument, ... and so on.

Other JDOQL named queries (not shown) follow the exact same pattern.

With respect to the other query APIs, the varargs parameters are optional, but allow for (client-side and managed) paging. The first parameter is the `start` (0-based, the second is the `count`.

It is also possible to query using DataNucleus' type-safe query API. For more details, see `IsisJdoSupport`.

11.5.3. Implementation

The default implementation of this domain service is `o.a.i.core.metamodel.services.repository.RepositoryServiceDefault`.

Configuration Properties

The default implementation of this domain service supports the following configuration properties:

Property	Value (default value)	Description
isis.services.container.disableAutoFlush	true,false (false)	Whether the RepositoryService (or DomainObjectContainer that delegates to it) should automatically flush pending changes prior to querying (via allMatches(), firstMatch() and so on).

11.5.4. Usage Notes

The FactoryService is often used in conjunction with the RepositoryService, to instantiate domain objects before persisting.

Chapter 12. Persistence Layer SPI

The persistence layer SPIs influence how the framework persists domain objects, for example controlling how to create an audit log of changes to domain objects.

The table below summarizes the persistence layer SPIs defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 14. Persistence Layer SPI

SPI	Description	Implementation	Notes
o.a.i.applib.services.audit AuditerService	Create an audit record for every changed property of every changed object within a transaction.	AuditerServiceLogging also AuditerServiceUsingJdo o.ia.m.audit isis-module-audit	
o.a.i.applib.services.audit AuditingService3	(deprecated, replaced by AuditerService); creates an audit record for every changed property of every changed object within a transaction.		
o.a.i.applib.services.publish EventSerializer	(deprecated, not used by replacement PublisherService) Creates a representation of either an action invocation or a changed object being published through the PublishingService.	RestfulObjects-SpecEventSerializer o.ia.m.publishing isis-module-publishing	
o.a.i.applib.services.publish PublisherService	Publish any action invocations/property edits and changed objects, typically for interchange with an external system in a different bounded context.	PublisherServiceLogging also PublisherService-UsingActiveMq o.ia.m.publishmq isis-module-publishmq	

SPI	Description	Implementation	Notes
o.a.i.applib.services.publish PublishingService	(deprecated, replaced by PublisherService) Publish any action invocations and changed objects, typically for interchange with an external system in a different bounded context.	PublishingService o.ia.m.publishing isis-module-publishing	
o.a.i.applib.services.userreg UserRegistrationService	Create a new user account with the configured security mechanism.	SecurityModule-AppUserRegistrationService o.ia.m.security isis-module-security	depends (implicitly) on: a configured EmailService

Key:

- o.a.i is an abbreviation for org.apache.isis
- o.ia.m is an abbreviation for org.isisaddons.module
- o.a.i.c.m.s is an abbreviation for org.apache.isis.core.metamodel.services
- o.a.i.c.r.s is an abbreviation for org.apache.isis.core.runtime.services

Where an implementation is available (on the classpath) then it is always registered automatically (that is, they are all (with one exception) annotated with @DomainService.

12.1. AuditerService

The AuditerService auditing service provides a simple mechanism to capture changes to data. It is called for each property that has changed on any domain object, as a set of pre- and post-values.

This service is intended to replace the now-deprecated AuditingService3. The difference between the two is that this service recognises that the transactionId is actually a request/interaction Id, and that an additional `sequence

12.1.1. SPI

The SPI for the service is:

```
public interface AuditerService {
    boolean isEnabled();                                    ①
    public void audit(
            UUID transactionId, int sequence,               ②
            String targetClassName,
            Bookmark target,                                ③
            String memberIdentifier,
            String propertyName,                            ④
            String preValue, String postValue,              ⑤
            String user, java.sql.Timestamp timestamp);     ⑥
 }
```

① whether this implementation is enabled. If all configured implementations are disabled, then auditing is suppressed (a minor performance optimization).

② together the `transactionId` (misnamed; really is the request/interaction Id) and the `sequence` uniquely identify the transaction in which the object was changed.

③ identifies the object that has changed

④ the property of the object that has changed. The combination of the `transactionId`, `sequence`, `target` and `propertyName` is unique.

⑤ the before and after values of the property (in string format). If the object was created then "[NEW]" is used as the pre-value; if the object was deleted then "[DELETED]" is used as the post-value.

⑥ the user that changed the object, and the date/time that this occurred.

The framework will call this for each and every domain object property that is modified within a transaction.

12.1.2. Implementations

The framework allows multiple implementations of this service to be registered; all will be called. The framework provides one implementation of its own, `AuditerServiceLogging` (in `o.a.i.applib.services.audit` package); this logs simple messages to an SLF4J logger.

For example, this can be configured to write to a separate log file by adding the following to `logging.properties`:

```
log4j.appender.AuditerServiceLogging=org.apache.log4j.FileAppender
log4j.appender.AuditerServiceLogging.File=./logs/AuditerServiceLogging.log
log4j.appender.AuditerServiceLogging.Append=false
log4j.appender.AuditerServiceLogging.layout=org.apache.log4j.PatternLayout
log4j.appender.AuditerServiceLogging.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss.SSS} %m%n

log4j.logger.org.apache.isis.applib.services.audit.AuditerServiceLogging=DEBUG,Auditer
ServiceLogging
log4j.additivity.org.apache.isis.applib.services.audit.AuditerServiceLogging=false
```

The (non-ASF) Incode Platform's audit module also provides an implementation, `org.isisaddons.module.audit.dom.AuditerServiceUsingJdo`. This creates an audit record for each changed property (ie every time that `AuditerService#audit(…)` is called.

The module also provides:

- `AuditingServiceMenu` service which provides actions to search for `AuditEntry`s, underneath an 'Activity' menu on the secondary menu bar.
- `AuditingServiceRepository` service to to search for persisted `AuditEntry``s. None of its actions are visible in the user interface (they are all `@Programmatic`).
- `AuditingServiceContributions` which contrbutes collections to the `HasTransactionId` interface. This will therefore display all audit entries that occurred in a given request/transaction, in other words whenever a command, a published event or another audit entry is displayed.

12.1.3. Usage

The typical way to indicate that an object should be audited is to annotate it with the `@DomainObject#auditing()` annotation.

12.1.4. Alternative Implementations

The (non-ASF) Incode Platform's audit module provides an implementation of this service (`AuditerService`), and also provides a number of related domain services (`AuditingServiceMenu`, `AuditingServiceRepository` and `AuditingServiceContributions`).

If menu items or contributions are not required in the UI, these can be suppressed either using security or by implementing a vetoing subscriber.

12.1.5. Related Services

The auditing service works very well with implementations of <code>PublisherService</code> that persist the <code>Interaction.Execution</code> objects obtained from the <code>InteractionContext</code> service. The interaction execution captures the _cause of an interaction (an action was invoked, a property was edited), while the <code>AuditerService</code> audit entries capture the effect of that interaction in terms of changed state.

The <code>CommandService</code> can also be combined with the auditer service, however <code>Command</code>s are primarily concerned with capture the _intent of an action, not the actual action invocation itself.

The `AuditerService` is intended to replace the (now-deprecated) `AuditingService3`, as the latter does not support the concept of multiple transactions within a single interaction.

12.2. `AuditingService3` (deprecated)

The `AuditingService3` auditing service (and its various supertypes)provides a simple mechanism to capture changes to data. It is called for each property that has changed on any domain object, as a set of pre- and post-values.

This service is deprecated, replaced by `AuditerService`.

12.2.1. SPI

The SPI for the service is:

```
public interface AuditingService3 {

    @Programmatic
    public void audit(
            final UUID transactionId, String targetClassName, final Bookmark target,
            String memberIdentifier, final String propertyName,
            final String preValue, final String postValue,
            final String user, final java.sql.Timestamp timestamp);
  }
```

The framework will call this for each and every domain object property that is modified within a transaction.

12.2.2. Implementation

The most full-featured available implementation is the (non-ASF) Incode Platform's audit module. This creates an audit records for each changed property (ie every time that `AuditingService3#audit(…)` is called. The implementation is `org.isisaddons.module.audit.dom.AuditingService`.

The module also provides:

- `AuditingServiceMenu` service which provides actions to search for `AuditEntry`s, underneath an 'Activity' menu on the secondary menu bar.
- `AuditingServiceRepository` service to to search for persisted `AuditEntry``s. None of its actions are visible in the user interface (they are all `@Programmatic`).
- `AuditingServiceContributions` which contrbutes collections to the `HasTransactionId` interface. This will therefore display all audit entries that occurred in a given transaction, in other words whenever a command, a published event or another audit entry is displayed.

If you just want to debug (writing to stderr), you can instead configure `o.a.i.applib.services.audit.AuditingService3$Stderr`

12.2.3. Usage

The typical way to indicate that an object should be audited is to annotate it with the `@DomainObject#auditing()` annotation.

12.2.4. Alternative Implementations

The (non-ASF) Incode Platform's audit module provides an implementation of this service (`AuditingService`), and also provides a number of related domain services (`AuditingServiceMenu`, `AuditingServiceRepository` and `AuditingServiceContributions`).

If menu items or contributions are not required in the UI, these can be suppressed either using security or by implementing a vetoing subscriber.

12.2.5. Related Services

This service has been deprecated and replaced by the equivalent `AuditerService`.

12.3. `EventSerializer` (deprecated)

The `EmailSerializer` service is a supporting service intended for use by (any implementation of) `PublishingService`. Its responsibility is to combine the `EventMetadata` and the `EventPayload` into some serialized form (such as JSON, XML or a string) that can then be published.

This service is deprecated, replaced with `PublisherService`.

See `PublishingService` for further discussion.

12.3.1. SPI

The SPI defined by this service is:

```
@Deprecated
public interface EventSerializer {
    Object serialize(                   ①
            EventMetadata metadata,     ②
            EventPayload payload);      ③
}
```

① returns an object for maximum flexibility, which is then handed off to the `PublishingService`.

② standard metadata about the event, such as the user, the `transactionId`, date/time etc

③ for published actions, will generally be an `EventPayloadForActionInvocation` (or subclass thereof); for published objects, will generally be an `EventPayloadForObjectChanged` (or subclass thereof)

It's important to make sure that the publishing service implementation is able to handle the serialized form. Strings are a good lowest common denominator, but in some cases a type-safe

equivalent, such as a w3c DOM `Document` or JSON node might be passed instead.

12.3.2. Implementation

There is no default implementation of this service provided by the core Apache Isis framework.

The (obsolete) Isis addons' publishing module provides an implementation (`org.isisaddons.module.publishing.dom.eventserializer.RestfulObjectsSpecEventSerializer`) that represents the event payload using the representation defined by the Restful Objects spec of (transient) objects, grafting on the metadata as additional JSON nodes.

For example, this is the JSON generated on an action invocation:

```
17:10:44,787  [FacetedMethodsBuilder 1209462641@qtp-130672250-0 INFO ]  introspecting org.apache.
{
  "metadata" : {
    "guid" : "001edb96-0d93-4879-8b1d-bec7ad4dde9f",
    "user" : "sven",
    "timestamp" : 1360343444475
  },
  "payload" : {
    "title" : "ACTION:dom.todo.ToDoItem#completed()\n    target=TODO:L_0^2:sven:1360343444475\n
    "members" : [ {
      "id" : "actionName",
      "memberType" : "property",
      "value" : "dom.todo.ToDoItem#completed()",
      "disabledReason" : "Always disabled"
    }, {
      "id" : "result",
      "memberType" : "property",
      "value" : {
        "rel" : "urn:org.restfulobjects:rels/object",
        "href" : "http://localhost:8080/restful/objects/TODO:L_0^2:sven:1360343444475",
        "method" : "GET",
        "type" : "application/json;profile=\"urn:org.restfulobjects:repr-types/domainobject\"",
        "title" : "Buy milk  - Completed!"
      },
      "disabledReason" : "Always disabled"
    }, {
      "id" : "target",
      "memberType" : "property",
      "value" : {
        "rel" : "urn:org.restfulobjects:rels/object",
        "href" : "http://localhost:8080/restful/objects/TODO:L_0^2:sven:1360343444475",
        "method" : "GET",
        "type" : "application/json;profile=\"urn:org.restfulobjects:repr-types/domainobject\"",
        "title" : "Buy milk  - Completed!"
      },
      "disabledReason" : "Always disabled"
    } ],
    "extensions" : {
      "isService" : false,
      "isPersistent" : false
    }
  }
}
17:11:10.041  [salonly              1209462641@qtp-130672250-0 DEBUG]  org.datanucleus.store.rdb
```

Figure 1. JSON representation of a published action invocation

while this is the object change JSON:

```
17:10:44,557  [FacetedMethodsBuilder 1209462641@qtp-130672250-0 INFO ]  introspecting org.apache.
{
  "metadata" : {
    "guid" : "001edb96-0d93-4879-8b1d-bec7ad4dde9f",
    "user" : "sven",
    "timestamp" : 1360343444475
  },
  "payload" : {
    "title" : "CHANGED_OBJECT:dom.todo.ToDoItem@6f5dab79",
    "members" : [ {
      "id" : "changed",
      "memberType" : "property",
      "value" : {
        "rel" : "urn:org.restfulobjects:rels/object",
        "href" : "http://localhost:8080/restful/objects/TODO:L_0^2:sven:1360343444475",
        "method" : "GET",
        "type" : "application/json;profile=\"urn:org.restfulobjects:repr-types/domainobject\"",
        "title" : "Buy milk  - Completed!"
      },
      "disabledReason" : "Always disabled"
    }, {
      "id" : "description",
      "memberType" : "property",
      "value" : "Buy milk",
      "disabledReason" : "Always disabled"
    } ],
    "extensions" : {
      "isService" : false,
      "isPersistent" : false
    }
  }
}
17:10:44,787  [FacetedMethodsBuilder 1209462641@qtp-130672250-0 INFO ]  introspecting org.apache.
```

Figure 2. JSON representation of a published changed object

You could if you wish change the representation by registering your own implementation of this API in `isis.properties`:

12.3.3. Related Services

This service is intended (though not mandated) to be used by implementations of `PublishingService`. The (non-ASF) Isis addons' publishing module does use it (though the (non-ASF) Incode Platform publishmq module does not).

12.4. `PublisherService`

The `PublisherService` API is intended for coarse-grained publish/subscribe for system-to-system interactions, from Apache Isis to some other system. Events that can be published are action invocations/property edits, and changed objects. A typical use case is to publish onto a pub/sub bus such as ActiveMQ with Camel to keep other systems up to date.

An alternative use is for profiling: for each execution (action invocation/property edit) the framework captures metrics of the number of objects loaded or dirtied as the result of that execution. If the `WrapperFactory` is used to call other objects then the metrics are captured for each sub-execution. The framework provides a default implementation, `PublisherServiceLogging`, that will log these execution graphs (in XML form, per the "ixn" schema) to an SLF4J logger.

Only actions/properties/domain objects annotated for publishing (using `@Action#publishing()`, `@Property#publishing()` or `@DomainObject#publishing()`) are published.

12.4.1. SPI

The SPI defined by the service is:

```
public interface PublisherService {
    void publish(final Interaction.Execution<?, ?> execution);  ①
    void publish(final PublishedObjects publishedObjects);      ②
}
```

① to publish an individual action invocation or property edit, as captured within an `Interaction.Execution`.

② to publish a set of changed objects.

Each `Interaction.Execution` has an owning `Interaction`; this is the same object obtainable from `InteractionContext`. Implementations that publish member executions can use `Interaction.Execution#getDto()` method to return a DTO (as per the "ixn" schema) which can be converted into a serializable XML representation using the `InteractionDtoUtils` utility class. The XML can either serialize a single execution, or can be a "deep" serialization of an execution and all sub-executions.

The full API of `PublishedObjects` itself is:

```
public interface PublishedObjects extends HasTransactionId, HasUsername {
    UUID getTransactionId();                                    ①
    String getUsername();                                       ②
    Timestamp getCompletedAt();                                 ③
    ChangesDto getDto();                                        ④

    int getNumberLoaded();                                      ⑤
    int getNumberCreated();
    int getNumberUpdated();
    int getNumberDeleted();
    int getNumberPropertiesModified();
}
```

① inherited from `HasTransactionId`, correlates back to the unique identifier of the transaction in which these objects were changed.

② inherited from `HasUsername`, is the user that initiated the transaction causing these objects to change

③ the time that this set of objects was collated (just before the completion of the transaction completes)..

④ returns a DTO (as per the "chg" schema) which can be converted into a serializable XML representation can be obtained using the `ChangesDtoUtils` utility class.

⑤ metrics as to the number of objects loaded, created, updated or deleted and the number of object properties modified (in other words the "size" or "weight" of the transaction).

12.4.2. Implementations

The framework allows multiple implementations of this service to be registered; all will be called. The framework provides one implementation of its own, `PublisherServiceLogging` (in `o.a.i.applib.services.publish` package); this logs "deep" serializations to an SLF4J logger.

For example, this can be configured to write to a separate log file by adding the following to `logging.properties`:

```
log4j.appender.PublisherServiceLogging=org.apache.log4j.FileAppender
log4j.appender.PublisherServiceLogging.File=./logs/PublisherServiceLogging.log
log4j.appender.PublisherServiceLogging.Append=false
log4j.appender.PublisherServiceLogging.layout=org.apache.log4j.PatternLayout
log4j.appender.PublisherServiceLogging.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss.SSS} %m%n

log4j.logger.org.apache.isis.applib.services.publish.PublisherServiceLogging=DEBUG,Pub
lisherServiceLogging
log4j.additivity.org.apache.isis.applib.services.publish.PublisherServiceLogging=false
```

The (non-ASF) Incode Platform's publishmq module also provides an implementation (`o.ia.m.publishmq.dom.servicespi.PublishingServiceUsingActiveMq`). This implementation publishes each member execution as an event on an ActiveMQ message queue. It also persists each execution as a `PublishedEvent` entity, allowing the event to be republished if necessary. The implementation also provides the ability to log additional `StatusMessage` entities, correlated on the transactionId, useful for diagnosing and monitoring the activity of subscribers of said message queues.

12.4.3. Usage

To indicate that an action invocation should be published, annotate it with the `@Action#publishing()` annotation.

To indicate that an property edit should be published, annotate it with the `@Property#publishing()` annotation.

To indicate that a changed object should be published is to annotate it with the `@DomainObject#publishing()` annotation.

12.4.4. Alternative Implementations

The (non-ASF) Incode Platform;s publishmq module provides an implementation of this service.

The module also provide services that contribute to the UI. If contributions are not required in the UI, these can be suppressed either using security or by implementing a vetoing subscriber.

12.4.5. Related Services

This service supports two main use cases:

- coarse-grained publish/subscribe for system-to-system interactions, from Apache Isis to some other system.

The `PublishingService` also supports this use case, but is deprecated: the `PublisherService` is intended as a replacement for `PublishingService`.

- profiling of interactions/transactions, eg to diagnose response/throughput issues.

To support these use cases several other services are involved:

- the `InteractionContext` is used to obtain the `Interaction` from which the member executions are published.
- the (internal) `ChangedObjectsServiceInternal` domain service is used to obtain the set of objects modified throughout the transaction
- the (internal) `PublisherServiceInternal` domain service filters these down to those changed objects that are also published (as per `@DomainObject#publishing()`) and delegates to the `PublisherService`.
- the `MetricsService` is used to obtain the objects that are loaded throughout the transaction; this info is used in order to instantiate the `PublishedObjects` object passed through to the `PublisherService`.

The `EventBusService` differs from the `PublisherService` in that it is intended for fine-grained publish/subscribe for object-to-object interactions within an Apache Isis domain object model. The event propagation is strictly in-memory, and there are no restrictions on the object acting as the event; it need not be serializable, for example. That said, it is possible to obtain a serialization of the action invocation/property edit causing the current event to be raised using `InteractionContext` domain service.

12.5. `PublishingService` (deprecated)

The `PublishingService` API is intended for coarse-grained publish/subscribe for system-to-system interactions, from Apache Isis to some other system. Here the only events published are those that action invocations and of changed objects. A typical use case is to publish onto a pub/sub bus such as ActiveMQ with Camel to keep other systems up to date.

This service is deprecated, replaced with `PublisherService`.

12.5.1. SPI

The SPI defined by the service is:

```
@Deprecated
public interface PublishingService {
    public void publish(
        EventMetadata metadata,                                    ①
        EventPayload payload);                                     ②
    void setEventSerializer(EventSerializer eventSerializer);     ③
}
```

① standard metadata about the event, such as the user, the `transactionId`, date/time etc

② for published actions, an `EventPayloadForActionInvocation` (or subclass thereof); for published objects, an `EventPayloadForObjectChanged` (or subclass thereof)

③ injects in the `EventSerializer` service. This is deprecated because not every implementation is required to use an `EventSerializer` so its inclusion within the SPI of `PublishingService` was in retrospect a mistake.

Typically implementations will use the injected `EventSerializer` to convert the metadata and payload into a form to be published:

```
public interface EventSerializer {
    public Object serialize(EventMetadata metadata, EventPayload payload);
}
```

The serialized form returned by `EventSerializer` must be in a form that the `PublishingService` implementation is able to handle. Strings are a good lowest common denominator, but (if custom implementations of both `EventSerializer` and `PublishingService` were in use) then it might also be some other type, for example an `org.w3c.dom.Document` or an `org.json.JSONObject` might be returned instead.

12.5.2. Implementation

There is no default implementation of this service provided by the core Apache Isis framework.

The (obsolete) Isis addons' publishing module provides an implementation (`org.isisaddons.module.publishing.dom.PublishingService`) that persists each event as a `PublishedEvent` entity. This holds the serialized form of the event metadata and payload as translated into a string by the injected `EventSerializer`. The module also provides its own implementation of `EventSerializer`, namely `RestfulObjectsSpecEventSerializer`, which represents the event payload using the representation defined by the Restful Objects spec of (transient) objects, grafting on the metadata as additional JSON nodes.

The `PublishedEvent` entity also has a `state` field taking the values either "QUEUED" or "PROCESSED". The intention here is that an event bus can poll this table to grab pending events and dispatch them to downstream systems. When `PublishedEvent`s are persisted initially they always take the value "QUEUED".

The framework provides no default implementations of this service.

12.5.3. Usage

To indicate that an action invocation should be published, annotate it with the `@Action#publishing()` annotation.

To indicate that a changed object should be published is to annotate it with the `@DomainObject#publishing()` annotation.

It is also possible to "fine-tune" the `EventPayload` using the `#publishingFactory()` attribute (for both annotations). By default the `EventPayload` that is serialized identifies the object(s) being interacted with or changed, and in the case of the action invocation provides details of the action arguments and result (if any) of that action. However, the payload does not (by default) include any information about the new state of these objects. It is therefore the responsibility of the subscriber to call back to Apache Isis to determine any information that has not been published.

The replacement `PublisherService` does *not* support the concept of "payload factories" (but is otherwise more flexible).

Although the representations (if using the Restful Object serializer and Restful Objects viewer) does include hrefs for the objects, this nevertheless requires an additional network call to obtain this information).

In some circumstances, then, it may make more sense to eagerly "push" information about the change to the subscriber by including that state within the payload.

To accomplish this, an implementation of a “PayloadFactory” must be specified in the annotation.

For actions, we implement the `PublishingPayloadFactoryForAction` (in `o.a.i.applib.annotation`):

```
@Deprecated
public interface PublishingPayloadFactoryForAction {
    public EventPayload payloadFor(
            Identifier actionIdentifier,
            Object target,
            List<Object> arguments,
            Object result);
}
}
```

The `EventPayloadForActionInvocation` abstract class (in the Isis applib) should be used as the base class for the object instance returned from `payLoadFor(…)`.

For objects, the interface to implement is `PublishingPayloadFactoryForObject`:

```
@Deprecated
public interface PublishingPayloadFactoryForObject {
    public EventPayload payloadFor(
        Object changedObject,
        PublishingChangeKind publishingChangeKind);     ①
}
```

① an enum taking the values `CREATE`, `UPDATE`, `DELETE`

Similarly, the `EventPayloadForObjectChanged` abstract class should be used as the base class for the object returned from `payLoadFor(…)`.

For example, the following will eagerly include a `ToDoItem`'s `description` property whenever it is changed:

```
@DomainObject(publishingPayloadFactory=ToDoItemPayloadFactory.class)
public class ToDoItem {
    ...
}
```

where `ToDoItemPayloadFactory` is defined as:

```
public class ToDoItemChangedPayloadFactory implements
PublishingPayloadFactoryForObject {
    public static class ToDoItemPayload
        extends EventPayloadForObjectChanged<ToDoItem> {
      public ToDoItemPayload(ToDoItem changed) { super(changed); }
      public String getDescription() { return getChanged().getDescription(); }
    }
    @Override
    public EventPayload payloadFor(Object changedObject, PublishingChangeKind kind) {
        return new ToDoItemPayload((ToDoItem) changedObject);
    }
}
```

12.5.4. Related Services

The `PublishingService` is intended for coarse-grained publish/subscribe for system-to-system interactions, from Apache Isis to some other system. Here the only events published are those that action invocations (for actions annotated with `@Action#publishing()`) and of changed objects (for objects annotated with `@DomainObject#publishing()`.

The `PublisherService` is intended as a replacement for this service. The use case for `PublisherService` is the same: coarse-grained publishing of events for system-to-system interactions. It is in most respects more flexible though: events are published both for action invocations (annotated with `@Action#publishing()`) and also for property edits (annotated with `@Property#publishing()`. It also publishes changed objects (for objects annotated with

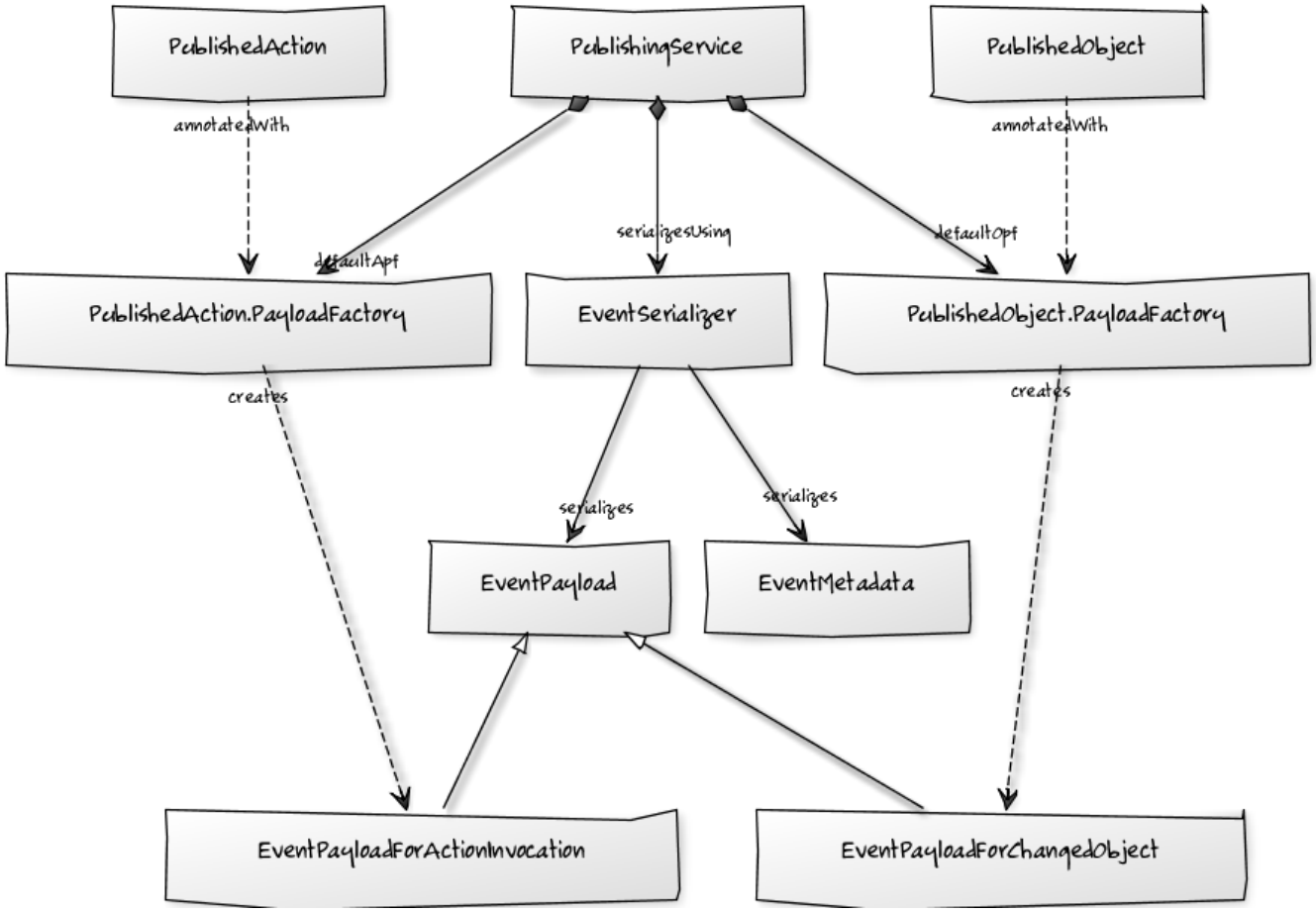
`@DomainObject#publishing()`). However, rather than publishing one event for every changed objects, it publishes a single event that identifies all objects created, updated or deleted.

Another significant difference between `PublishingService` and `PublisherService` is in the content of the events themselves. While the former uses the `MementoService` to create an ad-hoc serialization of the action being invoked, the latter uses the DTOs/XML schemas as a formal specification of the nature of the interaction (action invocation, property edit or changed objects).

The `EventBusService` meanwhile differs from both `PublishingService` and `PublisherService` in that it is intended for fine-grained publish/subscribe for object-to-object interactions within an Apache Isis domain object model. The event propagation is strictly in-memory, and there are no restrictions on the object acting as the event; it need not be serializable, for example. (That said, it is possible to obtain a serialization of the action invocation/property edit causing the current event to be raised using `InteractionContext` domain service).

12.5.5. Design Notes

The following class diagram shows how the above components fit together:



This yuml.me diagram was generated at yuml.me.

12.6. `UserRegistrationService`

The `UserRegistrationService` provides the ability for users to sign-up to access an application by providing a valid email address, and also provides the capability for users to reset their password if forgotten.

For user sign-up, the Wicket viewer will check whether an implementation of this service (and also the <code>EmailNotificationService</code>) is available, and if so will render a sign-up page where the user enters their email address. A verification email is sent (using the aforementioned <code>EmailNotificationService</code>) which includes a link back to the running application; this allows the user then to complete their registration process (choose user name, password and so on). When the user has provided the additional details, the Wicket viewer calls _this service in order to create an account for them, and then logs the user on.

For the password reset feature, the Wicket viewer will render a password reset page, and use the `EmailNotificationService` to send a "password forgotten" email. This service provides the ability to reset a password based on the user's email address.

It is of course possible for domain objects to use this service; it will be injected into domain object or other domain services in the usual way. That said, we expect that such use cases will be comparatively rare; the primary use case is for the Wicket viewer's sign-up page.

For further details on the user registration feature (as supported by the Wicket viewer), see here.

12.6.1. SPI

The SPI defined by the service is:

```
public interface UserRegistrationService {
    @Programmatic
    boolean usernameExists(String username);                                    ①
    @Programmatic
    boolean emailExists(String emailAddress);                                   ②
    @Programmatic
    void registerUser(String username, String password, String emailAddress);  ③
    @Programmatic
    boolean updatePasswordByEmail(String emailAddress, String password);       ④
}
```

① checks if there is already a user with the specified username

② checks if there is already a user with the specified email address

③ creates the user, with specified password and email address. The username and email address must both be unique (not being used by an existing user)

④ allows the user to reset their password

12.6.2. Implementation

The core Apache Isis framework itself defines only an API; there is no default implementation. Rather, the implementation will depend on the security mechanism being used.

That said, if you have configured your app to use the (non-ASF) Incode Platform's security module

then note that the security module does provide an abstract implementation (`SecurityModuleAppUserRegistrationServiceAbstract`) of the `UserRegistrationService`. You will need to extend that service and provide implementation for the two abstract methods: `getInitialRole()` and `getAdditionalInitialRoles()`.

For example:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class AppUserRegistrationService extends
SecurityModuleAppUserRegistrationServiceAbstract {
    protected ApplicationRole getInitialRole() {
    return findRole("regular-user");
    }
    protected Set<ApplicationRole> getAdditionalInitialRoles() {
        return Collections.singleton(findRole("self-registered-user"));
    }
    private ApplicationRole findRole(final String roleName) {
        return applicationRoles.findRoleByName(roleName);
    }
    @Inject
    private ApplicationRoles applicationRoles;
}
```

This is needed so that the self-registered users are assigned automatically to your application role(s) and be able to use the application. Without any role such user will be able only to see/use the logout link of the application.

12.6.3. Related Services

The most common use case is to allow users to sign-up through Apache Isis' Wicket viewer. Because the process requires email to be sent, the following services must be configured:

- `EmailService`
- `EmailNotificationService`
- `UserRegistrationService` (this service)

The `EmailService` in particular requires additional configuration properties to specify the external SMTP service.

Chapter 13. Bootstrapping SPI

Bootstrapping SPIs influence how the framework locates the components that make up the running application.

The table below summarizes the bootstrapping SPI defined by Apache Isis. It also lists their corresponding implementation, either a default implementation provided by Apache Isis itself, or provided by one of the (non-ASF) Incode Platform modules.

Table 15. Bootstrapping SPI

SPI	Description	Implementation	Notes
`o.a.i.applib.services.classdiscovery ClassDiscoveryService`	Mechanism to locate (from the classpath) classes with a specific annotation (eg `@DomainService`) Subtypes of a given type (eg `FixtureScript`).	`ClassDiscoveryService-UsingReflections o.a.i.core isis-core-applib`	requires `org.reflections:reflections` as Maven dependency

Key:

- `o.a.i` is an abbreviation for `org.apache.isis`
- `o.ia.m` is an abbreviation for `org.isisaddons.module`
- `o.a.i.c.m.s` is an abbreviation for `org.apache.isis.core.metamodel.services`
- `o.a.i.c.r.s` is an abbreviation for `org.apache.isis.core.runtime.services`

13.1. `ClassDiscoveryService2`

The `ClassDiscoveryService2` service (and its various supertypes) is used to automatically discover subclasses of any given type on the classpath. The primary use case is to support "convention-over-configuration" designs that work with a minimum of configuration.

This service is used by the `FixtureScripts` service to automatically locate any `FixtureScript` implementations.

13.1.1. SPI

The SPI defined by the service is:

```
public interface ClassDiscoveryService2  {
    @Programmatic
    <T> Set<Class<? extends T>> findSubTypesOfClasses(Class<T> type, String
packagePrefix);
    @Deprecated
    @Programmatic
    <T> Set<Class<? extends T>> findSubTypesOfClasses(Class<T> type);       ①
}
```

① no longer used

13.1.2. Implementation

Isis provides an implementation of this service, namely `o.a.i.applib.services.classdiscovery.ClassDiscoveryServiceUsingReflections`.

This implementation is also used to discover domain services annotated with `@DomainService`. Currently this logic uses the implementation directly, so is not pluggable.

13.1.3. Related Services

The `FixtureScripts` domain service uses `ClassDiscoveryService` to discover `FixtureScript`s implementations to present in the UI.

Note that the bootstrapping of the framework itself does *not* use this service (though it does use the same underlying library as the default implementation of this service, namely `org.reflections.Reflections`).