# Classification Subdomain

# Table of Contents

This module (`incode-module-classification`) provides the ability to classify any (arbitrary) domain entity as belonging to a pre-defined `Category` within a particular `Taxonomy`. A given domain object can only be associated with one `Category` per `Taxonomy`, but each `Category` can optionally have child (sub-) categories.

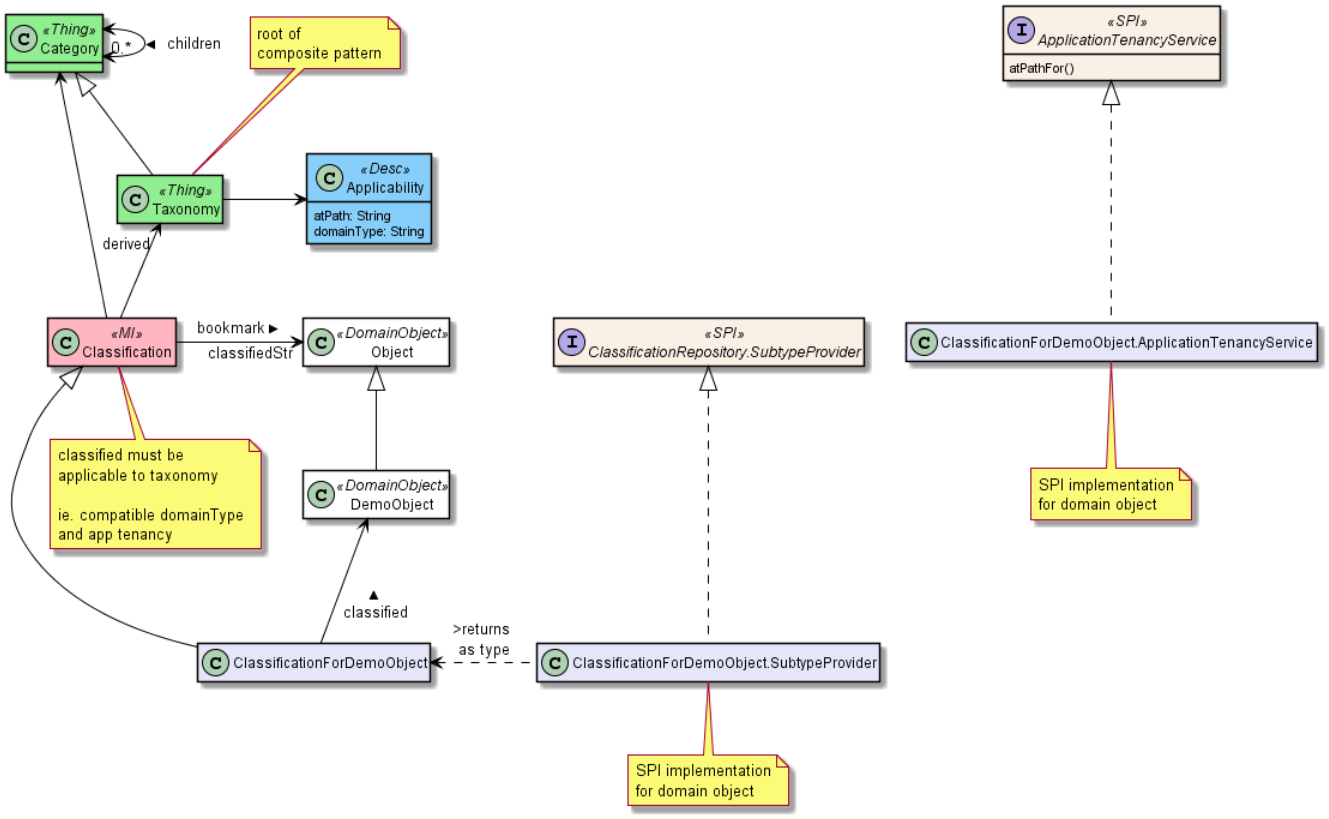
Not every `Taxonomy` is applicable to every domain object. Instead, the `Taxonomy` is qualified by both the domain object's type and also its application tenancy (eg country within a multi-tenancy environment).

There are *no* requirements for those domain objects implement any interfaces. A subclass of the `Classification` abstract class is required; this acts as the "glue" between the `Category` and the "classified" domain object. In total about 50 lines of boilerplate are required, details below.

This module expects that application tenancy path is hierarchical (so for example "/ITA/MIL" resides within both "/ITA" and "/"). This allows `Taxonomy`s to be declared as applicable at one level (eg "/ITA") and to be considered applicable for domain objects at that level and sub-levels (eg "/ITA/MIL"). The value of the application tenancy path of a domain object is provided through the `ApplicationTenancyService` SPI.

# Domain Model

The following class diagram highlights the main concepts:




(The colours used in the diagram are - approximately - from Object Modeling in Color).

## `Taxonomy` and `Category`

The `Taxonomy` entity defines a hierarchy of `Category`s, with the `Taxonomy` itself acting as the root of each such hierarchy. In many cases the `Taxonomy` will be only two levels deep, in effect defining a enum, eg:

- "Italian Colours" root classification, with children:
  - "Italian Colours/Red"
  - "Italian Colours/White"
  - "Italian Colours/Green"

However, deeper `Taxonomy`s are possible, eg:

- "Sizes `
  - "Sizes/Small"
    - "Sizes/Small/Smallest"
    - "Sizes/Small/Smaller"
    - "Sizes/Small/Small"

- "Sizes/Medium"
- "Sizes/Large"
  - "Sizes/Large/Large"
  - "Sizes/Large/Larger"
  - "Sizes/Large/Largest"

There can be many such taxonomies; any given domain object can only have one `Classification` per `Taxonomy` hierarchy (but not more than one classification per hierarchy). Thus, a domain object might be classified as both "Italian Colours/Red" and "Sizes/Medium", but it isn't possible to classify as both "Italian Colours/Red" and "Italian Colours/Green".

## `Applicability` and `ApplicationTenancyService`

Not every domain object can be classified with respect to every `Taxonomy`. Instead, the available set is restricted by the `Applicability` entity. This identifies which `Taxonomy`(s) can be associated with domain object types. This is further qualified by the application tenancy of the domain object (for multi-tenanted applications).

The `ApplicationTenancyService` SPI service is used to obtain the application tenancy of each domain object.

## `Classification`

The `Classification` is the tuple that associates a particular domain object with a particular `Category` in some `Taxonomy`. This must be with respect to some `Applicability`. `Classification` itself is an `abstract` class; for each domain object to be classified, a subclass of `Classification` is required, providing a type-safe (referential integrity) connection between the two entities.

The module does *not* prevent an `Applicability` from being removed, even if there are existing `Classification`s that rely upon that `Applicability`.

# Screenshots

The module's functionality can be explored by running the quickstart with example usage using the `org.incode.domainapp.example.app.modules.ExampleDomDomClassificationAppManifest`.

This sets up a small hierarchy of app tenancies, namely "/" (global), "/ITA" (Italy)", "/FRA" (France) and two sub-tenancies of Italy and France, "/ITA/MIL" (Milan) and "/FRA/PAR" (Paris).

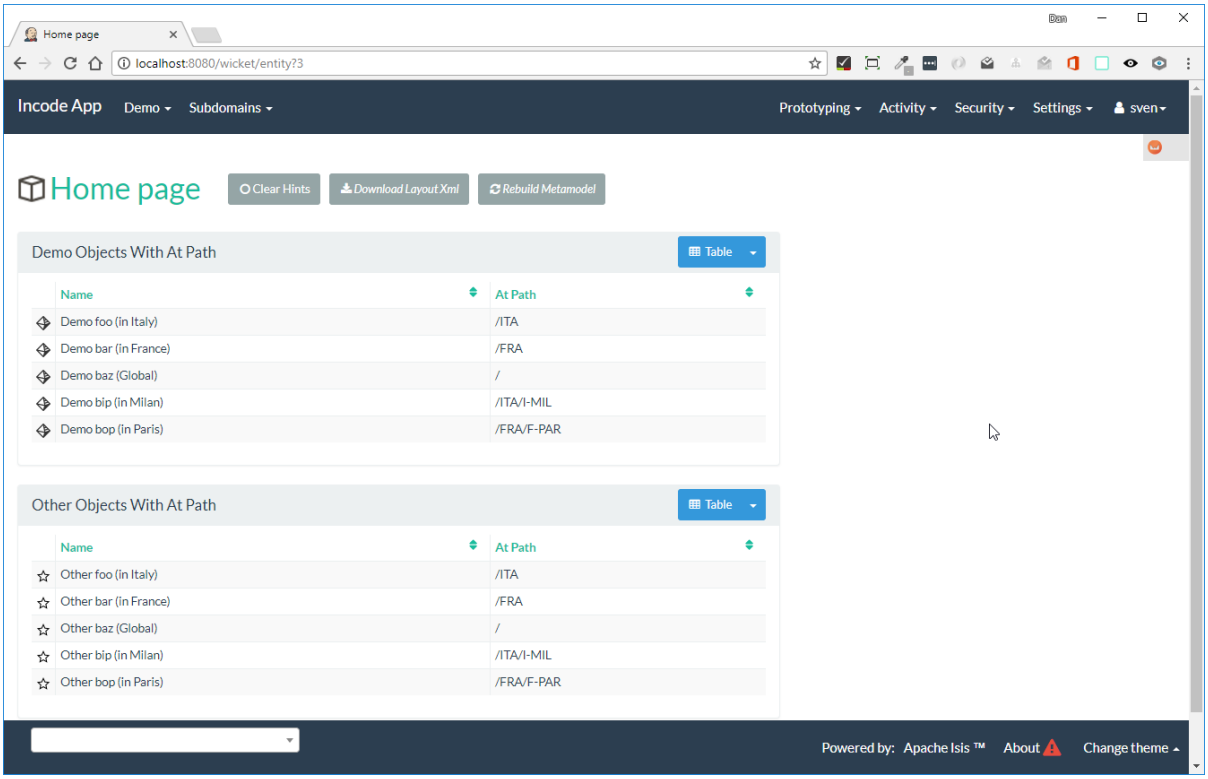
There are two separate domain object types, `DemoObject` and `OtherObject`. There are five instances of each, in the various app tenancies.

There are also three example taxonomies: "Sizes", "Italian Colours" and also "French Colours". These are set up so that "Sizes" is applicable globally, while the two different "colour" taxonomies apply only to their respective app tenancies.

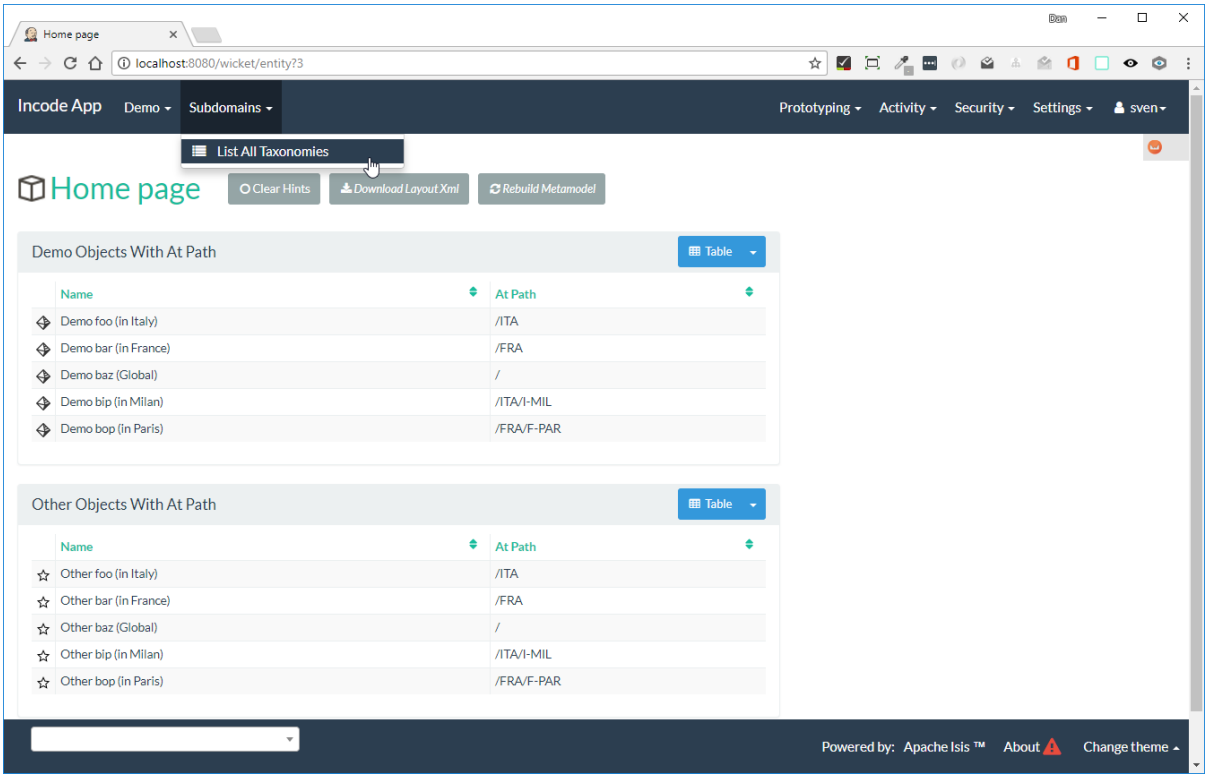
To demonstrate that domain type is significant, the "Sizes" and "French Colours" taxonomies apply to `DemoObject` but do *not* apply to the `OtherObject`. The "Italian Colour" taxonomy on the other hand applies to both `DemoObject` and to `OtherObject`.

# Taxonomies (reference data)

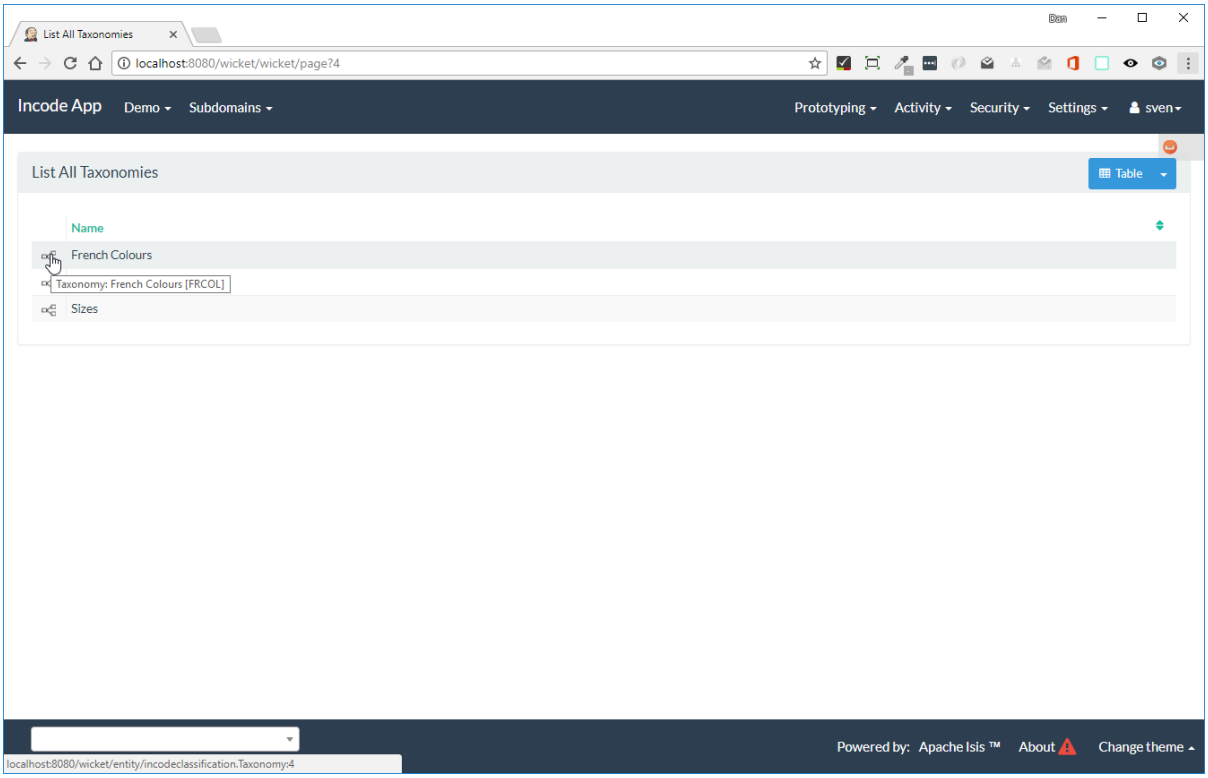
A home page is displayed when the app is run:



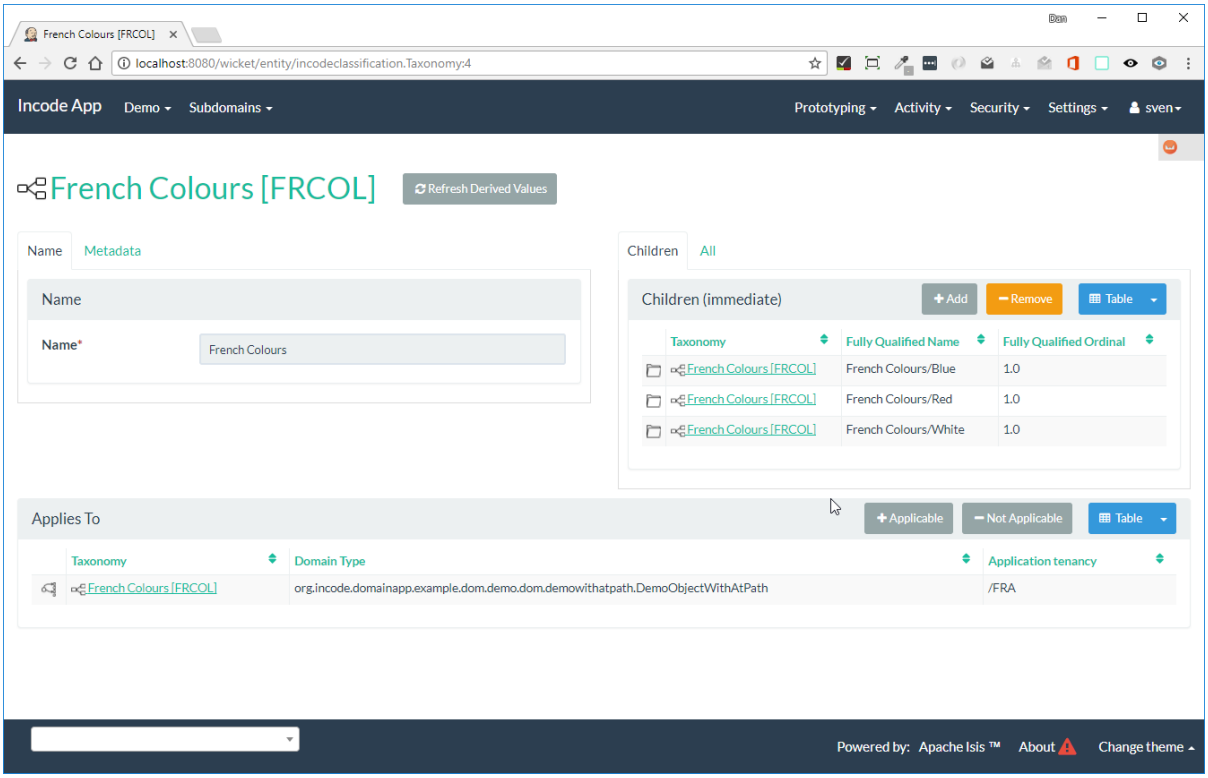

We can then list the taxonomies:



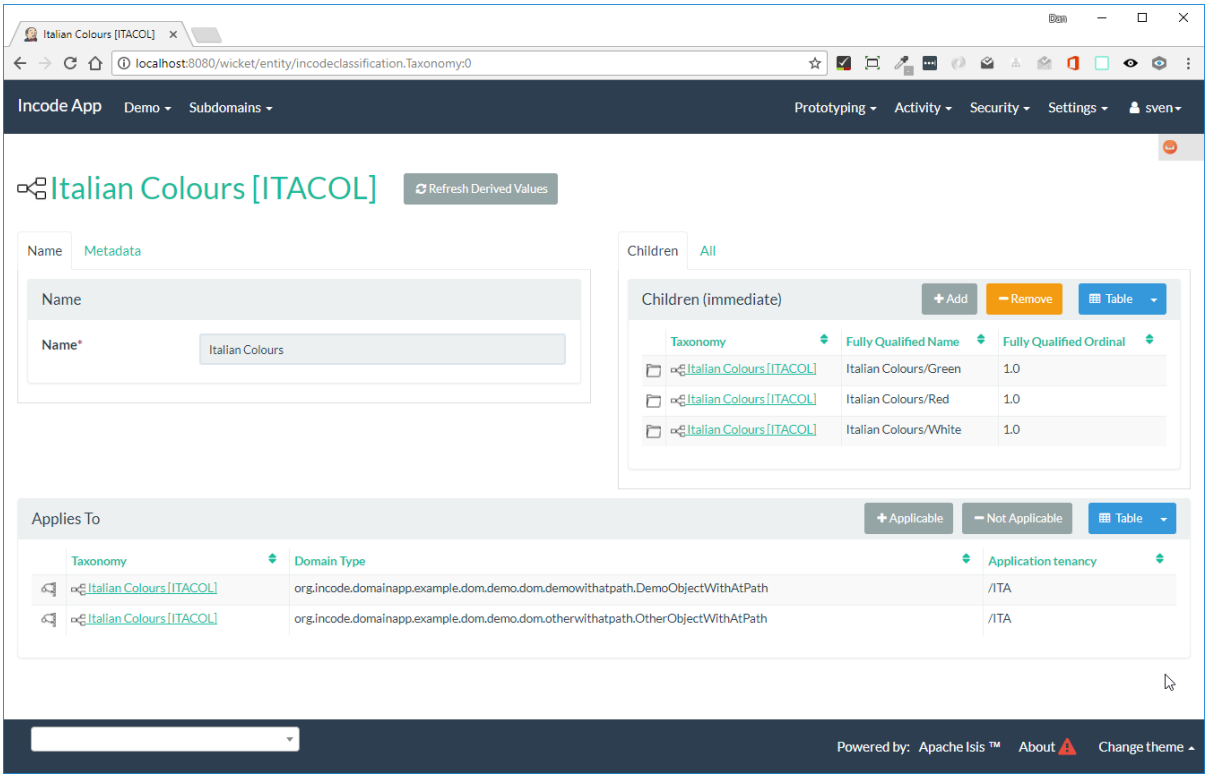

which returns the three demo taxonomies, "Size", "Italian Colours" and "French Colours":




The "French Colours" `Taxonomy` contains three `Category`s, namely "Red", "White" and "Blue":



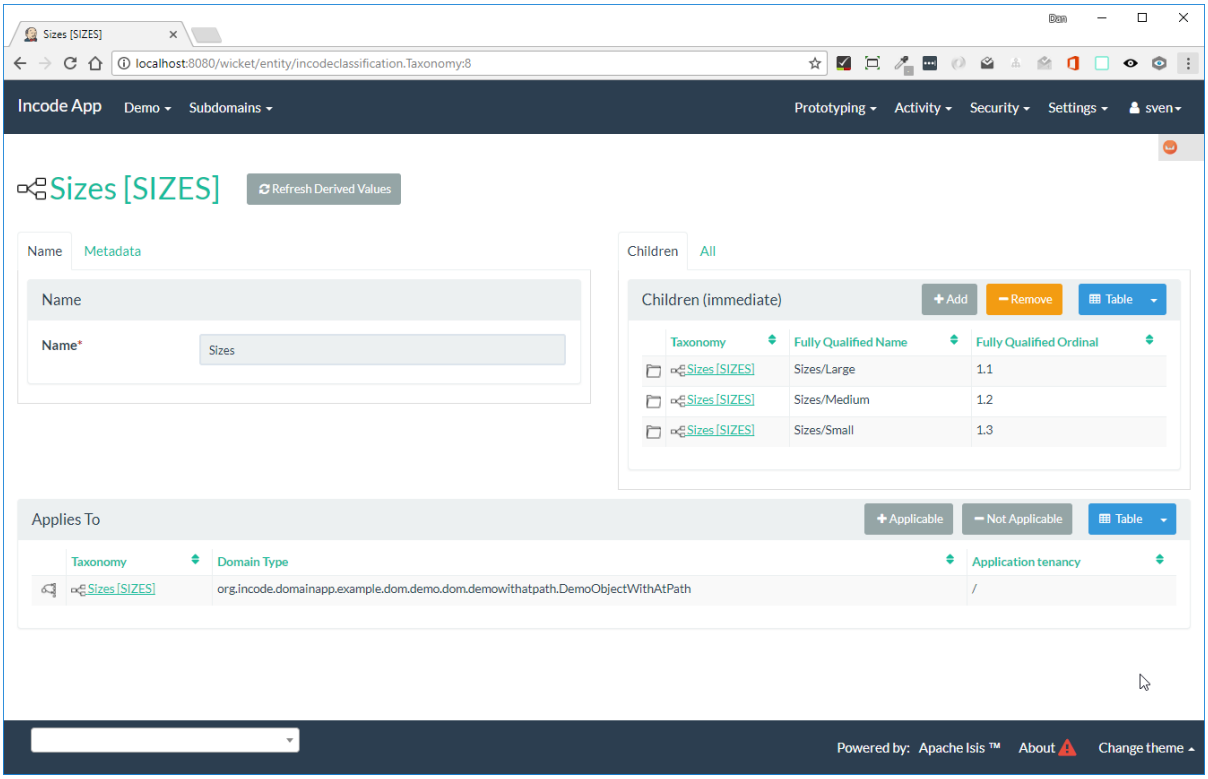

while the "Italian Colours" `Taxonomy` contains three different `Category`s, "Red", "White" and "Green":




Note that the "French Colours"' "Red" is different from the "Italian Colours"' "Red", also for "White". These are two different `Category`s in two different `Taxonomy`s that just happen to have the same (local) name.

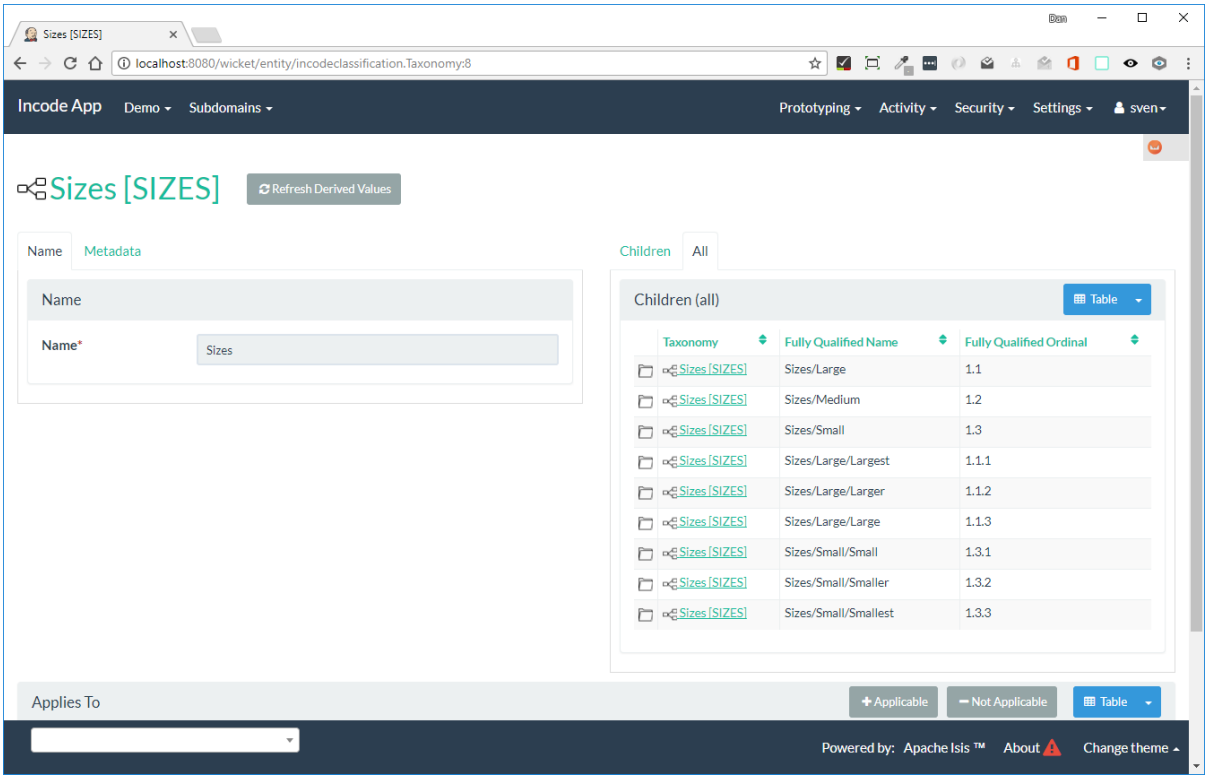
Also note (as can be guessed from their names) that the "French Colours" `Taxonomy` only applies to the "/FRA" app tenancy, while the "Italian Colours" `Taxonomy` applies only to the "/ITA" app tenancy. The former also only to the `DemoObject` domain type, while the latter apples to both `DemoObject` and also `OtherObject` domain types.

The final `Taxonomy` is "Size":




In contrast to the two "colour" taxonomies, the "Size" taxonomy is defined globally (for the "/" app tenancy). However, it only applies to the `DemoObject` domain type, not to the `OtherObject` domain type.

The "Size" taxonomy is also more complex than the other two taxonomies, in that contains categories and sub-categories:



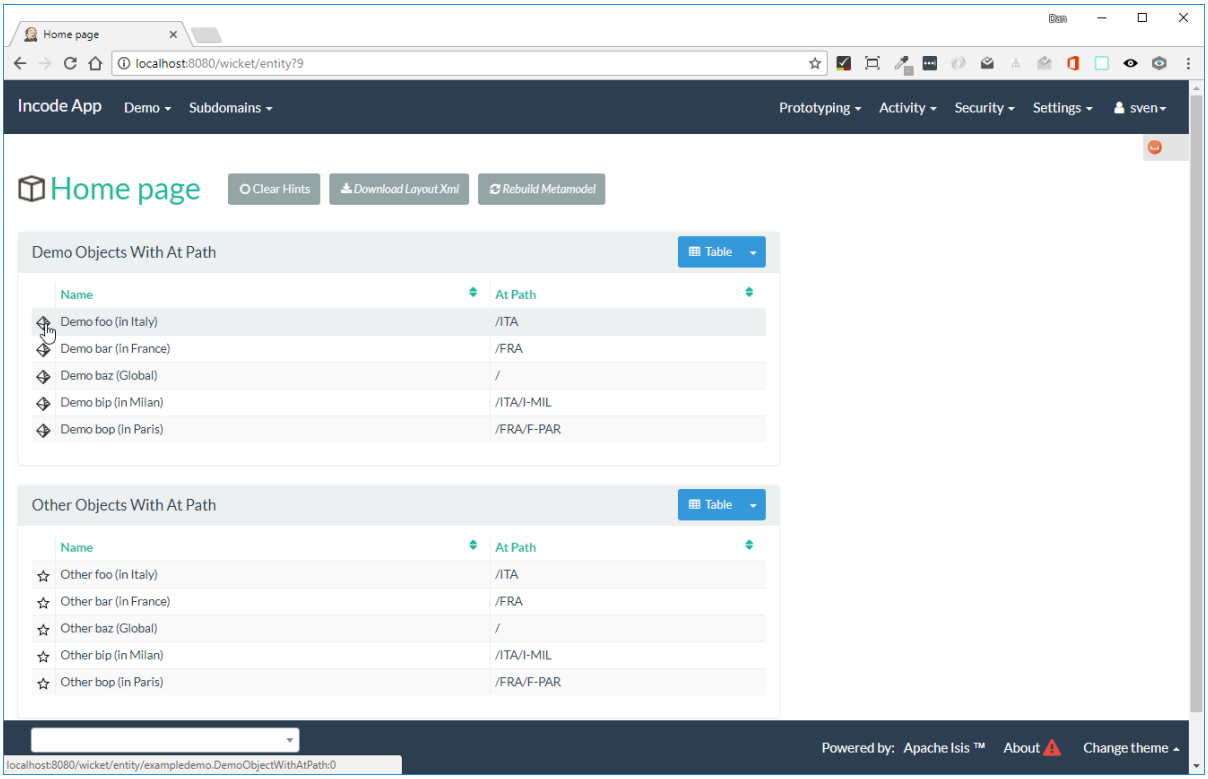

The table below summarizes the various taxonomies and their applicability:

*Table 1. Taxonomy applicability*

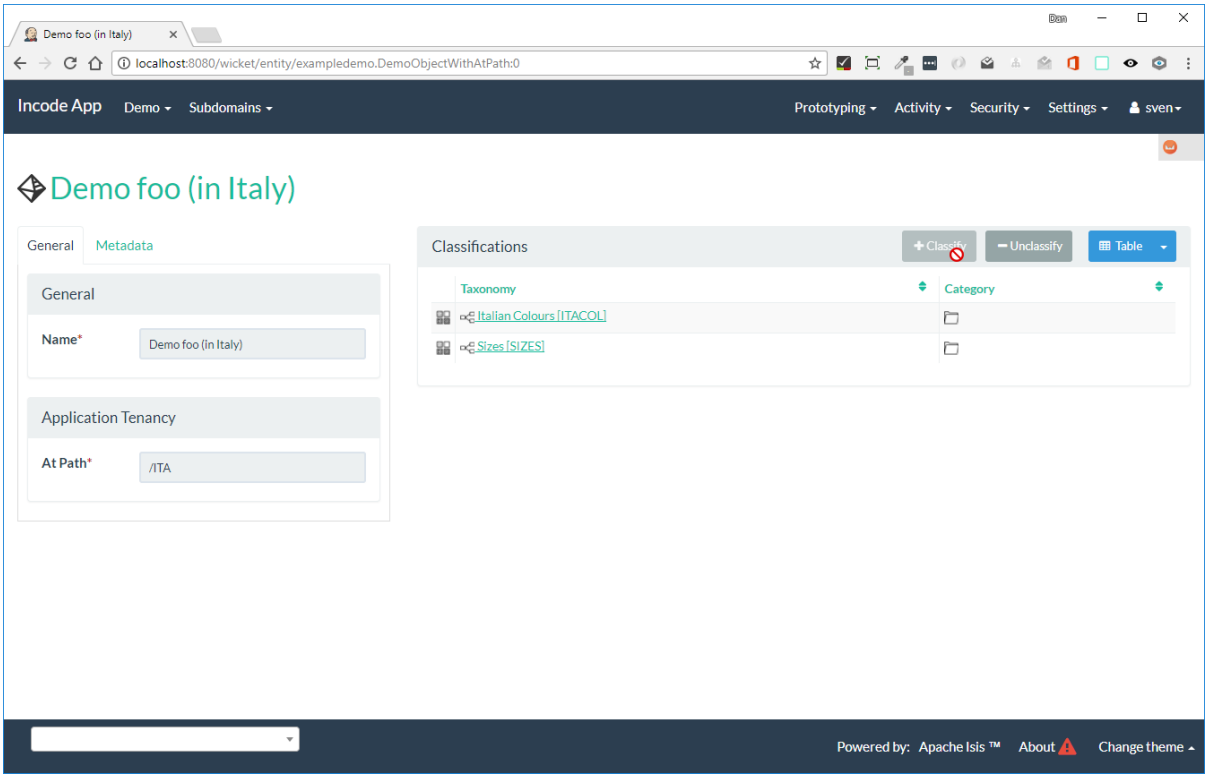
| Domain type | App tenancy | "Italian Colours" taxonomy | "French Colours" taxonomy | "Size" taxonomy |
|---|---|---|---|---|
| `DemoObject` | `/` | No | No | Yes |
| | `/ITA` | Yes | No | Yes |
| | `/FRA` | No | Yes | Yes |
| | `/ITA/MIL` | Yes | No | Yes |
| | `/FRA/PAR` | No | Yes | Yes |
| `OtherObject` | `/` | No | No | No |
| | `/ITA` | Yes | No | No |
| | `/FRA` | No | No | No |
| | `/ITA/MIL` | Yes | No | No |
| | `/FRA/PAR` | No | No | No |

# Domain Object Data

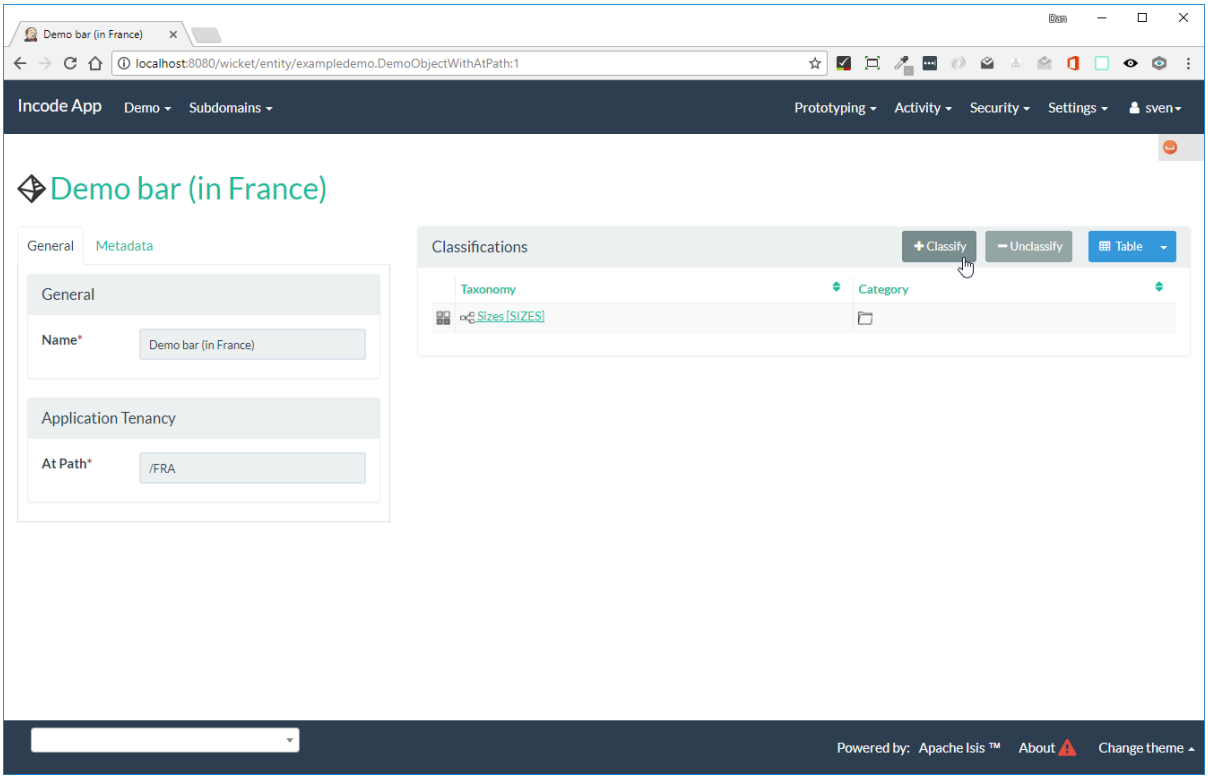
The example app creates 5 instances of `DemoObject`, each in a different app tenancy:



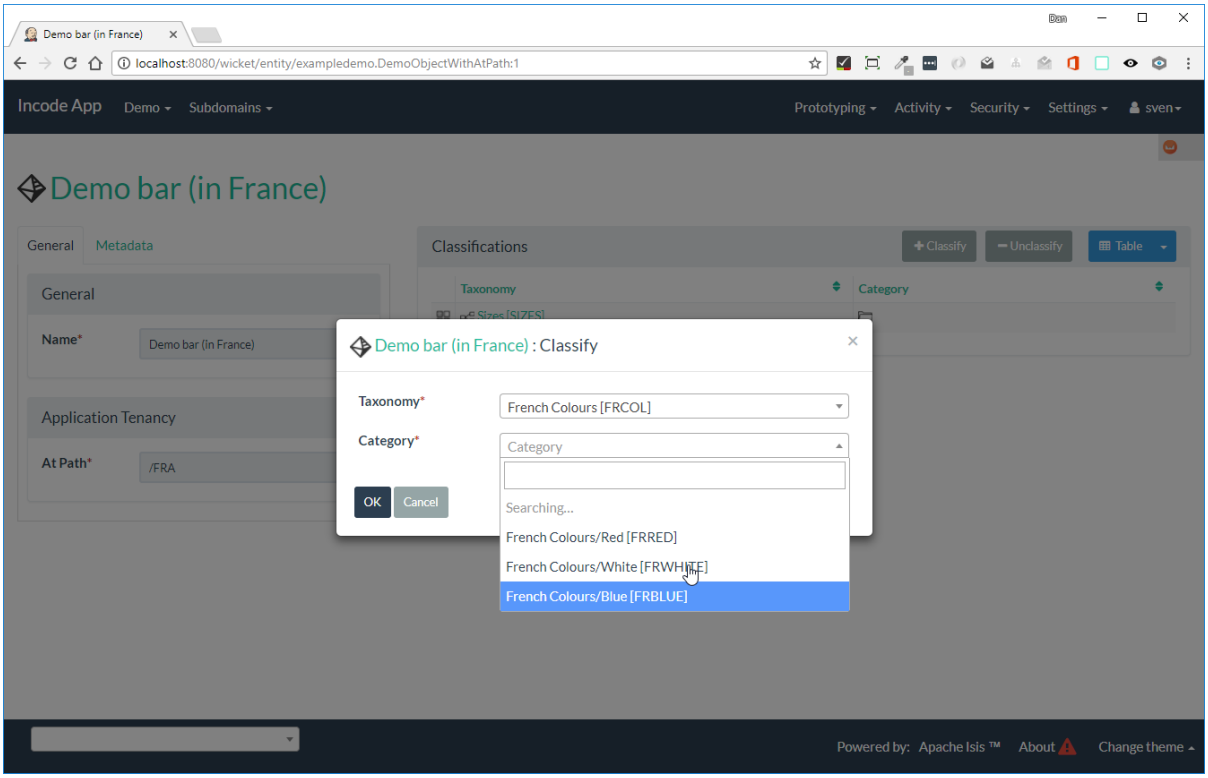

The "foo" `DemoObject` is in the "/ITA" app tenancy, which means that the "Italian Colours" and "Sizes" taxonomies both apply. The example seed data adds `Classification`s for this object in each of these taxonomies. As the screenshot shows, no further `Classification`s can be added:



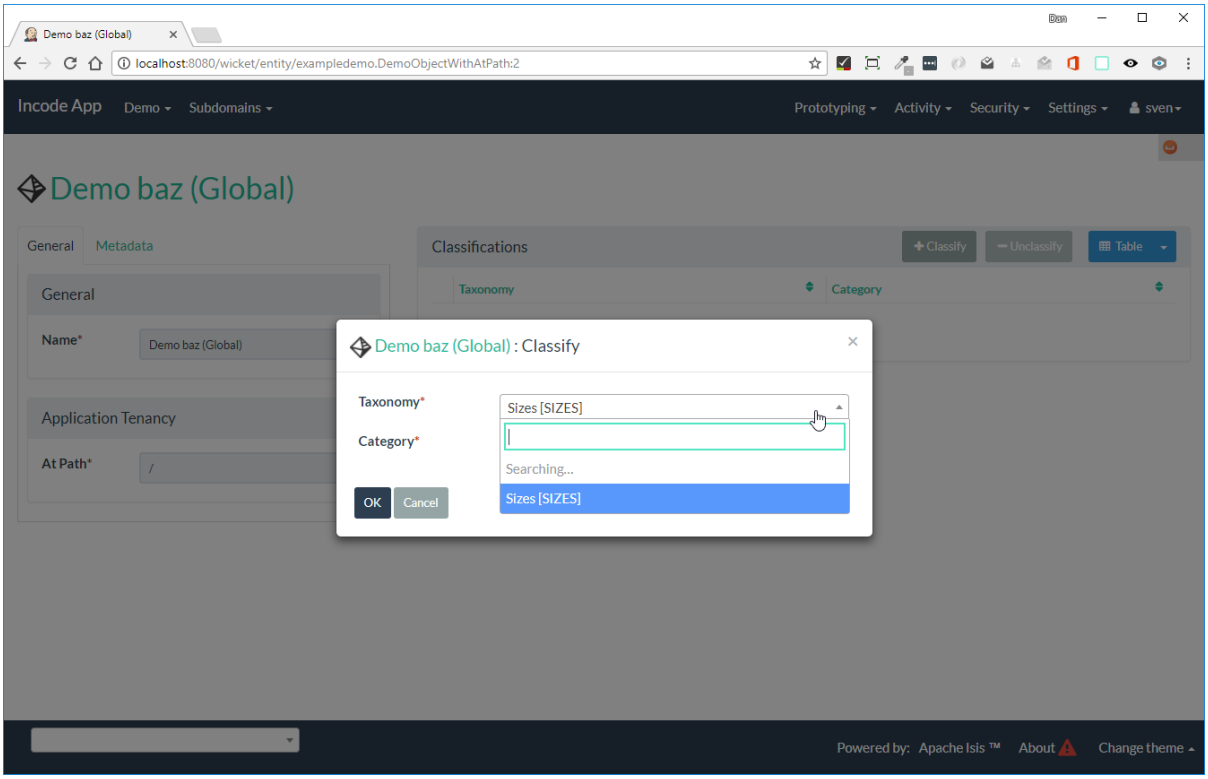

The "bar" `DemoObject` is in the "/FRA" app tenancy, which means that the "French Colours" and "Sizes" taxonomies both apply. The example seed data adds a `Classification` for the "Sizes" taxonomy, which means that the object can still be classified (in the "French Colours" taxonomy):



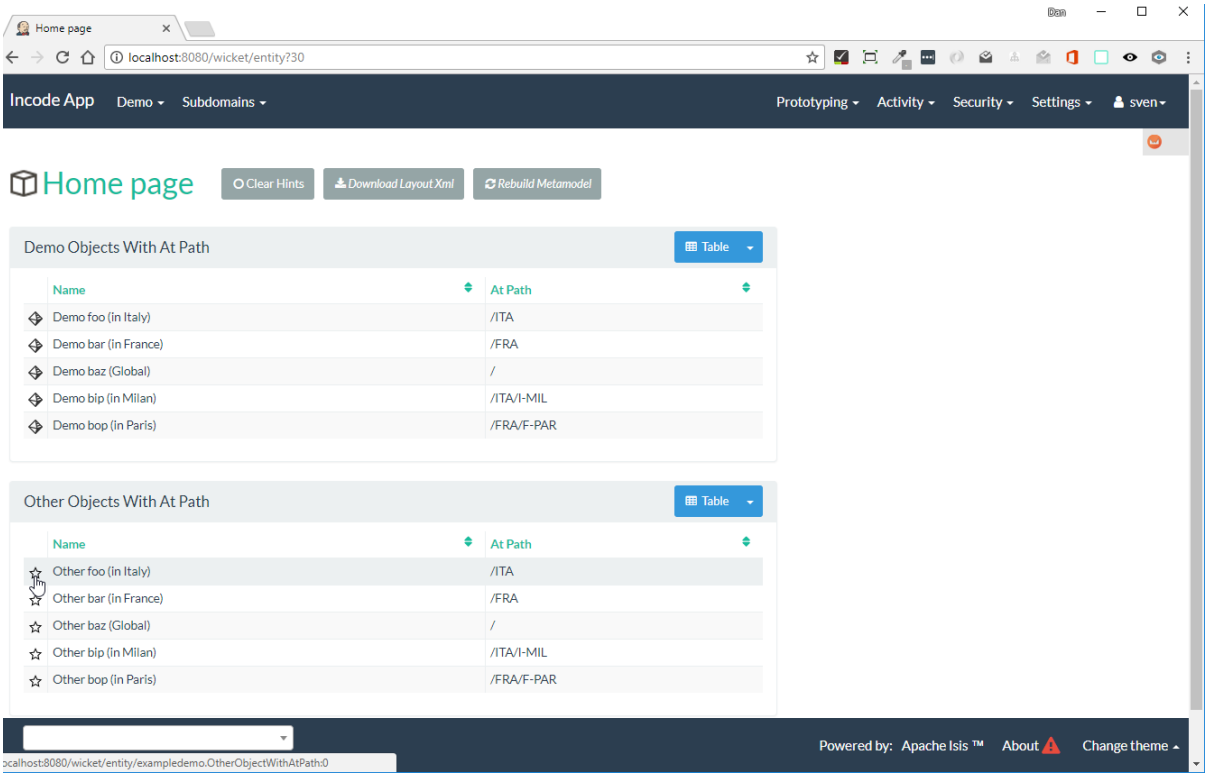

Since there is only one applicable taxonomy ("French Colours"), this is automatically defaulted. The end-user can then select the particular `Category` within that `Taxonomy`:



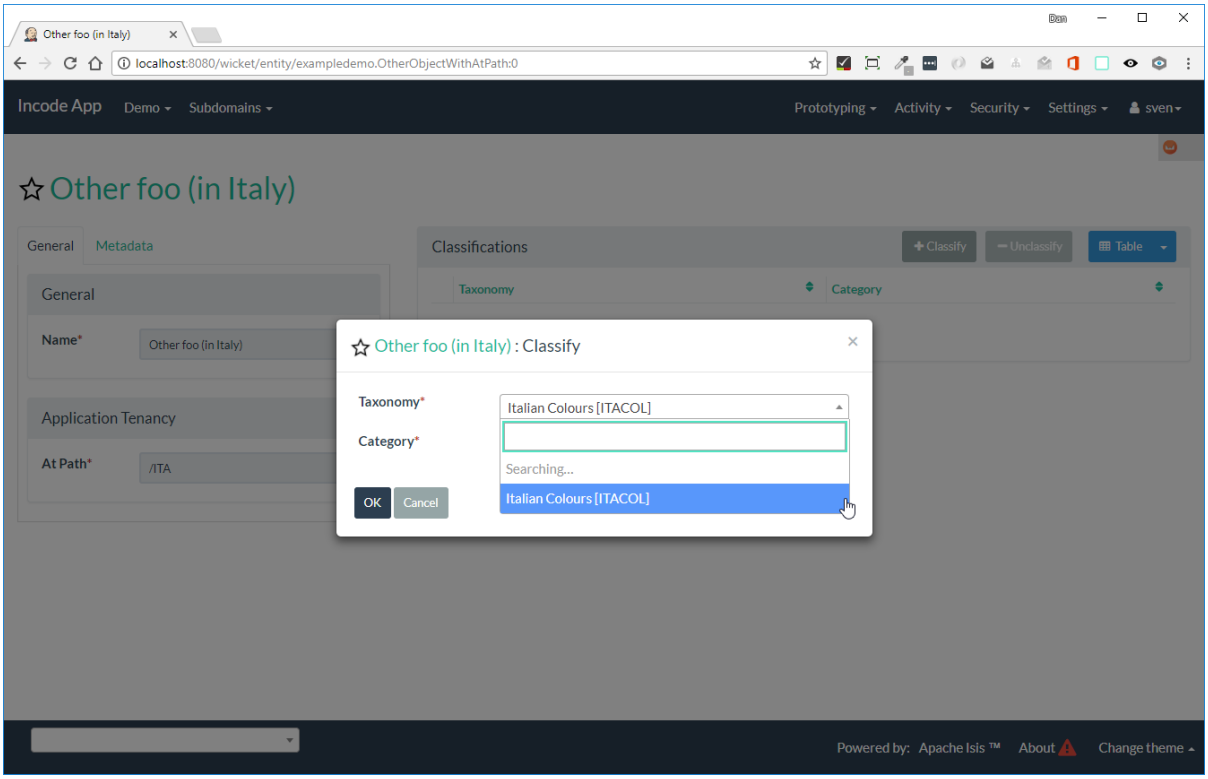

The "baz" `DemoObject` on the other hand starts off with no `Classification`s. Because this has global app tenancy, only the "Sizes" `Taxonomy` applies:



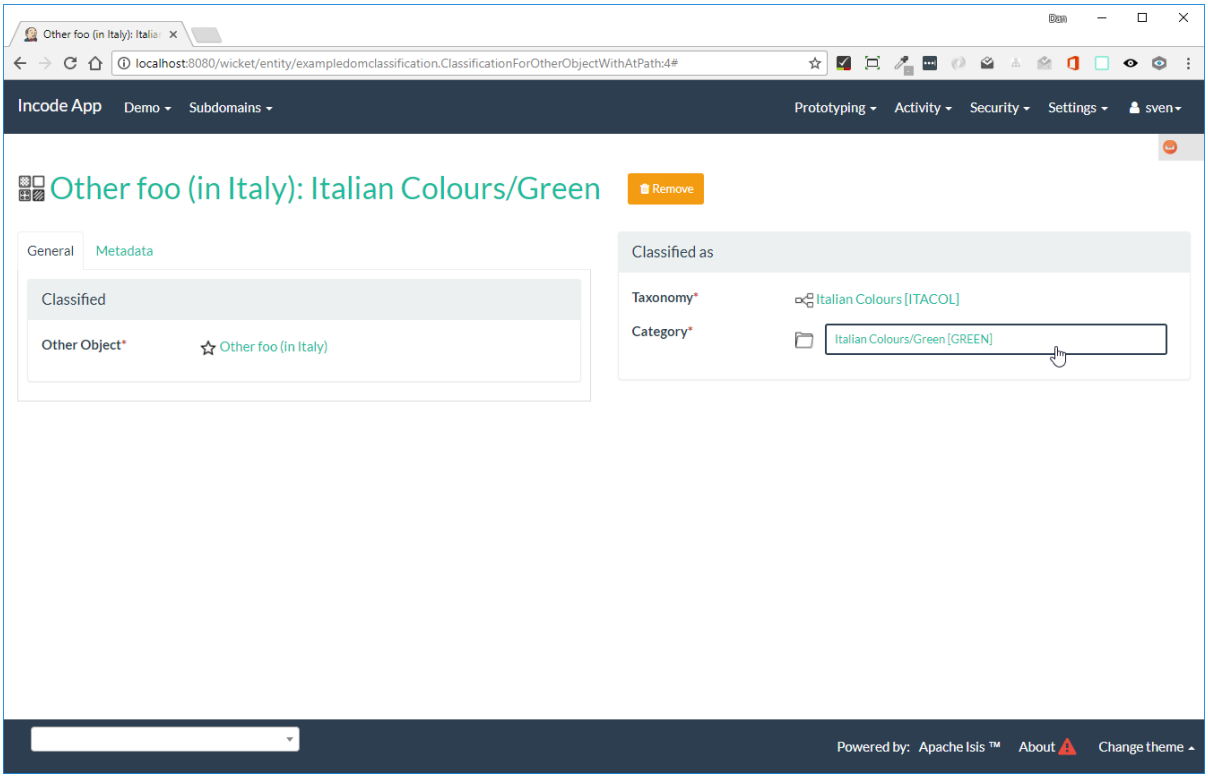

Like `DemoObject`, there are five instances of `OtherObject`, again each with a different app tenancy:



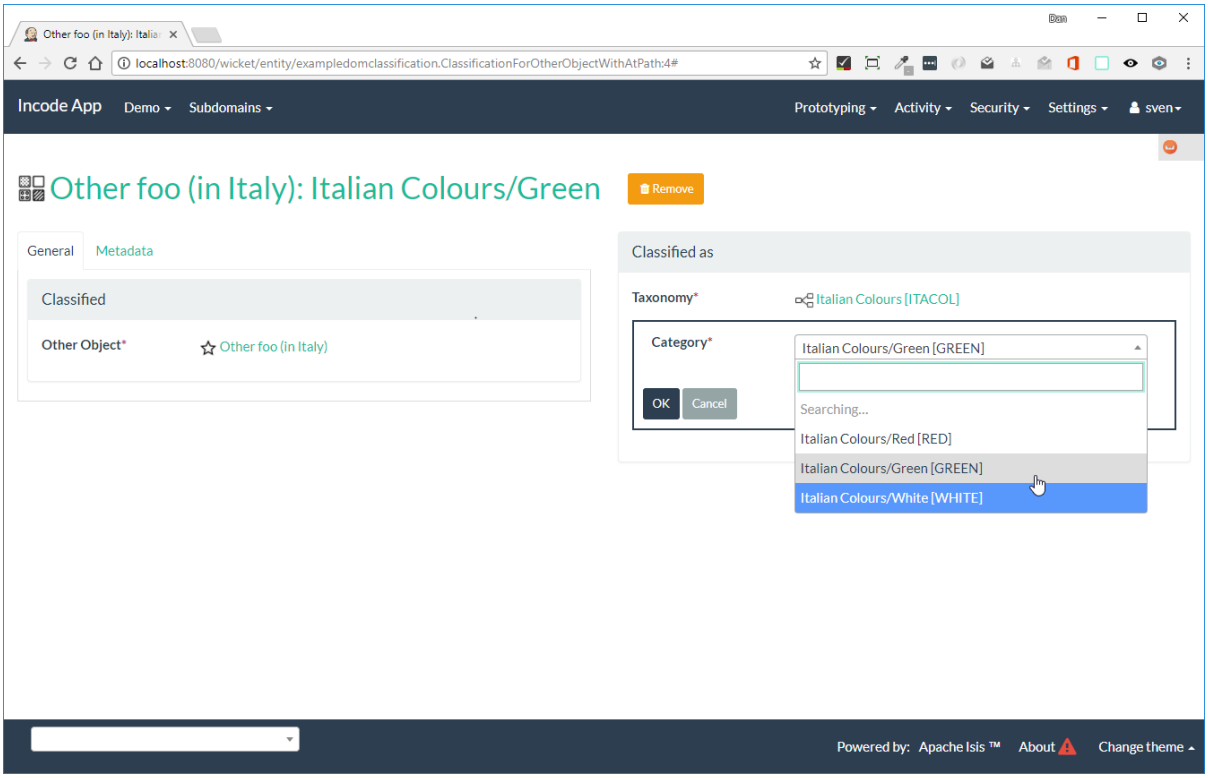

The difference between `OtherObject` and `DemoObject` is that neither the "Sizes" nor "French Colours" taxonomies are applicable to `OtherObject`. Thus, with the "foo" `OtherObject` the only available taxonomy to classify is "Italian Colours":



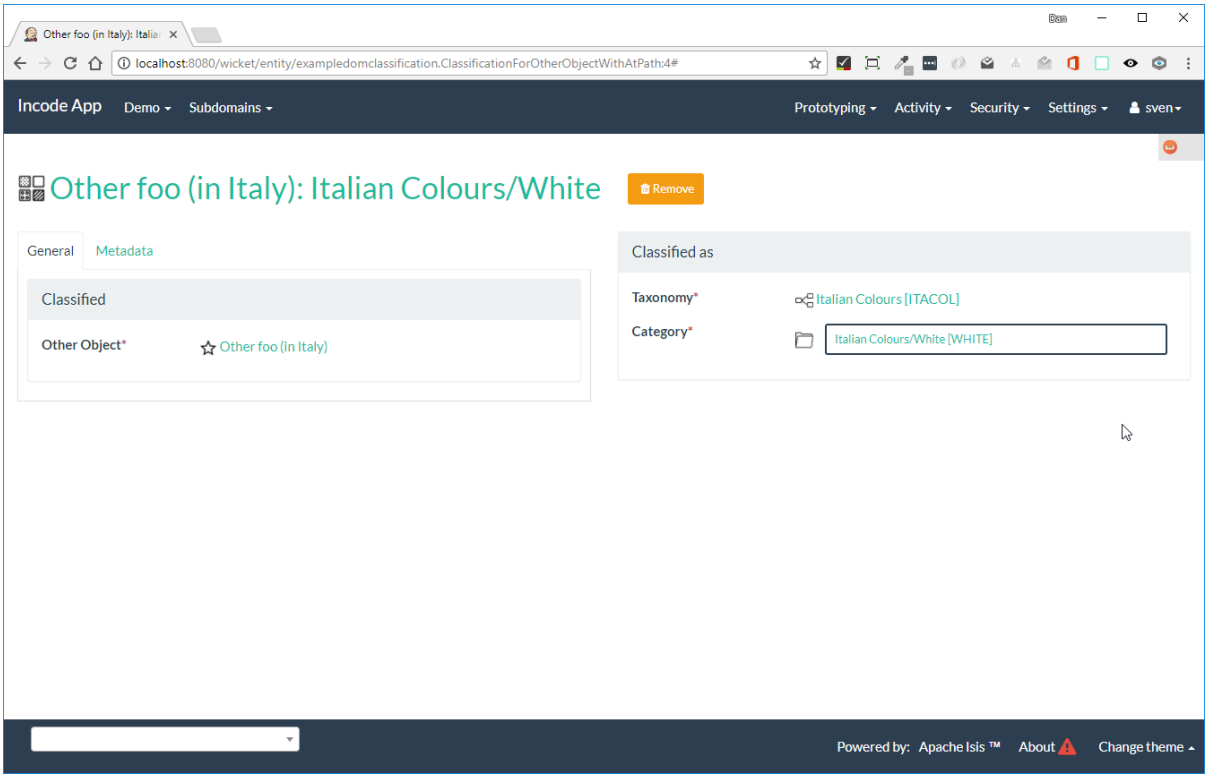

Once a `Classification` has been made, it can be altered to any other `Category` within the same `Taxonomy`:



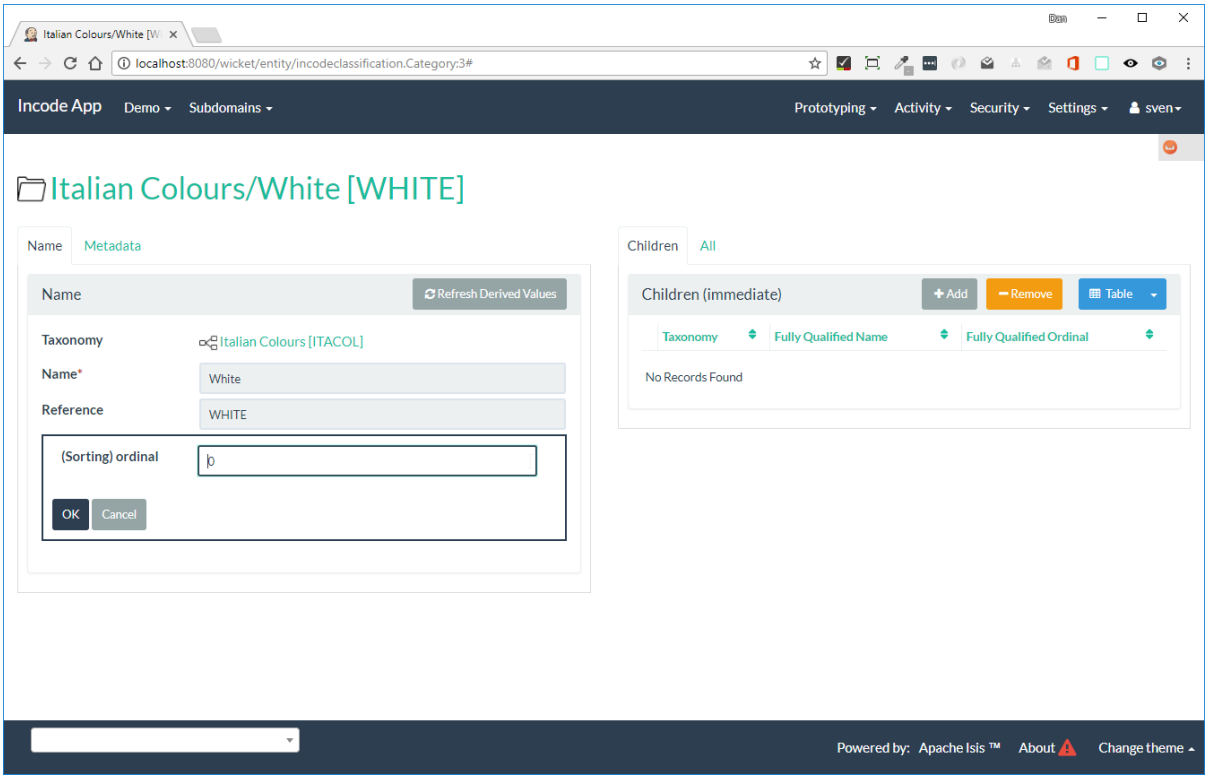

Here the `Classification` is being changed:




Which we can see *has* then been changed:




It is also possible to change each `Category`'s name, reference and (sorting) ordinal. If the name or ordinal are changed then the fully qualified name/ordinal are automatically updated for both the `Category` and any of its children.




(As of 1.15.0), the `name` and `reference` properties can only be modified if the global `isis.objects.editing` is set to `true`. The `sortingOrdinal`, however, is always editable.

# How to configure/use

## Classpath

Update your classpath by adding this dependency in your dom project's `pom.xml`:

```
<dependency>
    <groupId>org.incode.module.classification</groupId>
    <artifactId>incode-module-classification-dom</artifactId>
    <version>1.15.0</version>
</dependency>
```

Check for later releases by searching Maven Central Repo.

For instructions on how to use the latest `-SNAPSHOT`, see the contributors guide.

## Bootstrapping

In the `AppManifest`, update its `getModules()` method, eg:

```
@Override
public List<Class<?>> getModules() {
    return Arrays.asList(
            ...
            org.incode.module.classification.dom.ClassificationModule.class,
    );
}
```

## For each domain object...

For each domain object that you want to classify (that is, add `Classification`s to), you need to:

- implement a subclass of `Classification` for the domain object's type.

  This link acts as a type-safe tuple linking the domain object to the `Category`.

- implement the `ApplicationTenancyService` SPI interface:

  ```
  public interface ApplicationTenancyService {
      String atPathFor(final Object domainObjectToClassify);
  }
  ```

  This allows the module to find which taxonomies are applicable to the domain object.

- implement the `ClassificationRepository.SubtypeProvider` SPI interface:

```
public interface SubtypeProvider {
    Class<? extends Classification> subtypeFor(Class<?> domainObject);
}
```

This tells the module which subclass of `Classification` to use to attach to the "classified" domain object. The `SubtypeProviderAbstract` adapter can be used to remove some boilerplate.

- subclass `T_classify`, `T_unclassify` and `T_classifications` (abstract) mixin classes for the domain object.

  These contribute the "classifications" collection and actions to add and remove `Classification`s.

Typically the SPI implementations and the mixin classes are nested static classes of the `Classification` subtype.

For example, in the domain app's example module the `DemoObject` can be classified by virtue of the `ClassificationForDemoObject` subclass:

```
@javax.jdo.annotations.PersistenceCapable(identityType= IdentityType.DATASTORE,
schema="incodeClassificationDemo")
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
@DomainObject
public class ClassificationForDemoObject extends Classification {                  ①

    private DemoObject demoObject;
    @Column(allowsNull = "false", name = "demoObjectId")
    @Property(editing = Editing.DISABLED)
    public DemoObject getDemoObject() {                                             ②
        return demoObject;
    }
    public void setDemoObject(final DemoObject demoObject) {
        this.demoObject = demoObject;
    }

    public Object getClassified() {                                                 ③
        return getDemoObject();
    }
    protected void setClassified(final Object classified) {
        setDemoObject((DemoObject) classified);
    }

    @DomainService(nature = NatureOfService.DOMAIN)
    public static class ApplicationTenancyServiceForDemoObject
                    implements ApplicationTenancyService {                          ④
        @Override
        public String atPathFor(final Object domainObjectToClassify) {
            if(domainObjectToClassify instanceof DemoObject) {
                return ((DemoObject) domainObjectToClassify).getAtPath();
            }
```

```
            return null;
        }
    }

    @DomainService(nature = NatureOfService.DOMAIN)
    public static class SubtypeProvider
            extends ClassificationRepository.SubtypeProviderAbstract {              ⑤
        public SubtypeProvider() {
            super(DemoObject.class, ClassificationForDemoObject.class);
        }
    }

    @Mixin
    public static class _classifications extends T_classifications<DemoObject> {   ⑥
        public _classifications(final DemoObject classified) {
            super(classified);
        }
    }
    @Mixin
    public static class _classify extends T_classify<DemoObject> {
        public _classify(final DemoObject classified) {
            super(classified);
        }
    }
    @Mixin
    public static class _unclassify extends T_unclassify<DemoObject> {
        public _unclassify(final DemoObject classified) {
            super(classified);
        }
    }
}
```

① extend from `Classification`

② the type-safe reference property to the "classified" domain object (in this case `DemoObject`). In the RDBMS this will correspond to a regular foreign key with referential integrity constraints correctly applied.

③ implement the hook `setClassified(…)` method to allow the type-safe reference property to the "classified" (in this case `DemoObject`) to be set. Also implemented `getClassified()` similarly

④ implementation of the `ApplicationTenancyService` for the domain object, telling the module the app tenancy of the domain object to be classified. If there is no implementation of this service (but the mixins have been defined) then the contributed collections and actions will still be visible but the collection will remain empty and the actions disabled.

⑤ implementation of the `SubtypeProvider` SPI domain service, telling the module which subclass of `Classification` to instantiate to attach to the "classified" domain object

⑥ mixins for the collections and actions contributed to the "classified" domain object

# UI Concerns

The attached `Classification` objects are shown in two contexts: as a table of `Classification` objects for the "classified" domain object, and then as the actual subtype when the classification object itself is shown (eg `ClassificationForDemoObject` in the demo app).

In the former case (as a table) the `Classification` will be rendered according to the `Classification.layout.xml` provided by the module. In the latter (as an object) the classification will be rendered according to the layout provided by the consuming app, offering full control of the layout. The layout provided in the example module of the domain app (ie `ClassificationForDemoObject.layout.xml`) is a good starting point.

The module also allows the title, icon and CSS for `Classification`, `Category` and `Applicability` objects to be customised. In all three cases this done using subscribers. By default the values of the title/icon/CSS class is obtained using default subscribers, eg `Classification.TitleSubscriber`, `Classification.IconSubscriber` and `Classification.CssClassSubscriber`. The consuming module can override these values simply by providing alternative implementations.

# Other Services

The module provides the following domain services for querying aliases:

- `CategoryRepository`

  To search for existing `Category`s, and to create top-level `Taxonomy`s. Children are created from `Category` itself.

- `ClassificationRepository`

  To search for `Classification`s, ie the tuple that links an `Category` with an arbitrary "classified" domain object.

# Known issues

None known at this time.

# Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/dom/classification/impl -D excludeTransitive=true
```

which, excluding the Apache Isis modules, returns no direct compile/runtime dependencies.

The module *does* use icons from icons8.