Fundamentals

Table of Contents

Chapter 1. Fundamentals

This guide introduces the core concepts and ideas behind Apache Isis, and tells you how to get started with a Maven archetype.

It also describes a number of how-tos, describes how to influence the UI layout of your domain objects (this is ultimately just a type of metadata), and it catalogues various FAQs.

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 (this guide)
- Wicket viewer
- Restful Objects viewer
- Security
- Testing
- Beyond the Basics

The reference guides are:

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

The remaining guides are:

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

Chapter 2. Core Concepts

This introductory chapter should give you a good about what Apache Isis actually **is**: the fundamental ideas and principles that it builds upon, how it compares with other frameworks, what the fundamental building blocks are for actually writing an Isis application, and what services and features the framework provides for you to leverage in your own apps.

> Parts of this chapter have been adapted from Dan Haywood's 2009 book, 'Domain Driven Design using Naked Objects'. We've also added some new insights and made sure the material we've used is relevant to Apache Isis.

2.1. Philosophy and Architecture

This section describes some of the core ideas and architectural patterns upon which Apache Isis builds.

2.1.1. Domain-Driven Design

There's no doubt that we developers love the challenge of understanding and deploying complex technologies. But understanding the nuances and subtleties of the business domain itself is just as great a challenge, perhaps more so. If we devoted our efforts to understanding and addressing those subtleties, we could build better, cleaner, and more maintainable software that did a better job for our stakeholders. And there's no doubt that our stakeholders would thank us for it.

A couple of years back Eric Evans wrote his book Domain-Driven Design, which is well on its way to becoming a seminal work. In fact, most if not all of the ideas in Evans' book have been expressed before, but what he did was pull those ideas together to show how predominantly object-oriented techniques can be used to develop rich, deep, insightful, and ultimately useful business applications.

There are two central ideas at the heart of domain-driven design.

- the ***ubiquitous language*** is about getting the whole team (both domain experts and developers) to communicate more transparently using a domain model.
- Meanwhile, ***model-driven design*** is about capturing that model in a very straightforward manner in code.

Let's look at each in turn.

Ubiquitous Language

It's no secret that the IT industry is plagued by project failures. Too often systems take longer than intended to implement, and when finally implemented, they don't address the real requirements anyway.

Over the years we in IT have tried various approaches to address this failing. Using waterfall methodologies, we've asked for requirements to be fully and precisely written down before starting on anything else. Or, using agile methodologies, we've realized that requirements are likely to change anyway and have sought to deliver systems incrementally using feedback loops to refine the implementation.

But let's not get distracted talking about methodologies. At the end of the day what really matters is communication between the domain experts (that is, the business) who need the system and the techies actually implementing it. If the two don't have and cannot evolve a shared understanding of what is required, then the chance of delivering a useful system will be next to nothing.

Bridging this gap is traditionally what business analysts are for; they act as interpreters between the domain experts and the developers. However, this still means there are two (or more) languages in use, making it difficult to verify that the system being built is correct. If the analyst mistranslates a requirement, then neither the domain expert nor the application developer will discover this until (at best) the application is first demonstrated or (much worse) an end user sounds the alarm once the application has been deployed into production.

Rather than trying to translate between a business language and a technical language, with *DDD* we aim to have the business and developers using the same terms for the same concepts in order to create a single ***domain model***. This domain model identifies the relevant concepts of the domain, how they relate, and ultimately where the responsibilities are. This single domain model provides the vocabulary for the ubiquitous language for our system.

Ubiquitous Language

Build a common language between the domain experts and developers by using the concepts of the domain model as the primary means of communication. Use the terms in speech, in diagrams, in writing, and when presenting.

If an idea cannot be expressed using this set of concepts, then go back and extend the model. Look for and remove ambiguities and inconsistencies.

Creating a ubiquitous language calls upon everyone involved in the system's development to express what they are doing through the vocabulary provided by the model. If this can't be done, then our model is incomplete. Finding the missing words deepens our understanding of the domain being modeled.

This might sound like nothing more than me insisting that the developers shouldn't use jargon when talking to the business. Well, that's true enough, but it's not a one-way street. A **ubiquitous language** demands that the developers work hard to understand the problem domain, but it also demands that the business works hard in being **precise** in its naming and descriptions of those concepts. After all, ultimately the developers will have to express those concepts in a computer programming language.

Also, although here I'm talking about the "domain experts" as being a homogeneous group of people, often they may come from different branches of the business. Even if we weren't building a computer system, there's a lot of value in helping the domain experts standardize their own terminology. Is the marketing department's "prospect" the same as sales' "customer," and is that the same as an after-sales "contract"?

The need for precision within the ubiquitous language also helps us scope the system. Most business processes evolve piecemeal and are often quite ill-defined. If the domain experts have a very good idea of what the business process should be, then that's a good candidate for automation, that is, including it in the scope of the system. But if the domain experts find it hard to agree, then it's probably best to leave it out. After all, human beings are rather more capable of dealing with fuzzy situations than computers.

So, if the development team (business and developers together) continually searches to build their ubiquitous language, then the domain model naturally becomes richer as the nuances of the domain are uncovered. At the same time, the knowledge of the business domain experts also deepens as edge conditions and contradictions that have previously been overlooked are explored.

We use the ubiquitous language to build up a domain model. But what do we do **with** that model? The answer to that is the second of our central ideas.

Model-Driven Design

Of the various methodologies that the IT industry has tried, many advocate the production of separate analysis models and implementation models. One example (from the mid 2000s) was that of the *OMG*'s Model-Driven Architecture (*MDA*) initiative, with its platform-independent model (the *PIM*) and a platform-specific model (the *PSM*).

Bah and humbug! If we use our ubiquitous language just to build up a high-level analysis model, then we will re-create the communication divide. The domain experts and business analysts will look only to the analysis model, and the developers will look only to the implementation model. Unless the mapping between the two is completely mechanical, inevitably the two will diverge.

What do we mean by **model** anyway? For some, the term will bring to mind UML class or sequence diagrams and the like. But this isn't a model; it's a visual **representation** of some aspect of a model. No, a domain model is a group of related concepts, identifying them, naming them, and defining how they relate. What is in the model depends on what our objective is. We're not looking to simply model everything that's out there in the real world. Instead, we want to take a relevant abstraction or simplification of it and then make it do something useful for us. A model is neither right nor wrong, just more or less useful.

For our ubiquitous language to have value, the domain model that encodes it must have a straightforward, literal representation to the design of the software, specifically to the implementation. Our software's design should be driven by this model; we should have a model-driven design.

Model-Driven Design

There must be a straightforward and very literal way to represent the domain model in terms of software. The model should balance these two requirements: form the ubiquitous language of the development team and be representable in code.

Changing the code means changing the model; refining the model requires a change to the code.

Here also the word **design** might mislead; some might be thinking of design documents and design diagrams, or perhaps of user interface (UX) design. But by **design** we mean a way of organizing the domain concepts, which in turn leads to the way in which we organize their representation in code.

Luckily, using ***object-oriented*** (*OO*) languages such as Java, this is relatively easy to do; *OO* is based on a modeling paradigm anyway. We can express domain concepts using classes and interfaces, and we can express the relationships between those concepts using associations.

So far so good. Or maybe, so far so much motherhood and apple pie. Understanding the *DDD* concepts isn't the same as being able to apply them, and some of the *DDD* ideas can be difficult to put into practice. Time to discuss the naked objects pattern and how it eases that path by applying these central ideas of *DDD* in a very concrete way.

2.1.2. Naked Objects Pattern

Apache Isis implements the naked objects pattern, originally formulated by Richard Pawson. So who better than Richard to explain the origination of the idea?

<div class="extended-quote-first"><p>The Naked Objects pattern arose, at least in part, from my own frustration at the lack of success of the domain-driven approach. Good examples were hard to find—as they are still. </p></div>

<div class="extended-quote"><p>A common complaint from DDD practitioners was that it was hard to gain enough commitment from business stakeholders, or even to engage them at all. My own experience suggested that it was nearly impossible to engage business managers with UML diagrams. It was much easier to engage them in rapid prototyping — where they could see and interact with the results — but most forms of rapid prototyping concentrate on the presentation layer, often at the expense of the underlying model and certainly at the expense of abstract thinking. </p></div>

<div class="extended-quote"><p>Even if you could engage the business sponsors sufficiently to design a domain model, by the time you’d finished developing the system on top of the domain model, most of its benefits had disappeared. It’s all very well creating an agile domain object model, but if any change to that model also dictates the modification of one or more layers underneath it (dealing with persistence) and multiple layers on top (dealing with presentation), then that agility is practically worthless. </p></div>

<div class="extended-quote"><p>The other concern that gave rise to the birth of Naked Objects was how to make user interfaces of mainstream business systems more "expressive" — how to make them feel more like using a drawing program or CAD system. Most business systems are not at all expressive; they treat the user merely as a dumb process-follower, rather than as an empowered problem-solver. Even the so-called usability experts had little to say on the subject: try finding the word "empowerment" or any synonym thereof in the index of any book on usability. Research had demonstrated that the best way to achieve expressiveness was to create an object-oriented user interface (OOUI). In practice, though, OOUIs were notoriously hard to develop. </p></div>

<div class="extended-quote"><p>Sometime in the late 1990s, it dawned on me that if the domain

model really did represent the "ubiquitous language" of the business and those domain objects were behaviorally rich (that is, business logic is encapsulated as methods on the domain objects rather than in procedural scripts on top of them), then the UI could be nothing more than a reflection of the user interface. This would solve both of my concerns. It would make it easier to do domain-driven design, because one could instantly translate evolving domain modeling ideas into a working prototype. And it would deliver an expressive, object-oriented user interface for free. Thus was born the idea of Naked Objects. </p></div>

<div class="extended-quote-attribution"><p>-- Richard Pawson </p></div>

You can learn much more about the pattern in the book, Naked Objects, also freely available to read online. Richard co-wrote the book with one of Apache Isis' committers, Robert Matthews, who was in turn the author of the Naked Objects Framework for Java (the original codebase of of Apache Isis).

You might also want to read Richard's PhD on the subject.

> One of the external examiners for Richard's PhD was Trygve Reenskaug, who originally formulated the MVC pattern at Xeroc PARC. In his paper, Baby UML, Reenskaug describes that when implemented the first MVC, "the conventional wisdom in the group was that objects should be visible and tangible, thus bridging the gap between the human brain and the abstract data within the computer." Sound familiar? It's interesting to speculate what might have been if this idea had been implemented back then in the late 70s.
>
> Reenskaug then goes on to say that "this simple and powerful idea failed because … users were used to seeing [objects] from different perspectives. The visible and tangible object would get very complex if it should be able to show itself and be manipulated in many different ways."
>
> In Apache Isis the responsibility of rendering an object is not the object itself, it is the framework. Rather, the object inspects the object and uses that to decide how to render the object. This is also extensible. In the Isis Addons (non-ASF) the Isis addons' gmap3 wicket extension renders any object with latitude/longitude on a map, while Isis addons' fullcalendar2 wicket extension renders any object with date(s) on a calendar.

Object Interface Mapping

Another — more technical — way to think about the naked objects pattern is as an *object interface mapper*, or `OIM`. We sometimes use this idea to explain naked objects to a bunch of developers.

Just as an ORM (such as DataNucleus or Hibernate) maps domain entities to a database, you can think of the naked objects pattern as representing the concept of mapping domain objects to a user interface.

This is the way that the MetaWidget team, in particular Richard Kennard, the primary contributor, likes to describe their tool. MetaWidget has a number of ideas in common with Apache Isis, specifically the runtime generation of a UI for domain objects. You can hear more from Kennard and others on this Javascript Jabber podcast.

> We compare Apache Isis' with MetaWidget here.

What this means in practice

This screencast shows what all of this means in practice, showing the relationship between a running app and the actual code underneath.

> This screencast shows Apache Isis v1.0.0, Jan 2013. The UI has been substantially refined since that release.

2.1.3. Hexagonal Architecture

One of the patterns that Evans discusses in his book is that of a **layered architecture**. In it he describes why the domain model lives in its own layer within the architecture. The other layers of the application (usually presentation, application, and persistence) have their own responsibilities, and are completely separate. Each layer is cohesive and depending only on the layers below. In particular, we have a layer dedicated to the domain model. The code in this layer is unencumbered with the (mostly technical) responsibilities of the other layers and so can evolve to tackle complex domains as well as simple ones.

This is a well-established pattern, almost a de-facto; there's very little debate that these responsibilities should be kept separate from each other. With Apache Isis the responsibility for presentation is a framework concern, the responsibility for the domain logic is implemented by the (your) application code.

A few years ago Alistair Cockburn reworked the traditional layered architecture diagram and came up with the ***hexagonal architecture***:.

> The hexagonal architecture is also known as the Ports and Adapters architecture or (less frequently) as the Onion architecture.

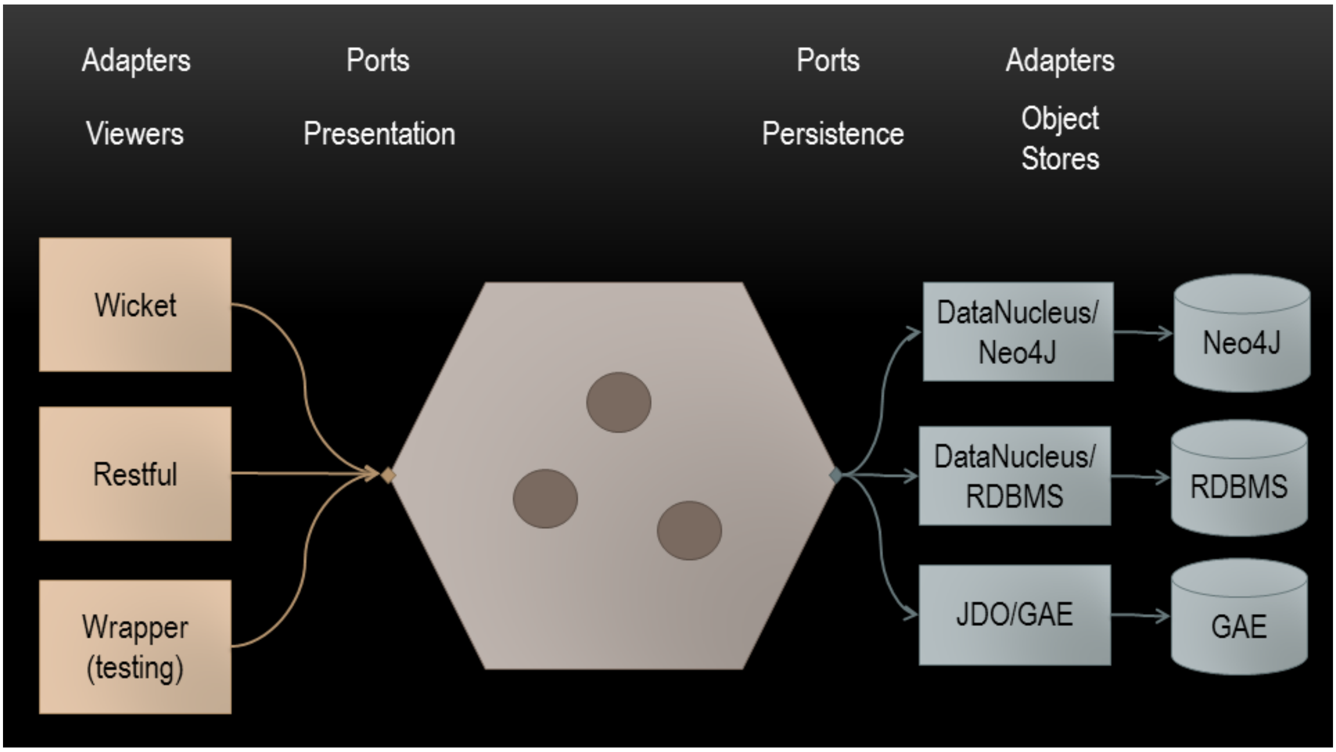


Figure 1. The hexagonal architecture emphasizes multiple implementations of the different layers.

What Cockburn is emphasizing is that there's usually more than one way **into** an application (what he called the ***user-side' ports***) and more than one way **out of** an application too (the ***data-side ports***). This is very similar to the concept of primary and secondary actors in use cases: a primary actor (often a human user but not always) is active and initiates an interaction, while a secondary actor (almost always an external system) is passive and waits to be interacted with.

Associated with each port can be an ***adapter*** (in fact, Cockburn's alternative name for this architecture is ***ports and adapters***). An adapter is a device (piece of software) that talks in the protocol (or *API*) of the port. Each port could have several adapters.

Apache Isis maps very nicely onto the **hexagonal architecture**. Apache Isis' viewers act as user-side adapters and use the Apache Isis metamodel API as a port into the domain objects. For the data side, we are mostly concerned with persisting domain objects to some sort of object store. Here Apache Isis delegates most of the heavy lifting to ORM implementing the JDO API. Most of the time this will be DataNucleus configured to persist to an RDBMS, but DataNucleus can also support other object stores, for example Neo4J. Alternatively Apache Isis can be configured to persist using some other JDO implementation, for example Google App Engine.

2.1.4. Aspect Oriented

Although not a book about object modelling, Evans' "Domain Driven Design" does use object orientation as its primary modelling tool; while naked objects pattern very much comes from an OO background (it even has 'object' in its name); Richard Pawson lists Alan Kay as a key influence.

It's certainly true that to develop an Apache Isis application you will need to have good object oriented modelling skills. But given that all the mainstream languages for developing business systems are object oriented (Java, C#, Ruby), that's not such a stretch.

However, what you'll also find as you write your applications is that in some ways an Isis application is more aspect-oriented than it is object oriented. Given that aspect-orientation — as a programming paradigm at least — hasn't caught on, that statement probably needs unpacking a little.

AOP Concepts

Aspect-orientation, then, is a different way of decomposing your application, by treating *cross-cutting concerns* as a first-class citizen. The canonical (also rather boring) example of a cross-cutting concern is that of logging (or tracing) all method calls. An aspect can be written that will weave in some code (a logging statement) at specified points in the code).

This idea sounds rather abstract, but what it really amounts to is the idea of interceptors. When one method calls another the AOP code is called in first. This is actually then one bit of AOP that is quite mainstream; DI containers such as Spring provide aspect orientation in supporting annotations such as `@Transactional` or `@Secured` to java beans.

Another aspect (ahem!) of aspect-oriented programming has found its way into other programming languages, that of a mix-in or trait. In languages such as Scala these mix-ins are specified statically as part of the inheritance hierarchy, whereas with AOP the binding of a trait to some other class/type is done without the class "knowing" that additional behaviour is being mixed-in to it.

Realization within Apache Isis

What has all this to do with Apache Isis, then?

Well, a different way to think of the naked objects pattern is that the visualization of a domain object within a UI is a cross-cutting concern. By following certain very standard programming conventions that represent the *Apache Isis Programming Model* (POJOs plus annotations), the framework is able to build a metamodel and from this can render your domain objects in a standard generic fashion. That's a rather more interesting cross-cutting concern than boring old logging!

Isis also draws heavily on the AOP concept of interceptors. Whenever an object is rendered in the UI, it is filtered with respect to the user's permissions. That is, if a user is not authorized to either view or perhaps modify an object, then this is applied transparently by the framework. The Isis addons' security module, mentioned previously, provides a rich user/role/permissions subdomain to use out of the box; but you can integrate with a different security mechanism if you have one already.

Another example of interceptors are the Isis addons' command and Isis addons' audit modules. The command module captures every user interaction that modifies the state of the system (the "cause" of a change) while the audit module captures every change to every object (the "effect" of a change). Again, this is all transparent to the user.

Apache Isis also has an internal event bus (you can switch between an underlying implementation of Gauva or Axon). A domain event is fired whenever an object is interacted with, and this allows any subscribers to influence the operation (or even veto it). This is a key mechanism in ensuring that Isis applications are maintainable, and we discuss it in depth in the section on Decoupling. But fundamentally its relying on this AOP concept of interceptors.

Finally, Isis also a feature that is akin to AOP mix-ins. A "contributed action" is one that is implemented on a domain service but that appears to be a behaviour of rendered domain object. In other words, we can dissociate behaviour from data. That's not always the right thing to do of course. In Richard Pawson's description of the naked objects pattern he talks about "behaviourally rich" objects, in other words where the business functionality encapsulated the data. But on the other hand sometimes the behaviour and data structures change at different rates. The single responsibility principle says we should only lump code together that changes at the same rate. Apache Isis' support for contributions (not only contributed actions, but also contributed properties and contributed collections) enables this. And again, to loop back to the topic of this section, it's an AOP concept that being implemented by the framework.

The nice thing about aspect orientation is that for the most part you can ignore these cross-cutting concerns and - at least initially at least - just focus on implementing your domain object. Later when your app starts to grow and you start to break it out into smaller modules, you can leverage Apache Isis' AOP support for (mixins), (contributions) and interceptors (the event bus) to ensure that your codebase remains maintainable.

2.1.5. How Apache Isis eases DDD

The case for *DDD* might be compelling, but that doesn't necessarily make it easy to do. Let's take a look at some of the challenges that *DDD* throws up and see how Apache Isis (and its implementation of the naked objects pattern) helps address them.

DDD takes a conscious effort

Here's what Eric Evans says about ubiquitous language:

<div class="extended-quote-first"><p>With a conscious effort by the [development] team the domain model can provide the backbone for [the] common [ubiquitous] language…connecting team communication to the software implementation." </p></div>

<div class="extended-quote-attribution"><p>-- Eric Evans </p></div>

The word to pick up on here is **conscious**. It takes a conscious effort by the entire team to develop the ubiquitous language. Everyone in the team must challenge the use of new or unfamiliar terms, must clarify concepts when used in a new context, and in general must be on the lookout for sloppy thinking. This takes willingness on the part of all involved, not to mention some practice.

With Apache Isis, though, the ubiquitous language evolves with scarcely any effort at all. For the business experts, the Apache Isis viewers show the domain concepts they identify and the relationships between those concepts in a straightforward fashion. Meanwhile, the developers can devote themselves to encoding those domain concepts directly as domain classes. There's no technology to get distracted by; there is literally nothing else for the developers to work on.

DDD must be grounded

Employing a model-driven design isn't necessarily straightforward, and the development processes used by some organizations positively hinder it. It's not sufficient for the business analysts or architects to come up with some idealized representation of the business domain and then chuck it over the wall for the programmers to do their best with.

Instead, the concepts in the model must have a very literal representation in code. If we fail to do this, then we open up the communication divide, and our ubiquitous language is lost. There is literally no point having a domain model that cannot be represented in code. We cannot invent our ubiquitous language in a vacuum, and the developers must ensure that the model remains grounded in the doable.

In Apache Isis, we have a very pure one-to-one correspondence between the domain concepts and its implementation. Domain concepts are represented as classes and interfaces, easily demonstrated back to the business. If the model is clumsy, then the application will be clumsy too, and so the team can work together to find a better implementable model.

Model must be understandable

If we are using code as the primary means of expressing the model, then we need to find a way to make this model understandable to the business.

We could generate UML diagrams and the like from code. That will work for some members of the business community, but not for everyone. Or we could generate a PDF document from Javadoc comments, but comments aren't code and so the document may be inaccurate. Anyway, even if we do create such a document, not everyone will read it.

A better way to represent the model is to show it in action as a working prototype. As we show in the Getting Started section, Apache Isis enables this with ease. Such prototypes bring the domain model to life, engaging the audience in a way that a piece of paper never can.

Moreover, with Apache Isis prototypes, the domain model will come shining through. If there are mistakes or misunderstandings in the domain model (inevitable when building any complex system), they will be obvious to all.

Architecture must be robust

DDD rightly requires that the domain model lives in its own layer within the architecture. The other layers of the application (usually presentation, application, and persistence) have their own responsibilities, and are completely separate.

However, there are two immediate issues. The first is rather obvious: custom coding each of those other layers is an expensive proposition. Picking up on the previous point, this in itself can put the kibosh on using prototyping to represent the model, even if we wanted to do so.

The second issue is more subtle. It takes real skill to ensure the correct separation of concerns between these layers, if indeed you can get an agreement as to what those concerns actually are. Even with the best intentions, it's all too easy for custom-written layers to blur the boundaries and put (for example) validation in the user interface layer when it should belong to the domain layer. At the other extreme, it's quite possible for custom layers to distort or completely subvert the underlying domain model.

Because of Apache Isis' generic *OOUI*s, there's no need to write the other layers of the architecture. Of course, this reduces the development cost. But more than that, there will be no leakage of concerns outside the domain model. All the validation logic **must** be in the domain model because there is nowhere else to put it.

Moreover, although Apache Isis does provide a complete runtime framework, there is no direct coupling of your domain model to the framework. That means it is very possible to take your domain model prototyped in Naked Objects and then deploy it on some other *J(2)EE* architecture, with a custom *UI* if you want. Apache Isis guarantees that your domain model is complete.

Extending the reach of DDD

Domain-driven design is often positioned as being applicable only to complex domains; indeed, the subtitle of Evans book is "Tackling Complexity in the Heart of Software". The corollary is that DDD is overkill for simpler domains. The trouble is that we immediately have to make a choice: is the domain complex enough to warrant a domain-driven approach?

This goes back to the previous point, building and maintaining a layered architecture. It doesn't seem cost effective to go to all the effort of a DDD approach if the underlying domain is simple.

However, with Apache Isis, we don't write these other layers, so we don't have to make a call on how complex our domain is. We can start working solely on our domain, even if we suspect it will be simple. If it is indeed a simple domain, then there's no hardship, but if unexpected subtleties arise, then we're in a good position to handle them.

If you're just starting out writing domain-driven applications, then Apache Isis should significantly ease your journey into applying *DDD*. On the other hand, if you've used *DDD* for a while, then you should find Isis a very useful new tool in your arsenal.

2.2. Principles and Values

This section describes some of the core principles and values that the framework aims to honour and support.

The first of these relate to how we believe your domain application should be written: it should be decoupled, testable and so on). Others relate to the implementation of the framework itself.

The section concludes by contrasting the framework with some other open source frameworks commonly used.

2.2.1. Your Applications

Apache Isis is primarily aimed at custom-built "enterprise" applications. The UI exposed by the Wicket viewer is intended to be usable by domain experts, typically end-users within the organization. The REST API exposed by the RestfulObjects viewer allows custom apps to be developed - eg using AngularJS or similar - for use by those requiring more guidance; typically end-users outside of the organization.

But should your organization buy, or build? Buying packaged software makes sense for statutory requirements, such as payroll or general ledger, or document management/retrieval. But it makes much less sense to buy packaged software for the systems that support the core business: the software should fit the business, not the other way around.

TODO - flesh out the following:

- Flexible, "just enough"
- Decoupled
- Long-term Cost of ownership
  - dependency injection of services
  - OO design techniques, eg dependency inversion principle
  - an in-memory event bus
  - applib
  - (no "Big Ball of Mud")
- Honouring the Single Responsibility Principle
  - behaviourally Complete vs Contributions/Mixins
- Testable

  While Apache Isis can be used (very effectively) for simple CRUD-style applications, it is also intended to be used for complex business domains. Ensuring that the business logic in such applications is correct means that the framework must (and does) provide robust testing support, both for developer-level unit testing and business-level (end-to-end) integration testing.

- Reusable building blocks

  Isis addons, catalog.incode.org

2.2.2. Apache Isis itself

This section discusses some of the principles and values we apply to the development of the Apache Isis framework itself.

Full-stack but Extensible

TODO

Focuses on its USP

TODO

add-ons

- Apache Isis is at heart a metamodel with runtime, and coordinates interations using an AOP set of principles
- Apache Isis vs Isis Addons

- Apache Isis vs Shiro vs DataNucleus
  i. all code has legacy in it…. parts of the Isis codebase are well over a decade old; and back then a lot of the JEE technologies that we'd like to be using just didn't exist, so we had to invent the features we required ourselves.
  ii. also, Apache Isis today is more pragmatic than previously
- a client/server solution, with AWT-based client
- a HTML browser, Scimpi (JSF-like, but not using JSF), …
- security
- objectstores

We're working hard to remove duplication, reuse existing open source/JEE, and simplify.

The areas of Apache Isis we consider mature are those that have been developed in support of real-world applications implemented by the committers. Foremost among these is Estatio.

Focus on enterprise / line-of-business applications, for use by internal staff.

- problem solvers, not process followers
- view models

2.2.3. Apache Isis vs …

Many other frameworks promise rapid application development and provide automatically generated user interfaces, so how do they compare to Apache Isis?

vs MVC server-side frameworks

Some of most commonly used frameworks today are Spring MVC, Ruby on Rails and Grails, all of which implement one flavour or another of the server-side MVC pattern. The MVC 1.0 specification (scheduled for JavaEE 8) is also similar.

These frameworks all use the classic **model-view-controller** (*MVC*) pattern for web applications, with scaffolding, code-generation, and/or metaprogramming tools for the controllers and views, as well as convention over configuration to define how these components interact. The views provided out of the box by these frameworks tend to be simple *CRUD*-style interfaces. More sophisticated behavior is accomplished by customizing the generated controllers.

The most obvious difference when developing an Apache Isis application is its deliberate lack of an explicit controller layer; non- *CRUD* behavior is automatically made available in its generic object-oriented _UI_s. More sophisticated UIs can be built either by extending Apache Isis' Wicket viewer or by writing a bespoke UI leveraging the REST (hypermedia) API automatically exposed by Isis' Restful Objects viewer. Other frameworks can also be used to implement REST APIs, of course, but generally they require a significant amount of development to get anywhere near the level of sophistication provided automatically by Apache Isis' REST API.

Although these frameworks all provide their own ecosystems of extensions, Apache Isis' equivalent Isis Addons (non-ASF) tend to work at a higher-level of abstraction. For example, each of these frameworks will integrate with various security mechanism, but the Isis addons' security module provides a full subdomain of users, roles, features and permissions that can be plugged into any Isis application. Similarly, the Isis addons' command and Isis addons' audit modules in combination provide a support for auditing and traceability that can also be used for out of the box profiling. Again, these addons can be plugged into any Isis app.

In terms of testing support, each of these other frameworks provide mechanisms to allow the webapp to be tested from within a JUnit test harness. Apache Isis' support is similar. Where Apache Isis differs though is that it enables end-to-end testing without the need for slow and fragile Selenium tests. Instead, Apache Isis provides a "WrapperFactory" domain service that allows the generic UI provided to in essence be simulated. On a more pragmatic level, the Isis addons' fakedata module does "what it says on the tin", allowing both unit- and integration-tests to focus on the salient data and fake out the rest.

vs CQRS

The CQRS architectural pattern (it stands for "Command Query Responsbility Separation") is the idea that the domain objects that mutate the state of the system - to which commands are sent and which then execute - should be separated from the mechanism by which the state of the system is queried (rendered). The former are sometimes called the "write (domain) model", the latter the "read model".

In the canonical version of this pattern there are separate datastores. The commands act upon a command/write datastore. The data in this datastore is then replicated in some way to the query/read datastore, usually denormalized or otherwise such that it is easy to query. CQRS advocates recommend using very simple (almost naive) technology for the query/read model; it should be a simple projection of the query datastore. Complexity instead lives elsewhere: business logic in the command/write model, and in the transformation logic betweeen the command/write and read/query datastores. In particular, there is no requirement for the two datastores to use the same technology: one might be an RDBMS while the other a NoSQL datastore or even datawarehouse.

In most implementations the command and query datastores are *not* updated in the same transaction; instead there is some sort of replication mechanism. This also means that the query datastore is eventually consistent rather than always consistent; there could be a lag of a few seconds before it is updated. This means in turn that CQRS implementations require mechanisms to cater for offline query datastores; usually some sort of event bus.

The CQRS architecture's extreme separation of responsibilities can result in a lot of boilerplate. Any given domain concept, eg `Customer`, must be represented both in the command/write model and also in the query/read model. Each business operation upon the command model is reified as a command object, for example `PlaceOrderCommand`.

Comparing CQRS to Apache Isis, the most obvious difference is that Apache Isis does not separate out a command/write model from a query/read model, and there is usually just a single datastore. But then again, having a separate read model just so that the querying is very straightforward is pointless with Apache Isis because, of course, Isis provides the UI "for free".

There are other reasons though why a separate read model might make sense, such as to precompute particular queries, or against denormalized data. In these cases Apache Isis can often provide a reasonable alternative, namely to map domain entities against RDBMS views, either materialized views or dynamic. In such cases there is still only a single physical datastore, and so transactional integrity is retained.

Or, the CQRS architecture can be more fully implemented with Apache Isis by introducing a separate read model, synchronized using the `PublishingService`, or using subscribers on the `EventBusService`. One can then use view models to surface the data in the external read datastore.

With respect to commands, Apache Isis does of course support the `CommandService` which allows each business action to be reified into a `Command`. However, names are misleading here: Apache Isis' commands are relatively passive, merely recording the intent of the user to invoke some operation. In a CQRS architecture, though, commands take a more active role, locating and acting upon the domain objects. More significantly, in CQRS each command has its own class, such as `PlaceOrderCommand`, instantiated by the client and then executed. With Apache Isis, though, the end-user merely invokes the `placeOrder(…)` action upon the domain object; the framework itself creates the `Command` as a side-effect of this.

In CQRS the commands correspond to the business logic that mutates the system. Whether this logic is part of the command class (`PlaceOrderCommand`) or whether that command delegates to methods on the domain object is an implementation detail; but it certainly is common for the business logic to be wholly within the command object and for the domain object to be merely a data holder of the data within the command/write datastore.

In Apache Isis this same separation of business logic from the underlying data can be accomplished most straightforwardly using mixins or contributions. In the UI (surfaced by the Wicket viewer) or in the REST API (surfaced by the RestfulObjects viewer) the behaviour appears to reside on the domain object; however the behaviour actually resides on separate classes and is mixed in (like a trait) only at runtime.

vs Event Sourcing

The CQRS architecture, discussed above, is often combined with *Event Sourcing* pattern, though they are separate ideas.

With event sourcing, each business operation emits a domain event (or possibly events) that allow other objects in the system to act accordingly. For example, if a customer places an order then this might emit the `OrderPlacedEvent`. Most significantly, the subscribers to these events can include the datastore itself; the state of the system is in effect a transaction log of every event that has occurred since "the beginning of time": it is sometimes called an event store. With CQRS, this event datastore corresponds to the command/write datastore (the query/read datastore is of course derived from the command datastore).

Although it might seem counter-intuitive to be able store persistent state in this way (as a souped up "transaction log"), the reality is that with modern compute capabilities make it quite feasible to replay many 10s/100s of thousands of events in a second. And the architecture supports some interesting use cases; for example it becomes quite trivial to rewind the system back to some previous point in time.

When combined with CQRS we see a command that triggers a business operation, and an event that results from it. So, a `PlaceOrderCommand` command can result in an `OrderPlacedEvent` event. A subscriber to this event might then generate a further command to act upon some other system (eg to dispatch the system). Note that the event might be dispatched and consumed in-process or alternatively this might occur out-of-process. If the latter, then the subscriber will operate within a separate transaction, meaning the usual eventual consistency concerns and also compensating actions if a rollback is required. CQRS/event sourcing advocates point out - correctly that this is just how things are in the "real world" too.

In Apache Isis every business action (and indeed, property and collection) emits domain events through the `EventBusService` , and can optionally also be published through the `PublishingService`. The former are dispatched and consumed in-process and within the same transaction, and for this reason the subscribers can also veto the events. The latter are intended for out-of-process consumption; the (non-ASF) Isis addons' publishing and Isis addons' publishmq modules provide implementations for dispatching either through a RDBMS database table, or directly through to an ActiveMQ message queue (eg wired up to Apache Camel event bus).

vs MetaWidget

MetaWidget (mentioned earlier has a number of ideas in common with Apache Isis, specifically the runtime generation of a UI for domain objects. And like Apache Isis, MetaWidget builds its own metamodel of the domain objects and uses this to render the object.

However, there is a difference in philosophy in that MW is not a full-stack framework and does not (in their words) try to "own the UI". Rather they support a huge variety of UI technologies and allow the domain object to be rendered in any of them.

In contrast, Apache Isis is full-stack and does generate a complete UI; we then allow you to customize or extend this UI (as per the various Isis Addons (non-ASF), and we also provide a full REST API through the Restful Objects viewer

Also, it’s worth noting that MetaWidget does have an elegant pipeline architecture, with APIs to allow even its metamodel to be replaced. It would be feasible and probably quite straightforward to use Apache Isis' own metamodel as an implementation of the MetaWidget API. This would allow MetaWidget to be able to render an Apache Isis domain application.

2.3. Building Blocks

In this section we run through the main building blocks that make up an Apache Isis application.

2.3.1. A MetaModel

At its core, Apache Isis is a metamodel that is built at runtime from the domain classes (eg `Customer.java`), along with optional supporting metadata (eg `Customer.layout.json`).

The contents of this metamodel is inferred from the Java classes discovered on the classpath: the entities and supporting services, as well the members of those classes. The detail of the metamodel is generally explicit, usually represented by Java annotations such as `@Title` or `@Action`. Notably the metamodel is extensible; it is possible to teach Apache Isis new programming conventions/rules (and conversely to remove those that are built in).

Most of the annotations recognized by the framework are defined by the Apache Isis framework itself. For example the `@Title` annotation - which identifies how the framework should derive a human-readable label for each rendered domain object - is part of the `org.apache.isis.applib.annotations` package. However the framework also recognizes certain other JEE annotations such as `@javax.inject.Inject` (used for dependency injection).

The framework uses DataNucleus for its persistence mechanism. This is an ORM that implements the JDO and JPA APIs, and which can map domain objects either to an RDBMS or to various NoSQL objectstores such as MongoDB or Neo4J. Apache Isis recognizes a number of the JDO annotations such as `@javax.jdo.annotations.Column(allowNulls=…)`.

In addition, the framework builds up the metamodel for each domain object using layout hints, such as `Customer.layout.json`. These provide metadata such as grouping elements of the UI together, using multi-column layouts, and so on. The layout file can be modified while the application is still running, and are picked up automatically; a useful way to speed up feedback.

> At the time of writing Apache Isis only recognizes and supports the JDO API, though we expect JPA to be supported in the future. We also expect to generalize support for `.layout.json` to be able to read such metadata from other sources.

2.3.2. Type of Domain Objects

Most domain objects that the end-user interacts with are **domain entities**, such as `Customer`, `Order`, `Product` and so on. These are persistent objects and which are mapped to a database (usually relational), using JDO/DataNucleus annotations. From the end-user's perspective the UI displays a single domain object per page; they can then inspect and modify its state, and navigate to related objects.

The next type of domain object to discuss is **domain services**. These are (usually) singleton stateless services that provide additional functionality. The behaviour of these services is rendered in various ways, though the most obvious is as the menu actions on the top-level menu bars in the Wicket viewer's UI.

Domain objects can also delegate to domain services; domain services are automatically injected into every other domain object; this includes domain entities as well as other services. This injection of domain services into entities is significant: it allows business logic to be implemented in the domain entities, rather than have it "leach away" into supporting service layers. Said another way: it is the means by which Apache Isis helps you avoid the anaemic domain model anti-pattern.

As well as domain entities - mapped to a datastore - Apache Isis also supports **view models**. End users interact with view models in the same way as a domain entity, indeed they are unlikely to distinguish one from the other. However view models are *not* mapped to the underlying database, rather they represent some aggregation of state from one or more underlying entities. Their state is serialized and recreated from their internal identifier; this identifier is visible as the object's URL in the Wicket viewer or RestfulObjects viewer.

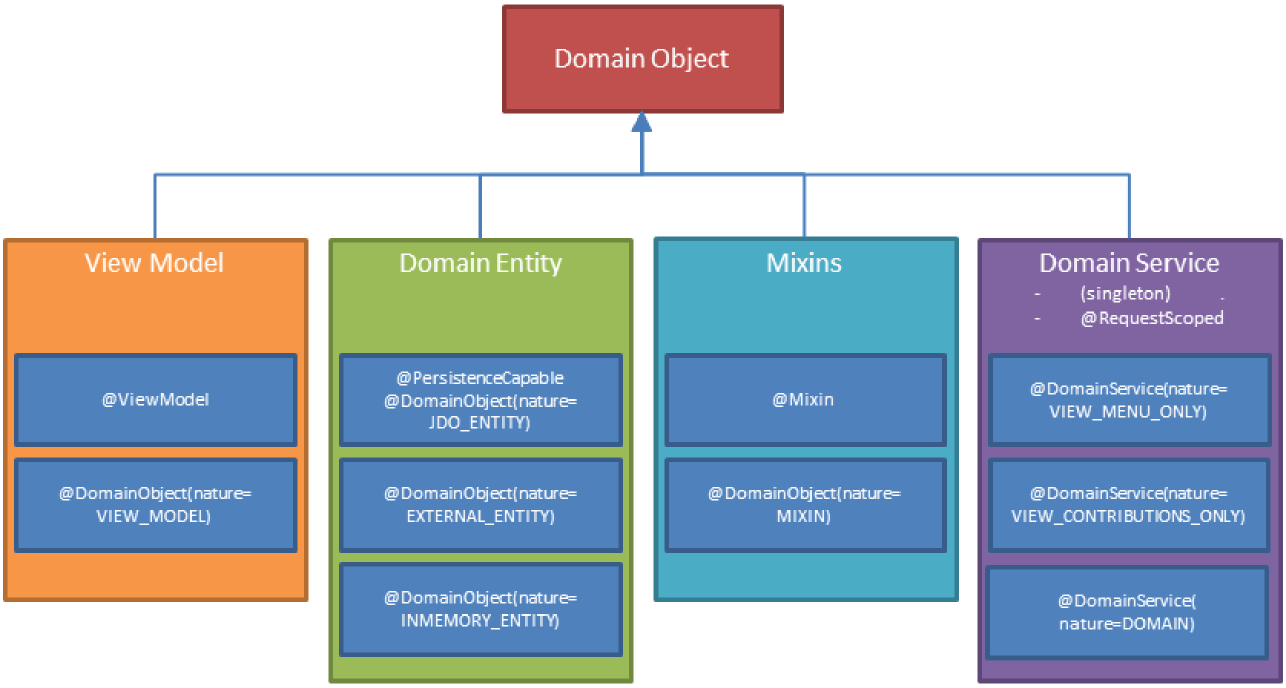
There's no need though for the view model to aggregate the state of regular domain entities. A view model could also be used as a proxy for some externally managed entity, accessed over a web service or REST API; it could even be a representation of state held in-memory (such as user preferences, for example).

There are also several types of domain services. Most easily described are those domain services (discussed above) that are represented as the menu actions on top-level menu bars. Another variation are **contributed services** - domain services that contribute behaviour or (derived) state to entities/view models. Finally domain services may also simply provide additional non-UI functionality; an example being to perform an address geocoding lookup against the google-maps API.

Also worth mentioning: domain services can also be either singletons (discussed above) or request-scoped; the latter being annotated with `@javax.enterprise.context.RequestScoped`. An example of the request-scoped service is the `Scratchpad` service, for sharing arbitrary data between multiple objects.

The final type of domain object is the **mixin**. These are similar to contributed services in that they also contribute (or rather, mixin) both behaviour or (derived) state to entities/view models. However, they provide a more control over contributed services, with a cleaner programming model similar to traits found in other languages.

The diagram below summarizes the various types of domain object:



The Apache Isis programming model uses annotations to distinguish these object types:

- **view models** are annotated either with `@DomainObject(nature=VIEW_MODEL)` or using `@ViewModel`. Which is used is a matter of personal preference.

  It is also possible to implement the `ViewModel` interface, for finer-grained control.

- **domain entities** that are persisted to the database (as the vast majority will) are annotated with `@DomainObject(nature=ENTITY)`. In addition such domain entities are annotated with the JDO/DataNucleus annotation of `@javax.jdo.annotations.PersistenceCapable`.

In addition, if a domain entity is a proxy for state managed in an external system, or merely for some state held in-memory, then `@DomainObject(nature=EXTERNAL_ENTITY)` or `@DomainObject(nature=INMEMORY_ENTITY)` can be used.

- **mixins** are annotated either with `@DomainObject(nature=MIXIN)` or using `@Mixin`. As for view models, which is used is a matter of personal preference.
- finally, **domain services**` are annotated with `@DomainService(nature=…)` where the nature is either `VIEW_MENU_ONLY` (for domain services whose actions appear on the top-level menu bars), or `VIEW_CONTRIBUTIONS_ONLY` (for domain services whose actions are contributed to entities or view models), or `DOMAIN` (for domain services whose functionality is simply for other domain objects to invoke programmatically).

  It is also possible to specify a nature of simply `VIEW`, this combining `VIEW_MENU_ONLY` and `VIEW_CONTRIBUTIONS_ONLY`. This is in fact the default, useful for initial prototyping. A final nature is `VIEW_REST_ONLY` which is for domain services whose functionality is surfaced only by the RestfulObjects viewer.

Worth emphasising is that domain entities and view models hold state, whereas domain services are generally stateless. If a domain service does hold state (eg the `Scratchpad` service noted above) then it should be `@RequestScoped` so that this state is short-lived and usable only within a single request.

2.3.3. Object Members

Every domain object in Apache Isis consists of (at most) three types of members:

- properties, such as a `Customer’s `firstName
- collections, such as a `Customer’s `orders` collection of `Order`s
- actions, such as a `Customer'`s `placeOrder(…)` method.

Some domain objects - specifically domain services and mixins - only have actions. In the case of contributing services and mixins these actions can (depending upon their semantics and signatures) be represented as derived properties or collections on the entity/view model to which they contribute/mix-in.

Properties

Properties follow the standard getter/setter pattern, with the return type being a scalar (a value object or another entity or view model).

For example, with:

```
public class Customer
    private String firstName;
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    ...
}
```

the framework infers the `Customer` domain entity, which in turn has a `firstName` string *property*.

Collections

Collections are also represented by a getter and setter, however the return type is a `Collection` or subtype.

For example, with:

```
public class Customer
    private SortedSet<Order> orders = new TreeSet<Order>();
    public SortedSet<Order> getOrders() { return orders; }
    public void setOrders(SortedSet<Order> orders) { this.orders = orders; }
    ...
}
```

the framework infers the `orders` *collection*.

> The most commonly used collection type is `java.util.SortedSet`; entities are most commonly mapped to a relational database (ie a datastore with set semantics) and we recommend that all entities define a natural ordering so that when rendered in the UI they will be ordered "meaningfully" to the end-user.

Actions

The third type of object member is actions. (To a first approximation), actions are all public methods that do not represent properties or collections.

For example:

```
public class Customer
    public Customer placeOrder(Product p, int quantity) { ... }
    ...
}
```

corresponds to the `placeOrder` *action*.

The above *is* a simplification; the Apache Isis programming model also recognizes a number of other supporting methods each of which has its own prefix such as `hide`, `disable` or `validate`. These can be considered as "reserved words" in Apache Isis, and do *not* correspond to actions even though they have public visibility.

2.3.4. Entities vs View Models

When developing an Apache Isis application you will most likely start off with the persistent domain entities: `Customer`, `Order`, `Product`, and so on. For some applications this may well suffice. However, if the application needs to integrate with other systems, or if the application needs to support reasonably complex business processes, then you may need to look beyond just domain entities.

To support these use cases we support view models. In the same way that an (RDBMS) database view can aggregate and abstract from multiple underlying database tables, so a view model sits on top of one or many underlying entities.

View models are not persisted, but nevertheless they can have behaviour (and titles, and icons) just like domain entities. Indeed, to a user of the system there is no particular distinction (again, in the same way that when using an RDBMS one can use database views and database tables pretty much interchangeably).

View models generally tend to be associated with supporting a particular use case; logically they are part of the application layer, not part of the domain layer (where entities live).

We introduce view models here because they do get mentioned quite often within the users and reference guide. However, we do consider them a more advanced topic; we generally recommend that you build your applications from the domain layer up, rather than from the view model down.

For further discussion on view models, see this topic.

2.3.5. Domain Services

Domain services consist of a set of logically grouped actions, and as such follow the same conventions as for entities. However, a service cannot have (persisted) properties, nor can it have (persisted) collections.

Domain services are instantiated once and once only by the framework, and are used to centralize any domain logic that does not logically belong in a domain entity or value. Apache Isis will automatically inject services into every domain entity that requests them, and into each other.

For convenience you can inherit from AbstractService or one of its subclasses, but this is not mandatory.

Domain Services vs View Services

TODO

`@DomainService(nature=…)`

> **Factories, Repositories and Services**
>
> A distinction is sometimes made between a factory (that creates object) and a repository (that is used to find existing objects). You will find them discussed separately in Evans' Domain Driven Design, for example.
>
> In Apache Isis these are all implemented as domain services. Indeed, it is quite common to have a domain service that acts as both a factory and a repository.

2.3.6. Mixins & Contributions

> TODO

For more information, see this topic on contributions, and this topic on mixins.

2.3.7. Domain Events

> TODO; see domain event classes.

UI Events

> TODO; see UI event classes.

2.3.8. OIDs

As well as defining a metamodel of the structure (domain classes) of its domain objects, Apache Isis also manages the runtime instances of said domain objects.

When a domain entity is recreated from the database, the framework keeps track of its identity through an "OID": an object identifier. Fundamentally this is a combination of its type (domain class), along with an identifier. You can think of it as its "primary key", except across all domain entity types.

For portability and resilience, though, the object type is generally an alias for the actual domain class: thus "customers.CUS", say, rather than "com.mycompany.myapp.customers.Customer". This is derived from an annotation. The identifier meanwhile is always converted to a string.

Although simple, the OID is an enormously powerful concept: it represents a URI to any domain object managed by a given Apache Isis application. With it, we have the ability to lookup any arbitrary domain objects.

Some examples:

- an OID allows sharing of information between users, eg as a deep link to be pasted into an email.
- the information within an OID could be converted into a barcode, and stamped onto a PDF form. When the PDF is scanned by the mail room, the barcode could be read to attach the

correspondence to the relevant domain object.

- as a handle to any object in an audit record (as used by `AuditingService` or `AuditerService` (`1.13.0-SNAPSHOT`));
- similarly within implementations of `CommandService` to persist `Command` objects
- similarly within implementations of `PublisherService` (`1.13.0-SNAPSHOT`) to persist published action invocations
- and of course both the RestfulObjects viewer and Wicket viewer use the oid tuple to look up, render and allow the user to interact with domain objects.

Although the exact content of an OID should be considered opaque by domain objects, it is possible for domain objects to obtain OIDs. These are represented as `Bookmark`s, obtained from the `BookmarkService`. Deep links meanwhile can be obtained from the `@DeepLinkService`.

OIDs can also be converted into XML format, useful for integration scenarios. The common schema XSD defines the `oidDto` complex type for precisely this purpose.

2.3.9. Value Objects (Primitives)

TODO

2.4. Framework-provided Services

Most framework domain services are API: they exist to provide support functionality to the application's domain objects and services. In this case an implementation of the service will be available, either by Apache Isis itself or by Isis Addons (non ASF).

Some framework domain services are SPI: they exist primarily so that the application can influence the framework's ehaviour. In these cases there is (usually) no default implementation; it is up to the application to provide an implementation.

General purpose:

- `DomainObjectContainer`; mostly deprecated, replaced by:
  - `ClockService`
  - `ConfigurationService`
  - `MessageService`
  - `RepositoryService`
  - `ServiceRegistry`
  - `TitleService`
  - `UserService`
- `IsisJdoSupport`
- `WrapperFactory`
- `EventBusService`
- `EmailService`

Commands/Interactions/Background/Auditing/Publishing/Profiling:

- `CommandContext` (SPI)
- `CommandService` (SPI)
- `CommandMementoService` (SPI)
- `InteractionContext` (SPI)
- `AuditingService` (SPI)
- `AuditerService` (SPI) (`1.13.0-SNAPSHOT`)
- `BackgroundService`
- `BackgroundCommandService` (SPI)
- `PublishingService` (SPI)
- `PublishererService` (SPI) (`1.13.0-SNAPSHOT`)
- `MetricsService`

Information Sharing:

- `Scratchpad`
- `ActionInvocationContext`
- `QueryResultsCache`

UserManagement:

- `UserProfileService` (SPI)
- `UserRegistrationService` (SPI)
- `EmailNotificationService` (SPI)

Bookmarks and Mementos:

- `BookmarkService`
- `MementoService`
- `DeepLinkService`
- `JaxbService`
- `XmlSnapshotService`

Layout and UI Management:

- `HomePageProviderService`
- `LayoutService`
- `GridLoaderService` (SPI)
- `GridService` (SPI)
- `GridSystemService` (SPI)
- `HintStore` (SPI)
- `RoutingService` (SPI)

* `UrlEncodingService` (SPI)

REST Support:

* `AcceptHeaderService`
* `SwaggerService`
* `ContentMappingService` (SPI)

Metamodel:

* `ApplicationFeatureRepository`
* `MetamodelService`

Other API:

* `FixtureScriptsDefault`
* `GuiceBeanProvider`
* `SudoService`
* `TransactionService`

Other SPI:

* `ClassDiscoveryService` (SPI)
* `ErrorReportingService` (SPI)
* `EventSerializer` (SPI)
* `ExceptionRecognizer` (SPI)
* `FixtureScriptsSpecificationProvider` (SPI)
* `LocaleProvider` (SPI)
* `TranslationService` (SPI)
* `TranslationsResolver` (SPI)
* `TranslationsResolver` (SPI)

A full list of services can be found in the Domain Services reference guide.

2.5. Isis Add-ons

The Isis Addons website provides a number of reusable modules and other extensions for Apache Isis. This chapter focuses just on the modules, all of which have a name of the form `isis-module-xxx`.

> Note that Isis addons, while maintained by Apache Isis committers, are not part of the ASF.

The modules themselves fall into four broad groups:

* modules that provide an implementations of API defined by Apache Isis

where Apache Isis has hooks to use the service if defined by provides no implementations of its own. The command, auditing, publishing, security and sessionlogger modules fall into this category. Typically the domain objects themselves wouldn't interact with these services

- modules that provide standalone domain services with their own API and implementation

  These are simply intended to be used by domain objects. The docx, excel, settings and stringinterpolator fall into this category.

- modules that provide standalone domain entities (and supporting services) for a particular subdomain

  The tags module falls into this category

- modules that provide developer utilities

  Not intended for use by either the framework or domain objects, but provide utilities that the developer themselves might use. The devutils module (not suprisingly) falls into this category

Each of the modules has a full README and example application demonstrating their usage. The sections below briefly outline the capabilities of these modules.

2.6. Other Deployment Options

Apache Isis is a mature platform suitable for production deployment, with its "sweet spot" being line-of-business enterprise applications. So if you're looking to develop that sort of application, we certainly hope you'll seriously evaluate it.

But there are other ways that you can make Apache Isis work for you; in this chapter we explore a few of them.

2.6.1. Deploy to production

Let's start though with the default use case for Apache Isis: building line-of-business enterprise applications, on top of its Wicket viewer.

Apache Wicket, and therefore Apache Isis in this configuration, is a stateful architecture. As a platform it is certainly capable of supporting user bases of several thousand (with perhaps one or two hundred concurrent); however it isn't an architecture that you should try to scale up to tens of thousands of concurrent users.

The UI UI generated by the Wicket viewer is well suited to many line-of-business apps, but it's also worth knowing that (with a little knowledge of the Wicket APIs) it relatively straightforward to extend. As described in Isis addons chapter, the viewer already has integrations with google maps, a full calendar and an export to Excel component. We are also aware of integrations with SVG images (for floor maps of shopping center) and of custom widgets displaying a catalogue (text and images) of medical diseases.

Deploying on Apache Isis means that the framework also manages object persistence. For many line-of-business applications this will mean using a relational database. It is also possible (courtesy of its integratinon with DataNucleus) to deploy an Isis app to a NoSQL store such as Neo4J or MongoDB; and it is also possible to deploy to cloud platforms such as Google App Engine (GAE).

2.6.2. Use for prototyping

Even if you don't intend to deploy your application on top of Apache Isis, there can be a lot of value in using Apache Isis for prototyping. Because all you need do to get an app running is write domain objects, you can very quickly explore a domain object model and validate ideas with a domain expert.

By focusing just on the domain, you'll also find that you start to develop a ubiquitous language - a set of terms and concepts that the entire team (business and technologists alike) have a shared understanding.

Once you've sketched out your domain model, you can then "start-over" using your preferred platform.

2.6.3. Deploy on your own platform

The programming model defined by Apache Isis deliberately minimizes the dependencies on the rest of the framework. In fact, the only hard dependency that the domain model classes have on Apache Isis is through the `org.apache.isis.applib` classes, mostly to pick up annotations such as `@Disabled`. So, if you have used Apache Isis for prototyping (discussed above), then note that it's quite feasible to take your domain model a the basis of your actual development effort; Apache Isis' annotations and programming conventions will help ensure that any subtle semantics you might have captured in your prototyping are not lost.

If you go this route, your deployment platform will of course need to provide similar capabilities to Apache Isis. In particular, you'll need to figure out a way to inject domain services into domain entities (eg using a JPA listener), and you'll also need to reimplement any domain services you have used that Apache Isis provides "out-of-the-box" (eg `QueryResultsCache` domain service).

2.6.4. Deploy the REST API

REST (Representation State Transfer) is an architectural style for building highly scalable distributed systems, using the same principles as the World Wide Web. Many commercial web APIs (twitter, facebook, Amazon) are implemented as either pure REST APIs or some approximation therein.

The Restful Objects specification defines a means by a domain object model can be exposed as RESTful resources using JSON representations over HTTP. Apache Isis' RestfulObjects viewer is an implementation of that spec, making any Apache Isis domain object automatically available via REST.

There are a number of use cases for deploying Isis as a REST API, including:

* to allow a custom UI to be built against the RESTful API

  For example, using AngularJS or some other RIA technology such as Flex, JavaFX, Silverlight

* to enable integration between systems

  REST is designed to be machine-readable, and so is an excellent choice for synchronous data interchange scenarios.

* as a ready-made API for migrating data from one legacy system to its replacement.

As for the auto-generated webapps, the framework manages object persistence. It is perfectly possible to deploy the REST API alongside an auto-generated webapp; both work from the same domain object model.

2.6.5. Implement your own viewer

Isis' architecture was always designed to support multiple viewers; and indeed Apache Isis out-of-the-box supports two: the Wicket viewer, and the Restful Objects viewer (or three, if one includes the Wrapper Factory).

While we mustn't understate the effort involved here, it is feasible to implement your own viewers too. Indeed, one of Apache Isis' committers does indeed have a (closed source) viewer, based on Wavemaker.

Chapter 3. Getting Started

To get you up and running quickly, Apache Isis provides a SimpleApp archetype to setup a simple application as the basis of your own apps. This is deliberately very minimal so that you won't have to spend lots of time removing generated artifacts. On the other hand, it does set up a standard multi-module maven structure with unit- and integration tests pre-configured, as well as a webapp module so that you can easily run your app. We strongly recommend that you preserve this structure as you develop your own Isis application.

In this chapter we also discuss the DataNucleus enhancer. DataNucleus is the reference implementation of the JDO (Java data objects) spec, and Apache Isis integrates with DataNucleus as its persistence layer. The enhancer performs post-processing on the bytecode of your persistent domain entities, such that they can be persisted by Apache Isis' JDO/DataNucleus objectstore.

> The SimpleApp archetype automatically configures the enhancer, so there's little you need to do at this stage. Even so we feel it's a good idea to be aware of this critical part of Apache Isis runtime; if the enhancer does not run, then you'll find the app fails to start with (what will seem like) quite an obscure exception message.

3.1. Prerequisites

Apache Isis is a Java based framework, so in terms of prerequisites, you'll need to install:

- Java 7 or 8 JDK
- Apache Maven 3.0.5 or later

You'll probably also want to use an IDE; the Apache Isis committers use either IntelliJ or Eclipse; in the Developers' Guide we have detailed setup instructions for using these two IDEs. If you're a NetBeans user you should have no problems as it too has strong support for Maven.

When building and running within an IDE, you'll also need to configure the Datanucleus enhancer. This is implemented as a Maven plugin, so in the case of IntelliJ, it's easy enough to run the enhancer as required. It should be just as straightforward for NetBeans too.

For Eclipse the maven integration story is a little less refined. All is not lost, however; DataNucleus also has an implementation of the enhancer as an Eclipse plugin, which usually works well enough.

3.2. SimpleApp Archetype

The quickest way to get started with Apache Isis is to run the simple archetype. This will generate a very simple one-class domain model, called `SimpleObject`, with a single property `name`.

There is also a corresponding `SimpleObjects` domain service which acts as a repository for `SimpleObject` entity. From this you can easily rename these initial classes, and extend to build up your own Apache Isis domain application.

Finally, the domain service also includes a `HomePageViewModel` which acts as a home page for the app.

3.2.1. Generating the App

Create a new directory, and `cd` into that directory.

To build the app from the latest stable release, then run the following command:

```
mvn archetype:generate  \
    -D archetypeGroupId=org.apache.isis.archetype \
    -D archetypeArtifactId=simpleapp-archetype \
    -D archetypeVersion=1.12.2 \
    -D groupId=com.mycompany \
    -D artifactId=myapp \
    -D version=1.0-SNAPSHOT \
    -B
```

where:

- `groupId` represents your own organization, and
- `artifactId` is a unique identifier for this app within your organization.
- `version` is the initial (snapshot) version of your app

The archetype generation process will then run; it only takes a few seconds.

We also maintain the archetype for the most current `-SNAPSHOT`; an app generated with this archetype will contain the latest features of Apache Isis, but the usual caveats apply: some features still in development may be unstable.

The process is almost identical to that for stable releases, however the `archetype:generate` goal is called with slightly different arguments:

```
mvn archetype:generate  \
    -D archetypeGroupId=org.apache.isis.archetype \
    -D archetypeArtifactId=simpleapp-archetype \
    -D archetypeVersion=1.13.0-SNAPSHOT \
    -D groupId=com.mycompany \
    -D artifactId=myapp \
    -D version=1.0-SNAPSHOT \
    -D archetypeRepository=http://repository-estatio.forge.cloudbees.com/snapshot/ \
    -B
```

where as before:

- `groupId` represents your own organization, and
- `artifactId` is a unique identifier for this app within your organization.
- `version` is the initial (snapshot) version of your app

but also:

* `archetypeVersion` is the SNAPSHOT version of Apache Isis.
* `archetypeRepository` specifies the location of our snapshot repo (hosted on CloudBees), and

The archetype generation process will then run; it only takes a few seconds.

3.2.2. Building the App

Switch into the root directory of your newly generated app, and build your app:

```
cd myapp
mvn clean install
```

where `myapp` is the `artifactId` entered above.

3.2.3. Running the App

The `simpleapp` archetype generates a single WAR file, configured to run both the Wicket viewer and the Restful Objects viewer. The archetype also configures the DataNucleus/JDO Objectstore to use an in-memory HSQLDB connection.

Once you've built the app, you can run the WAR in a variety of ways.

Using mvn Jetty plugin

First, you could run the WAR in a Maven-hosted Jetty instance, though you need to `cd` into the `webapp` module:

```
cd webapp
mvn jetty:run
```

You can also provide a system property to change the port:

```
cd webapp
mvn jetty:run -D jetty.port=9090
```

Using a regular servlet container

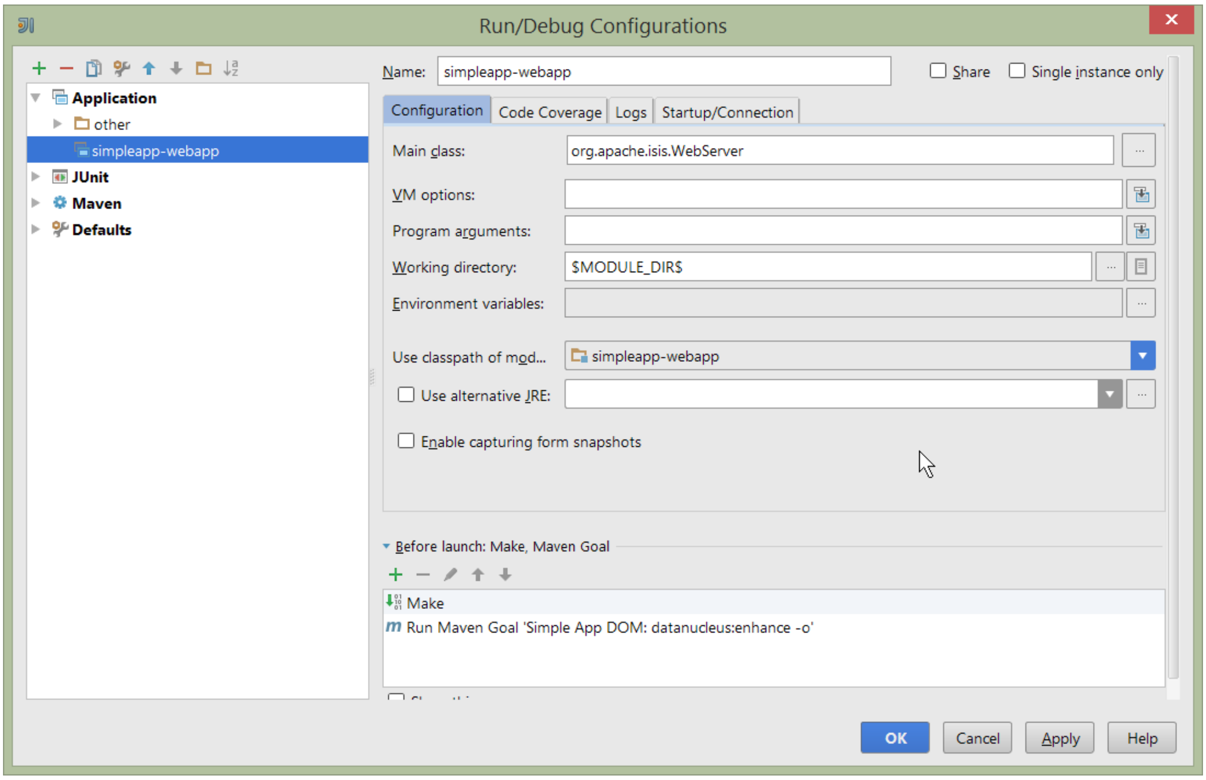
You can also take the built WAR file and deploy it into a standalone servlet container such as [Tomcat](http://tomcat.apache.org). The default configuration does not require any configuration of the servlet container; just drop the WAR file into the `webapps` directory.

From within the IDE

Most of the time, though, you'll probably want to run the app from within your IDE. The mechanics of doing this will vary by IDE; see the Developers' Guide for details of setting up Eclipse or IntelliJ

IDEA. Basically, though, it amounts to running `org.apache.isis.WebServer`, and ensuring that the DataNucleus enhancer has properly processed all domain entities.

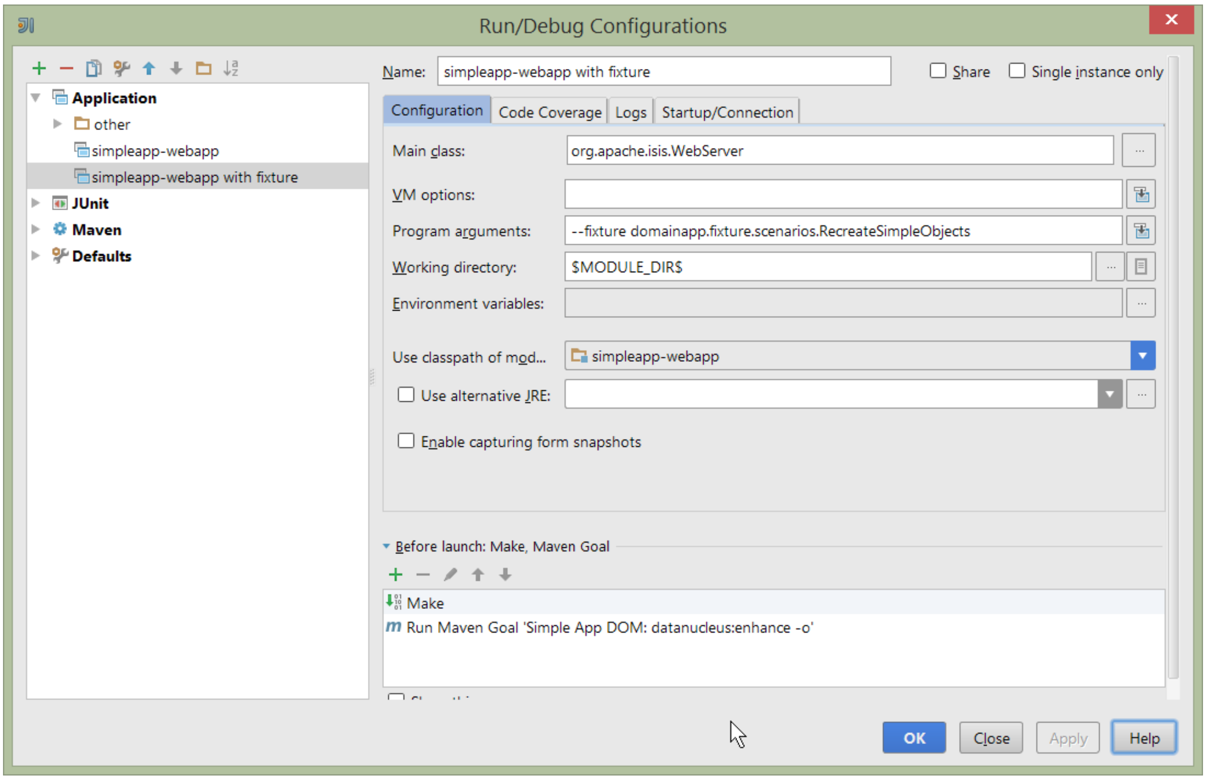
Here's what the setup looks like in IntelliJ IDEA:



3.2.4. Running with Fixtures

It is also possible to start the application with a pre-defined set of data; useful for demos or manual exploratory testing. This is done by specifying a fixture script on the command line.

If you are running the app from an IDE, then you can specify the fixture script using the `--fixture` flag. The archetype provides the `domainapp.fixture.scenarios.RecreateSimpleObjects` fixture script, for example:

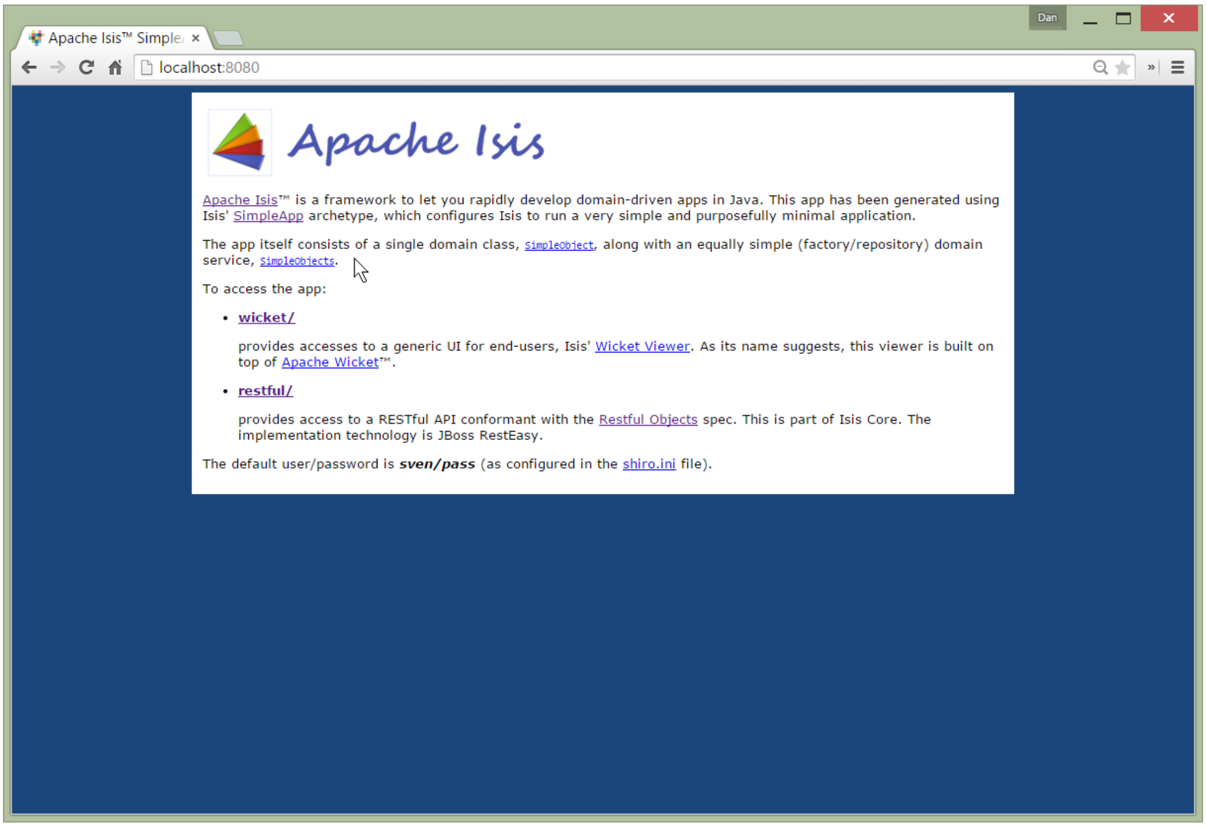


Alternatively, you can run with a different `AppManifest` using the `--appManifest` (or `-m`) flag. The archetype provides `domainapp.app.DomainAppAppManifestWithFixtures` which specifies the aforementioned `RecreateSimpleObjects` fixture.

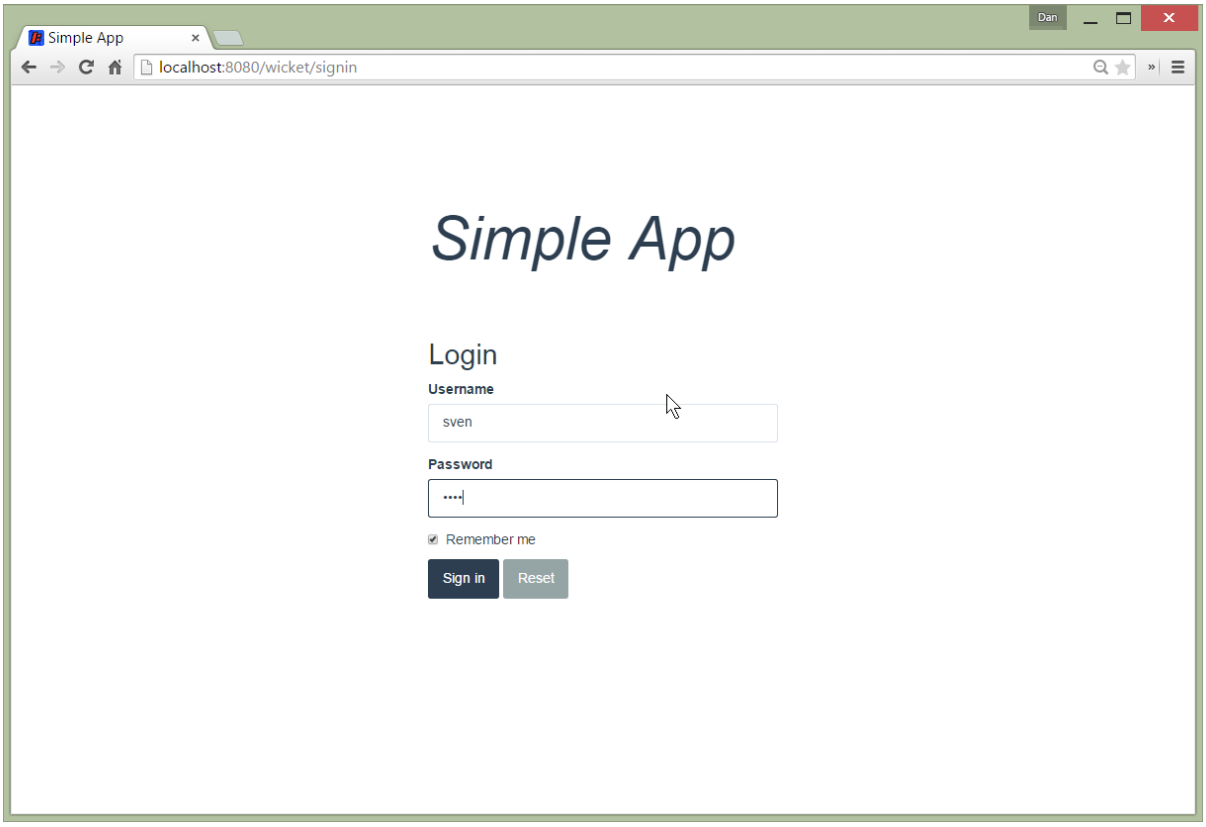
3.2.5. Using the App

> These screenshots are for v1.10.0.

When you start the app, you'll be presented with a welcome page from which you can access the webapp using either the Wicket viewer or the Restful Objects viewer:



The Wicket viewer provides a human usable web UI (implemented, as you might have guessed from its name, using Apache Wicket), so choose that and navigate to the login page:

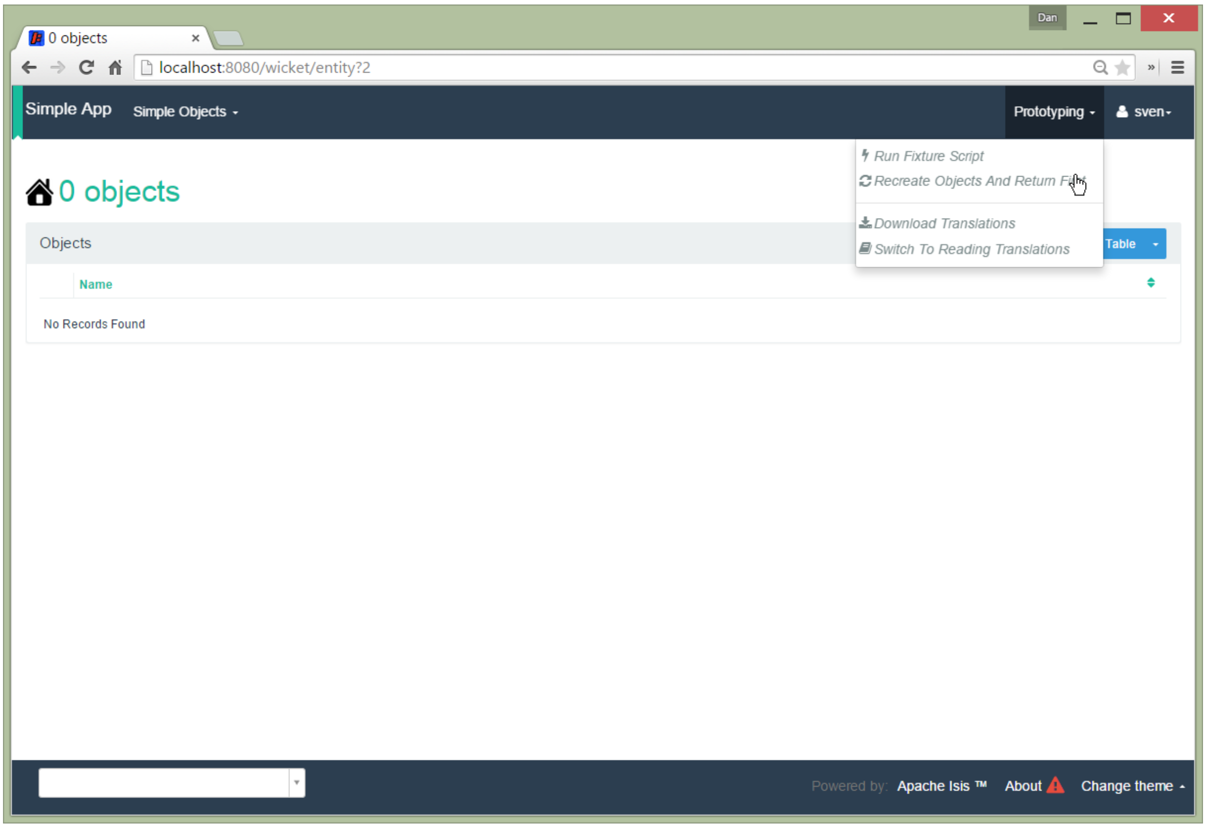


The app itself is configured to run using shiro security, as configured in the `WEB-INF/shiro.ini` config file. You can login with:

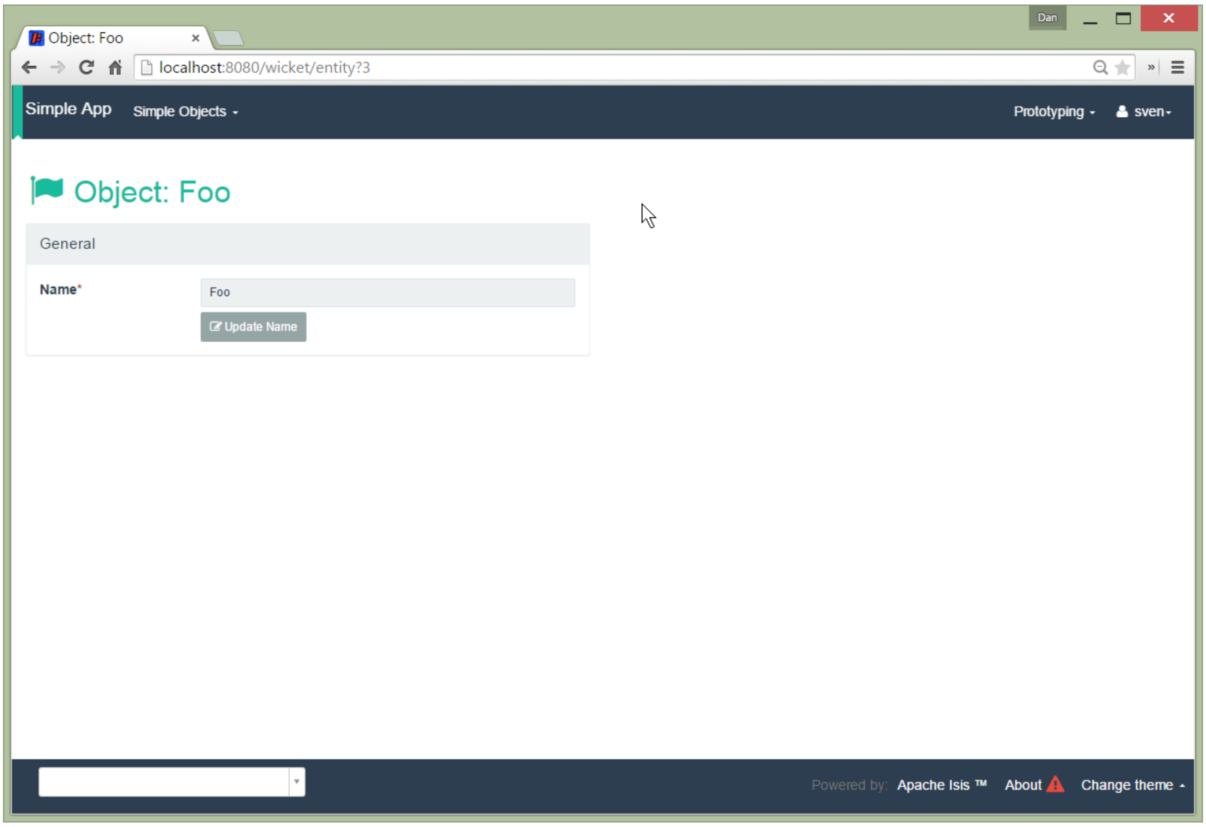
- username: *sven*

• password: *pass*

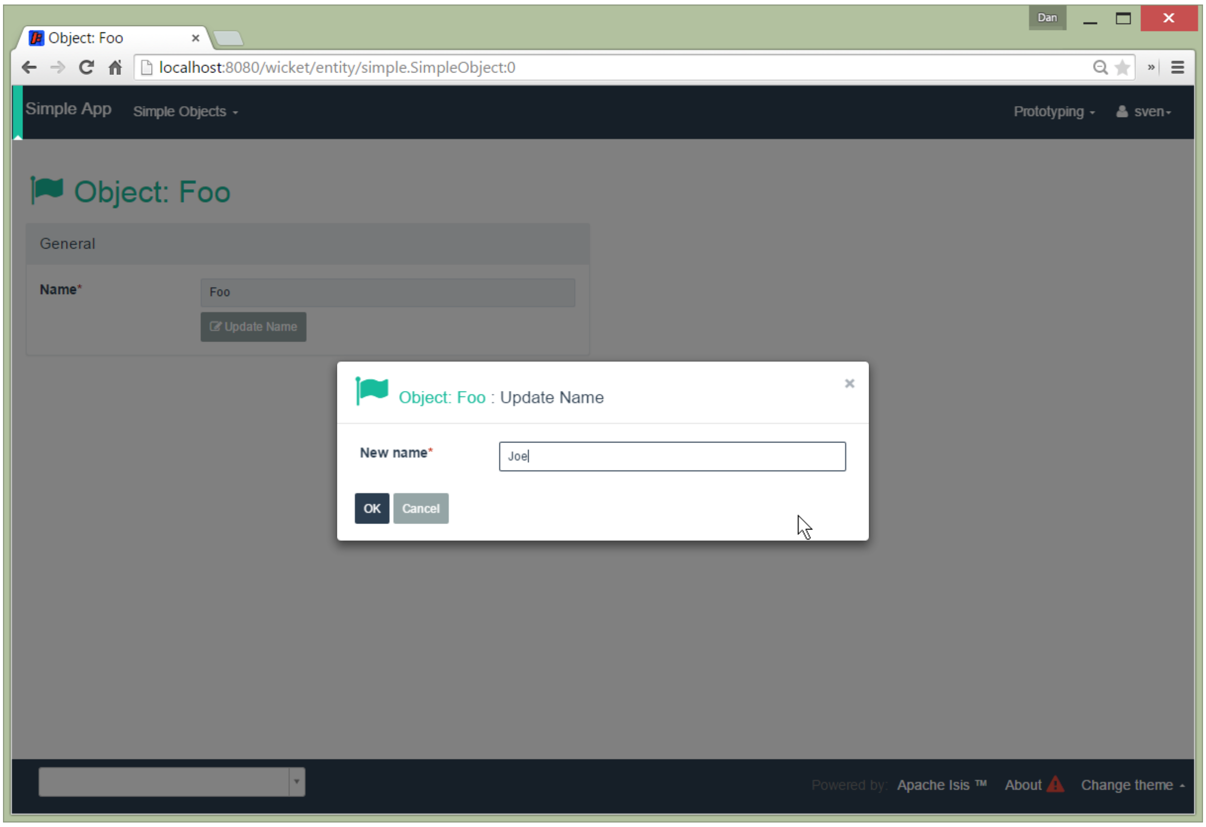
The application is configured to run with an in-memory database, and (unless you started the app with fixture scripts as described above), initially there is no data. We can though run a fixture script from the app itself:



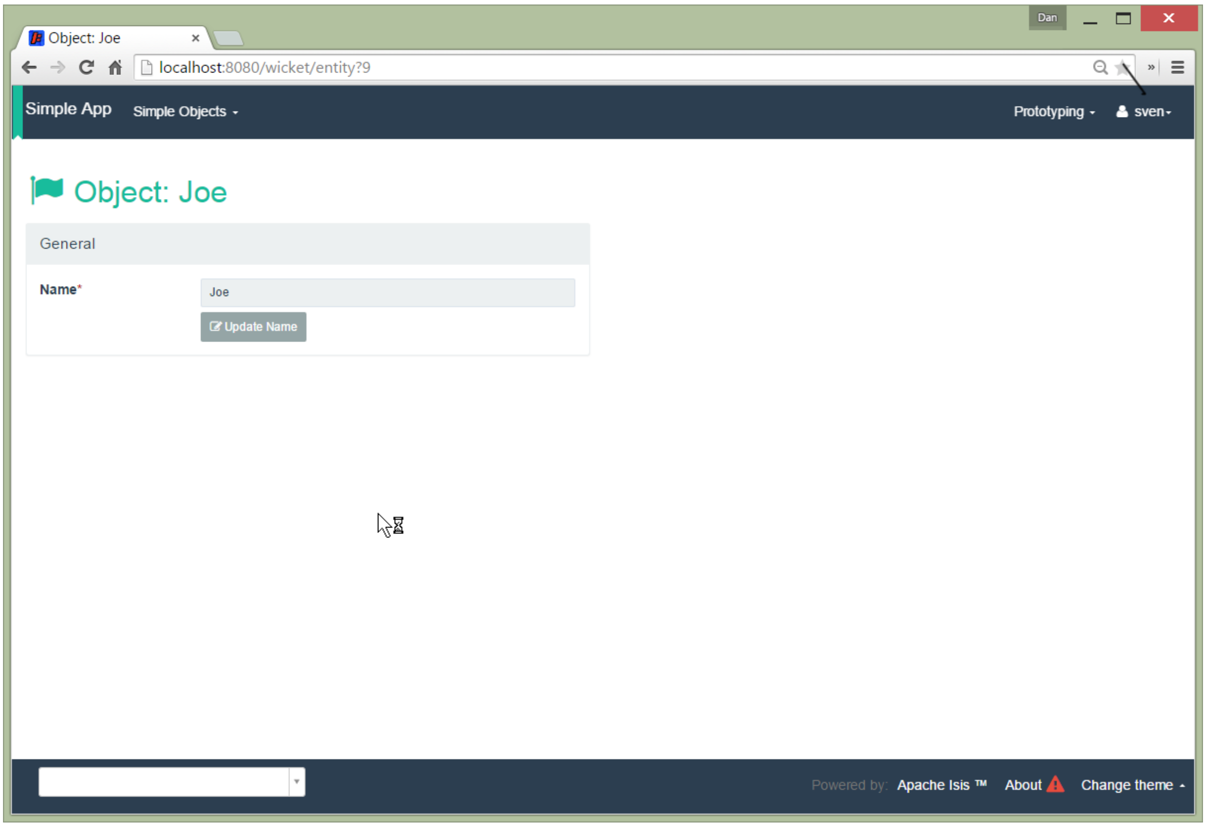
The fixture script creates three objects, and the action returns the first of these:



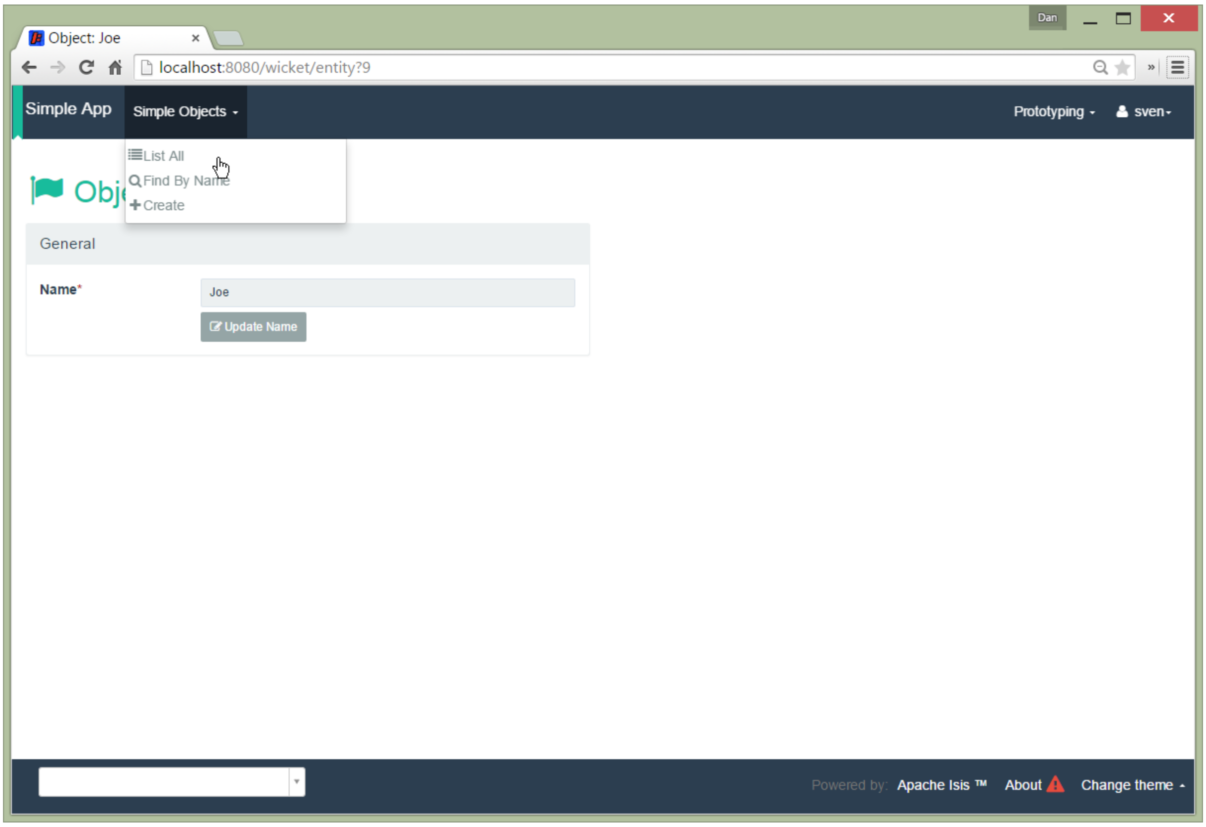
The application generated is deliberaetly very minimal; we don't want you to have to waste valuable time removing generated files. The object contains a single "name" property, and a single action to update that property:



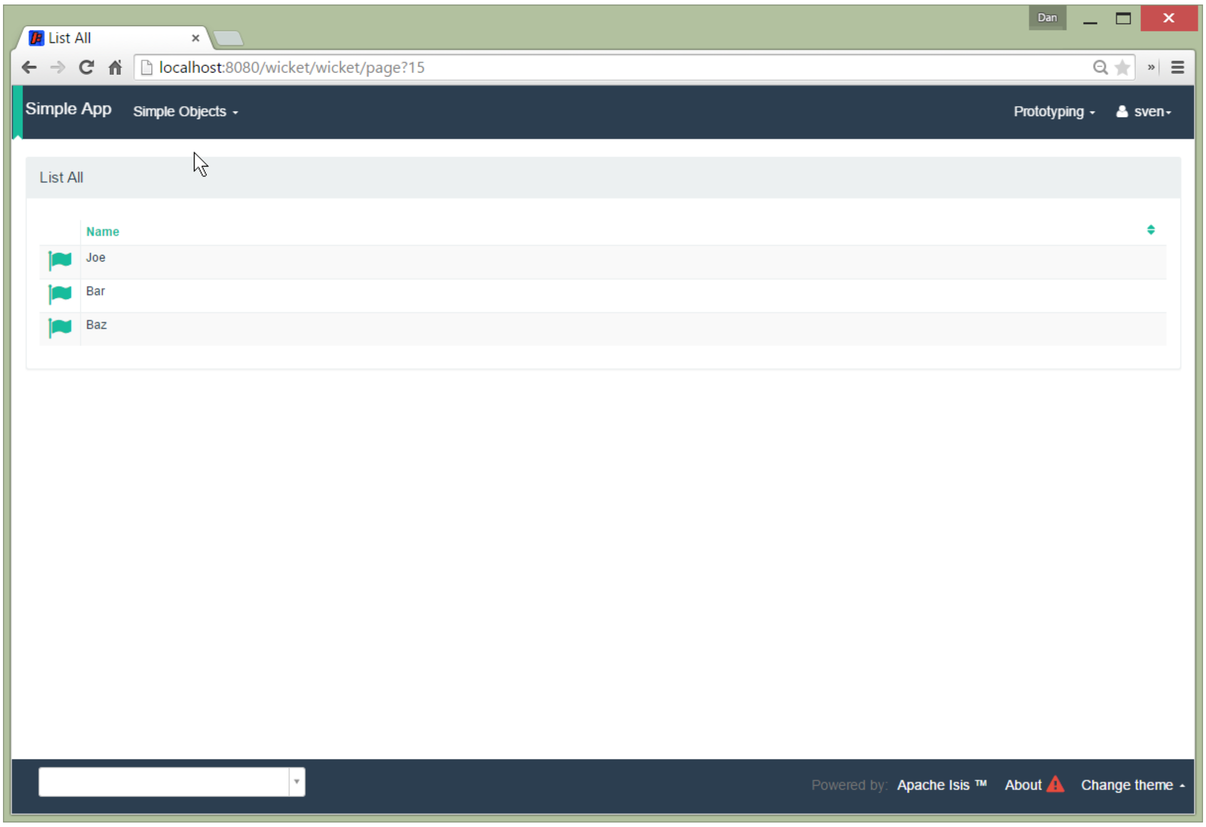
When you hit OK, the object is updated:



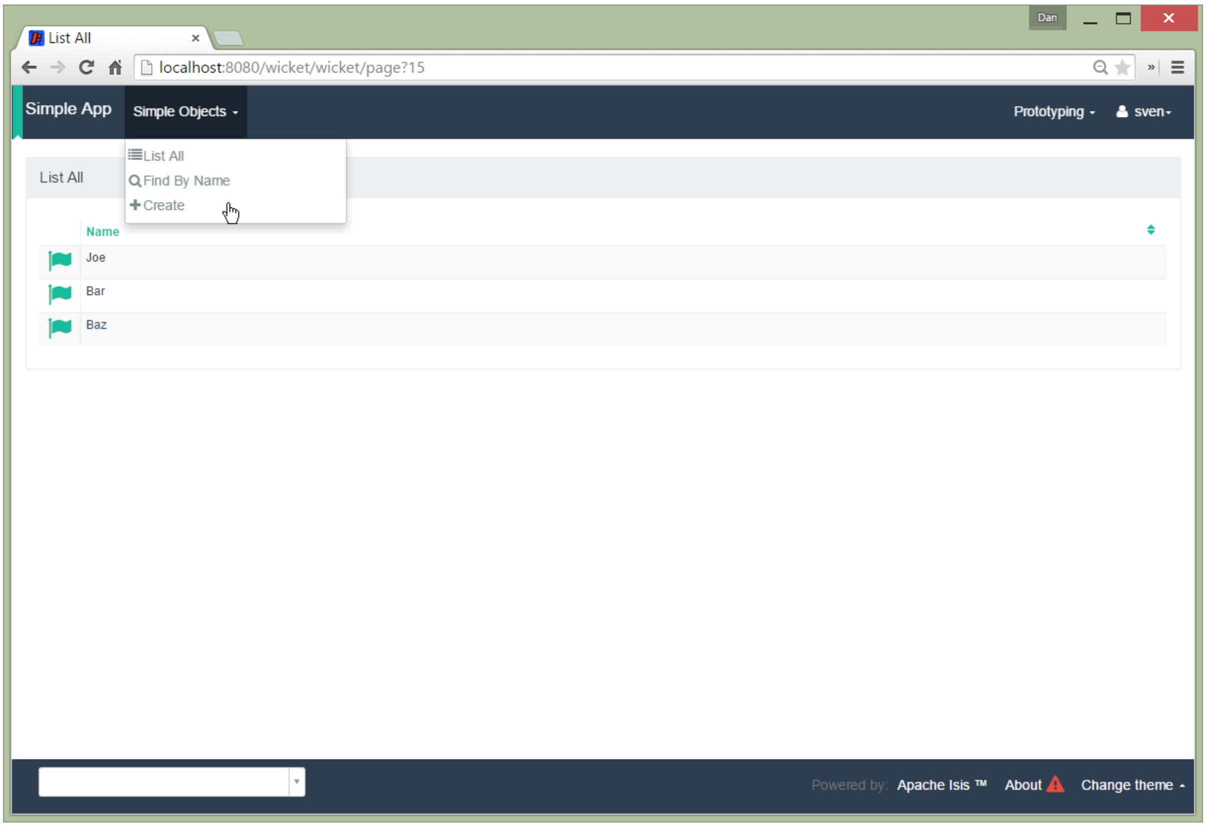
For your most signficant domain entities you'll likely have a domain service to retrieve or create instances of those obejcts. In the generated app we have a "Simple Objects" domain service that lets us list all objects:



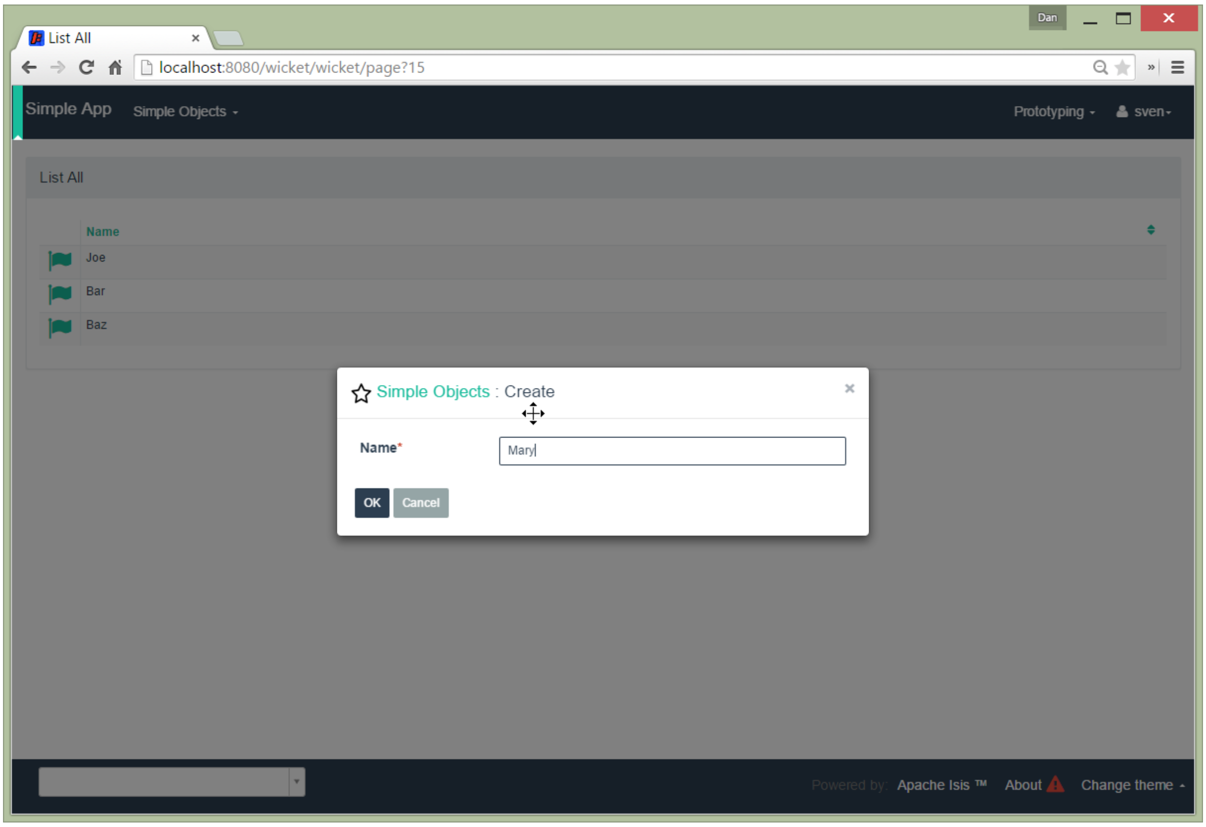
whereby we see the three objects created by the fixture script (one having been updated):



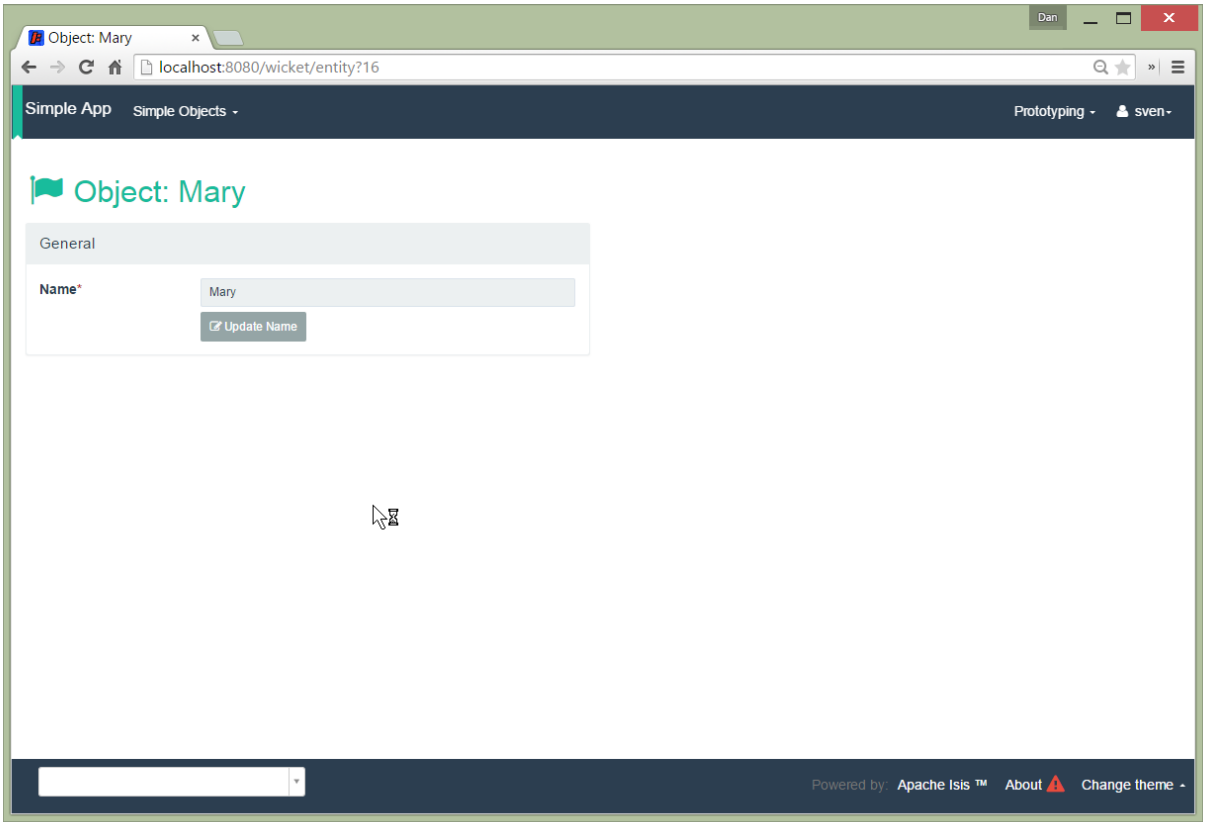
and we can also use the domain service to create new instances:



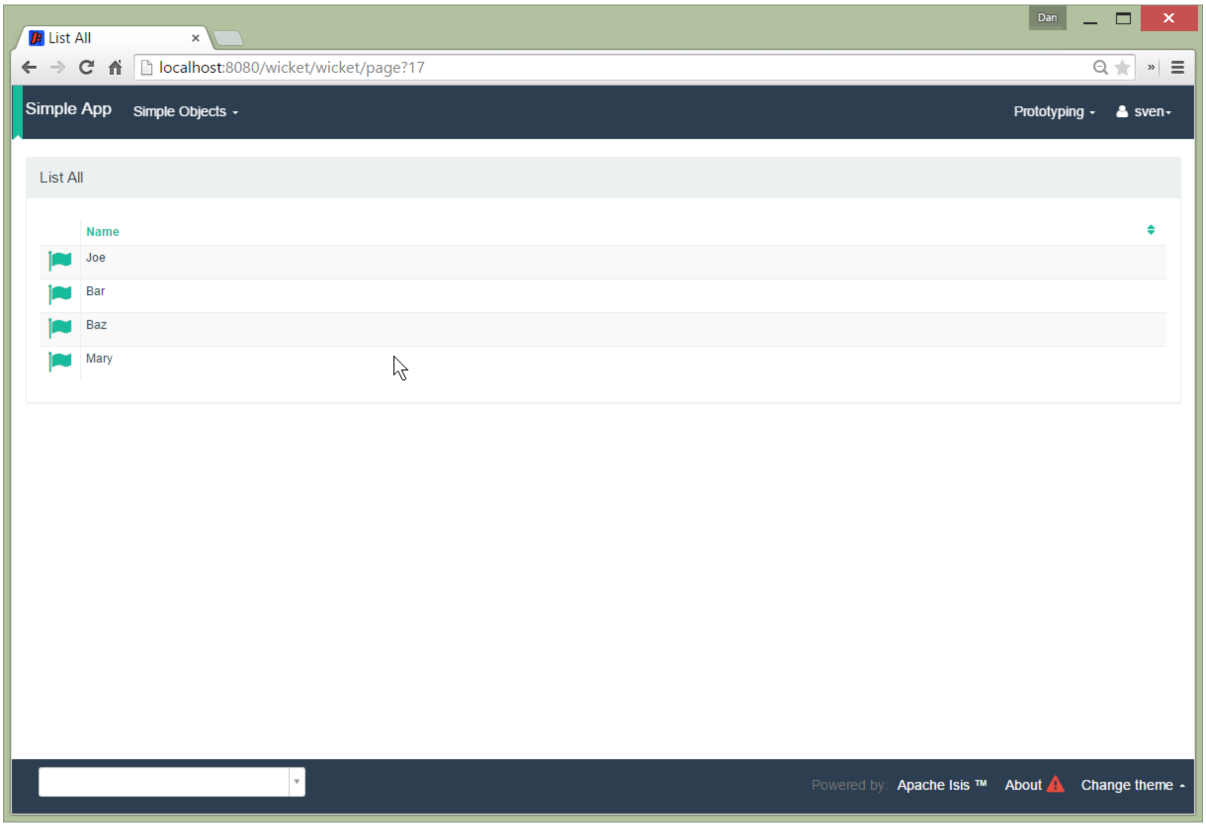
prompting us for the mandatory information (the name):



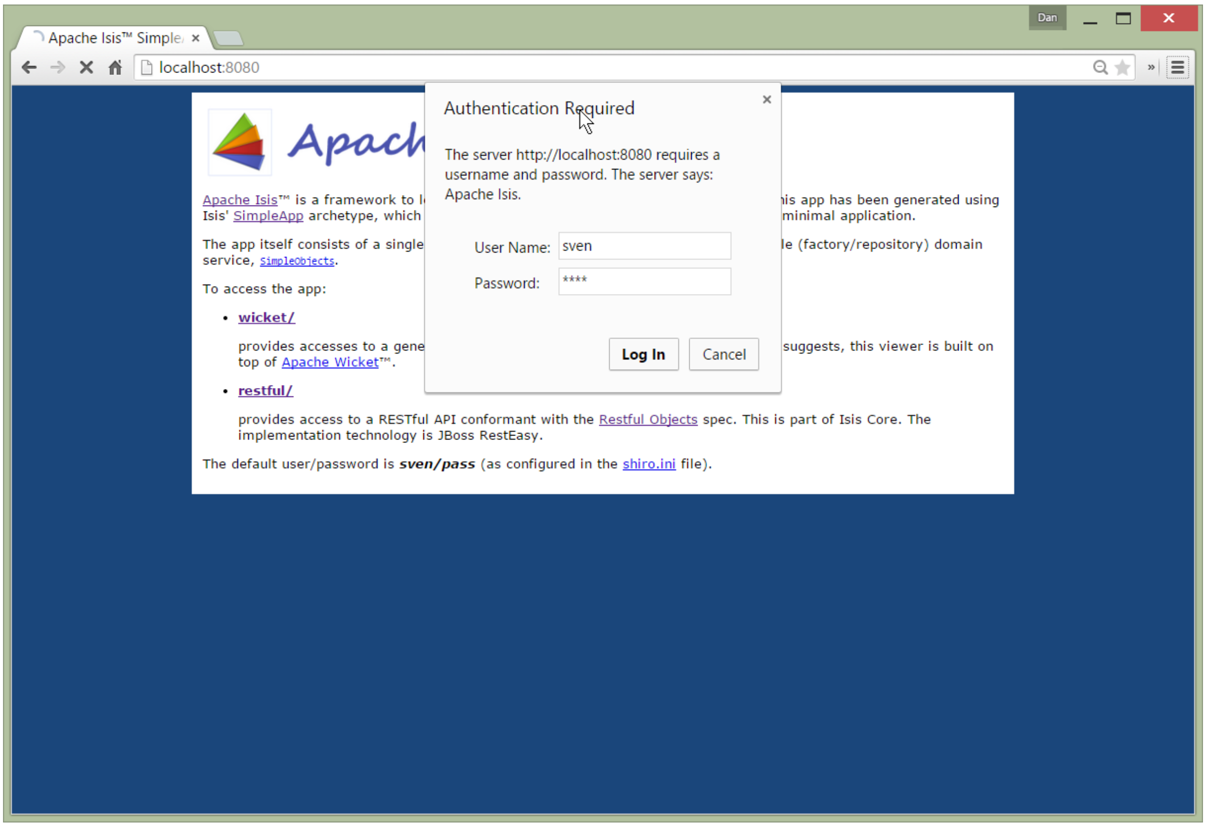
which, of course, returns the newly created object:



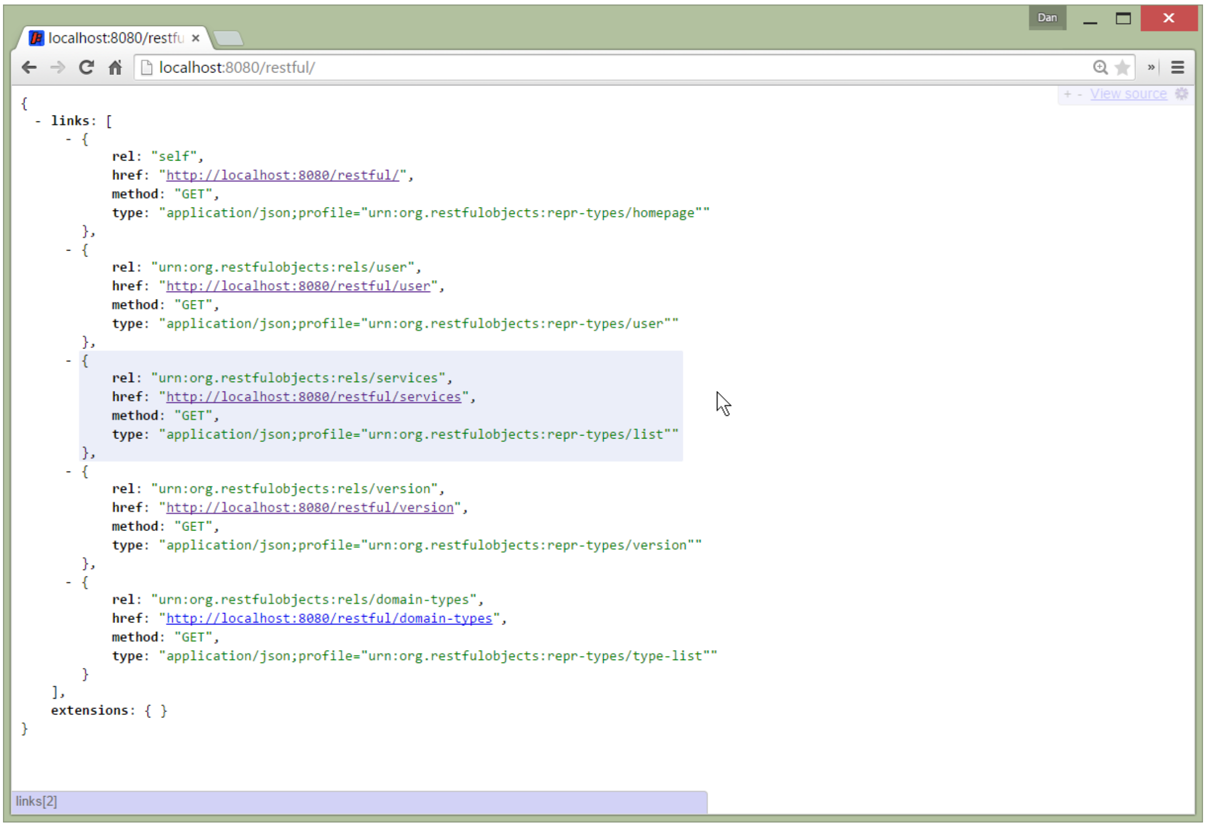
When we list all objects again, we can see that the object was indeed created:



Going back to the home page (localhost:8080) we can also access the Restful Objects viewer. The generated application is configured to use HTTP Basic Auth:

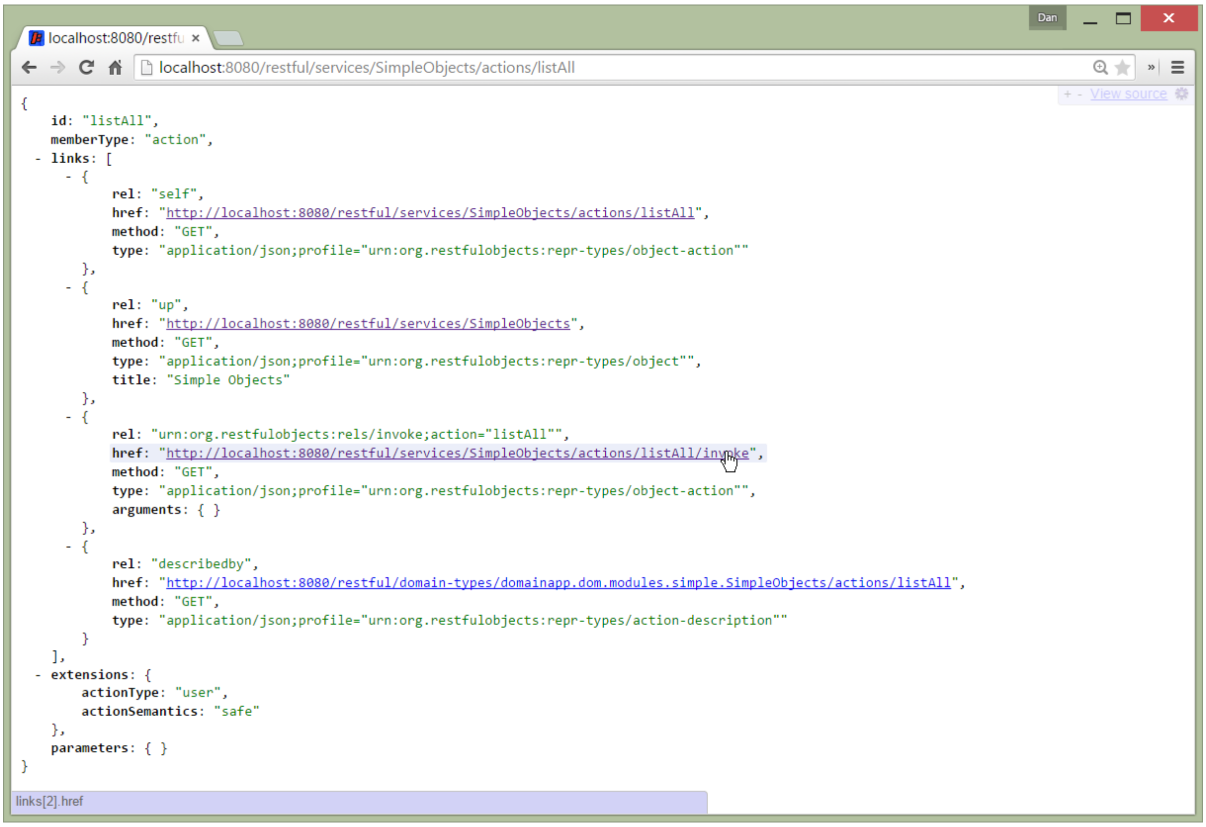


The Restful Objects viewer provides a REST API for computer-to-computer interaction, but we can still interact with it from a browser:

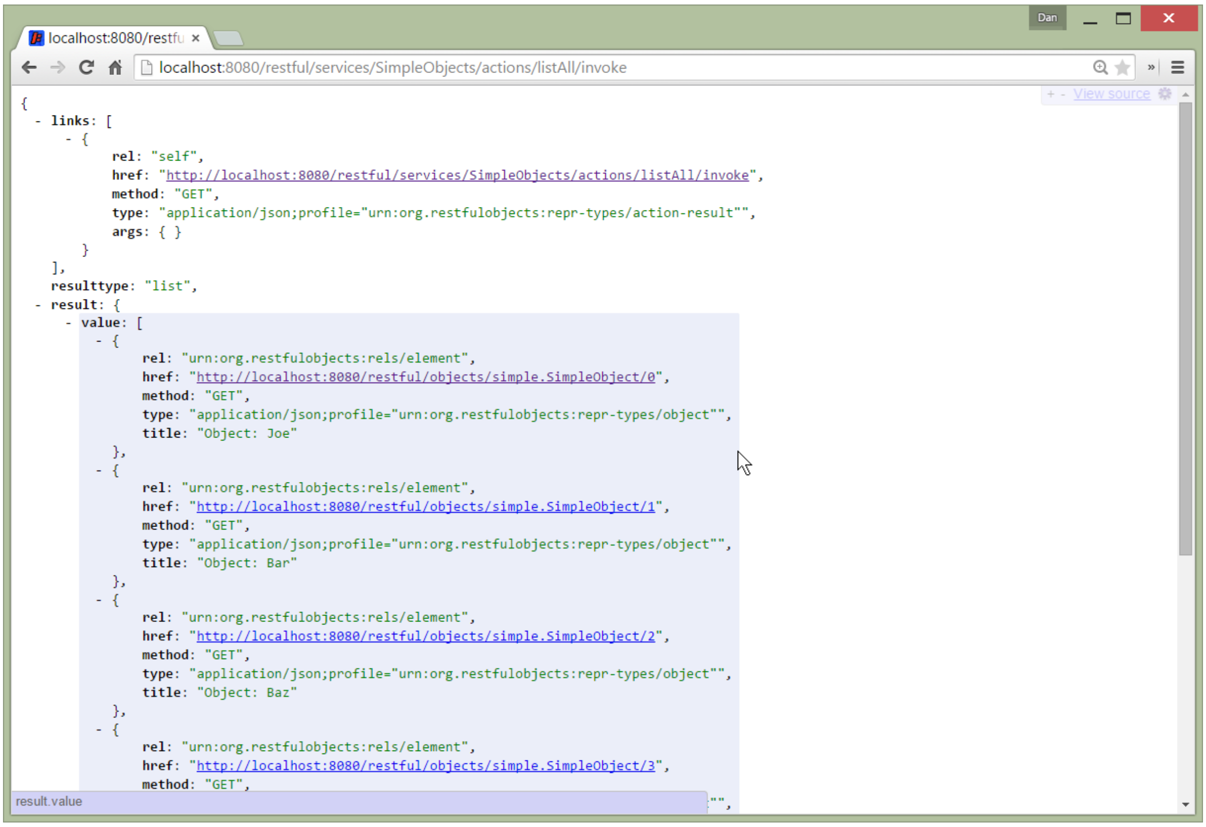


Depending on your browser, you may need to install plugins. For Chrome, we recommend json-view (which renders the JSON indented and automatically detects hyperlinks) and REST Postman.

The REST API is a complete hypermedia API, in other words you can follow the links to access all the behaviour exposed in the regular Wicket app. For example, we can navigate to the `listAll/invoke` resource:



which when invoked (with an HTTP GET) will return a representation of the domain objects.



To log in, use `sven/pass`.

3.2.6. Modifying the App

Once you are familiar with the generated app, you'll want to start modifying it. There is plenty of guidance on this site; check out the 'programming model how-tos' section on the main documentation page first).

If you use IntelliJ IDEA or Eclipse, do also install the live templates (for IntelliJ) / editor templates (for Eclipse); these will help you follow the Apache Isis naming conventions.

3.2.7. App Structure

As noted above, the generated app is a very simple application consisting of a single domain object that can be easily renamed and extended. The intention is not to showcase all of Apache Isis' capabilities; rather it is to allow you to very easily modify the generated application (eg rename `SimpleObject` to `Customer`) without having to waste time deleting lots of generated code.

Module	Description
`myapp`	The parent (aggregator) module
`myapp-app`	(1.9.0) The "app" module, containing the (optional) app manifest and any application-level services.
`myapp-dom`	The domain object model, consisting of `SimpleObject` and `SimpleObjects` (repository) domain service.
`myapp-fixture`	Domain object fixtures used for initializing the system when being demo'ed or for unit testing.
`myapp-integtests`	End-to-end integration tests that exercise from the UI through to the database
`myapp-webapp`	Run as a webapp (from `web.xml`) hosting the Wicket viewer and/or the RestfulObjects viewer

If you run into issues, please don't hesitate to ask for help on the users mailing list.

3.3. Datanucleus Enhancer

DataNucleus is the reference implementation of the JDO (Java data objects) spec, and Apache Isis integrates with DataNucleus as its persistence layer. Datanucleus is a very powerful library, allowing domain entities to be mapped not only to relational database tables, but also to NoSQL stores such as Neo4J, MongoDB and Apache Cassandra.

With such power comes a little bit of complexity to the development environment: all domain entities must be enhanced through the DataNucleus enhancer.

> Bytecode enhancement is actually a requirement of the JDO spec; the process is described in outline here.

What this means is that the enhancer — available as both a Maven plugin and as an Eclipse plugin — must, one way or another, be integrated into your development environment.

If working from the Maven command line, JDO enhancement is done using the `maven-datanucleus-plugin`. As of 1.9.0, we put all the configuration into an (always active) profile:

> The configuration described below is automatically set up by the SimpleApp archetype.

```xml
<profile>
    <id>enhance</id>
    <activation>
        <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
        <datanucleus-maven-plugin.version>4.0.1</datanucleus-maven-plugin.version>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.datanucleus</groupId>
                <artifactId>datanucleus-maven-plugin</artifactId>
                <version>${datanucleus-maven-plugin.version}</version>
                <configuration>
                    <fork>false</fork>
                    <log4jConfiguration>
${basedir}/log4j.properties</log4jConfiguration>
                    <verbose>true</verbose>
                    <props>${basedir}/datanucleus.properties</props>
                </configuration>
                <executions>
                    <execution>
                        <phase>process-classes</phase>
                        <goals>
                            <goal>enhance</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.datanucleus</groupId>
            <artifactId>datanucleus-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.datanucleus</groupId>
            <artifactId>datanucleus-jodatime</artifactId>
        </dependency>
        <dependency>
            <groupId>org.datanucleus</groupId>
            <artifactId>datanucleus-api-jdo</artifactId>
        </dependency>
    </dependencies>
</profile>
```

The SimpleApp archetype sets up the plugin correctly in the `dom` (domain object model) module. (It's actually a little bit more complex to cater for users of the Eclipse IDE using Eclipse's m2e plugin).

3.3.1. META-INF/persistence.xml

It's also a good idea to ensure that the `dom` module has a JDO `META-INF/persistence.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <persistence-unit name="simple"> ①
    </persistence-unit>
</persistence>
```

① change as required; typically is the name of the app.

Again, the SimpleApp archetype does this.

If running on Windows, then there's a good chance you'll hit the maximum path length limit. In this case the `persistence.xml` file is mandatory rather than optional.

This file is also required if you are using developing in Eclipse and relying on the DataNucleus plugin for Eclipse rather than the DataNucleus plugin for Maven. More information can be found here.

Chapter 4. How tos

This chapter provides a grab bag of "how-to"s and tips to help you go about actually developing Apache Isis domain applications.

4.1. Class Structure

Apache Isis works by building a metamodel of the domain objects: entities, view models and services. The class members of both entities and view models represent both state — (single-valued) properties and (multi-valued) collections — and behaviour — actions. The class members of domain services is simpler: just behaviour, ie actions.

In the automatically generated UI a property is rendered as a field. This can be either of a value type (a string, number, date, boolean etc) or can be a reference to another entity. A collection is generally rendered as a table.

In order for Apache Isis to build its metamodel the domain objects must follow some conventions: what we call the *Apache Isis Programming Model*. This is just an extension of the pojo / JavaBean standard of yesteryear: properties and collections are getters/setters, while actions are simply any remaining `public` methods.

Additional metamodel semantics are inferred both imperatively from *supporting methods* and declaratively from annotations.

In this section we discuss the mechanics of writing domain objects that comply with Apache Isis' programming model.

> In fact, the Apache Isis programming model is extensible; you can teach Apache Isis new programming conventions and you can remove existing ones; ultimately they amount to syntax. The only real fundamental that can't be changed is the notion that objects consist of properties, collections and actions.
>
> You can learn more about extending Apache Isis programming model here.

4.1.1. Class Definition

> TODO

Classes are defined both to Isis and (if an entity) also to JDO/DataNucleus.

We use Java packages as a way to group related domain objects together; the package name forms a namespace. We can then reason about all the classes in that package/namespace as a single unit.

In the same way that Java packages act as a namespace for domain objects, it's good practice to map domain entities to their own (database) schemas.

> For more on this topic, see the topic discussing modules and decoupling.

4.1.2. Property

TODO

Value vs Reference Types

TODO

The annotations for mapping value types tend to be different for properties vs action parameters, because JDO annotations are only valid on properties. The table in the Properties vs Parameters section provides a handy reference of each.

Optional Properties

JDO/DataNucleus' default is that a property is assumed to be mandatory if it is a primitive type (eg `int`, `boolean`), but optional if a reference type (eg `String`, `BigDecimal` etc). To override optionality in JDO/DataNucleus the `@Column(allowsNull="…")` annotations is used.

Apache Isis on the other hand assumes that all properties (and action parameters, for that matter) are mandatory, not optional. These defaults can also be overridden using Apache Isis' own annotations, specifically `@Property(optionality=…)`.

These different defaults can lead to incompatibilities between the two frameworks. To counteract that, Apache Isis also recognizes and honours JDO's `@Column(allowsNull=…)`.

For example, rather than:

```
@javax.jdo.annotations.Column(allowNulls="true")
private LocalDate date;
@Property(optionality=Optionality.OPTIONAL)
public LocalDate getDate() { ... }
public void setDate(LocalDate d) { ... }
```

you should instead simply write:

```
private LocalDate date;
@javax.jdo.annotations.Column(allowNulls="true")
public LocalDate getDate() { ... }
public void setDate(LocalDate d) { ... }
```

With JDO/DataNucleus it's valid for the `@Column` annotation to be placed on either the field or the getter. Apache Isis (currently) only looks for annotations on the getter. We therefore recommend that you always place `@Column` on the gettter.

In all cases the framework will search for any incompatibilities in optionality (whether specified explicitly or defaulted implicitly) between Isis' defaults and DataNucleus, and refuse to boot if any are found (fail fast).

Handling Mandatory Properties in Subtypes

If you have a hierarchy of classes then you need to decide which inheritance strategy to use.

- "table per hierarchy", or "rollup" (`InheritanceStrategy.SUPERCLASS_TABLE`)

  whereby a single table corresponds to the superclass, and also holds the properties of the subtype (or subtypes) being rolled up

- "table per class" (`InheritanceStrategy.NEW_TABLE`)

  whereby is a table for both superclass and subclass, in 1:1 correspondence

- "rolldown" (`InheritanceStrategy.SUBCLASS_TABLE`)

  whereby a single table holds the properties of the subtype, and also holds the properties of its supertype

In the first "rollup" case, we can have a situation where - logically speaking - the property is mandatory in the subtype - but it must be mapped as nullable in the database because it is n/a for any other subtypes that are rolled up.

In this situation we must tell JDO that the column is optional, but to Apache Isis we want to enforce it being mandatory. This can be done using the `@Property(optionality=Optionality.MANDATORY)` annotation.

For example:

```
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.SUPER_TABLE)
public class SomeSubtype extends SomeSuperType {
    private LocalDate date;
    @javax.jdo.annotations.Column(allowNulls="true")
    @Property(optionality=Optionality.MANDATORY)
    public LocalDate getDate() { ... }
    public void setDate(LocalDate d) { ... }
}
```

The `@Property(optionality=…)` annotation is equivalent to the older but still supported `@Optional` annotation and `@Mandatory` annotations. Its benefit is that it lumps together all Apache Isis' property metadata in a single annotation. Its downside is that it is rather verbose if the only semantic that needs to be specified — as is often the case — is optionality.

An alternative way to achieve this is to leave the JDO annotation on the field (where it is invisible to Apache Isis), and rely on Isis' default, eg:

```java
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.SUPER_TABLE)
public class SomeSubtype extends SomeSuperType {
    @javax.jdo.annotations.Column(allowNulls="true")
    private LocalDate date;
    // mandatory in Apache Isis by default
    public LocalDate getDate() { }
    public void setDate(LocalDate d) { }
}
```

We recommend the former mapping, though, using `@Property(optionality=Optionality.MANDATORY)`.

`String`s (Length)

TODO

Mapping JODA Dates

Isis' JDO objectstore bundles DataNucleus' built-in support for Joda `LocalDate` and `LocalDateTime` datatypes, meaning that entity properties of these types will be persisted as appropriate data types in the database tables.

It is, however, necessary to annotate your properties with `@javax.jdo.annotations.Persistent`, otherwise the data won't actually be persisted. See the JDO docs for more details on this.

Moreover, these datatypes are *not* in the default fetch group, meaning that JDO/DataNucleus will perform an additional `SELECT` query for each attribute. To avoid this extra query, the annotation should indicate that the property is in the default fetch group.

For example, the `ToDoItem` (in the todoapp example app (not ASF)) defines the `dueBy` property as follows:

`BigDecimal`s (Precision)

Working with `java.math.BigDecimal` properties takes a little care due to scale/precision issues.

For example, suppose we have:

```java
private BigDecimal impact;
public BigDecimal getImpact() {
    return impact;
}
public void setImpact(final BigDecimal impact) {
    this.impact = impact;
}
```

JDO/DataNucleus creates, at least with HSQL, the table with the field type as NUMERIC(19). No decimal digits are admitted. (Further details here).

What this implies is that, when a record is inserted, a log entry similar to this one appears:

```
INSERT INTO ENTITY(..., IMPACT, ....) VALUES (...., 0.5, ....)
```

But when that same record is retrieved, the log will show that a value of "0" is returned, instead of 0.5.

The solution is to explicitly add the scale to the field like this:

```
@javax.jdo.annotations.Column(scale=2)
private BigDecimal impact;
public BigDecimal getImpact() {
    return impact;
}
public void setImpact(final BigDecimal impact) {
    this.impact = impact;
}
```

In addition, you should also set the scale of the `BigDecimal`, using `setScale(scale, roundingMode)`.

More information can be found here and here.

Mapping Blobs and Clobs

Apache Isis configures JDO/DataNucleus so that the properties of type `org.apache.isis.applib.value.Blob` and `org.apache.isis.applib.value.Clob` can also be persisted.

As for Joda dates, this requires the `@javax.jdo.annotations.Persistent` annotation. However, whereas for dates one would always expect this value to be retrieved eagerly, for blobs and clobs it is not so clear cut.

Mapping Blobs

For example, in the `ToDoItem` class (of the todoapp example app (non-ASF) the `attachment` property is as follows:

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

The three `@javax.jdo.annotations.Column` annotations are required because the mapping classes that Apache Isis provides (IsisBlobMapping and IsisClobMapping) map to 3 columns. (It is not an error to omit these `@Column` annotations, but without them the names of the table columns are simply suffixed `_0`, `_1`, `_2` etc.

If the `Blob` is mandatory, then use:

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

If specifying a `sqlType` of "LONGVARBINARY" does not work, try instead "BLOB". There can be differences in behaviour between JDBC drivers.

Mapping Clobs

Mapping `Clob`s works in a very similar way, but the `jdbcType` and `sqlType` attributes will, respectively, be CLOB and LONGVARCHAR:

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

If specifying a sqlType of "LONGVARCHAR" does not work, try instead "CLOB". There can be differences in behaviour between JDBC drivers.

Mapping to VARBINARY or VARCHAR

Instead of mapping to a sqlType of LONGVARBINARY (or perhaps BLOB), you might instead decide to map to a VARBINARY. The difference is whether the binary data is held "on-row" or as a pointer "off-row"; with a VARBINARY the data is held on-row and so you will need to specify a length.

For example:

```
@javax.jdo.annotations.Column(name = "attachment_bytes", jdbcTypr="BLOB", sqlType =
"VARBINARY", length=2048)
```

The same argument applies to LONGVARCHAR (or CLOB); you could instead map to a regular VARCHAR:

```
@javax.jdo.annotations.Column(name = "attachment_chars", sqlType = "VARCHAR", length
=2048)
```

Support and maximum allowed length will vary by database vendor.

4.1.3. Collections

TODO

While Apache Isis support collections of references, the framework (currently) does not support collections of values. That is, it isn't possible to define a collection of type `Set<String>`.

Or, actually, you can, because that is a valid mapping supported by JDO/DataNucleus . However, Apache Isis has no default visualization.

One workaround is to mark the collection as `@Programmatic`. This ensures that the collection is ignored by Apache Isis.

Another workaround is to wrap each value in a view model, as explained in this tip.

4.1.4. Actions

TODO

While Apache Isis support actions whose parameters' types are scalar (values such as `String`, `int`, or references such as `Customer`), the framework (currently) does not support parameter types that are collections or maps.

The workaround is to mark the collection as `@Programmatic`, as described in Ignoring Methods. This ensures that the collection is ignored by Apache Isis.

4.1.5. Action Parameters

TODO

`String`s (Length)

TODO

`BigDecimal`s (Precision)

TODO

Optional

TODO

4.1.6. Injecting services

Apache Isis autowires (automatically injects) domain services into each entity, as well as into the domain services themselves, using either method injection or field injection. The applib `DomainObjectContainer` is also a service, so can be injected in exactly the same manner.

> Isis currently does *not* support qualified injection of services; the domain service of each type must be distinct from any other.
>
> If you find a requirement to inject two instances of type `SomeService`, say, then the work-around is to create trivial subclasses `SomeServiceA` and `SomeServiceB` and inject these instead.

Field Injection

Field injection is recommended, using the `@javax.inject.Inject` annotation. For example:

```
public class Customer {
    @javax.inject.Inject
    OrderRepository orderRepository;
    ...
}
```

We recommend using default rather than `private` visibility so that the field can be mocked out within unit tests (placed in the same package as the code under test).

Method Injection

Isis also supports two forms of method injection. All that is required to inject a service into a entity/service is to provide an appropriate method or field. The name of the method does not matter, only that it is prefixed either `set` or `inject`, is public, and has a single parameter of the correct type.

For example:

```
public class Customer {
    private OrderRepository orderRepository;
    public void setOrderRepository(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }
    ...
}
```

or alternatively, using 'inject' as the prefix:

```
public class Customer {
    private OrderRepository orderRepository;
    public void injectOrderRepository(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }
    ...
}
```

Note that the method name can be anything; it doesn't need to be related to the type being injected.

Constructor injection

Simply to note that constructor injection is *not* supported by Apache Isis (and is unlikely to be, because the JDO specification for entities requires a no-arg constructor).

4.1.7. Properties vs Parameters

TODO

In many cases the value types of properties and of action parameters align. For example, a `Customer` entity might have a `surname` property, and there might also be corresponding `changeSurname`. Ideally we want the surname property and surname action parameter to use the same value type.

Having first-class support for this is a goal of Apache Isis, and it is already possible (by writing some plumbing and glue code using Isis and DataNucleus APIs) to get some way towards this ideal. However, in the vast majority of cases it is much easier and more practical to simply use standard value types such as `java.lang.String`.

However, the JDO/DataNucleus annotations for specifying semantics such as optionality or maximum length are restricted to only apply to fields and to methods; they cannot be applied to action parameters. It is therefore necessary to use Apache Isis' equivalent annotations for action parameters.

The table below summarises the equivalence of some of the most common cases.

Table 1. Comparing annotations of Properties vs Action Parameters

value type/semantic	property	action parameter
string (length)		
big decimal (precision)		
Isis blob		
optional	`@Column(allowsNull="true")`	`ParameterLayout(optionality=Optionality.OPTIONAL)` or `@Optional`

4.1.8. Ignoring Methods

TODO

Sometimes you want to define a `public` method on a domain object that is not intended to be rendered in Apache Isis' UI.

To exclude such methods, use the `@Programmatic` annotation.

4.2. UI Hints

The Apache Isis programming model includes several mechanisms for a domain object to provide UI hints. These range from their title (so an end-user can distinguish one object from another) through to hints that can impact their CSS styling.

4.2.1. Object Titles and Icons

In Apache Isis every object is identified to the user by a title (label) and an icon. This is shown in several places: as the main heading for an object; as a link text for an object referencing another object, and also in tables representing collections of objects.

The icon is often the same for all instances of a particular class, but it's also possible for an individual instance to return a custom icon. This could represent the state of that object (eg a shipped order, say, or overdue library book).

It is also possible for an object to provide a CSS class hint. In conjunction with customized CSS this can be used to apply arbitrary styling; for example each object could be rendered in a page with a different background colour.

Object Title

The object title is a label to identify an object to the end-user. Generally the object title is a label to identify an object to the end-user. There is no requirement for it to be absolutely unique, but it should be "unique enough" to distinguish the object from other object's likely to be rendered on the same page.

The title is always shown with an icon, so there is generally no need for the title to include information about the object's type. For example the title of a customer object shouldn't include the literal string "Customer"; it can just have the customer's name, reference or some other meaningful business identifier.

Declarative style

The `@Title` annotation can be used build up the title of an object from its constituent parts.

For example:

```
public class Customer {
    @Title(sequence="1", append=" ")
    public String getFirstName() { ... }
    @Title(sequence="2")
    public Product getLastName() { ... }
    ...
}
```

might return "Arthur Clarke", while:

```
public class CustomerAlt {
    @Title(sequence="2", prepend=", ")
    public String getFirstName() { ... }

    @Title(sequence="1")
    public Product getLastName() { ... }
    ...
}
```

could return "Clarke, Arthur".

Note that the sequence is in Dewey Decimal Format. This allows a subclass to intersperse information within the title. For example (please forgive this horrible domain modelling (!)):

```
public class Author extends Customer {
    @Title(sequence="1.5", append=". ")
    public String getMiddleInitial() { ... }
    ...
}
```

could return "Arthur C. Clarke".

> Titles can sometimes get be long and therefore rather cumbersome in "parented" tables. If `@Title` has been used then the Wicket viewer will automatically exclude portions of the title belonging to the owning object.

Imperative style

> TODO - see `title()`

Object Icon

The icon is often the same for all instances of a particular class, but it's also possible for an individual instance to return a custom icon. This could represent the state of that object (eg a shipped order, say, or overdue library book).

> TODO - `iconName()`, `@DomainObjectLayout#cssClassFa()`

Object CSS Styling

It is also possible for an object to return a CSS class. In conjunction with customized CSS this can be used to apply arbitrary styling; for example each object could be rendered in a page with a different background colour.

> TODO - `cssClass()`, `@DomainObjectLayout#cssClass()` `@ActionLayout#cssClass()`, `@PropertyLayout#cssClass()`, `@CollectionLayout#cssClass()`

4.2.2. Names and Descriptions

TODO

see also Internationalization

Class (object)

TODO - `@DomainObjectLayout#named()`, `@DomainObjectLayout#describedAs()`

Property

TODO - `@PropertyLayout#named()`, `@PropertyLayout#describedAs()`

Collections

TODO - `@CollectionLayout#named()`, `@CollectionLayout#describedAs()`

Actions

TODO - `@ActionLayout#named()`, `@ActionLayout#describedAs()`

Action Parameters

TODO - `@ParameterLayout#named()`, `@ParameterLayout#describedAs()`

If you're running on Java 8, then note that it's possible to write Isis applications without using `@ParameterLayout(named=…)` annotation. Support for this can be found in the Isis addons' paraname8 metamodel extension (non-ASF). (In the future we'll fold this into core). See also our guidance on upgrading to Java 8.

4.2.3. Layout

See the object layout chapter.

4.2.4. Eager rendering

By default, collections all rendered lazily, in other words in a "collapsed" table view:

TODO - screenshot here

For the more commonly used collections we want to show the table expanded:

TODO - screenshot here

For this we annotate the collection using the `@CollectionLayout(render=RenderType.EAGERLY`; for example

```
@javax.jdo.annotations.Persistent(table="ToDoItemDependencies")
private Set<ToDoItem> dependencies = new TreeSet<>();
@Collection
@CollectionLayout(
    render = RenderType.EAGERLY
)
public Set<ToDoItem> getDependencies() {
    return dependencies;
}
```

Alternatively, it can be specified the `Xxx.layout.json` file:

```
"dependencies": {
    "collectionLayout": {
        "render": "EAGERLY"
    },
}
```

It might be thought that collections that are eagerly rendered should also be eagerly loaded from the database by enabling the `defaultFetchGroup` attribute:

```
@javax.jdo.annotations.Persistent(table="ToDoItemDependencies",
defaultFetchGroup="true")
private Set<ToDoItem> dependencies = new TreeSet<>();
...
```

While this can be done, it's likely to be a bad idea, because doing so will cause DataNucleus to query for more data than required even if the object is being rendered within some referencing object's table.

Of course, your mileage may vary, so don't think you can't experiment.

4.2.5. Action Icons and CSS

Apache Isis allows font awesome icons to be associated with each action, and for Bootstrap CSS to be applied to action rendered as buttons.

These UI hint can be applied either to individual actions, or can be applied en-masse using pattern matching.

Per action

TODO - `@ActionLayout#cssClass()` and `@ActionLayout#cssClassFa()`

Alternatively, you can specify these hints dynamically in the `Xxx.layout.json` for the entity.

Per pattern matching

Rather than annotating every action with `@ActionLayout#cssClassFa()` and `@ActionLayout#cssClass()` you can instead specify the UI hint globally using regular expressions.

The configuration property `isis.reflector.facet.cssClassFa.patterns` is a comma separated list of key:value pairs, eg:

```
isis.reflector.facet.cssClassFa.patterns=\
                        new.*:fa-plus,\
                        add.*:fa-plus-square,\
                        create.*:fa-plus,\
                        list.*:fa-list, \
                        all.*:fa-list, \
                        download.*:fa-download, \
                        upload.*:fa-upload, \
                        execute.*:fa-bolt, \
                        run.*:fa-bolt
```

where the key is a regex matching action names (eg `create.*`) and the value is a font-awesome icon name (eg `fa-plus`) to be applied (as per `@CssClassFa()`) to all action members matching the regex.

Similarly, the configuration property `isis.reflector.facet.cssClass.patterns` is a comma separated list of key:value pairs, eg:

```
isis.reflector.facet.cssClass.patterns=\
                        delete.*:btn-warning
```

where (again)the key is a regex matching action names (eg `delete.*`) and the value is a Bootstrap CSS button class (eg `btn-warning) to be applied (as per `@CssClass()`) to all action members matching the regex.

> We strongly recommend that you use this technique rather than annotating each action with `@ActionLayout#cssClassFa()` or `@ActionLayout#cssClass()`. Not only is the code more maintainable, you'll also find that it forces you to be consistent in your action names.

4.2.6. 'Are you sure?' idiom

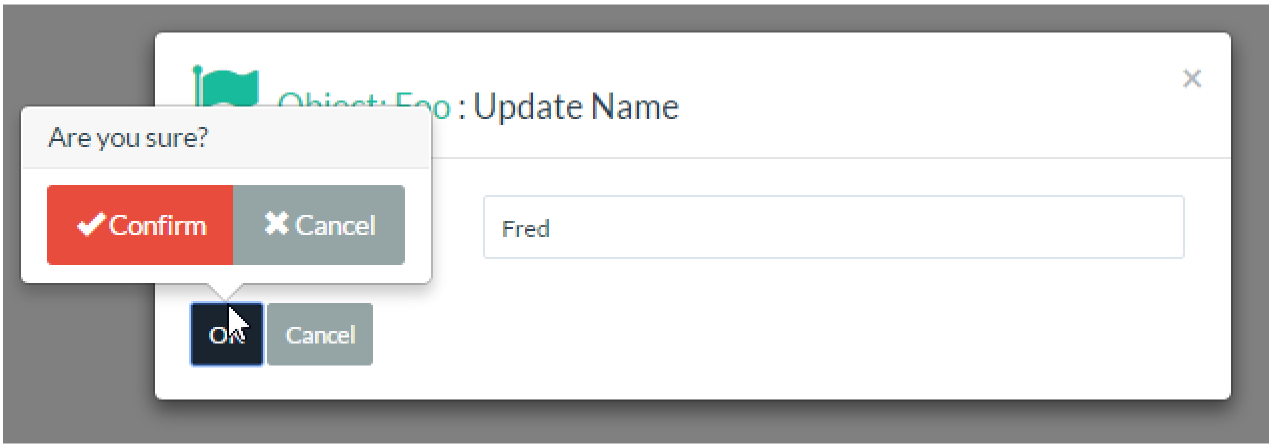
Sometimes an action might perform irreversible changes. In such a case it's probably a good idea for the UI to require that the end-user explicitly confirms that they intended to invoke the action.

Using action semantics

One way to meet this requirement is using the framework's built-in `@Action#semantics()` attribute:

```
@Action(
        semantics = SemanticsOf.IDEMPOTENT_ARE_YOU_SURE
)
public SimpleObject updateName(
        @Parameter(maxLength = NAME_LENGTH)
        @ParameterLayout(named = "New name")
        final String name) {
    setName(name);
    return this;
}
```

This will render as:



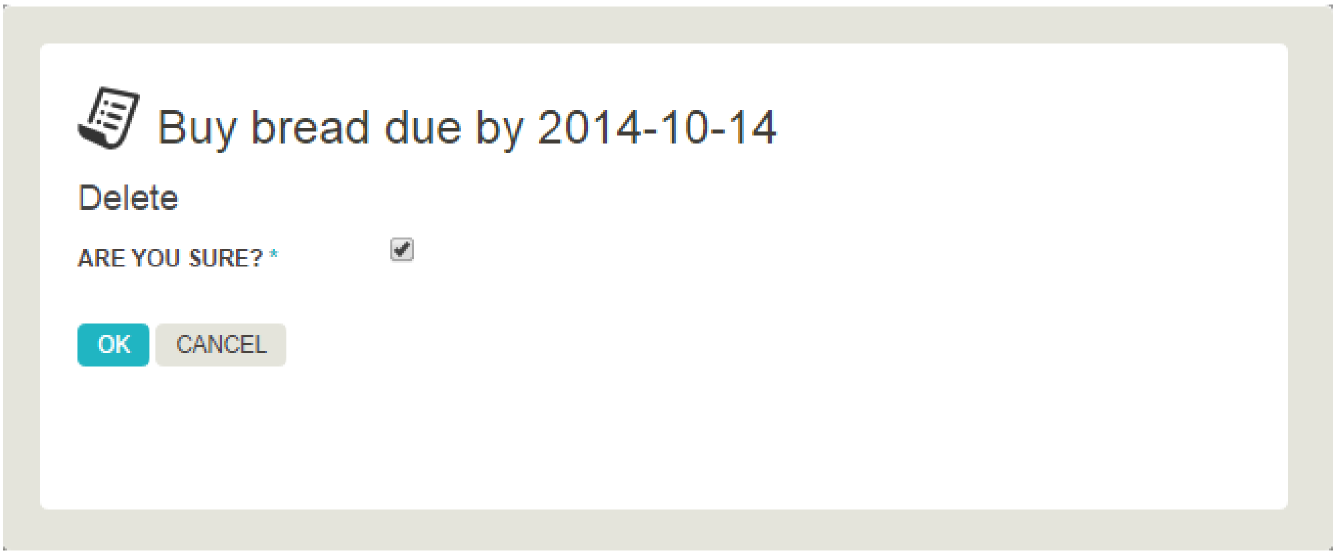
Using a checkbox

An alternative approach (for all versions of the framework) is to require the end-user to check a dummy checkbox parameter (and prevent the action from being invoked if the user hasn't checked that parameter).

For example:

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

If the user checks the box:



then the action will complete.

However, if the user fails to check the box, then a validation message is shown:

The code for this is pretty simple:

```
public List<ToDoItem> delete(@Named("Are you sure?") boolean areYouSure) {
    container.removeIfNotAlready(this);
    container.informUser("Deleted " + container.titleOf(this));
    return toDoItems.notYetComplete();          ①
}
public String validateDelete(boolean areYouSure) {
    return areYouSure? null: "Please confirm you are sure";
}
```

① invalid to return `this` (cannot render a deleted object)

Note that the action itself does not use the boolean parameter, it is only used by the supporting `validate…()` method.

4.3. Domain Services

In Apache Isis domain services have several responsibilities:

- to expose actions to be rendered in the menu
- to provide actions that are rendered as contributed actions/properties/collections on the contributee domain object
- they act as subscribers to the event bus
- they act as repositories (find existing objects) or as factories (create new objects)
- they provide other services (eg performing calculations, attach a barcode, send an email etc).
- to implement an SPI of the framework, most notably cross-cutting concerns such as security, command profiling, auditing and publishing.

It's worth extending the Hexagonal Architecture to show where domain services — and in particular the domain services provided by Isis Addons (non-ASF) — fit in:

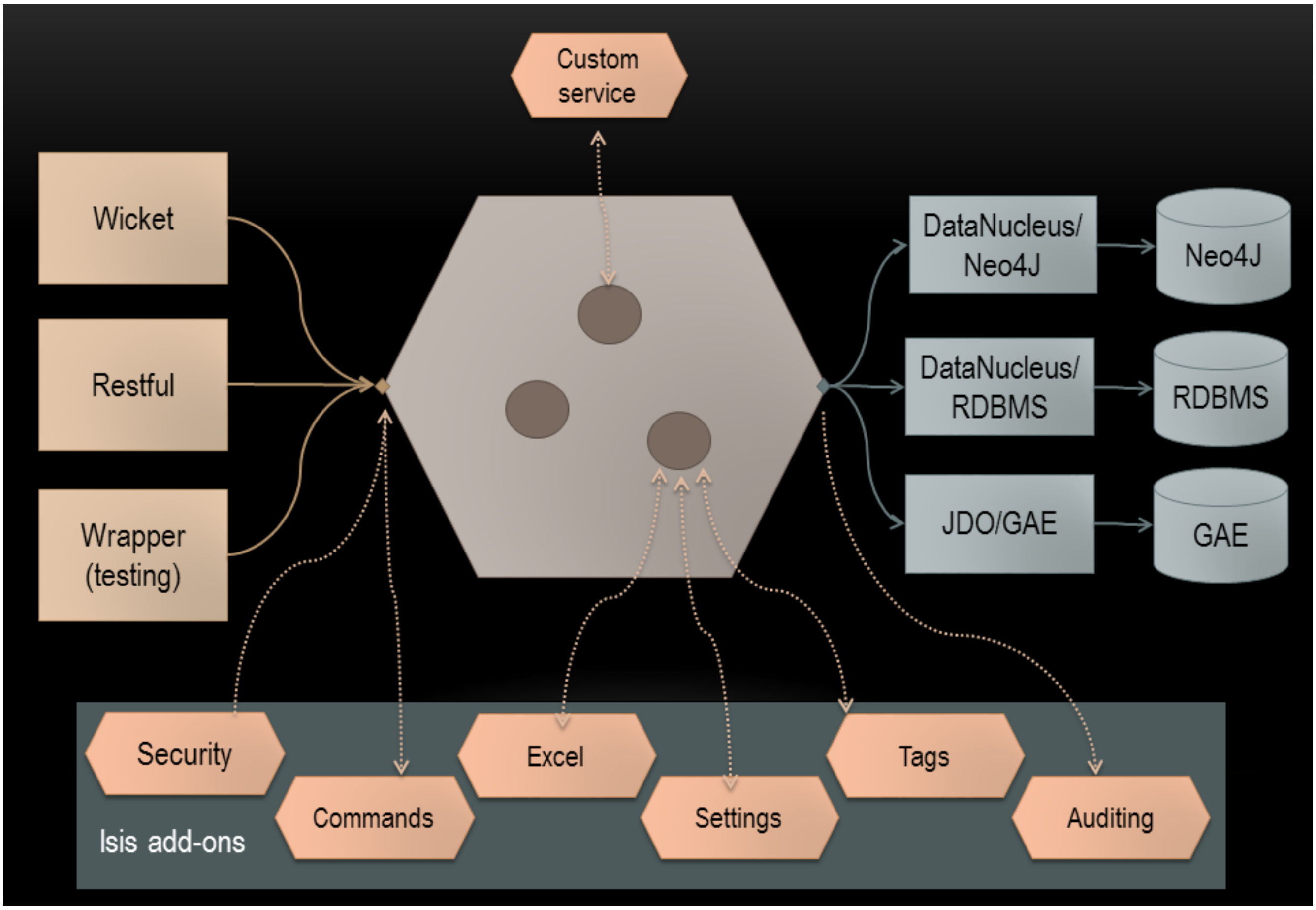


Figure 2. The hexagonal architecture with Isis addons

The (non-ASF) Isis Addons are a good source of domain services, providing SPI implementations of the common cross-cutting concerns, and also a number of APIs for domain objects to invoke (eg tags, excel, settings). Of course, you can also write your own domain services as well, for example to interface with some external CMS system, say.

The Apache Isis framework also provides numerous in-built domain services. These are catalogued in the domain services reference guide.

4.3.1. Organizing Services

In larger applications we have found it worthwhile to ensure that our domain services only act aligned with these responsibilities, employing a naming convention so that it is clear what the responsibilities of each domain service is.

The application provides the `@DomainService(nature=…)` annotation that helps distinguish some of these responsibilities:

- `VIEW` indicates that the actions should appear both on the menu and also be used as contributions
- `VIEW_MENU_ONLY` indicates that the actions should appear on the menu
- `VIEW_CONTRIBUTED_ONLY` indicates that the actions should appear on the menu
- `DOMAIN` indicates that the actions are for other domain objects to invoke (either directly or indirectly through the event bus), but in any case should not be rendered at all in the UI

Pulling all the above together, here are our suggestions as to how you should organize your domain services.

Factory and Repository

The factory/repository uses an injected `DomainObjectContainer` to both instantiate new objects and to query the database for existing objects of a given entity type. It is not visible in UI, rather other services delegate to it.

We suggest naming such classes `XxxRepository`, eg:

```
@DomainService(
    nature=NatureOfService.DOMAIN                                   ①
)
public CustomerRepository {
    public List<Customer> findCustomerBy...(...) {
        return allMatches(...);
    }
    public Customer newCustomer(...) {
        Customer Customer = container.newTransientInstance(Customer.class);
        ...
        persistIfNotAlready(Customer);
        return Customer;
    }
    public List<Customer> allCustomers() {
        return container.allInstances(Customer.class);
    }
    @Inject
    DomainObjectContainer container;
}
```

① interacted with only programmatically by other objects in the domain layer.

There is no need to annotate the actions; they are implicitly hidden because of the domain service's nature.

Menu

Menu services provide actions to be rendered on the menu.

For the Wicket viewer, each service's actions appear as a collection of menu items of a named menu, and this menu is on one of the three menu bars provided by the Wicket viewer. It is possible for more than one menu service's actions to appear on the same menu; a separator is shown between each.

For the Restful Objects viewer, all menu services are shown in the services representation.

We suggest naming such classes `XxxMenu`, eg:

```
@DomainService(
    nature = NatureOfService.VIEW_MENU_ONLY                         ①
)
@DomainServiceLayout(
        named = "Customers",                                        ②
        menuBar = DomainServiceLayout.MenuBar.PRIMARY,
        menuOrder = "10"
)
public class CustomerMenu {
    @Action(
            semantics = SemanticsOf.SAFE
    )
    @MemberOrder( sequence = "1" )
    public List<Customer> findCustomerBy...(...) {
        return CustomerRepository.findCustomerBy(...);              ③
    }

    @Action(
            semantics = SemanticsOf.NON_IDEMPOTENT
    )
    @MemberOrder( sequence = "3" )
    public Customer newCustomer(...) {
        return CustomerRepository.newCustomer(...);
    }

    @Action(
            semantics = SemanticsOf.SAFE,
            restrictTo = RestrictTo.PROTOTYPING
    )
    @MemberOrder( sequence = "99" )
    public List<Customer> allCustomers() {
        return CustomerRepository.allBankMandates();
    }

    @Inject
    protected CustomerRepository customerRepository;
}
```

① the service's actions should be rendered as menu items

② specifies the menu name. All services with the same menu name will be displayed on the same menu, with separators between

③ delegates to an injected repository.

Not every action on the repository need to be delegated to of course (the above example does but only because it is very simple).

Note also that while there's nothing to stop `VIEW_MENU` domain services being injected into other domain objects and interacted with programmatically, we recommend against it. Instead, inject the underlying repository. If there is additional business logic, then consider introducing a further `DOMAIN`-scoped service and call that instead.

Contributions

Services can contribute either actions, properties or collections, based on the type of their parameters.

We suggest naming such classes `XxxContributions`, eg:

```
@DomainService(
    nature=NatureOfService.VIEW_CONTRIBUTIONS_ONLY                ①
)
@DomainServiceLayout(
    menuOrder="10",
    name="...",
}
public OrderContributions {
    @Action(semantics=SemanticsOf.SAFE)
    @ActionLayout(contributed=Contributed.AS_ASSOCIATION)         ②
    @CollectionLayout(render=RenderType.EAGERLY)
    public List<Order> orders(Customer customer) {                ③
        return container.allMatches(...);
    }

    @Inject
    CustomerRepository customerRepository;
}
```

① the service's actions should be contributed to the entities of the parameters of those actions

② contributed as an association, in particular as a collection because returns a `List<T>`.

③ Only actions with a single argument can be contributed as associations

More information about contributions can be found here. More information about using contributions and mixins to keep your domain application decoupled can be found here and here.

Event Subscribers

Event subscribers can both veto interactions (hiding members, disabling members or validating changes), or can react to interactions (eg action invocation or property edit).

We suggest naming such classes `XxxSubscriptions`, eg:

```
@DomainService(
    nature=NatureOfService.DOMAIN                              ①
)
@DomainServiceLayout(
    menuOrder="10",
    name="...",
}
public CustomerOrderSubscriptions {
    @com.google.common.eventbus.Subscribe
    public void on(final Customer.DeletedEvent ev) {
        Customer customer = ev.getSource();
        orderRepository.delete(customer);
    }
    @Inject
    OrderRepository orderRepository;
}
```

① subscriptions do not appear in the UI at all, so should use the domain nature of service

4.3.2. Prototyping

While for long-term maintainability we do recommend the naming conventions described above, you can get away with far fewer services when just prototyping a domain.

If the domain service nature is not specified (or is left to its default, `VIEW`), then the service's actions will appear in the UI both as menu items *and* as contributions (and the service can of course be injected into other domain objects for programmatic invocation).

Later on it is easy enough to refactor the code to tease apart the different responsibilities.

4.3.3. Scoped services

By default all domain services are considered to be singletons, and thread-safe.

Sometimes though a service's lifetime is applicable only to a single request; in other words it is request-scoped.

The CDI annotation `@javax.enterprise.context.RequestScoped` is used to indicate this fact:

```
@javax.enterprise.context.RequestScoped
public class MyService extends AbstractService {
    ...
}
```

The framework provides a number of request-scoped services, include a `Scratchpad` service query results caching through the `QueryResultsCache`, and support for co-ordinating bulk actions through the `ActionInvocationContext` service. See the domain services reference guide for further details.

4.3.4. Registering domain services

The easiest way to register domain services is using `AppManifest` to specify the modules which contain `@DomainService`-annotated classes.

For example:

```
public class MyAppManifest implements AppManifest {
    public List<Class<?>> getModules() {
        return Arrays.asList(
                ToDoAppDomainModule.class,
                ToDoAppFixtureModule.class,
                ToDoAppAppModule.class,
                org.isisaddons.module.audit.AuditModule.class);
    }
    ...
}
```

will load all services in the packages underneath the four modules listed.

An alternative (older) mechanism is to registered domain services in the `isis.properties` configuration file, under `isis.services` key (a comma-separated list); for example:

```
isis.services = com.mycompany.myapp.employee.Employees\,
                com.mycompany.myapp.claim.Claims\,
                ...
```

This will then result in the framework instantiating a single instance of each of the services listed.

If all services reside under a common package, then the `isis.services.prefix` can specify this prefix:

```
isis.services.prefix = com.mycompany.myapp
isis.services = employee.Employees,\
                claim.Claims,\
                ...
```

This is quite rare, however; you will often want to use default implementations of domain services that are provided by the framework and so will not reside under this prefix.

Examples of framework-provided services (as defined in the applib) include clock, auditing, publishing, exception handling, view model support, snapshots/mementos, and user/application settings management; see the domain services reference guide for further details.

4.3.5. Initialization

Services can optionally declare lifecycle callbacks to initialize them (when the app is deployed) and to shut them down (when the app is undeployed).

An Apache Isis session *is* available when initialization occurs (so services can interact with the object store, for example).

The framework will call any `public` method annotated with `@PostConstruct` with either no arguments of an argument of type `Map<String,String>`

or

In the latter case, the framework passes in the configuration (`isis.properties` and any other component-specific configuration files).

Shutdown is similar; the framework will call any method annotated with `@PreDestroy`.

4.3.6. The getId() method

Optionally, a service may provide a `getId()` method. This method returns a logical identifier for a service, independent of its implementation.

4.4. Object Management (CRUD)

> TODO

4.4.1. Instantiating and Persisting Objects

> TODO - using `DomainObjectContainer`'s support for creation and persistence

4.4.2. Finding Objects

> TODO - using `DomainObjectContainer`

Using DataNucleus type-safe queries

> TODO - as described here

4.4.3. Deleting Objects

> TODO using `DomainObjectContainer`'s support for persistence

4.5. Entity Relationships

> TODO

4.5.1. Mandatory and Optional

TODO

4.5.2. 1-m bidir relationships

When an object is added to a 1:m bidirectional relationship, the child object must refer to the parent and the child must be added to the parent's children collection.

If there were no database involved then we would have recommended that you use the mutual registration pattern to ensure that both the parent and child are updated correctly. (The `modify…()` and `clear…()` supporting methods were introduced in the framework primarily to help support implement the mutual registration pattern.

However, in a relational database, these two operations in the domain object model correspond simply to updating the foreign key of the child table to reference the parent's primary key.

So long as the parent's children collection is a `java.util.Set` (rather than a `Collection` or a `List`), the JDO Objectstore will automatically maintain both sides of the relationship. All that is necessary is to set the child to refer to the parent.

For example, all you need write is:

```
public class Department {
    @javax.jdo.annotations.Persistent(mappedBy="department") ①
    private SortedSet<Employee> employees = new TreeSet<Employee>();

    public SortedSet<Employee> getEmployees() { ... }
    public void setEmployees(SortedSet<Employee> employees) { ... }
    ...
}
public class Employee {
    private Department department;
    public Department getDepartment() { ... }
    public void setDepartment(Department department) { ... }
    ...
}
```

① it's the `mappedBy` attribute that tells DataNucleus this is a bidirectional relationship. The value "department" refers to the Employee#department property.

Moreover, when maintaining a bidirectional 1-n relationship that is automatically managed by DataNucleus, it's preferred to "add" to the parent's child collection, don't set the parent on the child.

> If you don't do this then you may hit a `NullPointerException`. The above idiom fixes the issue.
>
> For more information, see this thread on the Apache Isis users mailing list, including this message with the above recommendation.

In fact, not only do you not need to manually maintain the relationship, we have noted on at least one occasion a subtle error if the code is programmatically added.

The error in that case was that the same object was contained in the parents collection. This of course should not happen for a `TreeSet`. However, JDO/DataNucleus replaces the `TreeSet` with its own implementation, and (either by design or otherwise) this does not enforce `Set` semantics.

The upshot is that you should NEVER programmatically add the child object to the parent's collection if using JDO Objectstore.

4.6. Contributed Members

TODO - generalize to discussion on contributed collections/properties; update to new annotations

Any n-parameter action provided by a service will automatically be contributed to the list of actions for each of its (entity) parameters. From the viewpoint of the entity the action is called a contributed action.

For example, given a service:

```
public interface Library {
    public Loan borrow(Loanable l, Borrower b);
}
```

and the entities:

```
public class Book implements Loanable { ... }
```

and

```
public class LibraryMember implements Borrower { ... }
```

then the `borrow(…)` action will be contributed to both `Book` and to `LibraryMember`.

This is an important capability because it helps to decouple the concrete classes from the services.

If necessary, though, this behaviour can be suppressed by annotating the service action with `@org.apache.isis.applib.annotations.NotContributed`.

For example:

```
public interface Library {
    @NotContributed
    public Loan borrow(Loanable l, Borrower b);
}
```

If annotated at the interface level, then the annotation will be inherited by every concrete class. Alternatively the annotation can be applied at the implementation class level and only apply to that particular implementation.

Note that an action annotated as being `@NotContributed` will still appear in the service menu for the service. If an action should neither be contributed nor appear in service menu items, then simply annotate it as `@Hidden`.

4.6.1. Contributed Action

TODO

4.6.2. Contributed Property

TODO

4.6.3. Contributed Collection

TODO

4.7. Business Rules

TODO

4.7.1. Visibility ("see it")

TODO - `hide…()`

Hide a Property

Hide a Collection

Hide an Action

Hide a Contributed Property, Collection or Action

All Members Hidden

4.7.2. Usability ("use it")

TODO - `disable…()`

Disable a Property

Disable a Collection

Disable an Action

Disable a Contributed Property, Collection or Action

All Members Unmodifiable (Disabling the Edit Button)

Sometimes an object is unmodifiable.

In the Wicket viewer this means disabling the edit button.

Declarative

`@DomainObject(editing=…)`

Imperative

4.7.3. Validity ("do it")

TODO - `validate…()`, `validateAddTo…()`, `validateRemoveFrom…()` and `validate()`

Validate (change to) a Property

Validate (adding or removing from) a Collection

Validate (arguments to invoke) an Action

Validating a Contributed Property, Collection or Action

Declarative validation

TODO - using `@Parameter#mustSatisfy()`, `@Property#mustSatisfy()`

4.8. Derived Members

TODO

4.8.1. Derived Property

TODO

4.8.2. Derived Collection

> TODO

While derived properties and derived collections typically "walk the graph" to associated objects, there is nothing to prevent the returned value being the result of invoking a repository (domain service) action.

For example:

```
public class Customer {
    ...
    public List<Order> getMostRecentOrders() {
        return orderRepo.findMostRecentOrders(this, 5);
    }
}
```

4.8.3. Trigger on property change

> TODO - `modify…()`, `clear…()`

4.8.4. Trigger on collection change

> TODO - `addTo…()`, `removeFrom…()`

4.9. Drop Downs and Defaults

> TODO

4.9.1. For Properties

> TODO

Choices for Property

> TODO - `choices…()`

Auto-complete for property

> TODO - `autoComplete…()`

Default for property

> TODO - `default…()`

4.9.2. For Action Parameters

> TODO

Choices for action parameter

> TODO - `choices…()`

Dependent choices for action params

> TODO - `choices…()`

Auto-complete for action param

> TODO - `autoComplete…()`

Default for action param

> TODO - `default…()`

4.9.3. For both Properties and Action Parameters

> TODO

Drop-down for limited number of instances

> TODO - `@DomainObject#bounded()`

Auto-complete (repository-based)

`@DomainObject#autoCompleteRepository()`

4.10. Bulk Actions

> TODO - `@Action#invokeOn()`

4.11. Collections of values

Although in Apache Isis you can have properties of either values (string, number, date etc) or of (references to other) entities, with collections the framework (currently) only supports collections of (references to) entities. That is, collections of values (a bag of numbers, say) are not supported.

However, it is possible to simulate a bag of numbers using view models.

4.11.1. View Model

TODO

4.11.2. Persistence Concerns

TODO - easiest to simply store using DataNucleus' support for collections, marked as `@Programmatic` so that it is ignored by Apache Isis. Alternatively can store as json/xml in a varchar(4000) or clob and manually unpack.

4.12. Subclass properties in tables

Suppose you have a hierarchy of classes where a property is derived and abstract in the superclass, concrete implementations in the subclasses. For example:

```
public abstract class LeaseTerm {
    public abstract BigDecimal getEffectiveValue();
    ...
}

public class LeaseTermForIndexableTerm extends LeaseTerm {
    public BigDecimal getEffectveValue() { ... }
    ...
}
```

Currently the Wicket viewer will not render the property in tables (though the property is correctly rendered in views).

For more background on this workaround, see ISIS-582.

The work-around is simple enough; make the method concrete in the superclass and return a dummy implementation, eg:

```
public abstract class LeaseTerm {
    public BigDecimal getEffectiveValue() {
        return null;        // workaround for ISIS-582
    }
    ...
}
```

Alternatively the implementation could throw a `RuntimeException`, eg

```
throw new RuntimeException("never called; workaround for ISIS-582");
```

Chapter 5. Object Layout

In implementing the naked objects pattern, Apache Isis aims to infer as much information from the domain classes as possible. Nevertheless, some metadata relating solely to the UI is inevitably required.

This chapter describes how this is done both for domain objects — statically or dynamically — and for the application menu bar (containing domain service' actions).

5.1. Static Object Layout

Metadata providing UI hints can be specified either statically, using annotations, or dynamically, using either a `layout.xml` file or a `.layout.json` file.

This section describes the static approach, using annotations.

> Tabs and tabgroups are only supported using `layout.xml` files; they are not supported by annotations.

5.1.1. `@MemberOrder`

The `@MemberOrder` annotation is used to specify the relative order of domain class properties, collections and actions.

The annotation defines two attributes, `name()` and `sequence()`. Their usage depends on the member type:

- for properties, the `name()` is used to group properties together into a member group (also called a property group or a fieldset. The `sequence()` then orders properties within these groups. If no `name()` is specified then the property is placed in a fallback "General" group, called "General".

  The name of these member groups/fieldsets are then referenced by `@MemberGroupLayout`.

- for collections, the `name()` attribute is (currently) unused. The `sequence()` orders collections relative to one another
- for actions, the `name()` attribute associates an action with either a property or with a collection.
  - If the `name()` attribute matches a property name, then the action's button is rendered close to the property, according to `@ActionLayout#position()` attribute.
  - On the other hand if the `name() attribute matches a collection name, then the action's button is rendered on the collection's header.
  - If there is no `name()` value, then the action is considered to pertain to the object as a whole, and its button is rendered close to the object's icon and title.

Within any of these, the `sequence()` then determines the relative ordering of the action with respect to other actions that have been similarly associated with properties/collections or left as "free-standing".

For example:

```
public class ToDoItem {
    @MemberOrder(sequence="1")
    public String getDescription() { ... }
    @MemberOrder(sequence="2")
    public String getCategory() { ... }
    @MemberOrder(sequence="3")
    public boolean isComplete() { ... }
    @MemberOrder(name="Detail", sequence="1")
    public LocalDate getDueBy() { ... }
    @MemberOrder(name="Detail", sequence="2")
    public BigDecimal getCost() { ... }
    @MemberOrder(name="Detail", sequence="4")
    public String getNotes() { ... }
    @MemberOrder(name="Misc", sequence="99")
    public Long getVersionSequence() { ... }
    ...
}
```

This defines three property (or member) groups, "General", "Detail" and "Misc"; "General" is the default if no `name` attribute is specified. Properties in the same member group are rendered together, as a fieldset.

In addition, actions can optionally be associated (rendered close to) either properties or actions. This is done by overloading the `@MemberOrder`'s `name()` attribute, holding the value of the property or collection.

For example:

```
public class ToDoItem {
    @MemberOrder(sequence="3")
    public boolean isComplete() { ... }
    @MemberOrder(name="complete", sequence="1")
    public ToDoItem completed() { ...}
    @MemberOrder(name="complete", sequence="2")
    public ToDoItem notYetCompleted() { ...}

    @MemberOrder(sequence="1")
    public SortedSet<ToDoItem> getDependencies() { ... }
    @MemberOrder(name="dependencies", sequence="1")
    public ToDoItem add(ToDoItem t) { ...}
    @MemberOrder(name="dependencies", sequence="2")
    public ToDoItem remove(ToDoItem t) { ...}
    ...
}
```

will associate the `completed()` and `notYetCompleted()` actions with the `complete` property, and will associate the `add()` and `remove()` actions with the `dependencies` collection.

The value of `sequence()` is a string. The simplest convention (as shown in the example above) is to use numbers — 1, 2, 3 — though it is a better idea to leave gaps in the numbers — 10, 20, 30 perhaps — such that a new member may be added without having to edit existing numbers.

Even better is to adopt the 'dewey-decimal' notation — 1, 1.1, 1.2, 2, 3, 5.1.1, 5.2.2, 5.2, 5.3 — which allows for an indefinite amount of future insertion. It also allows subclasses to insert their class members as required.

5.1.2. `@MemberGroupLayout`

The `@MemberGroupLayout` annotation specifies the relative positioning of property groups/fieldsets as being either in a left column, a middle column or in a right column. The annotation also specifies the relative width of the columns.

The property groups/fieldsets in this case are those that are inferred from the `@MemberOrder#name()` attribute.

> It is also possible to combine `@MemberOrder` with dynamic layouts, either using XML or JSON. The layout file defines only the regions of a grid structure (fieldsets/columns etc), but does *not* specify the properties/collections/actions within those grid regions. The `@MemberOrder` annotation in effect "binds" the properties or collections to those regions of the grid.
>
> When dynamic layouts are used this way, the `@MemberGroupLayout` annotation is essentially ignored, but the metadata from the `@MemberOrder` annotation (and the other layout annotations, `@ActionLayout`, `@PropertyLayout` and `@CollectionLayout`) are all still honoured.

For example:

```
@MemberGroupLayout(
     columnSpans={3,3,0,6},
     left={"General", "Misc"},
     middle="Detail"
 )
public class ToDoItem {
    ...
}
```

Four values are given in the `columnSpans` attribute. The first three are the relative widths of the three columns of property groups. The fourth, meanwhile, indicates the width of a final column that holds all the collections of the object.

The values of these spans are taken as proportions of 12 virtual columns across the page (this taken from the Bootstrap library).

For example:

* `{3,3,0,6}` indicates:
  * a left column of properties taking up 25% of the width
  * a middle column of properties taking up 25% of the width
  * a right column of collections taking up 50% of the width
* `{2,6,0,4}` indicates:
  * a left column of properties taking up ~16% of the width
  * a middle column of properties taking up 50% of the width
  * a right column of collections taking up ~33% of the width
* `{2,3,3,4}` indicates:
  * a left column of properties taking up ~16% of the width
  * a middle column of properties taking up 25% of the width
  * a right column of properties taking up 25% of the width
  * a far right column of collections taking up ~33% of the width

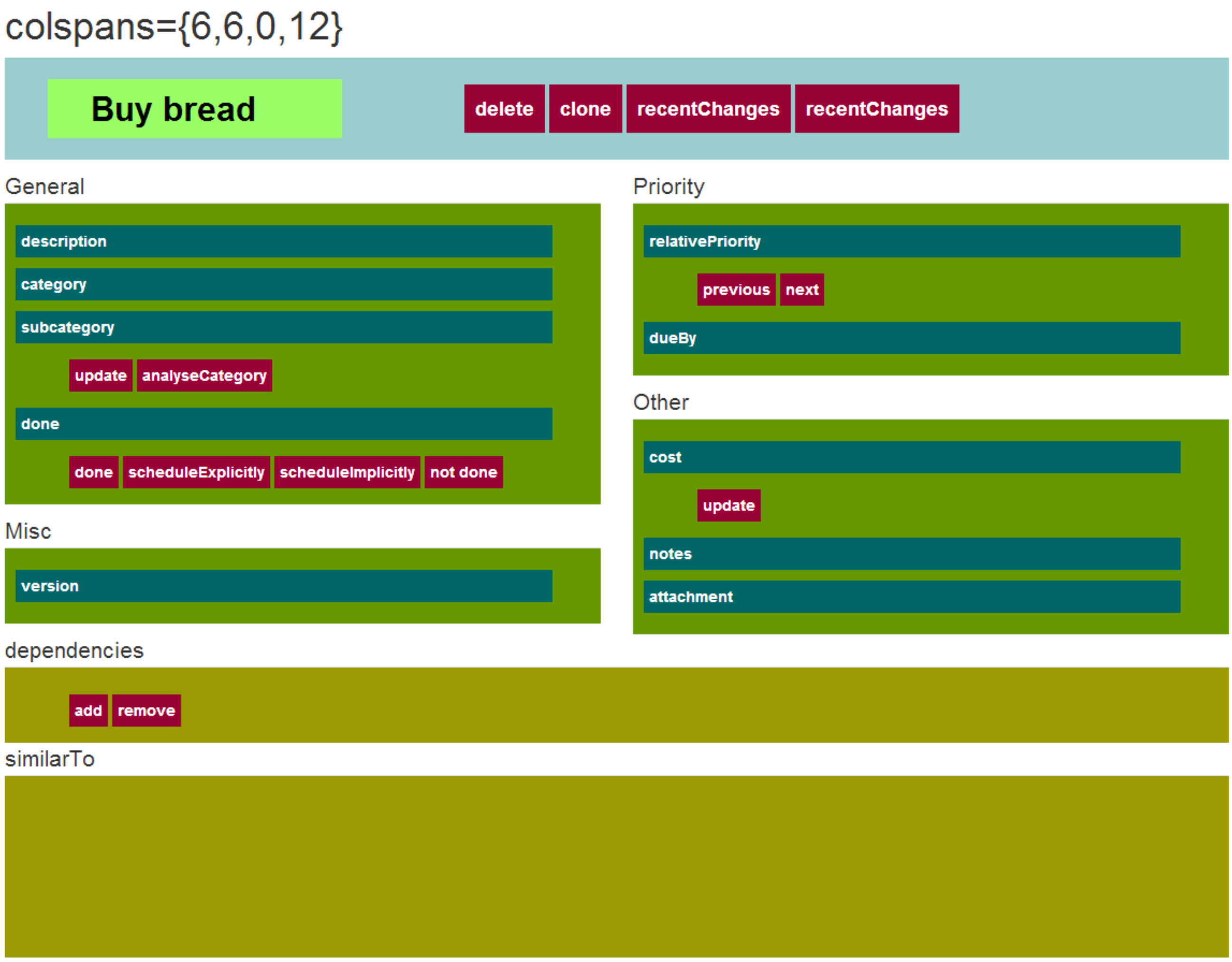
If the sum of all the columns exceeds 12, then the collections are placed underneath the properties, taking up the full span. For example:

* {4,4,4,12} indicates:
  * a left column of properties taking up ~33% of the width
  * a middle column of properties taking up ~33% of the width
  * a right column of properties taking up ~33% of the width
  * the collections underneath the property columns, taking up the full width

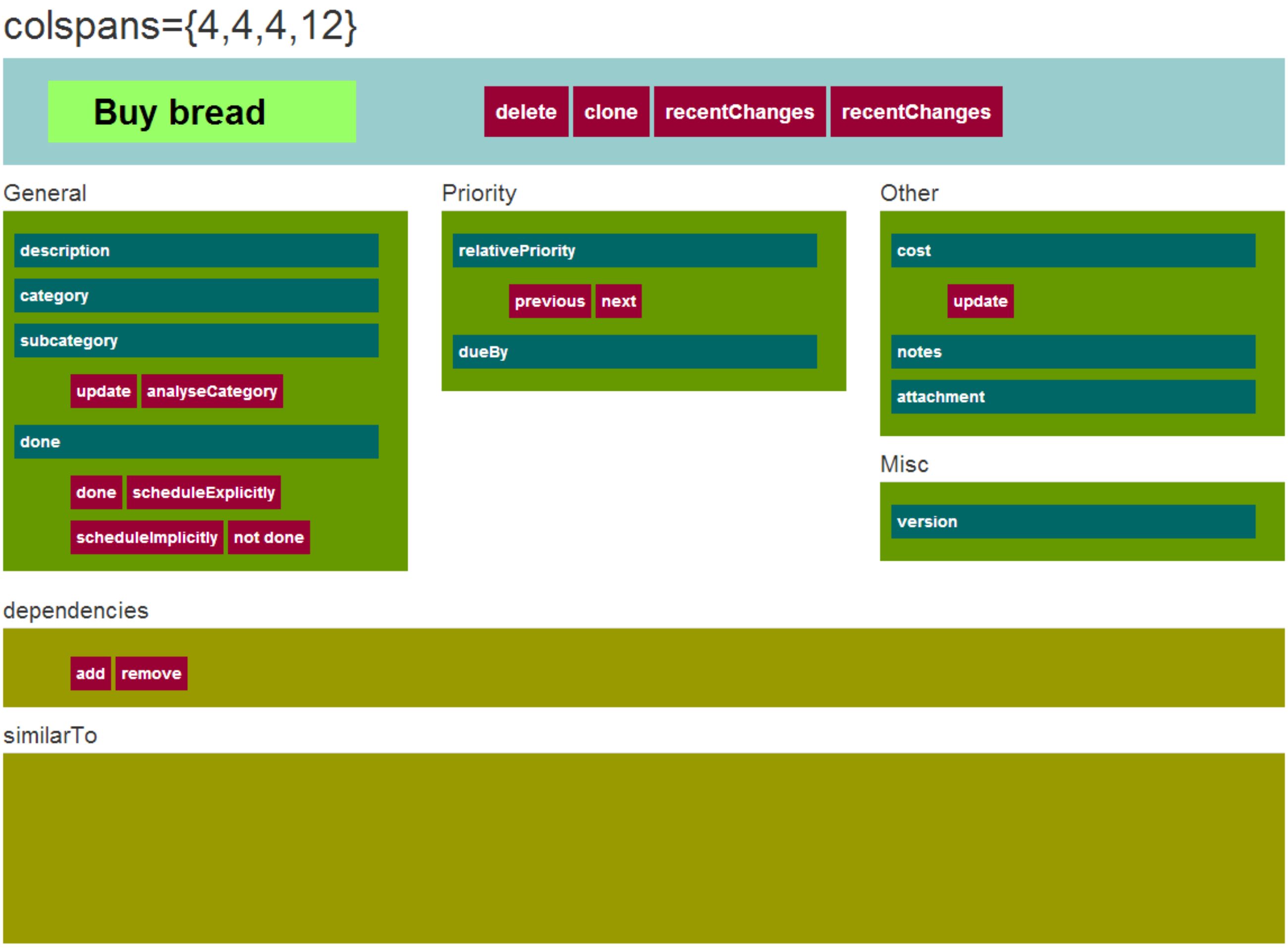
5.1.3. Example Layouts

Below are sketches for the layout of the ToDoItem class of the Isis addons example todoapp (not ASF):

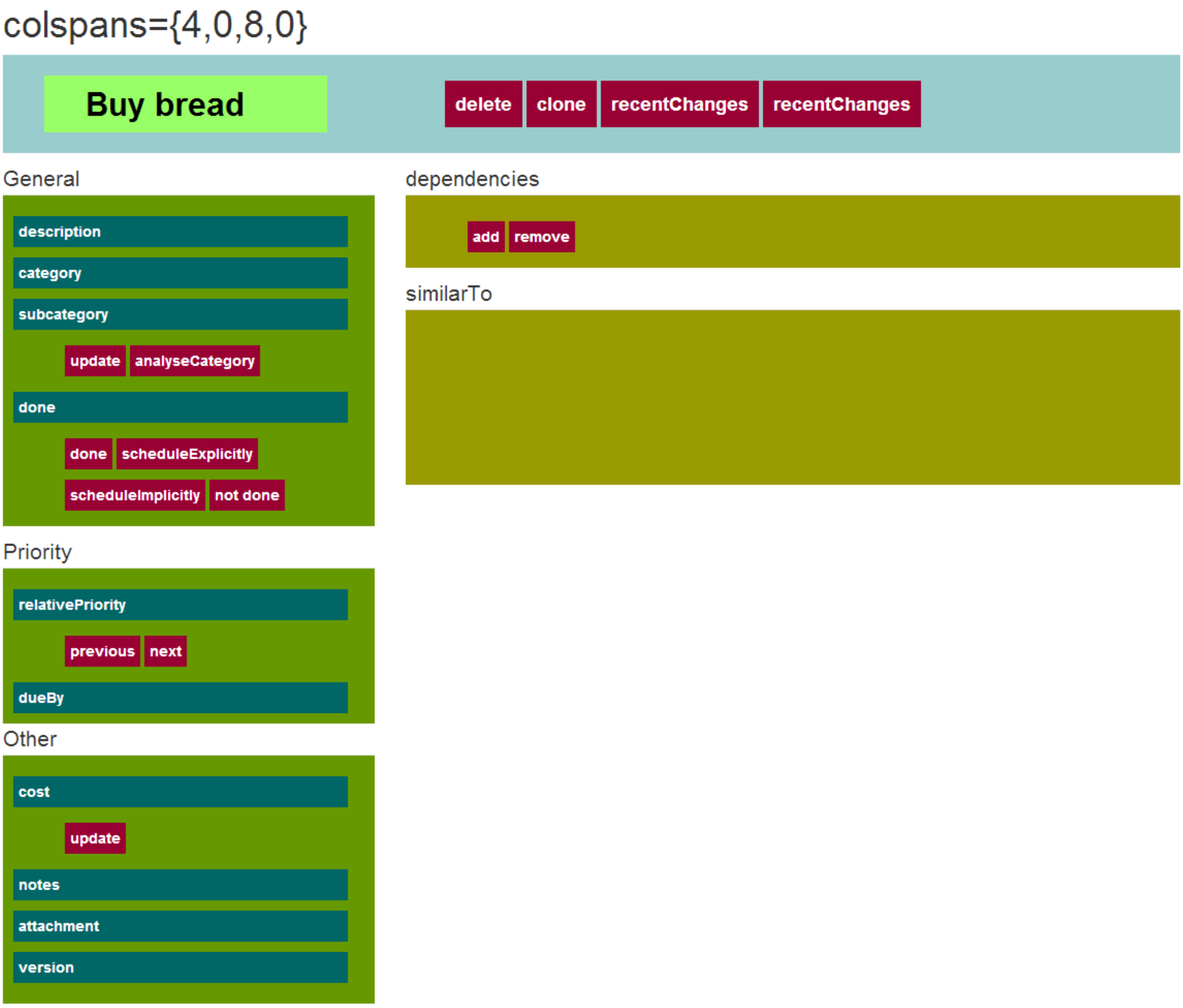
The first divides the properties into two equal sized columns (6-6-0) and puts the collections underneath (12):



The next divides the collections into three equal sized columns (4-4-4) and again puts the collections underneath (12):



The last puts the properties into a single column (4-0) and places the collections into the other larger column (8-0):



5.1.4. Other Annotations

As of 1.8.0, all the layout annotations have been consolidated into the various `XxxLayout` annotations: `@ActionLayout` `@CollectionLayout`, `@DomainObjectLayout`, `@DomainServiceLayout`, `@ParameterLayout`, `@PropertyLayout`, and `@ViewModelLayout`

5.2. Dynamic (XML) Layout

Metadata providing UI hints can be specified either statically, using annotations, or dynamically using an `Xxx.layout.xml` file (where `Xxx` is the entity or view model object to be rendered).

The `Xxx.layout.xml` file is just the serialized form of a `Grid` layout class defined within Apache Isis' applib. These are JAXB-annotated classes with corresponding XSD schemas; the upshot of that is that IDEs such as IntelliJ and Eclipse can provide "intellisense", making iteasy to author such layout files.

It is also possible to download an initial `.layout.xml` - capturing any existing layout metadata - using the `LayoutService` (exposed on the prototyping menu) or using a mixin action contributed to every domain object.

It is also possible to describe dynamic layouts using a `.layout.json` file, as discussed here. The `.layout.json` file should be considered as deprecated: the `.layout.xml` file also enables much more sophisticated layouts than those afforded by `.layout.json`.

5.2.1. Grids vs Components

The XML file distinguishes between two types of element:

- those that define a grid structure, of: rows, columns, tab groups and tabs.

  The rows and columns are closely modelled on Bootstrap 3 (used in the implementation of the Wicket viewer).

- those that defines common components, of: fieldsets (previously called member groups or property groups), properties, collections, actions and also the title/icon of the domain object itself.

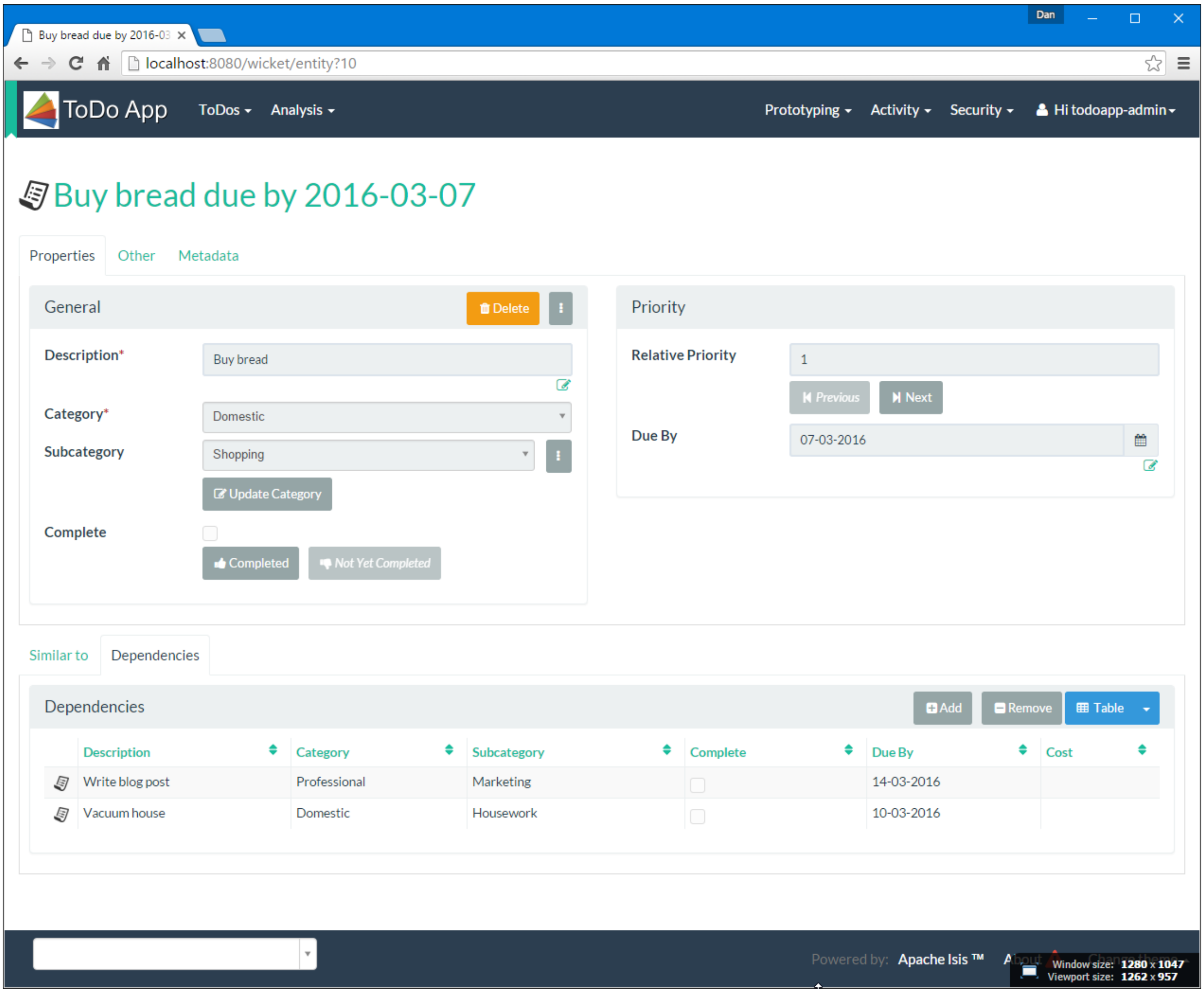
More information about these classes can be found in the reference guide. More information on Bootstrap 3's grid system can be found here.

5.2.2. Screencast

This screencast describes the feature.

5.2.3. Examples

Probably the easiest way to understand dynamic XML layouts is by example. For this we'll use the `ToDoItem` from the (non-ASF) Isis addons' todoapp:



Namespaces

First things first; every `.layout.xml` file must properly declare the XSD namespaces and schemas. There are two: one for the grid classes, and one for the common component classes:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bs3:grid
  xsi:schemaLocation="http://isis.apache.org/applib/layout/component
                      http://isis.apache.org/applib/layout/component/component.xsd
                      http://isis.apache.org/applib/layout/grid/bootstrap3

http://isis.apache.org/applib/layout/grid/bootstrap3/bootstrap3.xsd"
  xmlns:bs3="http://isis.apache.org/applib/layout/grid/bootstrap3"
  xmlns:c="http://isis.apache.org/applib/layout/component"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    ...
</bs3:grid>
```

Most IDEs will automatically download the XSD schemas from the specified schema locations, thereby providing "intellisense" help as you edit the file.

Rows, full-width cols, and tabs

The example layout consists of three rows: a row for the object/icon, a row containing a properties, and a row containing collections. In all three cases the row contains a single column spanning the full width of the page. For the property and collection rows, the column contains a tab group.

This corresponds to the following XML:

```
<bs3:row>
    <bs3:col span="12" unreferencedActions="true">
        <c:domainObject bookmarking="AS_ROOT"/>
    </bs3:col>
</bs3:row>
<bs3:row>
    <bs3:col span="12">
        <bs3:tabGroup>
            <bs3:tab name="Properties">...</bs3:tab>
            <bs3:tab name="Other">...</bs3:tab>
            <bs3:tab name="Metadata">...</bs3:tab>
        </bs3:tabGroup>
    </bs3:col>
</bs3:row>
<bs3:row>
    <bs3:col span="12">
        <bs3:tabGroup unreferencedCollections="true">
            <bs3:tab name="Similar to">...</bs3:tab>
            <bs3:tab name="Dependencies">...</bs3:tab>
        </bs3:tabGroup>
    </bs3:col>
</bs3:row>
```

You will notice that one of the `col`umns has an `unreferencedActions` attribute, while one of the `tabGroup`s has a similar `unreferencedCollections` attribute. This topic is discussed in more detail below.

Fieldsets

The first tab containing properties is divided into two columns, each of which holds a single fieldset of multiple properties. Those properties in turn can have associated actions.

This corresponds to the following XML:

```
            <bs3:tab name="Properties">
                <bs3:row>
                    <bs3:col span="6">
                        <c:fieldSet name="General" id="general"
unreferencedProperties="true">
                            <c:action id="duplicate" position="PANEL_DROPDOWN"/>
                            <c:action id="delete"/>
                            <c:property id="description"/>
                            <c:property id="category"/>
                            <c:property id="subcategory">
                                <c:action id="updateCategory"/>
                                <c:action id="analyseCategory" position="RIGHT"/>
                            </c:property>
                            <c:property id="complete">
                                <c:action id="completed" cssClassFa="fa-thumbs-up"/>
                                <c:action id="notYetCompleted" cssClassFa="fa-thumbs-
down"/>
                            </c:property>
                        </c:fieldSet>
                    </bs3:col>
                    <bs3:col span="6">
                        ...
                    </bs3:col>
                </bs3:row>
            </bs3:tab>
```

The tab defines two columns, each span of 6 (meaning half the width of the page).

In the first column there is a single fieldset. Notice how actions - such as `duplicate` and `delete` - can be associated with this fieldset directly, meaning that they should be rendered on the fieldset's top panel.

Thereafter the fieldset lists the properties in order. Actions can be associated with properties too; here they are rendered underneath or to the right of the field.

Note also the `unreferencedProperties` attribute for the fieldset; this topic is discussed in more detail below.

Collections

In the final row the collections are placed in tabs, simply one collection per tab. This corresponds to the following XML:

```xml
                <bs3:tab name="Similar to">
                    <bs3:row>
                        <bs3:col span="12">
                            <c:collection defaultView="table" id="similarTo"/>
                        </bs3:col>
                    </bs3:row>
                </bs3:tab>
                <bs3:tab name="Dependencies">
                    <bs3:row>
                        <bs3:col span="12">
                            <c:collection defaultView="table" id="dependencies">
                                <c:action id="add"/>
                                <c:action id="remove"/>
                            </c:collection>
                        </bs3:col>
                    </bs3:row>
                </bs3:tab>
```

As with properties, actions can be associated with collections; this indicates that they should be rendered in the collection's header.

5.2.4. Unreferenced Members

As noted in the preceding discussion, several of the grid's regions have either an `unreferencedActions`, `unreferencedCollections` or `unreferencedProperties` attribute.

The rules are:

- `unreferencedActions` attribute can be specified either on a column or on a fieldset.

  It would normally be typical to use the column holding the `<domainObject/>` icon/title, that is as shown in the example. The unreferenced actions then appear as top-level actions for the domain object.

- `unreferencedCollections` attribute can be specified either on a column or on a tabgroup.

  If specified on a column, then that column will contain each of the unreferenced collections, stacked one on top of the other. If specified on a tab group, then a separate tab will be created for each collection, with that tab containing only that single collection.

- `unreferencedProperties` attribute can be specified only on a fieldset.

The purpose of these attributes is to indicate where in the layout any unreferenced members should be rendered. Every grid *must* nominate one region for each of these three member types, the reason being that to ensure that the layout can be used even if it is incomplete with respect to the object members inferred from the Java source code. This might be because the developer forgot to update the layout, or it might be because of a new mixin (property, collection or action) contributed to many objects.

The framework ensures that in any given grid exactly one region is specified for each of the three `unreferenced…` attributes. If the grid fails this validation, then a warning message will be displayed, and the invalid XML logged. The layout XML will then be ignored.

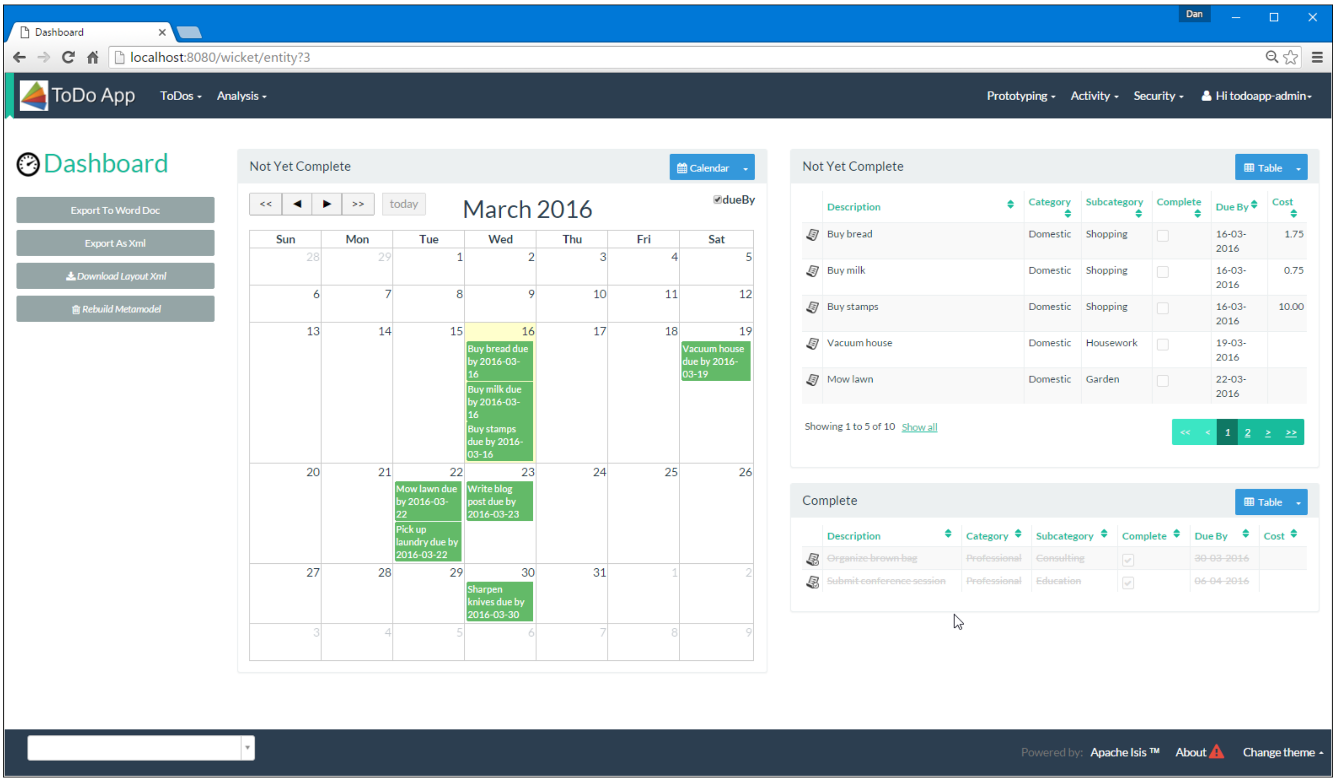
5.2.5. More advanced features

This section decribes a number of more features useful in more complex layouts.

Multiple references to a feature

One feature worth being aware of is that it is possible to render a single feature more than once.

For example, the dashboard home page for the (non-ASF) Isis addons' todoapp shows the "not yet complete" collection of todo items twice, once as a table and also as a calendar:



This is accomplished using the following (slightly abbreviated) layout:

```
<grid ...>
    <row>
        <col span="2" unreferencedActions="true">
            ...
        </col>
        <col span="5" unreferencedCollections="true" cssClass="custom-padding-top-20">
            <ns2:collection id="notYetComplete" defaultView="calendar"/>
①
        </col>
        <col span="5" cssClass="custom-padding-top-20">
            <ns2:collection id="notYetComplete" defaultView="table" paged="5"/>
②
            <ns2:collection id="complete" defaultView="table"/>
        </col>
        <col span="0">
            <ns2:fieldSet name="General" id="general" unreferencedProperties="true"/>
        </col>
    </row>
</grid>
```

① render the collection in "calendar" view

② also render the collection in "table" view

In the middle column the `notYetComplete` collection is rendered in "calendar" view, while in the right-most column it is rendered in "table" view.

It is also possible to reference object properties and actions more than once. This might be useful for a complex domain object with multiple tabs; certain properties or actions might appear on a summary tab (that shows the most commonly used info), but also on detail tabs.

Custom CSS

The ToDoApp's dashboard (above) also shows how custom CSS styles can be associated with specific regions of the layout:

```
<grid ...>
    <row>
        <col span="2" unreferencedActions="true">
            <ns2:domainObject/>
            <row>
                <col span="12" cssClass="custom-width-100">
①
                    <ns2:action id="exportToWordDoc"/>
                </col>
            </row>
            ...
        </col>
        <col span="5" unreferencedCollections="true" cssClass="custom-padding-top-20">
②
            ...
        </col>
        <col span="5" cssClass="custom-padding-top-20">
③
            ...
        </col>
    </row>
</grid>
```

① Render the column with the `custom-width-100` CSS class.

② Render the column with the `custom-padding-top-20` CSS class.

③ Ditto

For example the `custom-width-100` style is used to "stretch" the button for the `exportToWordDoc` action in the left-most column. This is accomplished with the following CSS in `application.css`:

```
.custom-width-100 ul,
.custom-width-100 ul li,
.custom-width-100 ul li a.btn {
    width: 100%;
}
```

Similarly, the middle and right columns are rendered using the `custom-padding-top-20` CSS class. This shifts them down from the top of the page slightly, using the following CSS:

```
.custom-padding-top-20 {
    padding-top: 20px;
}
```

5.2.6. Migrating from earlier versions

As noted earlier on, it is possible to download layout XML files using the `LayoutService` (exposed on the prototyping menu); this will download a ZIP file of layout XML files for all domain entities and view models. Alternatively the layout XML for a single domain object can be downloaded using the mixin action (contributed to every domain object).

There are four "styles":

- current
- complete
- normalized
- minimal

Ignorig the "current" style (which merely downloads the currently cached layout), 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, either from annotations in the metamodel or from an earlier `.layout.json` file if present. The table below summarises the choices:

Table 2. 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
`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.

Download either for a single domain object, or download all domain objects (entities and view models).

5.2.7. Domain Services

For more information about layouts, see:

* `LayoutService` (whose functionality is exposed on the prototyping menu as an action) and lso the a mixin action
* `GridService` and its supporting services, `GridLoaderService` and `GridSystemService`
* grid layout classes, defined in the Apache Isis applib

5.2.8. Required updates to the dom project's pom.xml

Any `.layout.xml` files must be compiled and available in the classpath. Ensure the following is defined in the dom project's `pom.xml`:

```
<resources>
    <resource>
        <filtering>false</filtering>
        <directory>src/main/resources</directory>
    </resource>
    <resource>
        <filtering>false</filtering>
        <directory>src/main/java</directory>
        <includes>
            <include>**</include>
        </includes>
        <excludes>
            <exclude>**/*.java</exclude>
        </excludes>
    </resource>
</resources>
```

If using an Apache Isis SimpleApp archetype, then the POM is already correctly configured.

5.3. Dynamic (JSON) Layout

Metadata providing UI hints can be specified either statically, using annotations, or dynamically, using either a `layout.xml` file or (as described here) a `.layout.json` file.

> The use of dynamic layouts through the `.layout.json` is DEPRECATED. Instead, use the `.layout.xml` file, which enables much more sophisticated custom layouts than those provided by `.layout.json`.
>
> It is possible to download initial `.layout.xml` files - which will capture all the metadata originally in the `.layout.json` file - using the `LayoutService` (exposed as an action on the prototyping menu). The `.layout.json` file will be ignored once a `.layout.xml` file is present.

5.3.1. JSON layout file

The JSON layout file for class `Xxx` takes the name `Xxx.layout.json`, and resides in the same package as the class.

The format of the file is:

```
{
  "columns": [                                // list of columns
    {
      "span": 6,                              // span of the left-hand property column
      "memberGroups": {                       // ordered map of member (property)
groups
        "General": {                          // member group name
          "members": {
            "description": {                  // property, no associated actions, but
with UI hint
              "propertyLayout": {
                "typicalLength": 50           // UI hint for size of field (no longer
used in ISIS 1.8.0)
              }
            },
            "category": {},
            "complete": {                     // property, with associated actions
              "propertyLayout": {
                "describedAs": "Whether this todo item has been completed"
              },
              "actions": {
                "completed": {
                  "actionLayout": {
                    "named": "Done",          // naming UI hint
                    "cssClass": "x-highlight" // CSS UI hint
                  }
                },
                "notYetCompleted": {
                  "actionLayout": {
                    "named": "Not done"
                  }
                }
              }
            }
          },
          "Misc": {
            "members": {
              "notes": {
                "propertyLayout": {
                  "multiLine": 5              // UI hint for text area
                }
              },
              "versionSequence": {}
```

```
          }
        }
      }
    }
  },
  {
    "span": 6,                          // span of the middle property column
    "memberGroups": { ... }
  },
  {
    "span": 0                           // span of the right property column (if
any)
  },
  {
    "span": 6,
    "collections": {                    // ordered map of collections
      "dependencies": {                 // collection, with associated actions
        "collectionLayout": {
          "paged": 10,                  // pagination UI hint
          "render": "EAGERLY"           // lazy-loading UI hint
        },
        "actions": {
          "add":{},
          "delete": {}
        },
      },
      "similarItems": {}                // collection, no associated actions
    }
  }
],
"actions": {                            // actions not associated with any
member
  "delete": {},
  "duplicate": {
    "actionLayout": {
      "named": {
        "value": "Clone"
      }
    }
  }
}
}
```

Although advisable, it is not necessary to list all class members in this file. Any members not listed with be ordered according either to annotations (if present) or fallback/default values.

Note also that the layout file may contain entries for contributed associations and actions; this allows each contributee classes to define their own layout for their contributions, possibly overriding any static metadata on the original domain service contributor.

5.3.2. Downloading an initial layout

The fastest way to get started is to use the (non-ASF) Isis addons' devutils module to download the layout file (derived from any existing static metadata defined by annotations).

5.3.3. Required updates to the dom project's pom.xml

Any `.layout.json` files must be compiled and available in the classpath. Ensure the following is defined in the dom project's `pom.xml`:

```
<resources>
    <resource>
        <filtering>false</filtering>
        <directory>src/main/resources</directory>
    </resource>
    <resource>
        <filtering>false</filtering>
        <directory>src/main/java</directory>
        <includes>
            <include>**</include>
        </includes>
        <excludes>
            <exclude>**/*.java</exclude>
        </excludes>
    </resource>
</resources>
```

If using an Apache Isis SimpleApp archetype, then the POM is already correctly configured.

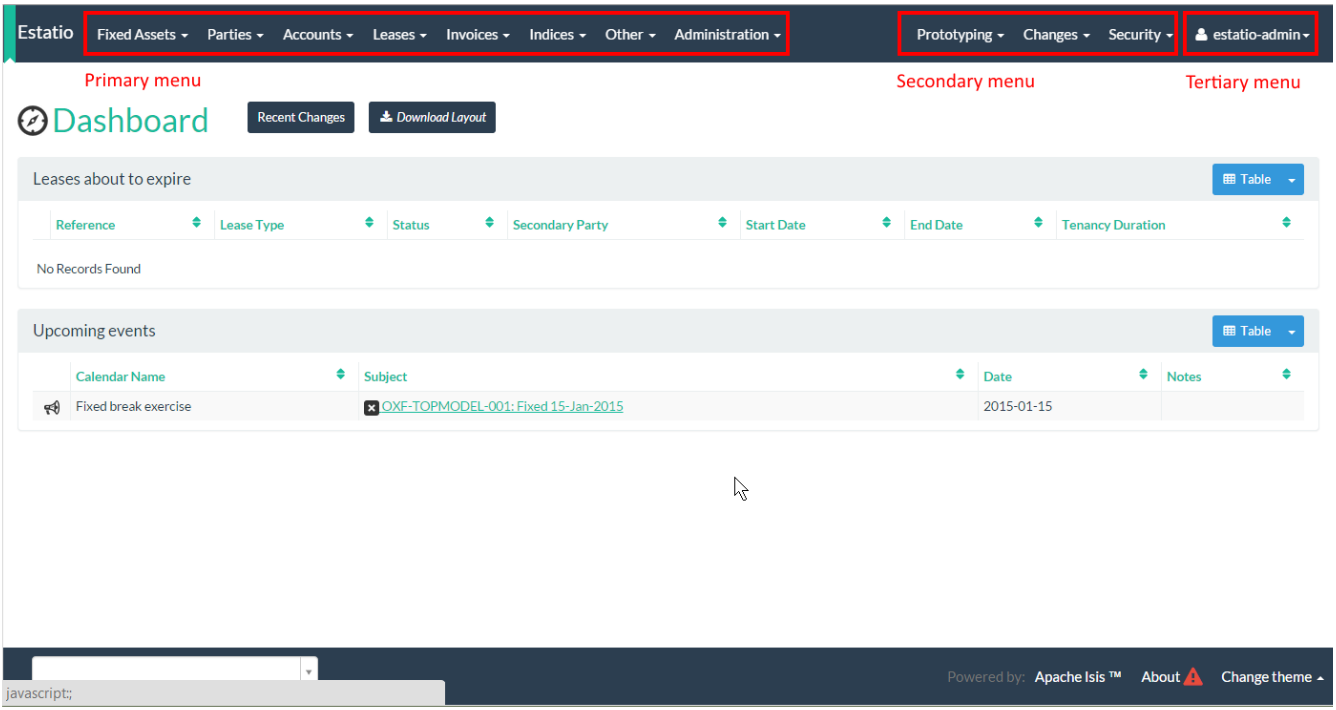
5.4. Application Menu Layout

The actions of domain services are made available as an application menu bar. By default each domain service corresponds to a single menu on this menu bar, with its actions as the drop-down menu items. This is rarely exactly what is required, however. The `@MemberOrder` and `@DomainServiceLayout` annotations can be used to rearrange the placement of menu items.

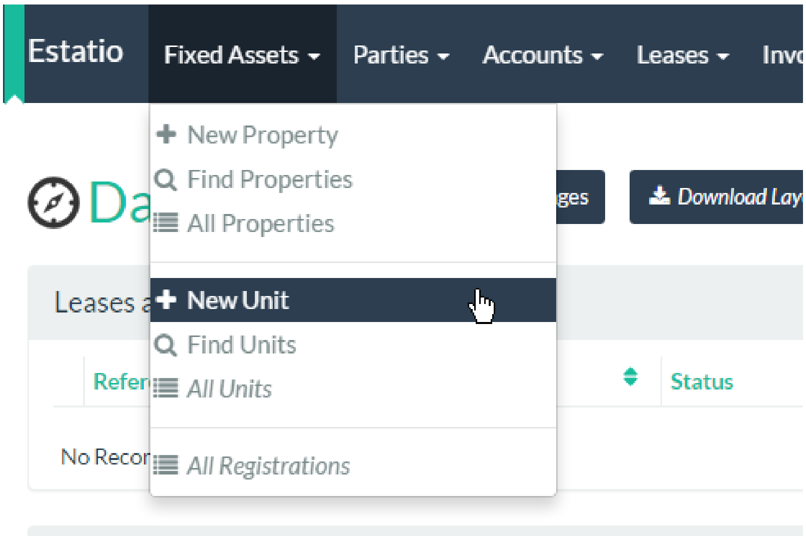
The screenshots below are taken from Estatio, an open source estate management application built using Apache Isis.

5.4.1. @DomainServiceLayout

Menus for domain services can be placed either on a primary, secondary or tertiary menu bar.



Within a single top-level menu (eg "Fixed Assets") there can be actions from multiple services. The Wicket viewer automatically adds a divider between each:



In the example above the top-level menu combines the actions from the `Properties`, `Units` and `FixedAssetRegistrations` services. The `Properties` service is annotated:

```
@DomainServiceLayout(
        named="Fixed Assets",
        menuBar = DomainServiceLayout.MenuBar.PRIMARY,
        menuOrder = "10.1"
)
public class Properties ... { ... }
```

while the `Units` service is annotated:

```
@DomainServiceLayout(
        named="Fixed Assets",
        menuBar = DomainServiceLayout.MenuBar.PRIMARY,
        menuOrder = "10.2"
)
public class Units ... { ... }
```

and similarly `FixedAssetRegistrations` is annotated:

```
@DomainServiceLayout(
        named="Fixed Assets",
        menuBar = DomainServiceLayout.MenuBar.PRIMARY,
        menuOrder = "10.3"
)
public class FixedAssetRegistrations ... { ... }
```

Note that in all three cases the value of the `named` attribute and the `menuBar` attribute is the same: "Fixed Assets" and PRIMARY. This means that all will appear on a "Fixed Assets" menu in the primary menu bar.

Meanwhile the value of `menuOrder` attribute is significant for two reasons:

- for these three services on the same ("Fixed Assets") top-level menu, it determines the relative order of their sections (`Properties` first, then `Units`, then `FixedAssetRegistrations`)
- it determines the placement of the top-level menu itself ("Fixed Assets") with respect to other top-level menus on the menu bar.

To illustrate this latter point, the next top-level menu on the menu bar, "Parties", is placed after "Fixed Assets" because the `menuOrder` of the first of its domain services, namely the `Parties` service, is higher than that for "Fixed Assets":

```
@DomainServiceLayout(
        named="Parties",
        menuBar = DomainServiceLayout.MenuBar.PRIMARY,
        menuOrder = "20.1"
)
public class Parties ... { ... }
```

Note that only the `menuOrder` of the *first* domain service is significant in placing the menus along the menu bar; thereafter the purpose of the `menuOrder` is to order the menu services sections on the menu itself.

5.4.2. Ordering menu actions

For a given service, the actions within a section on a menu is determined by the `@MemberOrder` annotation. Thus, for the `Units` domain service, its actions are annotated:

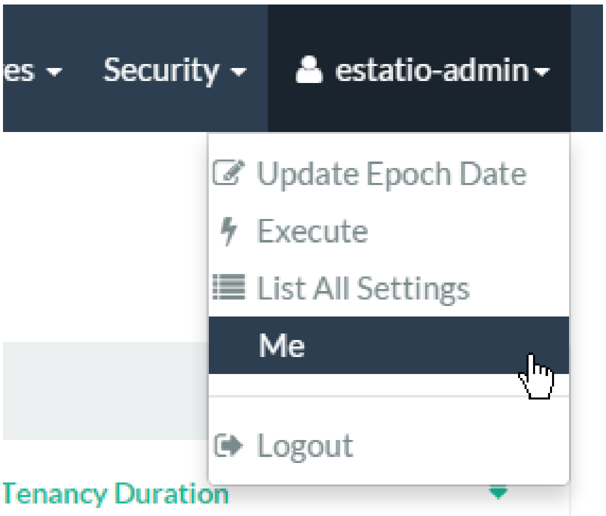
```
public class Units extends EstatioDomainService<Unit> {

    @MemberOrder(sequence = "1")
    public Unit newUnit( ... ) { ... }

    @MemberOrder(sequence = "2")
    public List<Unit> findUnits( ... ) { ... }

    @ActionLayout( prototype = true )
    @MemberOrder(sequence = "99")
    public List<Unit> allUnits() { ... }
    ...
}
```

Note that the last is also a prototype action (meaning it is only displayed in SERVER_PROTOTYPE (=Wicket Development) mode). In the UI it is rendered in italics.

(It is possible to override this place of a given action by specifying `@MemberOrder(name="…")` where the name is that of a top-level menu. Prior to 1.8.0 this was the only way of doing things, as of 1.8.0 its use is not recommended).

5.4.3. Tertiary menubar

The tertiary menu bar consists of a single unnamed menu, rendered underneath the user's login, top right. This is intended primarily for actions pertaining to the user themselves, eg their account, profile or settings:



Domain services' actions can be associated with the tertiary menu using the same `@DomainServiceLayout` annotation. For example, the `updateEpochDate(…)` and `listAllSettings(…)` actions come from the following service:

```
@DomainServiceLayout(
        menuBar = DomainServiceLayout.MenuBar.TERTIARY,
        menuOrder = "10.1"
)
public class EstatioAdministrationService ... {

    @MemberOrder(sequence = "1")
    public void updateEpochDate( ... ) { ... }

    @MemberOrder(sequence = "2")
    public List<ApplicationSetting> listAllSettings() { ... }
    ...
}
```

Because the number of items on the tertiary menu is expected to be small and most will pertain to the current user, the viewer does *not* place dividers between actions from different services on the tertiary menu.

5.4.4. Isis Add-on modules

Some of the (non-ASF) Isis Addons modules also provide services whose actions appear in top-level menus.

The security's module places its domain service menus in three top-level menus:

- its `ApplicationUsers`, `ApplicationRoles`, `ApplicationPermission`, `ApplicationFeatureViewModels` and `ApplicationTenancies` domain services are all grouped together in a single "Security" top-level menu, on the SECONDARY menu bar
- its `SecurityModuleAppFixturesService` domain service, which allows the security modules' fixture scripts to be run, is placed on a "Prototyping" top-level menu, also on the SECONDARY menu bar
- its `MeService` domain service, which provides the `me()` action, is placed on the TERTIARY menu bar.

Meanwhile the devutils module places its actions - to download layouts and so forth - on a "Prototyping" top-level menu, on the SECONDARY menu bar.

Currently there is no facility to alter the placement of these services. However, their UI can be suppressed using security or using a vetoing subscriber.

5.5. Static vs Dynamic Layouts

Using dynamic object layouts using JSON has the huge benefit that the layout can be updated without requiring a recompile of the code and redeploy of the app. Many developers also find it easier to rationalize about layout when all the hints are collated together in a single place (rather than scattered across the class members as annotations).

Another benefit of dynamic layout is that UI hints can be provided for contributed associations and actions that are synthesised at runtime.

The main downsides of using dynamic layouts are a lack of typesafety (a typo will result in the metadata not being picked up for the element) and syntactic fragility (an invalid JSON document will result in no metadata for the entire class).

Also, dynamic layouts have no notion of inheritance, whereas the dewey-decimal format `@MemberOrder` annotation allows the metadata of the subclass its superclasses to fit together relatively seamlessly.

5.5.1. Best of both worlds?

Using the (non-ASF) Isis addons' jrebel plugin comes close to getting the best of both words: metadata is specified in a type-safe way using annotations, but can be reloaded automatically.

The downsides are licensing cost, and also the fact that metadata for contributed actions in the contributee class cannot be specified.

Another open source alternative that you might also like to explore is DCEVM; there's a good write-up on the IntelliJ blog.

Chapter 6. FAQs

This chapter has FAQs (with solutions) for problems we've encountered ourselves or have been raised on the Apache Isis mailing lists.

See also Restful Objects hints-n-tips.

6.1. Enabling Logging

Sometimes you just need to see what is going on. There are various ways in which logging can be enabled, here are the ones we tend to use.

- In Apache Isis

  Modify `WEB-INF/logging.properties` (a log4j config file)

- In DataNucleus

  As per the DN logging page

- In the JDBC Driver

  Configure `log4jdbc` JDBC rather than the vanilla driver (see `WEB-INF/persistor_datanucleus.properties`) and configure log4j logging (see `WEB-INF/logging.properties`). There are examples of both in the SimpleApp archetype.

- In the database

  Details below.

Database logging can be configured:

- for HSQLDB

  by adding `;sqllog=3` to the end of the JDBC URL.

- for PostgreSQL:

  Can change `postgresql\9.2\data\postgresql.conf`; see this article for details.

- for MS SQL Server Logging:

  We like to use the excellent SQL Profiler tool.

6.2. Subtype not fully populated

Taken from this thread on the Apache Isis users mailing list…

If it seems that Apache Isis (or rather DataNucleus) isn't fully populating domain entities (ie leaving some properties as `null`), then check that your actions are not accessing the fields directly. Use getters instead. that is:

> ⚠ Properties of domain entities should always be accessed using getters. The only code that should access to fields should be the getters themselves.

Why so? Because DataNucleus will potentially lazy load some properties, but to do this it needs to know that the field is being requested. This is the purpose of the enhancement phase: the bytecode of the original getter method is actually wrapped in code that does the lazy loading checking. But hitting the field directly means that the lazy loading code does not run.

This error can be subtle: sometimes "incorrect" code that accesses the fields will seem to work. But that will be because the field has been populated already, for whatever reason.

One case where you will find the issue highlighted is for subtype tables that have been mapped using an inheritance strategy of `NEW_TABLE`, eg:

```
@javax.jdo.annotations.PersistenceCapable
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
public class SupertypeEntity {
    ...
}
```

and then:

```
@javax.jdo.annotations.PersistenceCapable
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
public class SubtypeEntity extends SupertypeEntity {
    ...
}
```

This will generate two tables in the database, with the primary key of the supertype table propagated as a foreign key (also primary key) of the subtype table (sometimes called "table per type" strategy). This means that DataNucleus might retrieve data from only the supertype table, and the lazily load the subtype fields only as required. This is preferable to doing a left outer join from the super- to the subtype tables to retrieve data that might not be needed.

On the other hand, if the `SUPERCLASS_TABLE` strategy (aka "table per hierarchy" or roll-up) or the `SUBCLASS_TABLE` strategy (roll-down) was used, then the problem is less likely to occur because DataNucleus would obtain all the data for any given instance from a single table.

Final note: references to other objects (either scalar references or in collections) in particular require that getters rather than fields to be used to obtain them: it’s hopefully obvious that DataNucleus (like all ORMs) should not and will not resolve such references (otherwise, where to stop… and the whole database would be loaded into memory).

In summary, there’s just one rule: **always use the getters, never the fields**.

6.3. How parse images in RO viewer?

From this thread on the Apache Isis users mailing list:

- *I am trying to display an image in a JavaScript client app, the image comes from an Isis RO web service as a string, but it won't show. Is there something I should do to change the message?*

The RO viewer returns the image as a string, in the form:

```
"Tacos.jpg:image/jpeg:/9j//4AAQSkZJRgABAQEAlgCWAAD/  ...."
```

This is in the form:

```
(filename):(mime type):(binary data in base64)
```

This is basically the `Blob` value type, in string form.

To use, split the parts then format the mime type and base64 data correctly before using as source in an `<img>` tag.

6.4. Enhance only (IntelliJ)

From the Apache Isis mailing list is:

- *Is there a simple way to make a run configuration in IntelliJ for running the datanucleus enhancer before running integration test?*

Yes, you can; here's one way:

- Duplicate your run configuration for running the webapp
  - the one where the main class is `org.apache.isis.WebServer`
  - there's a button for this on the run configurations dialog.
- then, on your copy change the main class to `org.apache.isis.Dummy`

6.5. Per-user Themes

From this thread on the Apache Isis users mailing list:

- *Is it possible to have each of our resellers (using our Isis application) use there own theme/branding with their own logo and colors? Would this also be possible for the login page, possibly depending on the used host name?*

Yes, you can do this, by installing a custom implementation of the Wicket Bootstrap's `ActiveThemeProvider`.

The Isis addons' todoapp (non-ASF) actually does this, storing the info via the Isis addons' settings module settings modules:

IActiveThemeProvider implementation

```
public class UserSettingsThemeProvider implements ActiveThemeProvider {
    ...
    @Override
    public ITheme getActiveTheme() {
        if(IsisContext.getSpecificationLoader().isInitialized()) {
            final String themeName = IsisContext.doInSession(new Callable<String>() {
                @Override
                public String call() throws Exception {
                    final Class<UserSettingsService> serviceClass =
UserSettingsService.class;
                    final UserSettingsService userSettingsService = lookupService
(serviceClass);
                    final UserSetting activeTheme = userSettingsService.find(
                            IsisContext.getAuthenticationSession().getUserName(),
ACTIVE_THEME);
                    return activeTheme != null ? activeTheme.valueAsString() : null;
                }
            });
            return themeFor(themeName);
        }
        return new SessionThemeProvider().getActiveTheme();
    }
    @Override
    public void setActiveTheme(final String themeName) {
        IsisContext.doInSession(new Runnable() {
            @Override
            public void run() {
                final String currentUsrName = IsisContext.getAuthenticationSession
().getUserName();
                final UserSettingsServiceRW userSettingsService =
                        lookupService(UserSettingsServiceRW.class);
                final UserSettingJdo activeTheme =
                        (UserSettingJdo) userSettingsService.find(currentUsrName,
ACTIVE_THEME);
                if(activeTheme != null) {
                    activeTheme.updateAsString(themeName);
                } else {
                    userSettingsService.newString(
                        currentUsrName, ACTIVE_THEME, "Active Bootstrap theme for
user", themeName);
                }
            }
        });
    }
    private ITheme themeFor(final String themeName) {
        final ThemeProvider themeProvider = settings.getThemeProvider();
        if(themeName != null) {
```

```
            for (final ITheme theme : themeProvider.available()) {
                if (themeName.equals(theme.name()))
                    return theme;
            }
        }
        return themeProvider.defaultTheme();
    }
    ...
}
```

and

Using the ActiveThemeProvider

```
@Override
protected void init() {
    super.init();

    final IBootstrapSettings settings = Bootstrap.getSettings();
    settings.setThemeProvider(new BootswatchThemeProvider(BootswatchTheme.Flatly));

    settings.setActiveThemeProvider(new UserSettingsThemeProvider(settings));
}
```

6.6. How i18n the Wicket viewer?

From this thread on the Apache Isis users mailing list:

- *I am trying to internationalize the label descriptions of form actions, eg those in* `ActionParametersFormPanel`*. Referencing those via their message id inside a .po file didn't work either. Can this be done?*

The above FAQ was raised against `1.10.0`. As of `1.11.0` (due to ISIS-1093) it *is* now possible to internationalize both the Wicket viewer's labels as well as the regular translations of the domain object metadata using the `.po` translation files as supported by the `TranslationService`.

Full details of the `msgId`s that must be added to the `translations.po` file can be found in i18n section of the beyond the basics guide.

In prior releases (`1.10.0` and earlier) it was necessary to use Wicket's internationalization support, namely resource bundles. This is still supported (as a fallback):

- create a directory structure inside the webapp resource folder following that pattern `org.apache.isis.viewer.wicket.ui.components.actions`
- Inside there create an equivalent `ActionParametersFormPanel_xx_XX.properties` or `ActionParametersFormPanel_xx.properties` file for the various locales that you want to support (eg `ActionParametersFormPanel_en_UK.properties`, `ActionParametersFormPanel_en_US.properties`, `ActionParametersFormPanel_de.properties` and so on).

6.7. How to handle void/null results

From this thread on the Apache Isis users mailing list:

- *When using a void action, let's say a remove action, the user is redirected to a page "no results". When clicking the back button in the browser the user sees "Object not found" (since you've just deleted this object).*
- *You can return a list for example to prevent the user from being redirect to a "No results" page, but I think it's not the responsibility of the controllers in the domain model.*
- *A solution could be that wicket viewer goes back one page when encountering a deleted object. And refresh the current page when receiving a null response or invoking a void action. But how to implement this?*

One way to implement this idea is to provide a custom implementation of the `RoutingService` SPI domain service. The default implementation will either return the current object (if not null), else the home page (as defined by `@HomePage`) if one exists.

The following custom implementation refines this to use the breadcrumbs (available in the Wicket viewer) to return the first non-deleted domain object found in the list of breadcrumbs:

```
@DomainService(nature = NatureOfService.DOMAIN)
@DomainServiceLayout(menuOrder = "1")                                              ①
public class RoutingServiceUsingBreadcrumbs extends RoutingServiceDefault {
    @Override
    public Object route(final Object original) {
        if(original != null) {                                                     ②
            return original;
        }
        container.flush();                                                         ③

        final BreadcrumbModelProvider wicketSession =                              ④
            (BreadcrumbModelProvider) AuthenticatedWebSession.get();
        final BreadcrumbModel breadcrumbModel =
            wicketSession.getBreadcrumbModel();
        final List<EntityModel> breadcrumbs = breadcrumbModel.getList();

        final Optional<Object> firstViewModelOrNonDeletedPojoIfAny =
                breadcrumbs.stream()                                               ⑤
                .filter(entityModel -> entityModel != null)
                .map(EntityModel::getObject)                                       ⑥
                .filter(objectAdapter -> objectAdapter != null)
                .map(ObjectAdapter::getObject)                                     ⑦
                .filter(pojo -> !(pojo instanceof Persistable) ||
                                !((Persistable)pojo).dnIsDeleted())                ⑧
                .findFirst();

        return firstViewModelOrNonDeletedPojoIfAny.orElse(homePage());            ⑨
    }
    private Object homePage() {
        return homePageProviderService.homePage();
    }
    @Inject
    HomePageProviderService homePageProviderService;
    @Inject
    DomainObjectContainer container;
}
```

① override the default imlpementation

② if a non-null object was returned, then return this

③ ensure that any persisted objects have been deleted.

④ reach inside the Wicket viewer's internals to obtain the list of breadcrumbs.

⑤ loop over all breadcrumbs

⑥ unwrap the Wicket viewer's serializable representation of each domain object (`EntityModel`) to the Isis runtime's representation (`ObjectAdapter`)

⑦ unwrap the Isis runtime's representation of each domain object (`ObjectAdapter`) to the domain object pojo itself

⑧

if object is persistable (not a view model) then make sure it is not deleted

⑨ return the first object if any, otherwise the home page object (if any).

Note that the above implementation uses Java 8, so if you are using Java 7 then you'll need to backport accordingly.

6.8. How to implement a spellchecker?

From this thread on the Apache Isis users mailing list:

- *What is the easiest way to add a spell checker to the text written in a field in a domain object, for instance to check English syntax?*

One way to implement is to use the event bus:

- Set up a domain event subscriber that can veto the changes.
- if the change is made through an action, you can use `@Action#domainEvent()`.

if if the change is made through an edit, you can use `@Property#domainEvent()`.

You'll need some way to know which fields should be spell checked. Two ways spring to mind:

- either look at the domain event's identifier
- or subclass the domain event (recommended anyway) and have those subclass events implement some sort of marker interface, eg a `SpellCheckEvent`.

And you'll (obviously) also need some sort of spell checker implementation to call.

6.9. How run fixtures on startup?

From this thread on the Apache Isis users mailing list:

- *my fixtures have grown into a couple of files the application needs to read in when it starts the first time (and possibly later on when the files content change). What is the right way to do this? Hook up into the webapp start? Use events?*

The standard approach is to use fixture scripts. These can be run in on start-up typically by being specified in the `AppManifest`, see for example the SimpleApp archetype.

Alternatively just set "isis.fixtures" and "isis.persistor.datanucleus.install-fixtures" properties.

In terms of implementations, you might also want to check out the (non-ASF) Isis addons' excel module, by using `ExcelFixture` and overriding `ExcelFixtureRowHandler` (same package). An example can be found in this (non ASF) contactapp, see `ContactRowHandler`.