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
- DataNucleus object store
- 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:

> 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.
>
> 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.
>
> 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.
>
> 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, *OOUI*s were notoriously hard to develop.
>
> 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.

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 (non-ASF) Incode Platform the gmap3 wicket extension renders any object with latitude/longitude on a map, while fullcalendar2 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 (we compare Apache Isis' with MetaWidget here), in particular the runtime generation of a UI for domain objects. You can hear more from Kennard and others on this Javascript Jabber podcast.

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.

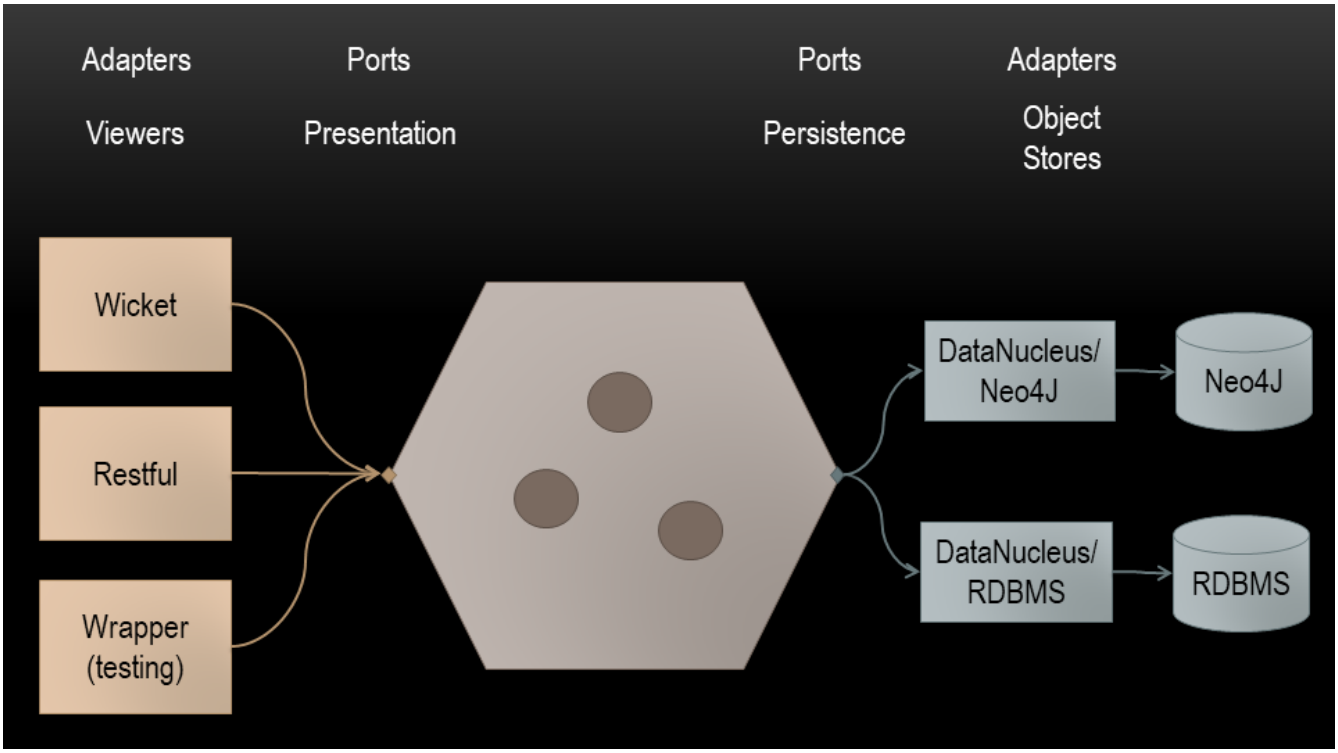


Figure 1. The hexagonal architecture emphasizes multiple implementations of the different layers.

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 — the originator of Naked Objects pattern — 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 (so to speak!) 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!

Apache 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 (non-ASF) Incode Platform's 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 (non-ASF) Incode Platform's command and 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 - 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), (contributed services) 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:

> 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.

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

Apache Isis is primarily aimed at custom-built "enterprise" applications. The UI provided 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 Angular or similar — for use by those requiring more guidance; typically end-users outside of the organization. This section describes some of the core principles and values that the framework aims to honour and support.

2.2.1. Why Build instead of Buy?

Buying packaged software makes sense for statutory requirements, such as payroll or general ledger, or document management/retrieval. But (we argue) 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.

Packaged software suffers from the problem of both having doing "too much" and "not enough":

- it does "too much" because it will have features that are not required by your business. These extra unnecessary features make the system difficult to learn and use.;
- but it may also do "too little" because there may be crucial functionality not supported by the

software.

The diagram below illustrates the dichotomy:

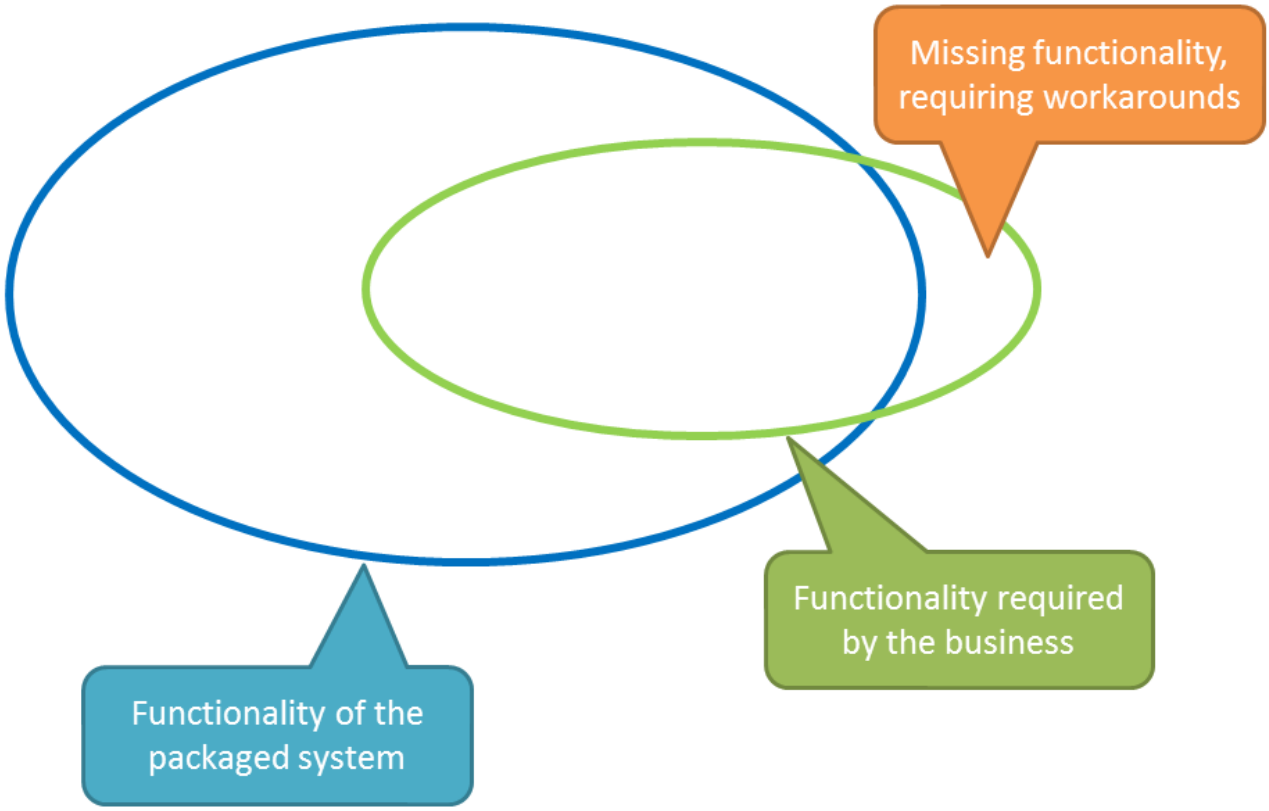


Figure 2. build-vs-buy

What happens in this case is that end-users — needing some sort of solution for their particular business problem — will end up using unused fields to store the information they need. We end up with no correlation between the fields definitions and the values stored therein, sometimes with the datatypes not even matching. Any business rules pertaining to this extra data have to be enforced manually by the users, rather than by the system. The end result is a system even more complicated to learn and use, with the quality of the data held within it degrading as end users inevitably make mistakes in using it.

There are other benefits too for building rather than buying. Packaged software is almost always sold with a support package, the cover of which can vary enormously. At one end of the spectrum the support package ("bronze", say) will amount to little more than the ability to raise bug reports and to receive maintenance patches. At the other end ("platinum"?), the support package might provide the ability to influence the direction of the development of the product, perhaps specific features missing by the business.

Even so, the more widely used is the software package, the less chance of getting it changed. Does anyone reading this think they could get a new feature added (or removed) from Microsoft Word, for example?

Here’s another reason why you should build, and not buy, the software supporting your core business domain. Although most packaged software is customisable to a degree, there is always a limit to what can be customised. The consequence is that the business is forced to operate according to the way in which the software requires.

This might be something as relatively innocuous as imposing its own terminology onto the business, meaning that the end-users must mentally translate concepts in order to use the software. But it might impose larger constraints on the business; some packaged software (we carefully mention no names) is quite notorious for this

If your business is using the same software as your competitor, then obviously there’s no competitive advantage to be gained. And if your competitor has well-crafted custom software, then your business will be at a competitive *dis*advantage.

So, our philosophy is that custom software — for your core business domain — is the way to go. Let’s now look more closely at the types of custom applications you can consider building with the framework.

2.2.2. For the long-term

Enterprise applications tend to stick around a long time; a business' core domains don’t tend to change all that often. What this means in turn is that the application needs to be maintainable, so that it is as easy to modify and extend when it’s 10 years old as when it was first written.

That’s a tall order for any application to meet, and realistically it *can* only be met if the application is modular. Any application that lacks a coherent internal structure will ultimately degrade into an unmaintable "big ball of mud", and the development team’s velocity/capacity to make changes will reduce accordingly.

Apache Isis' architecture allows the internal structure to be maintained in two distinct ways.

- first, the naked objects pattern acts as a "firewall", ensuring that any business logic in the domain layer doesn’t leak out into the presentation layer (it can’t, because the developer doesn’t write any controllers/views).
- second, the framework’s provides various features (discussed in more detail below) to allow the different modules *within* the domain layer to interact with each in a decoupled fashion.

The diagram below illustrates this:

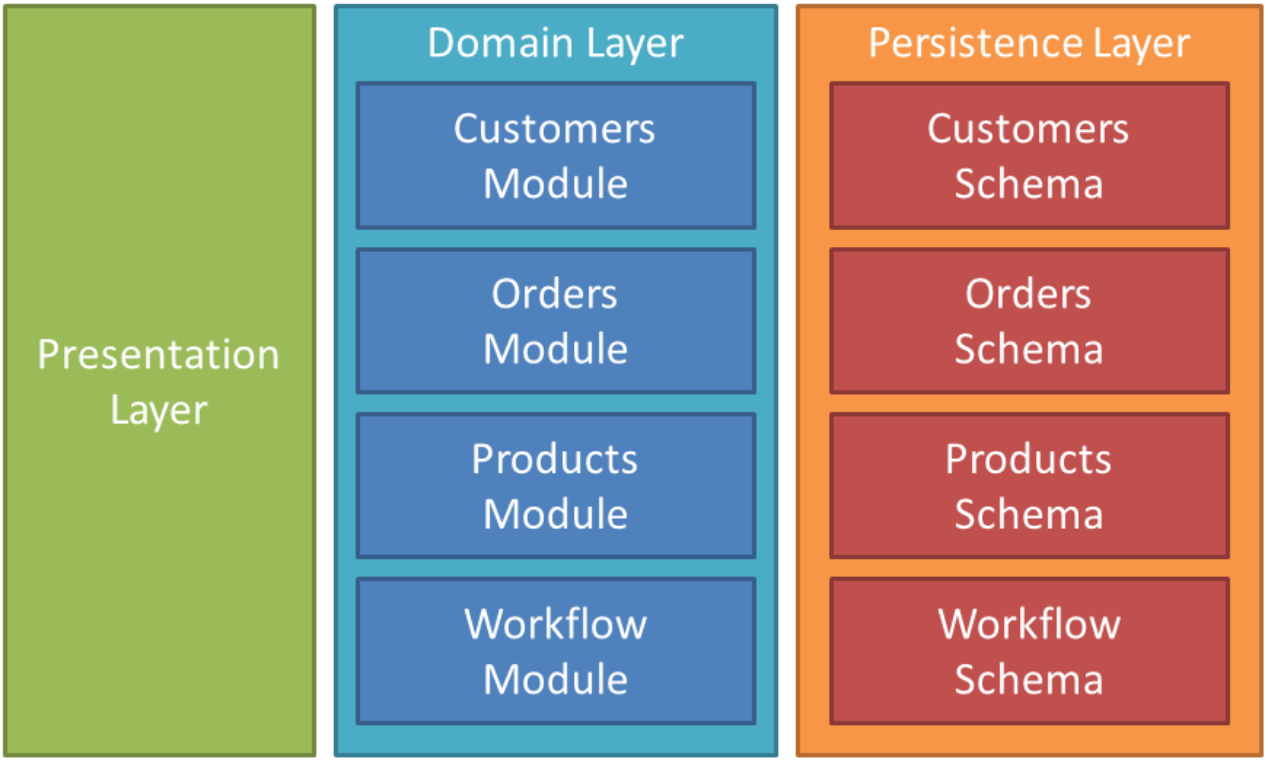


Figure 3. decoupled applications

Here, the presentation layer (Wicket UI or REST API) is handled by the framework, while the developer focusses on just the domain layer. The framework encourages splitting this functionality into modules; each such module has its counterpart (typically tables within a given RDBMS database schema) within the persistence layer.

This architecture means that it's impossible for business logic to leach out into the adjacent presentation layer because the developer doesn't (can't) write any code for presentation layer. We in effect have a "firewall" between the two layers.

To support the business domain being split into separate modules, the framework provides various features, the most important of which are:

- dependency injection of services

  Both framework-defined domain services and application-defined services (eg repositories and factories) are injected everywhere, using the regular `@javax.inject.Inject` annotation.

- mixins allow functionality defined in one module to appear (in the UI) to be provided by some other module.

  For example, a *Document* module might allow `Document` objects to be attached to any arbitrary domain object (such as `Order` or `Customer`) in other modules. A mixin would allow the UI for a `Customer` to also display these attached `Document`s, even though the *Customer* module would have no knowledge of/dependency on the *Workflow* module.

- internal event bus allows modules to influence other modules.

  A good example is supporting what (in an RDBMS we would call) referential integrity. Suppose

the *Customer* module has a `Customer` object and a `EmailAddress` object, with a customer having a collection of email addresses. A *Communications* module might then use those email addresses to create `EmailCommunication`s.

If the *Customer* module wants to delete an `EmailAddress` then the *Communications* module will probably want to veto this because they are "in use" by those `EmailCommunication`s. Or, it might conceivably perform a cascade delete of all associated communications. Either way, the *Communications* module receives an internal event representing the intention to delete the `EmailAddress`. It can then act accordingly, either vetoing the interaction or performing the cascade delete. The *Customer* module for its part does not know anything about this other module.

For those cases where a module needs to interact with other modules but does not know about their implementations, the module can either define its own SPI domain services or it can define custom domain events and fire them. This technique is also used extensively by the framework itself. For example, the `AuditerService` SPI enables custom auditing, and the `PublisherService` SPI enables custom publishing

When building a modular application, it's important to consider the logical layering of the modules: we don't need every module to be completely decoupled from every other. The most important requirement is that there are no cyclic dependencies, because otherwise we run the risk of the application degrading into a "big ball of mud". OO design techniques such as the dependency inversion principle can be used to any such

For further discussion on modular monolith's, check out Dan Haywood's pair of articles on InfoQ.

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?

2.3.1. vs MVC server-side

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 (originally scheduled for JavaEE 8 though since removed) 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 Incode Platform modules (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 (non-ASF) Incode Platform's security module provides a full subdomain of users, roles, features and permissions that can be plugged into any Isis application. Similarly, the Incode Platform's command and 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 Incode Platform's 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.

2.3.2. 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.

2.3.3. 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 (obsolete) Isis addons' publishing and the (non-ASF) Incode Platform's 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).

2.3.4. 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 (non-ASF) Incode Platform modules), 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.

2.4. 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 section we explore a

few of them.

2.4.1. 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 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 the (non-ASF) Incode Platform's gmap3, fullcalendar2 and excel Wicket components. 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).

2.4.2. 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.

2.4.3. 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).

2.4.4. 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 Angular 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.

2.4.5. 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. Building Blocks

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

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.xml`).

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(allowsNull=…)`.

In addition, the framework builds up the metamodel for each domain object using layout hints, such as `Customer.layout.xml`. 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.

3.2. Type of Domain Objects

Apache Isis supports recognises four main types of domain classes:

- **domain entities** - domain objects persisted to the database using JDO/DataNucleus; for example `Customer`
- **domain services** - generally singletons, automatically injected, and providing various functionality; for example `CustomerRepository`
- **view models** - domain objects that are a projection of some state held by the database, in support a particular use case; for example `CustomerDashboard` (to pull together commonly accessed information about a customer).
- **mixins** - allow functionality to be "contributed" in the UI by one module to another object,

similar to traits or extension methods provided in some programming languages. This is an important capability to help keep large applications decoupled.

From the end-user's perspective the UI displays a single domain object instance that has state (that is, a domain entity or a view model) per page. The end-user can then inspect and modify its state, and navigate to related objects.

Domain classes are generally recognized using annotations. Apache Isis defines its own set of annotations, while entities are annotated using JDO/DataNucleus (though XML can also be used if required). Apache Isis recognizes some of the JDO and JAXB annotations and infers domain semantics from these annotations. Similarly, JAXB annotations are typically used for view models. There is a smattering of other Java/JEE annotations that are also supported, such as `@javax.inject.Inject` and `@javax.annotation.Nullable`.

The following subsections explain this in further detail.

3.2.1. Domain Entities

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.

Some domain entities are really aggregates, a combination of multiple objects. A commonly cited example of this is an `Order`, which really consists of both a root `Order` entity and a collection of `OrderItem`s. From the end-users' perspective, when they talk of "order" they almost always mean the aggregate rather than just the `Order` root entity.

Eric Evans' Domain Driven Design has a lot to say about aggregate roots and their responsibilities: in particular that it is the responsibility of the aggregate root to maintain the invariants of its component pieces, and that roots may only reference other roots. There's good logic here: requiring only root-to-root relationships reduces the number of moving parts that the developer has to think about.

On the other hand, this constraint can substantially complicate matters when mapping domain layer to the persistenec layer. DDD tends to de-emphasise such matters: it aims to be completely agnostic about the persistence layer, with the responsibilities for managing relationships moved (pretty much by definition) into the domain layer.

As a framework though Apache Isis is less dogmatic about such things. Generally the domain objects are mapped to a relational database and so we can lean on the referential integrity capabilities of the persistence layer to maintain referential invariants. Said another way: we don't tend to require that only roots can maintain roots: we don't see anything wrong in an `InvoiceItem` referencing an `OrderItem`, for example.

Nonetheless the concepts of "aggregate" and "aggregate root" are worth holding onto. You'll likely find that you'll define a repository service (discussed in more detail below) for each aggregate root: for example `Order` will have a corresponding `OrderRepository` service. Similarly, you may also have a factory service, for example `OrderFactory`. However, you are less likely to have a repository service for the parts of an aggregate root: the role of retrieving `OrderItem`s should fall to the `Order` (typically by way of lazy loading of an "items" collection) rather than through an `OrderItemRepository` service.

Again, this isn't a hard-n-fast rule, but a good rule of thumb.

Details on how to actually write a domain entity (the programming model for domain entities) is here.

3.2.2. Domain Services

Domain services are (usually) singleton stateless services that provide additional functionality. 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.

A very common type of domain service is a repository, that is used to look up existing instances of a domain entity. For example, for the `Customer` entity there may be a `CustomerRepository`, while for `Order` entity there may be an `OrderRepository`.

Similarly, entities might also have a corresponding factory service: a `CustomerFactory` or an `OrderFactory`; Evans' Domain Driven Design, draws a clear distinction between a factory (that creates object) and a repository (that is used to find existing objects).

On the other hand, from an end-users' perspective the act of finding an existing object vs creating a new one are quite closely related. For this reason, in Apache Isis it's therefore quite common to have a single domain service that acts as both a factory and a repository (and is usually called just a "repository").

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 services can also be used for a number of other purposes:

- to provide additional non-UI functionality; an example being to perform an address geocoding lookup against the google-maps API, or to perform some calculation, or attach a barcode, send an email etc
- to act as a subscribers to the event bus, potentially influencing events fired by some other module (a key technique for decoupling large applications)

  This is discussed in more detail below, in the section on events.

- to implement an SPI of the Apache Isis framework, most notably cross-cutting concerns such as security, command profiling, auditing and publishing.
- to contribute behaviour or (derived) state to entities/view models.

Mixins can do everything that contributed services can, and have a cleaner programming model. As such, contributed services should be considered a deprecated feature; it may be removed in a future release.

Domain objects of any type (entities, other services, view models, mixins) can also delegate to domain services; domain services are automatically injected into every other domain object. 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.

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.

Details on how to actually write a domain service (the programming model for domain services) is here.

Hexagonal Arch. + services

It's worth extending the Hexagonal Architecture to show where domain services fit in:

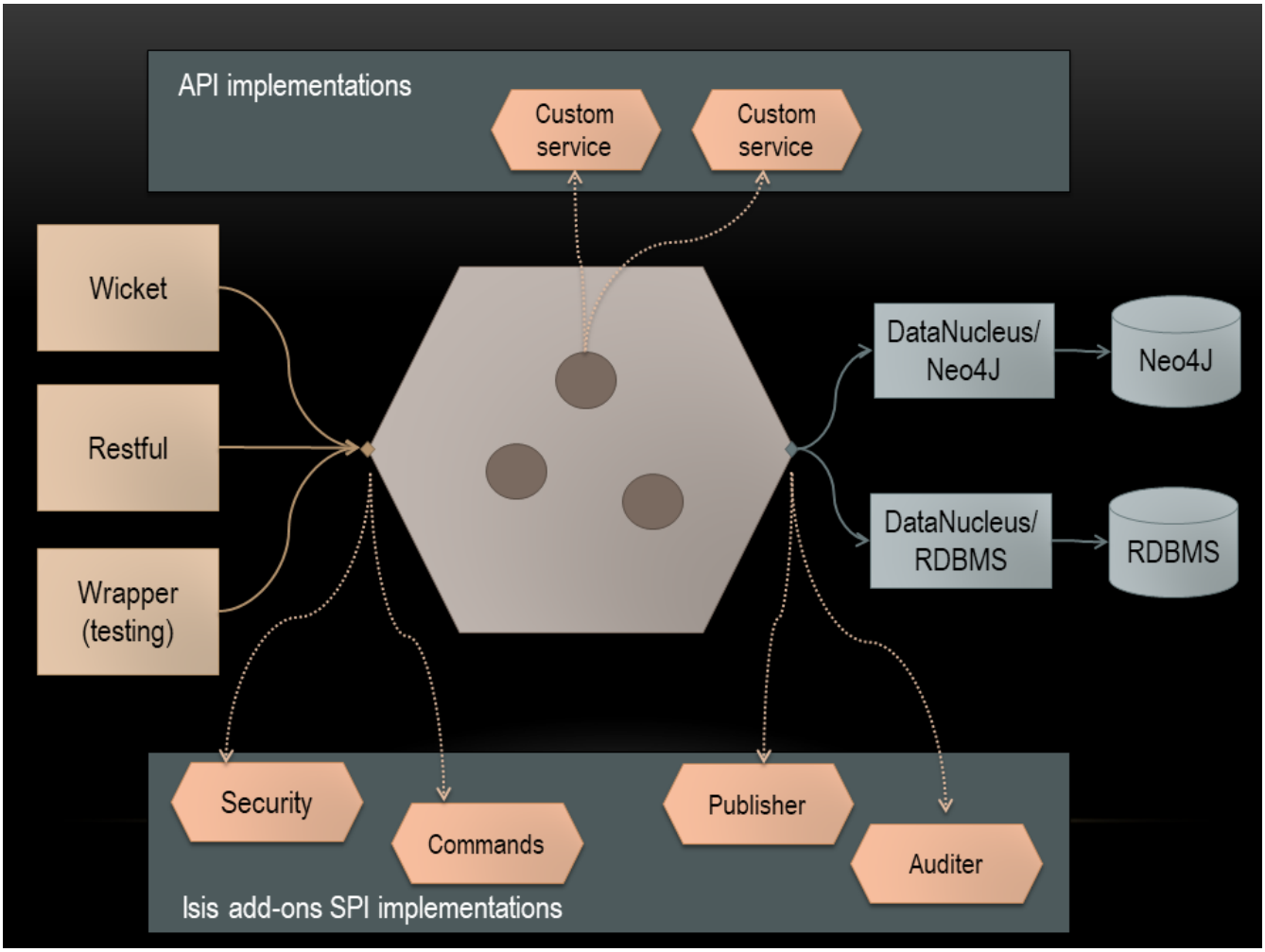


Figure 4. The hexagonal architecture with Isis addons

The (non-ASF) Incode Platform provide SPI implementations of the common cross-cutting concerns. They also provide a number of APIs for domain objects to invoke (not shown in the diagram). You can also write your own domain services as well, for example to interface with some external CMS system, say.

3.2.3. View Models

View models are similar to entities in that (unlike domain services) there can be many instances of any given type. 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, whereas domain entities are mapped to a datastore, view models are not. Instead they are recreated dynamically by serializing their state, ultimately into the URL itself (meaning their state it is in effect implicitly managed by the client browser). You will notice that the URL for view models (as shown in Wicket viewer or RestfulObjects viewer) tends to be quite long.

This capability opens up a number of more advanced use cases:

- In the same way that an (RDBMS) database view can aggregate and abstract from multiple underlying database tables, a view model sits on top of one or many underlying 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).
- view models can also be used to support a particular use case. An example that comes to mind is to expose a list of scanned PDFs to be processed as an "intray", showing the list of PDFs on one side of the page, and the current PDF being viewed on the other. Such view models are part of the application layer, not part of the domain layer (where entities live).

We explore these use cases in more detail below.

Details on how to actually write a view model (the programming model for view models) is here.

Externally-managed entities

Sometimes the entities that make up your application are persisted not in the local JDO/DataNucleus database but reside in some other system, for example accessible only through a SOAP web service. Logically that data might still be considered a domain entity and we might want to associate behaviour with it, however it cannot be modelled as a domain entity if only because JDO/DataNucleus doesn't know about the entity nor how to retrieve or update it.

There are a couple of ways around this: we could either replicate the data somehow from the external system into the Isis-managed database (in which case it is once again just another domain entity), or we could set up a stub/proxy for the externally managed entity. This proxy would hold the reference to the externally-managed domain entity (eg an external id), as well as the "smarts" to know how to interact with that entity (by making SOAP web service calls etc).

The stub/proxy is a type of view model: a view — if you like — onto the domain entity managed by the external system.

DataNucleus does in fact define its own Store Manager extension point, so an alternative architecture would be to implement this interface such that DataNucleus could make the calls to the external system; these externally-persisted domain entities would therefore be modelled as regular `@PersistenceCapable` entities after all. For entities not persisted externally the implementation would delegate down to the default RDBMS-specific `StoreManager` provided by DataNucleus itself.

An implementation that supported only reading from an external entity ought to be comparatively straight-forward, but implementing one that also supported updating external entities would need to carefully consider error conditions if the external system is unavailable; distributed transactions are most likely difficult/impossible to implement (and not desirable in any case).

In-memory entities

As a variation on the above, sometimes there are domain objects that are, conceptually at least entities, but whose state is not actually persisted anywhere, merely held in-memory (eg in a hash).

A simple example might be read-only configuration data that is read from a config file (eg log4j appender definitions) but thereafter is presented in the UI just like any other entity.

Application-layer view models

Domain entities (whether locally persisted using JDO/DataNucleus or managed externally) are the bread-and-butter of Apache Isis applications: the focus after all, should be on the business domain concepts and ensuring that they are solid. Generally those domain entities will make sense to the business domain experts: they form the *ubiquitous language* of the domain. These domain entities are part of the domain layer.

That said, it may not always be practical to expect end-users of the application to interact solely with those domain entities. For example, it may be useful to show a dashboard of the most significant data in the system to a user, often pulling in and aggregating information from multiple points of the app. Obtaining this information by hand (by querying the respective services/repositories) would be tedious and slow; far better to have a dashboard do the job for the end user.

A dashboard object is a model of the most relevant state to the end-user, in other words it is (quite literally) a view model. It is not a persisted entity, instead it belongs to the application layer.

A view model need not merely aggregate data; it could also provide actions of its own. Most likely these actions will be queries and will always ultimately just delegate down to the appropriate domain-layer service/repository. But in some cases such view model actions might also modify state of underlying domain entities.

Another common use for view models is to help co-ordinate complex business processes; for example to perform a quarterly invoicing run, or to upload annual interest rates from an Excel spreadsheet. In these cases the view model might have some state of its own, but in most cases that state does not need to be persisted per se.

Desire Lines

One way to think of application view models is as modelling the "desire line": the commonly-trod path that end-users must follow to get from point A to point B as quickly as possible.

To explain: there are documented examples that architects of university campus will only add in paths some while after the campus buildings are complete: let the pedestrians figure out the routes they want to take. The name we like best for this idea is "desire lines", though it has also been called a "desire path", "paving the path" or "paving the sidewalk".

What that means is you should add view models *after* having built up the domain layer, rather than before. These view models pave that commonly-trod path, automating the steps that the end-user would otherwise have to do by hand.

It takes a little practice though, because even when building the domain layer "first", you should still bear in mind what the use cases are that those domain entities are trying to support. You certainly *shouldn't* try to build out a domain layer that could support every conceivable use case before starting to think about view models.

Instead, you should iterate. Identify the use case/story/end-user objective that will deliver value to the business. Then build out the minimum domain entities to support that use case (refining the ubiquitous language as you go). Then, identify if there any view models that could be introduced which would simplify the end-user interactions with the system (perhaps automating several related use cases together).

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; view models are the tool of choice.

We strongly recommend that you build your applications from the domain layer up, rather than from the view model down.

DTOs

DTOs (data transfer objects) are simple classes that (according to wikipedia) "carry data between processes".

If those two processes are parts of the same overall application (the same team builds and deploys both server and client) then there's generally no need to define a DTO; just access the entities using Apache Isis' RestfulObjects viewer.

On the other hand, if the client consuming the DTO is a different application — by which we mean developed/deployed by a different (possible third-party) team — then the DTOs act as a formal contract between the provider and the consumer. In such cases, exposing domain entities over RestfulObjects would be "A Bad Thing"™ because the consumer would in effect have access to

implementation details that could then not be easily changed by the producer.

To support this use case, a view model can be defined such that it can act as a DTO. This is done by annotating the class using JAXB annotations; this allows the consumer to obtain the DTO in XML format along with a corresponding XSD schema describing the structure of that XML. A discussion of how that might be done using an ESB such as Apache Camel™ follows below.

In case it's not obvious, these DTOs are still usable as "regular" view models; they will render in the Wicket viewer just like any other. In fact (as the programming model section below makes clear), these JAXB-annotated view models are in many regards the most powerful of all the alternative ways of writing view models.

It's also worth noting that it is also possible to download the XML (or XSD) straight from the UI, useful during development. The view model simply needs to implement the `Dto` marker interface; the framework has mixins that contribute the download actions to the view model.

Details of how to consume such DTOs can be found here.

For REST Clients

The Restful Objects viewer automatically provides a REST API for both domain entities. Or, you can use it to only expose view models, taking care to map the state of the domain entity/ies into a view model.

Which beckons the question: when is it safe to safe to expose domain entities directly as REST resources, and when should view models be exposed instead.

If you go searching google you'll find plenty of discussion on this topic (eg here and here). Almost all of these recommend exposing only DTOs (which is to say view models), not domain entities, in REST APIs.

Not so fast, though: the real question is whether the REST API you are exposing is a public API or an internal private API.

- If it's a public API, which is to say that there are third-party clients out over which you have no control, then view models are the way to go.

  In this case view models provide an isolation layer which allow you to modify the structure of the underlying domain entities without breaking this API.

- If it's a private API, which is to say that the only clients of the REST API are under your control, then view models are an unnecessary overhead.

  In this case, just expose domain entities directly.

The caveat to the "private API" option is that private APIs have a habit of becoming public APIs. Even if the REST API is only exposed within your organisation's intranet, other teams may "discover" your REST API and start writing applications that consume it. If that REST API is exposing domain entities, you could easily break those other teams' clients if you refactor.

The Spring Data REST subproject has a similar capability of being able to expose domain entities as REST resources. This SO question, which debates the pros-and-cons, is also worth a read.

If your REST API is intended to be public (or you can't be sure that it will remain private), the exposing view models will entail a lot of marshalling of state from domain entities into view models. There are numerous open source tools that can help with that, for example Model Mapper, Dozer and Orika.

Or, rather than marshalling state, the view model could hold a reference to the underlying domain entity/ies and dynamically read from it (ie, all the view model's properties are derived from the entity's).

A third option is to define an RDBMS view, and then map a "non-durable" entity to that view. The RDBMS view then becomes the public API that must be preserved. The DataNucleus documents describe how to create non-durable views; a "real-world" example can also be found here (in Estatio).

3.2.4. Mixins

The final type of domain object is the **mixin**. These are similar to traits or extension methods in other programming languages, in that they contribute (or rather, mixin) both behaviour or (derived) state to entities/view models.

A mixin object allows one class to contribute behaviour - actions, (derived) properties and (derived) collections - to another domain object, either a domain entity or view model.

The allows the app to stay decoupled, so that it doesn't degrade into the proverbial "big ball of mud". Mixins allow dependencies to be inverted, so that the dependencies between modules can be kept acyclic and under control.

For example, the contributee (eg `Customer`, being mixed into) is in one module, while the contributor mixin (`DocumentHolder_documents`) is in some other module. The `customer` module knows about the `document` module, but not vice versa.

Mixins are also a convenient mechanism for grouping functionality even for a concrete type, helping to rationalize about the dependency between the data and the behaviour. Each mixin is in effect a single behavioural "responsibility" of the domain object.

There are also practical reasons for moving behaviour out of entities even within the same module, because structuring your application this way helps support hot-reloading of Java classes (so that you can modify and recompile your application without having to restart it). This can provide substantial productivity gains.

The Hotspot JVM has limited support for hot reloading; generally you can change method implementations but you cannot introduce new methods. However, the DCEVM open source project will patch the JVM to support much more complete hot reloading support. There are also, of course, commercial products such as JRebel.

The main snag in all this is the DataNucleus enhancer... any change to entities is going to require

the entity to be re-enhanced, and the JDO metamodel recreated, which invariably breaks things. So hot-reloading of an app whose fundamental structure is changing is likely to remain a no-no.

However, chances are that the structure of your domain objects (the data) will change much less rapidly than the behaviour of those domain objects. Thus, it's the behaviour that you're most likely wanting to change while the app is still running. If you move that behaviour out into mixins, then these can be reloaded happily. (And when running in prototype mode), Apache Isis will automatically recreate the portion of the metamodel for any domain object as it is rendered.

Details on how to actually write a mixin (the programming model for mixins) is here.

DCI Architecture

Mixins are an implementation of the DCI architecture architecture, as formulated and described by Trygve Reenskaug and Jim Coplien. Reenskaug was the inventor of the MVC pattern (and also the external examiner for Richard Pawson's PhD thesis), while Coplien has a long history in object-orientation, C++ and patterns.

DCI stands for Data-Context-Interaction and is presented as an evolution of object-oriented programming, but one where behaviour is bound to objects dynamically rather than statically in some context or other. The mixin pattern is Apache Isis' straightforward take on the same basic concept.

You might also wish to check out Apache Zest (formerly Qi4J), which implements a much more general purpose implementation of the same concepts.

3.3. Identifiers

The Apache Isis framework actively tracks the identity of each domain object. This identity is represented to the end-user in human-readable form so that they know which object they are interacting with, and is also used and is available internally/for integrations.

This section explores these two related concepts.

3.3.1. Title and Icon

To allow the end-user to distinguish one domain object from another, it is rendered with a title and an icon. The icon informally identifies the type of the domain object, while the title identifies the instance.

Title

The title of a domain object 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 title is not formally required to be a unique identify the object within its type, but it needs to be "unique enough" that a human user is able to distinguish one instance from another.

The title is usually just a simple string, but the framework also allows for the title to be translated into different locales.

Icon

Sometimes it's helpful for the icon to represent more than just the object's type; it might also indicate the state of an object. For example, a shipped `Order` might have a slightly different icon to a yet-to-be-shipped `Order`; or an overdue `Loan` might be distinguished separately from a

CSS Class

In addition to the title and icon, it is also possible for a domain 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.

Details on how to actually write titles, icons and CSS classes can be found here.

3.3.2. 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 `AuditerService` or `AuditingService` (the latter deprecated);
- similarly within implementations of `CommandService` to persist `Command` objects
- similarly within implementations of `PublisherService` 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.

3.4. 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.

3.4.1. 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*.

3.4.2. 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.

3.4.3. 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.

3.5. Events

When the framework renders a domain object, and as the end-user interacts with the domain object, the framework it emits multiple events using the intra-process event bus. These events enable other domain services (possibly in other modules) to influence how the domain object is rendered, or to perform side-effects or even veto an action invocation.

It is also possible to simulate the rendering of a domain object by way of the `WrapperFactory`. This allows business rules to be enforced for programmatic interactions between objects.

To receive the events, the domain service should subscribe to the `EventBusService`, and implement an appropriately annotated method to receive the events.

The framework has several categories of events: domain events, UI events and lifecycle events. These are explored in the sections below.

3.5.1. Domain Events

Domain events are fired — through the internal event bus — for every user interaction with each object member (property, collection or action).

By default, rendering a property causes a `PropertyDomainEvent` to be fired, though the `@Property#domainEvent()` attribute allows a custom subclass to be specified if necessary. Similarly, rendering a collection causes a `CollectionDomainEvent` to be fired, and rendering an action causes an `ActionDomainEvent` to be fired.

In fact, each event can be fired up to five times, with the event's `getEventPhase()` method indicating to the subscriber the phase:

- **hide** phase allows the subscriber to hide the member
- **disable** phase allows the subscriber to disable the member.

For a property this makes it read-only; for an action this makes it "greyed out". (Collections are implicitly read-only).

- **validate** phase allows the subscriber to validate the proposed change.

For a property this means validating the proposed new value of the property; for an action this means validating the action parameter arguments. For example, a referential integrity restrict could be implemented here.

- **executing** phase is prior to the actual property edit/action invocation, allowing the subscriber to perform side-effects.

For example, a cascade delete could be implemented here.

- **executed** phase is after the actual property edit/action invocation.

  For example, a business audit event could be implemented here.

For more details on the actual domain event classes, see the domain event section of the relevant reference guide.

3.5.2. UI Events

As explained earlier, to allow the end-user to distinguish one domain object from another, it is rendered with a title and an icon.

Normally the code to return title and icon of an object is part of the domain object's implementation. However, UI events allow this title and icon to be provided instead by a subscriber. UI events have higher precedence than the other mechanisms of supplying a title.

If annotated with `@DomainObjectLayout#titleUiEvent()`, the appropriate (subclass of) `TitleUiEvent`

will be emitted. Similarly for `iconUiEvent()` and `cssClassUiEvent()`.

This can be particularly useful for JAXB-style view models which are used as DTOs and so must have no dependencies on the rest of the Apache Isis framework.

It can also be more generally useful to allow one module to influence/fine-tune the title of entities provided by some other module. The obvious use case is when reusing library modules, eg as provided by (non-ASF) Incode Platform.

3.5.3. Lifecycle Events

Lifecycle events allow domain object subscribers to listen for changes to the persistence state of domain entities, and act accordingly.

Lifecycle events are *not* fired for view models.

The lifecycle events supported are:

- object created - just instantiated.

  Note that this requires that the object is instantiated using the framework, see here for further discussion

- updated loaded - just retrieved from the database
- object persisting - object about to be inserted into the database
- object persisted - object just inserted into the database
- object updating - object about to be updated
- object updated - object just updated
- object removing - object about to be deleted from the database

There is no lifecycle event for object creating because the framework doesn't know about newly created objects until they have been created; and there is no lifecycle event for obejcts removed because it is not valid to "touch" a domain entity once deleted.

For example, if annotated with `@DomainObjectLayout#updatingLifecycleEvent()`, the appropriate (subclass of) `ObjectUpdatingEvent` will be emitted. Similarly for `iconUiEvent()` and `cssClassUiEvent()`.

3.6. Modules

We tend to use Maven modules as a way to group related domain objects together; we can then reason about all the classes in that module as a single unit. By convention there will be a single top-level package corresponding to the module.

For example, the (non-ASF) Incode Platform's document module has a top-level package of `org.incode.module.document`. Within the module there may be various subpackages, but it's the module that defines the namespace.

The bootstrapping of Apache Isis also relies on module classes. (Currently) the only role of these classes is to identify a fully qualified package name, for example `org.incode.modules.document`. The framework then performs classpath scanning across all such packages to locate any domain entities provided by that module (though some modules have no entities), all domain services provided by the module (every module is likely to define at least one), and also any fixture scripts provided by the module.

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

The module class does not *need* to implement any specific interface; in this case the framework uses the module class simply to obtain a package to scan for entities and domain services. However, it's recommended to implement the `Module` interface because this allows the module to specify its direct dependencies; from these all transitive dependencies are calculated. This should match the compile-time dependencies declared within Maven.

The `Module` interface also allows a module to declare any configuration properties that should be contributed, as well as fixture scripts for reference data, and to tear down (for integration testing).

Chapter 4. HelloWorld Archetype

The quickest way to start learning about Apache Isis is to run the `helloworld` archetype. This will generate a tiny Apache Isis app, consisting of a simple one-class domain model, called `HelloWorldObject`, and a supporting `HelloWorldObjects` domain service. Both the business logic and supporting bootstrapping classes are in a single Maven module (in different Java packages).

We don't recommend that you use the helloworld archetype as the basis for your own applications. Instead, use the simpleapp archetype. This also creates a minimal application, but provides more structure and example tests, useful as you build out your own app.

4.1. 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.x

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.

4.2. 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=helloworld-archetype \
    -D archetypeVersion=1.15.1 \
    -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.

4.3. Structure of the App

As noted above, the application generated by the helloworld archetype is deliberately simplified, with everything contained within a single Maven module.

Under `src/main/java` we have:

```
domainapp/
 ├──── application/
 │     ├──── HelloWorldAppManifest.java
 │     └──── isis.properties
 ├──── dom/
 │     ├──── HelloWorldModule.java
 │     └──── impl/
 │          ├──── HelloWorldObject.java
 │          ├──── HelloWorldObject.layout.xml
 │          ├──── HelloWorldObject.png
 │          └──── HelloWorldObjects.java
 └──── app/
      ├──── HelloWorldApplication.java
      └──── welcome.html
META-INF/
 └──── persistence.xml
```

For simplicity, all the Java source files generated by the archetype are placed in a `domainapp` top-level package.

While it's more conventional to use the inverse domain name for package (eg `com.mycompany.myapp`, that's only really appropriate for library code that will be released for reuse by multiple applications in different orgnisations (eg open source).

For internal application though this is less of a concern; indeed, avoiding using the domain name means that if the company rebrands or is taken over then nothing needs be changed.

Of course, you are always free to move the classes to a different package if you wish.

In `domainapp.application` package is `HelloWorldAppManifest`, which implements the `AppManifest` interface defined by the Apache Isis applib. The class is only small, so is worth showing here in its entirety:

```
public class HelloWorldAppManifest extends AppManifestAbstract {
    public static final Builder BUILDER = Builder
            .forModules(HelloWorldModule.class)
            .withConfigurationPropertiesFile(HelloWorldAppManifest.class,
"isis.properties")
            .withAuthMechanism("shiro");

    public HelloWorldAppManifest() {
        super(BUILDER);
    }
}
```

Rather than implement `AppManifest` directly, `HelloWorldAppManifest` uses the builder provided by the convenience `AppManifestAbstract`); as you'll find when you work on bigger applications, it's common to have variations around the app manifest, so the builder makes it easy to create these variations without lots of boilerplate.

For now, we see that (using the builder) the app manifest defines three things:

- a list of modules - there's just one, `HelloWorldModule` for this class
- the location of a configuration file, `isis.properties`, read in as a resource from the classpath
- specifying an authentication/authorisation mechanism, in this case Apache Shiro integration.

In Apache Isis a module is simply an empty class, which is used simply to obtain a package name. The framework uses classpath scanning to find certain classes; rather than scan the entire classpath it limits its search to the package(s) obtained from the modules.

The packages are scanned to identify three types of classes:

- all entities.

  These are entities that are annotated with `@javax.jdo.annotations.PersistenceCapable`. These are passed through to the JDO (DataNucleus) object store, in order to create database mappings from the entities to relational tables

  In the helloworld application, the only entity is `HelloWorldObject`.

- all domain services

  These are classes that are annotated with the framework's `@DomainService` annotation. Depending on their nature, services are used to build up the menu, or are available to call programmatically, eg repositories. The framework instantiates an instance of each and will automatically inject the services into other domain objects and services.

  In the helloworld application, the only domain service is `HelloWorldObjects`. This appears in the menu, and also acts as a repository for the `HelloWorldObject` entity.

- all fixture scripts

These are classes that extend from the applib `FixtureScript` class, and are used to setup the database when running in prototype mode (against an in-memory database).

The helloworld application doesn't provide any examples of these.

The app manifest also identifies the `isis.properties` file (in the same package as `HelloWorldAppManifest`) as containing various configuration options. The helloworld application uses these for settings that are unlikely to change and so are loaded as a static resource from the classpath.

You'll find that there's another file called `isis.properties` that's in `WEB-INF/isis.properties`. This also provides configuration options (the framework simply combines them) but those in `WEB-INF/isis.properties` are restricted to settings that are likely to change from environment to environment, most notably JDBC URL connection strings. Separating these out makes it easy to reconfigure the application to run against different databases in different environments (dev, test, production etc).

Finally, the app manifest identifies Apache Shiro for authentication and authorisation. Shiro in turn is configured using the `WEB-INF/shiro.ini` file.

The security integration provided by Apache Isis and Shiro is quite sophisticated; to get started though you can just login using username: `sven`, password: `pass`.

In the `domainapp.dom` module ("dom" stands for "domain object model") is the `HelloWorldModule`:

```
package domainapp.dom;
public final class HelloWorldModule {
    private HelloWorldModule(){}
}
```

As already explained, this is simply used by the app manifest to identify "domainapp.dom" as the package to scan to locate entities (`HelloWorldObject`), services (`HelloWorldObjects`) and fixture scripts (none provided).

In the `domainapp.dom.impl` we have the classes that actually comprise our domain object. These are a little large to list here in their entirety, but it's worth calling out:

- `HelloWorldObject`:
  - is annotated with a bunch of JDO annotations, `@PersistenceCapable` being the most important.

    This annotation is what identifies the class as an entity, so that Apache Isis passes through to JDO/DataNucleus during bootstrapping (to create the ORM mappings).

  - is also annotated with Isis' own `@DomainObject`.

    This isn't mandatory, but since entities are the real building blocks on which Isis applications are built, it's very common to use this annotation.

    - also uses various Project Lombok annotations to remove boilerplate.

- `HelloWorldObject.layout.xml` defines the layout of the members (properties and actions) of the `HelloWorldObject`.

  Layout files are optional - Apache Isis is an implementation of the naked objects pattern, after all - but in most cases you'll find it easiest to use one

- `HelloWorldObject.png` is used as an icon for any instances of the domain object shown in the (Wicket) viewer
- `HelloWorldObjects` is a domain service by virtue of the fact that it is annotated with Isis' `@DomainService`.

  This acts as both a menu (the `@DomainService#nature` attribute) and also a repository to find/create instances of `HelloWorldObject`.

The `META-INF/persistence.xml` also relates to domain entities. JDO/DataNucleus allows metadata mapping information to be specified using either XML or annotations. In the Apache Isis community we generally prefer to use annotations, but nevertheless this file is required, even though it is basically empty:

```
<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="helloworld"/>
</persistence>
```

Finally, in the `domainapp.webapp` we have `HelloWorldApplication`. This is required to bootstrap the Wicket viewer (it is configured in `WEB-INF/web.xml`). Internally it uses Google Guice to configure various static resources served up by Wicket:

```
public class HelloWorldApplication extends IsisWicketApplication {
    ...
    protected Module newIsisWicketModule() {
        final Module isisDefaults = super.newIsisWicketModule();
        final Module overrides = new AbstractModule() {
            @Override
            protected void configure() {
                ...
            }
        };
        return Modules.override(isisDefaults).with(overrides);
    }
}
```

The `configure()` method is the place to change the name of the application for example, or to change the initial about and welcome messages. The text of the welcome page shown by the Wicket viewer can be found in `welcome.html`, loaded from the classpath and in the same package as `HelloWorldApplication`.

Under `src/main/webapp` we have various resources that are used to configure the webapp, or that are served up by the running webapp:

```
about/
 └──── index.html
css/
 └──── application.css
scripts/
 └──── application.js
swagger-ui/
WEB-INF/
 ├──── isis.properties
 ├──── logging.properties
 ├──── shiro.ini
 └──── web.xml
```

Most important of these is `WEB-INF/web.xml`, which bootstraps both the Wicket viewer and the Restful Objects viewer (the REST API derived from the domain object model).

The `about/index.html` is the page shown at the root of the package, providing links to either the Wicket viewer or to the Swagger UI. In a production application this is usually replaced with a page that does an HTTP 302 redirect to the Wicket viewer.

In `css/application.css` you can use to customise CSS, typically to highlight certain fields or states. The pages generated by the Wicket viewer have plenty of CSS classes to target. You can also implement the `cssClass()` method in each domain object to provide additional CSS classes to target.

Similarly, in `scripts/application.js` you have the option to add arbitrary Javascript. JQuery is available by default.

In `swagger-ui` is a copy of the Swagger 2.x UI classes, preconfigured to run against the REST API exposed by the Restful Objects viewer. This can be useful for developing custom applications, and is accessible from the initial page (served up by `about/index.html`).

Finally in `WEB-INF` we have the standard `web.xml` (already briefly discussed) along with several other files:

- `isis.properties` contains further configuration settings for Apache Isis itself.

  (As already discussed), these are in addition to the configuration properties found in the `isis.properties` that lives alongside and that is loaded by the `HelloWorldAppManifest` class. Those in the WEB-INF/isis.properties file are those that are likely to change when running the application in different environments.

- `logging.properties` configures log4j.

The framework is configured to use slf4j running against log4j.

- `shiro.ini` configures Apache Shiro, used for security (authentication and authorisation)
- `web.xml` configures the Wicket viewer and Restful Objects viewer. It also sets up various filters for serving up static resources with caching HTTP headers.

Under `src/test/java` we have:

```
domainapp/
└──── dom/
    └──── impl/
        ├──── HelloWorldObjectTest_delete.java
        └──── HelloWorldObjectTest_updateName.java
```

These are very simple unit tests of `HelloWorldObject`. They use JMock as the mocking library (with some minor extensions provided by Apache Isis itself).

Finally, at the root directory we of course have the `pom.xml`. Some notable points:

- since this module generates a WAR file, the `pom.xml` uses `<packaging>war</packaging>`
- maven mixins are used to remove boilerplate configuration of standard plugins (resources, compile, jar etc), for the DataNucleus enhancer, for surefire (tests), and for running the application using jetty plugin

Now you know your way around the code generated by the archetype, lets see how to build the app and run it.

4.4. 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.

4.5. Running the App

The `helloworld` 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.

4.5.1. 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: (using maven 3.5.0 / isis 1.15 there is no webapp module and there is no need to `cd`into a `webapp` module. just run mvn jetty:run to fire up the app)

```
mvn jetty:run
```

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

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

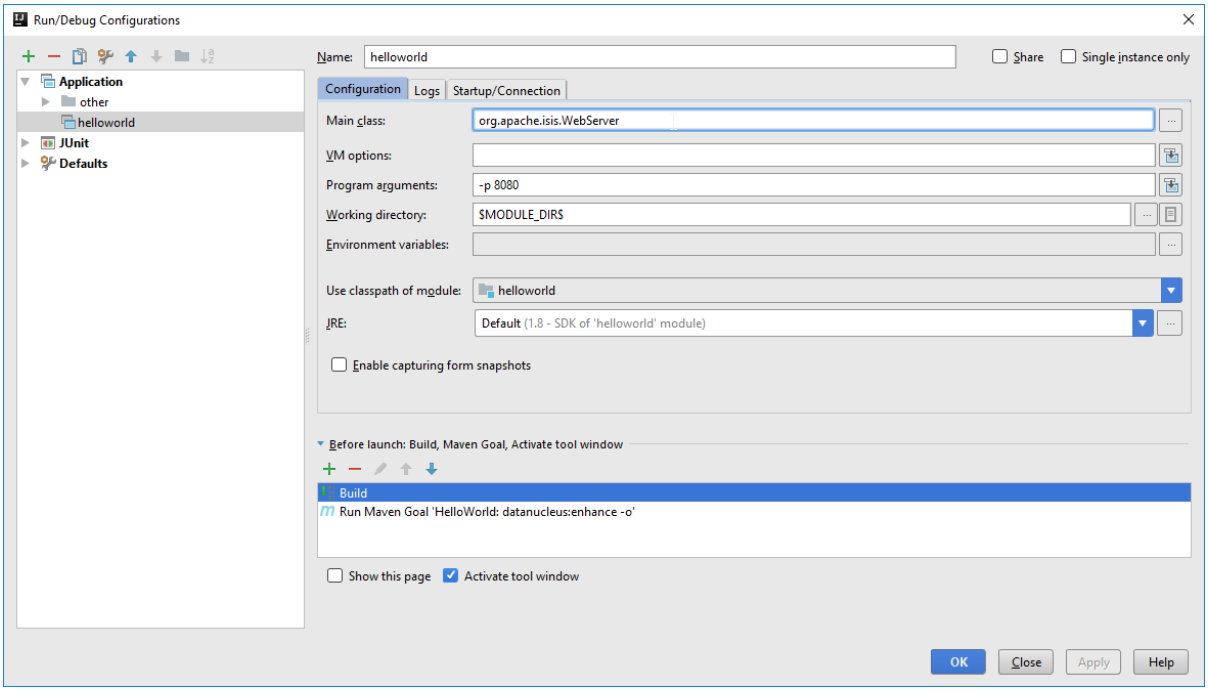
4.5.2. Using a regular servlet container

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.

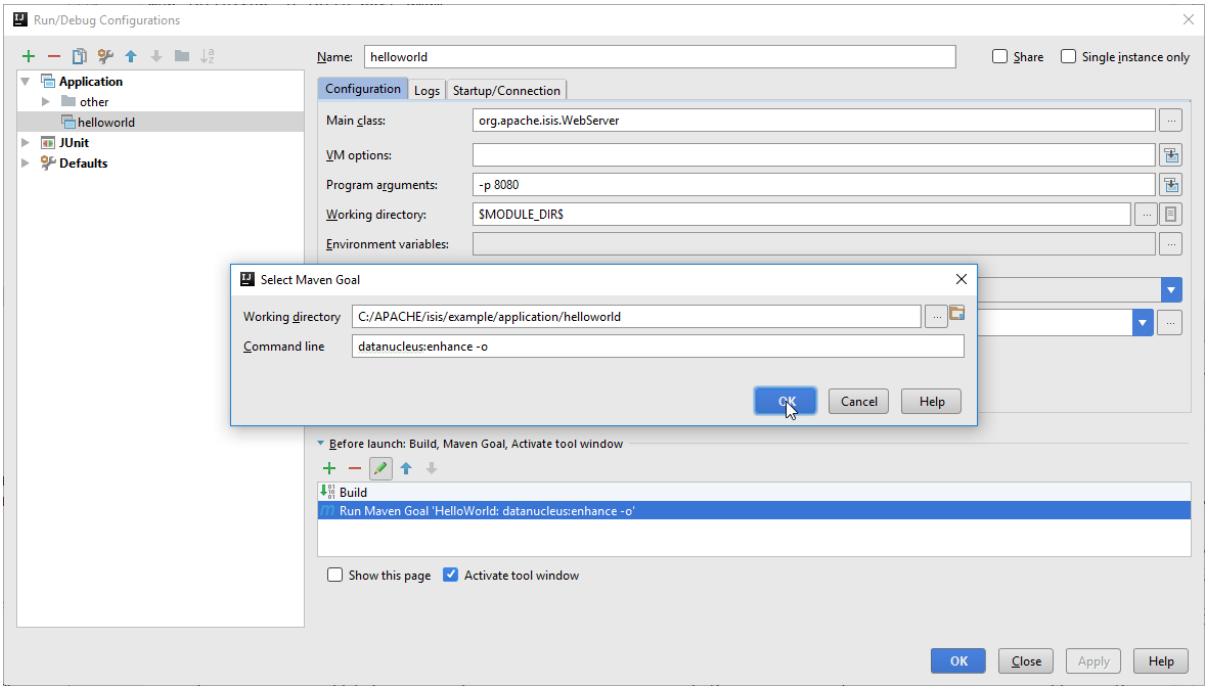
4.5.3. From within the IDE

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:



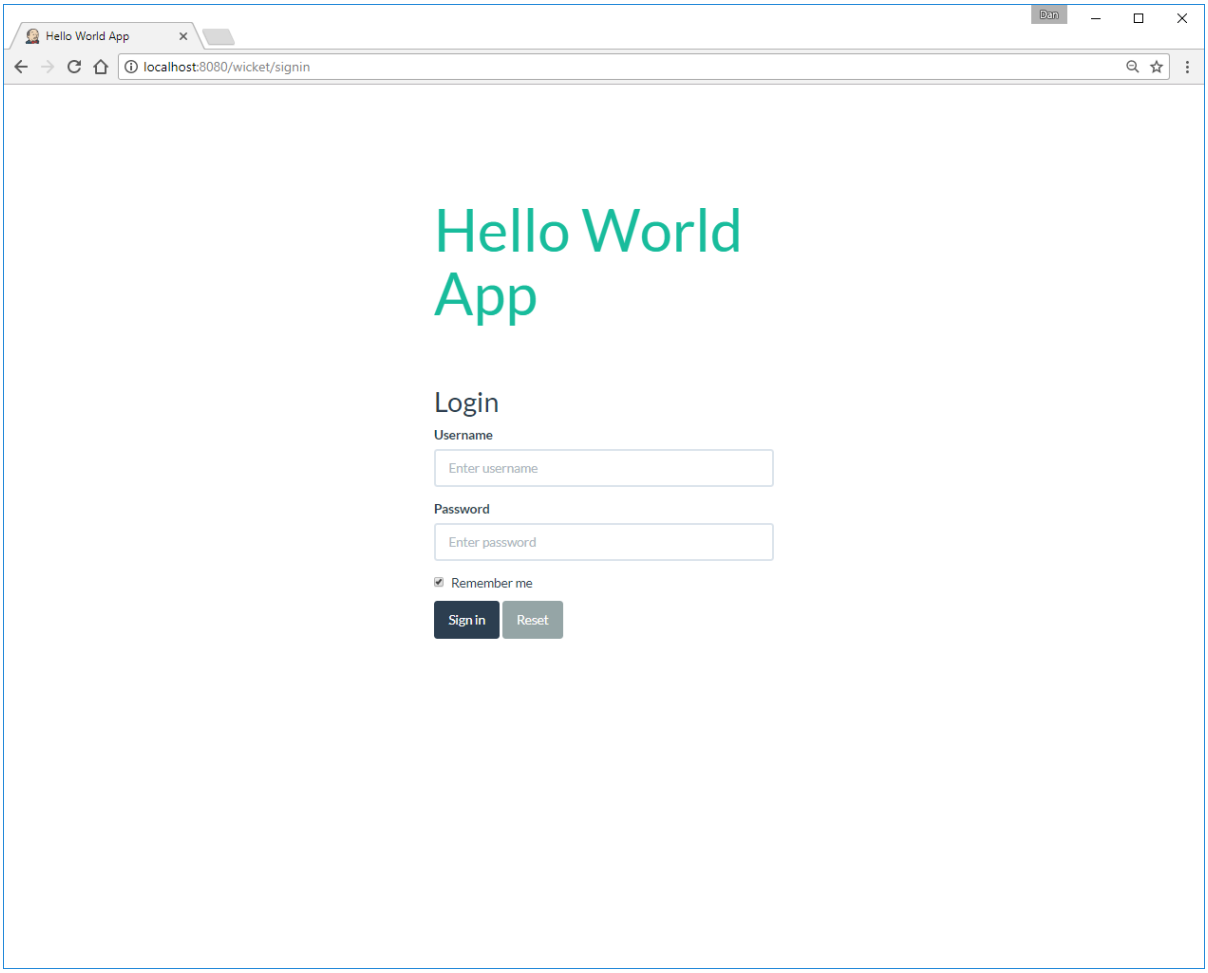
with the maven goal to run the DataNucleus enhancer (discussed in more detail here) before launch defined as:



4.6. Using the App

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:

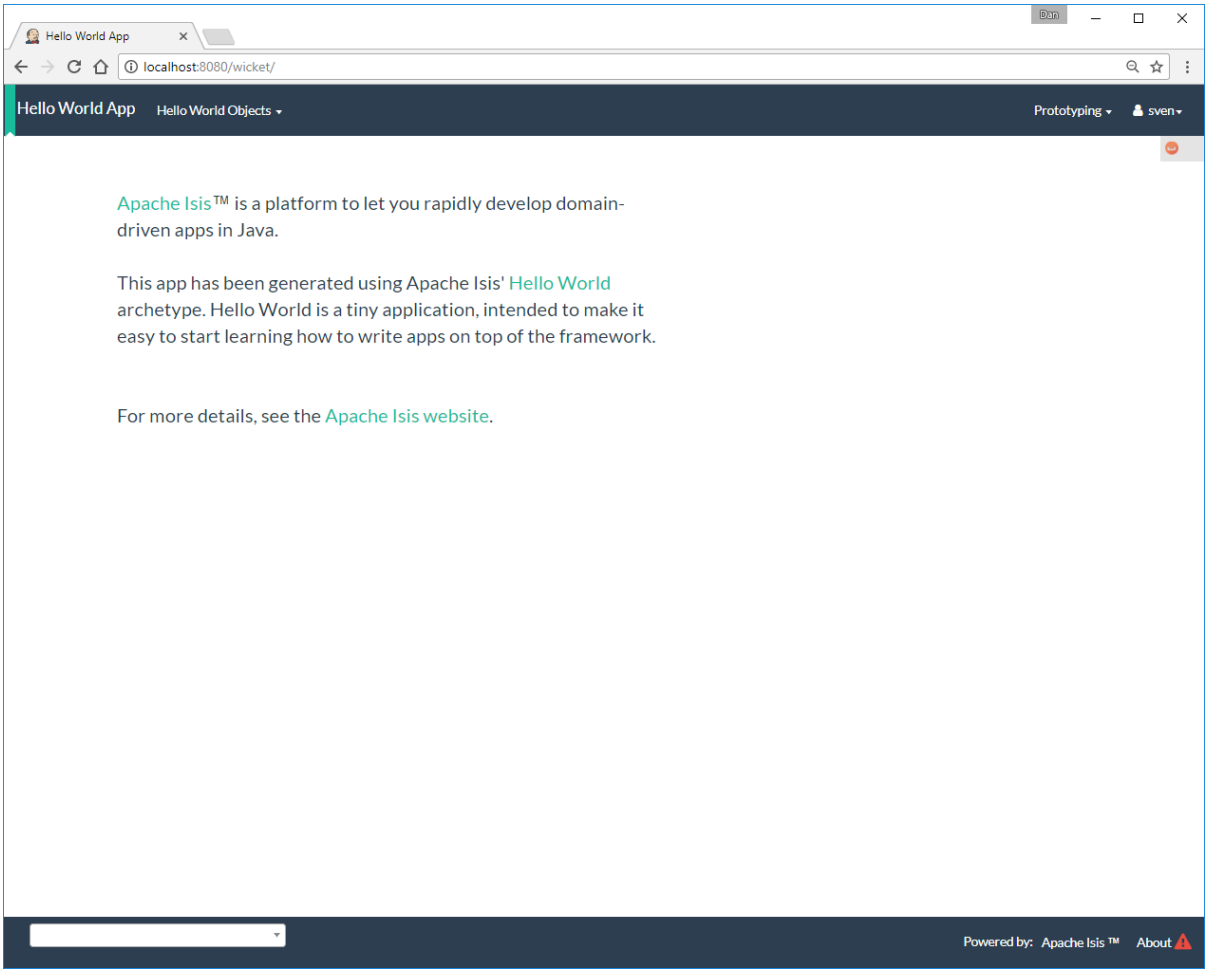
The Wicket viewer provides a human usable web UI (implemented, as you might have guessed from its name, using Apache Wicket), so choose that and navigate to the login page:



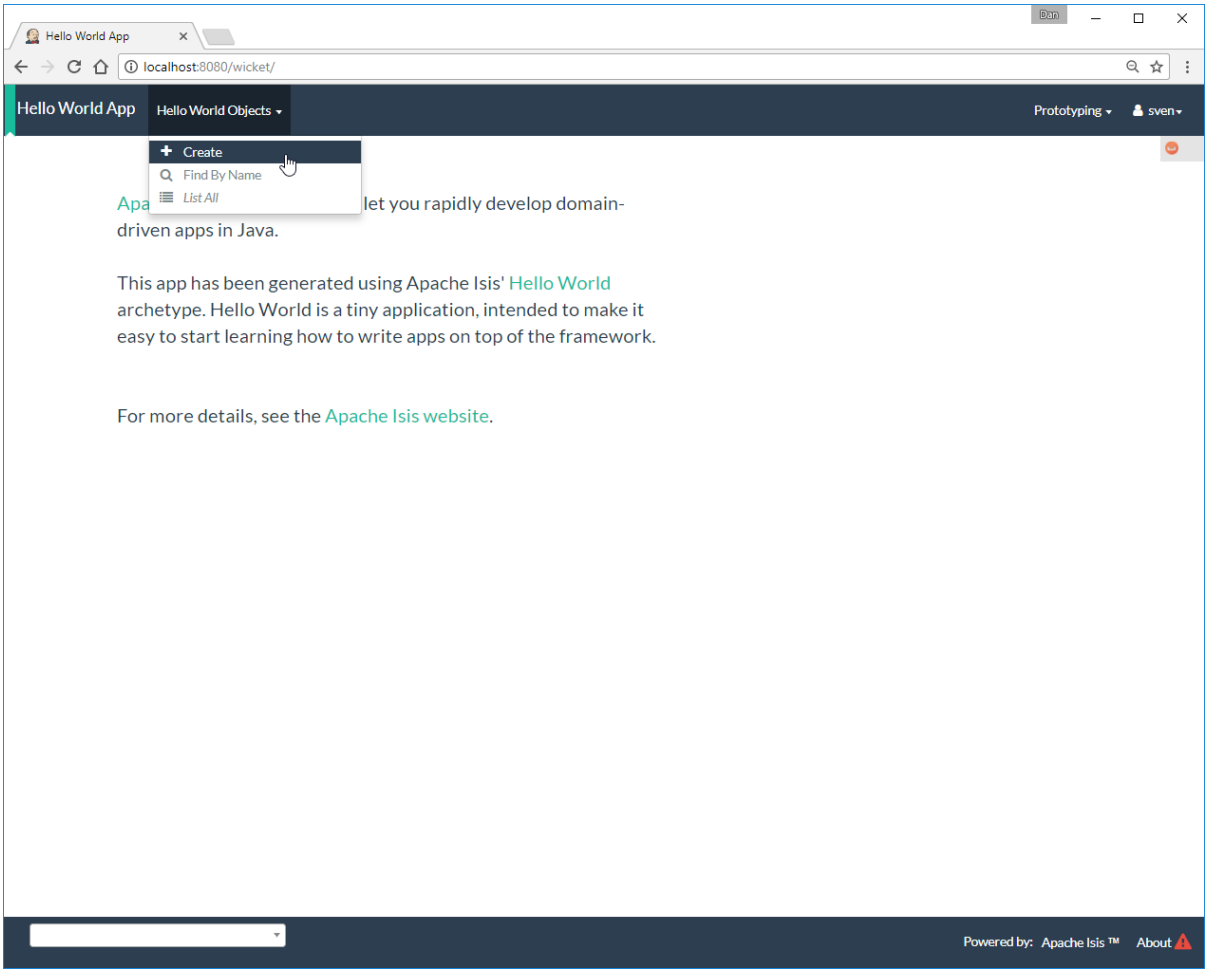
The app itself is configured to run using shiro security, as configured in the `WEB-INF/shiro.ini` config file. You can login with:

- username: *sven*
- password: *pass*

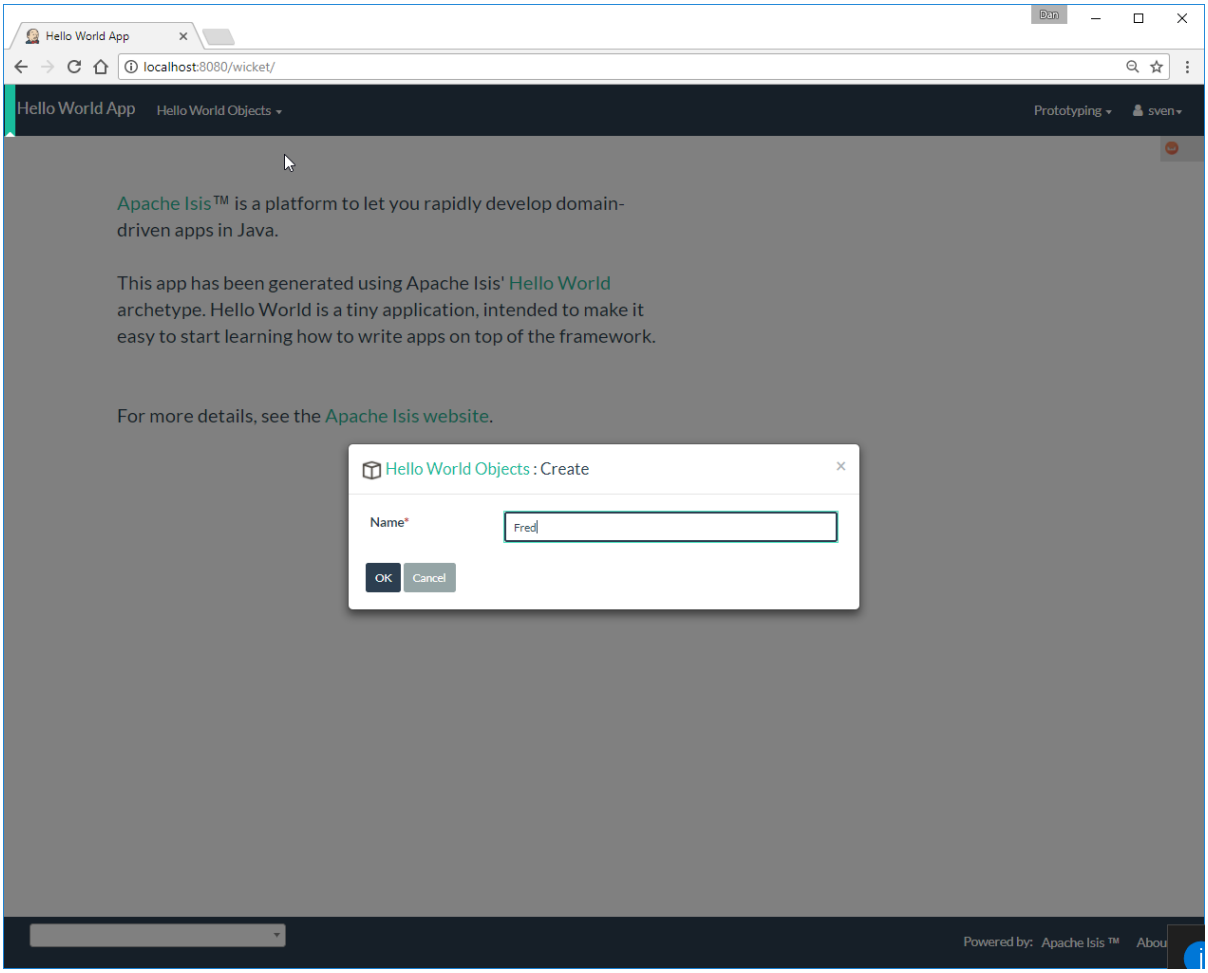
Once you've logged in you'll see the default home page:



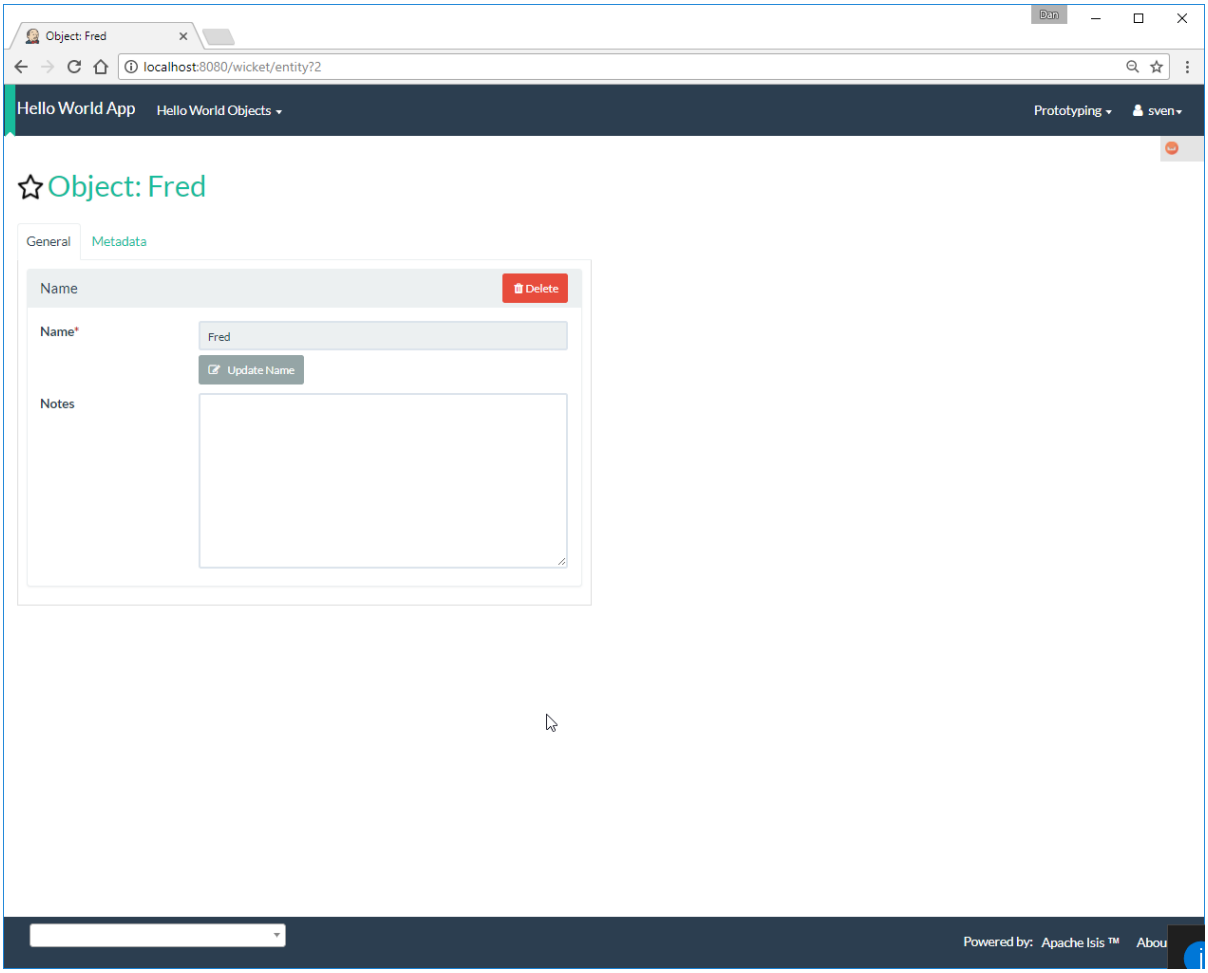
The application is configured to run with an in-memory database, so initially there is no data. Create an object using the menu:



which brings up a modal dialog:



hitting OK returns the created object:



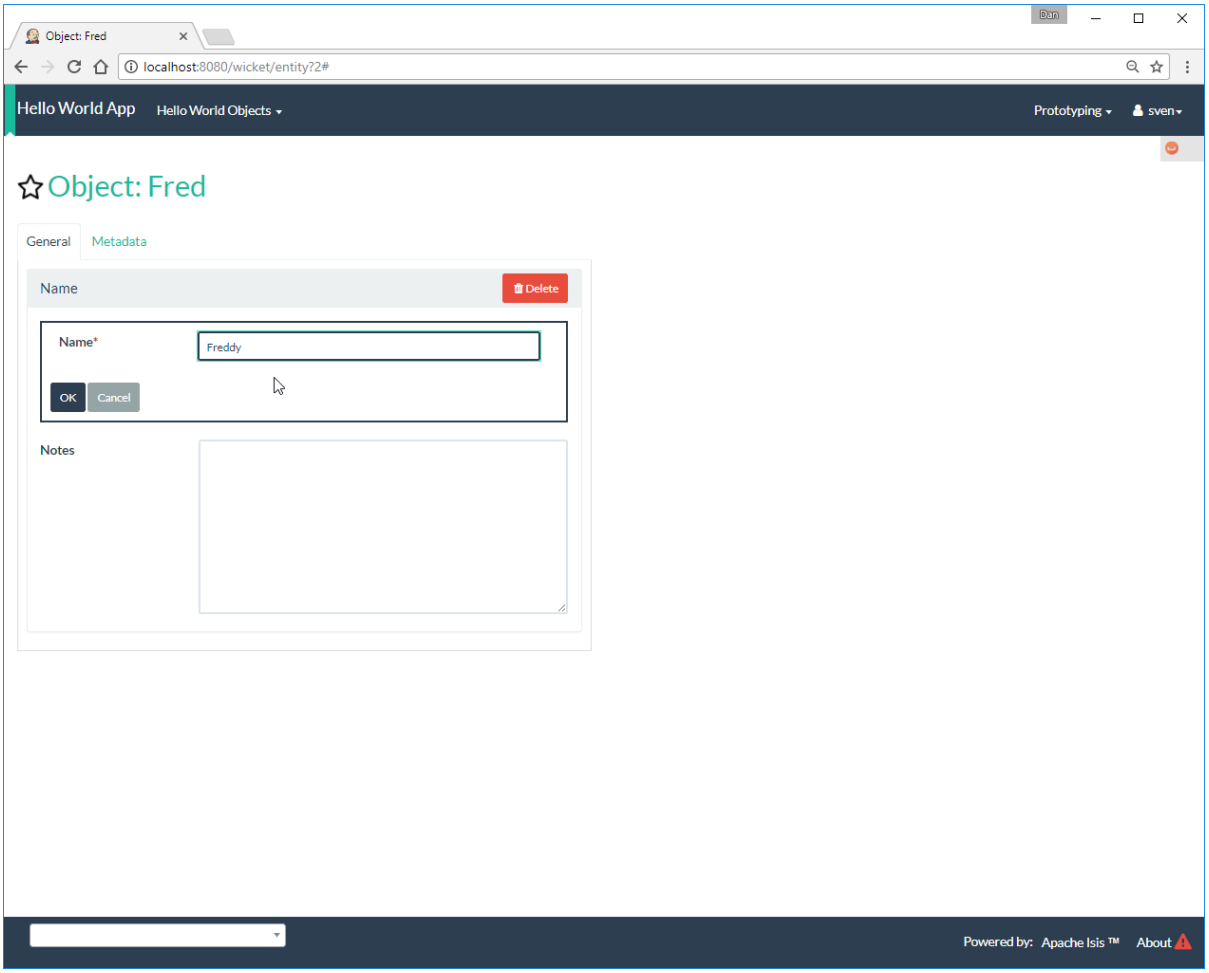
The above functionality is implemented by this code:

```
@Action(semantics = SemanticsOf.NON_IDEMPOTENT)
@MemberOrder(sequence = "1")
public HelloWorldObject create(
        @Parameter(maxLength = 40)
        @ParameterLayout(named = "Name")
        final String name) {
    final HelloWorldObject object = new HelloWorldObject(name);
    serviceRegistry.injectServicesInto(object);
    repositoryService.persist(object);
    return object;
}
```

The `HelloWorldObject` contains a couple of properties, and a single action to update that property.

- The `name` property is read-only, and can only be modified using the `updateName` action.

  For example:

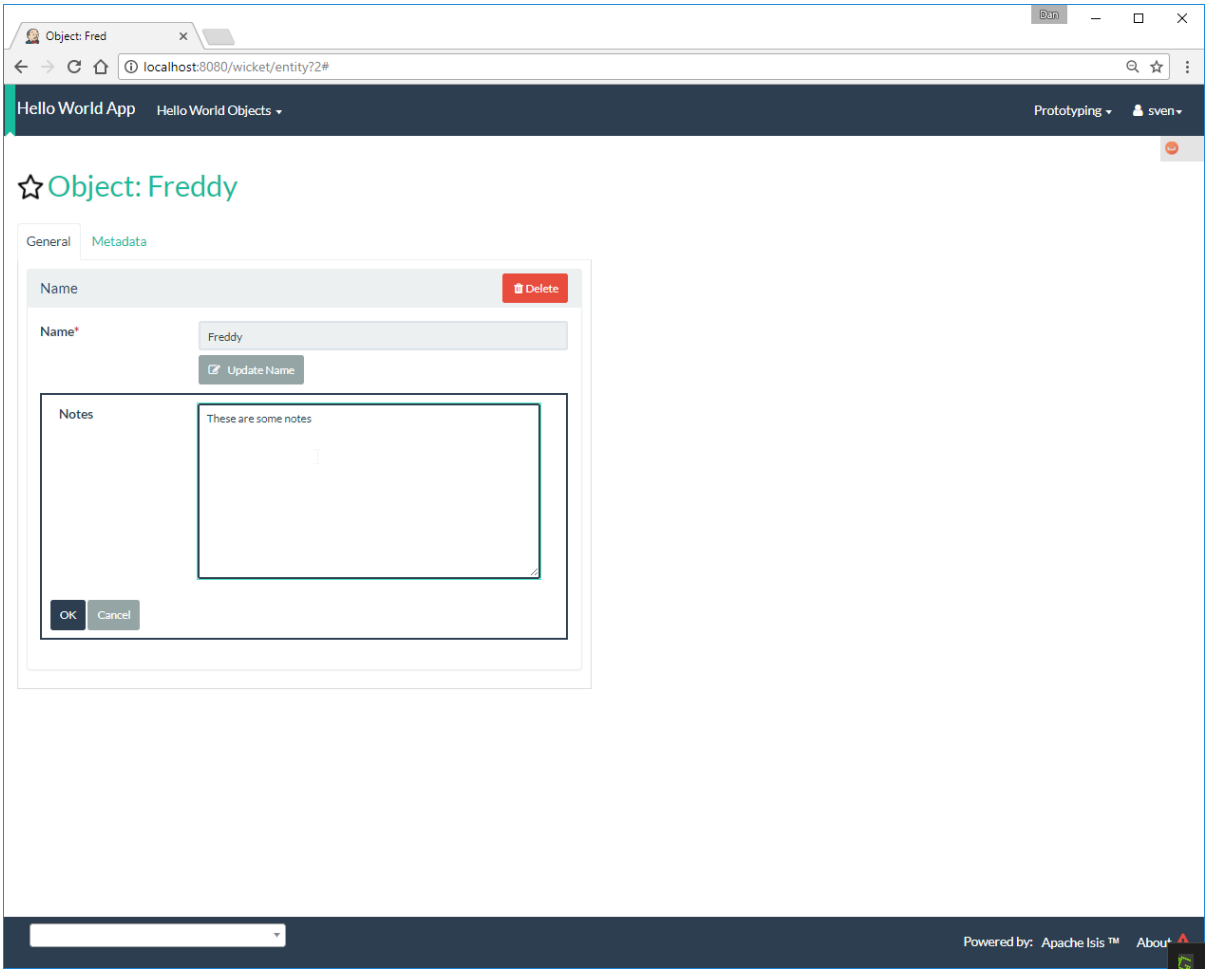


The above functionality is implemented by this code:

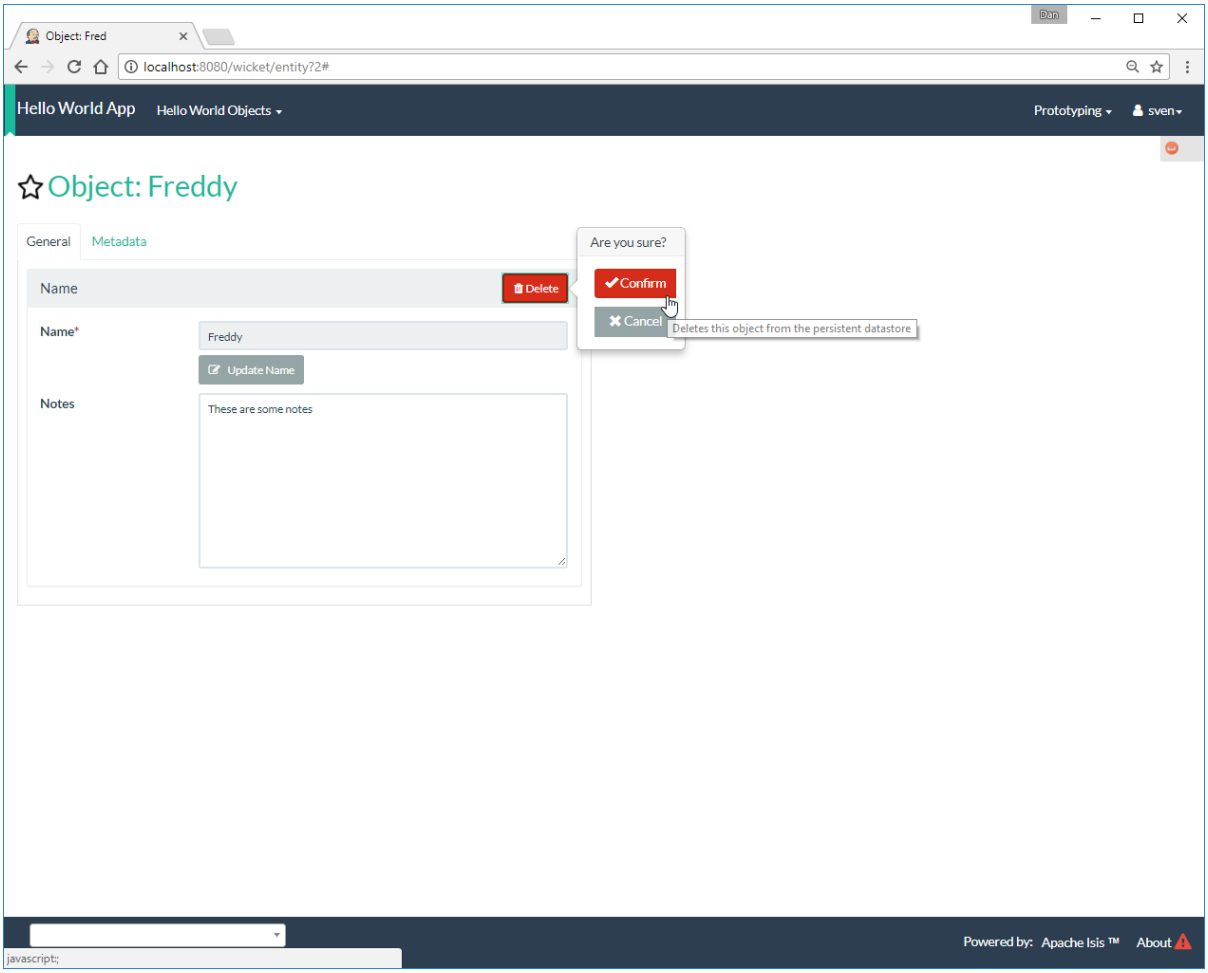
```
@Action(
    semantics = SemanticsOf.IDEMPOTENT,
    command = CommandReification.ENABLED,
    publishing = Publishing.ENABLED
)
public HelloWorldObject updateName(
        @Parameter(maxLength = 40)
        @ParameterLayout(named = "Name")
        final String name) {
    setName(name);
    return this;
}
```

- The `notes` property is editable, and can be edited in-place.

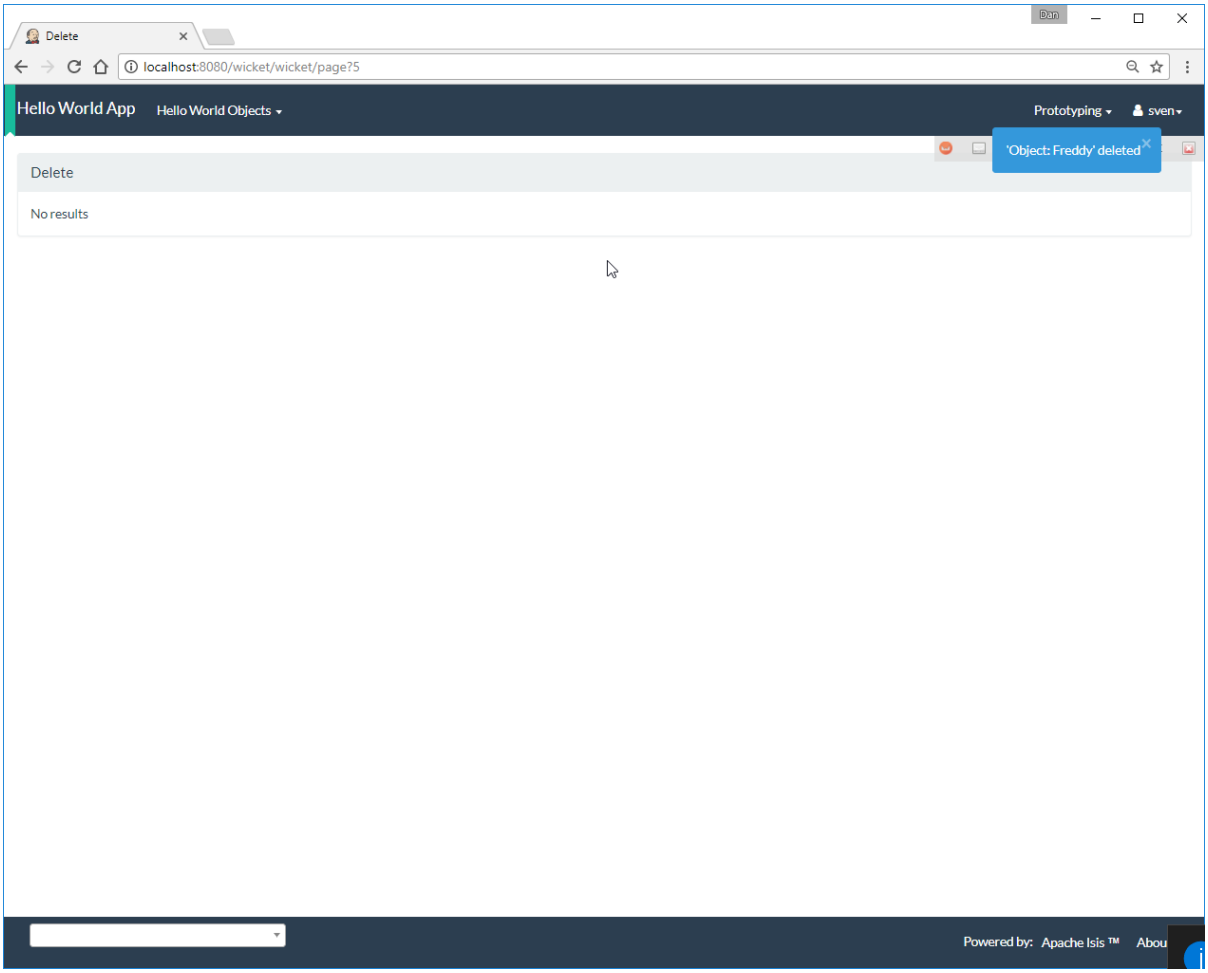
  For example:



It’s also possible to delete an object:



The viewer displays a message confirming that the object has been deleted:

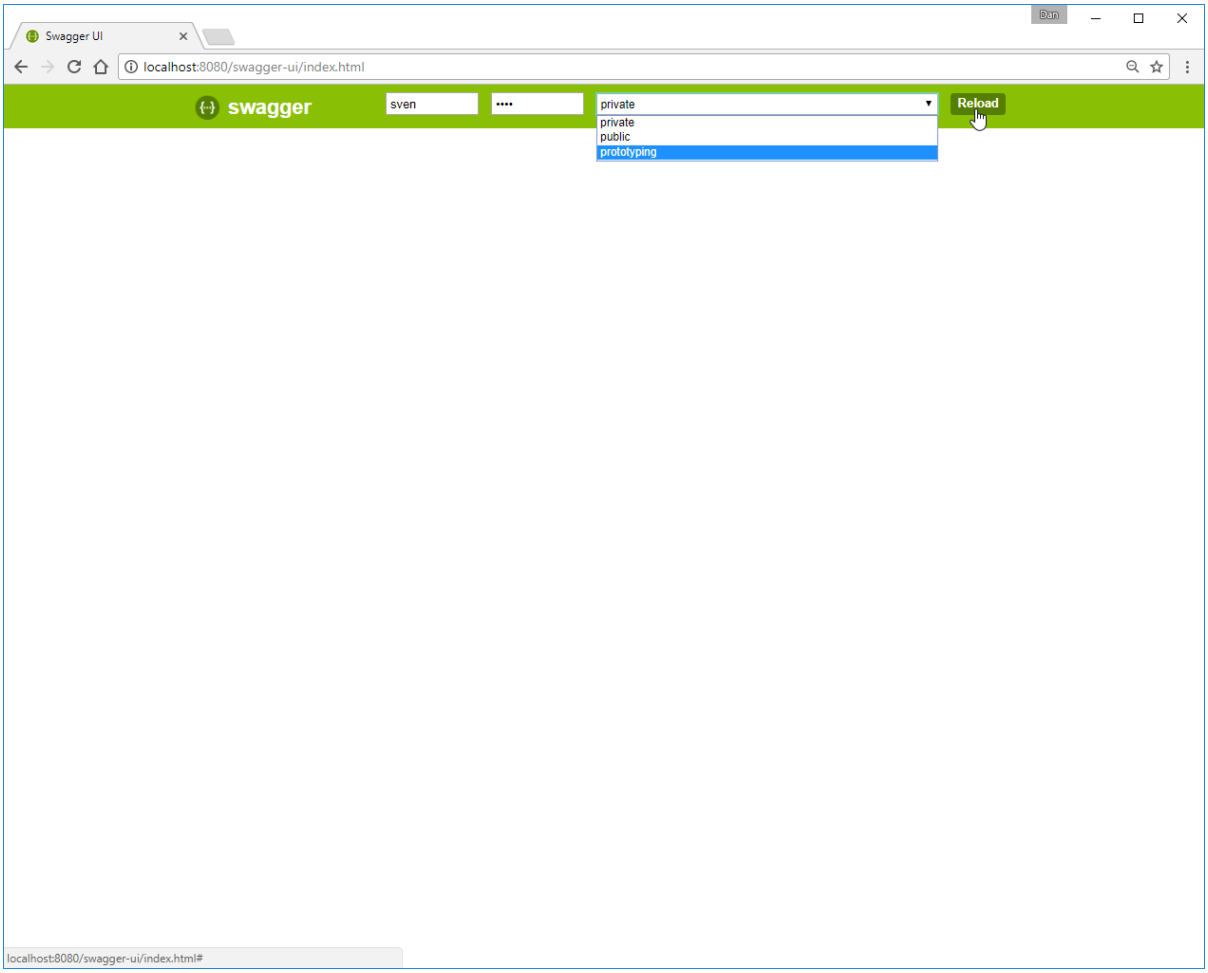


The above functionality is implemented by this code:

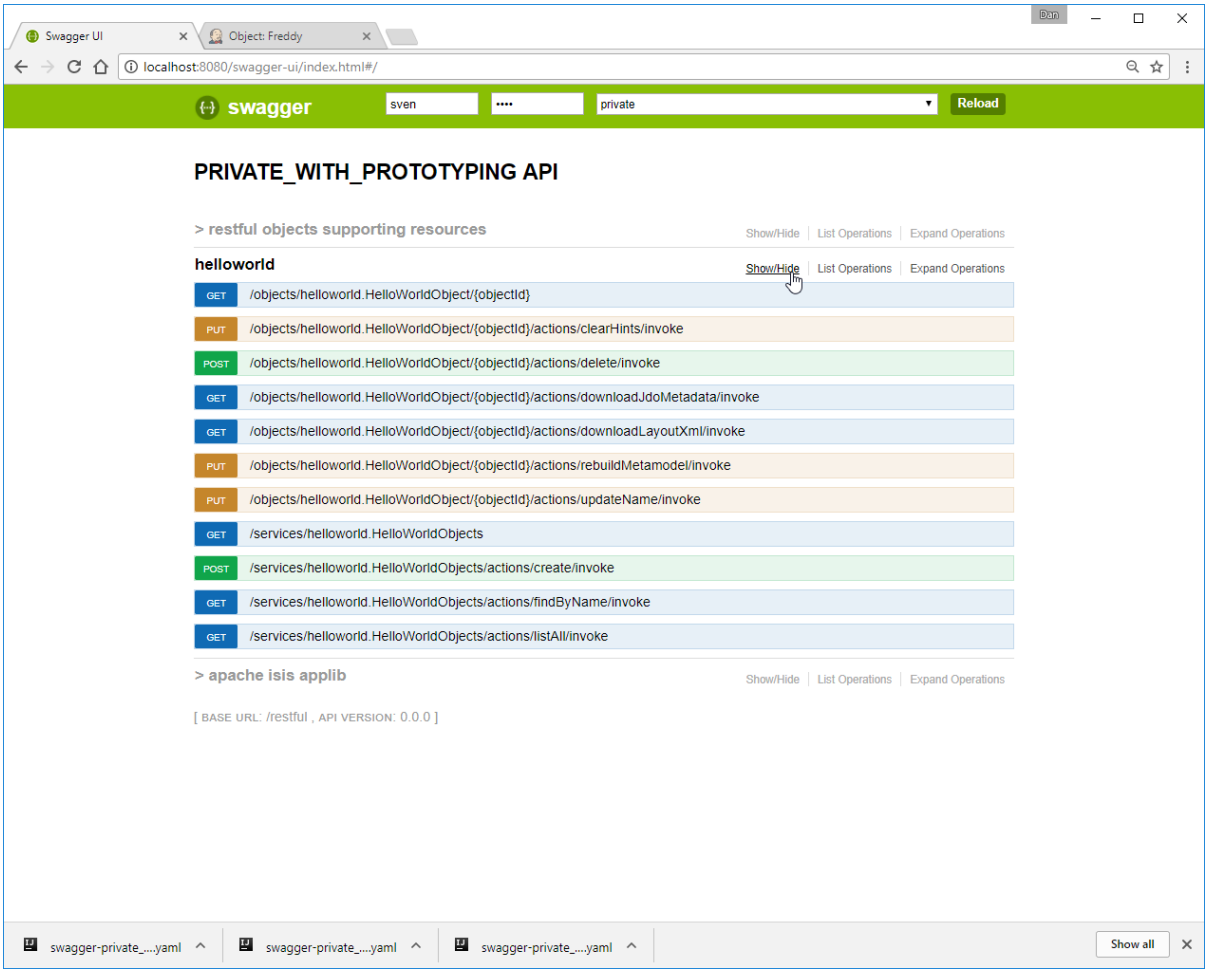
```
@Action(semantics = SemanticsOf.NON_IDEMPOTENT_ARE_YOU_SURE)
public void delete() {
    final String title = titleService.titleOf(this);
    messageService.informUser(String.format("'%s' deleted", title));
    repositoryService.removeAndFlush(this);
}
```

This uses three services provided by the framework; these are injected into the domain object automatically.

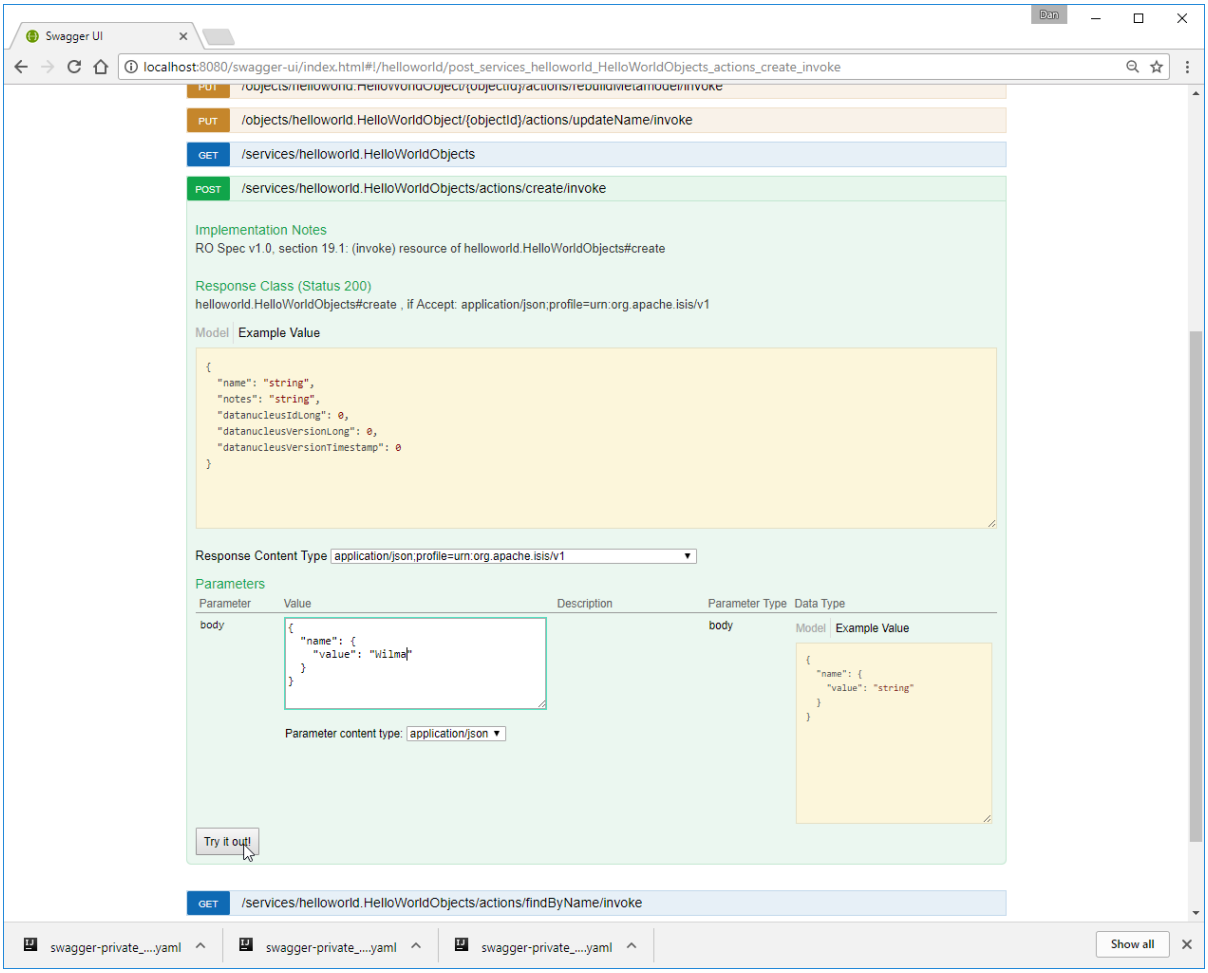
Going back to the home page (localhost:8080) we can use Swagger UI as a front-end to the REST API provided by the Restful Objects viewer.



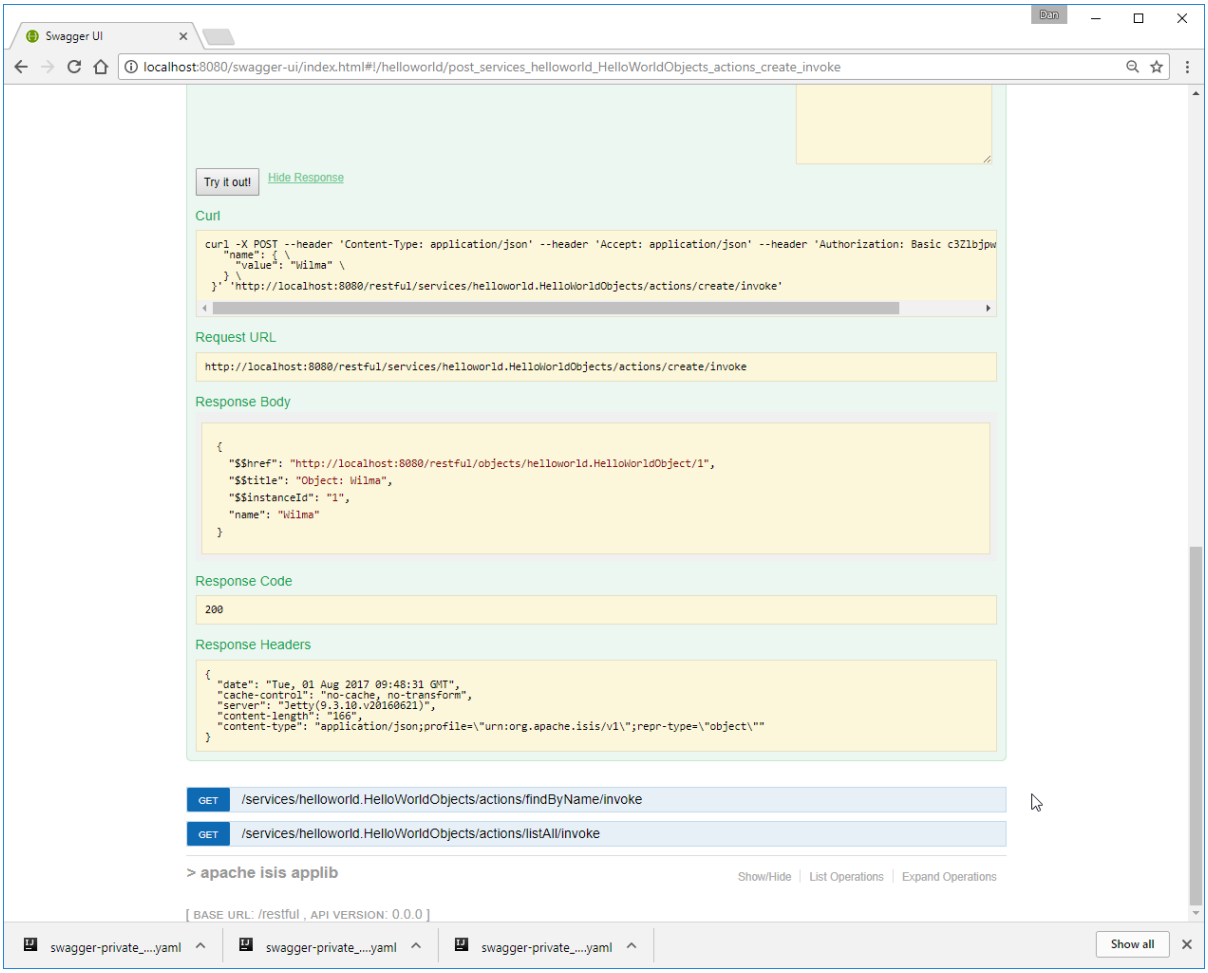
The Swagger UI is created dynamically from a Swagger schema definition (the schema definition file itself can be downloaded from the Wicket viewer's "Prototyping" menu). This Swagger schema definition groups resources according to Apache Isis metadata:



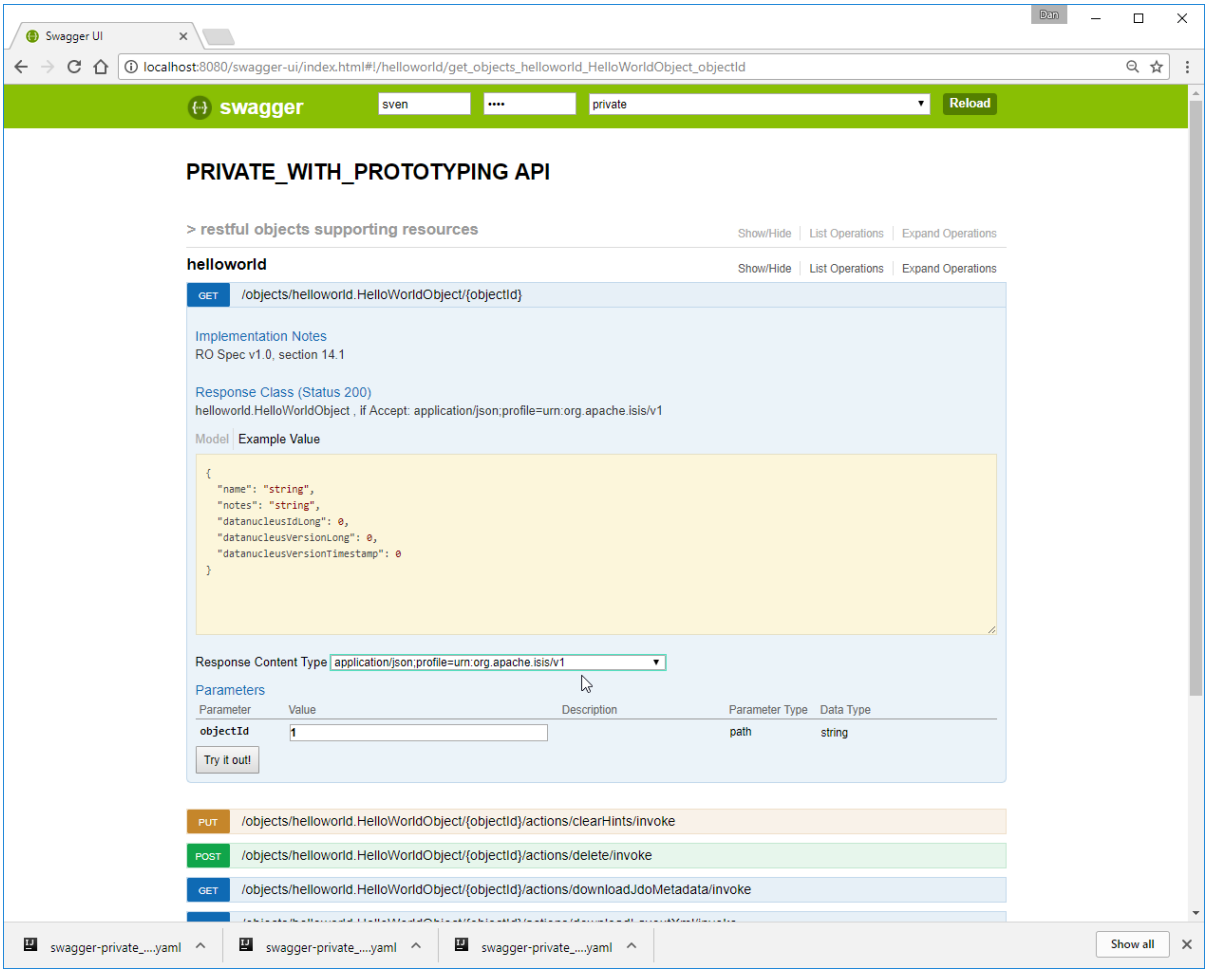
For example, an object can be created using the resource that represents the `HelloWorldObjects#create` action:



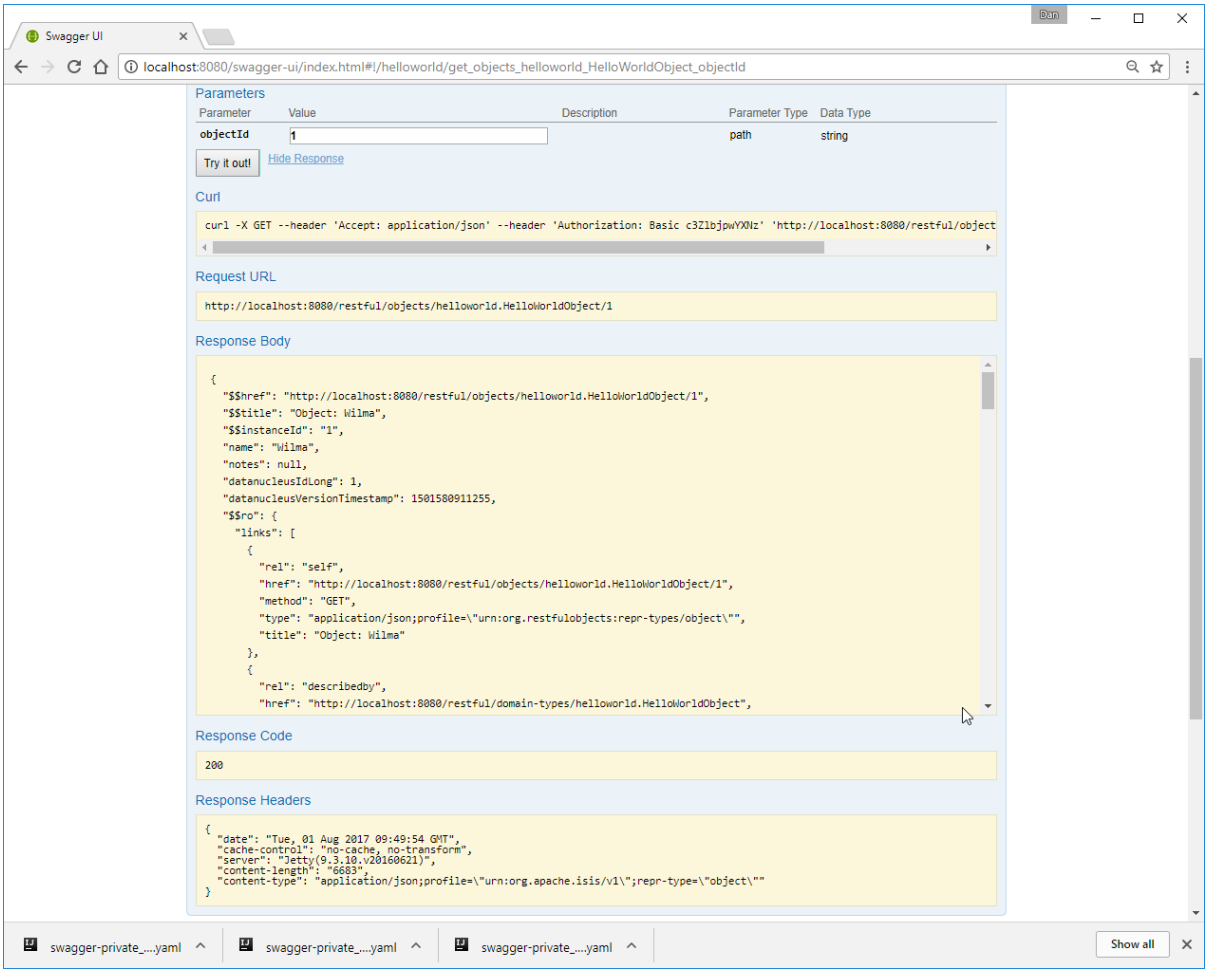
The response indicates that the object was successfully created:



The Swagger UI also provides a resource to retrieve any object:



This results in a representation of the domain object (as per the requested `Response Content Type`, ie `ACCEPT` header):



The Swagger UI is provided as a convenience; the REST API is actually a complete hypermedia API (in other words you can follow the links to access all the behaviour exposed in the regular Wicket app). The REST API implemented by Apache Isis is specified in the Restful Object spec.

4.7. Experimenting with the App

Once you are familiar with the generated app, try modifying it. There is plenty more guidance on this site; start with this guide (fundamentals) and then look at the other guides available the main documentation page.

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.

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

4.8. Moving on

When you are ready to start working on your own app, we *don't* recommend building on top of the helloworld app.

Instead, we suggest that you start with the simpleapp archetype instead. Although a little more complex, it provides more structure and tests, all of which will help you as your application grows.

Chapter 5. SimpleApp Archetype

The quickest way to get started building an application "for real" is to run the `simpleapp` archetype. Like the helloworld archetype, this too will generate a very simple one-class domain model (an entity called `SimpleObject` with a couple of properties).

However, the generated application also provides more structure to assist you as your application grows.

We'll talk more about the structure of the generated app below, but for now let's see how to generate the application.

The (non-ASF) Incode Platform's quickstart archetype builds upon the simpleapp archetype, but also adds in support for various Incode Platform modules such as security, auditing, commands and publishing.

5.1. 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=1.16.0 \
    -D groupId=com.mycompany \
    -D artifactId=myapp \
    -D version=1.0-SNAPSHOT \
    -B
```

where:

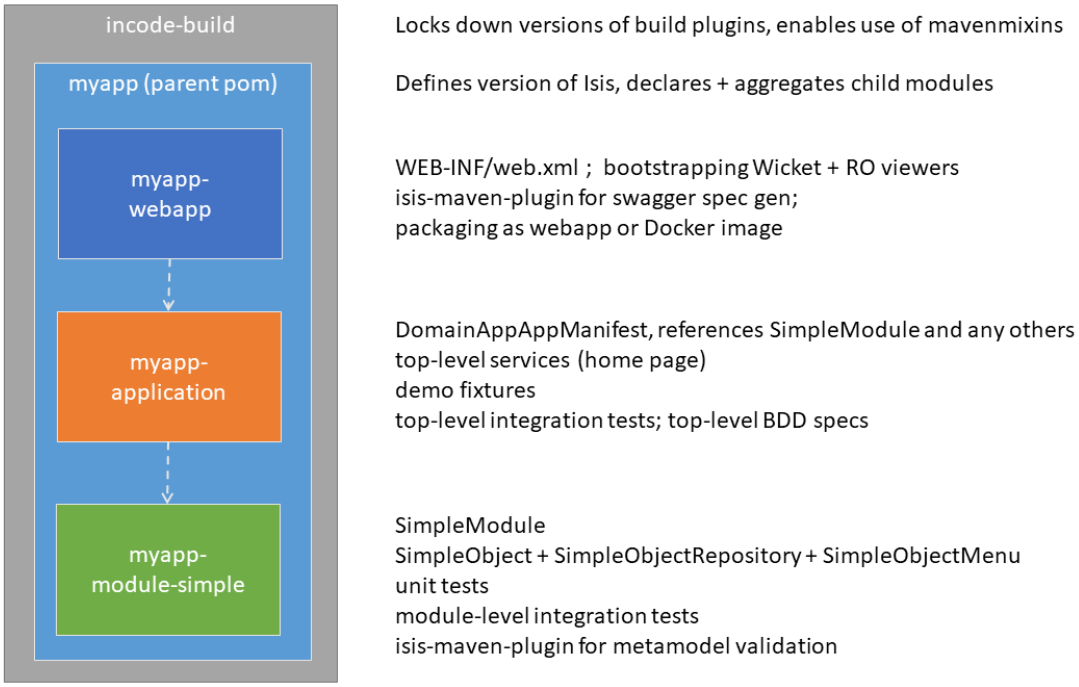
- `groupId` represents your own organization, and
- `artifactId` is a unique identifier for this app within your organization.
- `version` is the initial (snapshot) version of your app

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

5.2. Structure of the App

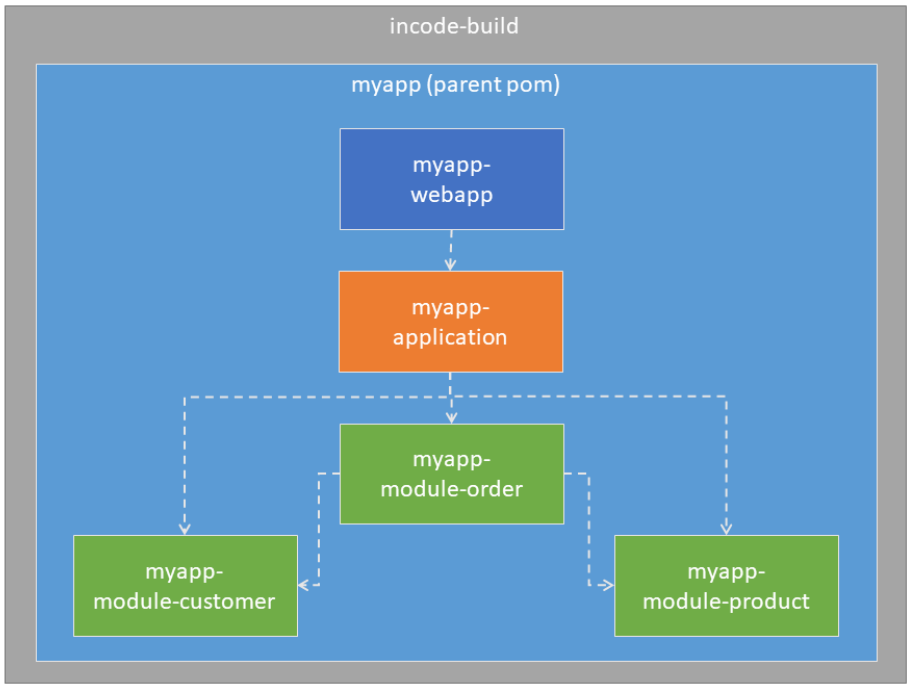
As mentioned above, the application generated by the simpleapp archetype is a multi-module project, structured so that you easily extend it as your application grows.

The application consists of three modules, with a top-level module acting as an aggregator and also parent:



The application separates domain object model holding the business logic (the `myapp-module-simple` Maven module containing `SimpleObject` entity and supporting domain classes) from the bootstrapping modules (`myapp-application` Maven module and the `myapp-webapp` module).

In a larger application there would likely be many more modules containing these domain object modules. For example, you might have a `myapp-module-customer` holding a `Customer` entity and related entities/services, a `myapp-module-product` holding a `Product` catalog, and a `myapp-module-order` to hold the `Order`s placed by `Customer`s:



We can use Maven dependency management to ensure that there are no cyclic dependencies (order "knows about" product but product does not know about orders) and ensure that the codebase

remains decoupled. When Java9 modules are commonplace, we'll also be able to restrict visibility of classes between modules.

Note that while Maven dependencies are transitive (in the example the `myapp-application` needs only depend directly on `myapp-module-order`, the modules defined in the `AppManifest` are *not* transitive: all modules must be listed.

Let's now review the contents of each of the generated modules.

5.2.1. myapp (parent)

The parent module is a fairly conventional, declaring its child modules (using `<dependencyManagement>` elements) and aggregating them (using `<module>` elements).

One thing you'll discover when you review the generated classes is that they all reside under the `domainapp` package.

While it's more conventional to use the inverse domain name for package (eg `com.mycompany.myapp`, that's only really appropriate for library code that will be released for reuse by multiple applications in different orgnisations (eg open source).

For internal application though this is less of a concern; indeed, avoiding the domain name means that if the company rebrands or is taken over then nothing needs be changed.

Of course, you are always free to move the classes to a different package if you wish.

5.2.2. myapp-application

The production classes for `myapp-application` module (in `src/main/java`) are:

```
domainapp/
└──── application/
    ├──── DomainAppApplicationModule.java
    ├──── fixture/
    │    ├──── DomainAppFixtureScriptsSpecificationProvider.java
    │    └──── scenarios/
    │         └──── DomainAppDemo.java
    ├──── manifest/
    │    ├──── DomainAppAppManifest.java
    │    ├──── DomainAppAppManifestBypassSecurity.java
    │    ├──── DomainAppAppManifestWithFixtures.java
    │    ├──── DomainAppAppManifestWithFixturesBypassSecurity.java
    │    └──── menubars.layout.xml
    └──── services
         └──── homepage
              ├──── HomePageService.java
              ├──── HomePageViewModel.java
              ├──── HomePageViewModel.layout.xml
              └──── HomePageViewModel.layout.png
```

There are also supporting files in this package in `src/main/resources`:

```
domainapp/
└──── application/
    └──── manifest/
         ├──── authentication_shiro.properties
         ├──── isis.properties
         ├──── persistor_datanucleus.properties
         ├──── viewer_restfulobjects.properties
         └──── viewer_wicket.properties
```

The `DomainAppAppManifest` is the key class here, typically being used to bootstrap the application. It is quite short:

```
public class DomainAppAppManifest extends AppManifestAbstract2 {

    public static final Builder BUILDER = Builder
            .forModule(new DomainAppApplicationModule())
            .withConfigurationPropertiesFile(DomainAppAppManifest.class,
                    "isis.properties",
                    "authentication_shiro.properties",
                    "persistor_datanucleus.properties",
                    "viewer_restfulobjects.properties",
                    "viewer_wicket.properties")
            .withAuthMechanism("shiro");

    public DomainAppAppManifest() {
        super(BUILDER);
    }
}
```

The manifest uses the builder defined by `AppManifestAbstract2` and references a single top-level (Isis) module, namely `DomainAppApplicationModule`:

```
public class DomainAppApplicationModule extends ModuleAbstract {
    @Override
    public Set<Module> getDependencies() {
        return Sets.<Module>newHashSet(new SimpleModule());
    }
}
```

where `SimpleModule` in defined in the `mypp-module-simple` module (below).

The primary purpose of the module class is to identify packages and subpackages that the framework should scan for entities and domain services. The transitive dependencies between modules are automatically resolved. The net effect is that all the domain services and entities in this module as well as those modules referenced are included into the app.

Going back to the manifest, it also defines a number of static configuration files, all loaded from the classpath. Each file contains configuration setting for a different part of the runtime (`isis.properties` for the core framework, shiro for security, datanucleus for the objectstore, and the two viewers).

The `menubars.layout.xml` file also resides in the same package as the manifest; this defines the menubar structure.

There are also several variations on the app manifest; these can be used to bootstrap the application with fixtures, or disabling security.

The `domainapp.application.services` package contains the `HomePageService` domain service. This simply has a single action annotated with `@HomePage`:

```
@Action(semantics = SemanticsOf.SAFE)
@HomePage
public HomePageViewModel homePage() {
    return factoryService.instantiate(HomePageViewModel.class);
}
```

which returns the `HomePageViewModel` for use as the home page. The `HomePageViewModel` itself just renders a collection of `SimpleObject`s in a list (`HomePageViewModel.layout.xml` defines the UI layout).

The final package in the application module is `domainapp.application.fixture`. The important class here is `DomainAppDemo`, a fixture script that can be used to setup the application with some dummy data. This is used in the app itself when running in prototype mode (against an in-memory database), and can also be used by integration tests.

There is in fact also a domain service defined here, namely `DomainAppFixtureScriptsSpecificationProvider`. This is just used to configure the run fixture script menu item shown on the "Prototyping" menu.

The module also defines a number of BDD specs and integration tests, in `src/test/java`. The BDD specs (run using Cucumber) reside under `domain.application.bdd`:

```
domainapp/
└──── application/
    └──── bdd/
        ├──── specglue/
        │    ├──── BootstrappingGlue.java
        │    └──── CatalogOfFixturesGlue.java
        └──── specs/
            ├──── RunIntegBddSpecs.java
            └──── SimpleObjectSpec_listAllAndCreate.feature
```

Here the `BootstrappingGlue` glue class inherits from the framework's `CukeGlueBootstrappingAbstract` class, and bootstraps using the `DomainAppApplicationModule` mentioned above.

There is just one feature file: `SimpleObjectSpec_listAllAndCreate.feature`, which is pretty simple:

```
@DomainAppDemo
Feature: List and Create New Simple Objects

  Scenario: Existing simple objects can be listed and new ones created
    Given there are initially 10 simple objects
    When  I create a new simple object
    Then  there are 11 simple objects
```

The `@DomainAppDemo` annotation causes the `DomainAppDemo` fixture script to be run; this is the purpose of the `CatalogOfFixturesGlue` glue class.

The specs themselves are run by the `RunIntegBddSpecs.java` class, which specifies which packages to search for "glue". This is just standard Cucumber bootstrapping.

The integration tests meanwhile are in `domainapp.application.integtests`:

```
domainapp/
 └──── application/
     └──── integtests/
         ├──── DomainAppIntegTestAbstract.java
         └──── Smoke_IntegTest.java
```

The `Smoke_IntegTest` inherits `DomainAppIntegTestAbstract`, which in turn inherits from `IntegrationTestAbstract3` and uses the `DomainAppApplicationModule` previously discussed. Moreover, the `Smoke_IntegTest` uses the same `DomainAppDemo` fixture script. The application and the smoke tests therefore run with the exact same state, making debugging easy.

With regard to the naming of these various BDD specs and integration tests, they follow the naming convention required by the (non-ASF) "surefire" mavenmixin that configures the maven surefire plugin.

5.2.3. myapp-module-simple

This module is where the domain object model lives, that is the business logic of the application itself. This typically comprises entities, domain services, mixins and view models.

As discussed above, larger applications will likely have multiple modules each containing their own slice of business logic.

Initially though you should probably just use regular Java packages to separate out functionality; you can carve out separate modules later on once the responsibilities of each have settled down.

The classes for the simple module reside in the `domainapp.modules.simple` package. Under `src/main/java` we have:

```
domainapp/
 └─── modules/
     └─── simple/
         ├─── SimpleModule.java
         ├─── SimpleModuleManifest.java
         ├─── dom/
         │    └─── impl/
         │        ├─── SimpleObject.java
         │        ├─── SimpleObject.layout.xml
         │        ├─── SimpleObject.png
         │        ├─── SimpleObjectMenu.java
         │        └─── SimpleObjectRepository.java
         └─── fixture/
             ├─── SimpleObject_persona.java
             └─── SimpleObjectBuilder.java
META-INF/
 └─── persistence.xml
```

The `SimpleModule` is the (single) module class referenced from the previously discussed `DomainAppApplicationModule`, meaning that all the entities, domain services and fixtures within it are included within the application.

- `SimpleObject` is the (one-and-only) domain entity defined (with `SimpleObject.layout.xml` defines its layout in the UI).
- `SimpleObjects` domain service's whos actions appear as menu items and which acts as a repository to create and find `SimpleObject`s.

The `SimpleModule` class also defines a teardown fixture, automatically called by integration tests.

```
public class SimpleModule extends ModuleAbstract {
    @Override
    public FixtureScript getTeardownFixture() {
        return new TeardownFixtureAbstract2() {
            @Override
            protected void execute(ExecutionContext ec) {
                deleteFrom(SimpleObject.class);
            }
        };
    }
    ...
}
```

In the `fixture` subpackage is the `SimpleObject_persona` "persona" which uses the corresponding `SimpleObjectBuilder` builder script; further discussion on this pattern here. These fixtures are also used by "local" integration tests, which reside under `src/test/java`.

There are also unit tests and "glue" for the BDD specs:

```
domainapp/
└─── modules/
    └─── simple/
        ├─── dom/
        │    └─── impl/
        │        ├─── SimpleObject_Test.java
        │        └─── SimpleObjectRepository_Test.java
        └─── integtests/
            ├─── SimpleModuleIntegTestAbstract.java
            │    └─── tests/
            │        ├─── SimpleObject_IntegTest.java
            │        └─── SimpleObjectMenu_IntegTest.java
            └─── specglue/
                └─── SimpleObjectMenuGlue.java
```

The simpleapp application has both "local" integration tests (defined within the `myapp-module-simple` module) and also "global" integration tests (the "smoke" tests in `myapp-application` module). There's a role for both: local integration tests should fully exercise the module but may need to mock out collaborations between modules, while global integration tests exercise the whole application (but an over-reliance on these can cause test run times to bloat).

With regard to the naming of these various BDD specs and integration tests, they follow the naming convention required by the (non-ASF) "surefire" mavenmixin that configures the maven surefire plugin. Integration tests include the name "IntegTest", while unit tests contain merely "Test".

The module also defines its own manifest, `SimpleModuleManifest`. This is used to run Isis' own maven plugin to validate the domain object model (eg to detect orphaned supported methods).

5.2.4. myapp-webapp

Finally, in the `myapp-webapp` module we have the classes and configuration to package and bootstrap the application as a webapp.

Under `src/main/java` there is:

```
domainapp/
└─── webapp/
    ├─── DomainApplication.java
    └─── welcome.html
```

The `DomainApplication` is required to bootstrap the Wicket viewer (it is configured in `WEB-INF/web.xml`, discussed below). Within `DomainApplication` is the bootstrapping of the Wicket viewer. Internally it uses Google Guice to configure various static resources served up by Wicket:

```
public class DomainApplication extends IsisWicketApplication {

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

The `configure()` method is the place to change the name of the application for example, or to change the initial about and welcome messages. The text of the welcome page shown by the Wicket viewer can be found in `welcome.html`, loaded from the classpath and in the same package as `DomainApplication`.

Under `src/main/webapp` we have various resources that are used to configure the webapp, or that are served up by the running webapp:

```
about/
 └──── index.html
css/
 └──── application.css
scripts/
 └──── application.js
swagger-ui/
WEB-INF/
 ├──── isis.properties
 ├──── logging.properties
 ├──── shiro.ini
 ├──── translations.po
 └──── web.xml
```

Most important of these is `WEB-INF/web.xml`, which bootstraps both the Wicket viewer and the Restful Objects viewer (the REST API derived from the domain object model):

web.xml

```
<web-app ...>
  ...
  <filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>domainapp.webapp.DomainApplication</param-value>
    </init-param>
  </filter>
  ...
  <context-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>
      org.apache.isis.viewer.restfulobjects.server.RestfulObjectsApplication
    </param-value>
  </context-param>
  ...
</web-app>
```

The `about/index.html` is the page shown at the root of the package, providing links to either the Wicket viewer or to the Swagger UI. In a production application this is usually replaced with a page that does an HTTP 302 redirect to the Wicket viewer.

In `css/application.css` you can use to customise CSS, typically to highlight certain fields or states. The pages generated by the Wicket viewer have plenty of CSS classes to target. You can also implement the `cssClass()` method in each domain object to provide additional CSS classes to target.

Similarly, in `scripts/application.js` you have the option to add arbitrary Javascript. JQuery is available by default.

In `swagger-ui` is a copy of the Swagger 2.x UI classes, preconfigured to run against the REST API exposed by the Restful Objects viewer. This can be useful for developing custom applications, and is accessible from the initial page (served up by `about/index.html`).

Finally in `WEB-INF` we have the standard `web.xml` (already briefly discussed) along with several other files:

- `isis.properties` contains further configuration settings for Apache Isis itself.

  (As already discussed), these are in addition to the configuration properties found in various configuration properties that live alongside and that are loaded by the `DomainAppAppManifest` class. Those in the WEB-INF/isis.properties file are those that are likely to change when running the application in different environments.

- `logging.properties` configures log4j.

  The framework is configured to use slf4j running against log4j.

- `shiro.ini` configures Apache Shiro, used for security (authentication and authorisation)
- `web.xml` configures the Wicket viewer and Restful Objects viewer. It also sets up various filters for serving up static resources with caching HTTP headers.

The webapp module's `pom.xml` also has several tricks up its sleeve:

- most fundamentally, it allows the application to be packaged up as a regular `.war` file for deployment to a servlet container such as Tomcat 8.x
- it also uses the jetty-console mavenmixin to package the application as a standalone executable JAR (with an embedded jetty container).

  The files in `src/main/jettyconsole` provide a splash image (if not run in headless mode).

- the docker mavenmixin packages up the application as a docker image.

  This uses the Dockerfile residing in `docker/Dockerfile` (under `src/main/resources`).

- also, the Apache Isis Maven plugin is also configured to generate a swagger spec file for the entire application, in the `xxx-webapp` module

Now you know your way around the code generated by the archetype, lets see how to build the app and run it.

5.3. 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.

5.4. 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.

5.4.1. 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:

```
mvn -pl webapp jetty:run
```

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

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

5.4.2. 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.

5.4.3. Using Docker

It's also possible to package up the application as a docker image to run as a container.

- To package up the application as a docker image (specifying the docker image name as a system property):

  ```
  mvn install -Dmavenmixin-docker -Ddocker-plugin.imageName=mycompany/myapp
  ```

  Alternatively, define the `${docker-plugin.imageName}` in the `webapp` module and use simply:

  ```
  mvn install -Dmavenmixin-docker
  ```

  The packaged image can be viewed using `docker images`.

- To run a docker image previously packaged:

  ```
  docker container run -d -p 8080:8080 mycompany/myapp
  ```

  This can then be accessed at localhost:8080.

  See mavenmixin-docker for further details on how to run docker images.

- To upload the application as a docker image to docker hub (or some other docker registry):

  ```
  mvn -pl webapp deploy -Dmavenmixin-docker
  ```

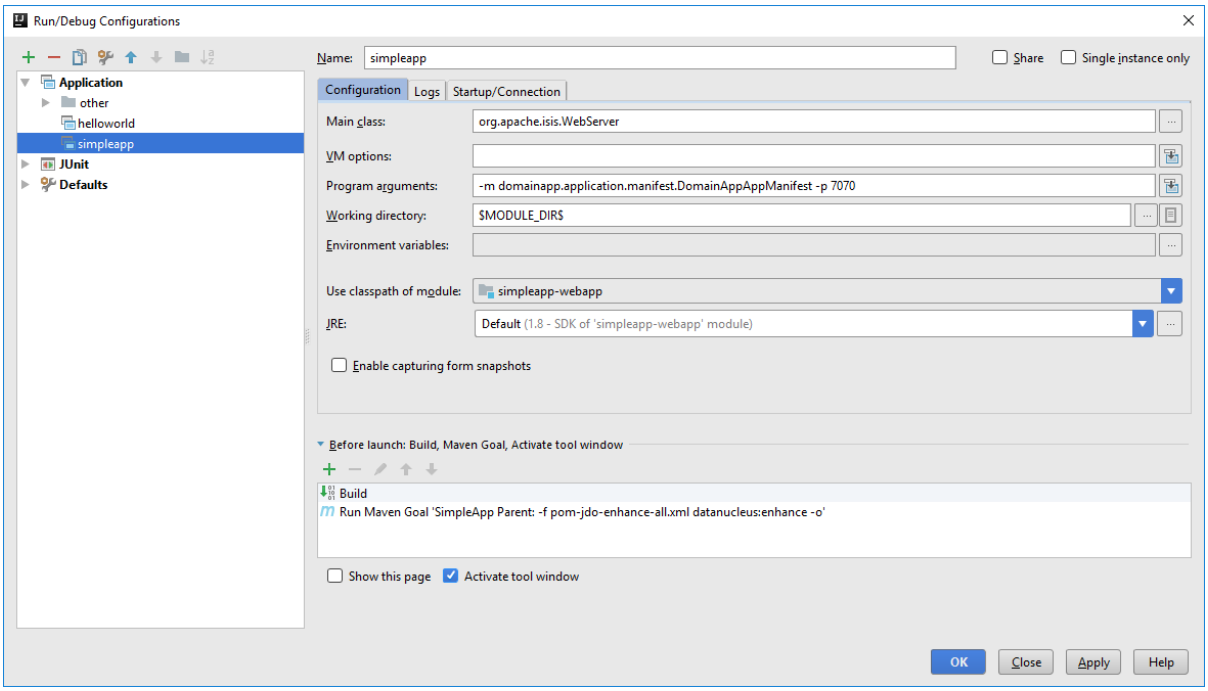
  This assumes that the `${docker-plugin.imageName}` property has been defined, *and* also that docker registry credentials have been specified in `~/.m2/settings.xml`. Once more, see mavenmixin-docker for further details.

5.4.4. From within the IDE

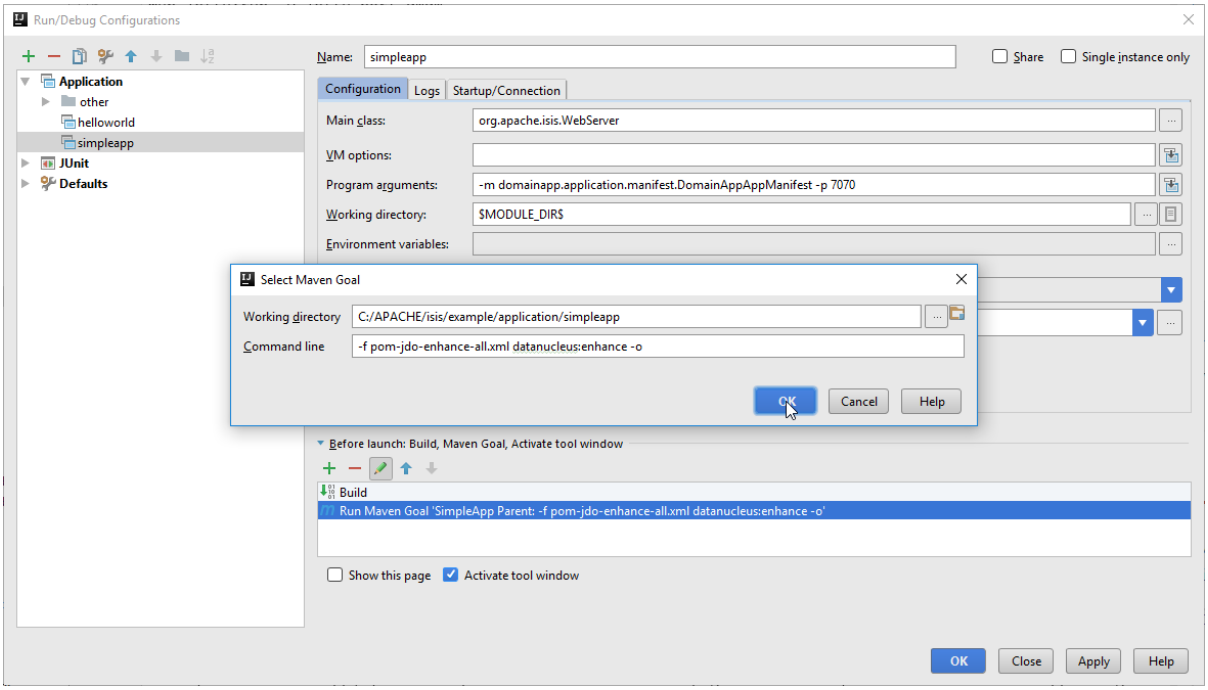
Most of the time, though, you'll probably want to run the app from within your IDE. The mechanics

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:



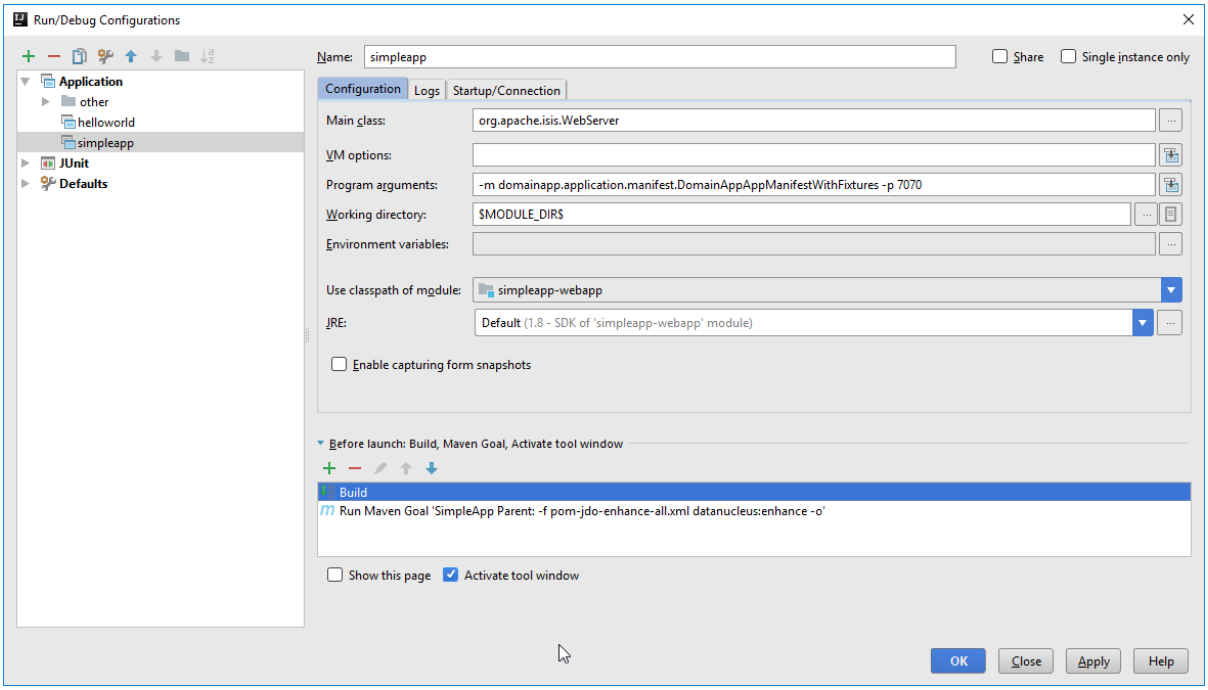
with the maven goal to run the DataNucleus enhancer (discussed in more detail here) before launch defined as:



5.5. 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:

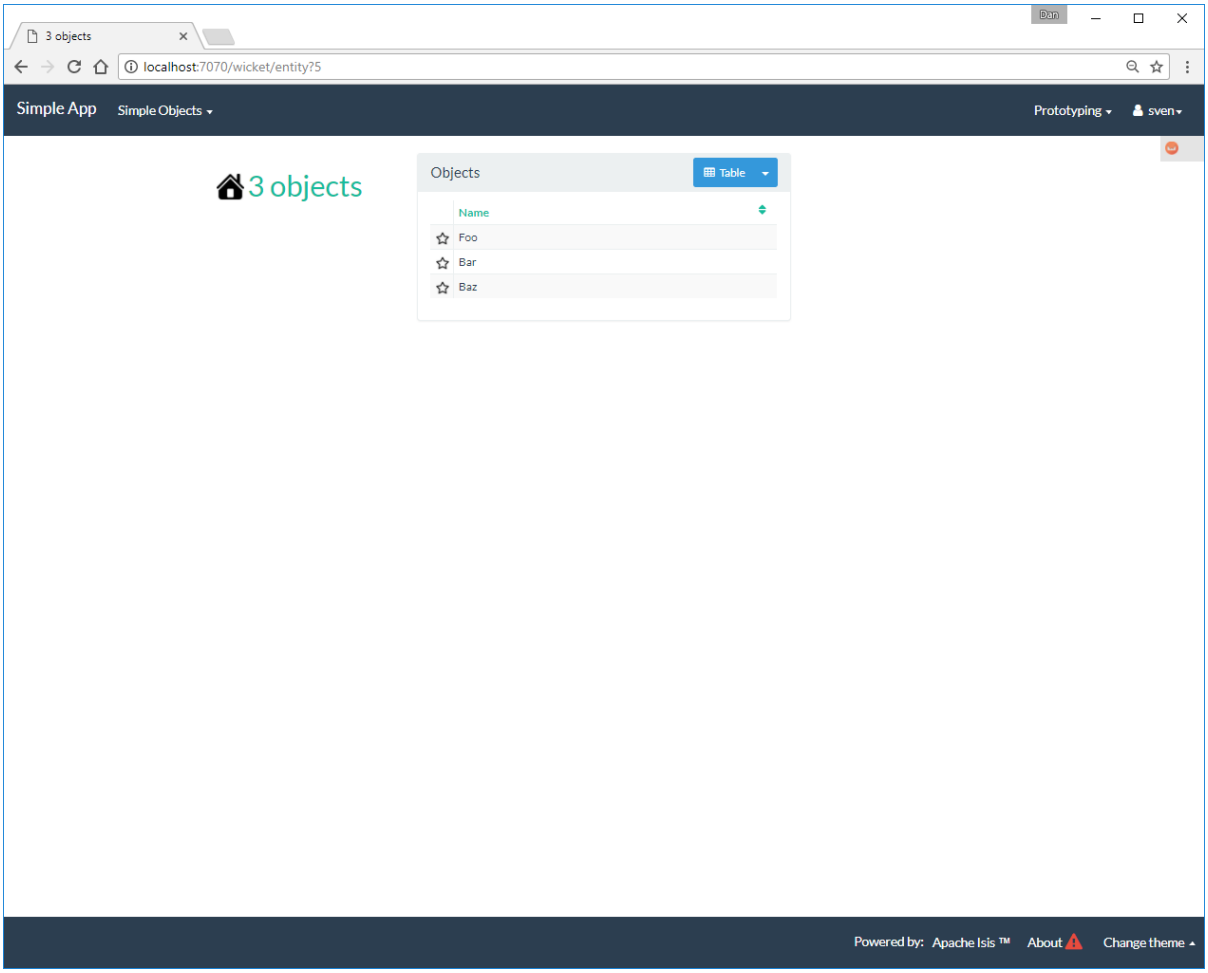


Alternatively, you can run with a different `AppManifest` using the `--appManifest` (or `-m`) flag. The archetype provides `domainapp.app.DomainAppAppManifestWithFixtures` which specifies the aforementioned `RecreateSimpleObjects` fixture.

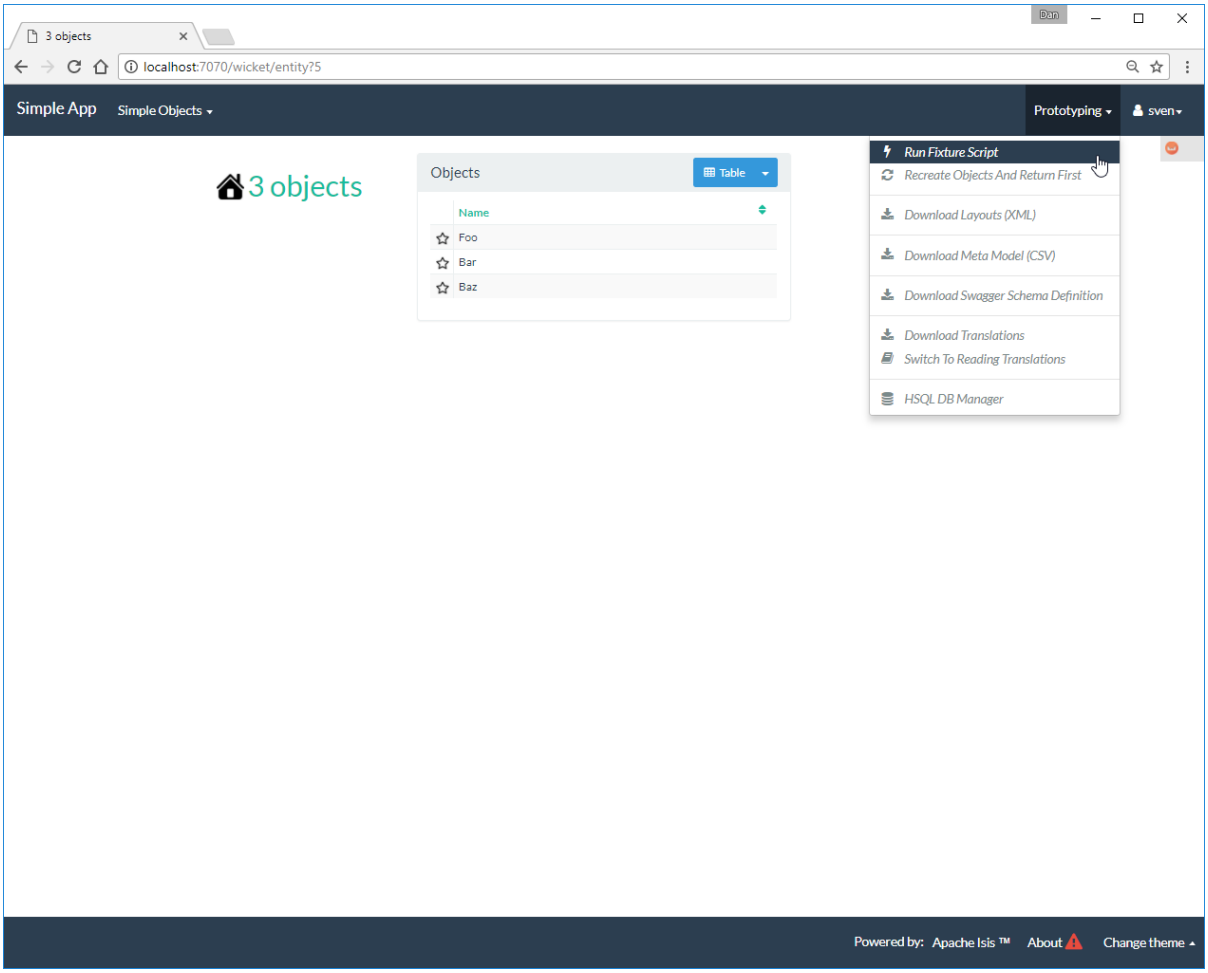
5.6. Using the App

The generated application is almost identical similar to that generated by helloworld archetype; a description of how to use it can be found here.

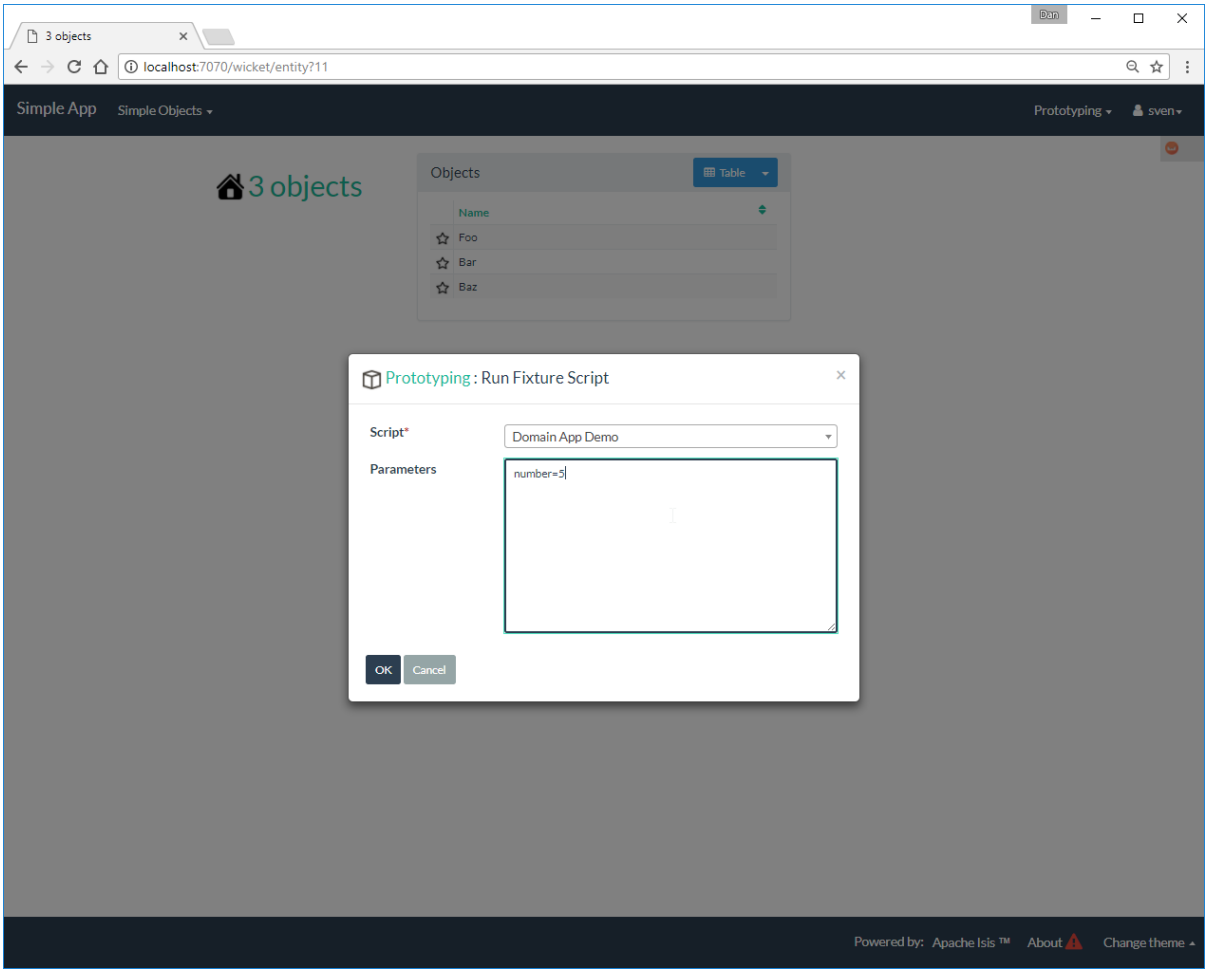
One additional feature that the simpleapp contains over the helloworld app is a home page. This shows all domain objects (as installed by fixture scripts, described above).



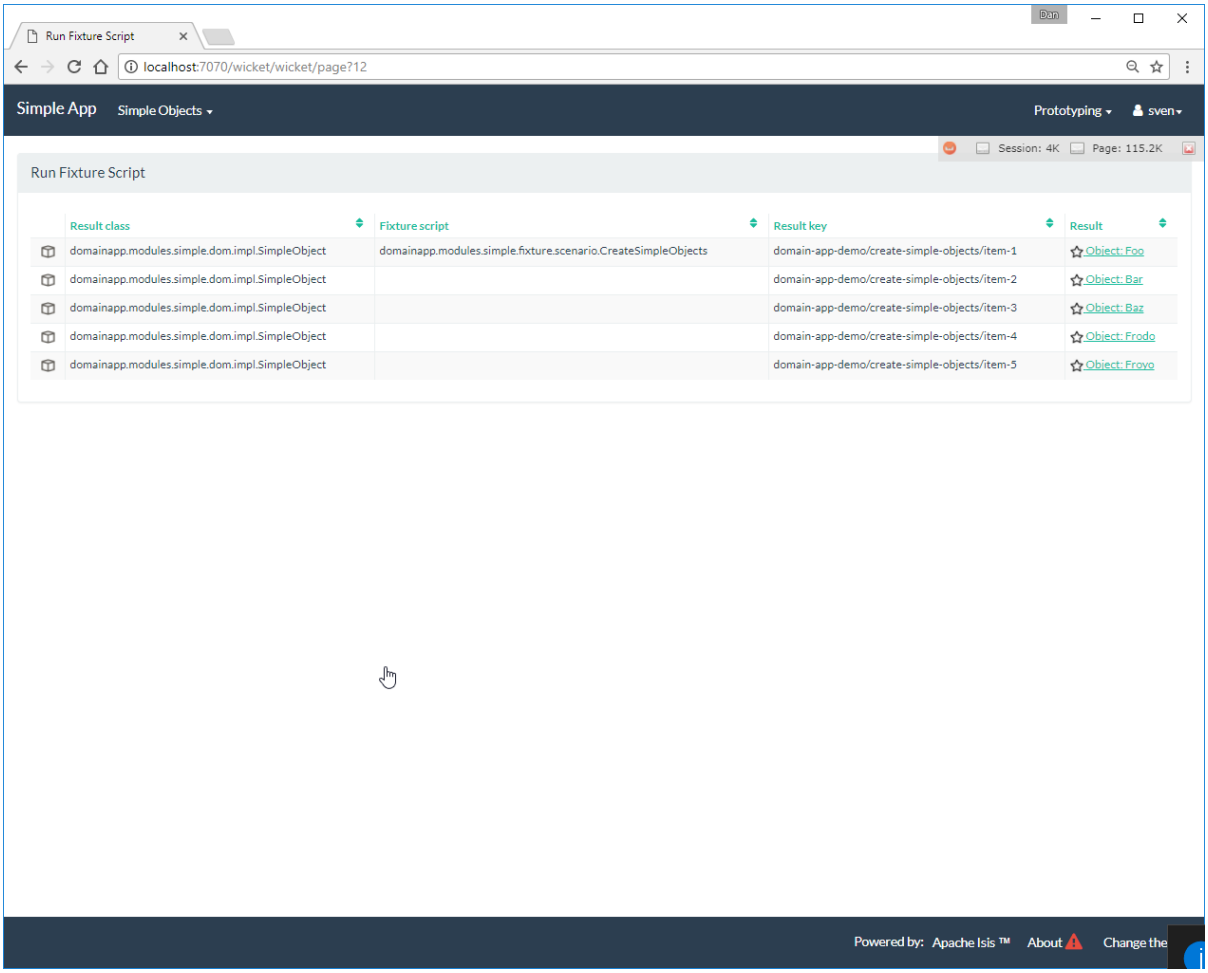
It’s also possible to run fixture scripts from the app itself:



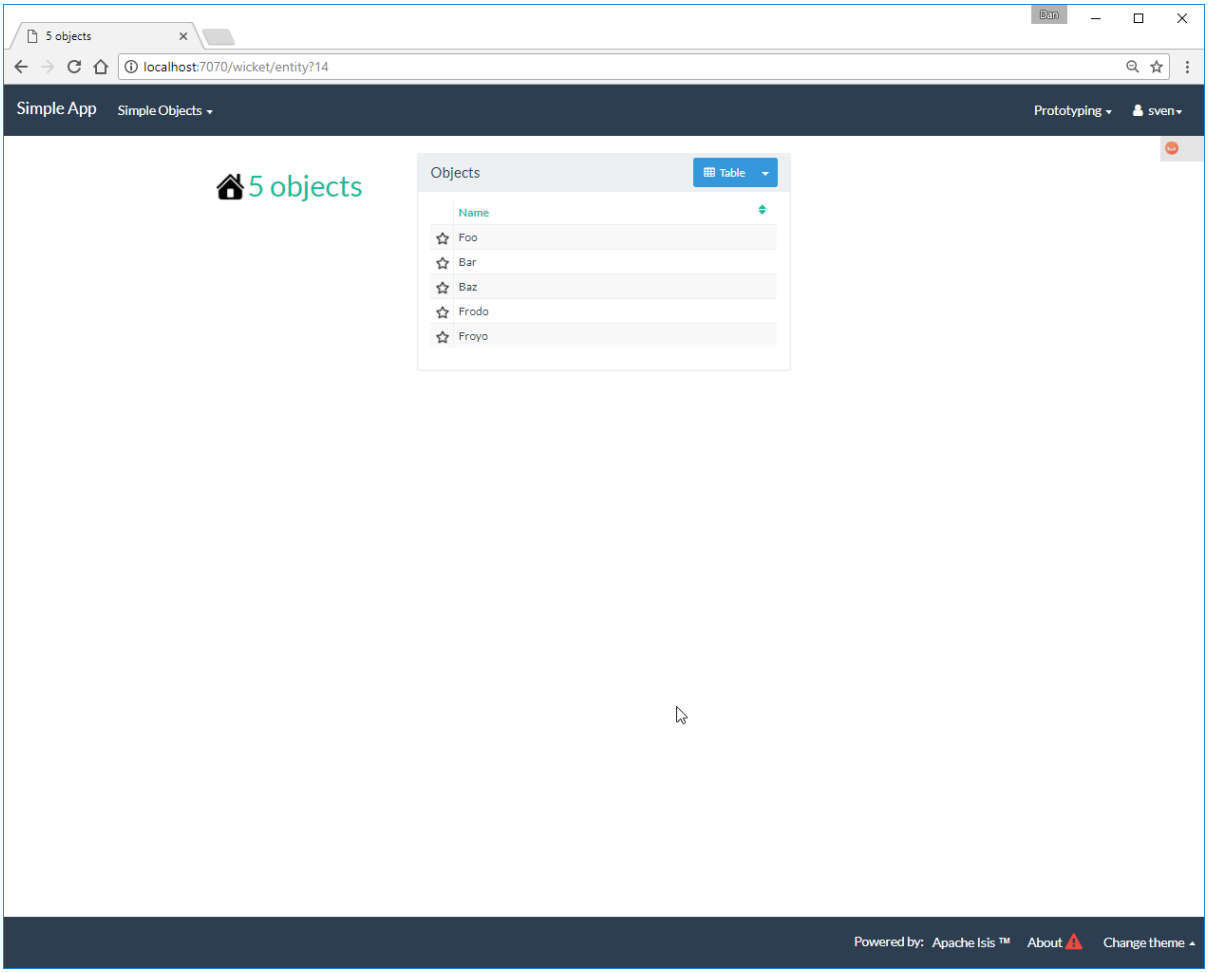
Some fixture scripts may allow their default behaviour to be tweaked ((eg specify how many objects to create):



The table summarises the resultant fixtures that were run:



Navigating back to the home page shows the newly created objects:



5.7. 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; start with this guide (fundamentals) and then look at the other guides available the main documentation page.

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.

Chapter 6. Programming Model

Apache Isis works by building a metamodel of the domain objects: entities, domain services, view models and mixins. Dependent on the sort of domain object, the class methods represent both state — (single-valued) properties and (multi-valued) collections — and behaviour — actions.

More specifically, both entities and view models can have properties, collections and actions, while domain services have just actions. Mixins also define only actions, though depending on their semantics they may be rendered as derived properties or collections on the domain object to which they contribute.

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.

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); the framework will automatically manage the view model's state (properties only, not collections).

  Or, they can be annotated using the JAXB `@XmlTypeAdapter` annotation, which allows the view models' properties *and* collections state to be managed.

- **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.

These entities' state is managed by the framework, in the same ways as view models. Indeed, they can additionally be annotated using `@XmlTypeAdapter` if required.

- **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.

You can generally recognize an Apache Isis domain class because it will be probably be annotated using `@DomainObject` and `@DomainService`.

It's 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.

The framework also defines supplementary annotations, `@DomainObjectLayout` and `@DomainServiceLayout`. These provide hints relating to the layout of the domain object in the user interface. (Alternatively, these UI hints can be defined in a supplementary `.layout.xml` file.

6.1. Domain Entities

Entities are persistent domain objects, with their persistence handled by JDO/DataNucleus. As such, they are mapped to a persistent object store, typically an RDBMS, with DataNucleus taking care of both lazy loading and also the persisting of modified ("dirty") objects.

Domain entities are generally decorated with both DataNucleus and Apache Isis annotations. Let's look at some of the most commonly-used annotations.

To start with, entities are flagged as being "persistence capable", indicating how JDO/DataNucleus should manage their identity:

```
@javax.jdo.annotations.PersistenceCapable(                          ①
        identityType=IdentityType.DATASTORE,                        ②
        schema = "simple",                                          ③
        table = "SimpleObject"
)
@javax.jdo.annotations.DatastoreIdentity(                           ④
        strategy=javax.jdo.annotations.IdGeneratorStrategy.IDENTITY,
        column="id"
)
@javax.jdo.annotations.Version(                                     ⑤
        strategy= VersionStrategy.DATE_TIME,
        column="version"
)
@DomainObject(                                                      ⑥
        objectType = "simple.SimpleObject"
)
public class SimpleObject { ... }
```

① The `@PersistenceCapable` annotation indicates that this is an entity to DataNucleus. The DataNucleus enhancer acts on the bytecode of compiled entities, injecting lazy loading and dirty object tracking functionality. Enhanced entities end up also implementing the `javax.jdo.spi.PersistenceCapable` interface.

② Indicates how identifiers for the entity are handled. Using `DATASTORE` means that a DataNucleus is responsible for assigning the value (rather than the application).

③ Specifies the RDBMS database schema and table name for this entity will reside. The schema should correspond with the module in which the entity resides. The table will default to the entity name if omitted.

④ For entities that are using `DATASTORE` identity, indicates how the id will be assigned. A common strategy is to allow the database to assign the id, for example using an identity column or a sequence.

⑤ The `@Version` annotation is useful for optimistic locking; the strategy indicates what to store in the `version` column.

⑥ The `@DomainObject` annotation identifies the domain object to Apache Isis (not DataNucleus). It isn't necessary to include this annotation — at least, not for entities — but it is nevertheless recommended. In particular, its strongly recommended that the `objectType` (which acts like an alias to the concrete domain class) is specified; note that it corresponds to the schema/table for DataNucleus' `@PersistenceCapable` annotation.

All domain entities will have some sort of mandatory key properties. The example below is a very simple case, where the entity is identified by a `name` property:

```
...
@javax.jdo.annotations.Unique(name="SimpleObject_name_UNQ", members = {"name"}) ①
public class SimpleObject
             implements Comparable<SimpleObject> {                                ②

    public SimpleObject(final String name) {                                      ③
        setName(name);
    }

    @javax.jdo.annotations.Column(allowsNull="false", length=50)                  ④
    @lombok.Getter @lombok.Setter
    private String name;

    @Override
    public String toString() {
        return ObjectContracts.toString(this, "name");                            ⑤
    }
    @Override
    public int compareTo(final SimpleObject other) {
        return ObjectContracts.compare(this, other, "name");                      ②
    }
}
```

① DataNucleus will automatically add a unique index to the primary surrogate id (discussed above), but additional alternative keys can be defined using the `@Unique` annotation. In the example above, the "name" property is assumed to be unique.

② Although not required, we strongly recommend that all entities are naturally `Comparable`. This then allows parent/child relationships to be defined using `SortedSet`s; RDBMS after all are set-oriented. The `ObjectContracts` utility class provided by Apache Isis makes it easy to implement the `compareTo()` method, but you can also just use an IDE to generate an implementation or roll your own.

③ Chances are that some of the properties of the entity will be mandatory, for example any properties that represent an alternate unique key to the entity. In regular Java programming we would represent this using a constructor that defines these mandatory properties, and in Apache Isis/DataNucleus we can likewise define such a constructor. When DataNucleus rehydrates domain entities from the database at runtime, it actually requires a no-arg constructor (it then sets all state reflectively). However, there is no need to provide such a no-arg constructor; it is added by the enhancer process.

④ The `name` property itself, using Lombok to generate the getter and setter. The `@Column` annotation specifies the length of the column in the RDBMS; this metadata is read by JDO/DataNucleus but Apache Isis itself also infers that the property is mandatory and its length from this annotation.

⑤ The `ObjectContracts` utility class also provides assistance for `toString()`, useful when debugging in an IDE.

It's also common for domain entities to have queries annotated on them. These are used by repository domain services to query for instances of the entity:

```
...
@javax.jdo.annotations.Queries({
    @javax.jdo.annotations.Query(                                          ①
        name = "findByName",                                               ②
        value = "SELECT "                                                  ③
                + "FROM domainapp.modules.simple.dom.impl.SimpleObject "   ④
                + "WHERE name.indexOf(:name) >= 0 ")                       ⑤
})
...
public class SimpleObject { ... }
```

① There may be several `@Query` annotations, nested within a `@Queries` annotation) defines queries using JDOQL.

② Defines the name of the query.

③ The definition of the query, using JDOQL syntax.

④ The fully-qualified class name. Must correspond to the class on which the annotation is defined (the framework checks this automatically on bootstrapping).

⑤ In this particular query, is an implementation of a LIKE "name%" query.

DataNucleus provides several APIs for defining queries, including entirely programmatic and type-safe APIs; but JDOQL is very similar to SQL and so easily learnt.

The corresponding repository method for the above query is:

```
public List<SimpleObject> findByName(String name) {
    return repositoryService.allMatches(                                   ①
            new QueryDefault<>(SimpleObject.class,                         ②
                        "findByName",                                      ③
                        "name",                                            ④
                        name);
}

@javax.inject.Inject
RepositoryService repositoryService;
```

① The `RepositoryService` is a generic facade over the JDO/DataNucleus API, provided by the Apache Isis framework.

② Specifies the class that is annotated with `@Query`

③ Corresponds to the `@Query#name()` attribute

④ Corresponds to the `:name` parameter in the query JDOQL string

See the DataNucleus objectstore guide for further information on annotating domain entities.

6.2. Domain Services

This section looks at the programming model for writing your own domain services.

6.2.1. 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_MENU_ONLY` indicates that the actions should appear on the menu of the Wicket viewer, and as top-level actions for the REST API provided by the Restful Objects viewer
- `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
- `VIEW_REST_ONLY` indicates that the actions should appear in the REST API provided by the Restful Objects viewer, but not rendered by the Wicket viewer.

There are also two other natures that should be considered "deprecated":

- `VIEW_CONTRIBUTIONS_ONLY` which indicates that the actions should be contributed as actions to its action parameters.

  This feature is deprecated because mixins are equivalent in functionality with a simpler programming model.

- `VIEW` combines both `VIEW_MENU_ONLY` and the deprecated `VIEW_CONTRIBUTIONS_ONLY`.

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.

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

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

6.2.2. Repository and Factory

The repository/factory uses an injected `RepositoryService` 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> findByName(String name) {
        return repositoryService.allMatches(                        ②
                new QueryDefault<>(Customer.class,
                           "findByName", "name", name);
    }
    public Customer newCustomer(...) {
        Customer Customer =
            repositoryService.instantiate(Customer.class);          ③
        ...
        repositoryService.persist(Customer);                        ③
        return Customer;
    }
    public List<Customer> allCustomers() {
        return repositoryService.allInstances(Customer.class);
    }
    @Inject
    RepositoryService repositoryService;
}
```

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

② uses injected `RepositoryService` to query via JDOQL.

③ uses injected `RepositoryService` to first instantiate and then save into the database a new `Customer` instance.

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

6.2.3. 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(                                                    ①
    nature = NatureOfService.VIEW_MENU_ONLY
)
@DomainServiceLayout(                                              ②
        named = "Customers",
        menuBar = DomainServiceLayout.MenuBar.PRIMARY,
        menuOrder = "10"
)
public class CustomerMenu {

    @Action(
            semantics = SemanticsOf.SAFE
    )
    @ActionLayout(bookmarking = BookmarkPolicy.AS_ROOT)
    @MemberOrder( sequence = "1" )
    public List<Customer> findByName(                              ③
            @ParameterLayout(named="Name")                         ④
            final String name
    ) {
        return customerRepository.findByName(name);                ⑤
    }

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

    @Action(
            semantics = SemanticsOf.SAFE,
            restrictTo = RestrictTo.PROTOTYPING                    ⑥
    )
    @MemberOrder( sequence = "99" )
    public List<Customer> listAll() {
        return customerRepository.listAll();
    }

    @Inject
    protected CustomerRepository customerRepository;               ④
}
```

① The (Apache Isis) `@DomainService` annotation is used to identify the class as a domain service. Apache Isis scans the classpath looking for classes with this annotation, so there very little configuration other than to tell the framework which packages to scan underneath. The `VIEW_MENU_ONLY` nature indicates that this service's actions should be exposed as menu items.

② The `@DomainServiceLayout` annotation provides UI hints. The menu is named "Customers" (otherwise it would have defaulted to "Customer Menu", based on the class name, while the

`menuOrder` attribute determines the order of the menu with respect to other menu services.

③ The `findByName` method is annotated with various Apache Isis annotations (`@Action`, `@ActionLayout` and `@MemberOrder` and is itself rendered in the UI as a "Find By Name" menu item underneath the "Simple Objects" menu.

④ The `@ParameterLayout` provides metadata for the parameter itself, in this case its name.

⑤ the action implementation delegates to an injected repository. The framework can inject into not just other domain services but will also automatically into domain entities and view models. There is further discussion of service injection below.

⑥ Prototype actions are rendered only in prototyping mode. A "list all" action such as this can be useful when exploring the domain with a small dataset.

Annotating action parameters with `@ParameterLayout#named()` can become somewhat tiresome. You can avoid doing this by configuring the (non-ASF) Incode Platform's paraname8 metamodel extension.

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

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.

6.2.4. Event Subscribers

Domain services acting as event subscribers can subscribe to domain, UI and lifecycle events, influencing the rendering and behaviour of other objects.

All subscribers must subscribe to the `EventBusService`; as this amounts to a few lines of boilerplate it's easiest to inherit from the convenience `AbstractSubscriber` class.

We suggest naming such classes `XxxSubscriptions`, for example:

```
@DomainService(
    nature=NatureOfService.DOMAIN                                   ①
)
@DomainServiceLayout(
    menuOrder="10",
    name="...",
}
public class CustomerOrderSubscriptions
                    extends AbstractSubscriber {                    ②

    @org.axonframework.eventhandling.annotation.EventHandler        ③
    @com.google.common.eventbus.Subscribe                           ③
    public void on(final Customer.DeletedEvent ev) {                ④
        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

② subclass from the `AbstractSubscriber` convenience superclass

③ The framework supports two different implementations for the `EventBusService` - Axon framework and Guava. Subscribers should use the appropriate annotation type depending on the implementation chosen (or as shown here, use both annotations).

④ the parameter type of the method corresponds to the event emitted on the event bus. The actual method name does not matter (though it must have `public` visibility).

6.2.5. Contributions (deprecated)

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

Contributed services should be considered a deprecated feature. Instead, contribute the behaviour using mixins.

We suggest naming such classes `XxxContributions`, eg:

```
@DomainService(
    nature=NatureOfService.VIEW_CONTRIBUTIONS_ONLY                ①
)
@DomainServiceLayout(
    menuOrder="10",
    name="...",
}
public class 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 contributed services).

6.2.6. 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.

6.2.7. Registering

The easiest way to register domain services with the framework is to use an `AppManifest`. This specifies 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.

6.2.8. 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>`. 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`.

6.3. Property

A property is an instance variable of a domain object, of a scalar type, that holds some state about either a domain entity or a view model.

For example, a `Customer`'s `firstName` would be a property, as would their `accountCreationDate` that they created their account. All properties have at least a "getter" method, and most properties have also a "setter" method (meaning that they are mutable). Properties that do *not* have a setter method are derived properties, and so are not persisted.

Formally speaking, a property is simply a regular JavaBean getter, returning a scalar value recognized by the framework. Most properties (those that are editable/modifiable) will also have a setter and (if persisted) a backing instance field. And most properties will also have a number of annotations:

- Apache Isis defines its own set own `@Property` annotation for capturing domain semantics. It also provides a `@PropertyLayout` for UI hints (though the information in this annotation may instead be provided by a supplementary `.layout.xml` file
- the properties of domain entities are often annotated with the JDO/DataNucleus `@javax.jdo.annotations.Column` annotation. For property references, there may be other annotations to indicate whether the reference is bidirectional. It's also possible (using annotations) to define a link table to hold foreign key columns.
- for the properties of view models, then JAXB annotations such as `@javax.xml.bind.annotation.XmlElement` will be present

Apache Isis recognises some of these annotations for JDO/DataNucleus and JAXB and infers some domain semantics from them (for example, the maximum allowable length of a string property).

Since writing getter and setter methods adds quite a bit of boilerplate, it's common to use Project Lombok to code generate these methods at compile time (using Java's annotation processor) simply by adding the `@lombok.Getter` and `@lombok.Setter` annotations to the field. The HelloWorld and SimpleApp archetypes use this approach.

6.3.1. Value vs Reference Types

Properties can be either a value type (strings, int, date and so on) or be a reference to another object (for example, an `Order` referencing the `Customer` that placed it).

For example, to map a string value type:

```
@lombok.Getter @lombok.Setter       ①
private String notes;
```

① using Project Lombok annotations to reduce boilerplate

You could also add the `@Property` annotation if you wished:

```
@Property
@lombok.Getter @lombok.Setter
private String notes;
```

Although in this case it is not required (none of its attributes have been set).

Or to map a reference type:

```
@lombok.Getter @lombok.Setter
private Customer customer;
```

It's ok for a domain entity to reference another domain entity, and for a view model to reference both view model and domain entities. However, it isn't valid for a domain entity to hold a persisted reference to view model (DataNucleus will not know how to persist that view model).

For further details on mapping associations, see the JDO/DataNucleus documentation for one-to-many associations, many-to-one associations, many-to-many associations, and so on.

For domain entities, 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.

6.3.2. Optional Properties

(For domain entities) 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=…)`, or (because it's much less verbose) using `@javax.annotation.Nullable`.

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, you can write:

```
@javax.jdo.annotations.Column(allowsNull="true")
@lombok.Getter @lombok.Setter
private LocalDate date;
```

rather than the more verbose:

```
@javax.jdo.annotations.Column(allowsNull="true")
@Property(optionality=Optionality.OPTIONAL)
@lombok.Getter @lombok.Setter
private LocalDate date;
```

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).

6.3.3. Editable Properties

Apache Isis provides the capability to allow individual properties to be modified. This is specified using the `@Property(editing=…)` attribute.

For example:

```
@Property(editing = Editing.ENABLED)
@lombok.Getter @lombok.Setter
private String notes;
```

If this is omitted then whether editing is enabled or disabled is defined globally, in the `isis.properties` configuration file; see reference configuration guide for further details.

6.3.4. Ignoring Properties

By default Apache Isis will automatically render all properties in the Wicket UI or in the REST API. To get Apache Isis to ignore a property (exclude it from its metamodel), annotate the getter using `@Programmatic`.

Similarly, you can tell JDO/DataNucleus to ignore a property using the `@javax.jdo.annotations.NotPersistent` annotation. This is independent of Apache Isis; in other words that property will still be rendered in the UI (unless also annotated with `@Programmatic`).

For view models, you can tell JAXB to ignore a property using the `@javax.xml.bind.annotation.XmlTransient` annotation. Again, this is independent of Apache Isis.

6.3.5. Derived Properties

Derived properties are those with a getter but no setter. Provided that the property has not been annotated with `@Programmatic`, these will still be rendered in the UI, but they will be read-only (not

editable) and their state will not be persisted.

Subtly different, it is also possible to have non-persisted but still editable properties. In this case you will need a getter and a setter, but with the getter annotated using `@NotPersistent`. The implementation of these getters and setters will most likely persist state using other properties (which might be hidden from view using `@Programmatic`).

For example:

```
@javax.jdo.annotations.NotPersistent
@Property(editing=Editing.ENABLED)
public String getAddress() { return addressService.toAddress( getLatLong() ); }
①
public void setAddress(String address) { setLatLong(addressService.toLatLong(address)
); }

@javax.jdo.annotations.Column
private String latLong;
@Programmatic
public String getLatLong() { return latLong; }
②
public void setLatLong(String latLong) { this.latLong = latLong; }

@javax.inject.Inject
AddressService addressService;
③
```

① the representation of the address, in human readable form, eg "10 Downing Street, London, UK"

② the lat/long representation of the address, eg "51.503363;-0.127625"

③ an injected service that can convert to/from address and latLong.

6.3.6. Data types

This section shows specific considerations for various datatypes, in particular how to annotate them for DataNucleus mapping to the persistence object store.

`String`s (Length)

By default JDO/DataNucleus will map string properties to a `VARCHAR(255)`. To limit the length, use the `@Column(length=…)` annotation.

For example:

```
@javax.jdo.annotations.Column(length=50)
@lombok.Getter @lombok.Setter
private String firstName
```

This is a good example of a case where Apache Isis infers domain semantics from the JDO

annotation.

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:

```
@javax.jdo.annotations.Persistent(defaultFetchGroup="true")
@javax.jdo.annotations.Column(allowsNull="true")
@Getter @Setter
private LocalDate dueBy;
```

`BigDecimal`s (Precision)

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

For example, suppose we have:

```
@lombok.Getter @lombok.Setter
private BigDecimal 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)
@lombok.Getter @lombok.Setter
private BigDecimal impact;
```

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

More information can be found here and here.

`Blob`s

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.

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")
})
@Property(
        optionality = Optionality.OPTIONAL
)
@lombok.Getter @lombok.Setter
private Blob 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")
})
@Property(
    optionality = Optionality.MANDATORY
)
@lombok.Getter @lombok.Setter
private Blob attachment;
```

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

`Clob`s

Mapping `Clob`s works in a very similar way to Blobs, 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.

6.4. Collections

A collection is an instance variable of a domain object, of a collection type that holds references to other domain objects. For example, a `Customer` may have a collection of `Order`s).

It's ok for a domain entity to reference another domain entity, and for a view model to reference both view model and domain entities. However, it isn't valid for a domain entity to hold a persisted reference to view model (DataNucleus will not know how to persist that view model).

Formally speaking, a collection is simply a regular JavaBean getter, returning a collection type (subtype of `java.util.Collection`). Most collections (those that are modifiable) will also have a setter and (if persisted) a backing instance field. And collections properties will also have a number of annotations:

- Apache Isis defines its own set own `@Collection` annotation for capturing domain semantics. It also provides a `@CollectionLayout` for UI hints (though the information in this annotation may instead be provided by a supplementary `.layout.xml` file
- the collections of domain entities are often annotated with various JDO/DataNucleus annotations, most notable `javax.jdo.annotations.Persistent`. This and other annotations can be used to specify if the association is bidirectional, and whether to define a link table or not to hold foreign key columns.
- for the collections of view models, then JAXB annotations such as `@javax.xml.bind.annotation.XmlElementWrapper` and `@javax.xml.bind.annotation.XmlElement` will be present

Apache Isis recognises some of these annotations for JDO/DataNucleus and JAXB and infers some domain semantics from them (for example, the maximum allowable length of a string property).

Unlike properties, the framework (at least, the Wicket viewer) does not allow collections to be "edited". Instead, actions can be written that will modify the contents of the collection as a side-effect. For example, a `placeOrder(…)` action will likely add an `Order` to the `Customer#orders` collection.

Since writing getter and setter methods adds quite a bit of boilerplate, it's common to use Project

Lombok to code generate these methods at compile time (using Java's annotation processor) simply by adding the `@lombok.Getter` and `@lombok.Setter` annotations to the field.

6.4.1. Mapping bidir 1:m

Bidirectional one-to-many collections are one of the most common types of associations between two entities. In the parent object, the collection can be defined as:

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

    @javax.jdo.annotations.Persistent(
        mappedBy = "parent",                                          ①
        dependentElement = "false"                                    ②
    )
    @Collection                                                       ③
    @lombok.Getter @lombok.Setter
    private SortedSet<ChildObject> children = new TreeSet<ChildObject>();   ④

}
```

① indicates a bidirectional association; the foreign key pointing back to the `Parent` will be in the table for `ChildObject`

② disable cascade delete

③ (not actually required in this case, because no attributes are set, but acts as a useful reminder that this collection will be rendered in the UI by Apache Isis)

④ uses a `SortedSet` (as opposed to some other collection type; discussion below)

while in the child object you will have:

```
public class ChildObject
        implements Comparable<ChildObject> {    ①

    @javax.jdo.annotations.Column(
        allowsNull = "false"                    ②
    )
    @Property(editing = Editing.DISABLED)       ③
    @lombok.Getter @lombok.Setter
    private ParentObject parent;
}
```

① implements `Comparable` because is mapped using a `SortedSet`

② mandatory; every child must reference its parent

③ cannot be edited directly

Generally speaking you should use `SortedSet` for collection types (as opposed to `Set`, `List` or `Collection`). JDO/Datanucleus does support the mapping of these other types, but RDBMS are set-

oriented, so using this type introduces the least friction.

For further details on mapping associations, see the JDO/DataNucleus documentation for one-to-many associations, many-to-one associations, many-to-many associations, and so on.

Also, while JDO/DataNucleus itself supports `java.util.Map` as a collection type, this is not supported by Apache Isis. If you do wish to use this collection type, then annotate the getter with `@Programmatic` so that it is ignored by the Apache Isis framework.

6.4.2. Value vs Reference Types

Apache Isis can (currently) only provide a UI for collections of references. While you can use DataNucleus to persist collections/arrays of value types, such properties must be annotated as `@Programmatic` so that they are ignored by Apache Isis.

If you want to visualize an array of value types in Apache Isis, then one option is to wrap value in a view model, as explained elsewhere.

6.4.3. Derived Collections

A derived collection is simply a getter (no setter) that returns a `java.util.Collection` (or subtype).

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);
    }
}
```

6.5. Actions

While properties and collections define the state held by a domain object (its "know what" responsibilities), actions define the object's behaviour (its "know how-to" responsibilities).

An application whose domain objects have only/mostly "know-what" responsibilities is pretty dumb: it requires that the end-user know the business rules and doesn't modify the state of the domain objects such that they are invalid (for example, an "end date" being before a "start date"). Such applications are often called CRUD applications (create/read/update/delete).

In more complex domains, it's not realistic/feasible to expect the end-user to have to remember all

the different business rules that govern the valid states for each domain object. So instead actions allow those business rules to be encoded programmatically. An Apache Isis application doesn't try to constrain the end-user as to way in which they interact with the user (it doesn't attempt to define a rigid business process) but it does aim to ensure that business rule invariants are maintained, that is that business objects aren't allowed to go into an invalid state.

For simple domain applications, you may want to start prototyping only with properties, and only later introduce actions (representing the most common business operations). But an alternative approach, recommended for more complex applications, is actually to start the application with all properties non-editable. Then, as the end-user requires the ability to modify some state, there is a context in which to ask the question "why does this state need to change?" and "are their any side-effects?" (ie, other state that changes at the same time, or other behaviour that should occur). If the state change is simple, for example just being able to correct an invalid address, or adding a note or comment, then that can probably be modelled as a simple editable property. But if the state change is more complex, then most likely an action should be used instead.

6.5.1. Defining actions

Broadly speaking, actions are all the `public` methods that are not getters or setters which represent properties or collections. This is a slight simplification; there are a number of other method prefixes (such as `hide` or `validate`) that represent business rules); these also not treated as actions. And, any method that are annotated with `@Programmatic` will also be excluded. But by and large, all other methods such as `placeOrder(…)` or `approveInvoice(…)` will be treated as actions.

For example:

```
@Action(semantics=SemanticsOf.IDEMPOTENT)       ①
public ShoppingBasket addToBasket(
        Product product,
        @ParameterLayout(named="Quantity")      ②
        int quantity
        ) {
    ...
    return this;
}
```

① `@Action` annotation is optional but used to specify additional domain semantics (such as being idempotent).

② The names of action parameters (as rendered in the UI) will by default be the parameter types, not the paramter names. For the `product` parameter this is reasonable, but not so for the `quantity` parameter (which would by default show up with a name of "int". The `@ParameterLayout` annotation provides a UI hint to the framework.

The (non-ASF) Incode Platform's paraname8 metamodel extension allows the parameter name to be used in the UI, rather than the type.

6.5.2. (Reference) Parameter types

Parameter types can be value types or reference types. In the case of primitive types, the end-user can just enter the value directly through the parameter field. In the case of reference types however (such as `Product`), a drop-down must be provided from which the end-user to select. This is done using either a supporting `choices` or `autoComplete` method. The "choices" is used when there is a limited set of options, while "autoComplete" is used when there are large set of options such that the end-user must provide some characters to use for a search.

For example, the `addToBasket(…)` action shown above might well have a :

```
@Action(semantics=SemanticsOf.IDEMPOTENT)
public ShoppingBasket addToBasket(
        Product product,
        @ParameterLayout(named="Quantity")
        int quantity
        ) {
    ...
    return this;
}
public List<Product> autoComplete0AddToBasket(              ①
    @MinLength(3)                                           ②
    String searchTerm) {
    return productRepository.find(searchTerm);              ③
}
@javax.inject.Inject
ProductRepository productRepository;
```

① Supporting `autoComplete` method. The "0" in the name means that this corresponds to parameter 0 of the "addToBasket" action (ie `Product`). It is also required to return a Collection of that type.

② The `@MinLength` annotation defines how many characters the end-user must enter before performing a search.

③ The implementation delegates to an injected repository service. This is typical.

Note that it is also valid to define "choices" and "autoComplete" for value types (such as `quantity`, above); it just isn't as common to do so.

Removing boilerplate

To save having to define an `autoCompleteNXxx(…)` method everywhere that a reference to a particular type (such as `Product`) appears as an action parameter, it is also possible to use the `@DomainObject` annotation on `Product` itself:

```
@DomainObject(
    autoCompleteRepository=ProductRepository.class          ①
    autoCompleteAction="find"                               ②
)
public class Product ... {
    ...
}
```

① Whenever an action parameter requiring a `Product` is defined, provide an autoComplete drop-down automatically

② Use the "find" method of `ProductRepository` (rather than the default name of "autoComplete")

(As noted above), if the number of available instances of the reference type is a small number (in other words, all of which could comfortably be shown in a drop-down) then instead the `choicesNXxx()` supporting method can be used. This too can be avoided by annotating the referenced class.

For example, suppose we have an action to specify the `PaymentMethodType`, where there are only 10 or so such (Visa, Mastercard, Amex, Paypal etc). We could define this as:

```
public Order payUsing(PaymentMethodType type) {
    ...
}
```

where `PaymentMethodType` would be annotated using:

```
@DomainObject(
    bounded=true                                ①
)
public class PaymentMethodType ... {
    ...
}
```

① only a small (ie "bounded") number of instances available, meaning that the framework should render all in a drop-down.

6.5.3. Collection Parameter types

Action parameters can also be collections of values (for example `List<String>`), or can be collections of references (such as `List<Customer>`).

For example:

```
@Action(semantics=SemanticsOf.IDEMPOTENT)
public ShoppingBasket addToBasket(
        List<Product> products,
        @ParameterLayout(named="Quantity") int quantity
        ) {
    ...
    return this;
}
public List<Product> autoComplete0AddToBasket(@MinLength(3) String searchTerm) {
    return ...
}
```

As the example suggests, any collection parameter type must provide a way to select items, either by way of a "choices" or "autoComplete" supporting method or alternatively defined globally using `@DomainObject` on the referenced type (described above).

6.5.4. Optional Parameters

Whereas the optionality of properties is defined using `@javax.jdo.annotations.Column#allowsNull()`, that JDO annotation cannot be applied to parameter types. Instead, either the `@Nullable` annotation or the `@Parameter#optionality()` annotation/attribute is used.

For example:

```
@javax.jdo.annotations.Column(allowsNull="true")                ①
@lombok.Getter @lombok.Setter
private LocalDate shipBy;

public Order invoice(
                PaymentMethodType paymentMethodType,
                @Nullable                                       ②
                @ParameterLayout(named="Ship no later than")
                LocalDate shipBy) {
    ...
    setShipBy(shipBy)
    return this;
}
```

① Specifies the property is optional.

② Specifies the corresponding parameter is optional.

See also properties vs parameters.

6.5.5. `String` Parameters (Length)

Whereas the length of string properties is defined using `@javax.jdo.annotations.Column#length()`, that JDO annotation cannot be applied to parameter types. Instead, the `@Parameter#maxLength()` annotation/attribute is used.

For example:

```
@javax.jdo.annotations.Column(length=50)                ①
@lombok.Getter @lombok.Setter
private String firstName;

@javax.jdo.annotations.Column(length=50)
@lombok.Getter @lombok.Setter
private String lastName;

public Customer updateName(
                @Parameter(maxLength=50)                ②
                @ParameterLayout(named="First name")
                String firstName,
                @Parameter(maxLength=50)
                @ParameterLayout(named="Last name")
                String lastName) {
    setFirstName(firstName);
    setLastName(lastName);
    return this;
}
```

① Specifies the property length using the JDO `@Column#length()` annotation

② Specifies the parameter length using the (Apache Isis) `@Parameter#maxLength()` annotation

Incidentally, note in the above example that the new value is assigned to the properties using the setter methods; the action does not simply set the instance field directly. This is important, because it allows JDO/DataNucleus to keep track that this instance variable is "dirty" and so needs flushing to the database table before the transaction completes.

See also properties vs parameters.

6.5.6. `BigDecimal`s (Precision)

Whereas the precision of BigDecimal properties is defined using `@javax.jdo.annotations.Column#scale()`, that JDO annotation cannot be applied to parameter types. Instead, the `@javax.validation.constraints.Digits#fraction()` annotation/attribute is used.

For example:

```
@javax.jdo.annotations.Column(scale=2)                                  ①
@lombok.Getter @lombok.Setter
private BigDecimal discountRate;

public Order updateDiscount(
                @javax.validation.constraints.Digits(fraction=2)    ②
                @ParameterLayout(named="Discount rate")
                String discountRate) {
    setDiscountRate(discountRate);
    return this;
}
```

① Specifies the property precision using `@Column#scale()`

② Specifies the corresponding parameter precision using `@Digits#fraction()`.

See also properties vs parameters.

6.6. 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 framework defines many additional services (such as `RepositoryService`); these are injected in exactly the same manner.

Sometimes there may be multiple services that implement a single type. This is common for example for SPI service, whereby one module defines an SPI service, and other module(s) in the application implement that service. To support this, the framework also allows lists of services to be injected.

When there are multiple service implementations of a given type, the framework will inject the service with highest priority, as defined through `@DomainService#menuOrder()` (even for domain services that are not menus), lowest first. If a list of services is injected, then that list will be ordered according to `menuOrder`, again lowest first.

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.

6.6.1. Field Injection

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

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

To inject a list of services, use:

```
public class DocumentService {
    ...
    @javax.inject.Inject
    List<PaperclipFactory> paperclipFactories;
}
```

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).

6.6.2. Method Injection

The framework 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;
    }
    ...
}
```

Lists of services can be injected in a similar manner.

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

6.6.3. 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).

6.7. Properties vs Parameters

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.

Since JDO/DataNucleus handles persistence, its annotations are requiredto specify semantics such as optionality or maximum length on properties. However, 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	(JDO) property	action parameter
string (length)	`@javax.jdo.annotations.Column(length=50)`	`@javax.jdo.annotations.Parameter(maxLength=50)`
big decimal (precision)	`@javax.jdo.annotations.Column(scale=2)`	`@javax.validation.constraints.Digits(fraction=2)`
optionality	`@Column(allowsNull="true")`	`@Nullable` or `ParameterLayout(optionality=Optionality.OPTIONAL)` (also `@Optional`, now deprecated)

6.8. View Models

As described in the introduction, view models are domain objects like domain entities, with behaviour and state. However, unlike domain entities, their state is _not_ persisted to a database but is instead serialized into its identifier (in effect, its URL).

The framework provides two main ways to implement a view model:

- The more powerful/flexible approach is to use JAXB annotations; this allows the state of the object's properties and also its collections.
- The other approach is to use Apache Isis specific annotations. While (arguably) these explain the intent of the view model better, they are more restrictive: only the state of the object's properties is serialized — collections are ignored — and not every datatype is recognized.

The serialized form of these view models is therefore XML, which also enables these view models to act as DTO (useful for various integration scenarios).

For these reasons we recommend that you use JAXB-style view models wherever possible. Indeed, the legacy approach for view models may be deprecated in the future.

In the sections below we consider JAXB view models both as "regular" view models, and also when using them to act as DTOs.

6.8.1. JAXB View Models

Here's a typical example of a JAXB view model, to allow (certain properties of) two `Customer`s to be compared:

```
@XmlRootElement(name = "compareCustomers")          ①
@XmlType(
        propOrder = {                               ②
            "customer1",
            "customer2"
        }
)
@XmlAccessorType(XmlAccessType.FIELD)               ③
public class CompareCustomers {

    @XmlElement(required = true)                    ④
    @Getter @Setter
    Customer customer1;

    @XmlElement(required = true)                    ④
    @Getter @Setter
    Customer customer2;

    @XmlTransient                                   ⑤
    public String getCustomer1Name() {
        return getCustomer1().getName();
    }

    @XmlTransient                                   ⑤
    public String getCustomer1Address() {
        return getCustomer1().getAddress();
    }

    ...
}
```

① The JAXB `@XmlRootElement` annotation indicates this is a view model to Apache Isis, which then uses JAXB to serialize the state of the view model between interactions

② All properties of the view model must be listed using the `XmlType#propOrder` attribute.

③ Specifying field accessor type allows the Lombok `@Getter` and `@Setter` annotations to be used.

④ The `XmlElement` indicates the property is part of the view model's state. For collections, the `@XmlElementWrapper` would also typically be used.

⑤ The `@XmlTransient` indicates that the property is derived and should be ignored by JAXB.

Use JAXB elements such as `@XmlElement` for properties and the combination of `@XmlElementWrapper` and `@XmlElement` for collections. Properties can be ignored (for serialization) using `@XmlTransient`.

The derived properties could also have been implemented using mixins.

Be aware that all the state will become the DTO's memento, ultimately converted into a URL-safe form, by way of the `UrlEncodingService`.

There are limits to the lengths of URLs, however. If the URL does exceed limits or contains invalid characters, then provide a custom implementation of `UrlEncodingService` to handle the memento string in some other fashion (eg substituting it with a GUID, with the memento cached somehow on the server). a URL.

Referencing Domain Entities

It's quite common for view models to be "backed by" (be projections of) some underlying domain entity. For example, the `CompareCustomers` view model described above actually references two underlying `Customer` entities.

It wouldn't make sense to serialize out the state of a persistent entity (even more so when the view model is also being used as a DTO). However, the identity of the underlying entity can be well defined; Apache Isis defines the common schema which defines the `<oid-dto>` element (and corresponding `OidDto` class): the object's type and its identifier. This is basically a formal XML equivalent to the `Bookmark` object obtained from the `BookmarkService`.

There is only one requirement to make this work: every referenced domain entity must be annotated with `@XmlJavaTypeAdapter`, specifying the framework-provided `PersistentEntityAdapter`. And this class is similar to the `BookmarkService`: it knows how to create an `OidDto` from an object reference.

Thus, in our view model we can legitimately write:

```
public class CompareCustomers {

    @XmlElement(required = true)
    @Getter @Setter
    Customer customer1;
    ...
}
```

All we need to do is remember to add that `@XmlJavaTypeAdapter` annotation to the referenced entity:

```
@XmlJavaTypeAdapter(PersistentEntityAdapter.class)
public class Customer ...  {
    ...
}
```

It's also possible for a DTO view models to hold collections of objects. These can be of any type, either simple properties, or references to other objects. The only bit of boilerplate that is required is the `@XmlElementWrapper` annotation. This instructs JAXB to create an XML element (based on the field name) to contain each of the elements. (If this is omitted then the contents of the collection are at the same level as the properties; almost certainly not what is required).

For example, we could perhaps generalize the view model to hold a set of `Customer`s to be compared:

```
public class CompareCustomers {
    ...
    @XmlElementWrapper
    @XmlElement(name = "customers")
    @Getter @Setter
    protected List<Customer> customersToCompare = Lists.newArrayList();
}
```

This capability is particularly useful when the JAXB view model is being used as a DTO.

JODA Time Datatypes

If your JAXB view model contains fields using the JODA datatypes (`LocalDate` and so on), then `@XmlJavaTypeAdapter` additional annotations in order to "teach" JAXB how to serialize out the state.

The Apache Isis applib provides a number of adapters to use out-of-the-box. For example:

```
@XmlRootElement(name = "categorizeIncomingInvoice")
@XmlType(
        propOrder = {
                ...
                "dateReceived",
                ...
        }
)
@XmlAccessorType(XmlAccessType.FIELD)
public class IncomingInvoiceViewModel extends IncomingOrderAndInvoiceViewModel {

    @XmlJavaTypeAdapter(JodaLocalDateStringAdapter.ForJaxb.class)
    private LocalDate dateReceived;

    ...
}
```

The full list of adapter classes are:

Table 2. JAXB adapters

JODA datatype	Adapter
`org.joda.time.DateTime`	`JodaDateTimeStringAdapter.ForJaxb`
	`JodaDateTimeXMLGregorianCalendarAdapter.ForJaxb`
`org.joda.time.LocalDate`	`JodaLocalDateStringAdapter.ForJaxb`
	`JodaLocalDateXMLGregorianCalendarAdapter.ForJaxb`
`org.joda.time.LocalDateTime`	`JodaLocalDateTimeStringAdapter.ForJaxb`
	`JodaLocalDateTimeXMLGregorianCalendarAdapter.ForJaxb`
`org.joda.time.LocalTime`	`JodaLocalTimeStringAdapter.ForJaxb`
	`JodaLocalTimeXMLGregorianCalendarAdapter.ForJaxb`
`java.sql.Timestamp`	`JavaSqlTimestampXmlGregorianCalendarAdapter.ForJaxb`

If you want to roll-your-own, take a look at this blog post.

6.8.2. DTOs

JAXB view models can also be used as DTOs. The examples in this section uses the DTO for `ToDoItem`, taken from the (non-ASF) Isis addons' todoapp.

This DTO is defined as follows:

```
package todoapp.app.viewmodels.todoitem.v1;                          ①
@XmlRootElement(name = "toDoItemDto")                                ②
@XmlType(
        propOrder = {                                                ③
            "majorVersion", "minorVersion",
            "description", "category", ...
            "toDoItem", "similarItems"
        }
)
@DomainObjectLayout(
        titleUiEvent = TitleUiEvent.Doop.class                       ④
)
public class ToDoItemV1_1 implements Dto {                           ⑤
    @XmlElement(required = true, defaultValue = "1")                 ⑥
    public final String getMajorVersion() { return "1"; }
    @XmlElement(required = true, defaultValue = "1")                 ⑦
    public String getMinorVersion() { return "1"; }

    @XmlElement(required = true)                                     ⑧
    @Getter @Setter
    protected String description;
    @XmlElement(required = true)
    @Getter @Setter
    protected String category;
    ...

    @Getter @Setter                                                  ⑨
    protected ToDoItem toDoItem;
    @XmlElementWrapper                                               ⑩
    @XmlElement(name = "todoItem")
    @Getter @Setter
    protected List<ToDoItem> similarItems = Lists.newArrayList();
}
```

① package name encodes major version; see discussion on versioning

② identifies this class as a view model and defines the root element for JAXB serialization

③ all properties in the class must be listed; (they can be ignored using `@XmlTransient`)

④ demonstrating use of UI events for a subscriber to provide the DTO's title; see `@DomainObjectLayout#titleUiEvent()`.

⑤ class name encodes (major and) minor version; see discussion on versioning

⑥ again, see discussion on versioning

⑦ again, see discussion on versioning

⑧ simple scalar properties

⑨ reference to a persistent entity; discussed here

⑩ reference to a collection of persistent entities; again discussed here

Versioning

The whole point of using DTOs (in Apache Isis, at least) is to define a formal contact between two inter-operating but independent applications. Since the only thing we can predicate about the future with any certainty is that it one or both of these applications will change, we should version DTOs from the get-go. This allows us to make changes going forward without unnecessarily breaking existing consumers of the data.

There are several ways that versioning might be accomplished; we base our guidelines on this article taken from Roger Costello's blog, well worth a read.

We can distinguish two types of changes:

- backwardly compatible changes
- breaking changes.

We can immediately say that the XSD namespace should change only when there is a major/breaking change, if following semantic versioning that means when we bump the major version number v1, v2, etc.

XML namespaces correspond (when using JAXB) to Java packages. We should therefore place our DTOs in a package that contains only the major number; this package will eventually contain a range of DTOs that are intended to be backwardly compatible with one another. The package should also have a `package-info.java`; it is this that declares the XSD namespace:

```
@javax.xml.bind.annotation.XmlSchema(
        namespace = "http://viewmodels.app.todoapp/todoitem/v1/Dto.xsd",        ①
        xmlns = {
                @javax.xml.bind.annotation.XmlNs(
                        namespaceURI = "http://isis.apache.org/schema/common",
                        prefix = "com"
                )
        },
        elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED
)
package todoapp.app.viewmodels.todoitem.v1;                                     ②
```

① the namespace URI, used by the DTO residing in this package.

② the package in which the DTO resides. Note that this contains only the major version.

Although there is no requirement for the namespace URI to correspond to a physical URL, it should be unique. This usually means including a company domain name within the string.

As noted above, this package will contain multiple DTO classes all with the same namespace; these represent a set of minor versions of the DTO, each subsequent one intended to be backwardly compatible with the previous. Since these DTO classes will all be in the same package (as per the advice above), the class should therefore include the minor version name:

```
package todoapp.app.viewmodels.todoitem.v1;     ①
...
public class ToDoItemV1_1 implements Dto {      ②
    ...
}
```

① package contains the major version only

② DTO class contains the (major and) minor version

We also recommend that each DTO instance should also specify the version of the XSD schema that it is logically compatible with. Probably most consumers will not persist the DTOs; they will be processed and then discarded. However, it would be wrong to assume that is the case in all cases; some consumers might choose to persist the DTO (eg for replay at some later state).

Thus:

```
public class ToDoItemV1_1 implements Dto {
    @XmlElement(required = true, defaultValue = "1")
    public final String getMajorVersion() { return "1"; }   ①
    @XmlElement(required = true, defaultValue = "1")
    public String getMinorVersion() { return "1"; }         ②
    ...
}
```

① returns the major version (in sync with the package)

② returns the minor version (in sync with the class name)

These methods always return a hard-coded literal. Any instances serialized from these classes will implicitly "declare" the (major and) minor version of the schema with which they are compatible. If a consumer has a minimum version that it requires, it can therefore inspect the XML instance itself to determine if it is able to consume said XML.

If a new (minor) version of a DTO is required, then we recommend copying-and-pasting the previous version, eg:

```
public class ToDoItemV1_2 implements Dto {
    @XmlElement(required = true, defaultValue = "1")
    public final String getMajorVersion() { return "1"; }
    @XmlElement(required = true, defaultValue = "2")
    public String getMinorVersion() { return "2"; }
    ...
}
```

Obviously, only changes made must be backward compatible, eg new members must be optional.

Alternatively, you might also consider simply editing the source file, ie renaming the class and bumping up the value returned by `getMinorVersion()`.

We also *don't* recommend using inheritance (ie `ToDoItemV1_2` should not inherit from `ToDoItemV1_1`; this creates unnecessary complexity downstream if generating XSDs and DTOs for the downstream consumer.

Generating XSDs and DTOs

In the section above it was explained how a view model DTO can transparent reference any "backing" entities; these references are converted to internal object identifiers.

However, if the consumer of the XML is another Java process (eg running within an Apache Camel route), then you might be tempted/expect to be able to use the same DTO within that Java process. After a little thought though you'll realize that (duh!) of course you cannot; the consumer runs in a different process space, and will not have references to those containing entities.

There are therefore two options:

- either choose not to have the view model DTO reference any persistent entities, and simply limit the DTO to simple scalars.

  Such a DTO will then be usable in both the Apache Isis app (to generate the original XML) and in the consumer. The `BookmarkService` can be used to obtain the object identifiers

- alternatively, generate a different DTO for the consumer from the XSD of the view model DTO.

The (non-ASF) Isis addons' todoapp uses the second approach; generating the XSD and consumer's DTO is mostly just boilerplate `pom.xml` file. In the todoapp this can be found in the `todoapp-xsd` Maven module, whose `pom.xml` is structured as two profiles:

```
<project ... >
    <artifactId>todoapp-xsd</artifactId>
    <dependencies>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>todoapp-app</artifactId>
        </dependency>
    </dependencies>
    <profiles>
        <profile>
            <id>isis-xsd</id>
            <activation>
                <property>
                    <name>!skip.isis-xsd</name>
                </property>
            </activation>
            ...
        </profile>
        <profile>
            <id>xjc</id>
            <activation>
                <property>
                    <name>!skip.xjc</name>
                </property>
            </activation>
            ...
        </profile>
    </profiles>
</project>
```

The `isis-xsd` profile generates the XSD using the `xsd` goal of Isis' maven plugin:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.isis.tool</groupId>
            <artifactId>isis-maven-plugin</artifactId>
            <version>${isis.version}</version>
            <configuration>
                <appManifest>todoapp.dom.ToDoAppDomManifest</appManifest>
                <jaxbClasses>
                    <jaxbClass>
todoapp.app.viewmodels.todoitem.v1.ToDoItemV1_1</jaxbClass>
                </jaxbClasses>
                <separate>false</separate>
                <commonSchemas>false</commonSchemas>
            </configuration>
            <dependencies>
                <dependency>
```

```
                    <groupId>${project.groupId}</groupId>
                    <artifactId>todoapp-dom</artifactId>
                    <version>${project.version}</version>
                </dependency>
                <dependency>
                    <groupId>com.google.guava</groupId>
                    <artifactId>guava</artifactId>
                    <version>16.0.1</version>
                </dependency>
            </dependencies>
            <executions>
                <execution>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>xsd</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>2.5.3</version>
            <configuration>
                <descriptor>src/assembly/dep.xml</descriptor>
            </configuration>
            <executions>
                <execution>
                    <id>create-archive</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The `todoapp.dom.ToDoAppDomManifest` is a cut-down version of the app manifest that identifies only the `dom` domain services.

The `xjc` profile, meanwhile, uses the `maven-jaxb2-plugin` (a wrapper around the `schemagen` JDK tool) to generate a DTO from the XSD generated by the preceding profile:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jvnet.jaxb2.maven2</groupId>
            <artifactId>maven-jaxb2-plugin</artifactId>
            <version>0.12.3</version>
```

```
            <executions>
                <execution>
                    <id>xjc-generate</id>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <removeOldOutput>true</removeOldOutput>
                <schemaDirectory>
                    target/generated-resources/isis-xsd/viewmodels.app.todoapp
                </schemaDirectory>
                <schemaIncludes>
                    <schemaInclude>todoitem/v1/Dto.xsd</schemaInclude>
                </schemaIncludes>
                <bindingDirectory>src/main/resources</bindingDirectory>
                <bindingIncludes>
                    <bindingInclude>binding.xml</bindingInclude>
                </bindingIncludes>
                <catalog>src/main/resources/catalog.xml</catalog>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>build-helper-maven-plugin</artifactId>
            <version>1.9.1</version>
            <executions>
                <execution>
                    <id>add-source</id>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>add-source</goal>
                    </goals>
                    <configuration>
                    <sources>
                        <source>target/generated-sources/xjc</source>
                    </sources>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

DTO Consumers

The actual consumers of DTOs will generally obtain the XML of the view models either by requesting the XML directly, eg using the RestfulObjects viewer, or may have the XML sent to them

asynchronously using an ESB such as Apache Camel.

In the former case, the consumer requests the DTO by calling the REST API with the appropriate HTTP `Accept` header. An appropriate implementation of `ContentMappingService` can then be used to return the appropriate DTO (as XML).

For the latter case, one design is simply for the application to instantiate the view model, then call the `JaxbService` to obtain its corresponding XML. This can then be published onto the ESB, for example using an Apache ActiveMQ ™ queue.

However, rather than try to push all the data that might be needed by any of these external systems in a single XML event (which would require anticipating all the requirements, likely a hopeless task), a better design is to publish only the fact that something of note has changed - ie, that an action on a domain object has been invoked - and then let the consumers call back to obtain other information if required. This can once again be done by calling the REST API with an appropriate HTTP `Accept` header.

This is an example of the VETRO pattern (validate, enrich, transform, route, operate). In our case we focus on the validation (to determine the nature of the inbound message, ie which action was invoked), and the enrich (callback to obtain a DTO with additional information required by the consumer).

The (non-ASF) Incode Platform's publishmq module provides an out-of-the-box solution of this design. It provides an implementation of the `PublishingService`, but which simply publishes instances of `ActionInvocationMemento` to an ActiveMQ queue. Camel (or similar) can then be hooked up to consume these events from this queue, and use a processor to parse the action memento to determine what has changed on the source system. Thereafter, a subsequent Camel processor can then call back to the source - via the Restful Objects viewer - to enrich the message with additional details using a DTO.

6.8.3. Non-JAXB View Models

Instead of using JAXB to specify a view model, it is also possible to use Apache Isis-specific annotations.

As was explained earlier, the approach is described here is neither as flexible nor as powerful as using the JAXB-style of view models. As such, it may be deprecated in the future.

While the underlying technique is the same irrespective of use case, the programming model provides various ways of defining a view model so that the original intent is not lost. They are:

Table 3. View model programming model

Use case	Code	Description
External entity	[source,java] ---- @DomainObject(nature=Nature.EXTERNAL_ENTITY) public class CustomerRecordOnSAP { … } ----	Annotated with `@DomainObject#nature()` and a nature of `EXTERNAL_ENTITY`, with memento derived automatically from the properties of the domain object. Collections are ignored, as are any properties annotated as not persisted.
In-memory entity	[source,java] ---- @DomainObject(nature=Nature.INMEMORY_ENTITY) public class Log4JAppender { … } ----	As preceding, but using a nature of `INMEMORY_ENTITY`.
Application view model	[source,java] ---- @DomainObject(nature=Nature.VIEW_MODEL) public class Dashboard { … } ----	As preceding, but using a nature of `VIEW_MODEL`.
Application view model	[source,java] ---- @ViewModel public class Dashboard { … } ----	Annotated with `@ViewModel` annotation (effectively just an alias)' memento is as preceding: from "persisted" properties, collections ignored
Application view model	[source,java] ---- public class ExcelUploadManager implements ViewModel { public String viewModelMemento() { … } public void viewModelInit(String memento) { … } }	Implement `ViewModel` interface. The memento is as defined by the interface's methods: the programmer has full control (but also full responsibility) for the string memento.

6.9. Mixins

A mixin acts like a trait or extension method, allowing one module to contribute behaviour or derived state to another object.

Syntactically, a mixin is defined using either the `@Mixin` annotation or using `@DomainObject#nature()` attribute (specifying a nature of `Nature.MIXIN`).

```
@Mixin(method="coll")                                   ①
public class Customer_orders {                          ②

    private final Customer customer;
    public Customer_orders(final Customer customer) {   ③
        this.customer = customer;
    }

    @Action(semantics=SemanticsOf.SAFE)                 ④
    @ActionLayout(contributed=Contributed.AS_ASSOCIATION) ④
    @CollectionLayout(render=RenderType.EAGERLY)
    public List<Order> coll() {                         ①
        return repositoryService.findOrdersFor(customer);
    }

    @Inject
    RepositoryService repositoryService;
}
```

① indicates that this is a mixin, with "coll" as the name of the main method

② The contributed member is inferred from the name, after the "_"; in other words "orders"

③ The mixee is `Customer`. This could also be an interface.

④ Indicates that the action should be interpreted as a collection. This requires that the action has safe semantics, ie does not alter state/no side-effects.

6.9.1. Contributed Collection

The example below shows how to contribute a collection:

```
@Mixin(method="coll")
public class DocumentHolder_documents {

    private final DocumentHolder holder;
    public DocumentHolderDocuments(DocumentHolder holder) { this.holder = holder; }

    @Action(semantics=SemanticsOf.SAFE)                 ①
    @ActionLayout(contributed = Contributed.AS_ASSOCIATION)  ②
    @CollectionLayout(render = RenderType.EAGERLY)
    public List<Document> coll() {                      ③
        ...
    }
    public boolean hideColl() { ... }                   ④
}
```

① required; actions that have side-effects cannot be contributed as collections

② required; otherwise the mixin will default to being rendered as an action

③ must accept no arguments. The mixin is a collection rather than a property because the return

type is a collection, not a scalar.

④ supporting methods follow the usual naming conventions. (That said, in the case of collections, because the collection is derived/read-only, the only supporting method that is relevant is `hideColl()`).

The above will result in a contributed collection "documents" for all types that implement/extend from `DocumentHolder`.

6.9.2. Contributed Property

Contributed properties are defined similarly, for example:

```
@Mixin(method="prop")
public class DocumentHolder_mostRecentDocument {

    private final DocumentHolder holder;
    public DocumentHolderDocuments(DocumentHolder holder) { this.holder = holder; }

    @Action(semantics=SemanticsOf.SAFE)                          ①
    @ActionLayout(contributed = Contributed.AS_ASSOCIATION)      ②
    public Document prop() {                                     ③
        ...
    }
    public boolean hiderProp() { ... }                           ④
}
```

① required; actions that have side-effects cannot be contributed as collections

② required; otherwise the mixin will default to being rendered as an action

③ must accept no arguments. The mixin is a property rather than a collection because the return type is a scalar.

④ supporting methods follow the usual naming conventions. (That said, in the case of properties, because the property is derived/read-only, the only supporting method that is relevant is `hideProp()`).

6.9.3. Contributed Action

Contributed actions are defined similarly, for example:

```
@Mixin(method="act")
public class DocumentHolder_addDocument {

    private final DocumentHolder holder;
    public DocumentHolderDocuments(DocumentHolder holder) { this.holder = holder; }

    @Action()
    @ActionLayout(contributed = Contributed.AS_ACTION)          ①
    public Document> act(Document doc) {
        ...
    }
    public boolean hideAct() { ... }                            ②
}
```

① recommended

② supporting methods follow the usual naming conventions.

6.9.4. Inferred Name

Where the mixin follows the naming convention `SomeType_mixinName` then the method name can be abbreviated, and the name of the member being contributed is inferred from the name of the class itself, beingeverything after the last '_'.

The default abbreviation to "$$".

For example:

```
@Mixin(method="act")
public class DocumentHolder_documents {

    private final DocumentHolder holder;
    public DocumentHolder_documents(DocumentHolder holder) { this.holder = holder; }

    @Action(semantics=SemanticsOf.SAFE)
    @ActionLayout(contributed = Contributed.AS_ASSOCIATION)
    @CollectionLayout(render = RenderType.EAGERLY)
    public List<Document> act() {
        ...
    }
    public boolean hideAct() { ... }
}
```

Alternatively, if the `@Mixin#method()` attribute is specified, then this can nominate a different abbreviation.

The examples above (for property, collection and action) demonstrate this.

The character "$" is also recognized as a separator between the mixin type and mixin name. This is

useful for mixins implemented as nested static types, discussed below.

6.9.5. As Nested Static Classes

As noted in the introduction, while mixins were originally introduced as a means of allowing contributions from one module to the types of another module, they are also a convenient mechanism for grouping functionality/behaviour against a concrete type. All the methods and supporting methods end up in a single construct, and the dependency between that functionality and the rest of the object is made more explicit.

When using mixins in this fashion, it is idiomatic to write the mixin as a nested static class, using the naming convention described above to reduce duplication.

For example:

```
public class Customer {

    @Mixin(method="act")
    public static class placeOrder {                                    ①

        private final Customer customer;
        public documents(Customer customer) { this.customer = customer; }   ②

        @Action
        @ActionLayout(contributed = Contributed.AS_ACTION)
        public List<Order> act(Product p, int quantity) {                    ③
            ...
        }
        public boolean hideAct() { ... }                                     ④
        public String validate0Act(Product p) { ...  }
    }
}
```

① Prior to 1.13.2, had to be prefixed by an "_"; this is no longer required because "$" is also recognized as a way of parsing the class name in order to infer the mixin's name (eg `Customer$placeOrder`).

② typically contributed to concrete class

③ supporting methods as usual

The mixin class can also be capitalized if desired. Thus:

```
public class Customer {

    @Mixin(method="act")
    public static class PlaceOrder {
        private final Customer customer;
        public documents(Customer customer) { this.customer = customer; }

        @Action
        @ActionLayout(contributed = Contributed.AS_ACTION)
        public List<Order> act(Product p, int quantity) {
            ...
        }
        public boolean hideAct() { ... }
        public String validate0Act(Product p) { ...  }
    }
}
```

In other words, all of the following are allowed:

- `public static class Documents { ⋯ }`
- `public static class documents { ⋯ }`
- `public static class _Documents { ⋯ }`
- `public static class _documents { ⋯ }`

6.9.6. Programmatic usage

When a domain object is rendered, the framework will automatically instantiate all required mixins and delegate to them dynamically. If writing integration tests or fixtures, or (sometimes) just regular domain logic, then you may need to instantiate mixins directly.

For this you can use the xref:../rgsvc/rgsvc.adoc#_rgsvc_core-domain-api_DomainObjectContainer_object-creation-api[`DomainObjectContainer#mixin(⋯)` method.

For example:

```
DocumentHolder_documents mixin = container.mixin(DocumentHolder_documents.class,
customer);
```

The `IntegrationTestAbstract` and `FixtureScript` classes both provide a `mixin(⋯)` convenience method.

6.9.7. Contributed services (deprecated)

Contributed services are very similar to mixins; indeed mixins are an evolution and refinement of the contributions concept. As such, contributions should be considered as deprecated, and eventually removed in a future version of the framework, to be replaced entirely by mixins.

The main difference between contributed services and mixins is that the actions of a contributed

service will contribute to *all* the parameters of its actions, whereas a mixin only contributes to the type accepted in its constructor.

Also, contributed services are long-lived singletons, whereas mixins are instantiated as required (by the framework) and then discarded almost immediately.

Syntax

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`.

Chapter 7. 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.

7.1. Layout

The most significant aspect of the UI is the layout of the object's members: its properties, collections and actions. These can be organized into columns, rows and tabs.

This can be accomplished using either annotations or through a separate file-based layout. Since this is a large topic, it has its own layout chapter in the Wicket viewer guide.

7.2. 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.

7.2.1. 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

Alternatively, the title can be provided simply by implementing the `title()` reserved method.

For example:

```
public class Author extends Customer {

    public String title() {
        StringBuilder buf = new StringBuilder();
        buf.append(getFirstName());
        if(getMiddleInitial() != null) {
            buf.append(getMiddleInitial()).append(". ");
        }
        buf.append(getLastName();
        return buf.toString();
    }
    ...
}
```

A variation on this approach also supports localized names; see beyond-the-basics guide for further details.

Using a UI subscriber

A third alternative is to move the responsibility for deriving the title into a separate subscriber object.

In the target object, we define an appropriate event type and use the `@DomainObjectLayout#titleUiEvent()` attribute to specify:

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

The subscriber can then populate this event:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class AuthorSubscriptions extends AbstractSubscriber {

    @org.axonframework.eventhandling.annotation.EventHandler
    @com.google.common.eventbus.Subscribe
    public void on(Author.TitleUiEvent ev) {
        Author author = ev.getSource();
        ev.setTitle(titleOf(author);
    }

    private String titleOf(Author author) {
        StringBuilder buf = new StringBuilder();
        buf.append(author.getFirstName());
        if(author.getMiddleInitial() != null) {
            buf.append(author.getMiddleInitial()).append(". ");
        }
        buf.append(author.getLastName();
        return buf.toString();
    }
}
```

7.2.2. Object Icon

The icon is often the same for all instances of a particular class, and is picked up by convention.

It's is also possible for an individual instance to return a custom icon, typically so that some significant state of that domain object is represented. For example, a custom icon could be used to represent a shipped order, say, or an overdue library loan.

Declarative style

If there is no requirement to customize the icon (the normal case), then the icon is usually picked up as the `.png` file in the same package as the class. For example, the icon for a class `org.mydomain.myapp.Customer` will be `org/mydomain/myapp/Customer.png` (if it exists).

Alternatively, font-awesome icon can be used. This is specified using the `@DomainObjectLayout#cssClassFa()` attribute.

For example:

```
@DomainObjectLayout(
    cssClassFa="play"                ①
)
public class InvoiceRun {
    ...
}
```

① will use the "fa-play" icon.

Imperative style

To customise the icon on an instance-by-instance basis, we implement the reserved `iconName()` method.

For example:

```
public class Order {
    public String iconName() {
        return isShipped() ? "shipped": null;
    }
    ...
}
```

In this case, if the `Order` has shipped then the framework will look for an icon image named "Order-shipped.png" (in the same package as the class). Otherwise it will just use "Order.png", as normal.

Using a UI subscriber

As for title, the determination of which image file to use for the icon can be externalized into a UI event subscriber.

In the target object, we define an appropriate event type and use the `@DomainObjectLayout#iconUiEvent()` attribute to specify.

For example

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

The subscriber can then populate this event:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class OrderSubscriptions extends AbstractSubscriber {

    @org.axonframework.eventhandling.annotation.EventHandler
    @com.google.common.eventbus.Subscribe
    public void on(Order.IconUiEvent ev) {
        Order order = ev.getSource();
        ev.setIconName(iconNameOf(order);
    }

    private String iconNameOf(Order order) {
        StringBuilder buf = new StringBuilder();
        return order.isShipped() ? "shipped": null;
    }
}
```

7.2.3. 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.

Declarative style

To render an object with a particular CSS, use `@DomainObjectLayout#cssClass()`.

When the domain object is rendered on its own page, this CSS class will appear on a top-level `<div>`. Or, when the domain object is rendered as a row in a collection, then the CSS class will appear in a `<div>` wrapped by the `<tr>` of the row.

One possible use case would be to render the most important object types with a subtle background colour: `Customer`s shown in light green, or `Order`s shown in a light pink, for example.

Imperative style

To customise the icon on an instance-by-instance basis, we implement the reserved `cssClass()` method.

For example:

```
public class Order {
    public String cssClass() {
        return isShipped() ? "shipped": null;       ①
    }
    ...
}
```

① the implementation might well be the same as the `iconName()`.

If non-null value is returned then the CSS class will be rendered *in addition* to any declarative CSS class also specified.

Using a UI subscriber

As for title and icon, the determination of which CSS class to render can be externalized into a UI event subscriber.

In the target object, we define an appropriate event type and use the `@DomainObjectLayout#cssClassUiEvent()` attribute to specify.

For example

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

The subscriber can then populate this event:

```
@DomainService(nature=NatureOfService.DOMAIN)
public class OrderSubscriptions extends AbstractSubscriber {

    @org.axonframework.eventhandling.annotation.EventHandler
    @com.google.common.eventbus.Subscribe
    public void on(Order.CssClassUiEvent ev) {
        Order order = ev.getSource();
        ev.setIconName(iconNameOf(order);
    }

    private String cssClassOf(Order order) {
        StringBuilder buf = new StringBuilder();
        return order.isShipped() ? "shipped": null;
    }
}
```

7.3. 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 hints can be applied either to individual actions, or can be applied en-masse using pattern matching.

It is also possible to specify additional CSS for an object's members (not just actions).

7.3.1. Icons

Action icons can be specified in several ways.

One option is to use the `@ActionLayout#cssClassFa()`. For example:

```
@ActionLayout(cssClassFa="refresh")
public void renew() {
    ...
}
```

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

```
<cpt:action id="renew" cssClassFa="refresh"/>
```

Rather than annotating every action with `@ActionLayout#cssClassFa()` and `@ActionLayout#cssClass()` you can instead specify the UI hint globally using regular expressions. Not only does this save a lot of boilerplate/editing, it helps ensure consistency across all actions.

To declare fa classes globally, use the configuration property `isis.reflector.facet.cssClassFa.patterns` (a comma separated list of key:value pairs).

For example:

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

Here:

- the key is a regex matching action names (eg `create.*`), and
- the value is a font-awesome icon name

For example, "fa-plus" is applied to all action members called "newXxx"

7.3.2. CSS

Similarly, a CSS class can be specified for object members:

- `@ActionLayout#cssClass()` for actions
- `@PropertyLayout#cssClass()` for properties, and
- `@CollectionLayout#cssClass()` for collections.

Again, this CSS class will be attached to an appropriate containing `<div>` or `<span>` on the rendered page.

Possible use cases for this is to highlight the most important properties of a domain object.

It is also possible to specify CSS classes globally, using the configuration property `isis.reflector.facet.cssClass.patterns` configuration property.

For example:

```
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

7.4. Names and Descriptions

The name of classes and class members are usually inferred from the Java source code directly. For example, an action method called `placeOrder` will be rendered as "Place Order", and a collection called `orderItems` is rendered as "Order Items".

Occasionally though the desired name is not possible; either the name is a Java reserved word (eg "class"), or might require characters that are not valid, for example abbreviations.

In such cases the name can be specified declaratively.

It is also possible to specify a description declaratively; this is used as a tooltip in the UI.

The table below summarizes the annotations available:

Table 4. Names and descriptions

Feature	Named	Description
Class	`@DomainObjectLayout#named()`	`@DomainObjectLayout#describedAs()`
Property	`@PropertyLayout#named()`	`@PropertyLayout#describedAs()`
Collection	`@CollectionLayout#named()`	`@CollectionLayout#describedAs()`
Action	`@ActionLayout#named()`	`@ActionLayout#describedAs()`

Feature	Named	Description
Action Parameters	`@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 (non-ASF) Incode Platform's paraname8 metamodel extension (non-ASF). (In the future we'll fold this into core). See also our guidance on upgrading to Java 8.

The framework also supports i18n: locale-specific names and descriptions. for more information, see the beyond-the-basics guide.

7.5. Eager rendering

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

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

For this we annotate the collection using `@CollectionLayout#defaultView()`; for example

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

The `defaultView()` attribute replaces the deprecated approach of using `@CollectionLayout#render()` eagerly.

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

```
<bs3:col span="5" cssClass="custom-padding-top-20">
    <cpt:collection id="notYetComplete" defaultView="table" />
    <cpt:collection id="complete" defaultView="table"/>
</bs3col>
```

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 irrespective of how the object is used/rendered.

Of course, your mileage may vary, so do experiment.

Chapter 8. Object Management (CRUD)

This chapter shows the idioms for creating, reading, updating and deleting domain entities.

It also shows how to instantiate view models and how to programmatically instantiate mixins (useful primarily for integration testing).

The main domain services used are `RepositoryService` and `FactoryService`. These (and some other services) replace the now deprecated `DomainObjectContainer`.

8.1. Instantiating

Both domain entities and view models can be instantiated using the `FactoryService` provided by the framework. For example:

```
Customer customer = factoryService.instantiate(Customer.class);
```

or

```
Dashboard dashboardVM = factoryService.instantiate(Dashboard.class);
```

When the framework instantiates the object, all services are injected into the framework, and an `ObjectCreatedEvent` lifecycle event will also be emitted.

For this to work the target class *must* have a no-arg constructor.

However, you may prefer for your domain objects to have regular constructor defining their minimum set of mandatory properties.

For example:

```
public class Customer {

    public Customer(String reference, String firstName, String lastName) { ... }

    ...
}
```

In such cases, the domain object cannot be instantiated using `FactoryService`. Instead the `ServiceRegistry` service can be used to inject services:

```
Customer customer = new Customer(reference, firstName, lastName);
serviceRegister.injectServicesInto(customer);
```

Note that this does *not* raise any lifecycle event.

To instantiate a mixin, a different API of `FactoryService` is used:

```
Customer_placeOrder placeOrderMixin = factoryService.mixin(Customer_placeOrder.class,
customer);
```

Alternatively and equivalently, the mixin can be manually instantiated, and then the services injected directly:

```
Customer_placeOrder placeOrderMixin = new Customer_placeOrder(customer);
serviceRegistry.injectServicesInto(placeOrderMixin);
```

8.2. Persisting

Once a domain entity has been instantiated, it must be persisted. This is done using the `RepositoryService`. For example:

```
Customer customer = ...

repositoryService.persist(customer);
```

If using the no-arg form to instantiate the entity, then (to save having to inject two services), the `RepositoryService` can also be used to instantiate. This gives rise to this common idiom:

```
Customer customer = repositoryService.instantiate(Customer.class);
customer.setReference(reference);
customer.setFirstName(firstName);
customer.setLastName(lastName);
...
repositoryService.persist(customer);
```

It's worth being aware that the framework does *not* eagerly persist the object. Rather, it queues up an internal command structure representing the object persistence request. This is then executed either at the end of the transaction, or if a query is run, or if the internal queue is manually flushed using `TransactionService`'s `flush()` method.

Alternatively, you can use:

```
repositoryService.persistAndFlush(customer);
```

to eagerly perform the object insertion into the database.

When an object is persisted the framework will emit `ObjectPersistingEvent` and

`ObjectPersistedEvent` lifecycle events.

It is also possible to configure DataNucleus to automatically persist domain entities if they are associated with other already-persistent entities. This avoid the need to explicitly call "persist".

For this, configure the persistence-by-reachability property:

isis.properties

```
isis.persistor.datanucleus.impl.datanucleus.persistenceByReachabilityA
tCommit=true
```

The downside is that the code is arguably less easy to debug.

8.3. Finding Objects

Retrieving domain entities typically requires a JDOQL query defined on the domain entity, and a corresponding repository service for that domain entity type. This repository calls the framework-provided `RepositoryService` to actually submit the query.

For example:

```
@javax.jdo.annotations.Queries({
    @javax.jdo.annotations.Query(                       ①
        name = "findByName",                            ②
        value = "SELECT "                               ③
                + "FROM com.mydomain.myapp.Customer "   ④
                + "WHERE name.indexOf(:name) >= 0 ")    ⑤
})
...
public class Customer { ... }
```

① There may be several `@Query` annotations, nested within a `@Queries` annotation) defines queries using JDOQL.

② Defines the name of the query.

③ The definition of the query, using JDOQL syntax.

④ The fully-qualified class name. Must correspond to the class on which the annotation is defined (the framework checks this automatically on bootstrapping).

⑤ In this particular query, is an implementation of a LIKE "name%" query.

and

```
@DomainService(nature=NatureOfService.DOMAIN)
public class CustomerRepository {

    public List<Customer> findByName(String name) {
        return repositoryService.allMatches(            ①
                new QueryDefault<>(Customer.class,      ②
                        "findByName",                   ③
                        "name",                         ④
                        name);
    }

    @javax.inject.Inject
    RepositoryService repositoryService;
}
```

① The `RepositoryService` is a generic facade over the JDO/DataNucleus API.

② Specifies the class that is annotated with @Query

③ Corresponds to the `@Query#name()` attribute

④ Corresponds to the `:name` parameter in the query JDOQL string

Whenever a query is submitted, the framework will automatically "flush" any pending changes. This ensures that the database query runs against an up-to-date table so that all matching instances (with respect to the current transaction) are correctly retrieved.

When an object is loaded from the database the framework will emit `ObjectLoadedEvent` lifecycle event.

8.3.1. Type-safe queries

DataNucleus also supports type-safe queries; see here for further details.

8.4. Updating Objects

There is no specific API to update a domain entity. Rather, the ORM (DataNucleus) automatically keeps track of the state of each object and will update the corresponding database rows when the transaction completes.

That said, it is possible to "flush" pending changes:

- `TransactionService` acts at the Apache Isis layer, and flushes any pending object persistence or object deletions
- using `IsisJdoSupport` it is possible to reach down to the underlying JDO API, and perform a flush of pending object updates also.

When an object is updated the framework will emit `ObjectUpdatingEvent` and `ObjectUpdatedEvent` lifecycle events.

8.5. Deleting Objects

Domain entities can be deleted using `RepositoryService`. For example:

```
Customer customer = ...
repositoryService.remove(customer);
```

It's worth being aware that (as for persisting new entities) the framework does *not* eagerly delete the object. Rather, it queues up an internal command structure representing the object deletion request. This is then executed either at the end of the transaction, or if a query is run, or if the internal queue is manually flushed using `TransactionService`'s `flush()` method.

Alternatively, you can use:

```
repositoryService.removeAndFlush(customer);
```

to eagerly perform the object deletion from the database.

When an object is deleted the framework will emit `ObjectRemovingEvent` lifecycle event.

Chapter 9. Business Rules

When a domain object is rendered in the UI or the end-user interacts with the domain object through the UI, the framework applies a series of precondition business rules to each object member (property, collection or action).

When the object is being rendered, the framework checks:

- is the object member visible?

  Members that are not visible are simply omitted from the page. If all the members in a fieldset (property group) are hidden, then the fieldset is not shown. If all the members in a tab are hidden, then the tab is not shown. If all the members of the object are hidden, then a "404" style message ("no such object") is returned to the user.

- if the object member is visible, is the object member enabled?

  An enabled property can be edited (otherwise it is read-only), and an enabled action can be invoked (otherwise it's button is "greyed-out"). Note that collections are always read-only.

- for enabled object members, if the user then interacts with that member, are the supplied values valid (can the user "do it").

  For an editable property this means validating the proposed new value of the property. For an invokable action this means validating that arguments being used to invoke the action.

One way to remember this is: **"see it, use it, do it"**

The framework provides a multitude of ways to implement these business rules. One decoupled approach is using domain events; the checks above correspond to the phases of the domain event. A simpler approach is to just implement the business rules imperatively in the domain object, or in a mixin for the object.

9.1. Visibility ("see it")

To hide an object action:

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

If the `hideXxx` method returns `true` then the action is hidden. Note that the supporting method takes no parameters.

A property or collection can be hidden similarly:

```
public String getStatus() { ... }
public void setStatus(String status) { ... }

public boolean hideStatus() { ... }
```

For more information, see the `hide···()` section in the appropriate reference guide.

9.2. Usability ("use it")

To disable an object action:

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

Here a non-`null` value is used as the reason why the action is disabled. It is also possible to return a localized string by returning a `TranslatableString`.

Properties are similar (note that collections are implicitly read-only in the Wicket viewer:

```
public String getStatus() { ... }
public void setStatus(String status) { ... }

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

For more information, see `disable···()` section in the appropriate reference guide.

It is also possible to make all properties of a domain object unmodifiable, using:

```
@DomainObject(
    editing=Editing.DISABLED
)
public class Customer { ... }
```

This can be made a global policy using a configuration setting:

isis.properties

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

9.3. Validity ("do it")

Action arguments can be validated either singly or as a set. For example:

```
public Customer placeOrder(Product p, int quantity) { ... }
public String validate0PlaceOrder(Product p) { ... }              ①
public String validate1PlaceOrder(int quantity) { ... }           ②
public String validatePlaceOrder(Product p, int quantity) { ... } ③
```

① validates the 0th argument of the action (0-based numbering), ie `Product`

② validates the 1st argument of the action, ie `int quantity`

③ validates all the arguments of the action together.

The framework validates each argument separately; only if all are valid does it check all the arguments together.

As for usability check, a non-`null` value is used as the reason why the action arguments are invalid. It is also possible to return a localized string by returning a `TranslatableString`.

Similarly, property edits can also be validated:

```
public String getStatus() { ... }
public void setStatus(String status) { ... }

public String validateStatus(String status) { ... }
```

For more information, see the `validate···()` section in the appropriate reference guide. The reference guide also explains how to define validation declaratively, using the `@Parameter#mustSatisfy()` or `@Property#mustSatisfy()` attributes.

9.4. Actions

Of course, the precondition business rules described above are only one type of business rule.

More generally, business rules are implemented in the form of the implementation of actions. Rather than have the end-user have to edit individual properties of numerous objects, an action can encode these rules and allow only safe transformations of the application from one consistent state to the next.

9.5. Side effects

Often when the state of an object is modified there is a business rule for some sort of side-effect. For example, suppose that a `Person` has a persisted derived property:

```
public class Person {

    @Getter @Setter                         ①
    private LocalDate dateOfBirth;

    @Getter @Setter
    private int age;                        ②
```

① using Lombok to remove boilerplate.

② persisted derived property based on `dateOfBirth`.

One design would be to only allow the two properties to be modified through an action:

```
public Person updateDateOfBirth(LocalDate dateOfBirth) {
    setDateOfBirth(dateOfBirth);
    setAge(Years.between(clockService.now(), dateOfBirth));
    return this;
}
```

Alternatively we could allow the property to be edited, and then compute the side-effect.

```
public void setDateOfBirth(LocalDate dateOfBirth) {
    this.dateOfBirth = dateOfBirth;
    setAge(Years.between(clockService.now(), dateOfBirth));
    return this;
}
```

Datanucleus does *not* call the setter when reloading the object from the database, so there is no issue in having the side-effect code (calling `setAge(…)`).

The framework also allows side-effect code to be placed in separate `modify…()`, `clear…()` supporting methods; if present then these will be called by the framework rather than the setter. However, because of DataNucleus' smart handling of setters, these supporting methods are in essence redundant, and so should be considered deprecated.

Chapter 10. Drop Downs and Defaults

Invoking an action whose parameters are primitives or values (int, date, string etc) is simple: the user can just type in or use a date picker. Invoking an action with a parameter of reference type (such as `Customer` or `Order`) requires the viewer to provide some mechanism by which the end-user can select the relevant instance.

If the list of available options is fixed then the developer can provided a list a `choices…()` supporting method (for either and action parameter or when editing a property). These are rendered in a drop-down.

If the list of available options is much larger, then the developer can use an `autoComplete…()` supporting method. The user user enters a few characters and this is used to search for matching reference(s), again rendered in a drop-down.

Similarly, when invoking an action, there may well be suitable defaults for the action arguments. For example, if placing an `Order` then — even if the `Product` argument might not have a sensible default — the quantity argument could reasonably be defaulted to 1. Or, the `Product` might indeed have a default, say the product previously placed by this user. The developer indicates this using a `default…()` supporting method.

10.1. Choices and Default

For example, *choices* for a property are specified using:

```
@Getter @Setter
private String paymentMethod;

public List<String> choicesPaymentMethod() {
    return Arrays.asList("Visa", "Mastercard", "Amex");
}
```

Note the "choices" prefix, while the suffix matches up with the getter.

Or, for an action the *choices* and a suitable *default* method would be:

```
public Order changePaymentMethod(
                @ParameterLayout(named="Payment Method")
                String paymentMethod) {
    setPaymentMethod(paymentMethod);
    return this;
}

public List<String> choices0ChangePaymentMethod() {
    return Arrays.asList("Visa", "Mastercard", "Amex");
}

public String default0ChangePaymentMethod() {
    return getPaymentMethod();
}
```

Note the "choices" and "default" prefix, the digit is the 0-based argument number, while the suffix matches up with the action's name.

10.2. AutoComplete

The *autocomplete* is similar to *choices*, but accepts a string parameter, to search for matching results. A property for example might have:

```
@Getter @Setter
private Product product;

public List<Product> autoCompleteProduct(@MinLength(2) String search) {
    return productRepository.findByReferenceOrName(search);
}
```

Note the "autoComplete" prefix, while (again) the suffix matches up with the getter. The `@MinLength(…)` annotation indicates the minimum number of characters that must be entered before a search is initiated.

Actions are very similar:

```
public Order changeProduct(Product product) {
    setProduct(product);
    return this;
}

public List<Product> autoComplete0Product(@MinLength(2) String search) {
    return productRepository.findByReferenceOrName(search);
}
```

An *autoComplete* method can be used in conjunction with a *default* method, but it doesn't make

sense to provide both an *autoComplete* and a *choices* method.

10.3. "Globally" defined drop-downs

Very often the set of available choices depends on the data type of the property/action parameter, rather than the individual property/parameter itself. And similarly the algorithm to search for references again may well depend only on that reference type.

In the case of *choices*, annotating a class as "bounded" (as in a "bounded" or fixed number of instances) means that a *choices* drop-down will automatically be defined. For example:

```
@DomainObject(
    bounded = true
)
public class Product { ... }
```

For more on this, see `@DomainObject#bounded()`.

Or, if the data type is an enum, then a drop-down will be provided automatically. A payment method is a good example of this:

```
public enum PaymentMethod {
    VISA, MASTERCARD, AMEX;
}
```

Something similar can be achieved for *autoComplete*. Here the domain object indicates a repository query to execute. For example:

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

with:

```
@DomainService(nature=NatureOfService.VIEW_MENU_ONLY)
public class Customers {
    @Action(semantics=SemanticsOf.SAFE)
    public List<Customer> findByReferenceOrName(@MinLength(3) String refOrName) {
        ...
    }
}
```

For more on this, see `@DomainObject#autoCompleteRepository()`.

There's no need for the nominated method to be an actual action; any method of any domain service will do, so long as it accepts a string and returns the correct list.

10.4. Multi-select action parameters

As well as scalar values, action parameters can also be collections. For this to be valid, a *choices* or *autoComplete* supporting method must be provided.

For example, suppose we want to "tag" or "label" an object:

```
public StoryCard tag(List<Tag> tags) {
    getTags().addAll(tags);
}

public List<Tag> autoCompleteTag(@MinLength(1) search) {
    return tagRepository.findByName(search);
}
```

10.5. Dependent choices for action parameters

For action it is also possible (in a limited form) to define dependencies between parameters. Specifically, if one parameter is a drop-down choice, then other drop-down choices can be derived from it.

A good example is a category/sub-category:

```
public ToDoItem categorize(Category category, Subcategory subcategory) {
    setCategory(category);
    setSubcategory(subcategory);
}

public List<Category> choices0Categorize() {
    return categoryRepository.allCategories();
}
public List<Subcategory> choices1Categorize(Category category) {
    return subcategoryRepository.findBy(category);
}
```

Note how the *choices* method for the 2nd parameter also accepts the first parameter.

Chapter 11. Available Domain Services

There are a number of domain services/modules already implemented for you to use directly within your applications. Some of these are provided by the framework, while others are third-party (but still open source).

This chapter surveys some of the domain services currently available.

11.1. 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 behaviour. 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)
- `InteractionContext` (SPI)
- `AuditingService` (SPI) (deprecated)
- `AuditerService` (SPI)
- `BackgroundService`
- `BackgroundCommandService` (SPI)
- `PublishingService` (SPI) (deprecated)

- PublishererService (SPI)
- 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:

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

REST Support:

- AcceptHeaderService
- SwaggerService
- ContentMappingService (SPI)

Metamodel:

- ApplicationFeatureRepository
- MetamodelService

Other API:

- `FixtureScripts`
- `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.

11.2. Incode Platform

The (non-ASF) Incode Platform provides a number of reusable modules for Apache Isis, focusing either on specific technologies, technical cross-cutting concerns, or providing business logic for generic subdomains. Some of these modules implement SPIs defined by the framework.

Note that the Incode Platform, although maintained by Apache Isis committers, are not part of the ASF.

The modules themselves fall into a number of broader groups:

- technical libraries, such as excel, word and freemarker
- SPI implementations, such as security, commands, auditing and publishing
- framework extensions, such as flyway, quartz and feature toggles
- wicket components, such as maps, calendars and PDF viewer
- business modules, such as documents, notes and communications.

Each module can be used independently or combined, and the Incode Platform also provides a quickstart application to get you started quickly.