

Communications Subdomain

Table of Contents

Screenshots	2
Sending an Email	3
Sending a Postal Communication	9
Supporting Documents	11
Domain Model	16
(CommunicationChannel) Contributions	19
(Document) Mixins	20
Document_sendByEmail & Document_sendByPost	20
Document_communications	20
Document_communicationAttachments	20
Document_coverNoteFor	20
Services (API)	21
Services (SPI)	22
DocumentCommunicationSupport (required)	22
CurrentUserEmailAddressProvider	23
How to configure/use	24
Classpath	24
Bootstrapping	24
Known issues	25
Dependencies	26

This module (`incode-module-communications`) defines `CommunicationChannels` (email address, postal address or phone/fax number), and also allows `Documents` (implemented by the `Document subdomain` module) to be sent as `Communications` either to an email address or to a postal address.

A `Document` is an entity that holds some sort of content, either binary such as PDF or Word document or text such as HTML; they can be rendered in various ways. `Documents` can be attached to arbitrary domain objects using `Paperclips`.

A `Communication` is an entity that is a record of sending a `Document` to some party by way of their `CommunicationChannel`; these are the "correspondents" of the `Communication`. A `Communication` can be created either by sending the `Document` by email (ie to an email address), or by post (ie to a postal address). The module does not currently provided any capability to "send" a `Document` by phone or by fax.

Email `Communications` always have an HTML cover note `Document` generated and associated with the `Communication`; the original `Document` is also associated. This cover note is used as the body of the actual email, while the original `Document` is included as an email attachment. If the original `Document` had supporting `Documents` associated with it (eg a tax or supplier receipt for an invoice), then these supporting documents are also included as attachments.

Postal `Communications` do *not* have any sort of cover note, and the act of sending them is manual: the end-user downloads the original `Document` "through" the `Communication`; this marks the `Communication` as having been sent. If the original `Document` had supporting `Documents` associated with it then (as a convenience) a single PDF is downloaded that merges together the original `Document` along with any supporting documents.



There is some overlap between this module and the `CommunicationChannel subdomain` module). At some stage we intend to refactor this module to reuse the `CommunicationChannel` module, and then to remove any duplicate concepts (`EmailAddress` etc).

Also, note that the discriminators for `CommunicationChannel` subtypes currently are hard-coded to those for `Estatio`. These can be overridden using `DataNucleus .orm` files if required.

Screenshots

The module's functionality can be explored by running the [quickstart with example usage](#) using the `org.incode.domainapp.example.app.modules.ExampleDomDomCommunicationsAppManifest`.

This creates:

- demo customers, with a variety of `CommunicationChannels`.
- demo invoices, each of which has a `Document` that has an associated document (simulating it having been generated already). Each invoice is also associated with ("owned" by) a customer
- prerequisite reference data to generate email cover notes:
 - A `DocumentType` for the email cover notes.

This is referenced by the demo app's implementation of the `DocumentCommunicationSupport` SPI service

- A corresponding `DocumentTemplate` for email cover notes
- The Freemarker rendering strategy (from the [Freemarker DocRendering library](#) module). This is referenced by the `DocumentTemplate`, instructing the document module to render the email cover note using freemarker.
- `Country` ref data (from the [Country subdomain](#) module).

This is required because `PostalAddress` communication channels reference the `Country`.

The home page lists the demo customers and invoices:

The screenshot shows the Incode App Home page in a web browser. The page has a dark blue header with the Incode App logo and navigation links. Below the header, there are three buttons: "Clear Hints", "Download Layout.Xml", and "Rebuild Metamodel". The main content area is divided into two sections: "Demo Invoices" and "Demo Objects With Notes".

Demo Invoices

Customer	Num
Fred HasEmailAndPhone	1
Fred HasEmailAndPhone	2
Mary HasPhoneAndPost	1
Mary HasPhoneAndPost	2
Joe HasPostAndEmail	1
Joe HasPostAndEmail	2

Demo Objects With Notes

Name	Email Address	Phone Numbers
Fred HasEmailAndPhone	fred@gmail.com fred@msn.com	555 1234
Mary HasPhoneAndPost		777 0987
Joe HasPostAndEmail	joe@yahoo.com joey@friends.com	

The footer of the page indicates it is powered by Apache Isis and includes links for "About" and "Change theme".



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).

Sending an Email

If we inspect one of the invoices for "Fred" (who has email addresses), we see it has an attached **Document** (simulating it having been generated from the invoice):

Invoice #1

localhost:8080/wicket/entity/incodecommunicationsdemo.DemoInvoice:0

Communications Demo App Demo Customers ▾ Prototyping ▾ Activity ▾ Fakes ▾ sven ▾

Invoice #1 for Fred HasEmailAndPhone

General Metadata

General

Customer* Fred HasEmailAndPhone

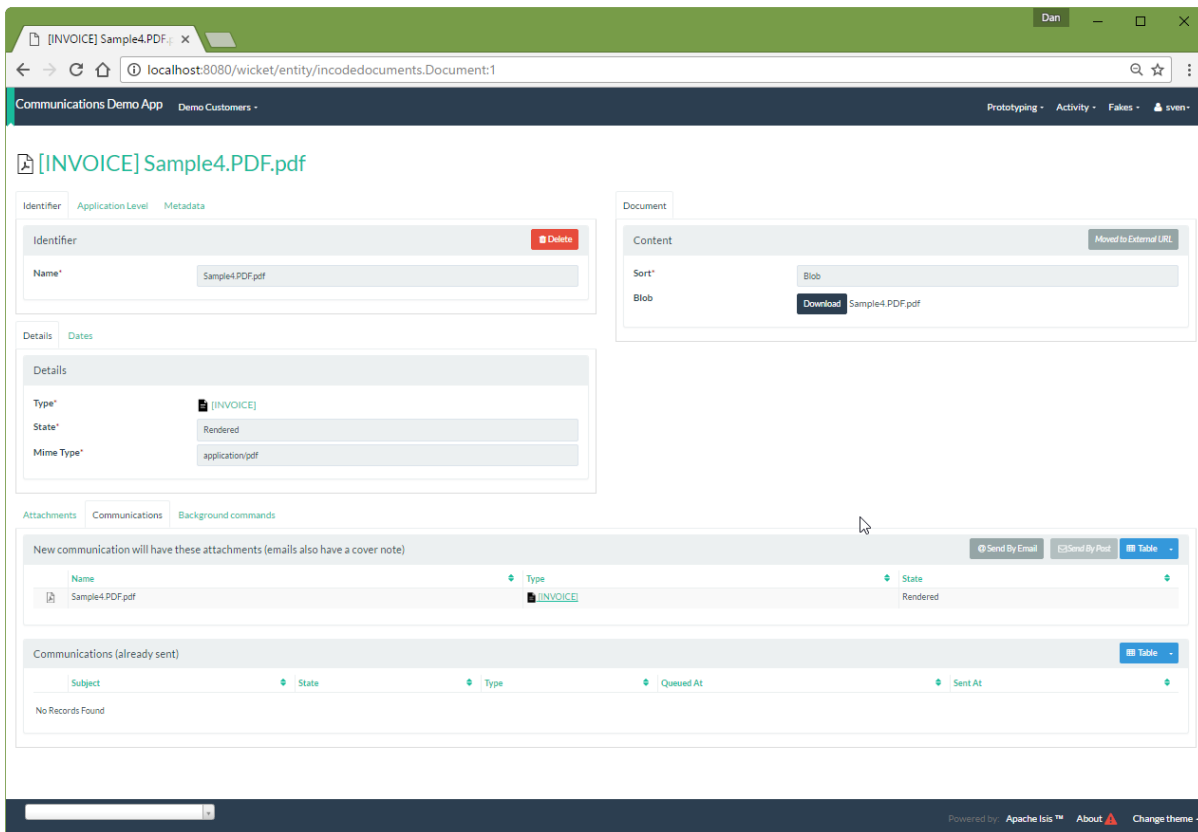
Num* 1

Documents

Document	Document Date	Document State	Role Name
[INVOICE] Sample4.PDF.pdf	02-03-2017 06:59	Rendered	

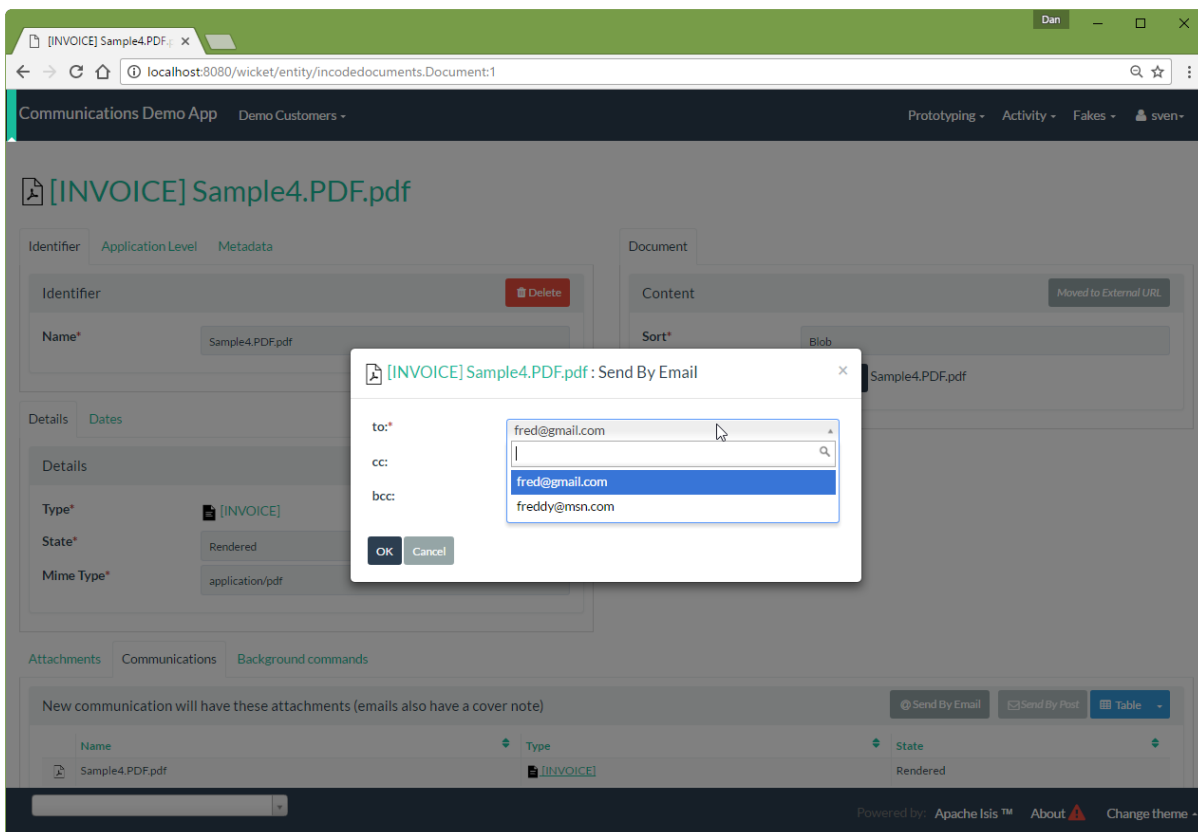
Powered by: Apache Isis™ About ⚠ Change theme ▾

If we inspect that **Document** in turn, we can see that the "send by email" action is enabled:

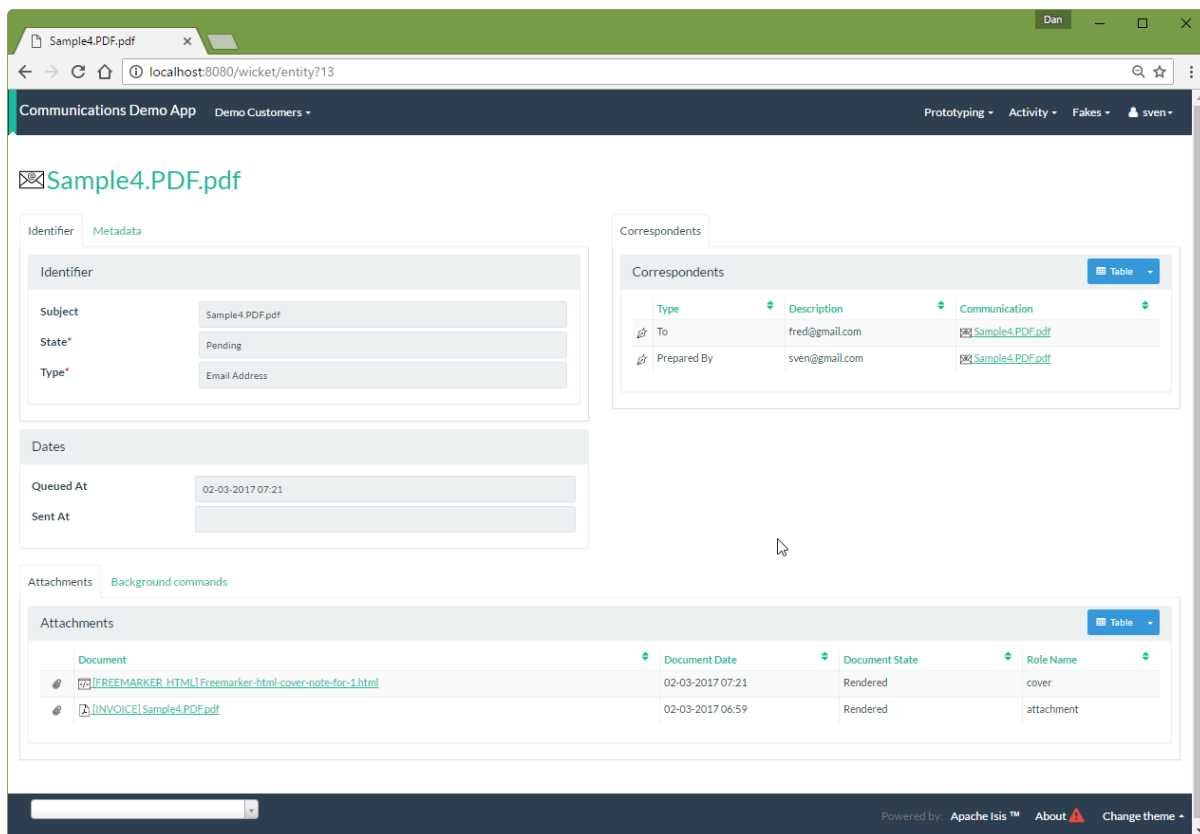


This is because the demo app's implementation of the `DocumentCommunicationSupport` SPI service was able to figure out an email address to use (the document's invoice's customer).

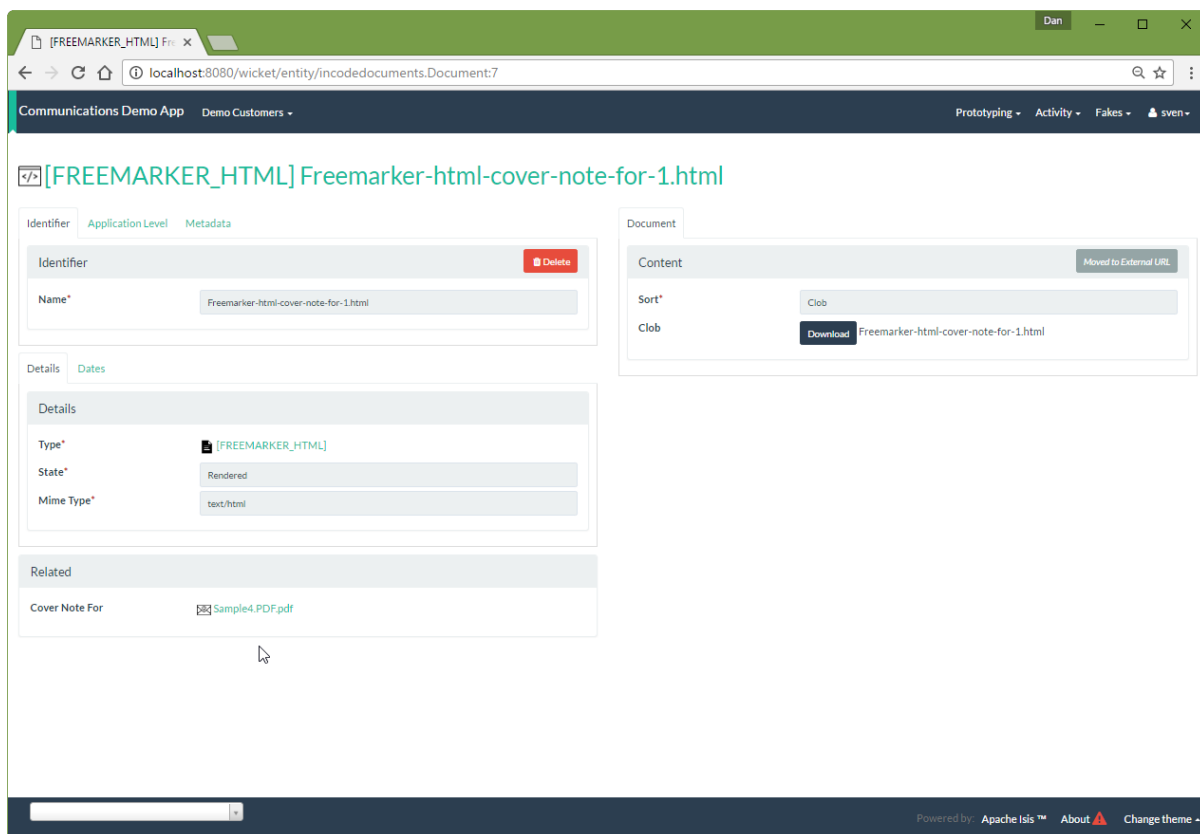
The "sendByEmail" action prompt shows these emails:



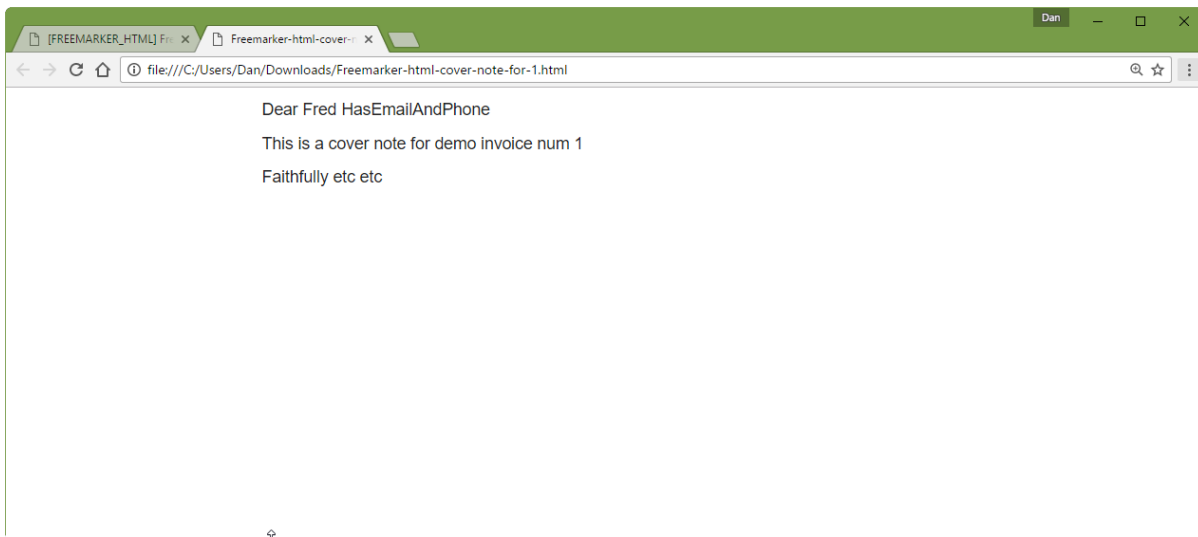
Invoking the action results in an email `Communication`:



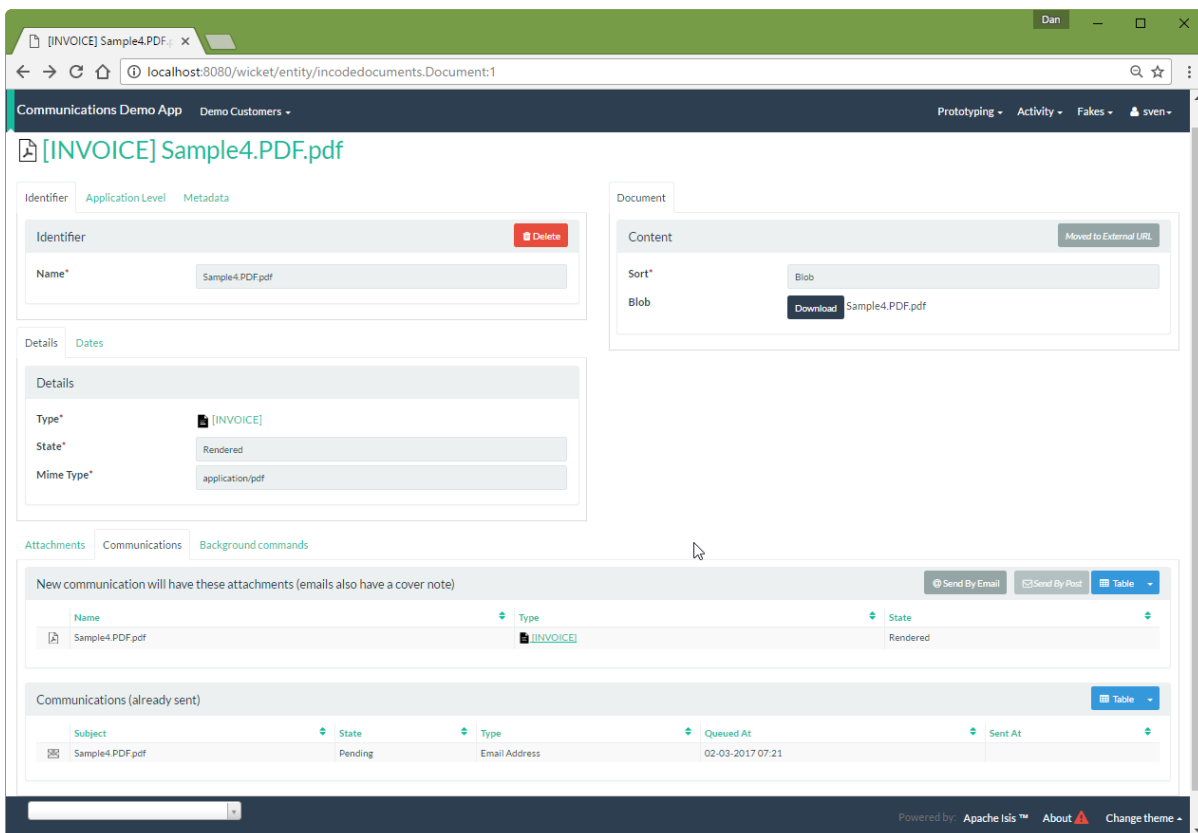
When an email **Communication** is created, it also automatically create a cover note **Document**. The cover note is used as the body of the email, while the original 'Document' is sent as an attachment. The cover note **Document** is automatically associated with the **Communication**, shown by the "coverNoteFor" property:



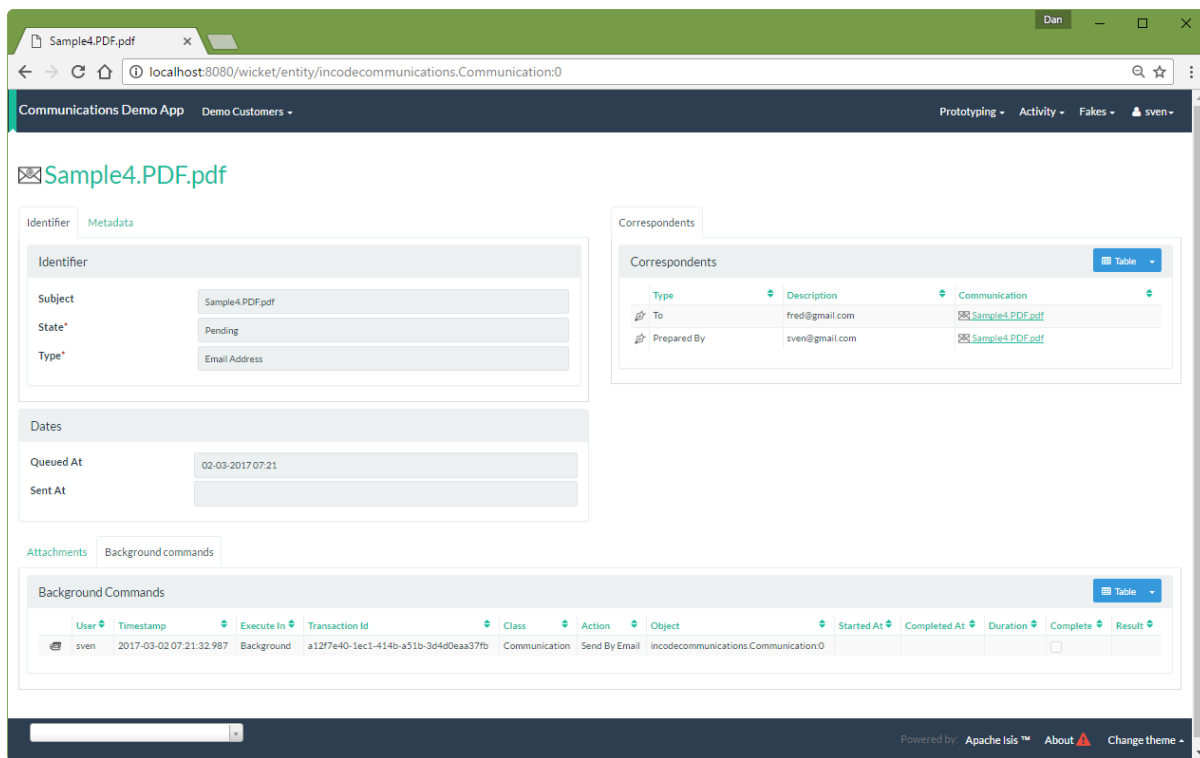
The generated cover note is required to be HTML (so that it can be used as the body of the email). In the case of the demo app this cover note is generated using Freemarker:



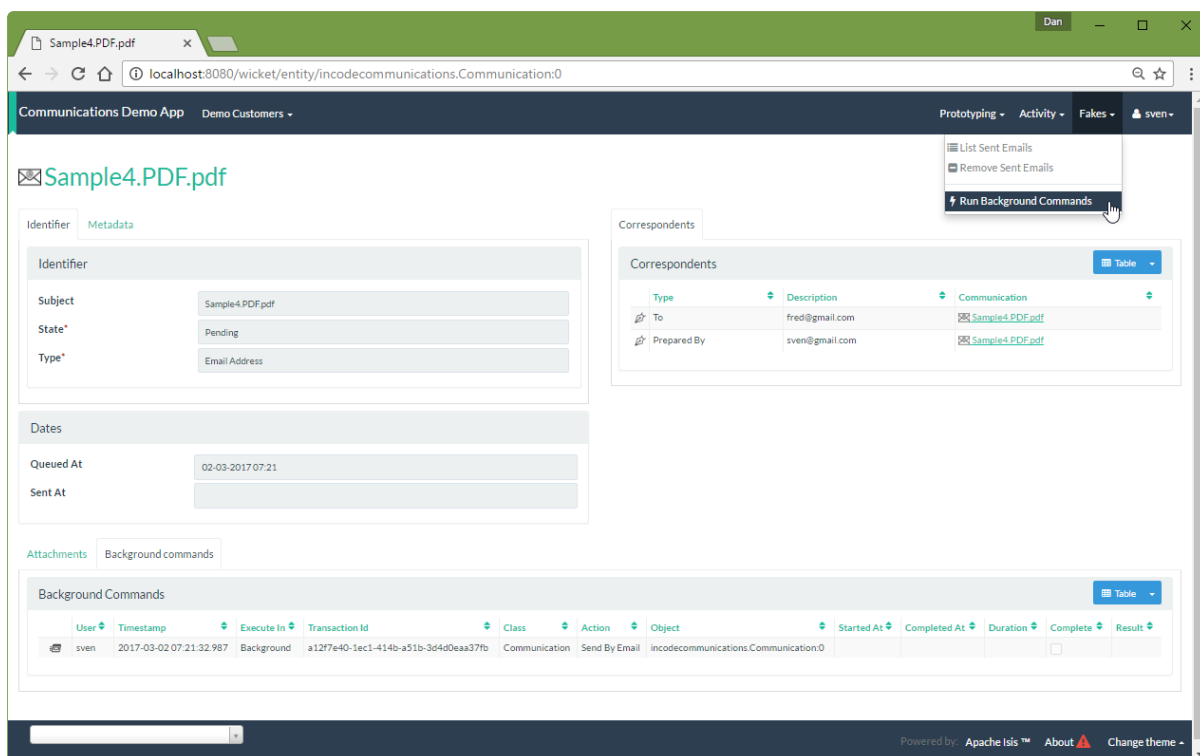
The generated **Communication** is also associated with the original **Document**:



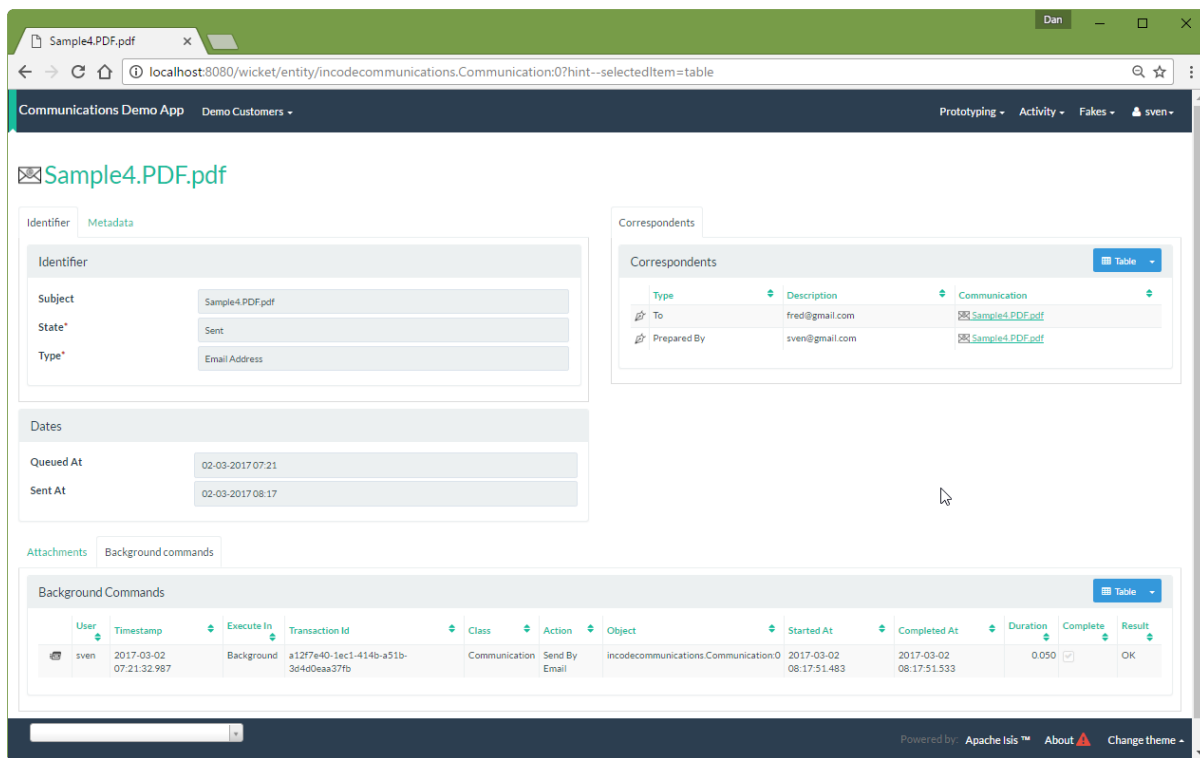
Looking again at the generated email **Communication**, we see that it is scheduled to be sent in the background command:



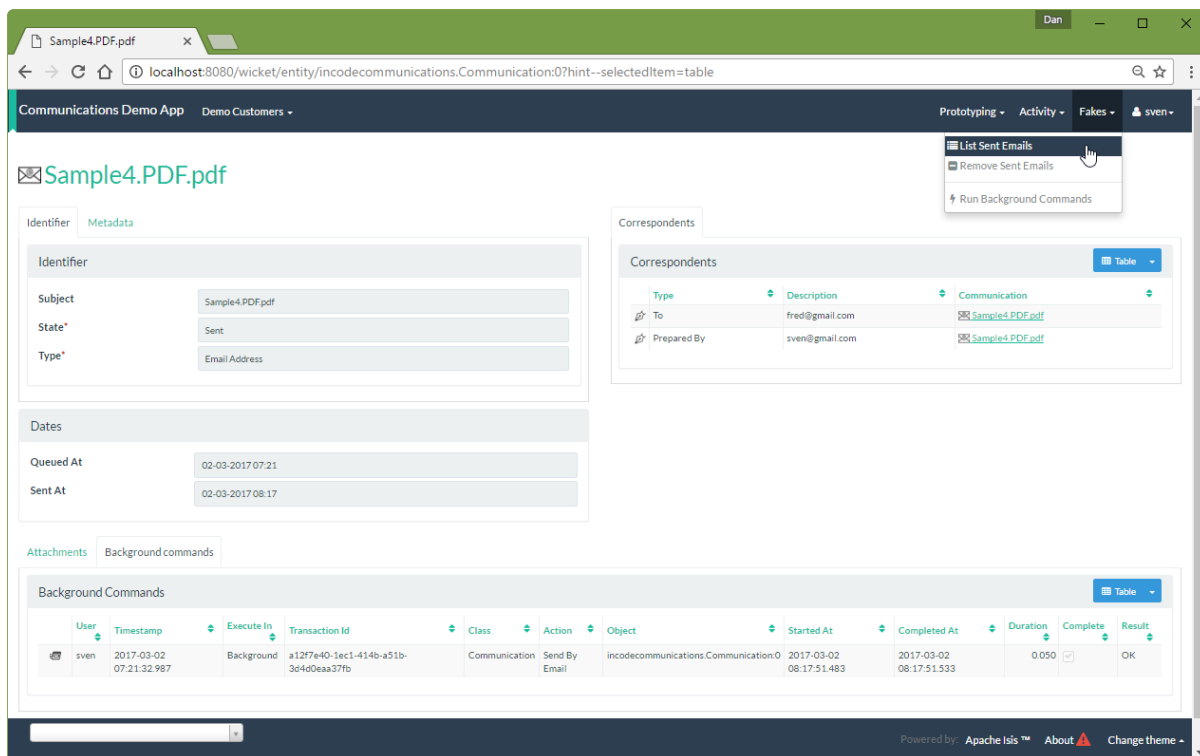
The demo app has not been configured with a background scheduler, but does provide a "fake" scheduler which can be used to run such commands:



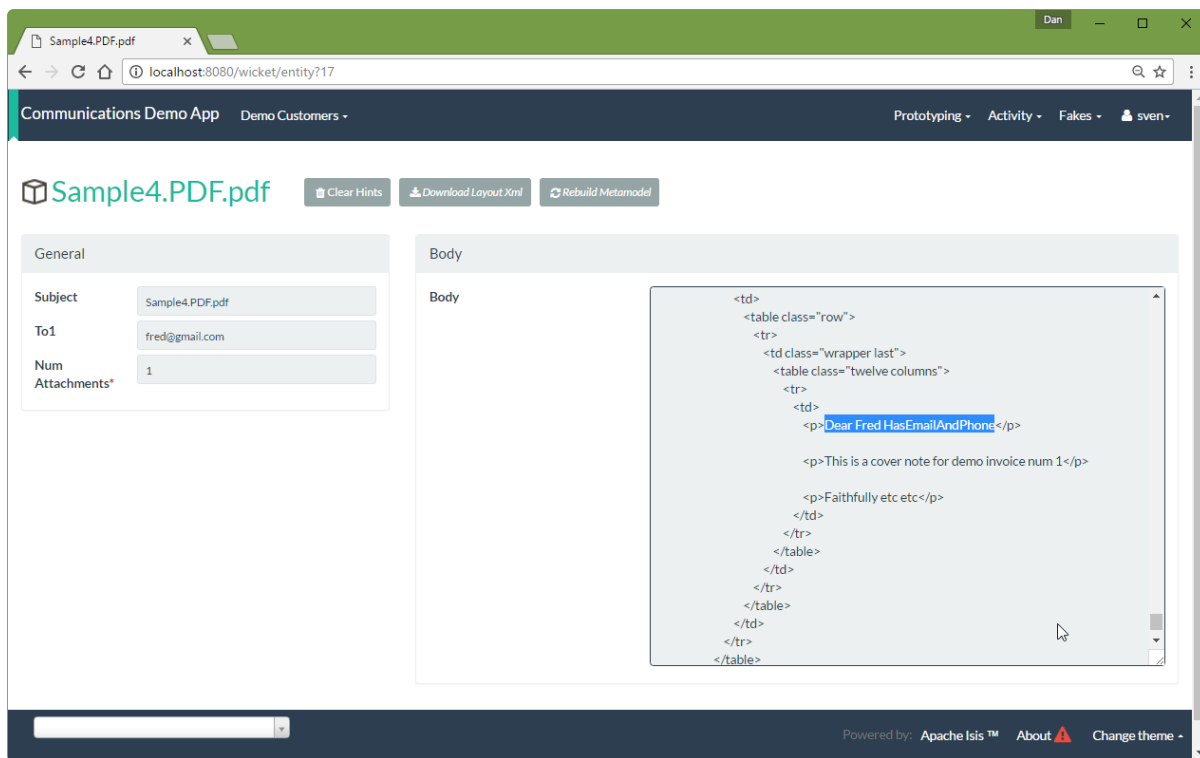
Which results in the email being sent:



In fact, the demo app is not configured with a real email service either; instead it has a fake service that allows "sent" email messages to be inspected:

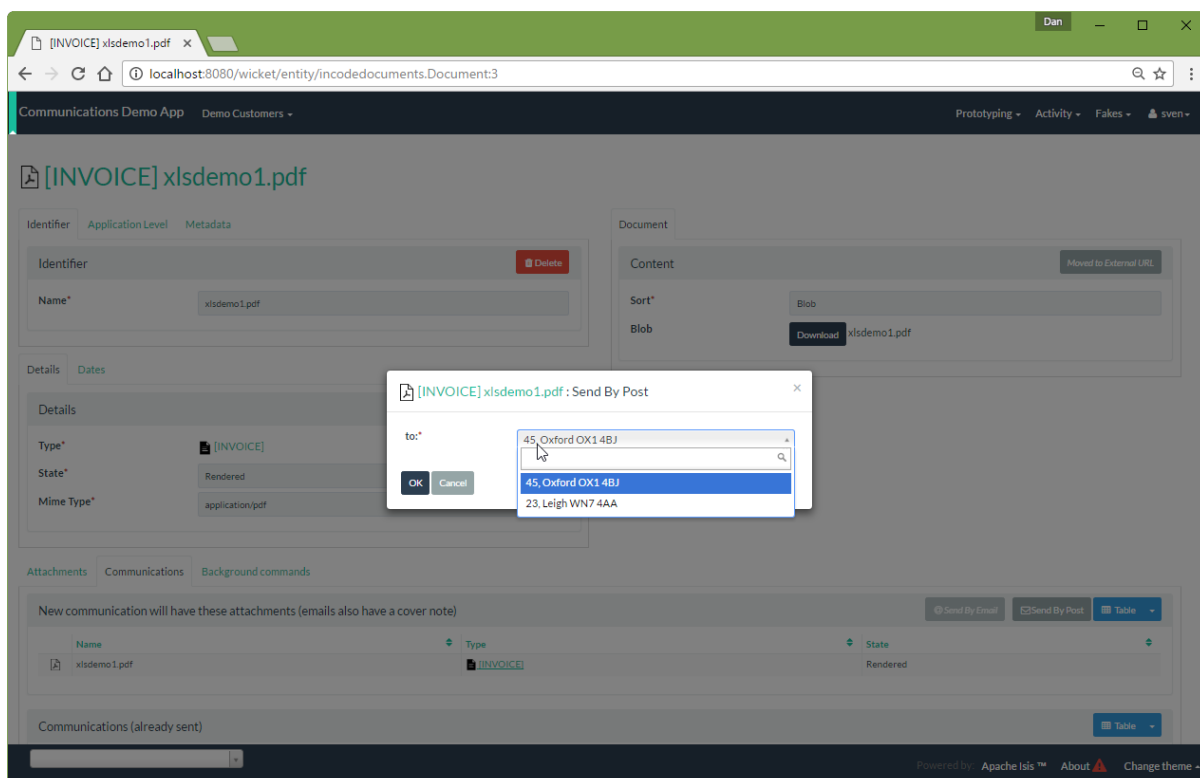


The sent email has the correct body, and one attachment (the original **Document**):

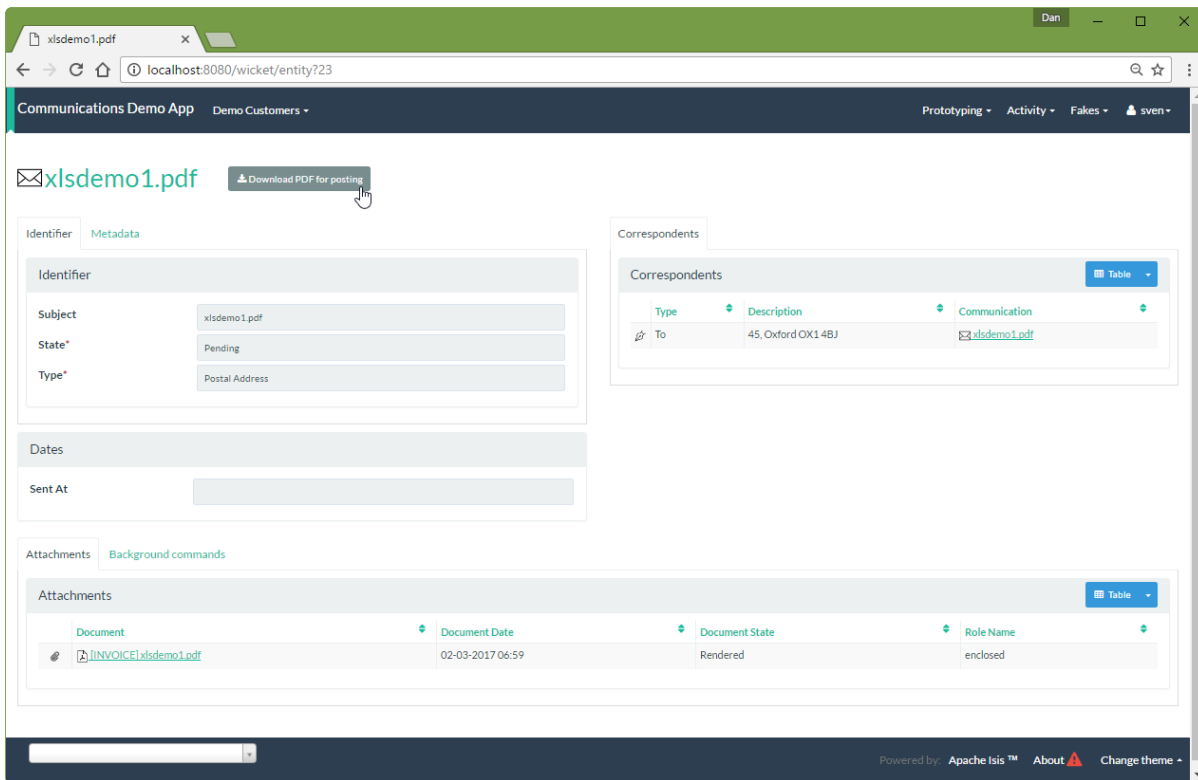


Sending a Postal Communication

The "Mary" demo customer has postal addresses, so the **Documents** attached to her invoices can be sent by post.

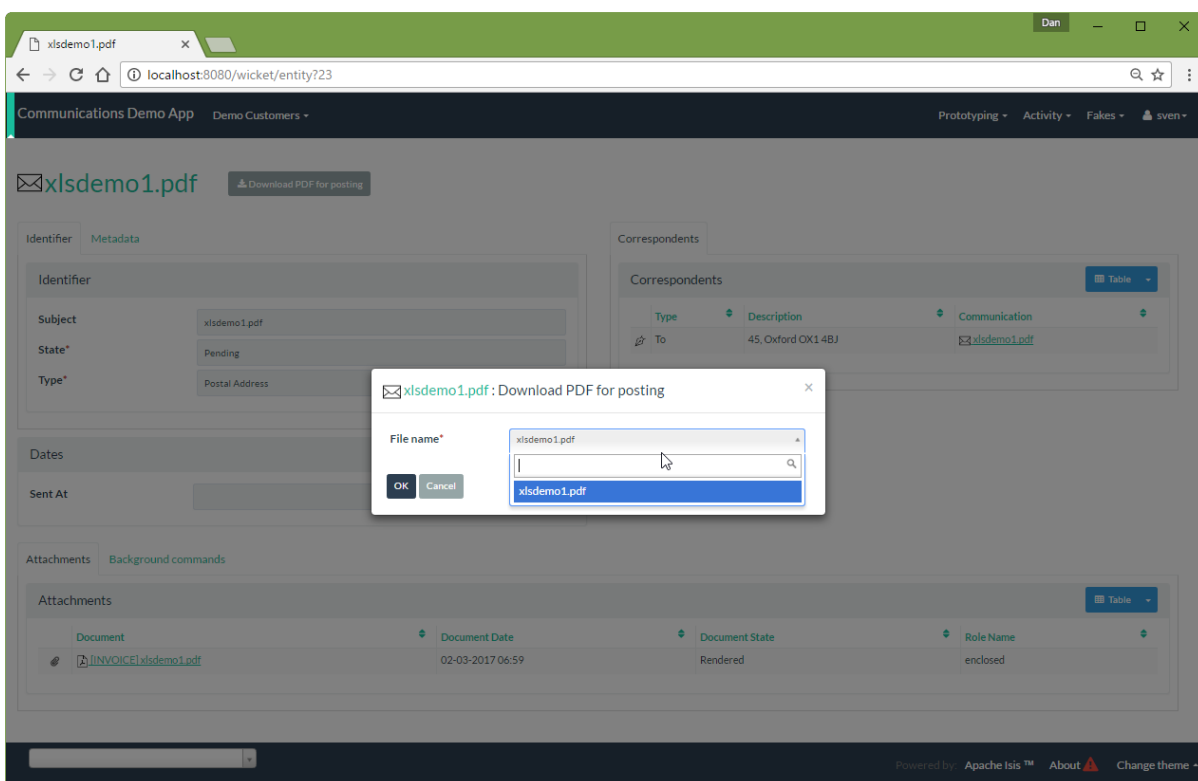


As for email, this also results in a **Communication**:

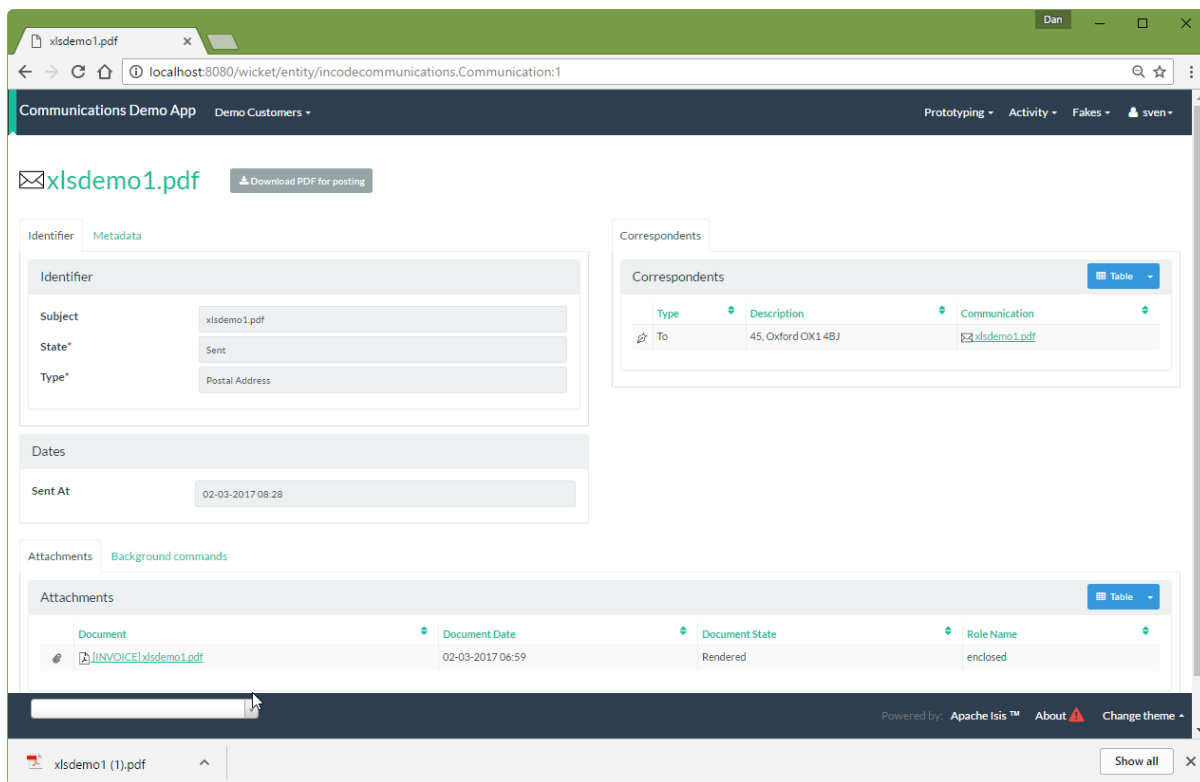


What's different here is that there is no cover note, there is no "prepared by" correspondent, and there is no background command.

Instead, the object provides the "download PDF for posting" action:



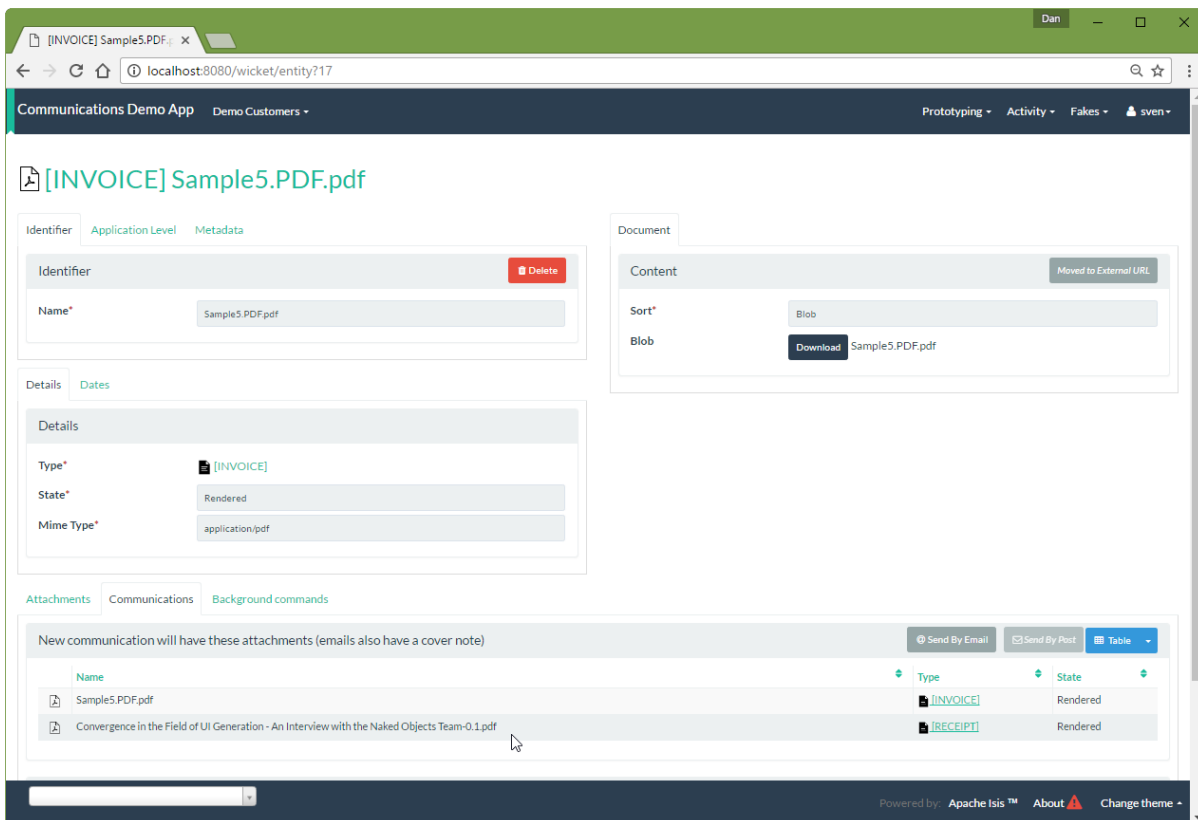
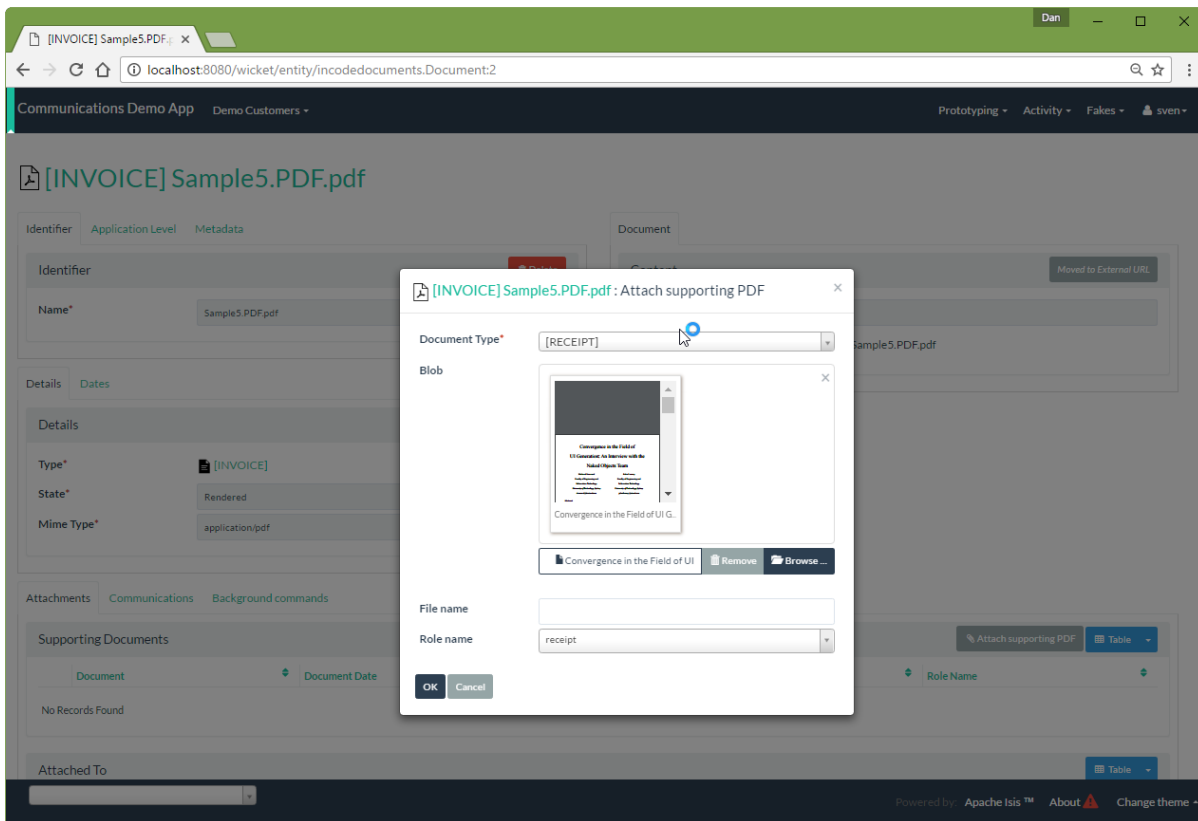
The idea is that (through the **Communication**) the user just downloads the original (PDF) **Document** that it references; the act of doing this marks the **Communication** as sent:

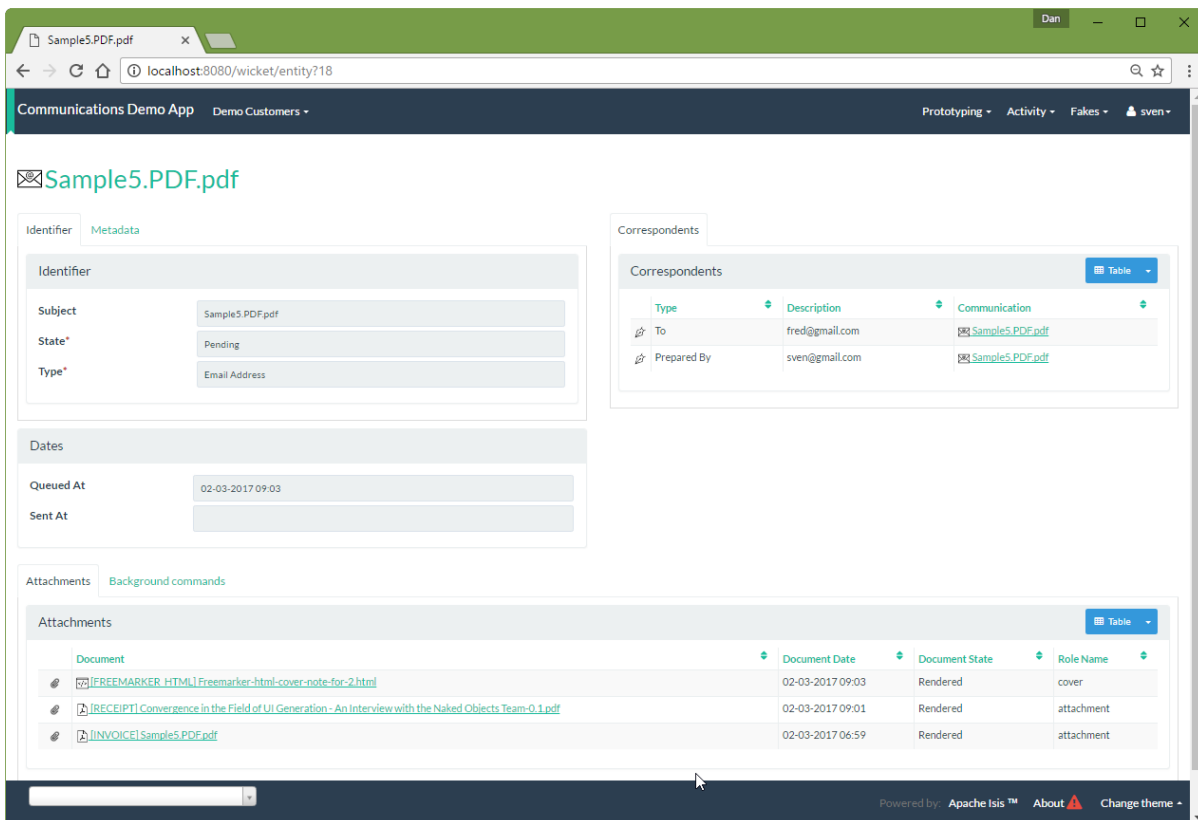
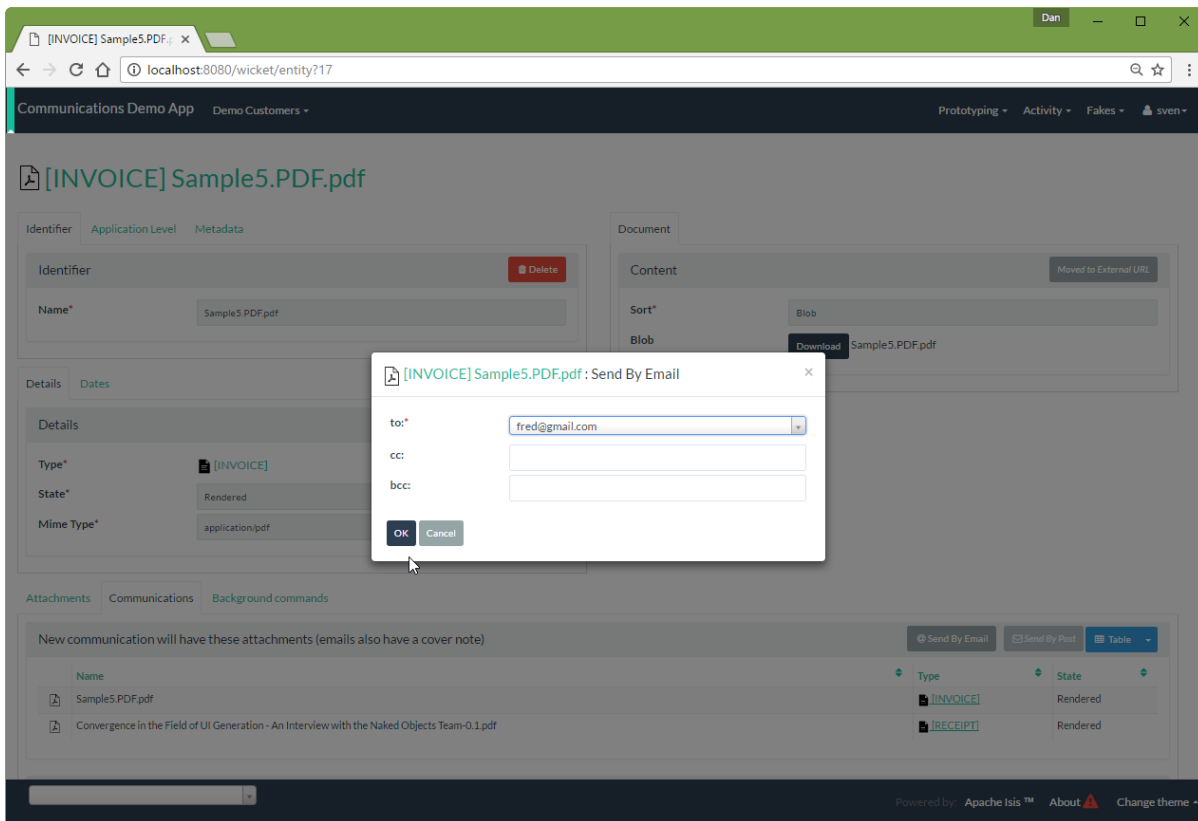


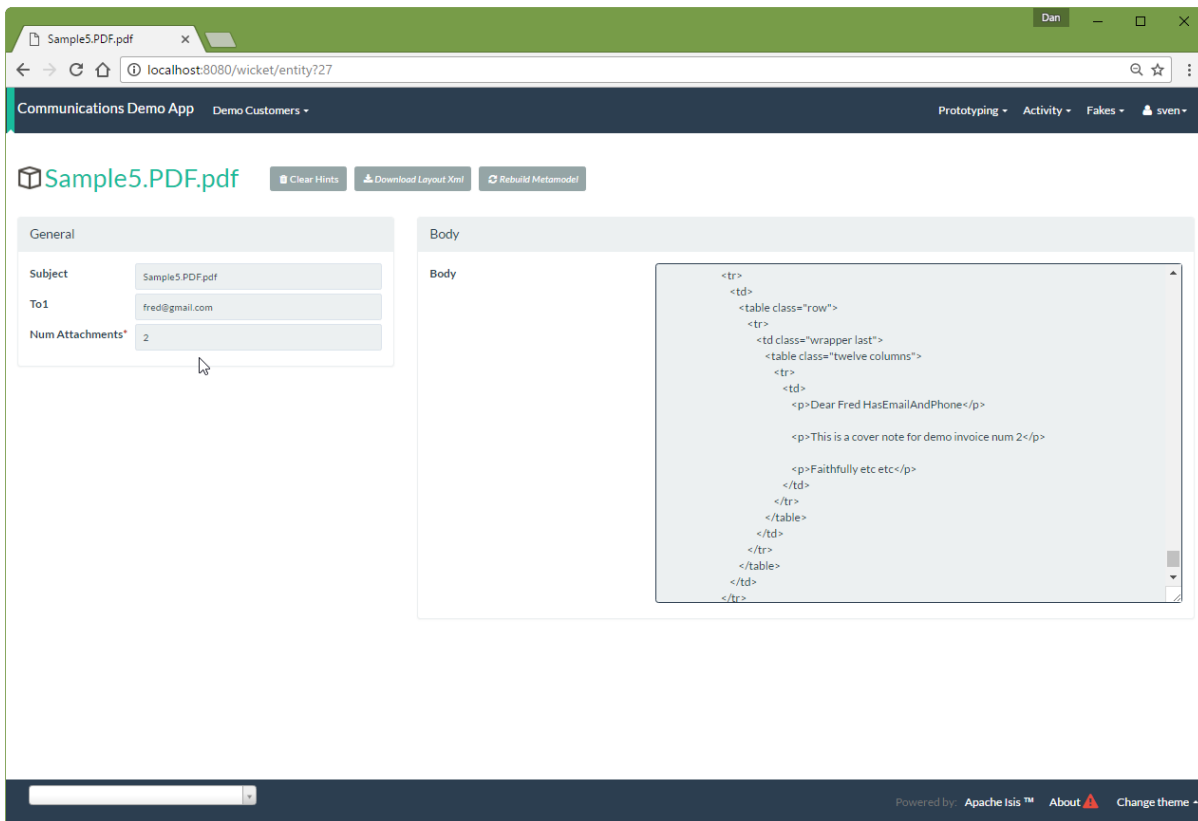
The user can then open up the downloaded PDF, manually print it and manually put it into an envelope.

Supporting Documents

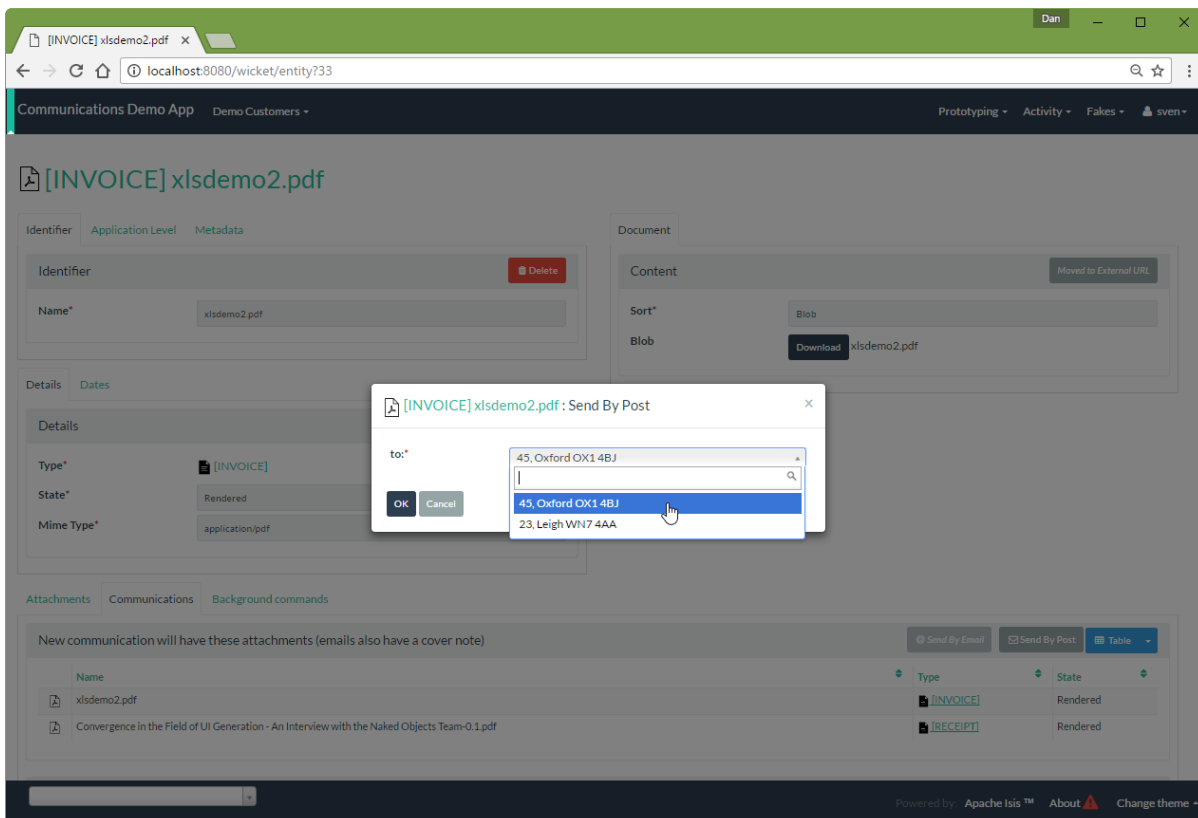
The [Document subdomain](#) module (on which this communications module) depends has the concept of "supporting" documents. For example, a generated **Document** of an invoice might have associated tax or supplier receipts which have been previously scanned in and which are available as PDFs.



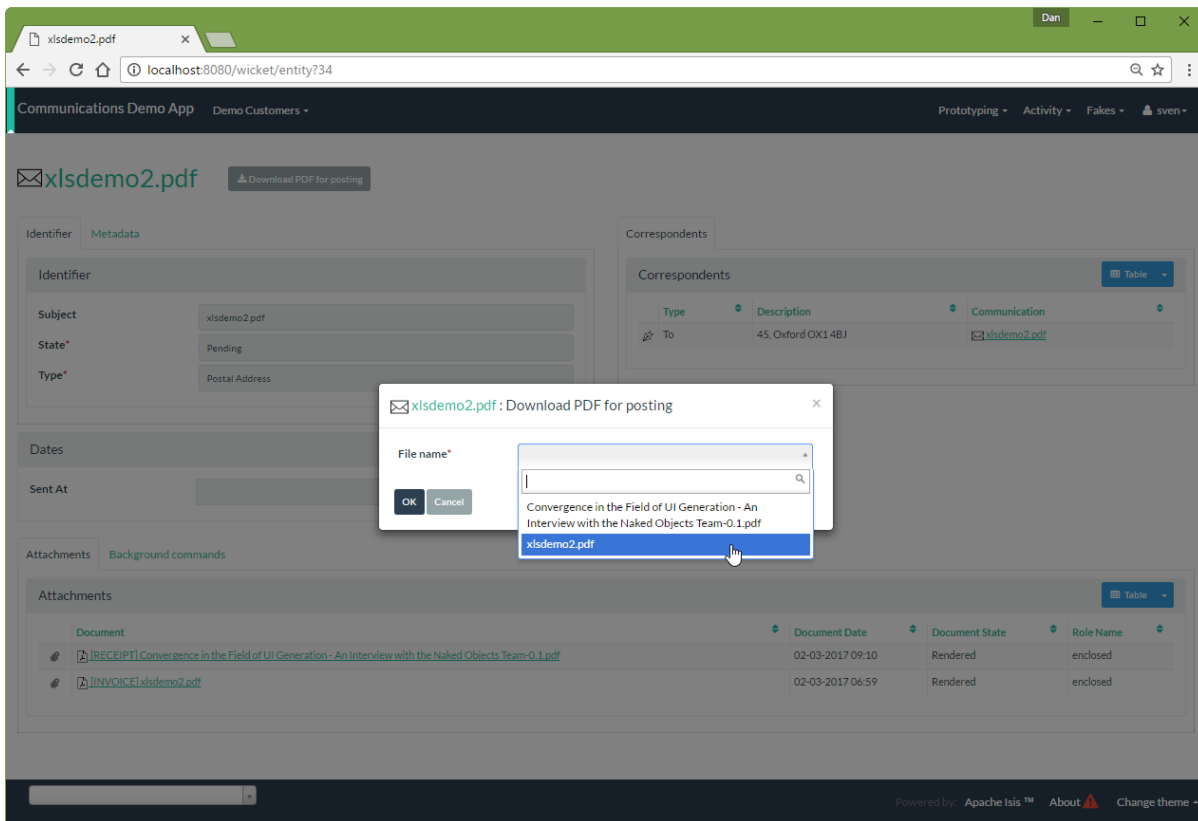




It is also possible to send a postal communication with supporting documents:

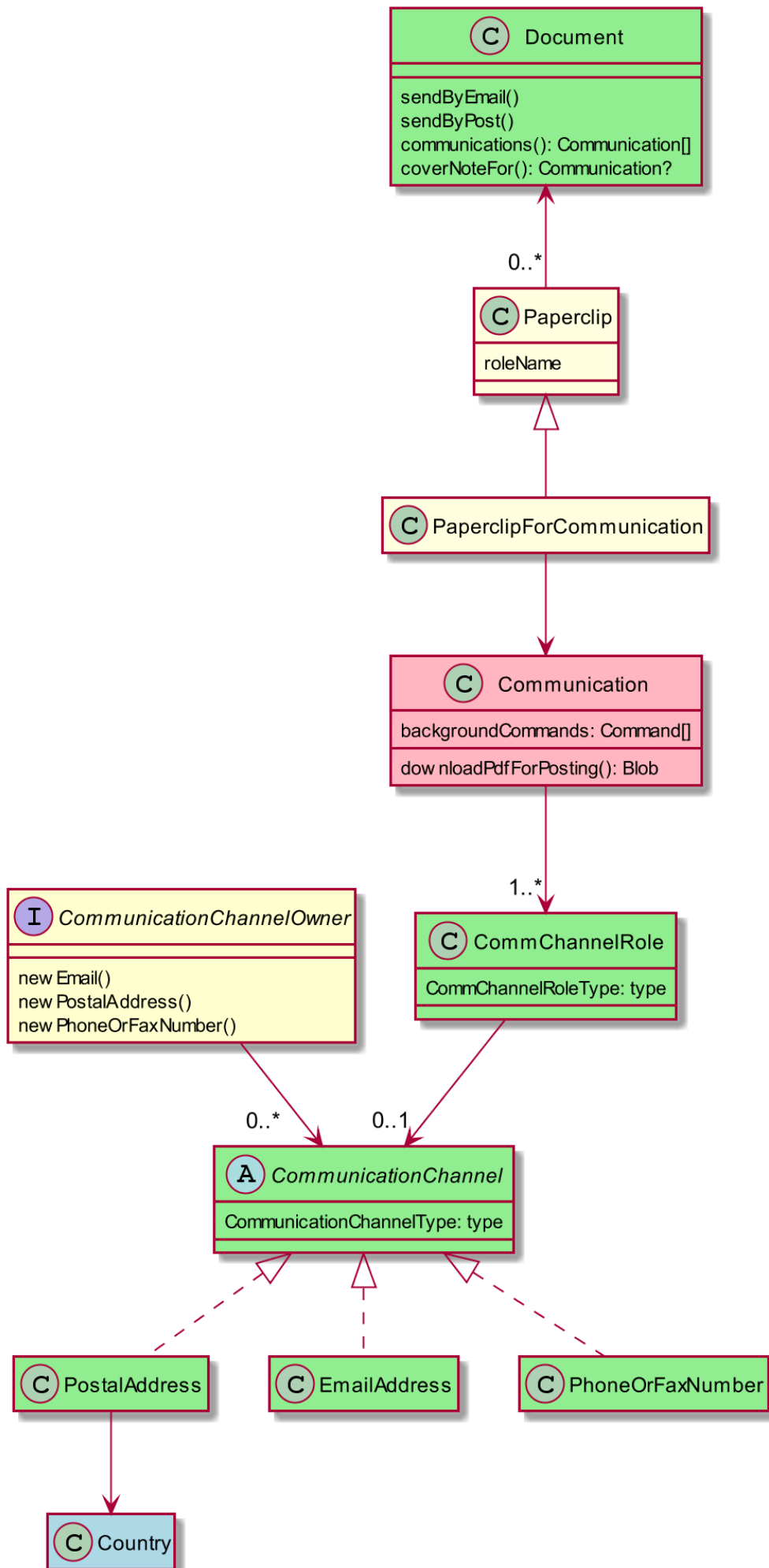


The only difference is that, when the **Documents** are downloaded for printing via the **Communication**, for convenience the PDFs will be stitched together into a single PDF for printing. The action prompt suggests a filename based on the original **Document** and supporting **Documents**.



Domain Model

The main concepts of the module are shown below:



One side of the domain model defines `CommunicationChannels`, owned by `CommunicationChannelOwners`.

On the other side is `Communication`, which relates to a `Document` by way of an implementation of the (`Document subdomain` module's) `Paperclip` class. Each `Communication` relates to one or more `CommunicationChannels` by way of `CommChannelRole`, basically indicating the nature of the correspondent in that `Communication`.

(CommunicationChannel) Contributions

The abstract `CommunicationChannelOwner_newChannelContributions` domain service contributes:

- `communicationChannels` collection
- `newPostalAddress` action
- `newEmailAddress` action
- `newPhoneOrFax` action

To use, the consuming application should create a subclass, for example:

```
@DomainService(nature = NatureOfService.VIEW_CONTRIBUTIONS_ONLY)
public class DemoAppCommunicationChannelOwner_newChannelContributions extends
    CommunicationChannelOwner_newChannelContributions {

    public DemoAppCommunicationChannelOwner_newChannelContributions() {
        super(DemoAppCommunicationChannelOwner_newChannelContributions.class);
    }
}
```

The `CommunicationChannelOwner_emailAddressTitles` & `CommunicationChannelOwner_phoneNumberTitles` are abstract mixins that contribute properties for use in tables; these summarise (any and all of the) emails or phone numbers that a `CommunicationChannelOwner` might have, using a specified separator.

For example:

```
@Mixin(method = "prop")
public class DemoObjectWithNotes_emailAddress extends
    CommunicationChannelOwner_emailAddressTitles {
    public DemoObjectWithNotes_emailAddress(
        final DemoObjectWithNotes demoCustomer) {
        super(demoCustomer, " | ");
    }
}
```

and

```
@Mixin(method = "prop")
public class DemoObjectWithNotes_phoneNumbers extends
    CommunicationChannelOwner_phoneNumberTitles {
    public DemoObjectWithNotes_phoneNumbers(
        final DemoObjectWithNotes demoCustomer) {
        super(demoCustomer, " | ");
    }
}
```

(Document) Mixins

The module provides a number of mixins that, by default, will be rendered in the UI. In the case of this module, all mixins are on the **Document** entity.

The mixins can be suppressed if necessary using vetoing subscribers to their corresponding domain events.

Document_sendByEmail & Document_sendByPost

These mixins on **Document** are used to create either email or postal **Communications**. They are supported by the **DocumentCommunicationSupport** SPI service, described [below](#).

The **DocumentCommunicationSupport** SPI provides the cover note template to use.



Note that the cover note template should have an **AttachmentAdvisor** set to "attach to none"; the mixin action has the responsibility of wiring the cover note **Document** to the newly created **Communication**.

Document_communications

The **Document_communications** collection mixin shows all **Communications** to which a **Document** has been sent as an attachment (in the case of an email) or to be printed out (in the case of a postal comm).

Note that this mixin is suppressed for cover notes; instead these have the **Document_coverNoteFor** mixin, described [below](#).

Document_communicationAttachments

The **Document_communicationAttachments** collection mixin lists all of the **Documents** that would be included as attachments if and when a new **Communication** is created.

This list always includes the target **Document** itself, and will also include any supporting **Document**'s that may have been attached (using the **Document** [subdomain](#) module's **'Document_attachSupportingPdf** mixin).

Document_coverNoteFor

The **Document_coverNoteFor** property mixin applies only to **Documents** that have been created as email cover notes. It returns a reference to the email **Communication** to which it was associated (with a role of "cover note"); its content is used as the body of the actual email.

Services (API)

The module currently does not provide a service to programmatically create **Communications**. Instead, the various **mixins** can be used.

Services (SPI)

SPI services are called by the module.

DocumentCommunicationSupport (required)

An implementation of the `DocumentCommunicationSupport` SPI domain service is required to send communications of any type. Its signature is:

```
public interface DocumentCommunicationSupport {
    DocumentType emailCoverNoteDocumentTypeFor(Document document);
    void inferEmailHeaderFor(Document document, CommHeaderForEmail header);
    void inferPrintHeaderFor(Document document, CommHeaderForPost header);
}
```

where `CommHeaderForEmail` is:

```
public class CommHeaderForEmail ... {

    @Getter @Setter
    private EmailAddress toDefault;
    @Getter
    private final Set<EmailAddress> toChoices = Sets.newTreeSet();

    @Getter @Setter
    private String cc ;
    @Getter @Setter
    private String bcc;

    @Getter @Setter
    private EmailAddress from;

    @Getter @Setter
    private String disabledReason; ①
}
```

① Reason, if any, why the communication cannot be sent by email.

and where `CommHeaderForPost` is:


```

public class CommHeaderForPost ... {
    @Getter @Setter
    private PostalAddress toDefault;
    @Getter
    private final Set<PostalAddress> toChoices = Sets.newTreeSet();

    @Getter @Setter
    private String disabledReason; ①
}

```

① Reason, if any, why the communication cannot be sent by post.



Note that the cover note template should have an `AttachmentAdvisor` set to "attach to none"; the mixin action has the responsibility of wiring the cover note `Document` to the newly created `Communication`.

CurrentUserEmailAddressProvider

The optional `CurrentUserEmailAddressProvider` SPI service provides the email address of the current user, in order to create a `CommChannelRole` indicating that the `Communication` was "prepared by" such-and-such a user.

Its signature is:

```

public interface CurrentUserEmailAddressProvider {
    String currentUserEmailAddress();
}

```

The module does provide a default implementation, `CurrentUserEmailAddressProvider.UsingMeService`, that uses the `MeService` of the `security` module. In many case therefore there will be no need to provide an alternative implementation of this 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.communications</groupId>
  <artifactId>incode-module-communications-dom</artifactId>
  <version>1.15.1.1</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.communications.dom.CommunicationsModule.class,
    );
}
```

Known issues

None known at this time.

Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/dom/communications/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
- [poly library](#) module
- [pdfbox library](#) module
- [country generic subdomain](#) module
- [document generic subdomain](#) module
- [command spi](#) module
- [security spi](#) module

The module *also* uses icons from [icons8](#).