Document Subdomain

Table of Contents

This module (`incode-module-document`) provides the ability to create and attach `Document` objects to arbitrary domain objects from `DocumentTemplate`s.

`Document`s can be rendered using a number of technologies:

- Apache Freemarker (using the Freemarker DocRendering library module)
- stringinterpolator module to interpolate and download a URL (using the StringInterpolator DocRendering module).

  This can be used to integrate with pre-existing reporting servers (for example Microsoft's SQL Server Reporting Services).

- XDocReport (using the XDocReport DocRendering module).

The rendering mechanism is pluggable; additional implementations can be plugged in as required.

Also, the rendering can be done either in the foreground or the background. A "backgroundCommands" mixin collection will show any such background commands scheduled for a `Document`.

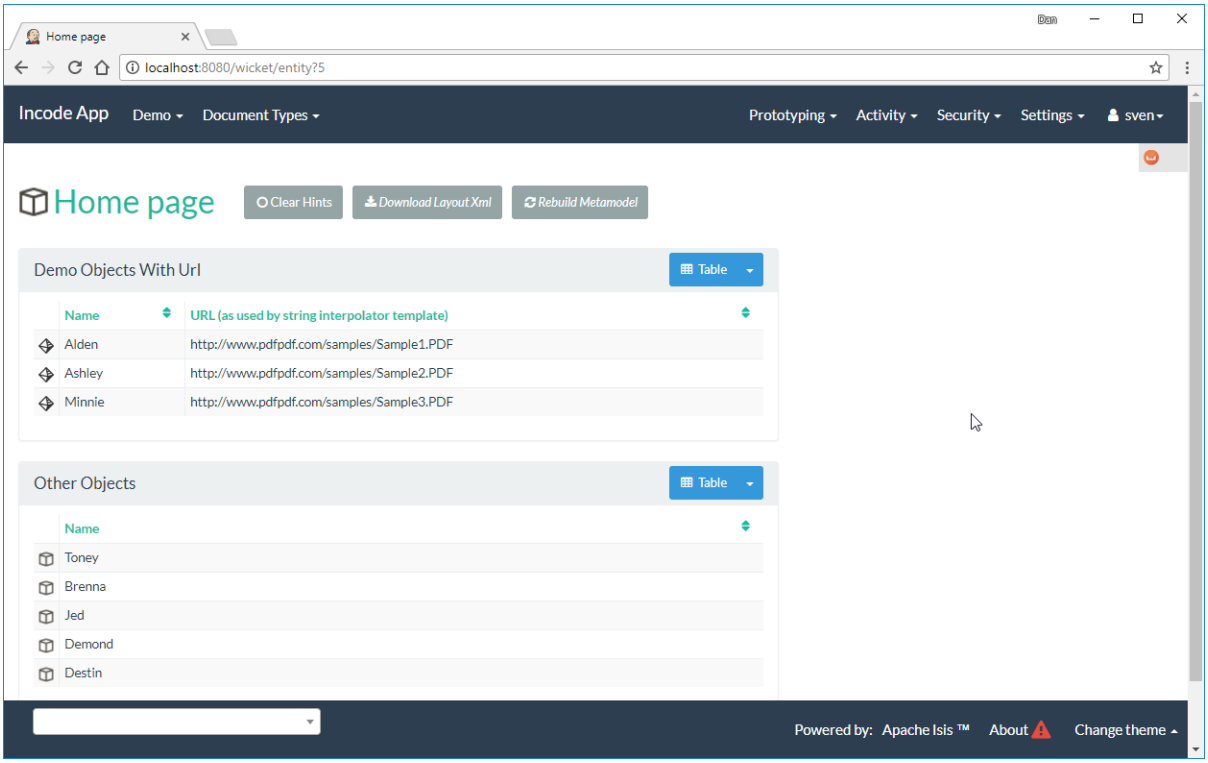
Once a `Document` has been created, other supporting PDFs (for example tax receipts for a generated invoice) can then be attached to it. A `Document` is created to wrap these PDFs. Like all `Document`s, a `DocumentType` to required which helps categorise the supporting document, but no corresponding `DocumentTemplate` is required because they are never rendered (they exist already). An SPI service (`DocumentAttachmentAdvisor`) is used to obtain the list of available `DocumentType`s to use.

The module implements a rule that chains of `Document`s are not allowed: a `Document` either has supporting `Document`s attached to it, or it *is* a supporting `Document`. In the former case a "supportingDocuments" mix-in collection lists all the supporting `Document`s that have been attached. In the latter, a "supports" mixin property points back to the `Document` being supported.

Screenshots

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

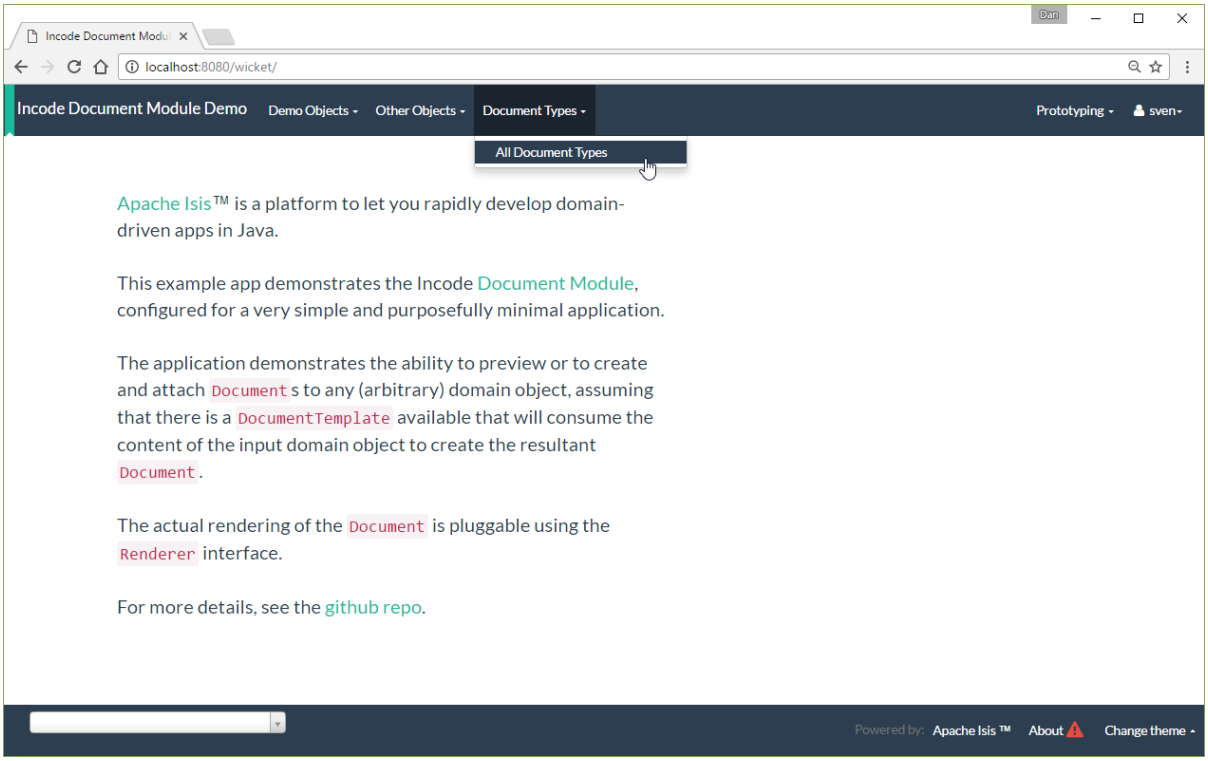
A home page is displayed when the app is run:



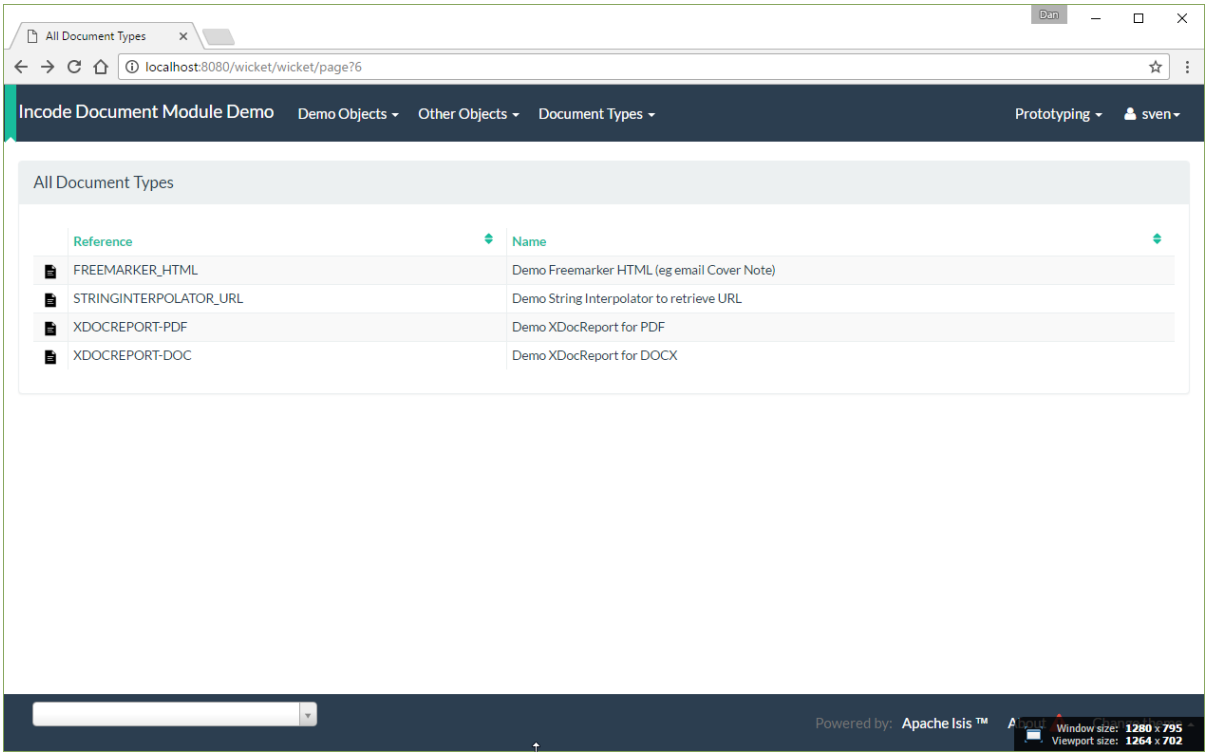
The remaining screenshots below **do** demonstrate the functionality of this module, but are out of date in that they are taken from the original isisaddons/incodehq module (prior to being amalgamated into the incode-platform).

Document Types

The app manifest's fixture script defines a set of `DocumentType`s. These are reference data:



Four example `DocumentType`s are set up, showcasing the four `RenderingStrategy`s provided by the Freemarker docrendering library, StringInterpolator docrendering library and XDocReport docrendering library modules:



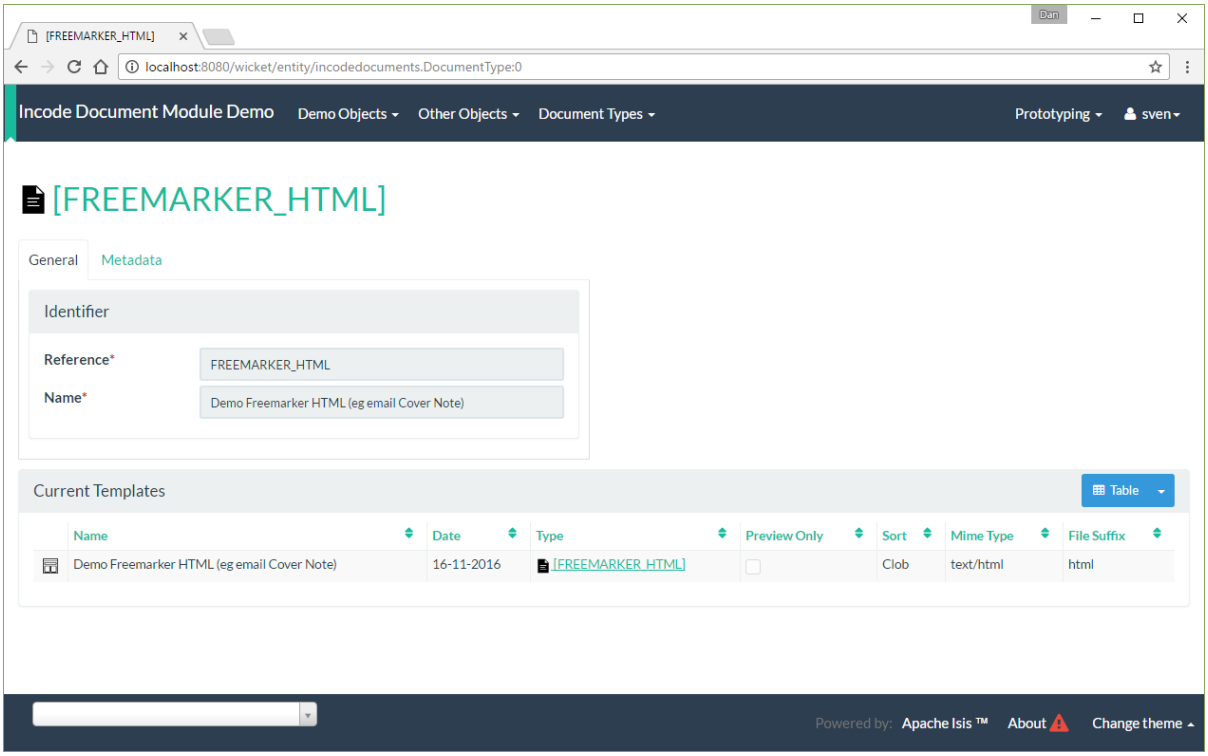
The two `DocumentType`s for XDocreport are very similar; the difference is only that one results in a PDF, while the other results in a Word `.docx` document.

Freemarker Document Template

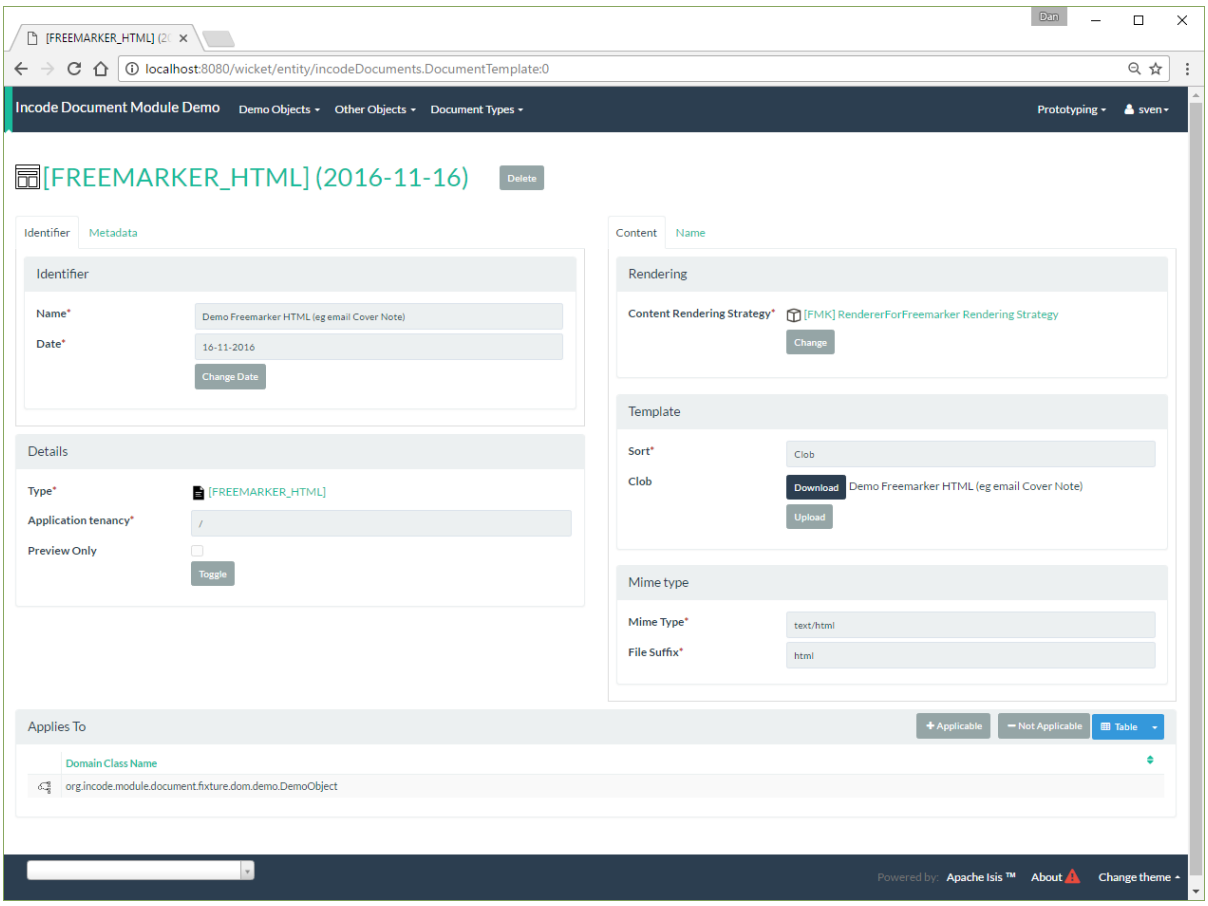
A `DocumentType` holds a collection of `DocumentTemplate`s, by date. This allows new versions of

template to be altered/evolved over time.

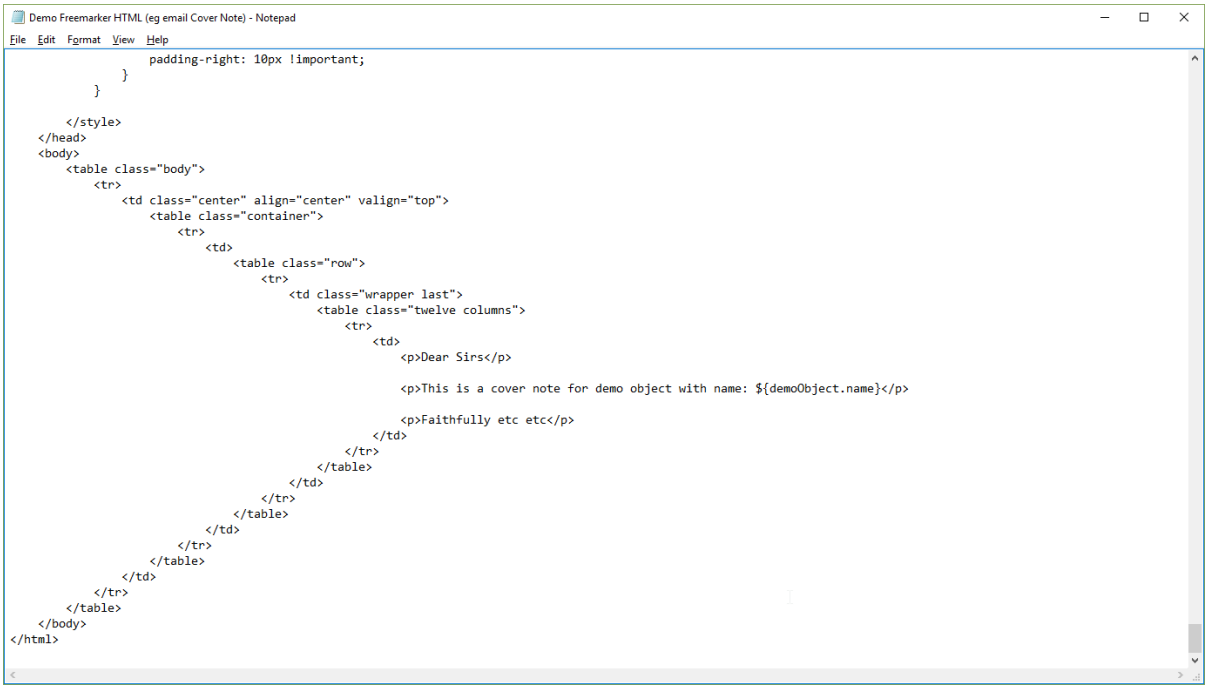
The example `DocumentType` for Freemarker defines just a single `DocumentTemplate`:



The `DocumentTemplate` contains template text that can be either text, clob or a blob. In the case of the demo freemarker template it is a clob:

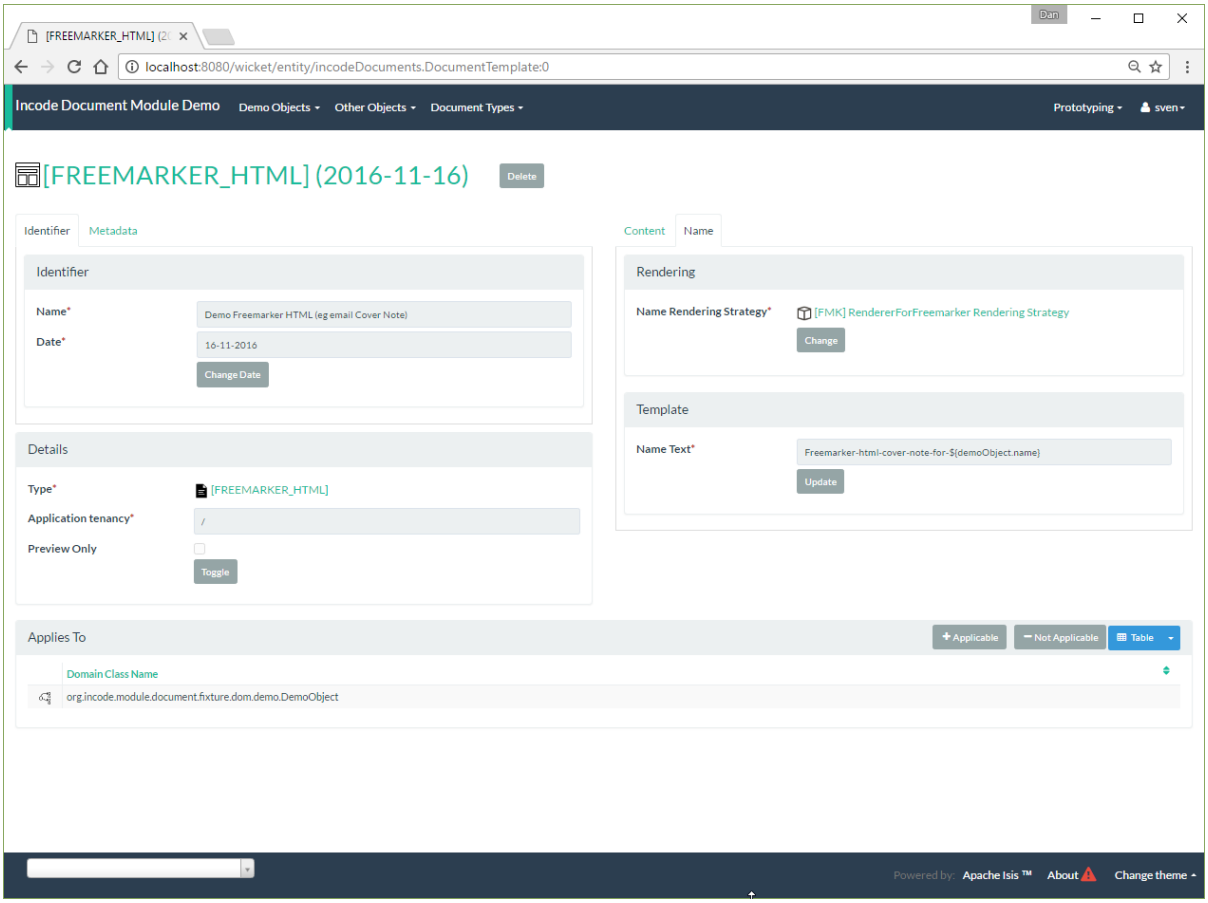


The clob itself is an HTML email file. This can be downloaded from the template:



The template text clob can be modified by uploading new versions. However, once `Document`s have been created from a `DocumentTemplate`, the template should be considered as immutable and not be updated; instead create a new version.

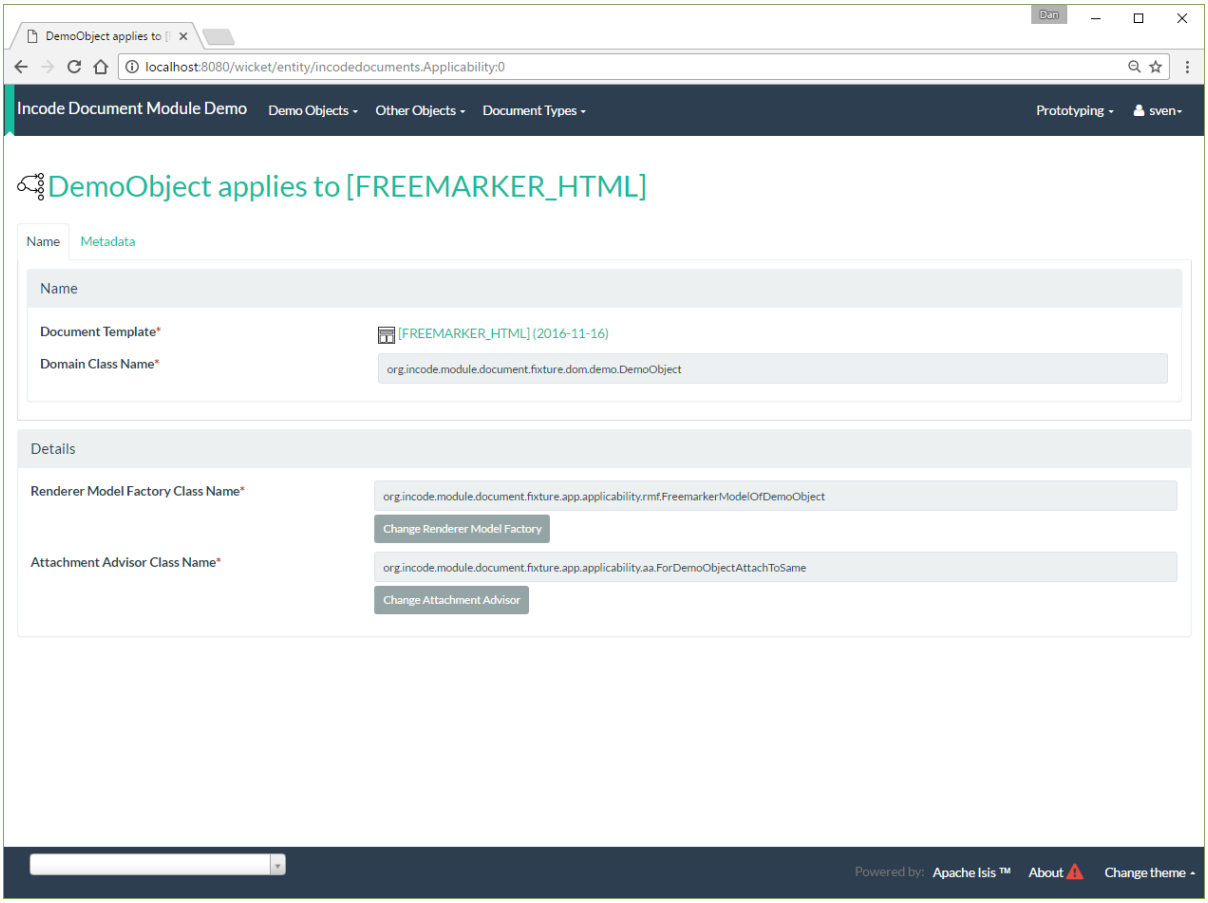
Each `DocumentTemplate` also defines placeholder text for the name of the resultant `Document`:



Along with the "content" and "name" text/clob/blob, the template also specifies the `RenderingStrategy` for each; these are used to interpolate the content/name.

Applicability

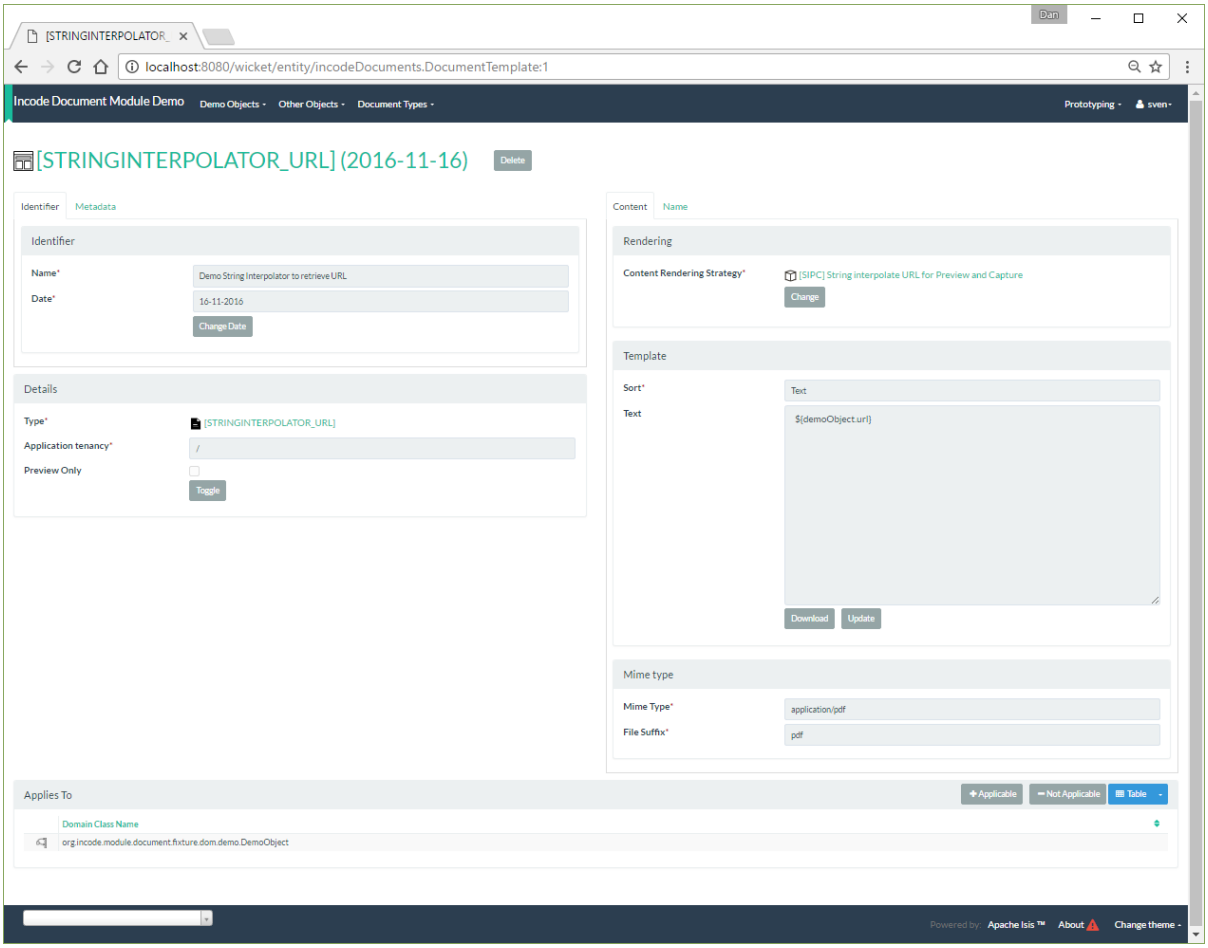
The renderer(s) associated with each `DocumentTemplate` require data (a "renderer model") to interpolate the placeholders in the content/name text; this renderer model ultimately is obtained from a domain object. Obviously not every domain object can be used with every `DocumentTemplate`; the `Applicability` entity catalogues which domain object types can be used as the input to the renderer(s) of its associated `DocumentTemplate`:



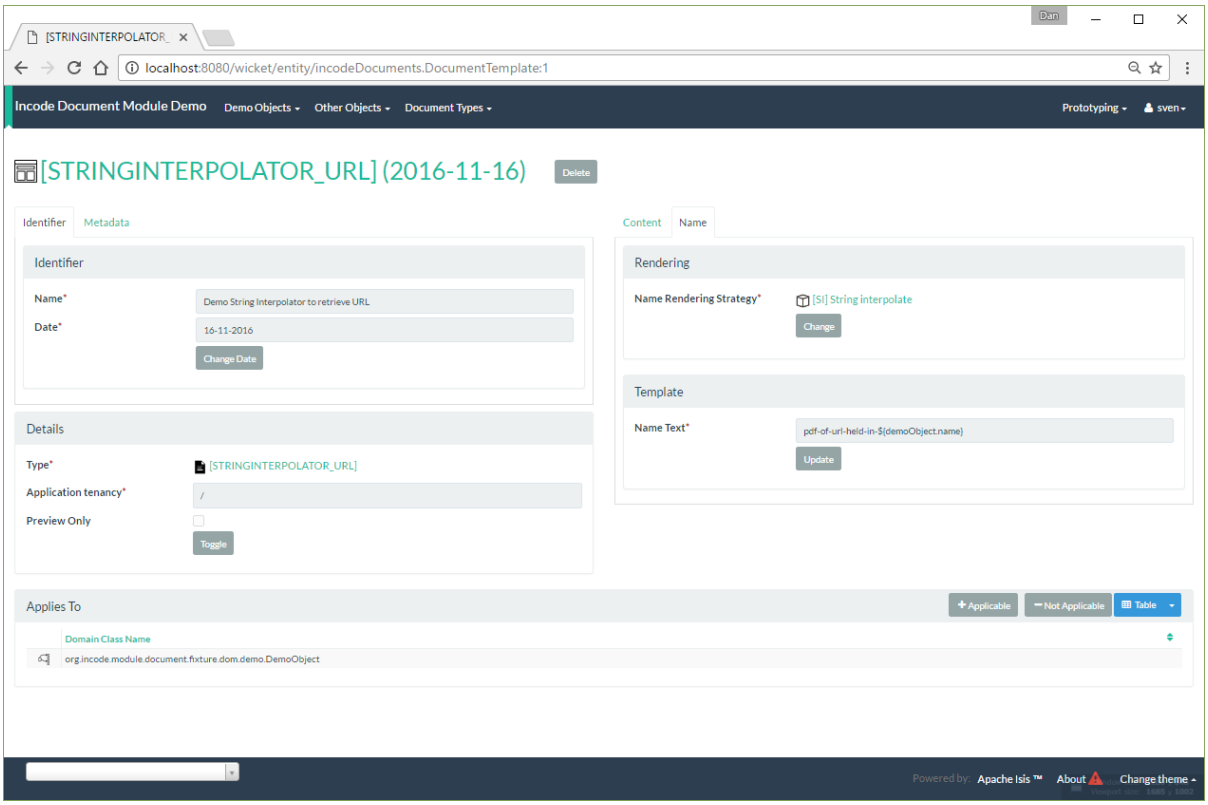
The `RendererModelFactory` of the `Applicability` is used to create the "renderer model" from the input domain object, while the `AttachmentAdvisor` is used to indicate which domain object(s) the resultant `Document` should be attached (often just the input domain object, but potentially to other domain objects also).

String Interpolator Template

The example String Interpolator `DocumentTemplate` obtains its content by interpolating (using the stringinterpolator module) the content placeholder text; the resultant string is parsed as a URL and the contents of that URL downloaded:



The name of `Document`s generated from this template also uses the stringinterpolator module:

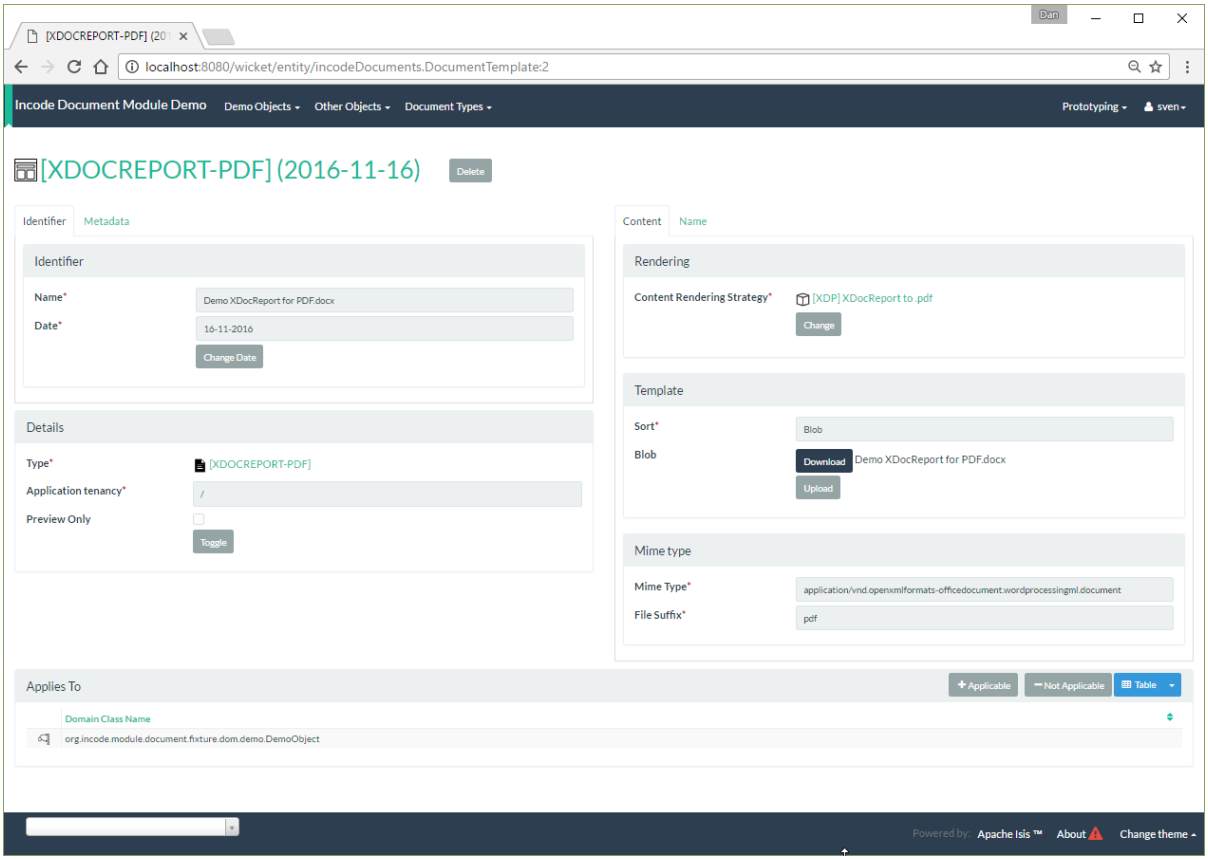


The "renderer model" created (by an `DocumentTemplate`'s `Applicability` for some domain object type) must be compatible with the `RenderingStrategy` for both content and name. This is true for all `DocumentTemplate`s.

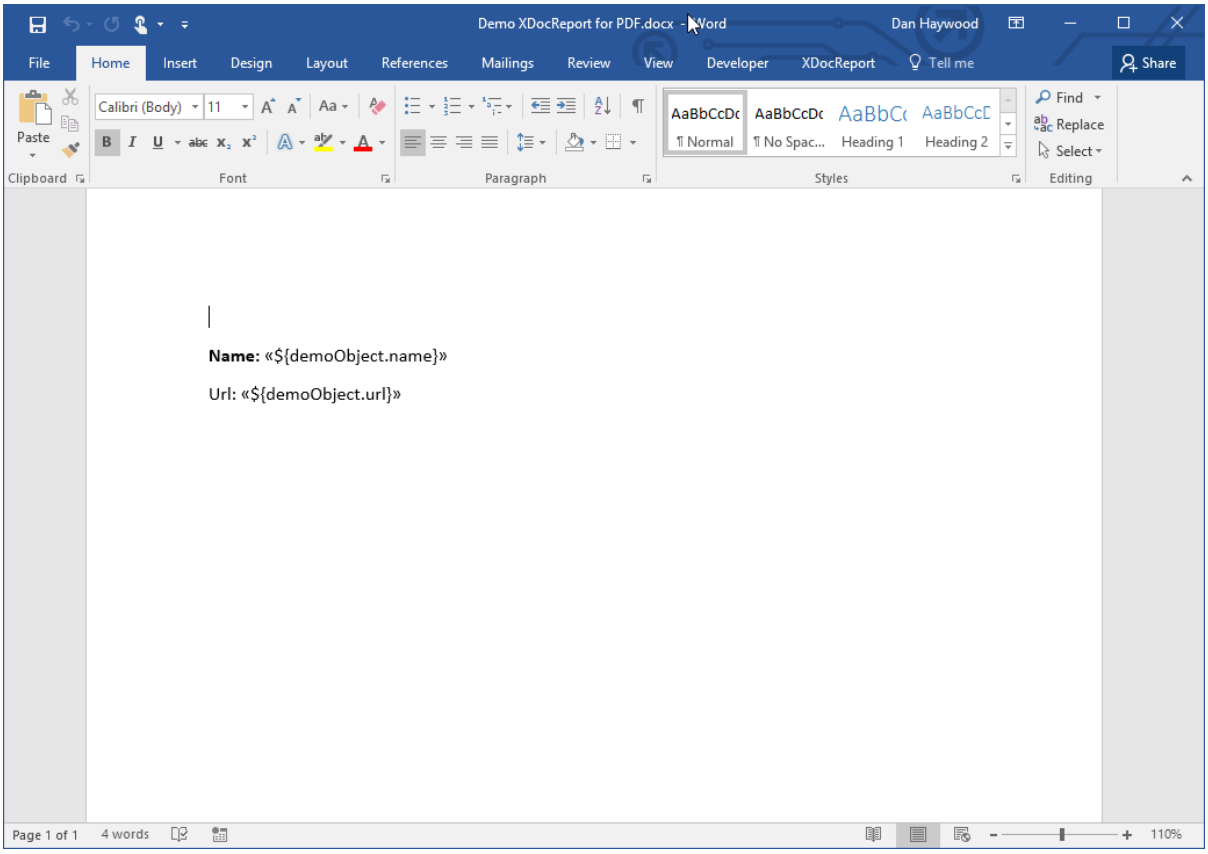
XDocReport Templates

There are two example `DocumentTemplate`s that use XDocReport for rendering. The content in both cases is a Word `.docx` file. The difference between them is simply that one renders this `.docx` and outputs a PDF, while the other produces an outputs another `.docx` file.

The example `DocumentTemplate` for the XDocReportPdf has the following content:



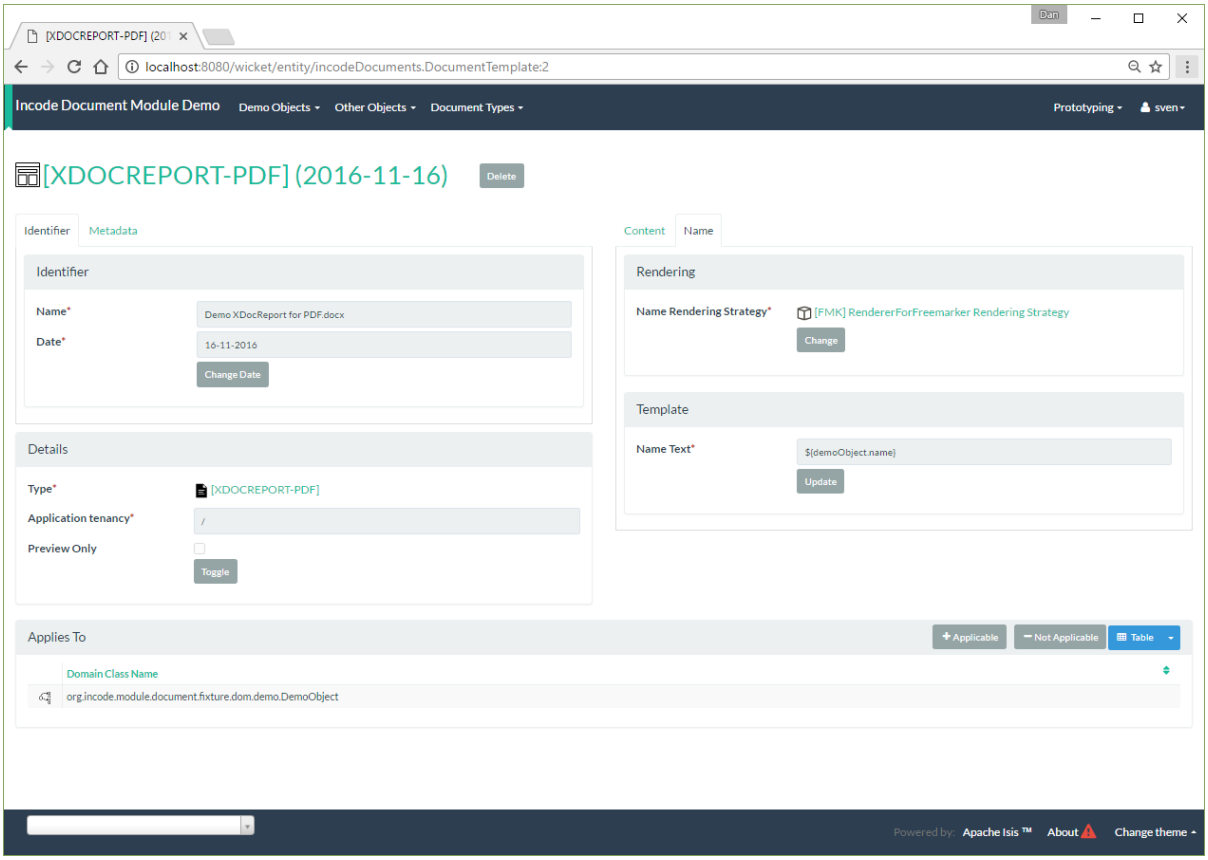
Where the BLOB is a Word document:



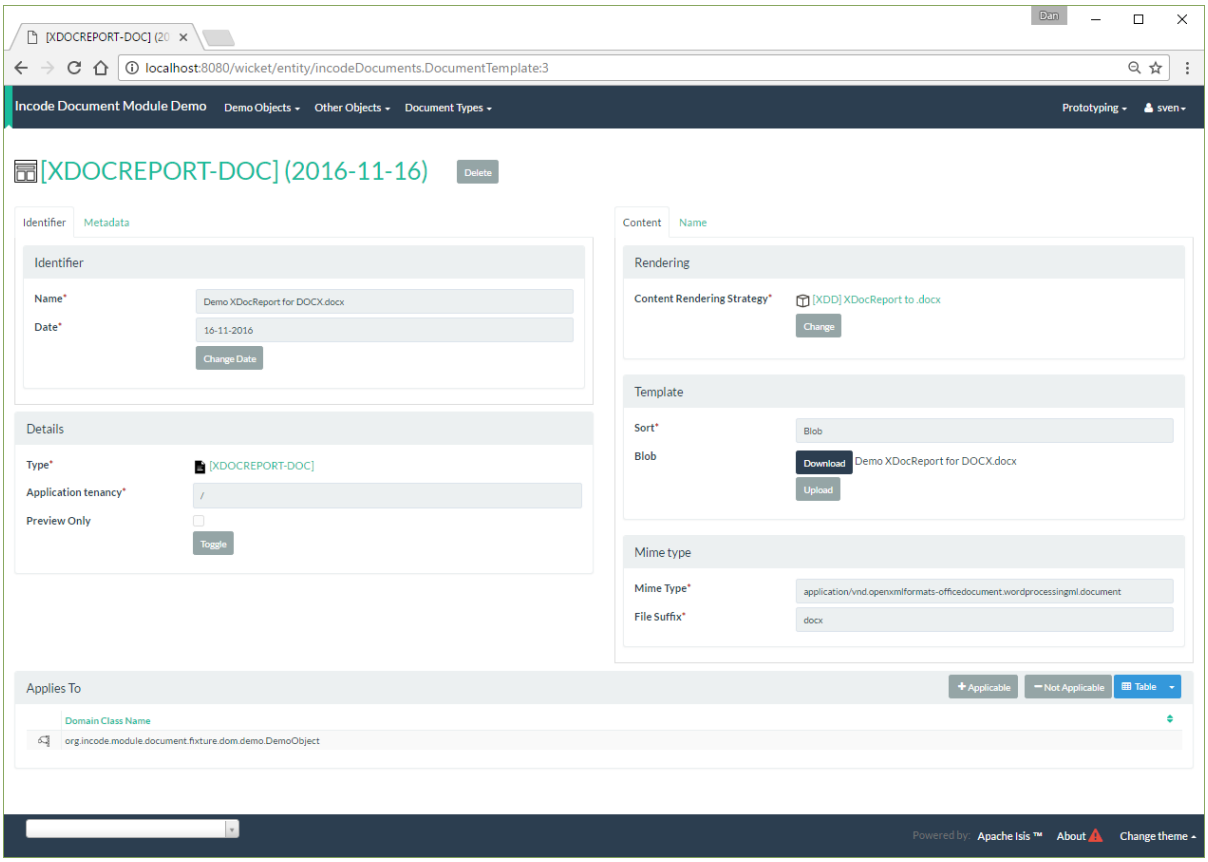
This Word file uses Freemarker placeholders.

While XDocReport itself as a technology supports both Freemarker and Velocity, the integration here (in XDocReport docrendering library module) allows only Freemarker to be used.

The name text (used to create the name of the resultant `Document`) is also interpolated using Freemarker:



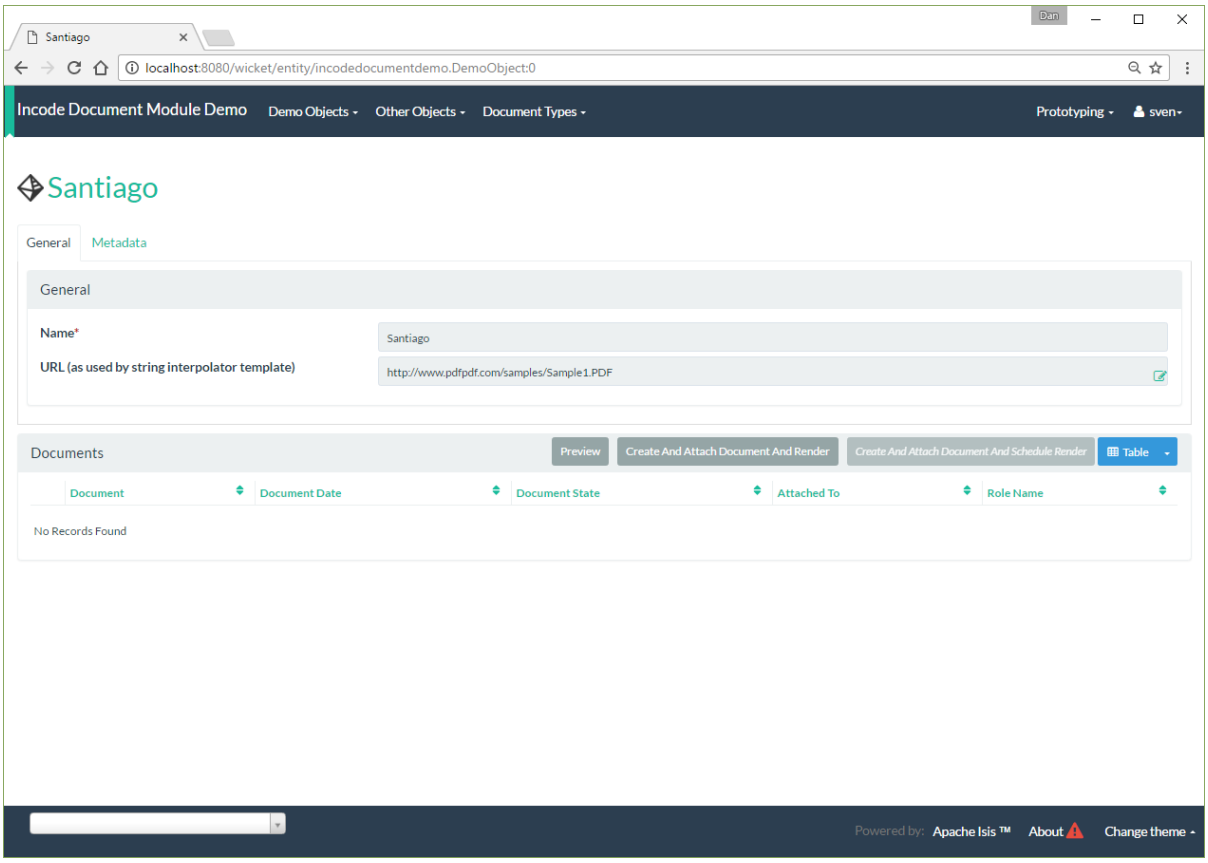
The content of example `DocumentTemplate` for XDocReportDocx is almost identical:



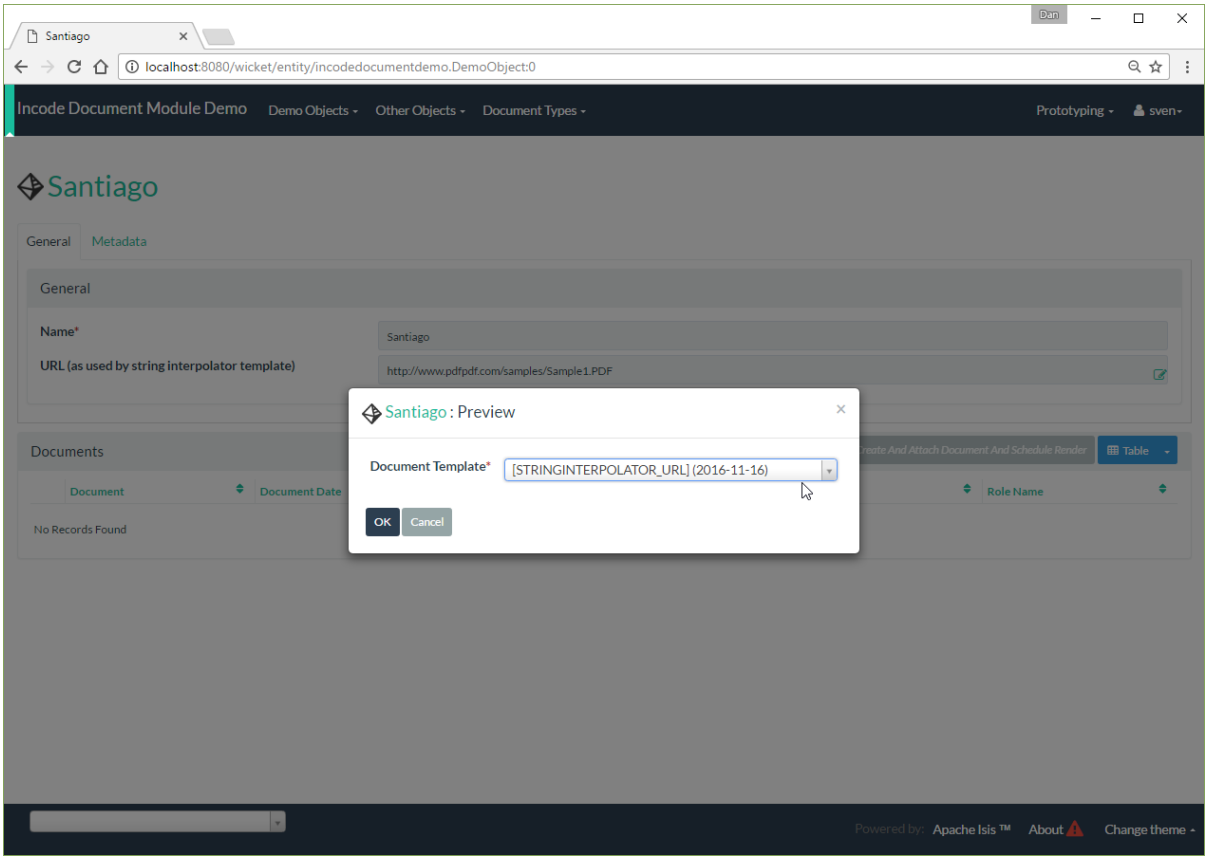
The only difference is that a different `RenderingStrategy` is used.

Previewing Documents

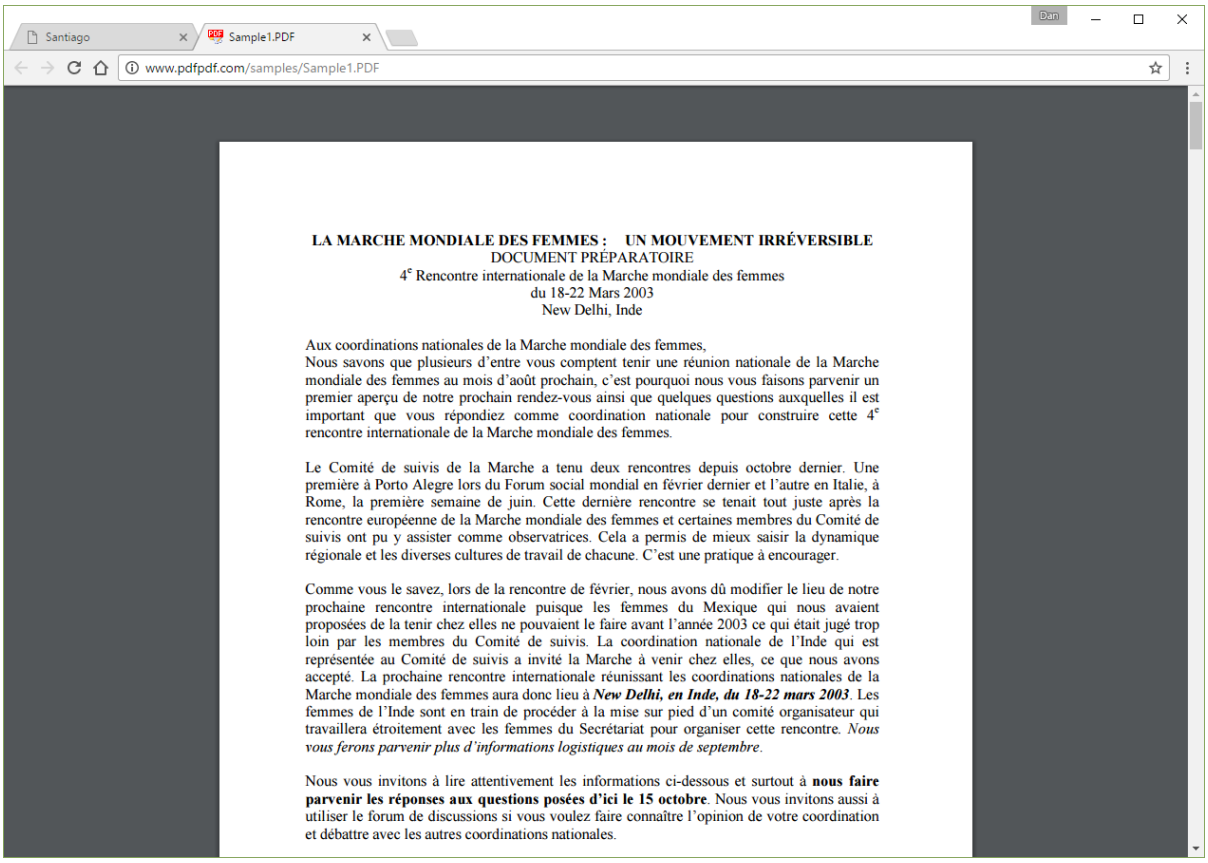
The fixture script also defines a number of demo domain objects, set up to allow `Document`s to be generated from them (for all the `DocumentTemplate`s described above) and for those resultant `Document`s to be attached to them:



In the case of the String Interpolator `DocumentTemplate`, this also supports previewing:

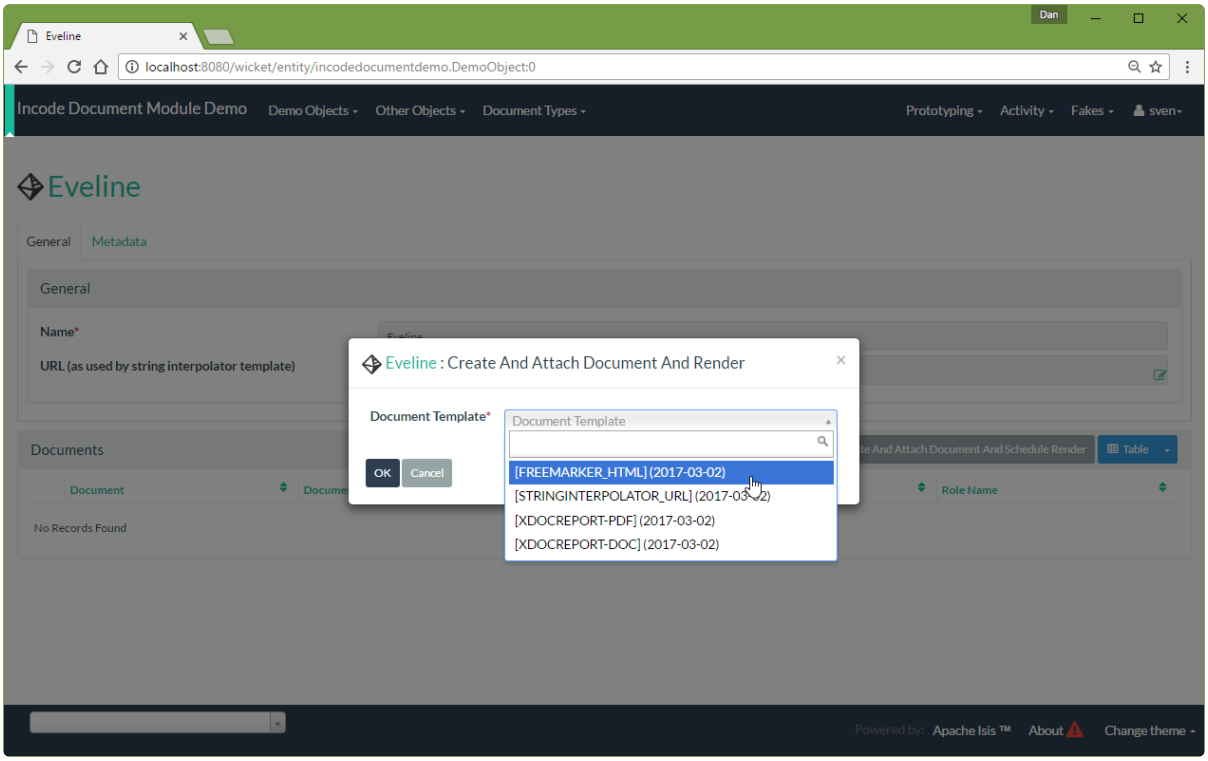


The resultant URL is opened up as a new tab; no new `Document` is created:

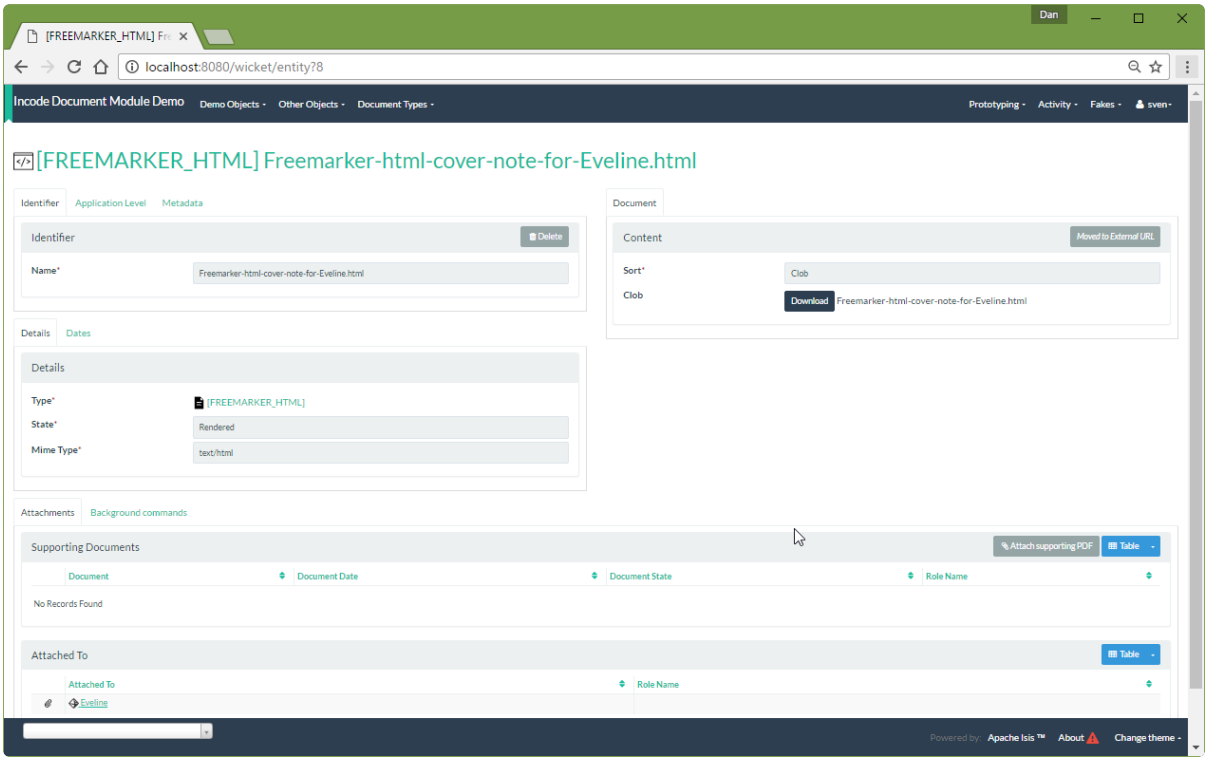


Generating Documents

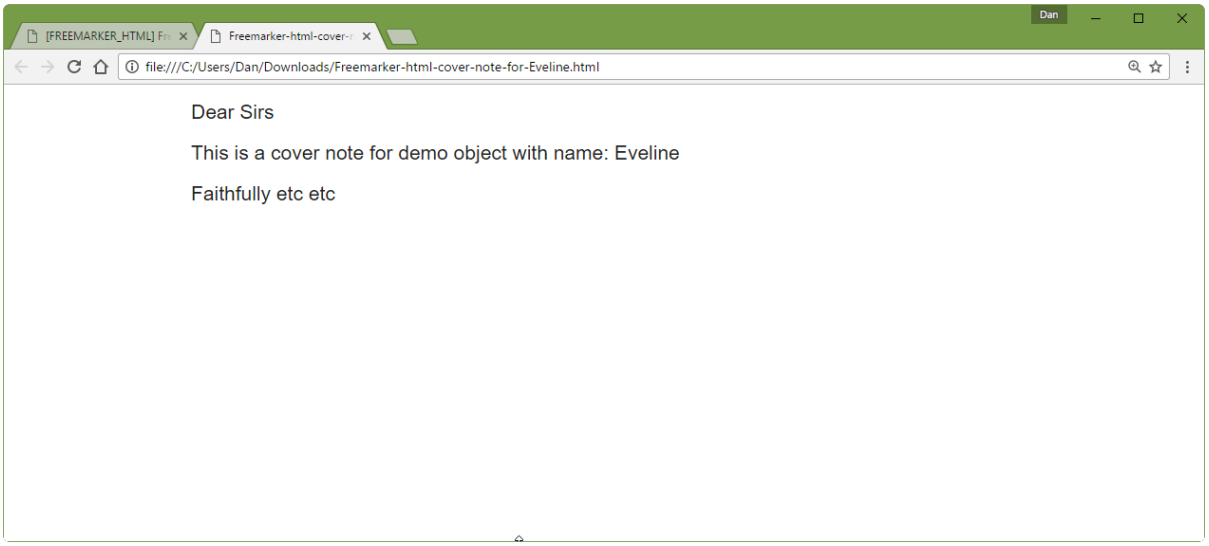
Generating a `Document` for the Freemarker `DocumentTemplate`:



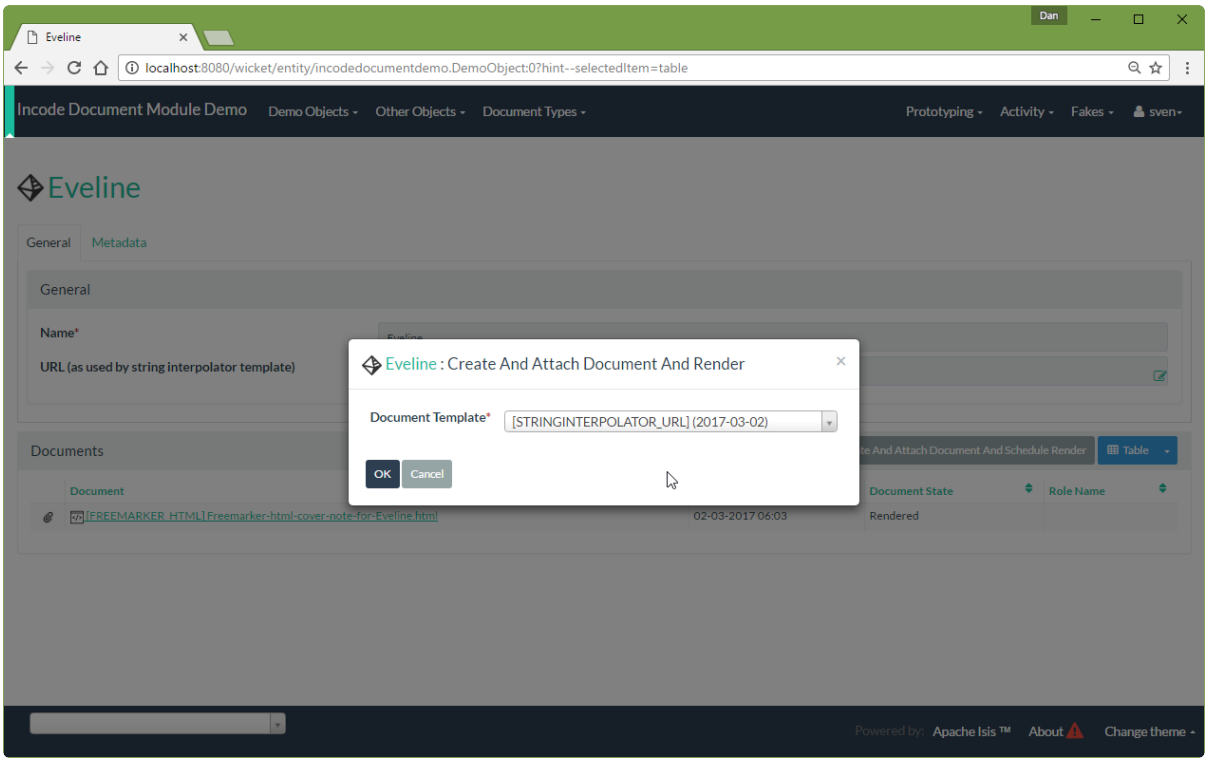
Results in a new `Document` attached to the demo object:



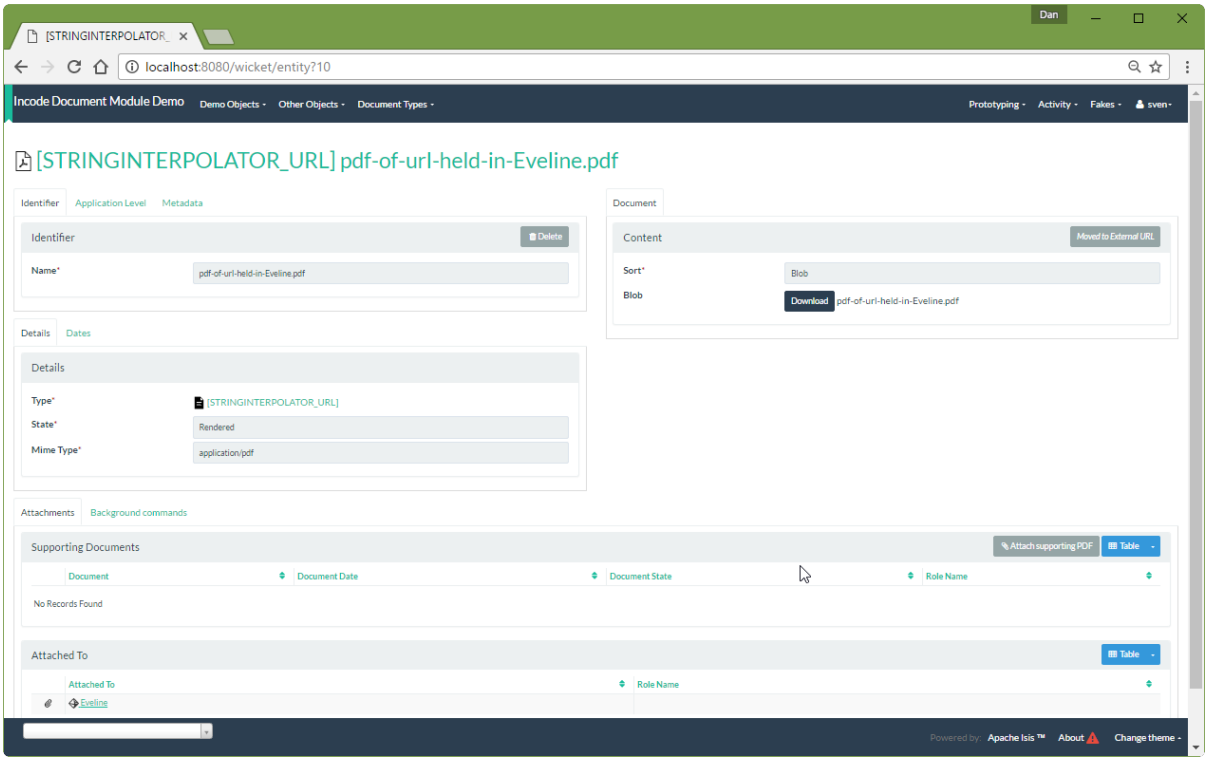
The content of this `Document` (HTML text) has correctly interpolated the details from the input demo object:



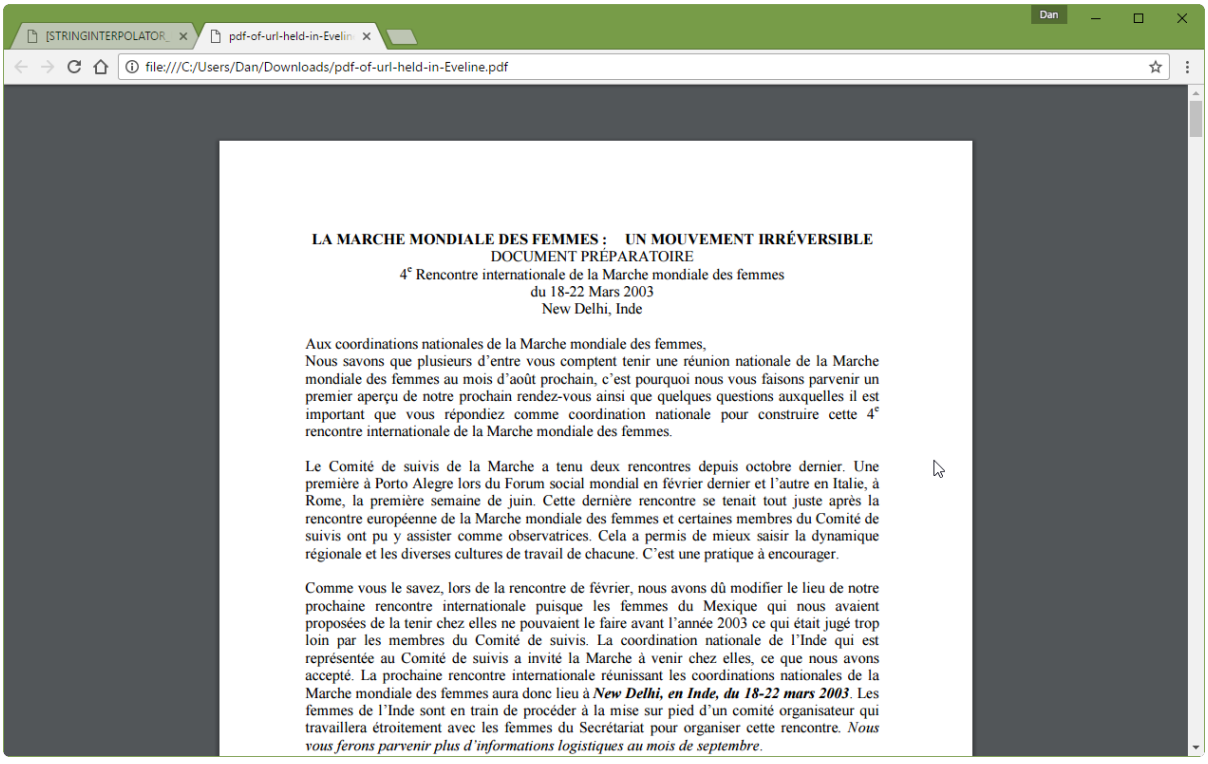
The StringInterpolator `DocumentTemplate` can similarly be used:



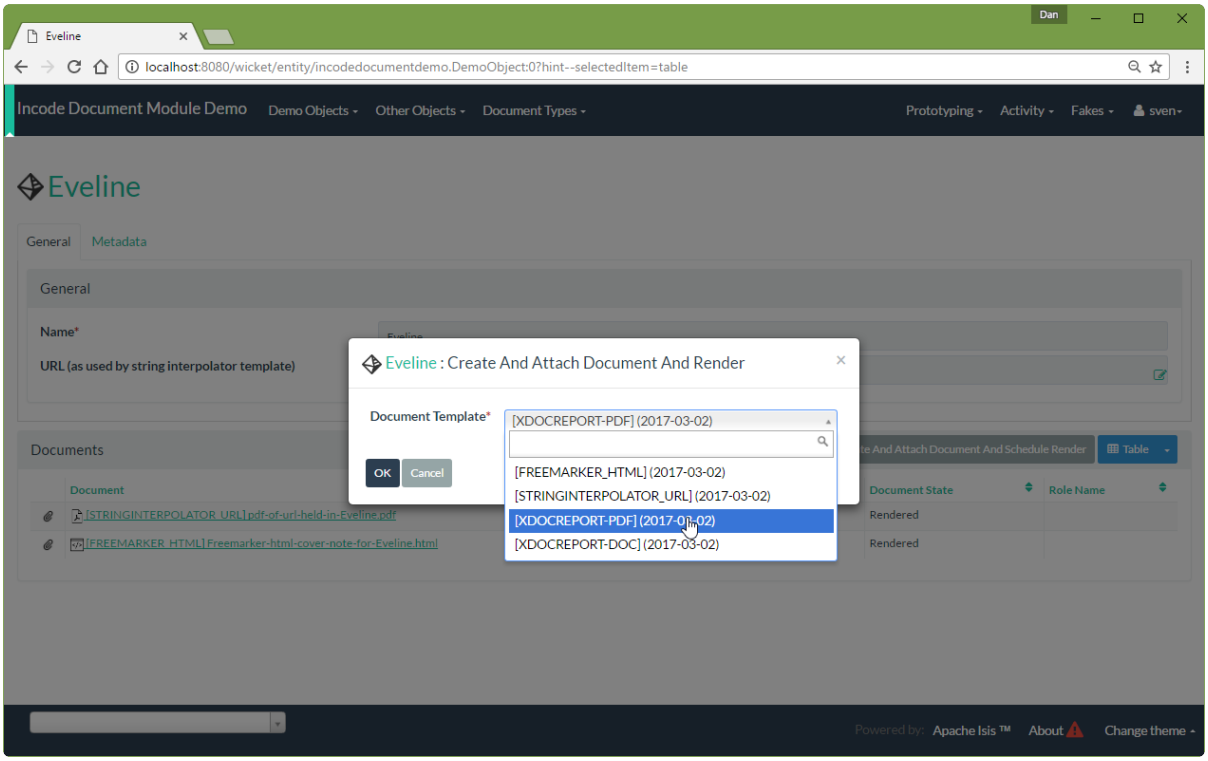
To create a new `Document` attached to the demo object:



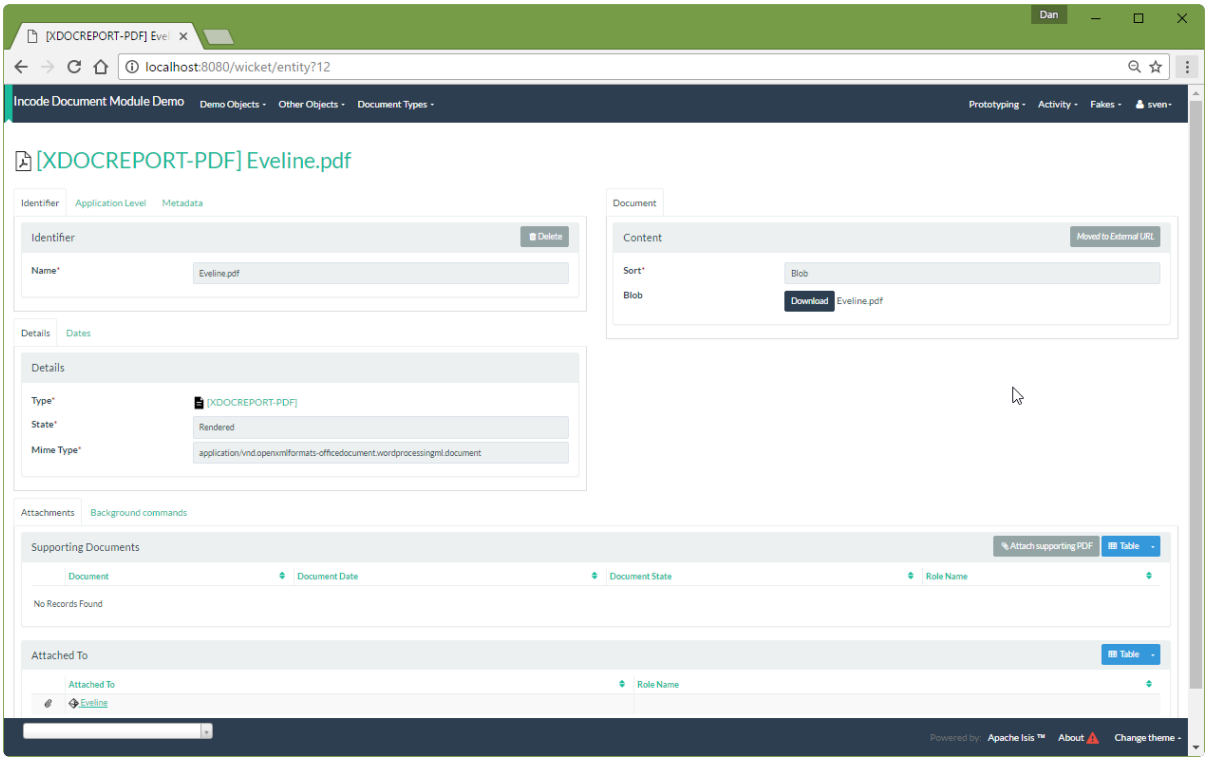
Its content is the contents of the interpolated URL:



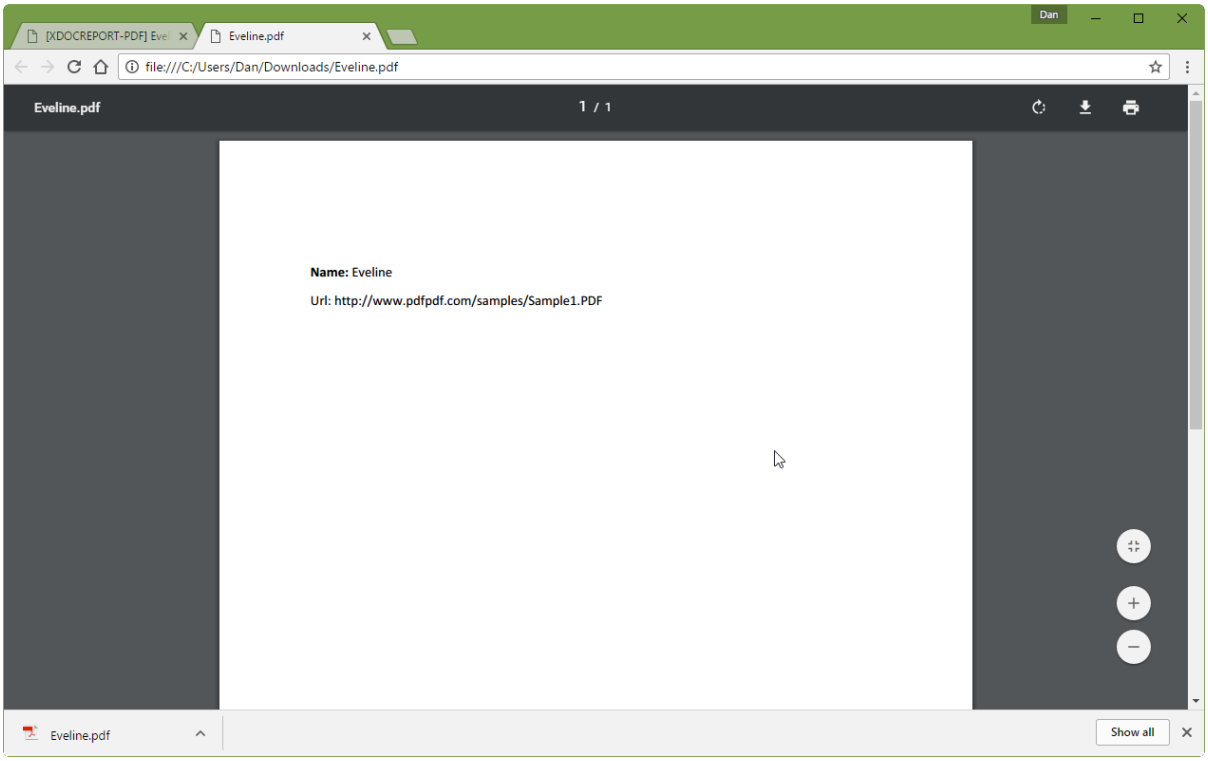
And again, the XDocReportPdf `DocumentTemplate` can be used:



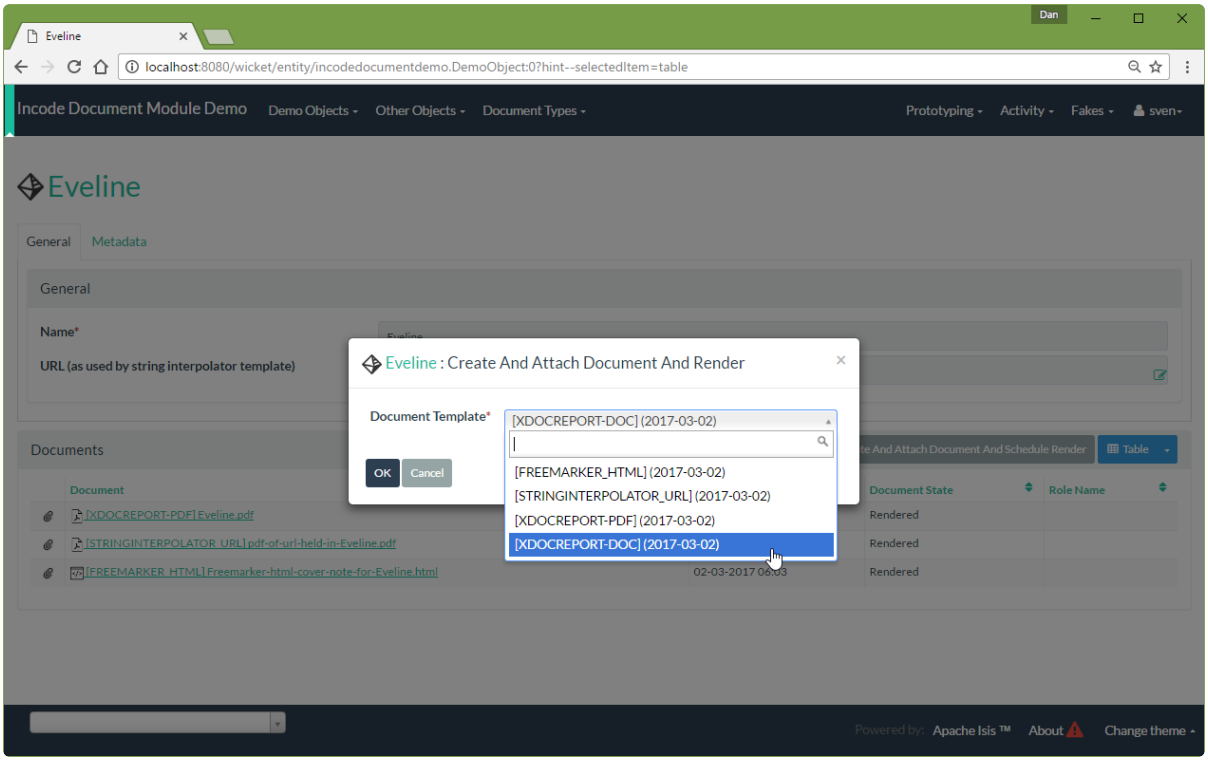
To create a new `Document` attached to the demo object:



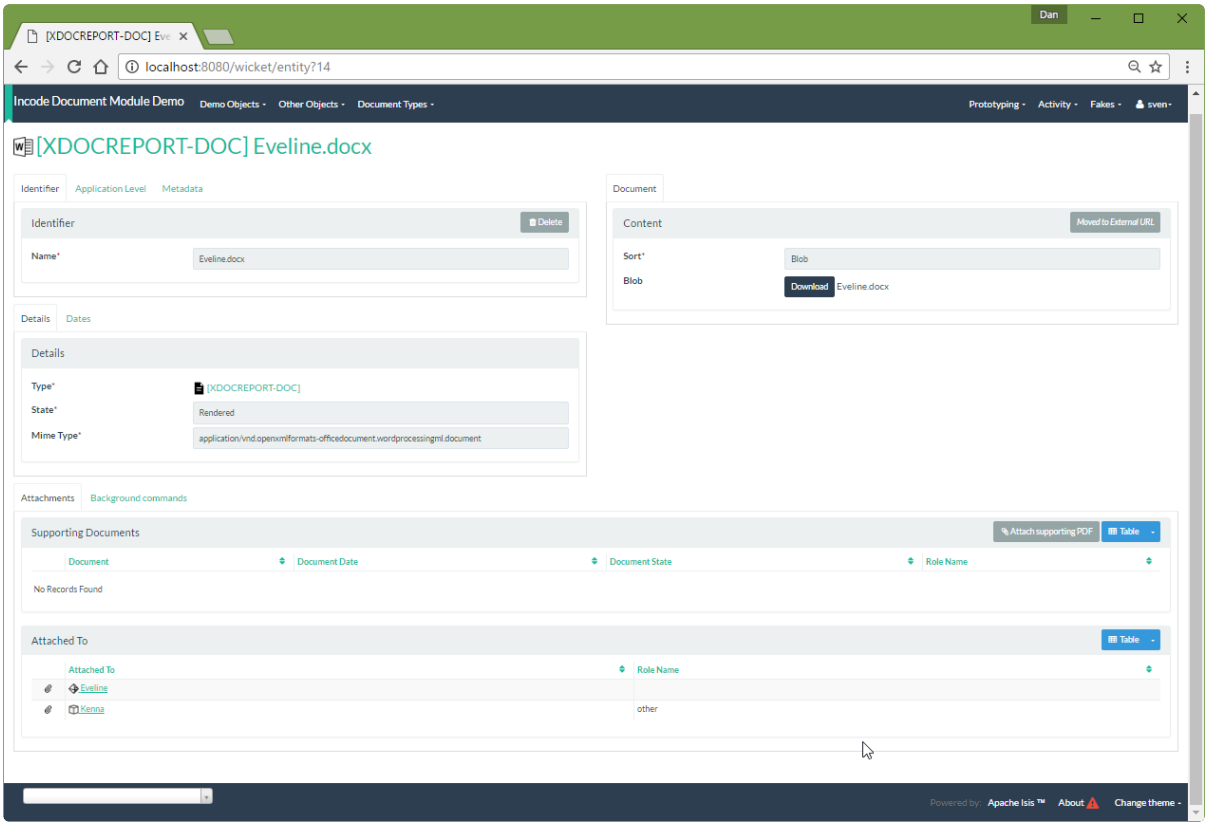
Its content is a PDF generated from the Word `.docx` of the template:



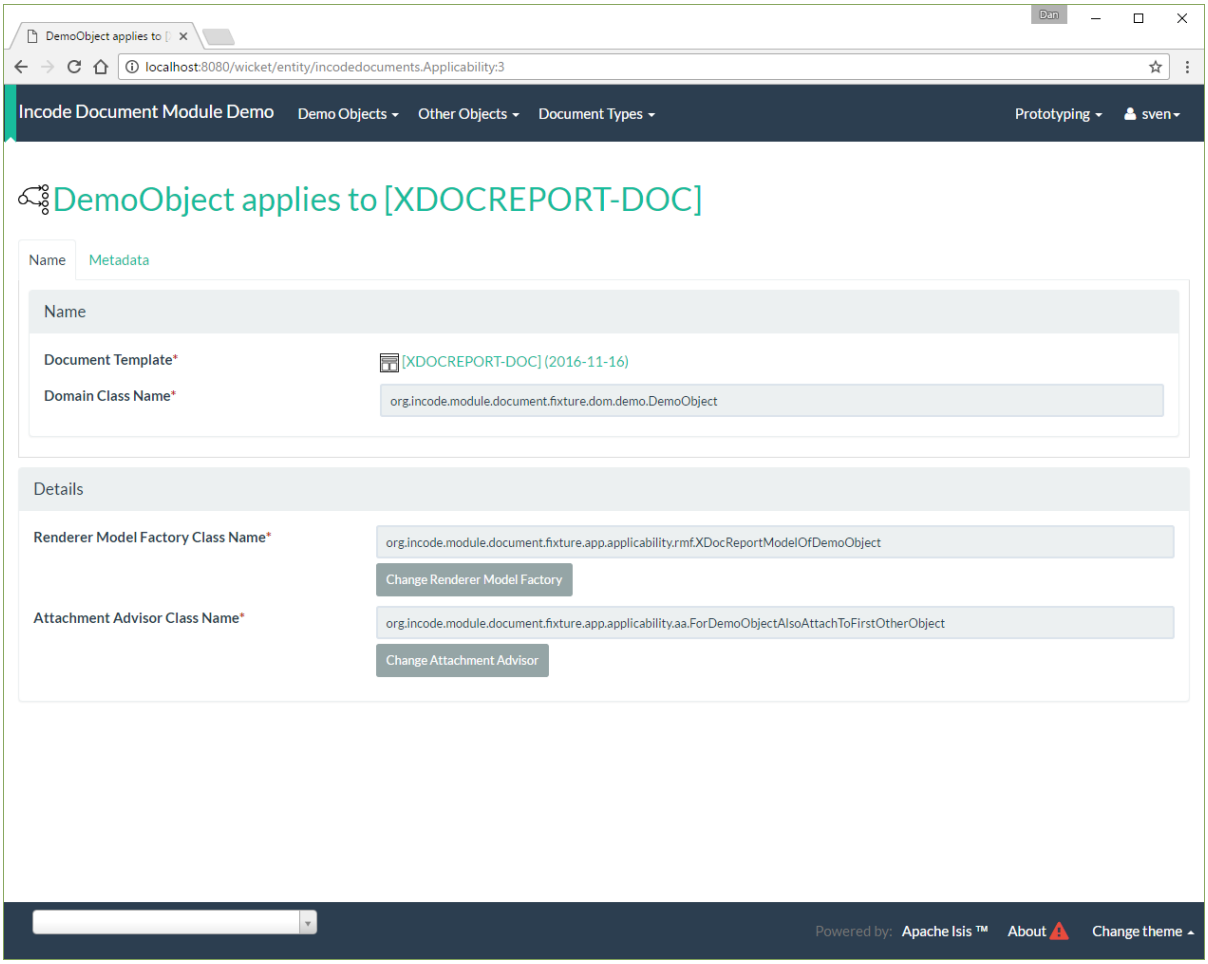
Finally, the XDocReportDoc `DocumentTemplate` can be used:



To create a new `Document` attached to the demo object, where the content is in this case a Word document. To demonstrate that `Document`s can be attached to arbitrary objects, this final template is set up so that the generated `Document` is attached both to the input demo object and also to one other object:

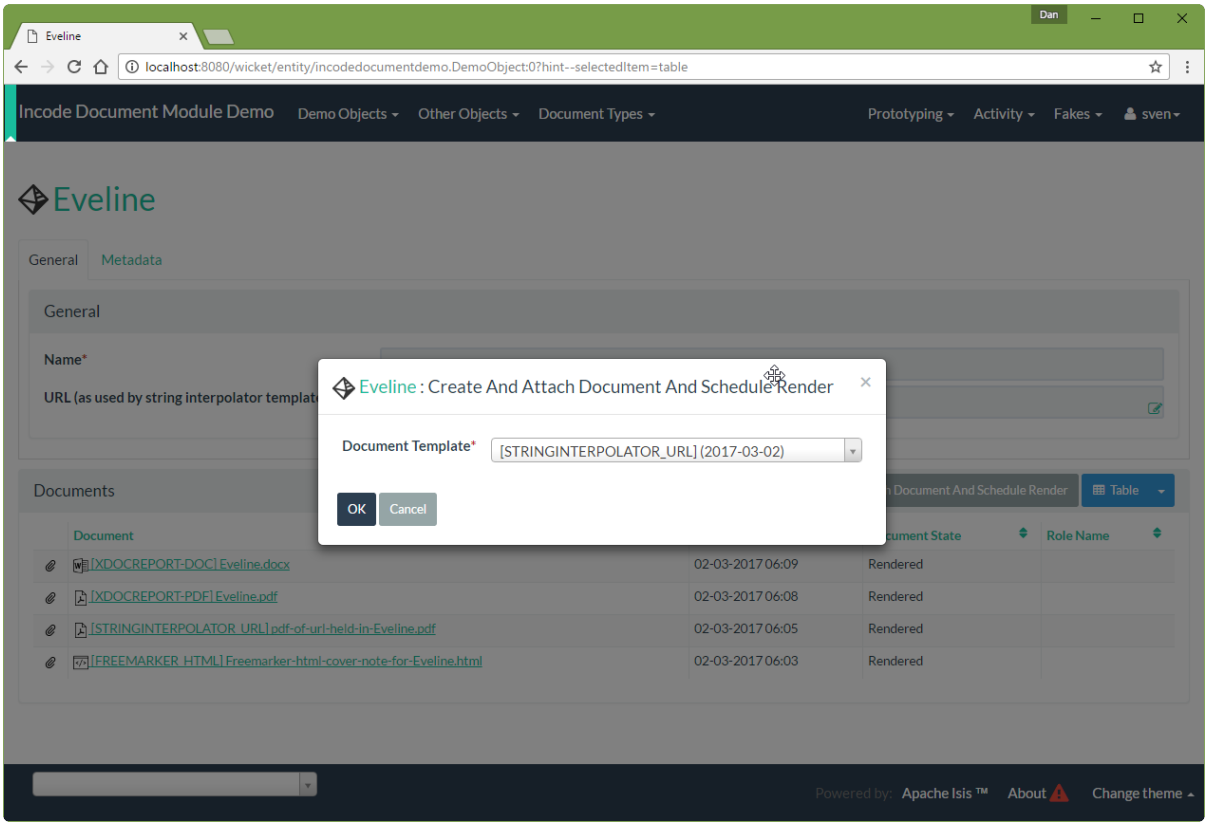


This is configured through the `AttachmentAdvisor` of the relevant `Applicability` of the `DocumentTemplate` for this input demo object's type:

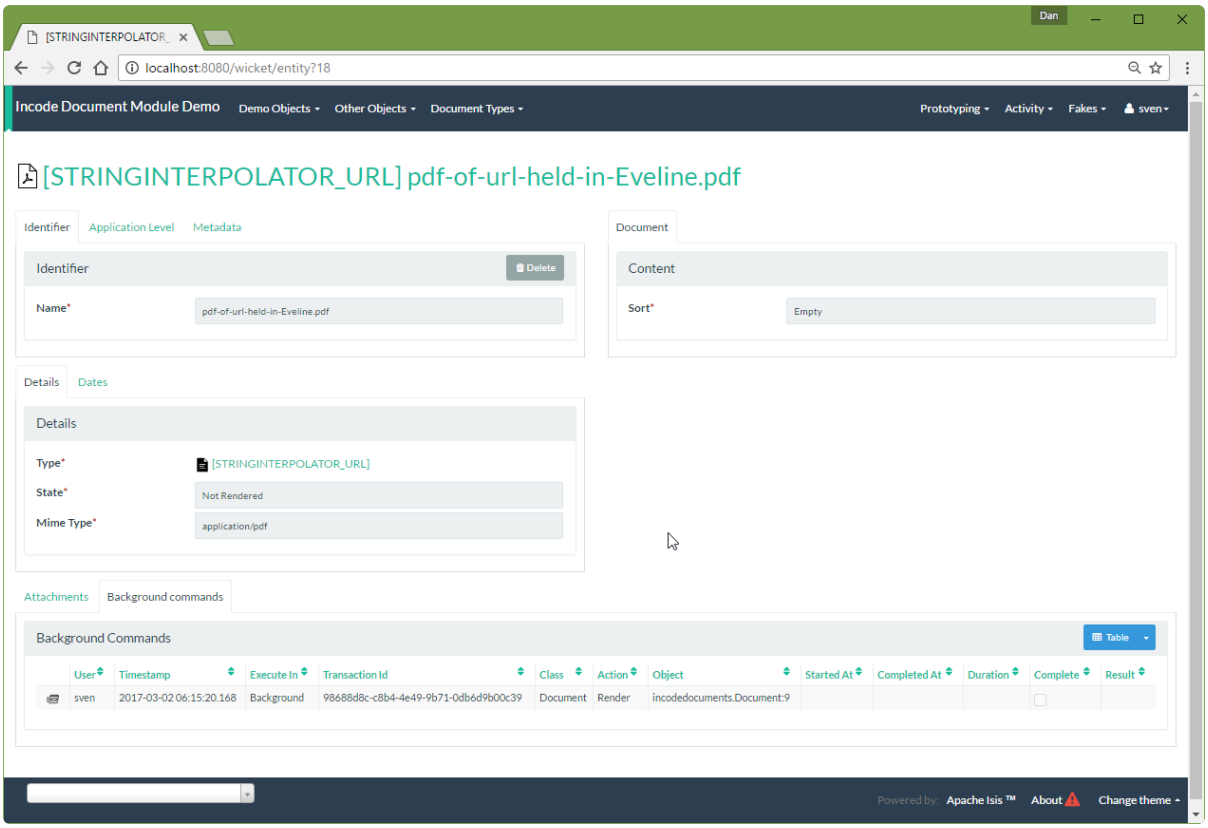


Generating Documents in Background

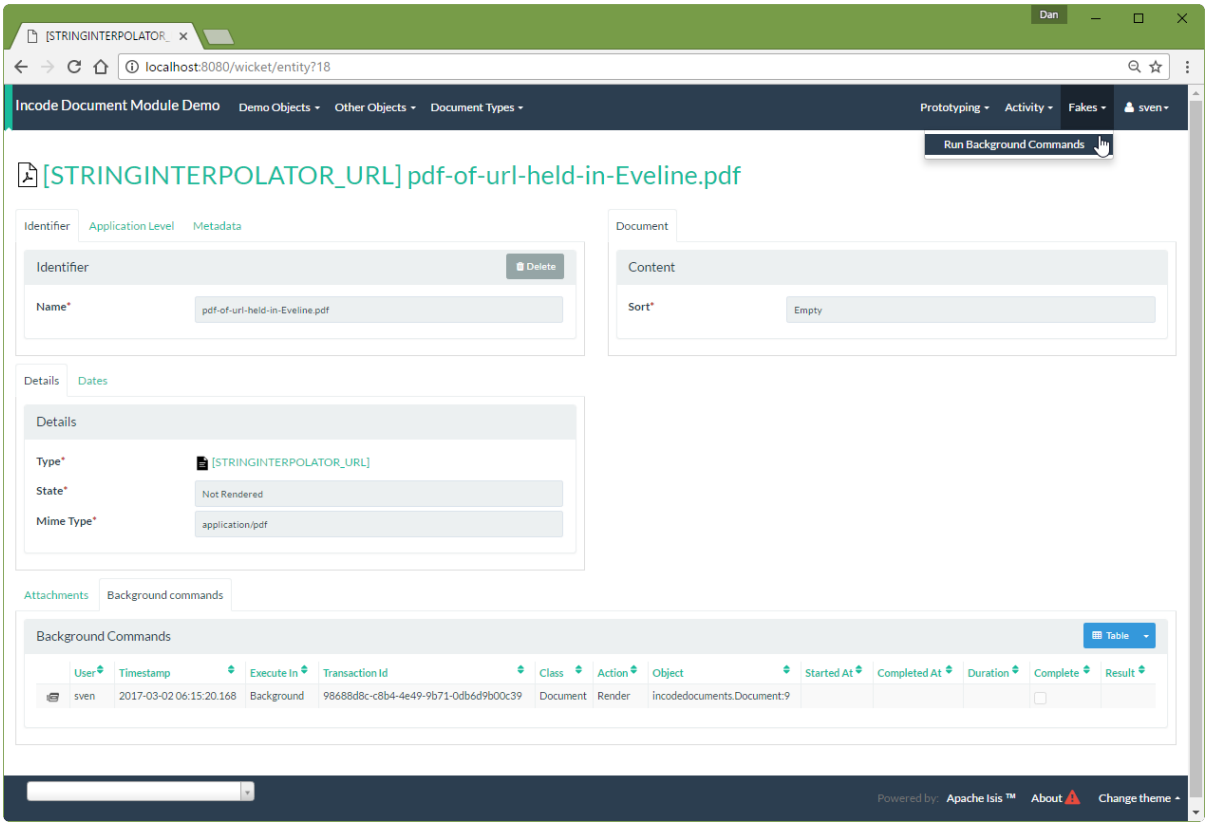
Documents can also be rendered in the background:



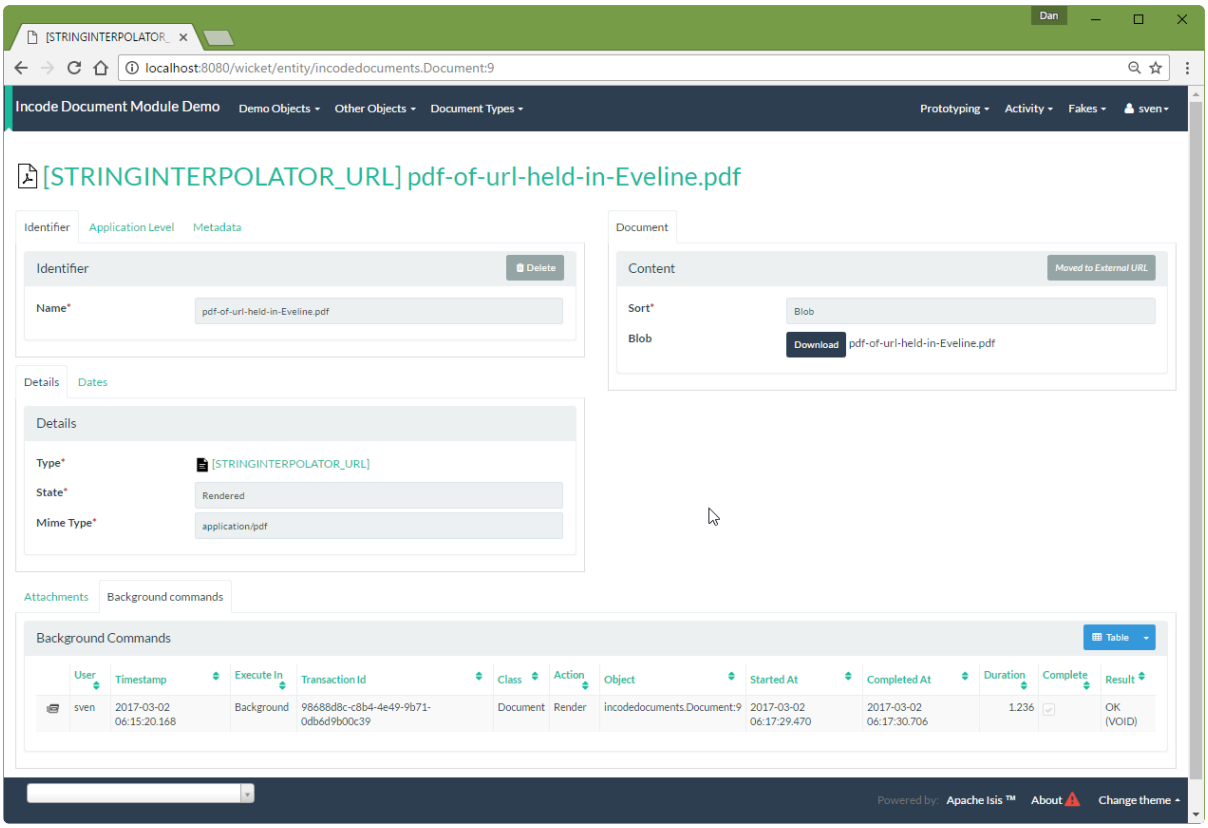
This results in a `Document` with an associated background command. Note that the document, at this stage, has a state of "Not rendered" and it has no content:



The demo app has not been configured with a background scheduler, but does provide a "fake" scheduler which can be used to run such commands:

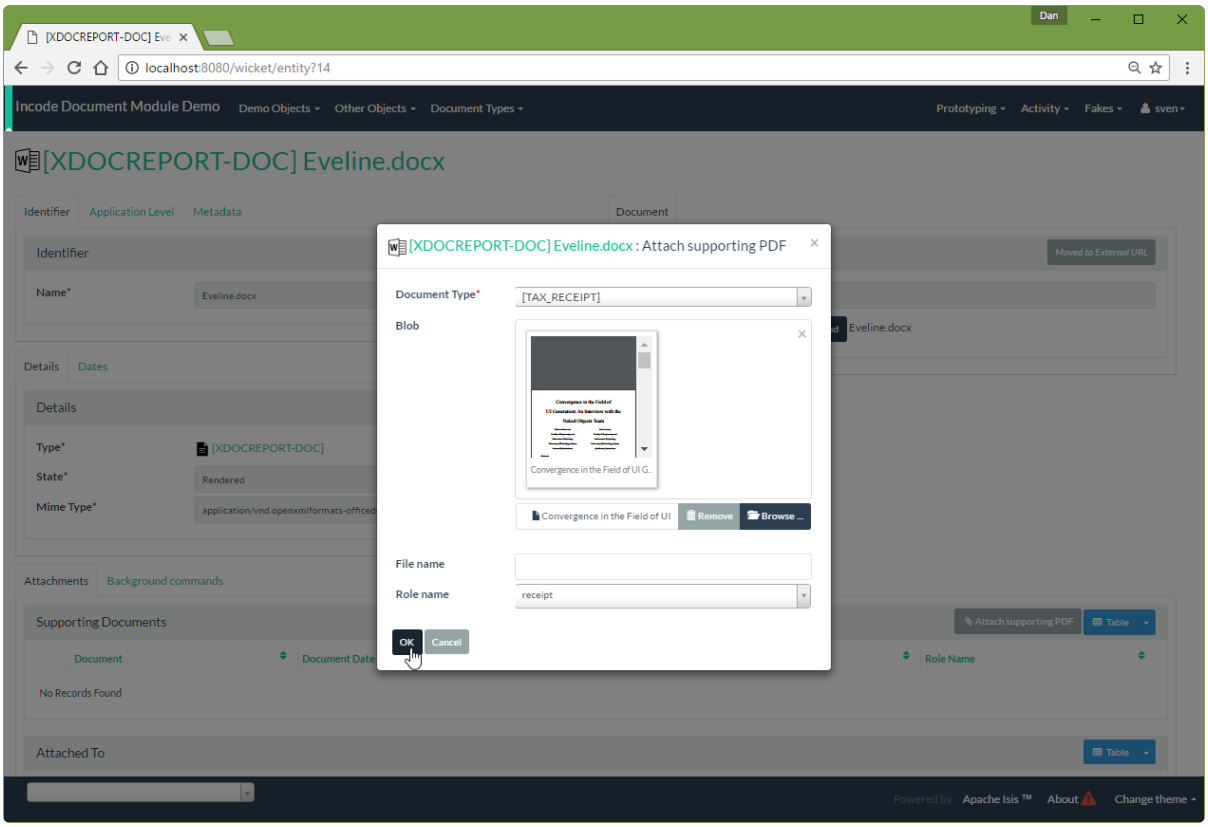


After the background commands have run, the document now has content and its state changes to "Rendered":

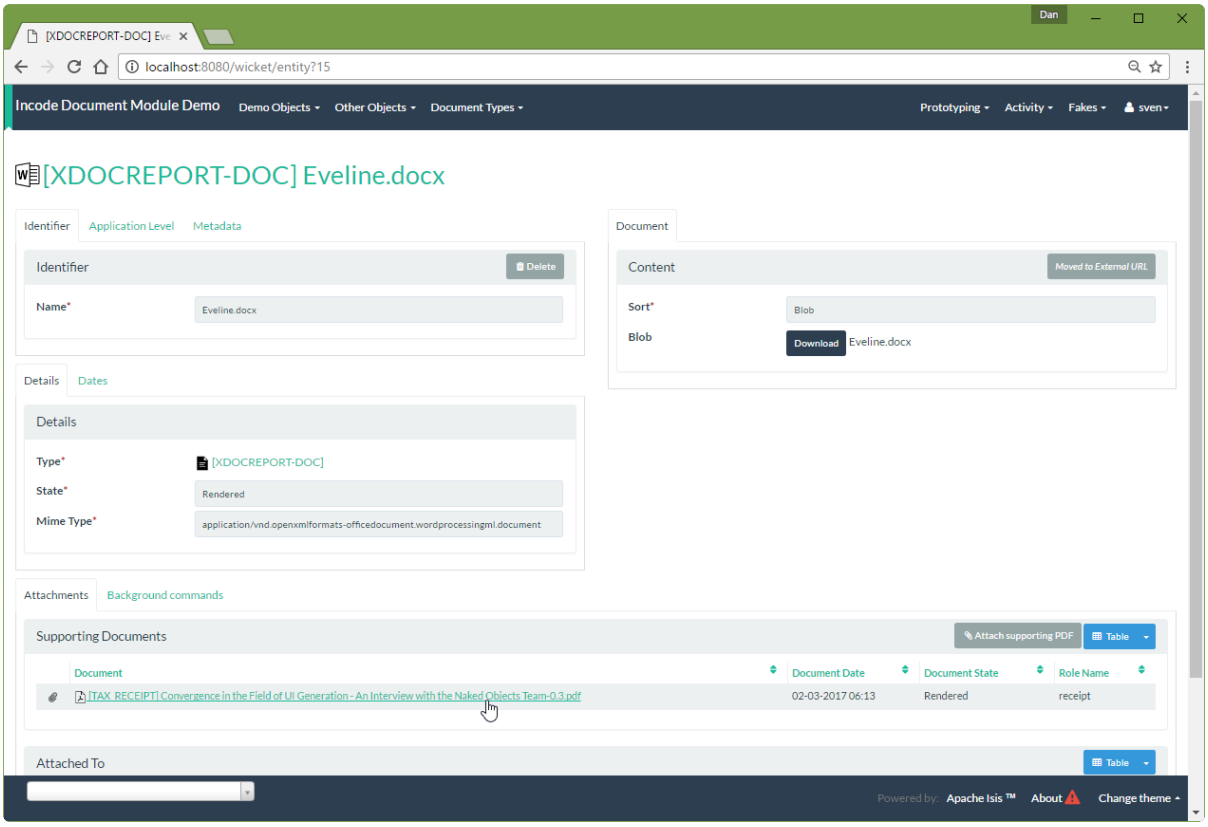


Attaching Supporting Documents

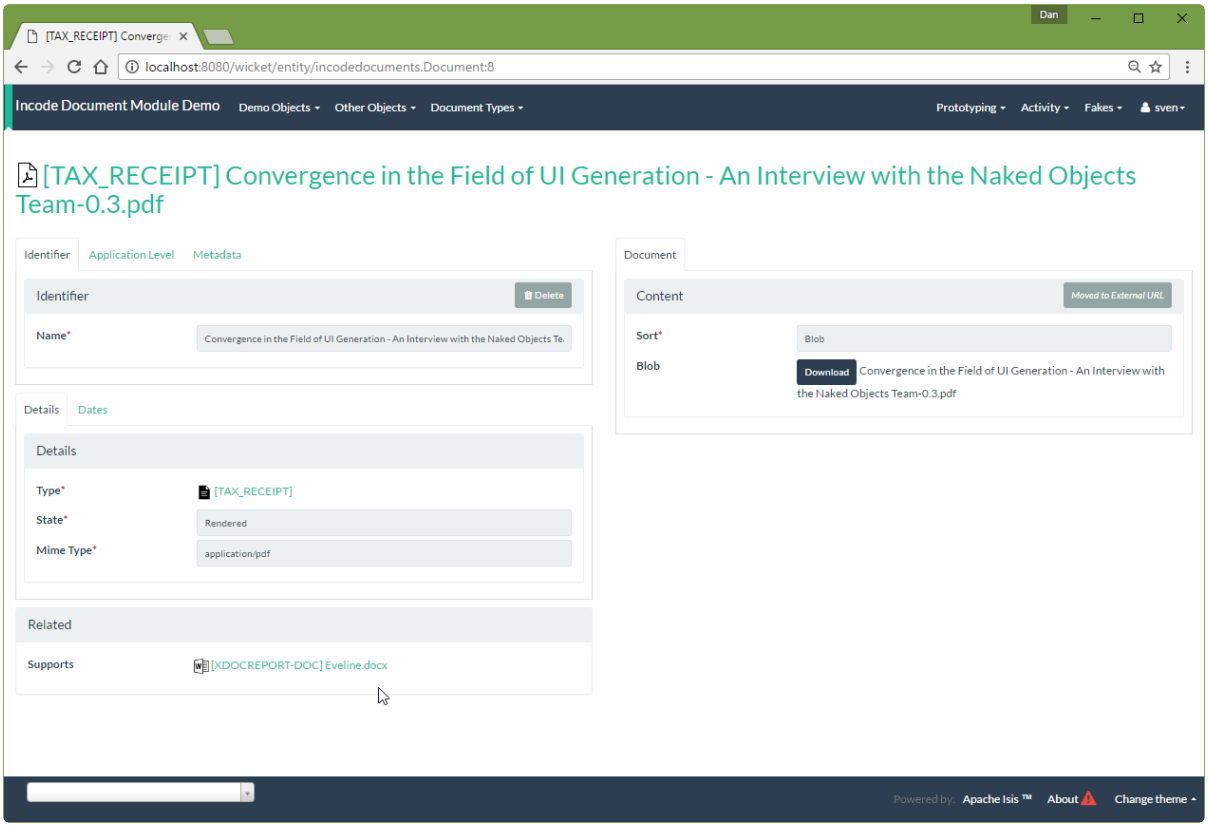
On any (generated) `Document`, it is also possible to attach supporting PDF documents. For example, this could be a tax or supplier receipt. The "attachSupportingPdf" action uses the `DocumentAttachmentAdvisor` SPI service to obtain a list of appropriate `DocumentType`s to display:



The uploaded PDFs are wrapped in a `Document` of their own, and listed in the "supporting documents" collection:



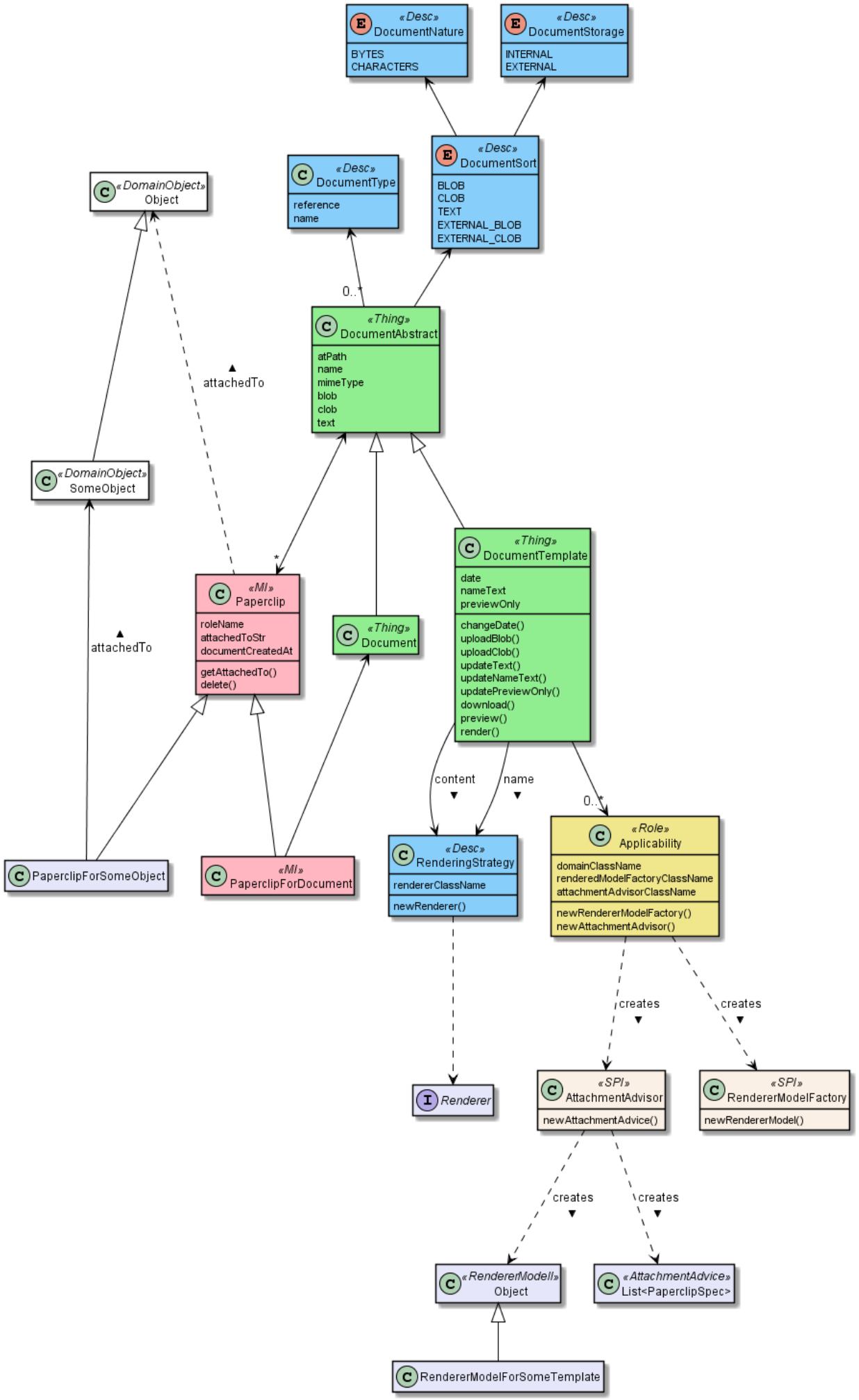
The supporting `Document` itself can also be viewed. The "supports" property refers back to the `Document` that it supports:



Domain Model

Document, DocumentTemplate and Paperclip

The following class diagram highlights the main concepts:


«Desc»
DocumentNature
BYTES
CHARACTERS
«Desc»
DocumentStorage
INTERNAL
EXTERNAL
«Desc»
DocumentSort
BLOB
CLOB
TEXT
EXTERNAL_BLOB
EXTERNAL_CLOB
«Desc»
DocumentType
reference
name
«DomainObject»
Object
0..*
«Thing»
DocumentAbstract
atPath
name
mimeType
blob
clob
text
attachedTo
«DomainObject»
SomeObject
*
«MI»
Paperclip
roleName
attachedToStr
documentCreatedAt
getAttachedTo()
delete()
attachedTo
«Thing»
Document
«Thing»
DocumentTemplate
date
nameText
previewOnly
changeDate()
uploadBlob()
uploadClob()
updateText()
updateNameText()
updatePreviewOnly()
download()
preview()
render()
content
name
0..*
PaperclipForSomeObject
«MI»
PaperclipForDocument
«Desc»
RenderingStrategy
rendererClassName
newRenderer()
«Role»
Applicability
domainClassName
renderedModelFactoryClassName
attachmentAdvisorClassName
newRendererModelFactory()
newAttachmentAdvisor()
creates
creates
Renderer
«SPI»
AttachmentAdvisor
newAttachmentAdvice()
«SPI»
RendererModelFactory
newRendererModel()
creates
creates
«RendererModel»
Object
«AttachmentAdvice»
List<PaperclipSpec>
RendererModelForSomeTemplate

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

The central concept is, of course, Document. Documents have content that is either a Blob, Clob or is text, these attributes being defined in the DocumentAbstract supertype (more on this shortly). Alternatively, the Document's content can be stored externally, eg in a CMS or cloud storage service, in which case the Document's own externalUrl attribute is used. The DocumentSort determines how the content of the Document is physically stored (along with the supporting DocumentNature and DocumentStorage enums). Conceptually Documents are immutable (though if their content is moved to an external URL, the original entity would be update in that case).

Each Document also has a corresponding DocumentType, eg "Invoice" or perhaps a form id, eg "ABC123".

The DocumentTemplate is also a document (ie subclass of DocumentAbstract), however its content will have placeholders. These placeholders are populated with respect to some sort of domain object acting as an input (like a "mail merge"), to generate a resultant Document. The DocumentTemplate also has a DocumentType, and so it is the DocumentType that acts as the link between the DocumentTemplate with the Documents created from those templates. It is possible for there to be multiple DocumentTemplates over time for a particular DocumentType (distinguished by date), to allow for minor changes to a template over time. The domain model deliberately does **not** keep track of which particular DocumentTemplate was used to create a Document, just the type is used.

Each DocumentTemplate has a RenderingStrategy, this being a mechanism to actually produce its content by interpolating the template text with placeholders.

Actually, each DocumentTemplate has two sets of placeholders and also corresponding RenderingStrategys. The "content" template text is used to generate the actual content of the resultant Document's content; this could be characters (eg a HTML email) or bytes (eg a PDF). The "name" template text , while the other is used to interpolate the name of the resultant Document; this will always result in a simple character string.

Each DocumentTemplate also has an associated set of Applicabilitys. Each of these identifies a domain class that can be used as an input the rendering of the DocumentTemplate, with a corresponding implementation of the RendererModelFactory interface being responsible for actually creating an input "renderer model" used to feed into the template's RenderingStrategy. The Applicability also defines the implementation of AttachmentAdvisor interface; this is used to attach the resultant Document to arbitrary domain objects (usually the input domain object, and perhaps others also).

Every Document is created from a DocumentTemplate, but rather than hold a reference to this original template, instead Document and DocumentTemplate are unified through the DocumentType entity. The DocumentType can be considered as a set of versioned DocumentTemplates (identified by date), along with all the Documents that were created from (any of) those DocumentTemplates.

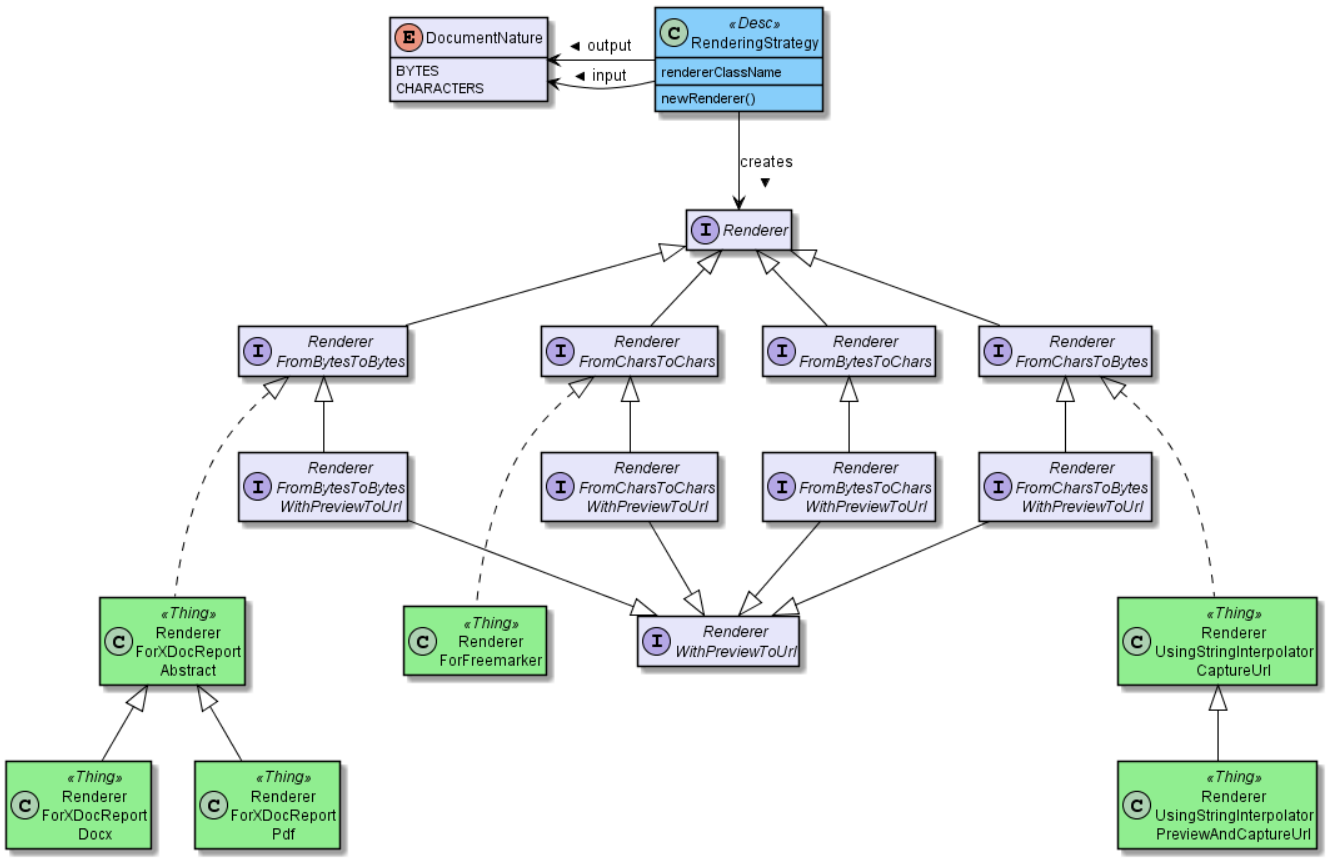
Once a Document has been created it is attached to one or more target domain object using Paperclip. This requires a custom subclass for the domain object in question; the polymorphic pattern ("table of two halves") is used for this linkage. The module uses this capability itself for PaperclipForDocument, which is used to attach supporting (PDF) Documents to generated Documents.

Based upon the implementation of `RenderingStrategy` and `Renderer`, each `DocumentTemplate` can support either previewing and/or rendering. Previewing means to return a representation as a URL; the end-user can then navigate to this URL without any change in state to the application. Rendering on the other hand means the creation and persisting of a `Document` from the `DocumentTemplate`.

The `createAndAttachDocumentAndRender()` mixin is contributed to all domain objects where there is a `DocumentTemplate` available for the domain object's application tenancy path (`atPath`) that supports either previewing and/or rendering. The similar `createAndAttachDocumentAndScheduleRender()` mixin is also available, allowing the rendering to be performed as a background task using command spi module. This can be scheduled using, for example, the quartz extension module.

RenderingStrategy & Renderer

The `Renderer` interface has the following subtypes and (example) implementations:



The owning `RenderingStrategy` for each `Renderer` identifies the nature of the inputs and outputs (bytes or characters) of each `RenderingStrategy`; the associated `Renderer` implementation must meet those constraints. Note that a `Renderer` may produce nature of the inputs vs outputs may vary: a character template might result in byte array output.

External blob/clob storage

When a `Document` is initially generated, it will contain content as either a text string, a clob or as a blob; its `#getSort()` accessor - returning the `DocumentSort` enum - specifies which.

Storing blobs or clobs within a single database table can become unwieldy - backing up the database and performing other DB maintenance activities can start taking significant resources/time. At the same time, the `Document` entity itself is immutable; the blobs/clobs stored within never change once created.

Therefore the `Document` allows for the blob/clob to be moved into an offsite storage, and then to hold the URL to access that blob/clob. Typically this would be performed by some background process that would:

- query for all newly created `Document`s that contain a blob or clob
- copy the blob/clob to some external storage, for example an external document management system running on-premise, or perhaps an off-site Cloud storage. A URL would represent a key to retrieve this blob/clob whenever required
- update the `Document`, updating its `externalUrl` property, and setting its blob/clob to null. It would also update the `Document` so that `#getSort()` accessor indicates that the storage is stored externally.

  The `Document_movedToExternalUrl` mixin action captures these tasks.

The above algorithm is idempotent and so resilient to potential failure.

Once a `Document`'s content has been moved to be stored externally, it can subsequently be retrieved dynamically as required using the `UrlDownloadService` SPI service.

How to configure/use

Classpath

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

```
<dependency>
    <groupId>org.incode.module.document</groupId>
    <artifactId>incode-module-document-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:

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

Input

For each domain object class that you want to use as the input data to a `DocumentTemplate`, you need to:

- implement `ApplicationTenancyService`

  To return the application tenancy path of the domain object in order that available `DocumentTemplate`s can be located:

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

- implement a `RendererModelFactory`

  This constructs the "renderer model" from the input domain object, which is then fed into the

`RenderingStrategy` of the `DocumentTemplate`:

```
public interface RendererModelFactory {
    @Programmatic
    Object newRendererModel(
            DocumentTemplate documentTemplate,    ①
            Object domainObject);                 ②
}
```

① the template to which this implementation applies, as per `DocumentTemplate#getAppliesTo()` and `Applicability#getRendererModelFactoryClassName()`

② provides the input for the renderer model

The `RendererModelFactoryAbstract<T>` can be used to implement the `RendererModelFactory` interface, adding the capability of verifying the input document is of the correct type.

- implement a `AttachmentAdvisor`

  This returns a data structure (`List<PaperclipSpec>`) which describes to which object(s) the resultant `Document` should be attached:

```
public interface AttachmentAdvisor {
    @lombok.Data                                  ①
    public static class PaperclipSpec {
        private final String roleName;
        private final Object attachTo;
        private final Object createdDocument;
    }
    List<PaperclipSpec> advise(
            DocumentTemplate documentTemplate,    ②
            Object domainObject,                  ③
            Document createdDocument);            ④
}
```

① immutable value type, defined using `@Data` annotation from Project Lombok

② to which this implementation applies, as per `DocumentTemplate#getAppliesTo()` and `Applicability#getAttachmentAdvisorClassName()`

③ acting as the context for document created, from which derive the objects to attach the newly created `Document`

④ the document that has been created. Note that this may be `null` when the advisor is being asked if it *could* be used to attach for the domain object.

The `PaperclipSpec` describes how create instances of `Paperclip` from attach the resultant `Document` to other domain objects.

The `AttachmentAdvisorAbstract<T>` can be used to implement the `AttachmentAdvisor` interface, adding the capability of verifying the input document is of the correct type.

Renderers

For each rendering technology, an implementation of `Renderer` is required. A number of such `Rendererer`s have been developed, using Freemarker, XDocReport or just capturing the content of arbitrary URLs (eg as exposed by an external reporting server such as SQL Server Reporting Services).

Paperclips (attach output)

For each domain object that you want to attach `Document`s (that is, add `Paperclip`s to), you need to

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

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

- implement the `PaperclipRepository.SubtypeProvider` SPI interface:

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

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

For example:

```
@javax.jdo.annotations.PersistenceCapable(identityType=IdentityType.DATASTORE)
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
@DomainObject(objectType = "estatioAssets.PaperclipForInvoice")
@DomainObjectLayout( bookmarking = BookmarkPolicy.AS_ROOT)
public class PaperclipForInvoice extends Paperclip {                    ①

    @Column( allowsNull = "false", name = "invoiceId" )
    @Getter @Setter
    private Invoice invoice;

    @NotPersistent
    @Override
    public Object getAttachedTo() {                                      ②
        return getInvoice();
    }
    @Override
    protected void setAttachedTo(final Object object) {
        setInvoice((Invoice) object);
    }

    @DomainService(nature = NatureOfService.DOMAIN)
    public static class SubtypeProvider                                  ③
            extends PaperclipRepository.SubtypeProviderAbstract {
        public SubtypeProvider() {
            super(Invoice.class, PaperclipForInvoice.class);
        }
    }
}
```

① inherit from `Paperclip`

② implement hook methods

③ SubtypeProvider SPI implementation

To view the `Paperclip`s once created there is also a `T_paperclips` mixin collection, discussed below.

Mixins

The document module is fully data-driven, in that the ability to be able to preview and create a document for any given domain entity is defined by the data held in `DocumentTemplate` (its `atPath`) and `Applicability` (the `domainClassName` and corresponding `RendererModelFactory` and `AttachmentAdvisor` implementations).

This is generally done using mixins (though `Document`s can also be created programmatically, see below.

T_preview

The `T_preview` mixin action provides the ability to preview a document without creating it.

To support this the `DocumentTemplate` selected must have a content `RenderingStrategy` that supports previewing to URL.

To use, the mixin simply need to be subclassed. For example:

```
@Mixin
public class Invoice_preview extends T_preview<Invoice> {
    public Invoice_preview(Invoice invoice) { super(invoice); }
}
```

Add similar mixins for all classes where there exists a `DocumentTemplate` and `Applicability` capable of consuming the object as an input to the template, and where (as noted just above) the `DocumentTemplate` has a content `RenderingStrategy` that supports previewing to a URL.

T_createDocumentAndRender, T_createDocumentAndScheduleRender

The `T_createDocumentAndRender` and `T_createDocumentAndScheduleRender` mixin actions provide the ability to create and render a document, either in the foreground or as a background command (so that the rendering can be performed asynchronously).

To use, the mixin simply need to be subclassed. For example:

```
@Mixin
public class Invoice_createDocument extends T_createDocumentAndRender<Invoice> {
    public Invoice_createDocument(Invoice invoice) { super(invoice); }
}
```

Add similar mixins for all classes where there exists a `DocumentTemplate` and `Applicability` capable of consuming the object as an input to the template.

Note that this *doesn't* necessarily require that there is an implementation of `Paperclip` for the target

object: where the generated Document is attached depends upon the definition of the DocumentTemplate.

If you want make this action available for all domain objects, simply use:

```
@Mixin
public class Object_createDocument extends T_createDocumentAndRender
<Object> {
    public Object_createDocument(Object object) { super(object); }
}
```

If there is no DocumentTemplate/Applicability, then the action will be hidden in the UI. The reason that the module doesn't just provide this mixin out-of-the-box is (a) for consistency with other modules and (b) for understandability/traceability ("not **too** much magic").

T_documents

The T_documents mixin collection returns the list of Paperclips that each attach a Document to the specified domain object.

Since Paperclips can only be created for domain objects where a subclass of Paperclip has been defined (see above), it's typical for this mixin to be defined as a nested static class of that Paperclip subclass. For example:

```
...
public class PaperclipForInvoice extends Paperclip {
    ...
    @Mixin
    public static class _documents extends T_documents<Invoice> {
        public _documents(Invoice invoice) {
            super(invoice);
        }
    }
}
```

Document_supportingDocuments, Document_attachSupportingPdf & Document_supports

These three mixins work together.

For generated Documents, the Document_attachSupportingPdf mixin allows PDFs to be associated (eg a supplier receipt), and a corresponding Document is created to hold that PDF. These are then displayed in the Document_supportingDocuments mixin collection.

For the supporting Documents themselves, the Document_supports collection mixin points back to the

associated `Document`. (Sometimes a supporting `Document` might be attached to multiple `Document`s - eg a piece of general correspondence - which is why this is a collection rather than a single property).

The `_supportingDocuments` collection mixin is hidden for "secondary" supporting documents themselves, conversely the `_supports` property is hidden for the "primary" supported documents. In other words these form a parent/child relationship.

Services (API)

DocumentService

The `DocumentService` service allows documents to be created and attached (using `Paperclip`s) programmatically to other domain objects. It also allows existing blobs (PDFs) to be created and optionally attached.

The API is:

```
public class DocumentService {
    public boolean canCreateDocumentAndAttachPaperclips(        ①
            Object domainObject,
            DocumentTemplate template);
    public Document createDocumentAndAttachPaperclips(          ②
            Object domainObject,
            DocumentTemplate template);

    public Document createForBlob(
            DocumentType documentType,
            String documentAtPath,
            String documentName,                                ③
            Blob blob);
    public Document createAndAttachDocumentForBlob(
            DocumentType documentType,
            String documentAtPath,
            String documentName,                                ③
            Blob blob,
            String paperclipRoleName,
            Object paperclipAttachTo);
}
```

① allows a programmatic check as to whether the provided `DocumentTemplate` is applicable to the domain object.

② go ahead and actually create the new `Document`, attaching it as specified by the `AttachmentAdvisor` associated with the `DocumentTemplate` ('s `Applicability` for this domain object).

③ documentName - override the name of the blob (if null, then uses the blob's name)

SPI Services

UrlDownloadService

The `UrlDownloadService` is used to download any `Document`s whose content is stored as an external URL, eg in an on-site CMS or on a cloud storage service.

A default implementation of this service is provided that simply uses Java's `HttpUrlConnection` to download the URL; in particular the URL must be accessible and require no user credentials/passwords.

The service can be optionally overridden if credentials are required.

The service is defined as:

```
public interface UrlDownloadService {
    public Blob downloadAsBlob(Document document) { ... }
    public Clob downloadAsClob(Document document) { ... }
}
```

RendererModelFactoryClassNameService

The `RendererModelFactoryClassNameService`, if implemented, provides UI to allow the renderer model factory class name to be changed on an `Applicability`:

```
public interface RendererModelFactoryClassNameService {
    List<ClassNameViewModel> rendererModelFactoryClassNames();
}
```

This can most conveniently be implemented using the `ClassNameServiceAbstract` convenience class, eg:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class RendererModelFactoryClassNameServiceForDemo
        extends ClassNameServiceAbstract<RendererModelFactory>
        implements RendererModelFactoryClassNameService {
    public RendererModelFactoryClassNameServiceForDemo() {
        super(RendererModelFactory.class, "org.incode.module.document.fixture");
    }
    public List<ClassNameViewModel> rendererModelFactoryClassNames() {
        return this.classNames();
    }
}
```

AttachmentAdvisorClassNameService

The `AttachmentAdvisorClassNameService`, if implemented, provides UI to allow the renderer model factory class name to be changed on an `Applicability`:

```
public interface AttachmentAdvisorClassNameService {
    List<ClassNameViewModel> attachmentAdvisorClassNames();
}
```

Like `RendererModelFactoryClassNameService` (above), this can most conveniently be implemented using the `ClassNameServiceAbstract` convenience class.

RendererClassNameService

The `RendererClassNameService`, if implemented, provides UI to allow the renderer class name to be changed on an `Applicability`:

```
public interface RendererClassNameService {
    public List<ClassNameViewModel> renderClassNamesFor(
            DocumentNature inputNature,
            DocumentNature outputNature);
    <C extends Renderer> Class<C> asClass(String className);
}
```

This can most conveniently be implemented using the `ClassNameServiceAbstract` convenience class, eg:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class RendererClassNameServiceForDemo
         extends ClassNameServiceAbstract<Renderer>
         implements RendererClassNameService {
    public RendererClassNameServiceForDemo() {
        super(Renderer.class, "org.incode.module.document.fixture");
    }
    public List<ClassNameViewModel> renderClassNamesFor(
            final DocumentNature inputNature, final DocumentNature outputNature) {
        if(inputNature == null || outputNature == null){
            return Lists.newArrayList();
        }
        return classNames(x -> inputNature.canActAsInputTo(x) && outputNature
.canActAsOutputTo(x));
    }
    public Class<Renderer> asClass(final String className) {
        return super.asClass(className);
    }
}
```

DocumentAttachmentAdvisor

The DocumentAttachmentAdvisor service, if implemented, is used by the "attachSupportingPdf" action. It allows (existing) PDFs (eg supplier receipts) to be attached to generated Documents.

```
public interface DocumentAttachmentAdvisor {
    public List<DocumentType> documentTypeChoicesFor(Document document);
    public DocumentType documentTypeDefaultFor(Document document);
    public List<String> roleNameChoicesFor(Document document);
    public String roleNameDefaultFor(Document document);
}
```

SupportingDocumentsEvaluator

Some applications may have the concept of a "supporting document", whereby one document is attached to another document and supports it in some way. For example, an application could generates a document for an invoice, and this invoice might have receipts attached to it.

To continue this example, such receipts most likely exist already, for example as PDFs. In such a case the application would directly create the supporting document representing that PDF Blob programmatically:

```
final Document receiptDoc = documentRepository.create(
        documentType, atPath, name, blob.getMimeType().getBaseType());
receiptDoc.setRenderedAt(clockService.nowAsDateTime());
receiptDoc.setState(DocumentState.RENDERED);
receiptDoc.setSort(DocumentSort.BLOB);
receiptDoc.setBlobBytes(blob.getBytes());
```

For such supporting documents much of the state normally associated with a Document should be suppressed:

- a supporting document are probably not generated, so the "backgroundCommands" is not required.
- the "attachedTo" collection does not make sense either, to avoid chains of Documents (one attached to another, attached to another).

On the other hand we might want to explicitly identify that one document supports another, and so the "supportedBy" and "supports" collections help reinforce the semantics of the relatinoship.

The SupportingDocumentsEvaluator is a SPI to tell the document module that such-and-such a Document is a supporting document:

```
public interface SupportingDocumentsEvaluator {
    enum Evaluation {
        SUPPORTING,
        NOT_SUPPORTING,
        UNKNOWN
    }
    Evaluation evaluate(Document candidateSupportingDocument);
    List<Document> supportedBy(Document candidateSupportingDocument);
}
```

The module provides a default implementation that will indicate a `Document` is supporting if it can find any other `Document` that attaches to the candidate document. This implementation can be suppressed if necessary in the usual fashion of a higher priority implementation returning a definitive `Evaluation` one way or the other for the document in question.

Internal Services

These are services that are not part of the formal API/SPI, but nevertheless allow the behaviour of the module to be overridden/fine-tuned.

DocumentTemplateForAtPathService

The `DocumentTemplateForAtPathService` service is used to return the choices for `DocumentTemplate`s for the "preview" and "createAndAttach" mixins.

```
@DomainService(nature = NatureOfService.DOMAIN)
public class DocumentTemplateForAtPathService {
    public List<DocumentTemplate> documentTemplatesForPreview(
                                        Object domainObject) { ... }
    public List<DocumentTemplate> documentTemplatesForCreateAndAttach(
                                        Object domainObject) { ... }
}
```

The default implementation of this service uses the `ApplicationTenancyService` to determine the application tenancy of the supplied domain object, and from that looks up the appropriate (possibly localized) template to use.

However, the "ForAtPath" bit of the name of this service is a mistake, because the service could in fact use any any attributes of the provided domain object to determine the list of `DocumentTemplate`s to make available.

Known issues

When using with PostgreSQL or MsSQL server you are likely to run into data-type issues with the mapping of jdbc-type `BLOB` and/or `CLOB`. By using `.orm`-files we can override the mapping. To activate use setting `isis.persistor.datanucleus.impl.datanucleus.Mapping=xxx` in `persistor_datanucleus.properties`. Thus, setting to `postgres` will activate `DocumentAbstract-postgres.orm` and to `sqlserver` `DocumentAbstract-sqlserver.orm` by naming convention.

Dependencies

Maven can report modules dependencies using:

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

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

From the Incode Platform it uses:

- base library module
- command spi module

The module also uses icons from icons8.