Excel Library

Table of Contents

Screenshots 2
- Installing the Fixture Data 2
- Exporting items using the (example) bulk update manager 2
- Importing Exporting Excel 4
- View models represent the Excel rows 6
- (Limited) pivot support for Import 7

How to configure/use 9
- Classpath 9
- Bootstrapping 9

`ExcelService` API 10
- API 10
- Usage 13

`ExcelFixture` 16
- API 16
- Usage 17

Known issues 19

Related Modules 20

Dependencies 21

This module (`isis-module-excel`) provides a domain service so that a collection of (view model) object scan be exported to an Excel spreadsheet, or recreated by importing from Excel.

It also provides a fixture to allow data to be imported from an Excel spreadsheet, with each row either corresponding to a persistent entity or alternatively to a view model which in turn persists data.

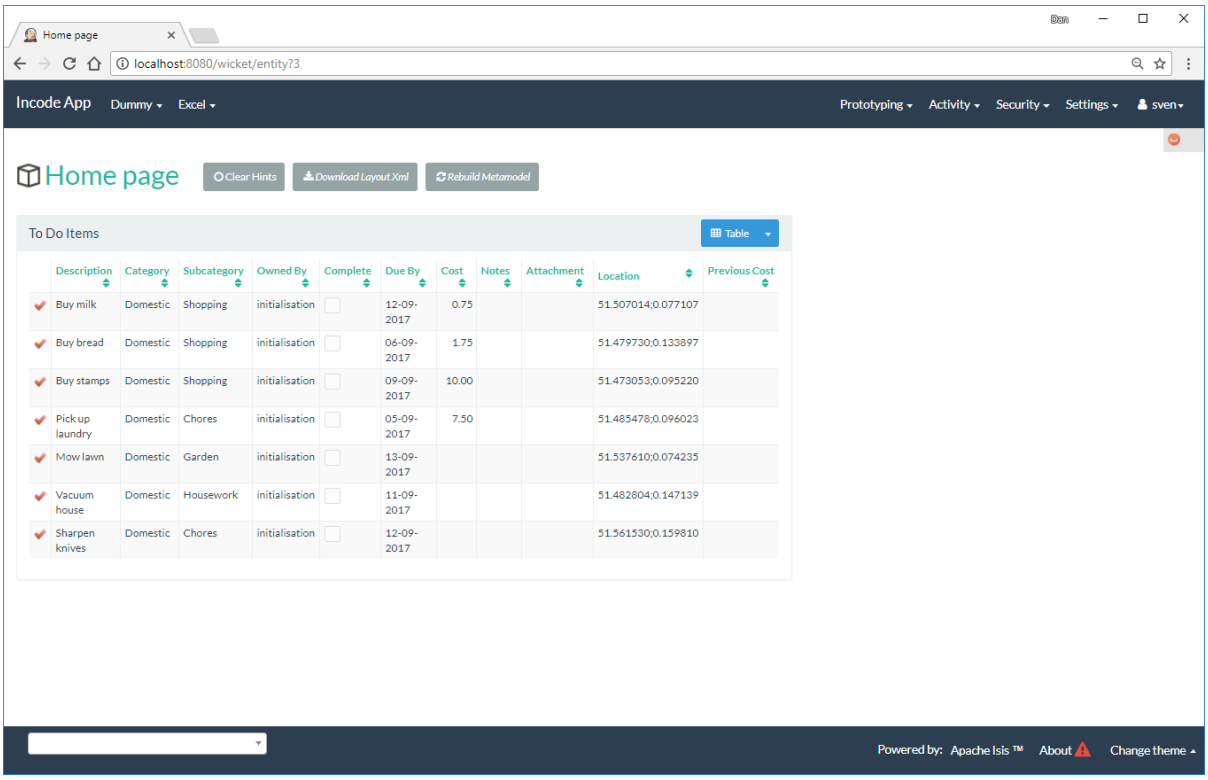
The underlying technology used is Apache POI.

Screenshots

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

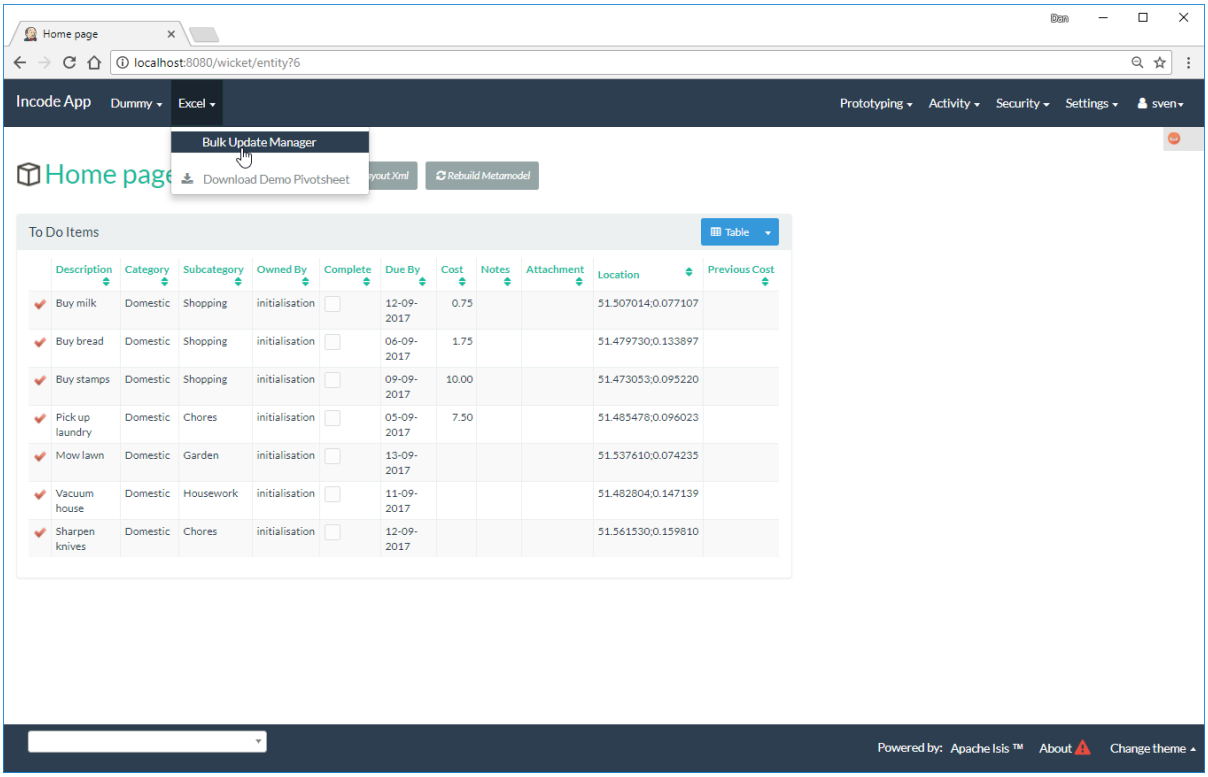
Installing the Fixture Data

A home page is displayed when the app is run:

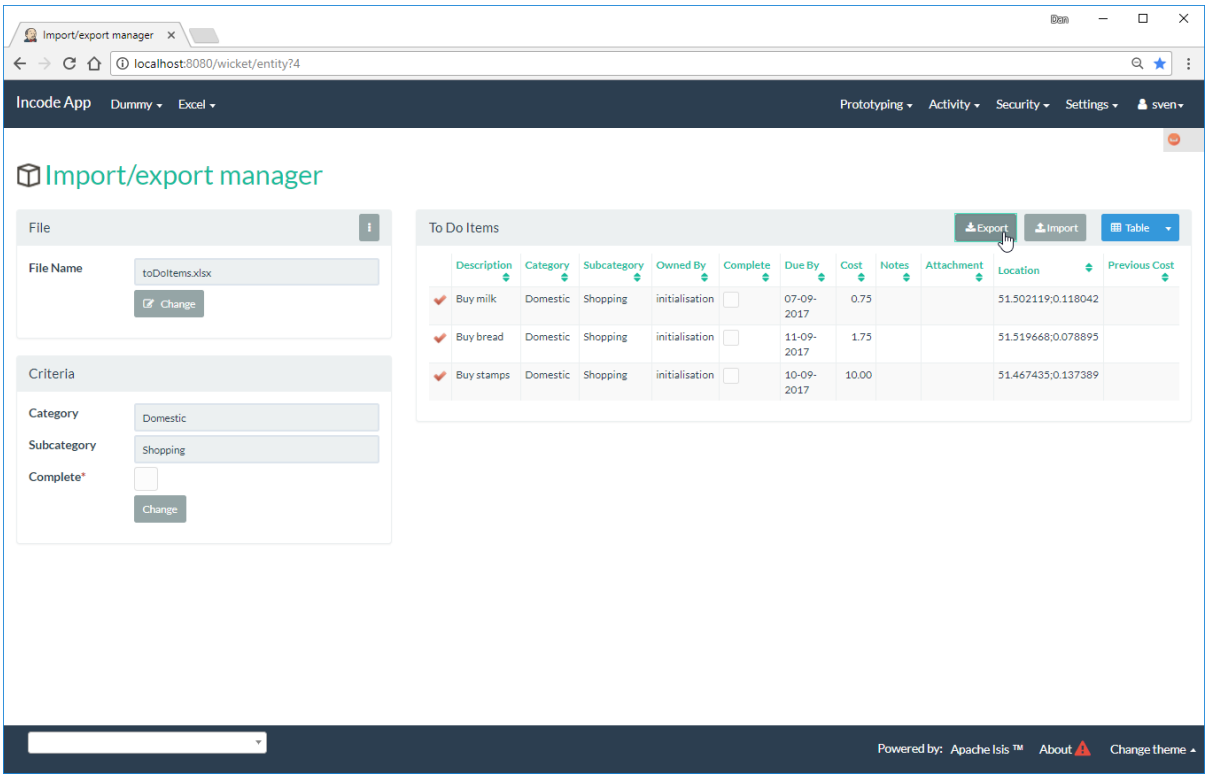


Exporting items using the (example) bulk update manager

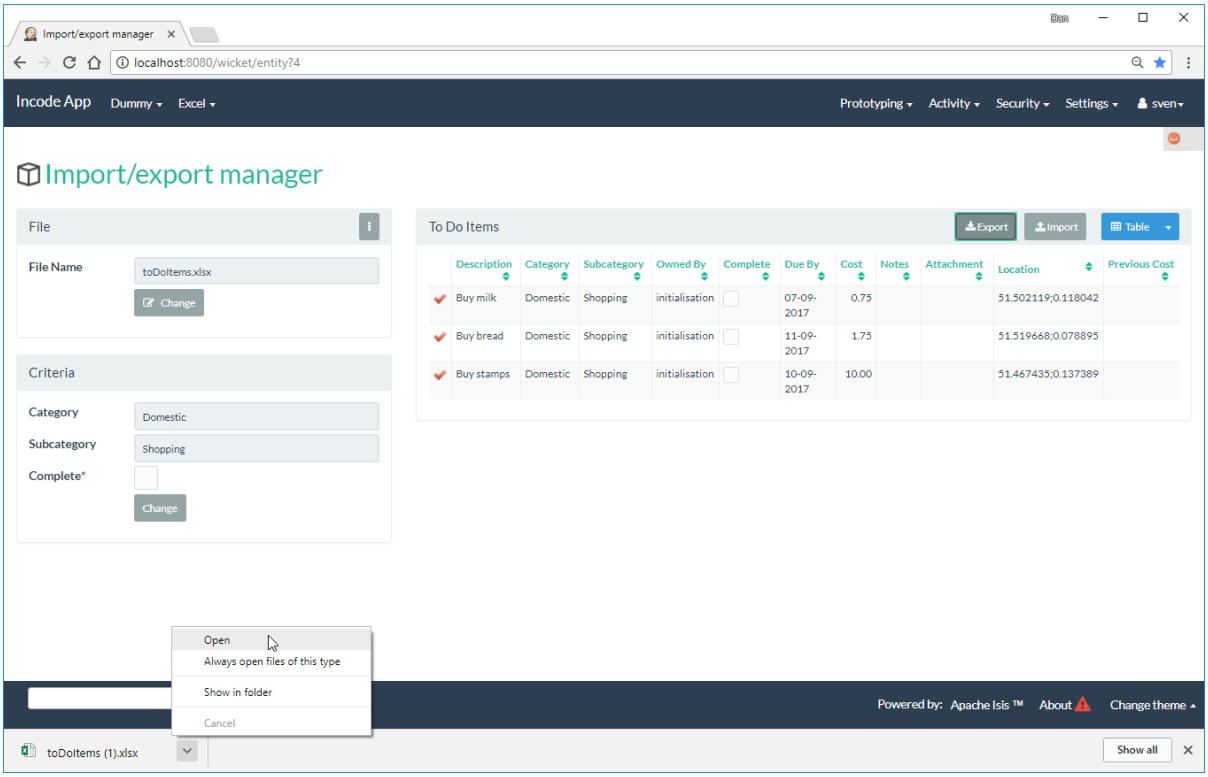
The example app has a bulk update manager as a wrapper around the ExcelService:



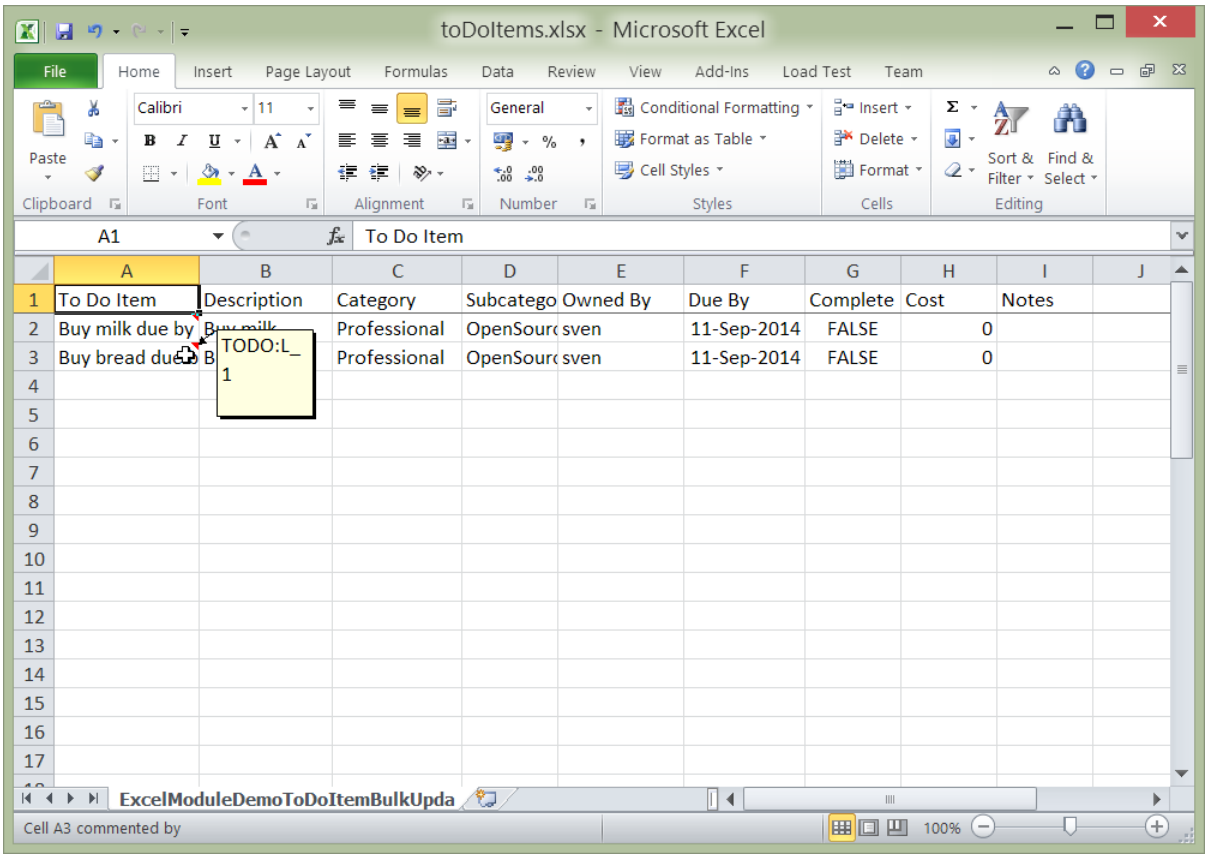
The (example) bulk update manager allows the end-user to define a criteria to exporting a (sub)set of items:



which are then downloaded ...

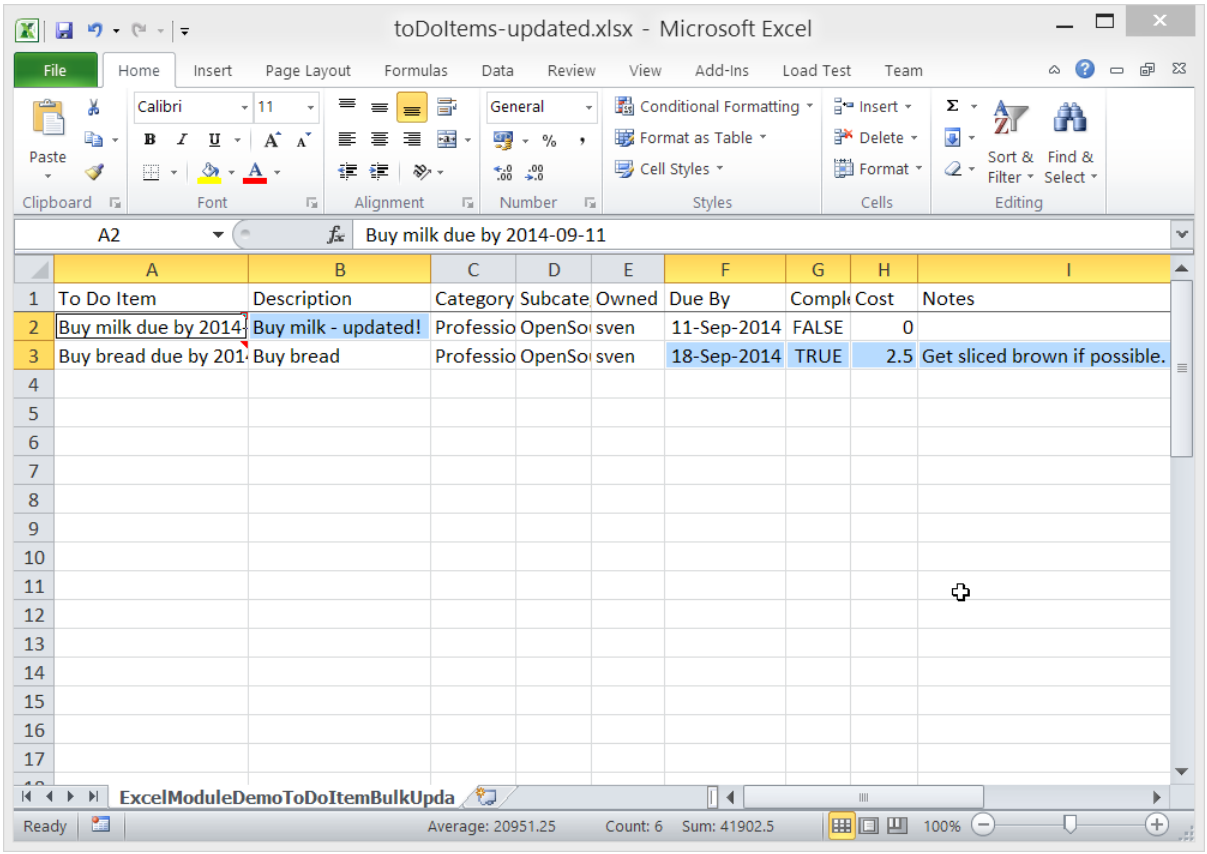


... and can be viewed in Microsoft Excel:

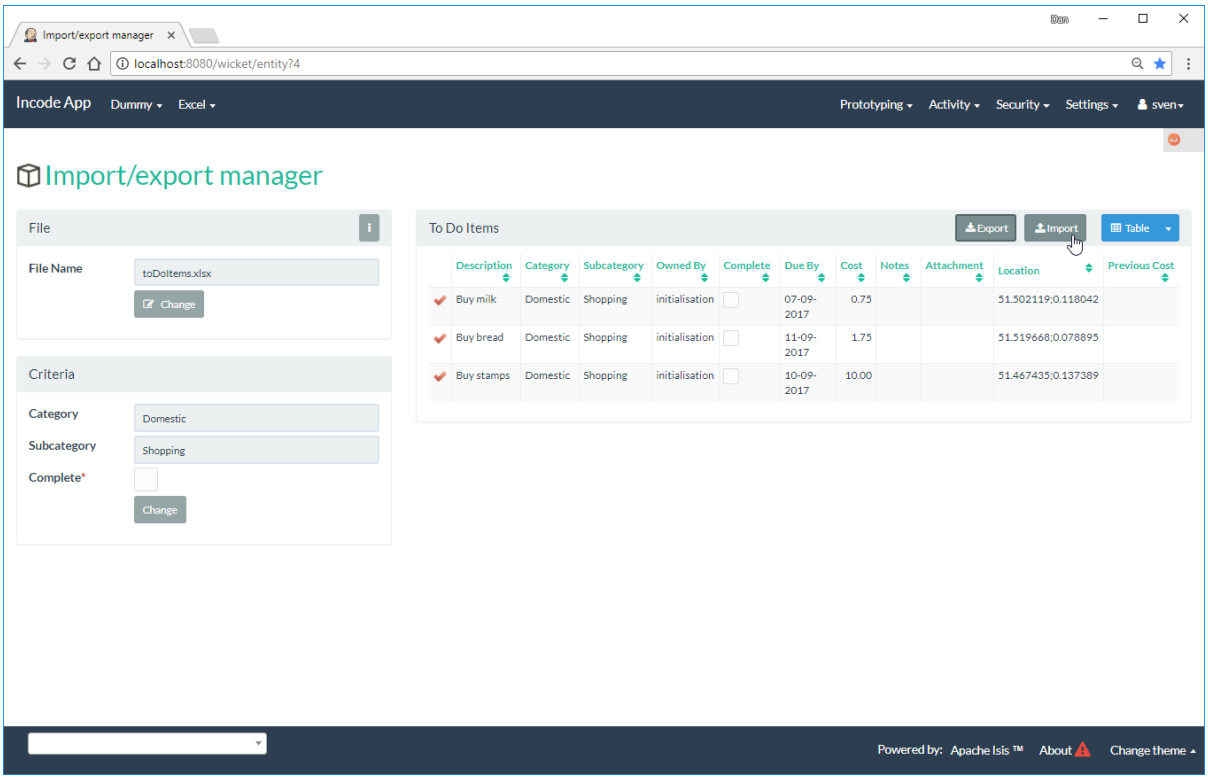


Importing Exporting Excel

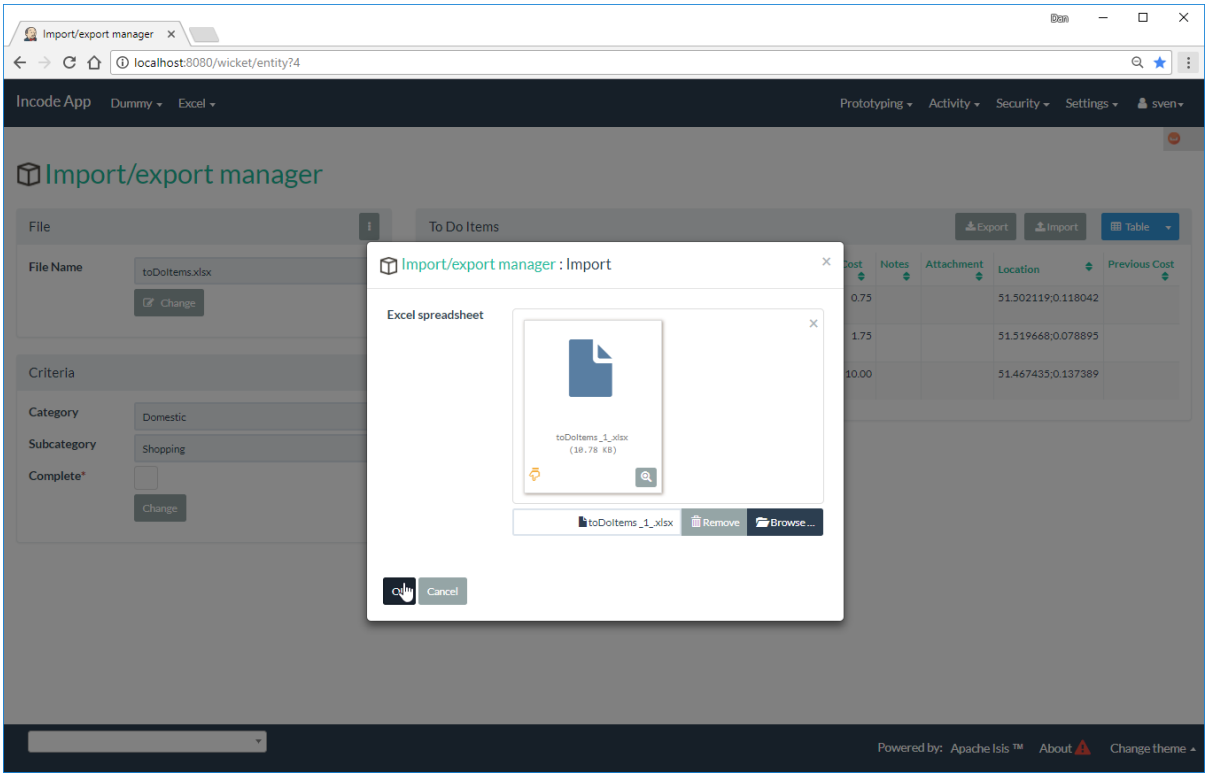
Using Excel the user can update data:



... and the use the (example) bulk update manager to import:

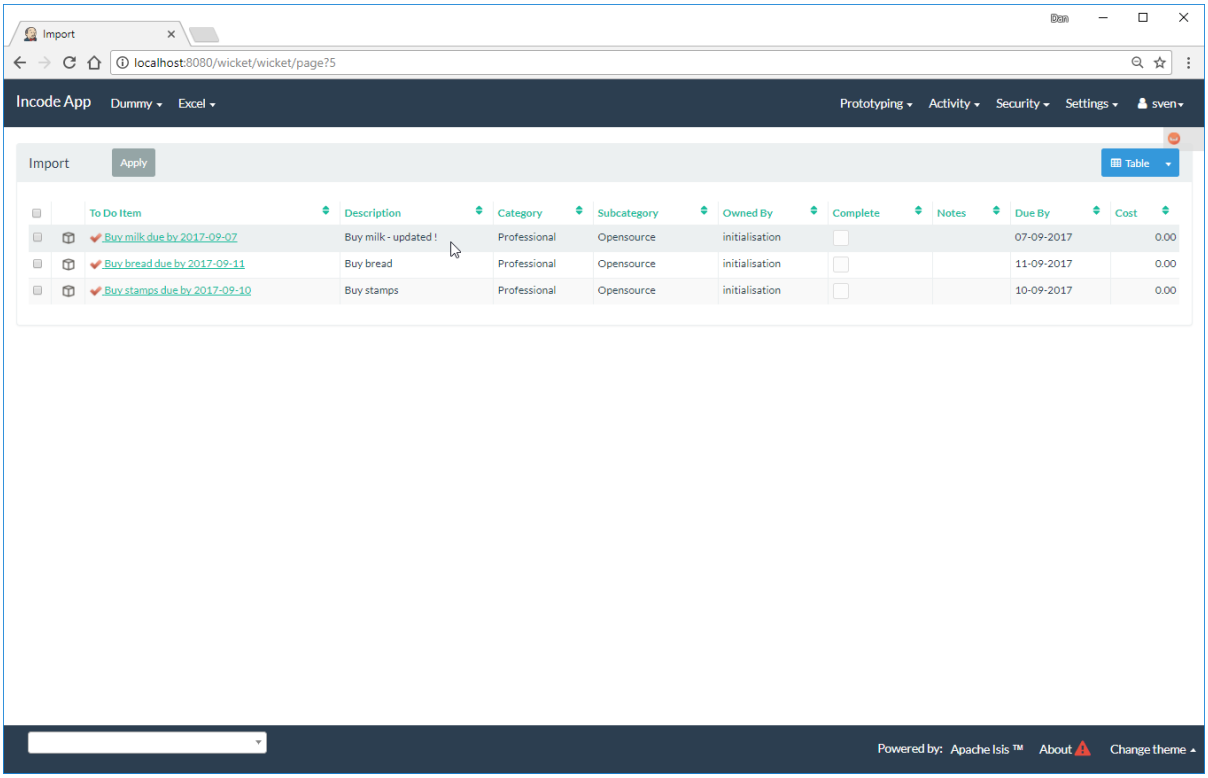


specifying the updated spreadsheet in the dialog:

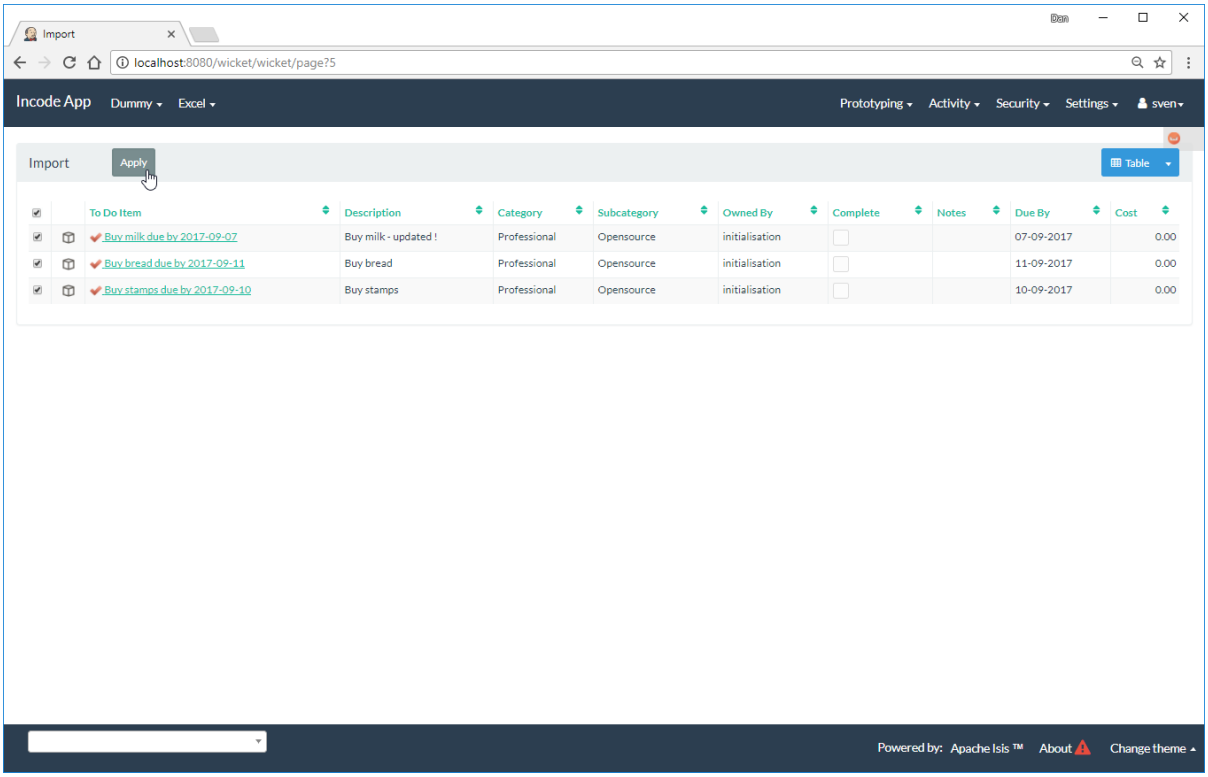


View models represent the Excel rows

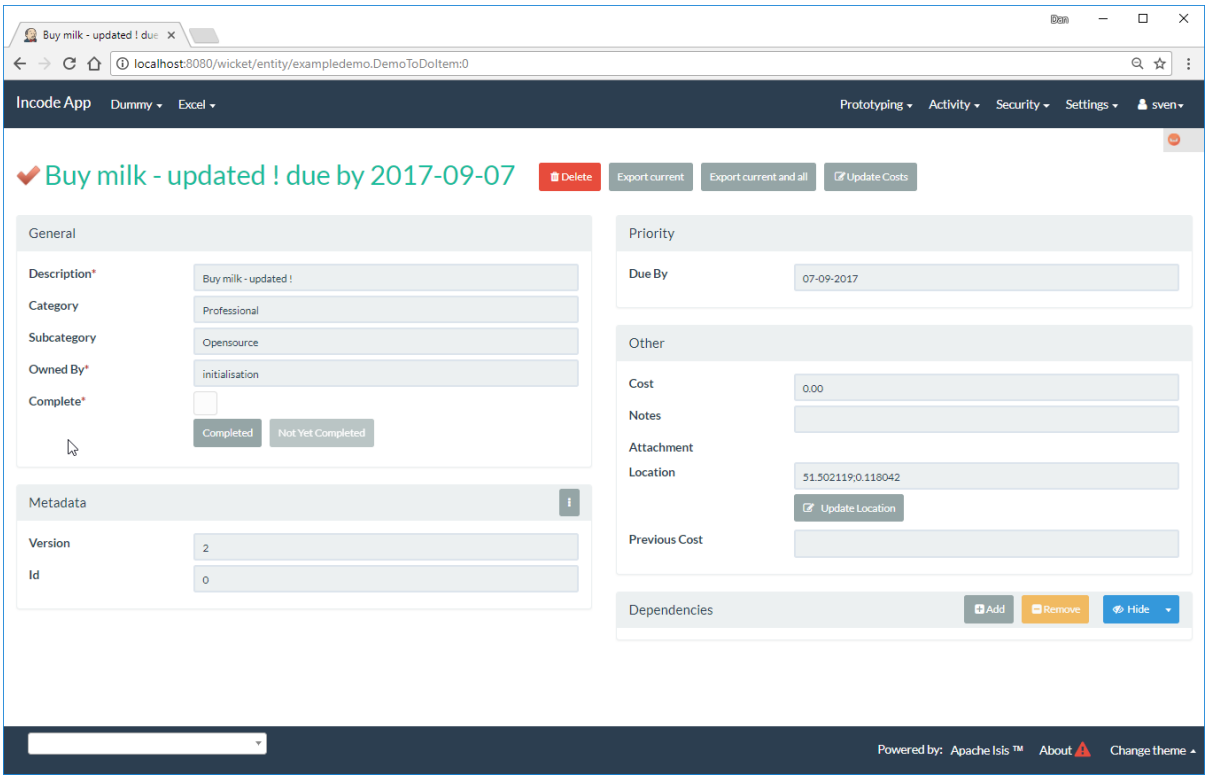
For each row in the spreadsheet the `ExcelService` instantiates a corresponding view model.



The view model can then provide a bulk `apply` action...



to update the corresponding entity:



(Limited) pivot support for Import

This module has support for pivot tables (export only) which is demonstrated by:

which are then downloaded and can be viewed in Microsoft Excel:

	A	B	C	D	E	F	G
1	Category	Domestic	Other	Professional			
2	Subcategory	Cost (sum)	Cost (sum)	Cost (sum)			
3	Shopping	12,5			12,5		
4	Chores	7,5			7,5		
5	Garden				0		
6	Housework				0		
7	Other				0		
8	Marketing				0		
9	Consulting				0		
10	Education				0		
11	OpenSource				0		
12		20	0	0	20		
13							
14							
15							

How to configure/use

Classpath

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

```
<dependency>
    <groupId>org.isisaddons.module.excel</groupId>
    <artifactId>isis-module-excel-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.isisaddons.module.excel.ExcelModule.class,
    );
}
```

ExcelService API

The ExcelService is intended for use by domain object classes.

API

The API exposed by ExcelService breaks into two.

Import

The first set of methods allow domain objects to be read (imported) from an Excel workbook:

```
public class ExcelService {
    public <T> List<T> fromExcel(                          ①
        Blob excelBlob,
        Class<T> cls) { ... }
    public <T> List<T> fromExcel(                          ②
        Blob excelBlob,
        WorksheetSpec worksheetSpec) { ... }
    public List<List<?>> fromExcel(                        ③
        Blob excelBlob,
        List<WorksheetSpec> worksheetSpecs) { ... }
    public List<List<?>> fromExcel(                        ④
        Blob excelBlob,
        WorksheetSpec.Factory factory) { ... }
    public List<List<?>> fromExcel(                        ⑤
        Blob excelBlob,
        WorksheetSpec.Factory factory,
        WorksheetSpec.Sequencer sequencer,
        ) { ... }
    ...
}
```

① converts a single-sheet workbook into a list of domain objects

② converts a single-sheet workbook into a list of domain objects, using WorksheetSpec (discussed below)

③ converts a multiple-sheet workbook into a list of list of domain objects.

④ converts all "matched" worksheets for a multiple-sheet workbook, with the supplied WorksheetSpec.Factory returning the WorksheetSpec to handle the sheet

⑤ as previous, but with the sheets resequenced using the supplied WorksheetSpec.Sequencer.

The WorksheetSpec is a data structure that specifies what is on each worksheet of the Excel workbook (ie which sheet of the workbook to read to obtain the domain objects):

WorksheetSpec.java

```
public class WorksheetSpec {
    public <T> WorksheetSpec (
        final Class<T> cls,                 ①
        final String sheetName) { ... }     ②
    ...
}
```

① the class of those domain objects

② the name of the sheet to be read. If omitted, then the simple name of the domain object class will be used. In all cases the sheet name must be 30 characters or less in length.

The `WorksheetSpec.Factory` API is:

WorksheetSpec.Factory.java

```
public class WorksheetSpec {
    ...
    public interface Factory {
        WorksheetSpec fromSheet(String sheetName);  ①
    }
}
```

① returns the `WorksheetSpec` indicating how the sheet should be handled, or `null` otherwise.

And the `Worksheet.Sequencer` API is simply:

WorksheetSpec.Sequencer.java

```
public class WorksheetSpec {
    ...
    public interface Sequencer {
        List<WorksheetSpec> sequence(List<WorksheetSpec> specs);
    }
}
```

Export

The second set of methods allow domain objects to be written out (exported) to an Excel workbook:

```
public class ExcelService {
    ...
    public <T> Blob toExcel(                                          ①
            final List<T> domainObjects,
            final Class<T> cls,
            final String fileName) { ... }
    public <T> Blob toExcel(                                          ②
            final WorksheetContent worksheetContent,
            final String fileName) { ... }
    public Blob toExcel(
            final List<WorksheetContent> worksheetContents,           ③
            final String fileName)  { ... }
```

① converts a list of domain objects to a single-sheet workbook, specifying the type of those domain objects.

② converts a list of domain objects to a single-sheet workbook, using `WorksheetContent` (discussed below)

③ converts a list of worksheet contents to a multi-sheet workbook

The fileName provided is used as the name of the returned `Blob`

Here `WorksheetContent` is a data structure that wraps the list of domain objects to be exported along with the afore-mentioned `WorksheetSpec`:

```
public class WorksheetContent {
    public <T> WorksheetContent(
        final List<T> domainObjects,        ①
        final WorksheetSpec spec) { ... }   ②
    ...
}
```

① the list of domain objects to be exported as an excel sheet

② the `WorksheetSpec`, describing the class of those domain objects and the worksheet name to use

In a likewise manner the following methods allow (annotated) domain objects to be exported to an Excel workbook in a pivot table.

```
public class ExcelService {
    ...
    public <T> Blob toExcelPivot(
            final List<T> domainObjects,
            final Class<T> cls,
            final String fileName) { ... }
    public <T> Blob toExcelPivot(
            final WorksheetContent worksheetContent,
            final String fileName) { ... }
    public Blob toExcelPivot(
            final List<WorksheetContent> worksheetContents,
            final String fileName)  { ... }
```

Usage

Given:

```
public class ToDoItemExportImportLineItem extends AbstractViewModel { ... }
```

which are wrappers around `ToDoItem` entities:

```
final List<ToDoItem> items = ...;
final List<ToDoItemExportImportLineItem> toDoItemViewModels =
    Lists.transform(items,
        new Function<ToDoItem, ToDoItemExportImportLineItem>(){
            @Override
            public ToDoItemExportImportLineItem apply(final ToDoItem toDoItem) {
                return container.newViewModelInstance(
                    ToDoItemExportImportLineItem.class,
                    bookmarkService.bookmarkFor(toDoItem).getIdentifier());
            }
        });
```

then the following creates an Isis `Blob` (bytestream) containing the spreadsheet of these view models:

```
return excelService.toExcel(
         toDoItemViewModels, ToDoItemExportImportLineItem.class, fileName);
```

and conversely:

```
Blob spreadsheet = ...;
List<ToDoItemExportImportLineItem> lineItems =
    excelService.fromExcel(spreadsheet, ToDoItemExportImportLineItem.class);
```

recreates view models from a spreadsheet.

Alternatively, more control can be obtained using `WorksheetSpec` and `WorksheetContent`:

```
WorksheetSpec spec = new WorksheetSpec(ToDoItemExportImportLineItem.class, "line-
items");

// export
WorksheetContents contents = new WorkbookContents(toDoItemViewModels, spec);
Blob spreadsheet = excelService.toExcel(contents, fileName);

// import
List<List> objects = excelService.fromExcel(spreadsheet, Collections.singletonList
(spec));
List<ToDoItemExportImportLineItem> items = objects.get(0);
```

more on the creation of pivot tables

In order to create a pivot table from a list of domain objects (normally Viewmodels) the following annotations on properties can be used.

```
@PivotRow
```

Indicates that the property will be used as row label in the pivot table (left most column). This annotation is mandatory and only 1 is allowed.

```
@PivotColumn(order = ..)
```

Indicates that the distinct values of the property will be used as column labels in the pivot table. This annotation is mandatory. More than 1 annotation is supported and they will be used in the order specified.

```
@PivotValue(order = .. , type = ..)
```

Indicates that the values of the property will be used as pivoted values in the pivot table. This annotation is mandatory. More than 1 annotation is supported and they will be used in the order specified. Type specifies the aggregation type, that defaults to AggregationType.SUM. At the moment the other supported type is AggregationType.COUNT

```
@PivotDecoration(order = ...)
```

Indicates that the distinct values of the property will be used as 'extra' values besides the row label (they "decorate" the label). This annotation is optional. More than 1 annotation is supported and they will be used in the order specified. Decoration assumes that all distinct labels are decorated

with the same values. This is not enforced however: the first decoration found will be used.

Here is the example used in the demo application

```
@DomainObject(nature = Nature.VIEW_MODEL)
public class ExcelModuleDemoPivot {

    ...

    @PivotRow
    private ExcelModuleDemoToDoItem.Subcategory subcategory;

    @PivotColumn(order = 1)
    private ExcelModuleDemoToDoItem.Category category;

    @PivotValue(order = 1, type = AggregationType.SUM)
    private BigDecimal cost;

}
```

ExcelFixture

The ExcelFixture is intended for use as part of the application's fixtures, as used for prototyping/demos and for integration tests. Behind the scenes it (re)uses the ExcelService.

API

The constructor for the ExcelFixture is:

```
public class ExcelFixture {
    public ExcelFixture(
        final URL excelResource,                        ①
        final Class... classes) {                       ②
            ...
        }
    }
    public void setExcelResourceName(String rn) { ... } ③
}
```

① the URL to the Excel spreadsheet

② a list of classes to process each of the sheets in the spreadsheet.

③ optionally, specify the name of the sheet. This is used only to disambiguate any results added to the FixtureResultList (displayed in the UI) if multiple spreadsheets are loaded using different ExcelFixture instances.

Each of the classes must either be a persistable entity or must implement the ExcelFixtureRowHandler interface:

```
public interface ExcelFixtureRowHandler {
    List<Object> handleRow(
            final FixtureScript.ExecutionContext executionContext,  ①
            final ExcelFixture excelFixture,                        ②
            final Object previousRow);                              ③
}
```

① to look up execution parameters, and to call addResult(…) (to make results available in the UI)

② provided principally so that addResult(…) can be called.

③ to support sparsely populated spreadsheets where a null cell means to use the value from the previous row. Particularly useful for spreadsheets that group together multiple entities (eg category/subcategory/item).

The fixture is instantiated and executed in the usual way, as per any other fixture script.

The fixture uses the class name to lookup the sheet of the workbook:

- it first tries to find a sheet with the class' simpleName

- if a sheet cannot be found, and if the class' simpleName ends with "RowHandler", then it will look for a sheet without this suffix.

For example, the class `ExcelModuleDemoToDoItemRowHandler` will match a sheet named "ExcelModuleDemoToDoItemRowHandler".

> Excel sheet names can be no longer than 30 characters

Assuming the sheet has been located, the fixture will instantiate an instance of the class for each row, and set the properties of the sheet according to the headers. If the class is persistable, it will then attempt to persist the object using `DomainObjectContainer#persist(…)`. Otherwise (where the class implements `ExcelFixtureRowHandler`), the `handleRow(…)` method will be called.

The fixture makes all created objects available to the caller through two accessors:

- `getObjects()` returns all objects created by any of the sheets
- `getObjects(Class)` returns all objects created by an entity/row handler for a given sheet

Usage

The `ExcelFixture` is used as follows:

```
final URL excelResource = Resources.getResource(getClass(), "ToDoItems.xlsx");
①
final ExcelFixture excelFixture = new ExcelFixture(excelResource,
ExcelModuleDemoToDoItemRowHandler.class); ②
executionContext.executeChild(this, excelFixture);
③
List<Object> items = excelFixture.getObjects(ExcelModuleDemoToDoItemRowHandler.class);
④
```

① eg using google guava library

② expect a single sheet

③ execute in the usual way

④ obtain the objects created by the `ExcelModuleDemoToDoItemRowHandler` for its corresponding sheet

where:

```
public class ExcelModuleDemoToDoItemRowHandler implements ExcelFixtureRowHandler {
①
    ...
②
    @Override
    public List<Object> handleRow(
            final FixtureScript.ExecutionContext executionContext,
③
            final ExcelFixture fixture,
            final Object previousRow) {
        final ExcelModuleDemoToDoItem toDoItem = ...;
        executionContext.addResult(fixture, todoItem);
④
        return Collections.<Object>singletonList(toDoItem);
⑤
    }
    ..
⑥
}
```

① implement the `ExcelFixtureRowHandler` interface

② getters and setters omitted

③ `ExecutionContext` can be used to pass parameters down to the row handler, and to call addResult

④ make available in the UI

⑤ return a list of objects instantiated by this row handler.

⑥ eg inject domain services/repositories to delegate to for instantiating objects

Known issues

None known at this time.

Related Modules

See also the Excel wicket component, which makes every collection downloadable as an Excel spreadsheet.

Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/lib/excel/impl -D excludeTransitive=true
```

which, excluding Apache Isis itself, returns these compile/runtime dependencies:

```
org.apache.poi:poi:jar:3.9
org.apache.poi:poi-ooxml-schemas:jar:3.9
org.apache.poi:poi-ooxml:jar:3.9
```

For further details on 3rd-party dependencies, see:

- Apache POI