

FLEXIBLE FORMS

DEV REPORT

rev 1.4

Porto, April 2022

Table of Contents

1	INTRODUCTION.....	1
2	TECHNOLOGIES.....	1
2.0.1	<i>Programming Languages.....</i>	<i>1</i>
2.0.2	<i>Frameworks.....</i>	<i>1</i>
2.0.3	<i>Libraries.....</i>	<i>2</i>
2.0.4	<i>Django Apps.....</i>	<i>4</i>
2.0.5	<i>Databases.....</i>	<i>4</i>
3	ARCHITECTURE AND APPLICATIONS.....	5
3.0.1	<i>Concept.....</i>	<i>5</i>
3.0.2	<i>System.....</i>	<i>5</i>
3.0.3	<i>Database.....</i>	<i>7</i>
4	DESCRIPTION AND SOURCE CODE.....	9
4.1	BUILDING BLOCKS.....	9
4.2	CONTEXT.....	10
4.3	URLs.....	10
4.4	SIGNALS.....	10
4.5	MODALS.....	16
4.6	DESIGNER.....	18
4.6.1	<i>Elements.....</i>	<i>19</i>
4.6.1.1	<i>Dimensions.....</i>	<i>22</i>
4.6.1.2	<i>Properties / Visibility.....</i>	<i>23</i>
4.6.1.3	<i>Availability.....</i>	<i>26</i>
4.6.1.4	<i>Validations.....</i>	<i>27</i>
4.6.1.5	<i>Element ID.....</i>	<i>28</i>
4.6.1.6	<i>Move / Resize.....</i>	<i>28</i>
4.6.1.7	<i>Selection.....</i>	<i>29</i>
4.6.1.8	<i>Locking Elements.....</i>	<i>30</i>
4.6.1.9	<i>Repeatable Elements.....</i>	<i>30</i>
4.6.1.10	<i>Copying / Cloning Elements.....</i>	<i>31</i>
4.6.1.11	<i>Copy/Paste – the Brush.....</i>	<i>31</i>
4.6.1.12	<i>Data sources.....</i>	<i>32</i>
4.6.1.13	<i>Dropdowns.....</i>	<i>32</i>
4.6.1.14	<i>DB Elements.....</i>	<i>32</i>
4.6.2	<i>Pagination System.....</i>	<i>33</i>
4.6.3	<i>Zoom.....</i>	<i>34</i>
4.6.4	<i>Grid and Snap.....</i>	<i>34</i>
4.6.5	<i>Load / Save.....</i>	<i>37</i>
4.6.6	<i>New / Open Form.....</i>	<i>39</i>
4.7	PREVIEW.....	40
4.8	FORMS MANAGER.....	41
4.9	OPERATIONS MANAGER.....	41
4.10	OPERATIONS EDITOR.....	42
4.11	INSPECTOR’S APP.....	42
4.12	BUILDING THE FORMVIEW / LISTVIEW.....	43
4.12.1	<i>Elements.....</i>	<i>43</i>
4.12.2	<i>Synchronization between the Views.....</i>	<i>44</i>
4.12.3	<i>Sections.....</i>	<i>45</i>

4.12.4	<i>Groups</i>	46
4.12.5	<i>E/A System</i>	48
4.12.5.1	<i>Events</i>	49
4.12.5.2	<i>Actions</i>	50
4.12.6	<i>Load / Save</i>	51
4.13	VALIDATIONS.....	53
4.14	PRINTING / EXPORTING TO PDF.....	55
4.15	OFFLINE SYSTEM.....	56
4.15.1	<i>Technologies</i>	57
4.15.2	<i>Development</i>	57
4.15.2.1	<i>Stores</i>	58
4.15.2.2	<i>App and ServiceWorker comms</i>	60
4.15.2.3	<i>Synchronization</i>	61
4.16	BARCODE READER.....	66
4.17	SMART CARD SIGNATURE.....	68
4.17.1	<i>Desktop/Windows</i>	69
4.17.2	<i>Android</i>	73
4.18	MULTI-LANGUAGE SUPPORT.....	73
4.18.1	<i>By Django</i>	73
4.18.2	<i>By Javascript</i>	75
4.19	TEMPLATE THEME.....	76
4.20	ADDING A NEW ELEMENT TYPE.....	76
4.20.1	<i>In Designer</i>	77
4.20.2	<i>In SForm</i>	78
5	SECURITY	79
5.1	ACCOUNTS.....	79
5.2	TOKENS.....	80
5.3	PERMISSIONS.....	81
6	ADMINISTRATION	83
6.1	FILES.....	83
6.2	AUTOMATIC TASKS.....	84
7	APIS	86
8	TODO	90

List of Figures

Fig. 1: <i>The views</i>	5
Fig. 2: System composition.....	6
Fig. 3: <i>Database diagram</i>	7
Fig. 4: <i>Are you sure modal</i>	17
Fig. 5: <i>Error modal</i>	18
Fig. 6: <i>Warning modal</i>	18
Fig. 7: <i>The designer application</i>	18
Fig. 8: The different elements <i>available at the Elements sidebar</i>	19
Fig. 9: <i>The Properties sidebar, depending on element's selection</i>	24
Fig. 10: <i>Preview button in the Designer</i>	40
Fig. 11: <i>Main view of the Preview</i>	41
Fig. 12: <i>Screen transitions</i>	42
Fig. 13: <i>Inputs validation</i>	55
Fig. 14: <i>The printing process</i>	56
Fig. 15: <i>IndexedDB Stores</i>	59
Fig. 16: <i>App synchronizing</i>	62
Fig. 17: <i>Offline warning</i>	62
Fig. 18: <i>Barcode Modal</i>	66
Fig. 19: <i>Signing Modal</i>	68
Fig. 20: <i>Language selection</i>	74
Fig. 21: <i>Setting permissions in the administrator page</i>	82

List of Tables

Tab. 1: <i>Programming languages</i>	1
Tab. 2: <i>Frameworks</i>	1
Tab. 3: <i>Libraries</i>	3
Tab. 4: <i>Django Apps</i>	4
Tab. 5: <i>Databases</i>	4
Tab. 6: <i>Description of the different blocks</i>	6
Tab. 7: <i>“Extra” applications</i>	6
Tab. 8: <i>The Asset_type table content</i>	7
Tab. 9: <i>The Status table content</i>	8
Tab. 10: <i>The Files table content</i>	8
Tab. 11: <i>Designer's signals</i>	15
Tab. 12: <i>Designer Modals</i>	16
Tab. 13: <i>Forms Manager Modals</i>	17
Tab. 14: <i>SForm Modals</i>	17
Tab. 15: <i>Global Modals</i>	17
Tab. 16: <i>Designer's main scripts</i>	19
Tab. 17: <i>Editable elements</i>	21
Tab. 18: <i>Static elements</i>	21
Tab. 19: <i>Auto elements</i>	22
Tab. 20: <i>Elements fields</i>	22
Tab. 21: <i>List of all available properties</i>	24
Tab. 22: <i>Properties visibility vs selected Element(s) – 1 is visible, 0 is not visible</i>	26
Tab. 23: <i>Availability</i>	27

Tab. 24: Possible <i>validations per element</i>	27
Tab. 25: <i>Pagination System</i>	33
Tab. 26: <i>Grid System</i>	34
Tab. 27: <i>Load/Save System</i>	38
Tab. 28: <i>Json representing a Form</i>	39
Tab. 29: <i>App screens</i>	42
Tab. 30: <i>Elements fields</i>	57
Tab. 31: <i>Stores</i>	59
Tab. 32: <i>Cases when synchronizing operations</i>	65
Tab. 33: <i>Back-office tables</i>	83
Tab. 34: <i>Designer APIs</i>	87
Tab. 35: <i>FormsManager APIs</i>	87
Tab. 36: <i>Operations APIs</i>	88
Tab. 37: <i>Rest APIs</i>	89

List of Abbreviations and Symbols

CRSF	Cross-Site Request Forgery
PWA	Progressive Web Application
SPA	Single Page Application

1 Introduction

This document presents a brief overlook on the technologies, architecture and the system itself.

2 Technologies

Tables 1, 2, 3, 4 and 5 list the main technologies used in the system's development.

2.0.1 Programming Languages

Name	Version
Python	3.10.0
Javascript	-
C++	C++17

Tab. 1: Programming languages

2.0.2 Frameworks

Name	Version	Description	Url
Django	3.1	High-level Python web framework.	https://www.djangoproject.com/
Django Rest Framework	3.12	Powerful and flexible toolkit for building Web APIs.	https://www.django-rest-framework.org/
Bootstrap 4	4.6.0	Sleek, intuitive, and powerful front-end framework for faster and easier web development.	https://github.com/twbs/bootstrap/tree/v4-dev

Tab. 2: Frameworks

2.0.3 Libraries

Name	Version	Description	Url
jQuery	3.6.0	jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.	https://jquery.com/
jQuery UI	1.12.1	jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library. Whether you're building highly interactive web applications or you just need to add a date picker to a form control, jQuery UI is the perfect choice.	https://jqueryui.com/
uuid		For the creation of RFC4122 UUIDs	https://github.com/uuidjs/uuid
jstree	3.3.11	jsTree is jquery plugin, that provides interactive trees.	https://www.jstree.com/
js-signals		Custom Event/Messaging system for JavaScript	http://millermedeiros.github.io/js-signals/
Grid-builder	1.2	gridBuilder.js is a jQuery plugin that draws a grid as a background.	https://github.com/Kilian/grid-Builder.js/
Sheets-of-Paper		Emulates sheets of paper in web documents (but without skeuomorphic paper textures)	https://github.com/delight-im/HTML-Sheets-of-Paper
Font awesome	5.15.3	Get vector icons and social logos	https://fontawesome.com/
Flag Icon	3.5.0	A collection of all country flags in SVG — plus the CSS for easier integration.	https://github.com/shadowsky20/flag-icon-css
Leafletjs	1.7.1	an open-source JavaScript library for mobile-friendly interactive maps	https://leafletjs.com/

FileSaver	2.0.4	FileSaver.js is the solution to saving files on the client-side, and is perfect for web apps that generates files on the client.	https://github.com/eligrey/FileSaver.js
jszip	3.2.0	JSZip is a javascript library for creating, reading and editing .zip files, with a lovely and simple API.	https://stuk.github.io/jszip/
Quagga	0.12.1	QuaggaJS is a barcode-scanner entirely written in JavaScript supporting real-time localization and decoding of various types of barcodes such as EAN, CODE 128, CODE 39, EAN 8, UPC-A, UPC-C, I2of5, 2of5, CODE 93 and CODABAR. The library is also capable of using <code>getUserMedia</code> to get direct access to the user's camera stream. Although the code relies on heavy image-processing even recent smartphones are capable of locating and decoding barcodes in real-time.	https://serratus.github.io/quaggaJS
PDFlib	1.16.0	Create and modify PDF documents in any JavaScript environment.	https://pdf-lib.js.org
Html2-Canvas	1.2.1	Screenshots with JavaScript	https://html2canvas.hertzen.com/
Leaflet-image	0.0.4	Export images out of Leaflet maps without a server component, by using Canvas and CORS .	https://github.com/mapbox/leaflet-image
Dexie	3.2.0	A Minimalistic Wrapper for IndexedDB	https://dexie.org/
papaparse	5.0.2	The powerful, in-browser CSV parser for big boys and girls	https://www.papaparse.com/

Tab. 3: Libraries

2.0.4 Django Apps

Name	Version	Description	Url
django-adminlte3	0.1.6	Admin LTE templates, admin theme, and template tags for Django	https://pypi.org/project/django-adminlte3/
django-pwa	1.0.10	A Django app to include a <i>manifest.json</i> and Service Worker instance to enable progressive web app behavior	https://pypi.org/project/django-pwa/
django-cleanup	6.0.0	Automatically deletes files for <i>FileField</i> , <i>ImageField</i> and subclasses	https://pypi.org/project/django-cleanup/

Tab. 4: Django Apps

2.0.5 Databases

Name	Version	Description	Url
MariaDB	10.6.4	MariaDB is a community-developed, commercially supported fork of the MySQL	https://mariadb.org/

Tab. 5: Databases

3 Architecture and Applications

3.0.1 Concept

The envisioned solution consists in a dual view approach: Form View and List View. Since both views are synchronized with each-other, the form's user has the ability to switch between the view, whatever he desires (fig. 1).

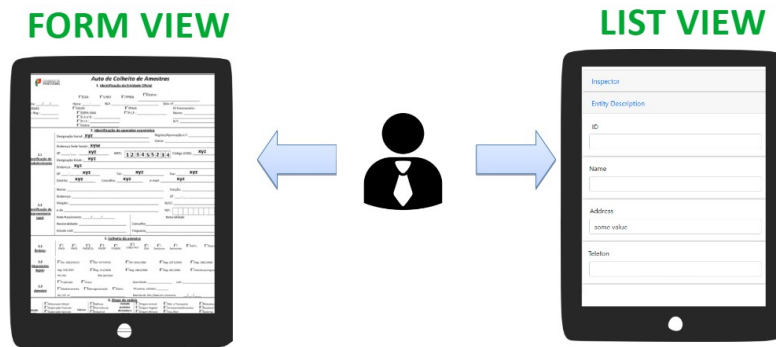


Fig. 1: The views

The reasons for this dual view were:

- since printing a form was one of the requirements, the Form View was a given;
- using the Form View in a small device, like a smartphone can become cumbersome. The List View provides a view similar to Google Forms, which is more adequate in these type of small screen devices;
- most people are used to fill forms in a certain way, so providing both, the users can begin to adjust their work habits.

3.0.2 System

The system consists of seven applications (fig. 2). These are described in table 6.



Fig. 2: System composition

What	Description
Designer	Editor of forms
Preview	Previews of any form
Forms Manager	Manages all forms, regardless of their state
Operations Manager	Manages all operations, regardless of their state
Operations Editor	Edits any non COMPLETED operation
Inspectors App	Creates and manages a users' Operations
Rest API	Available to managers and administrators to manage Operations

Tab. 6: Description of the different blocks

There are other application which are only relevant to the administrator. These are described in table 7.

Application	Description
Accounts	Custom users management.
Files	Manages data, config and misc. files, required for the system

Tab. 7: "Extra" applications

3.0.3 Database

The database structure (apart from Django's own tables, such as, *auth_permissions* e *django_admin_log*), is composed by only 10 tables (see fig. 3). The content of *asset_type* and *status* and *files* tables are represented in tables 8, 9 and 10.

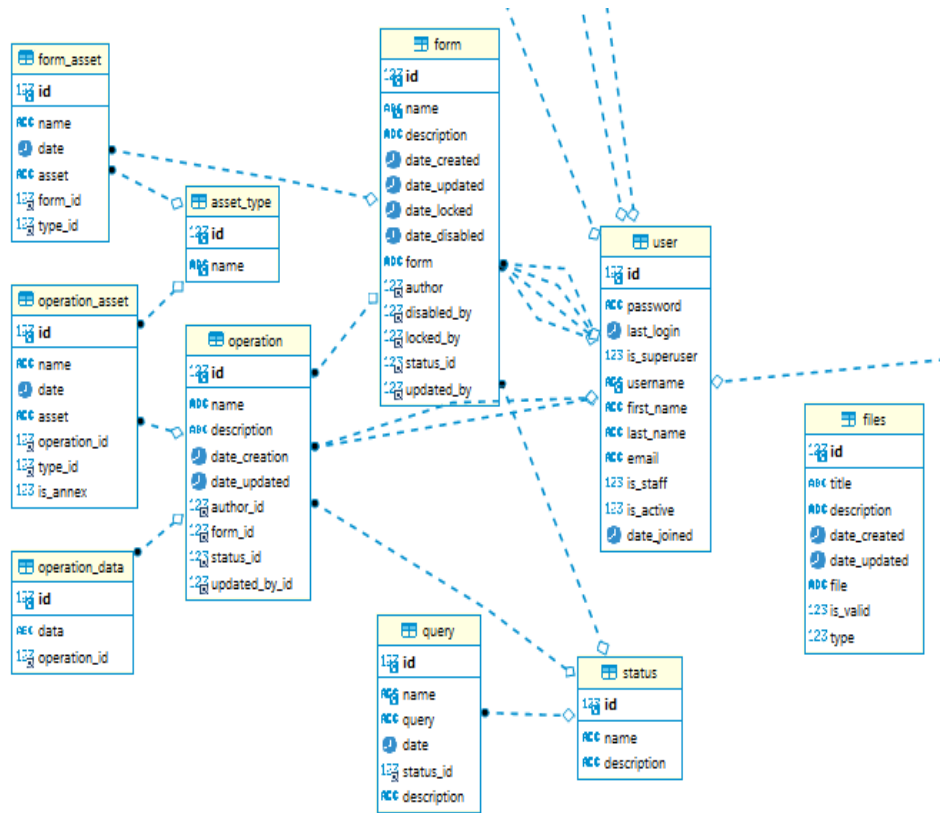


Fig. 3: Database diagram

ID	Name
1	IMAGE
2	CSV
3	JSON
4	PDF
5	OTHER

Tab. 8: The Asset_type table content

ID	Name	Description
2	CLOSED	Filling is completed and ready for data transfer.
3	OPEN	Form is being filled.
4	COMPLETED	Filling completed + data in the database.
5	LOCKED	-
6	DISABLED	Form is disabled => not editable, not usable. Obsolete
7	TEMPORARY	Temporary form. Can be removed or saved (=> not temp any more) – Designer
8	IN USE	Form is current and usable. Not editable.
9	EDITABLE	Form can be edited.
10	NONE	-

Tab. 9: The Status table content

ID	Name	Description
1	foodex_attributes_EN.csv	Latest Foodex2 attributes
2	foodex_terms_EN.csv	Latest Foodex2 terms
3	CCNativeApp.exe	Application for windows 64bits, for signing a PDF document.
4	chrome_extension.zip	Chrome extension, for signing a PDF document.
5	users_manual.pdf	Users' manual
6	forms_dev_notes.pdf	Development notes.

Tab. 10: The Files table content

4 Description and Source Code

Around 90% of the code is dynamically generated on the client side. The code does not follow any pre-determined style or principle. It's mixes *jquery* with pure *javascript*. Each one is used depending on the situation, and the decision of using one or another, lies with which one was the provided the simplest and easiest way to achieve a certain task.

For example, for selection, in some cases it was used the javascript `document.getElementById` and in other the simple *jquery* selector `$`. The first one was used primarily for selecting static elements, while the latter for dynamically generated elements.

4.1 Building Blocks

Except for rare occasion, all elements are build in pure javascript, using the classes defined in *BuildingBlocks.js*. The reason for this, was to harmonize and flexibilize the build. For example, in the future, it would be quite simple to add some pre-processing to every *Input* element.

So, instead of:

```
const div = $("<div>");  
X.append(div);
```

or

```
const div = document.createElement("div");  
x.appendChild(div);
```

We do:

```
const div = new Div().attachTo(X);
```

4.2 Context

The Context object, can be considered as a global object, which is passed to most objects in an application.

4.3 URLs

All urls used by the application are centrally defined in */static/js/urls.js*.

```
export const URL_DESIGNER = '/designer/';  
export const URL_PREVIEW = '/preview/';  
export const URL_FORMS_MANAGER = '/formsmanager/';  
...
```

Note however, that *sform/js/offline/offlinedb.js* repeats many of these urls. Therefore, any change required both files to be updated.

4.4 Signals

All applications have a signals systems based on the observer pattern, that allows different parts of the application to respond to events or signals from other parts, without the need to use a convoluted system of callback functions arguments everywhere.

Declaration:

```
var Signal = signals.Signal;  
this.signals = {  
  onX: new Signal()  
}
```

Subject or Dispatchers:

```
signals.onX.dispatch([param[, param[, ... param]]]);
```


Observer or Listeners:

```
signals.onX.add(
    ([param[, param[, ... param]]) => {}
);
```

Table 11 lists all signals used in the *Designer*.

Event	Dispatchers	Listeners	Parameters
onAnyElementChanged	EAManager	EAElementSelector	-
onAYS	FormView ListView GroupsView DBTableColumn- ValuesSelector TableColumnVal- ueSelector	Main	function, function
onChange	*	Main	-
onCheckRadioChanged	GroupsManager	CRElementSelector	-
onCursorGrid- Changed	Grid	BaseElement	delta
onDBFieldSelected	DBFields2UIModal FieldSelection- Modal	main	-
onEACloned	EA	EAManager EAManager	ea_id
onEACreated	EAManager	EAView	ea_id, name
onEARemoved	EA	EAView EAManager	ea_id
onEAStatusChanged	ActionSelector OperatorSelector EventsSelector LogicalOpSelector EAElementSelector EAManager AppendActionCard AppendActionRow DBQueryOriginAc- tionRow	EA	self, state

	DBQueryTargetActionRow FormatActionCard FormatActionRow SelectionActionCard SelectionActionRow StatusActionCard StatusActionRow TableQueryActionCard TableQueryActionRow TemporalActionCard TemporalActionRow VisibilityActionCard VisibilityActionRow EventRow EventsCard TableColumnSelector TableSelector main DBQueryActionCard		
onElementAdded2-Group	Group	Main ElementsManager CRSelector SectionsManager GroupsManager	element_id, group_id
onElementCreated	ElementsManager	FormView SectionManager GroupsManager main DropBoxElementsManager EAManager	Element, parent_id, section_id
onElement-GroupChanged	FormView	GroupsManager	element, group_id

	PropertyTabActions		
onElementLabelChanged	PropertyTabActions	GroupsManager main CRSelector Group GroupsManager EAManager	Element, new_label
onElementMoved	BaseElement	FormView main	Element (class)
onElementMoveStoped	BaseElement	FormView	element
onElementRemoved	ElementsManager	FormView, SectionManager, main, CRSelector, Group, GroupsManager, EAElementSelector, DropBoxElementsManager, RepeatablesElementsManager, TableElementMenu	Element_id, type, group_id, isRepeatable
onElementRemovedFromGroup	Group	SectionsManager, ElementsManager, CRSelector, GroupsManager, main	element_id
onElementRenamed	PropertyTabActions	SectionManager, main, EAManager, EAElementSelector, ElementsManager	Element (class), new_name
onElementResized	BaseElement	FormView, main, RadioElement, CheckBoxElement	Element (class), width, height
onElementResizeStoped	BaseElement	FormView	element
onElementSectionChanged	PropertyTabActions	SectionManager, main, EAManager	element_id, section_id
onElementSelected	BaseElement	FormView	Element (class), focus_scroll?
onElementShiftSelected	BaseElement	FormView	Element (class)
onElementsLoadEnded	Json2Form	EAManager, GroupsManager, EAManager	-
onError	Main FormView DBFields2UIModal FieldSelectionModal	Main	string

4. Description and Source Code

	OpenFormModal		
onGroupCreated	GroupsManager	GroupsView, main, FormView, SectionsManager, EAManager	group_id, group_name
onGroupRemoved	Group	GroupsView, SectionsManager, FormView, GroupsManager, ElementsManager, main, EAManager, EAElementSelector	group_id
onGroupRenamed	Group	main, FormView, SectionsManager	group_id, new_name
onGroup-TypeChanged	Group	FormView	group_id, group_type
onLoadEnded	main		
onLoadStarted	main	EAManager, GroupsManager, EAManager	-
onLockEAList	FormView	EAManager	-
onLockElement	Lock	BaseElement	element_id
onLockGroupsList	FormView	GroupsManager	-
onMainTabChanged	main	EAManager, FormView, ListView, GroupsView	tab_id
onMouseGrid-Changed	Grid	BaseElement	delta
onMultiMoveStoped	MultiSelection	main	top, left
onPageAdded	PagesManager	FormView, main, Grid, BaseElement (10)	page_number, Page
onPageRemoved	PagesManager	main, BaseElement, ElementsManager, BaseElement, FormView, EAManager	Page, page_number
onPagesChanged	PagesManager FormView	main, PagesCounterDisplay	
onPagesScrolled	PagesScroller FormView	PagesCounterDisplay	-
onPagesSwapped	PagesManager	Element	from, to
onProgress	FormView EAManager ListView GroupsView main Json2Form	main	msg

onPropertyBtnClicked	UIPropertiesItem	FormView	input_id, new_value
onPropertyChanged	UIPropertiesItem	FormView, main	input_id, new_value
onRepeatableCreated	RepeatablesElementsManager	EAManager	element_id
onRepeatableRemoved	RepeatablesElementsManager	EAElementSelector, EAManager	element_id
onRequireDBFieldSelection	FormView Group	Main	callback function
onSaved	main	ItemsModal	-
onSectionCreated	SectionsManager	FormView, main, ListView, EAManager	id, name
onSectionItemMoved	Section, SectionsManager	SectionManager, ElementsManager, EAManager	element_id, target_id
onSectionNameChanged	Section	FormView, main, SectionsManager, EAManager	section_id, new_name
onSectionRemoved	SectionsManager	FormView, main, ElementsManager, ListView, EAManager	section_id
onStuffDone	FormView EAView GroupsView ListView ElementsManager PagesManager SectionsManager UIFormElementsMenu UIPropertiesMenu	main	string
onTableAdded	TablesManagerModal	TableSelector	-
onTableRemoved	TablesManagerModal	DropBoxElementsManager, ItemsModal	table_asset_name
onUnlockEAList	FormView	EAManager	-
onUnlockElement	Lock	BaseElement	element_id
onUnlockGroupsList	FormView	GroupsManager	-
removeThisElement	Element	ElementsManager	Element (class)

Tab. 11: Designer's signals

4.5 Modals

Except for a few, these all are dynamically generated. All modals are listed and described in tables 12, 13, 14 e 15.

Modal	Description	DG
<i>AutoSaveModal</i>	Auto-save settings.	Yes
<i>DBFields2UIModal</i>	DB elements creator.	Yes
<i>FieldSelectionModal</i>	Database field selector.	Yes
<i>ImageModal</i>	Image selector, for Images element.	Yes
<i>ItemsModal</i>	Define items/source for Dropdown elements.	Yes
<i>OpenFormModal</i>	Form select to load/open.	Yes
<i>PagesManagerModal</i>	Pages manager: order, add/remove, background images.	Yes
<i>PatternsModal</i>	To select or specify an input pattern.	Yes
<i>PropertiesModal</i>	Forms properties.	Yes
<i>RasterModal</i>	To set a background image in the current page.	Yes
<i>SectionNameModal</i>	To set the name for the selected section.	Yes
<i>TableModal</i>	Show the contents of a csv table.	Yes
<i>TablesManagerModal</i>	To add/remove csv tables to the form.	Yes
<i>TextModal</i>	Input text/label.	Yes

Tab. 12: Designer Modals

Modal	Description	DG
<i>ChangeModal</i>	Change the name/description of a form.	Yes

Tab. 13: Forms Manager Modals

Modal	Description	DG
<i>AnnexModal</i>	Add/remove annexes to an operation.	Yes
<i>BarcodeModal</i>	To read a barcode from an image/video.	Yes
<i>FoodexModal</i>	Select foodex name/code.	No
<i>SignModal</i>	To sign an operation with a smart card.	No

Tab. 14: SForm Modals

Modal	Description	DG
<i>AreYouSureModal</i>	Are you sure dialogue (fig. 4).	Yes
<i>ErrorModal</i>	Error dialogue (fig. 5).	Yes
<i>Modal</i>	Master class for all DG modals.	Yes
<i>WarningModal</i>	Warning dialogue (fig. 6).	Yes

Tab. 15: Global Modals

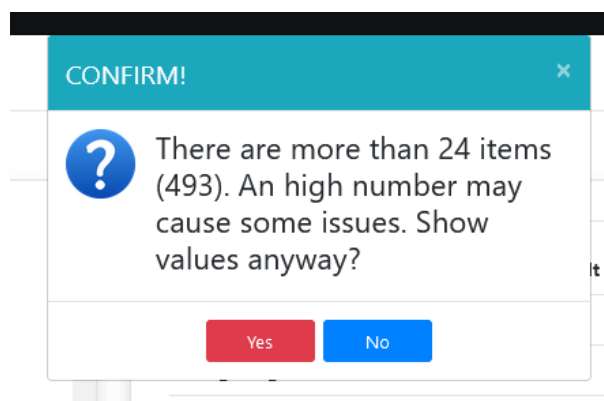


Fig. 4: Are you sure modal

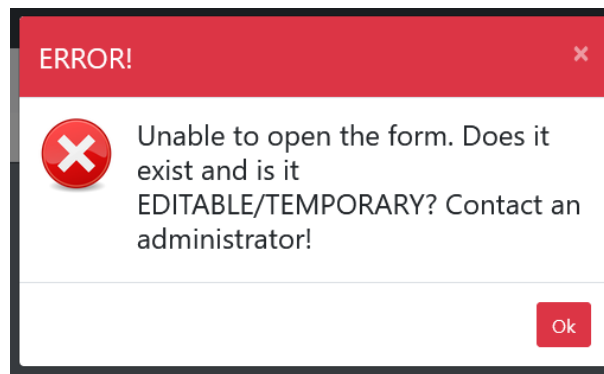


Fig. 5: Error modal

4.6 Designer

The *Designer* or *Editor* is where one can build and edit forms. Fig. 7 presents the main view of the designer.

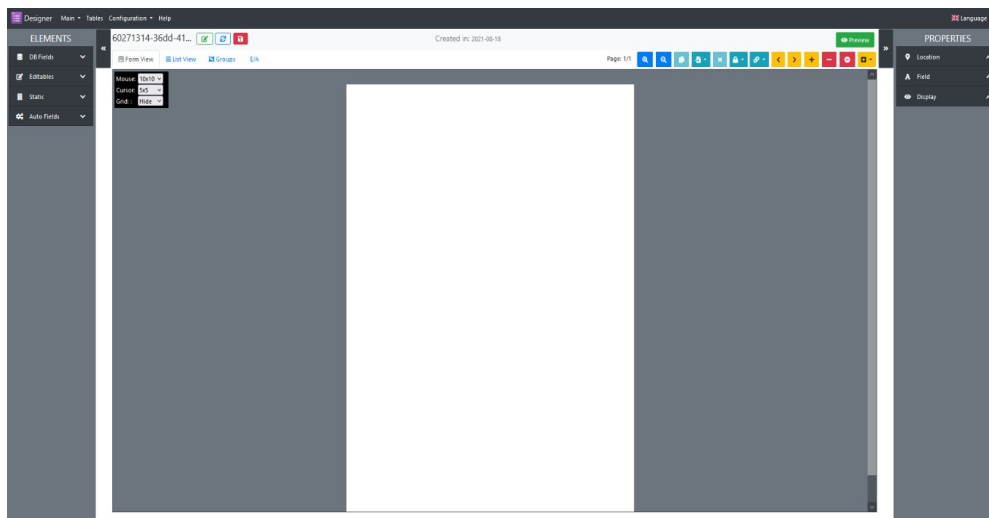


Fig. 7: The designer application

The designer is composed by 6 main scripts:

<i>Element</i>	<i>Description</i>
main.js	The main script. Responsible for load/save operations and start/end the application.
EAViews.js	Main script for the E/A system.
FormView.js	Main script for designing the Form View of the form.
GroupsView.js	Main script for the Groups system.
ListView.js	Main script for setting the List View of the form.
Context.js	The global object.

Tab. 16: Designer's main scripts

4.6.1 Elements

Figure 8 depicts all elements available. The *DB Elements* (see section 4.6.1.14) a user created and therefore empty at the start.

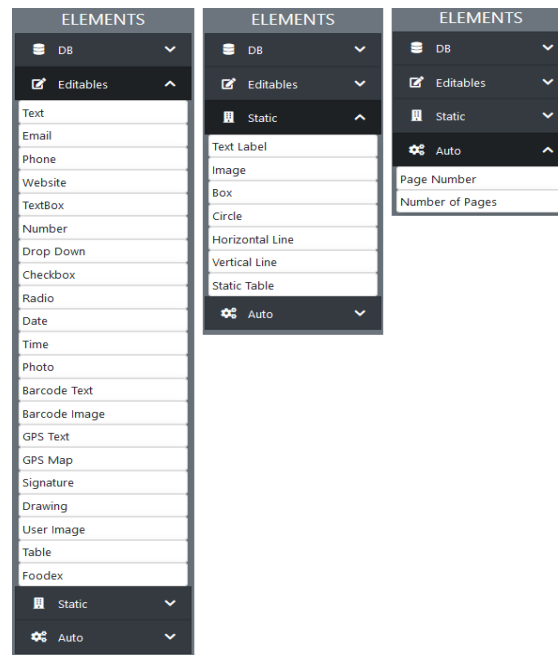


Fig. 8: The different elements available at the Elements sidebar

There are four types of elements:

- **DB**
 - *DB* (database) elements are automatically created by selecting specific fields from a database.
- **Editables** (table 17)
 - are elements that interacting either either with the user or with the E/A system. It allows the user, for example, to:
 - input or select values;
 - select or take pictures;
 - draw;
 - gps;
 - read barcodes;
 - ...
- **Static** (table 18)
 - are elements that can't be changed by the user of the form.
- **Auto** (table 19)
 - are elements that are not directly controlled by the user, and whose values are automatically set depending on the situation.

Element	Description
Text	-
Email	-
Phone	-
Website	-
TextBox	Input extensive text with multiple lines
Number	-
Drop Down / List	Select one from a group of items
Checkbox	Select one or more from a group of options
Radio	Select just one from a group of options
Date	Input date with format: DD-MM-YY
Time	Input time with format HH:MM
Photo	Take a photograph
Barcode Text	Input or take the barcode
Barcode Image	Input or take the barcode and its picture
GPS Text	Input the GPS in text format
GPS Map	Input the GPS in map format
Signature	Draw a signature
Drawing	-
User Image	Select a picture from local storage
Table	Dynamic table, where each cell is an input (textbox)
Foodex	Foodex name or code

Tab. 17: Editable elements

Element	Description
Text Label	-
Image	-
Box	-
Circle	Basically a box or rectangle with rounded corners
Horizontal Line	-
Vertical Line	-
Static Table	Simple non-dynamic table without any input, other than the ones the designer manually places in each cell.

Tab. 18: Static elements

Element	Description
Page Number	Current page number
Number of Pages	Total number of pages

Tab. 19: Auto elements

In the designer, all element are children of the *BaseElement* class. The elements factory and manager is the *ElementsManager*.

elements/constants.js is where the elements' main properties are defined (**ELEMENTS_TYPE**). The **ELEMENTS_TYPE** object lists all elements and each element is defined by 5 fields:

Modal	Description
name	String with the same name as the element.
dimensions	Default dimensions of the element. The dragger object helper also uses these dimensions.
visibility	Which properties are visible.
availability	Where the element is available.
validations	Available validations for this element.

Tab. 20: Elements fields

4.6.1.1 Dimensions

The dimensions defined in this fields, are used both as the default size of the element once in the page, but also for the jquery-ui draggable's helper (**UIFormElementsItem**)

4.6.1.2 Properties / Visibility

The visibility of the properties (see fig. 9) is determined by the visibility field of the **ELEMENTS_TYPE**. This field has 4 other fields which are listed in the first row of table 21.

Group	Property	Description
location	Page	Page number, starting from 0
	Section	Section name
	LV Header	Is a header element (List View only)?
field	ID	Element's ID
	Name	Element's name
	Label/Text	-
	Show Label	Show label?
	Default Value	-
	Enabled	Enable by default?
	Crossed	Crossed by default?
	Placeholder	-
	Max Length	Max characters
	Max Value	Max numeric value
	Min Value	Min numeric value
	Step	Numeric step
	Uppercase	Automatically convert letters to uppercase?
	Required	Required field?
	Checked	Is it checked by default?
	Items	Dropdown/list items
	Database	Database field [deprected]
	Group	Group to which the checkbox/radio element belongs to
	Pattern	Input pattern
	Image Url	-
display	Visible	Is visible?
	Z-Index	-
	Top	Top coordinate (px)
	Left	Left coordinate (px)
	Width	Element's width (px)
	Height	Element's height (px)
	Font	Font style
	Font Size	Font size (default: 16px, 1em)

	Style	Normal or italic
	Decoration	Overline, line-through, underline
	Weight	Bold or normal
	H. Alignment	Left, right, center, justify
	Color	RGB color
	Back Color	RGB color
	Back Alpha	Alpha for the back color
	Borders	All, up, down, left, right
	Border Style	Solid, dashed, dotted, inset
	Border Width	In px
	Border Radius	In px
	Rotation	Element's rotation
groups	[reserved]	-

Tab. 21: List of all available properties



Fig. 9: The Properties sidebar, depending on element's selection

The `ELEMENTS_TYPE.visibility` value is the hexadecimal representation of the binary number resulting from table 22.

The properties of each element is set by `setPropertiesVisibility` in *UIPropertiesMenu.js*:

Which is called:

4.6.1.3 Availability

26

4.6.1.5 Element ID

The element ID is create through the following function in the ElementsManager class (the elements' factory):

```
generateID(type) {
  if (ElementsManager.counter.hasOwnProperty(type)) {
    ElementsManager.counter[type] += 1;
  } else {
    ElementsManager.counter[type] = 0;
  }
  return type.toLowerCase() + '_' + ElementsManager.counter[type];
}
```

So, a *TEXT* element, would have an ID of the form: **text_X**, where X corresponds to a number, which comes from a static counter in the factory class, that keeps track of the number of elements of a certain type. This counter is also saved and restored when the form is saved/opened, which means, there's no danger of multiple elements with the same ID.

4.6.1.6 Move / Resize

All elements are movable and resizable, except Tables, which are only movable. The moving and resize are done through jquery-ui's *draggable* and *resizable*. Elements also can be moved through the arrow keys or between pages through the context menu (*FormView.js*). All these operations are defined in the *BaseElement* class.

The movement and resizable intervals are dependent on the dynamic grid value (see section 4.6.4), which are defined in the Context.

```
grid: [self.context.grid, self.context.grid],
```

Table elements, both the dynamic and the static are not resizable.

4.6.1.7 Selection

Elements are selected either individually or in group using shift select:

```
this.dom.addEventListener('click', function(event) {
  if (event.shiftKey) {
    if (self.isSelected) {
      self.context.signals.onElementShiftUnSelected.dispatch(self);
    } else {
      self.context.signals.onElementShiftSelected.dispatch(self);
    }
  } else {
    self.context.signals.onElementSelected.dispatch(self, false);
  }
})
```

or using the selection marquee, defined by *MutiSelection* and *SelectionMarquee* classes. The *SelectionMarquee* is responsible for drawing the selection rectangle and get all selectable elements inside. The *MutiSelection* receives these elements, parent them to itself, allows to move all of them as one. The coordinates of each element to be the subtraction between the coordinates of the element with the coordinates of the multiselection:

```
this.dom.appendChild(elements[i].dom);
elements[i].dom.style.left = parseInt(elements[i].dom.style.left) -
m_left;
elements[i].dom.style.top = parseInt(elements[i].dom.style.top) -
m_top;
```

Once deselected, the elements are re-parent to their original page. This return requires the coordinates of each element to be the sum between the relative coordinates of the element with the absolute coordinates of the multiselection:

```
const e1 = this.page.appendChild(element.dom);
e1.style.left = parseInt(e1.style.left) + parseInt(this.-
dom.style.left);
e1.style.top = parseInt(e1.style.top) + parseInt(this.dom.style.top);
```

4.6.1.8 Locking Elements

Locking elements disables the draggable, resizable and selectable properties of the elements. Locked elements are identified by the `bLocked` variable in the *BaseElement* class. These elements are managed by the *Lock* class.

4.6.1.9 Repeatable Elements

Repeatable elements are identified by the `bRep` variable in the *BaseElement* class. Once an element is set as repeatable, then, it will start a chain or link, which is basically an array. A duplicate of this element will be create on every exiting or new pages, with the same properties (except the page number). All these alike-elements are place on the same chain and managed by the *RepeatablesElementsManager* class, which keeps track of all chains and page operations (add/remove). Every-time an operation is applied to an element (`ELEMENT_TYPE.X.availability.repeatable`), such as clone, delete, move, resizeable or changes in any property, it checks if `bRep` is true. If so, get repeat the same operation to all element in the chain.

Example o cascading an operation to all elements in a chain, in this case the width property (from *PropertyTabActions.js*):

```
case PROPERTIES_ID.WIDTHPROPERTY:
  _widthProp(context, selected_elements[0], value);
  if (selected_elements[0].isRepeatable()) {
    const chain = context.repeatables_manager.getChain(selected_elements[0].dom.id);
    chain.forEach(element_id => {
      if (element_id === selected_elements[0].dom.id) return false;
      const _element = context.elements_manager.getElement(element_id);
      _widthProp(context, _element, value);
      _element.props[PROPERTIES_ID.WIDTHPROPERTY] = value;
    })
  }
}
```

4.6.1.10 Copying / Cloning Elements

Copying or cloning elements is done through the context menu, where elements can be clone to current page or to the next or previous page. These actions are defined in the *FormView* class.

Cloning procedure - single element:

- 1 props \leftarrow properties of the selected element;
- 2 type \leftarrow type of the selected element;
- 3 create new element
 - 3.1 new_top = top(selected_element) + delta
 - 3.2 new_left = left(selected_element) + delta
 - 3.3 new_element \leftarrow elements_manager(type, props, new_top, new_left)
 - 3.4 if type = DROPDOWN:
 - 3.4.1 clone data
 - 3.5 if type = CHECKBOX | RADIO:
 - 3.5.1 set new element in the same group
- 4 select newly cloned element.

The cloning procedure for multiple elements is simply a loop that clones all selected elements and in the end, selects all the cloned elements.

4.6.1.11 Copy/Paste – the Brush

The *Brush* class is the mainly responsible for copying and pasting properties from one element to another or to a group of elements.

The process is quite simple:

- 1 select one or more elements;
- 2 select brush
 - 2.1 the *Brush* class will makes a copy of all relevant properties of the first selected element;
- 3 select one or more elements to paste the properties
 - 3.1 the *Brush* class will sets the properties of all relevant properties

4.6.1.12 Data sources

Data source refers to 2 types of elements: *dropdowns/lists* and *DB elements*.

4.6.1.13 Dropdowns

There are 3 sources for list elements:

- manual data;
- column from some *csv* table;
- column from a database table.

The *ItemsModal* class is where this selection is set.

4.6.1.14 DB Elements

The system is ready to support *MariaDB/MySQL*, *PostgreSQL* and *Oracle* as the source for these elements. In *settings.py*:

```
DATABASES = {
    'default': env.db('IDRISK_URL'),
    'asae': env.db('ASAE_URL'),
    'world': env.db('WORLD_URL'),
    'postgreDB': POSTGRE,
    'oracleDB': env.db('ORACLE_URL'),
}

# New elements and field<->element links can only be created from
these databases
FIELDS_ORIGIN_DATABASES = {
    'world': {'NAME': 'world', 'ENGINE': 'mysql'},
    'asae': {'NAME': 'asae', 'ENGINE': 'mysql'},
    'postgreDB': {'NAME': 'postgreDB', 'ENGINE': 'postgresql'},
    'oracleDB': {'NAME': 'oracleDB', 'ENGINE': 'oracle'},
}
```

In *databases.py* is where the types inference, tables and fields informations are determine.

All different types of all databases, are grouped in 5 types, where in many instances they overlap each other, since for example, a number can also be text:

```
TYPES = {
  'NUMBER': ['INT', 'TINYINT', 'SMALLINT', 'MEDIUMINT', 'BIGINT',
    'FLOAT', 'DOUBLE', 'DECIMAL', 'INTEGER', 'BIGSERIAL', 'BOOLEAN', 'DOUBLE
    PRECISION', 'MONEY', 'REAL', 'SMALLSERIAL', 'BIT', 'SERIAL', 'NU-
    MERIC', 'NUMBER', 'RAW', 'BINARY_FLOAT', 'BINARY_DOUBLE'],

  'DATE': ['DATE', 'YEAR', 'DATETIME', 'TIMESTAMP'],

  'TIME': ['TIME', 'DATETIME', 'TIMESTAMP', 'INTERVAL'],

  'DROPDOWN': ['ENUM'],

  'TEXT': ['CHAR', 'VARCHAR', 'BLOB', 'TEXT', 'TINYBLOB', 'TINY-
    TEXT', 'MEDIUMBLOB', 'MEDIUMTEXT', 'LONGBLOB', 'LONGTEXT', 'ENUM',
    'DATETIME', 'TIMESTAMP', 'POINT', 'GEOMETRY', 'LINESTRING', 'POLY-
    GON', 'MULTIPOINT', 'MULTILINESTRING', 'MULTIPOLYGON', 'GEOMETRYCOL-
    LECTION', 'FLOAT', 'DOUBLE', 'DECIMAL', 'CHARACTER VARYING', 'CHARAC-
    TER', 'BIT VARYING', 'BYTEA', 'CIDR', 'INET', 'CHAR', 'NATIONAL CHARAC-
    TER', 'NATIONAL CHARACTER VARYING', 'MACADDR', 'UUID', 'XML',
    'JSON', 'TSVECTOR', 'TSQUERY', 'ARRAY', 'TXID_SNAPSHOT', 'BOX',
    'CIRCLE', 'POINT', 'LINE', 'LSEG', 'PATH', 'DOUBLE PRECISION',
    'MONEY', 'REAL', 'SMALLSERIAL', 'BIGSERIAL', 'NUMERIC',
    'RAW', 'VARCHAR2', 'CLOB', 'BINARY_FLOAT', 'BINARY_DOUBLE'],
```

Apart from these 5, any field can also be a checkbox or a radio. Therefore, we actually have 7 possible types, with 2 always available.

4.6.2 Pagination System

The page system is composed by 4 files, listed in table 25.

Modal	Description
Page.js	Class representing a A4 page.
PagesCounterDisplay.js	Class used only to keep the current page display up- dated.
PagesManager.js	The factory and manager of pages.
PagesScroller.js	It determines when to signal the system that the pages area was scrolled.

Tab. 25: Pagination System

4.6.3 Zoom

The zoom is performed by the *Zoom* class. It uses the css *transform* to scale the area of the pages:

```
PAGES_AREA.style['transform'] = `scale(${this.context.zoom})`;
```

Since the transform origin is set to (top, left), the scale by itself, would cause to zoom in/out into or from the top left corner of the area. To prevent this and keep the pages centred horizontally, we had to adjust the scroll after the scale:

```
PAGES_AREA.style['transform'] = `scale(${this.context.zoom})`;
const horizontal = ($(TAB_AREA).width() * this.context.zoom - $(PAGES_AREA).width()) / 2;
$(TAB_AREA).animate({scrollLeft: horizontal}, 0);
```

4.6.4 Grid and Snap

The grid system is composed by 2 files, listed in table 26.

Modal	Description
<i>Grid.js</i>	Sets and deals with the events of the grid menu.
<i>GridPage.js</i>	Adds a grid system to a page.

Tab. 26: Grid System

Grid.js contains the events of changes of the grid:

```
// GRID - mouse and cursor movements
GRID_MOUSE_CONFIG.addEventListener("change paste", (e) => {
  context.grid = parseInt(GRID_MOUSE_CONFIG.value);
  context.signals.onMouseGridChanged.dispatch( GRID_MOUSE_CONFIG.value);
});

GRID_CURSOR_CONFIG.addEventListener("change paste", (e) => {
  context.grid_cursor = parseInt(GRID_CURSOR_CONFIG.value);
  context.signals.onCursorGridChanged.dispatch( GRID_CURSOR_CONFIG.value);
});
```


The grid values are stored in the Context:

```
// grid cell size
this.grid = 10;
this.grid_cursor = 5;
```

The signal signals all relevant parties to update their grid snap, in this case *BaseElement* class:

```
this.signal_onMouseGridChanged = context.signals.onMouseGrid-
Changed.add((delta) => {
  $(this.dom).draggable( "option", "grid", [ delta, delta ] );
  if (type !== ELEMENTS_TYPE.TABLE && type !== ELEMENTS_TYPE.STAT-
ICTABLE) $(this.dom).resizable( "option", "grid", [ delta, delta ] );
});

this.signal_onCursorGridChanged = context.signals.onCursorGrid-
Changed.add((delta) => {
  $(this.dom).draggable( "option", "grid", [ delta, delta ] );
  if (type !== ELEMENTS_TYPE.TABLE && type !== ELEMENTS_TYPE.STAT-
ICTABLE) $(this.dom).resizable( "option", "grid", [ delta, delta ] );
});
```

Snapping elements to the grid, depends not only on the grid size, but also on the current zoom.

It starts on dropping an element into the page (*FormView.js*):

```
// snap to grid
const grid_s = context.grid * self.context.zoom;
const _top_drop = Math.round((ui.position.top - rect.top) /
grid_s) * grid_s;
const _top_left = Math.round((ui.position.left - rect.left) /
grid_s) * grid_s;
const top = Math.round(_top_drop / self.context.zoom);
const left = Math.round(_top_left / self.context.zoom);
```

In *BaseElement.js*, the draggable and resizable are also set to consider the zoom:

```

$(this.dom).draggable({
  cancel: null,
  grid: [self.context.grid, self.context.grid],
  zIndex: 20000,
  drag: function( event, ui ) {
    // snap to grid
    const grid_s = self.context.grid * self.context.zoom;
    const _top_drop = Math.round(ui.position.top / grid_s) *
grid_s;
    const _top_left = Math.round(ui.position.left / grid_s) *
grid_s;

    ui.position.top = Math.round(_top_drop / self.context.zoom);
    ui.position.left = Math.round(_top_left / self.context.zoom);

    // keep it inside the parent
    if (ui.position.left < 0)
      ui.position.left = 0;
    if (ui.position.left + $(this).width() > width)
      ui.position.left = width - $(this).width();
    if (ui.position.top < 0)
      ui.position.top = 0;
    if (ui.position.top + $(this).height() > height)
      ui.position.top = height - $(this).height();

    self.context.signals.onElementMoved.dispatch(self);
  },
  stop: function( event, ui ) {
    self.context.signals.onElementMoveStoped.dispatch(self);
  }
});

```

and:

```

$(this.dom).resizable({
  autoHide: true, // only show when mouse over the Element
  grid: [self.context.grid, self.context.grid],
  zIndex: 20000,
  handles: "e, s, se",
  start: function( event, ui ) {},
  resize: function( event, ui ) {
    var changeWidth = ui.size.width - ui.originalSize.width; // find
change in width
    var newWidth = ui.originalSize.width + changeWidth / self.con-
text.zoom; // adjust new width by our zoomScale
    var changeHeight = ui.size.height - ui.originalSize.height; //
find change in height
    var newHeight = ui.originalSize.height + changeHeight / self.con-
text.zoom; // adjust new height by our zoomScale

    // keep the resizable inside the parent
    if (ui.position.left + ui.size.width < width) {
      ui.size.width = newWidth;
      last_valid_width = newWidth;
    } else {
      ui.size.width = last_valid_width;
    }
  }
});

```

```

    }

    if (ui.position.top + ui.size.height < height) {
        ui.size.height = newHeight;
        last_valid_height = newHeight;
    } else {
        ui.size.height = last_valid_height;
    }

    self.context.signals.onElementResized.dispatch(self, ui.size.width, ui.size.height);
},
stop: function( event, ui ) {
    self.context.signals.onElementResizeStopped.dispatch(self);
}
});

```

In the multiselection case (*MultiSelection.js*):

```

$(this.dom).draggable({
    containment: "parent",
    grid: [context.grid, context.grid],

    crag: function( event, ui ) {
        // snap to grid
        const grid_s = self.context.grid * self.context.zoom;
        const _top_drop = Math.round(ui.position.top / grid_s) * grid_s;
        const _top_left = Math.round(ui.position.left / grid_s) * grid_s;

        ui.position.top = Math.round(_top_drop / self.context.zoom);
        ui.position.left = Math.round(_top_left / self.context.zoom);
    },

    stop: function( event, ui ) {
        self.context.signals.onMultiMoveStopped.dispatch(parseInt(ui.position.top), parseInt(ui.position.left));
    }
});

```

The grid overlay is constant: 50px square.

4.6.5 Load / Save

The load/save system is composed by the 2 files listed in table 27.

Modal	Description
<i>Form2Json.js</i>	Takes a <i>json</i> representing each part of the form (elements, groups, sections, E/As) and make a single json.
<i>Json2Form.js</i>	Takes a <i>json</i> representing the form, and restores all elements, groups, sections, E/As.

Tab. 27: Load/Save System

The load/save actions are initiated in *main*.

The json representation of the form is composed by 16 fields (table 28). Each main part and manager has a `save()` method that returns its respective data object. The same way, it has a `restore(data)` methods that restores its saved state. For example, for the groups are saved/restored simply through (*GroupsManager.js*):

```
save() {
  const data = [];
  for (const gp in this.groups) {
    data.push({id: gp, name:this.groups[gp].name, database:this.-
groups[gp].db_field.input.dom.value, required: this.groups[gp].re-
quired_check.getValue()});
  }
  return data;
}
```

```
restore(data) {
  for (const key in data) {
    const gp = this.createGroup(data[key].name, data[key].id);
    if (data[key].db_field !== undefined)
      gp.db_field.input.dom.value = data[key].database;
    gp.required_check.setValue(data[key].hasOwnProperty('required') ?
data[key].required:'no');
  }
}
```

Field	Description
id	Form's ID.
name	Form's name.
description	A description.
data_created	When was the form created.
rasters	Ordered list of the background images. Pages without images = null.
pages	Ordered list of the pages ID.
dropdowns	
repeatables	
counter	
groups	
elements	
tables	
sections	
sections_items	
sections_groups	
cas	

Tab. 28: Json representing a Form

4.6.6 New / Open Form

The designer template file has a secret field:

```
<input type="hidden" id="form-id-hidden" value="{{form_id}}">
```

which is set by the server with the ID of the form to be opened.

```
def designerID(request, form_id):
    return render(request, 'designer/designer.html', context =
{'form_id': form_id})
```

Main, checks whether or not this hidden field has a value. If yes, then opens the corresponding form, else sets up a new form:

```

if ($('#form-id-hidden').val() !== '') {
  const form_id = ($('#form-id-hidden').val());
  context.properties.is_temp = false;
  isLoading = true;
  openForm(form_id);
} else {
  context.properties.is_temp = true;
  isLoading = false;
  newForm();
}

```

4.7 Preview

Preview allows the user to visualize, test and print the current opened form. By pressing the *Preview* button in the *Designer* application (fig. 10), the form is automatically saved, opens a new tab with the form (fig. 11). It heavily depends on files from *SForm* and *Operations* apps.

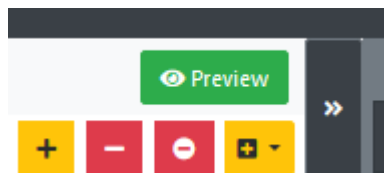


Fig. 10: Preview button in the Designer

```

PREVIEW_BTN.addEventListener('click', function() {
  save(() => {
    window.open(URL_PREVIEW + context.properties.id, "_blank");
  });
});

```

The preview template file has a secret field:

```



```

which is set by the server with the ID of the form to be opened.

```
def previewID(request, form_id):  
    """Preview a specific form."""  
    return render(request, 'preview/preview.html', context =  
{'form_id': form_id})
```

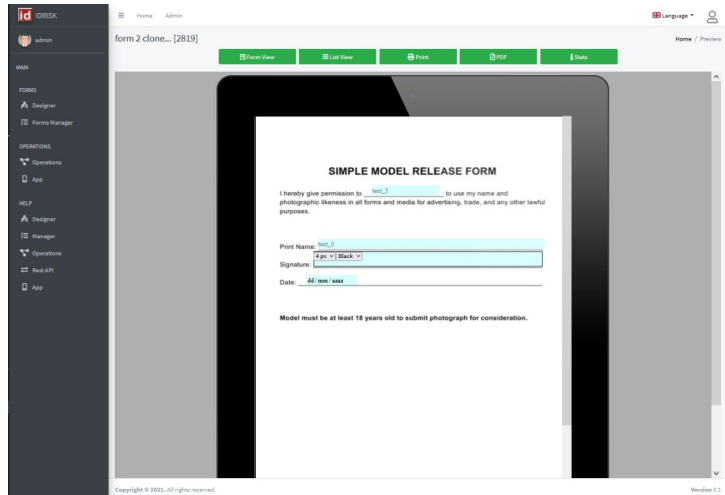


Fig. 11: Main view of the Preview

4.8 Forms Manager

Nothing relevant, except the assets listing are dynamic in the sense, everytime they are listed they are fetched from the server.

4.9 Operations Manager

Like in the Forms Manager, the assets listing are also dynamic.

4.10 Operations Editor

4.11 Inspector's App

The Inspector's app, is Single Page Application (SPA) which was turned into a Progressive Web Application (PWA), as indicated in section 4.15, which means it's designed to make the user feels like it's a native application and also, can operate without being connected to the web.

The application is composed by 6 screens (tab. 29). Figure 12 presents the transition diagram of the application' screens.

Screen	Description
Main	Main menu.
FormSelection	Selection of a Form to be used in the operation from a table with all current <i>IN USE</i> forms.
Manager	Manages all the <i>OPEN/CLOSED</i> operations of the current user.
FillForm	Presents the form, ready to be filled.
OperationsDetails	Name and description of the operation.
ResumeOperationSelection	Selection of an operation to continue inputting from a table with all current <i>OPEN</i> operations.

Tab. 29: App screens

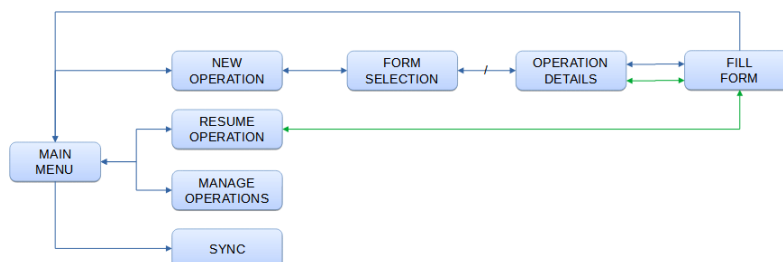


Fig. 12: Screen transitions

The transition between the different screens is triggered by signals. For example, to display the *FillForm* screen:

```
context.signals.onFillForm.dispatch();
```

which triggers:

```
context.signals.onFillForm.add(() => {  
    fill_form.show();  
})
```

Note that, before sending a signal to change screens, the current screen (the content of the screen and the title) must be hidden:

```
$(OPERATION_DETAILS_SECTION).collapse('hide');  
this.title.hide();  
context.signals.onFillForm.dispatch();
```

4.12 Building the FormView / ListView

To use or visualize a Form and by extension an Operation, it's necessary to create the views. *FormViewBuilder* and *ListViewBuilder* are the ones responsible for building the views with all the elements, setting up the *E/As*, groups and so on.

4.12.1 Elements

Each element descends from `BaseElement` class. This parent class, setups common properties, like the visibility and crossed, as well its respective signals to set them up.

```
context.signals.onHidden.add((id) => {
  if (id === this.real_id || id === this.section || id === this.-
group) {
    $(this.dom).hide();
  }
})
```

4.12.2 Synchronization between the Views

The synchronization is done through signals. When an input of the FormView or from the ListView changes, the element in question emits a signal, containing the ID and the new value(s).

A form view element changed:

```
this.context.signals.onFormValueChanged.dispatch(this.real_id,
value);
```

A list view element changed:

```
this.context.signals.onListValueChanged.dispatch(this.real_id,
value);
```

Each element also has a listener implemented, which will update its value(s) if their ID matches.

A form view element, receives signals that a list element changed:

```
context.signals.onListValueChanged.add((id, value) => {
  if (id === this.real_id) {
    this.input.dom.value = value;
  }
})
```

A list view element, receives signals that a form element changed:

```
context.signals.onFormValueChanged.add((id, value) => {
  if (id === this.real_id) {
    this.input.dom.value = value;
  }
})
```

4.12.3 Sections

Sections visualization is only relevant in the *List View*. The process is quite straightforward.

First creates the sections:

```
for (const key in data.sections) {
  sections[data.sections[key].id] = new Section(context, data.sections[key].id, data.sections[key].name);
  parent.appendChild(sections[data.sections[key].id].dom);
}
```

Where to store:

```
let elements_by_section = {};
```

Each element has a section property. Once the element is created it's added to a dictionary where elements per actions are placed.

```
if (elements_by_section.hasOwnProperty(section)) {
  elements_by_section[section].push({id:id.substring(0,id.lastIndexOf('_')), element:element});
} else {
  elements_by_section[section] = [{id:id.substring(0,id.lastIndexOf('_')), element:element}];
}
```

Once all elements are processed, it attach them according in order as established in the designer.

```
// append elements in order
for (const section_id in sections) {
  const order = data.sections_items[section_id];

  for (let i = 0; i < order.length; i++) {
    for (let k=0; k<elements_by_section[section_id].length; k++) {
      if (order[i] === elements_by_section[section_id][k].id) {
        elements_by_section[section_id][k].element.attachTo(sections[section_id].body);
        // founded, so break, unless it's none group, which means
        continue
        // because there can be another none.
        // this happens, when there are radios and checkboxes not
        in an user defined group.
        if (order[i] !== 'none') break;
      }
    }
  }
}
```

Naturally, only sections with element are presented to the user:

```
for (const i in sections) {
  if (sections[i].body.dom.childNodes.length == 0) {
    sections[i].dom.classList.add('collapse');
  }
}
```

4.12.4 Groups

Checkboxes and radio elements are joined into groups. In both the `FormViewBuilder` and `ListViewBuilder`, both these elements have a special treatment, as their are inserted into groups.

While in `FormViewBuilder` the element creation is straightforward:

```

    case ELEMENTS_TYPE.CHECKBOX:
        element = new CheckboxElement(context, data.elements[i].props,
id, type.name);
        break;
    case ELEMENTS_TYPE.RADIO:
        element = new RadioElement(context, data.elements[i].props, id,
type.name);
        break;

```

In `ListViewBilder`, the process is more complicated, since dom has be considered, i.e., elements of the same group must be siblings in the dom:

```

case ELEMENTS_TYPE.CHECKBOX:
{
    let ch_id = id;
    if (!check_groups.hasOwnProperty(group)) {
        const base = new ListBaseElement(context)
        const new_group = new Div();
        new_group.setId('radio-' + group);
        new_group.dom.innerHTML = getGroupName(group, data.groups);
        new_group.addClass("mb-2");
        new_group.attachTo(base.body);
        check_groups[group] = base;
        element = check_groups[group];
        id = group + "_gp-check-listview";
        section = data.sections_groups[group].section;
    }
    const r_ele = new CheckboxElement(context, data.elements[i].props, ch_id, type.name);
    r_ele.attachTo(check_groups[group].body);
    this.values[data.elements[i].id] = r_ele;
}
break;

```

In the elements themselves, the name attribute of radio element of the same group must be the same:

```
this.input.setAttribute('name', group + ID_FORMVIEW_APPEND);
```

or:

```
this.input.setAttribute('name', group + ID_LISTVIEW_APPEND);
```

4.12.5 E/A System

A E/A is composed by 2 parts: the *Event* and the *Action*. The system is quite simple:

- 1 for each E/A:
 - 1.1 for each event:
 - 1.1.1 get the form and list elements

```
const form_element = document.getElementById(ev.field + ID_FOR-
MVIEW_APPEND);
const list_element = document.getElementById(ev.field +
ID_LISTVIEW_APPEND);
```

- 1.1.2 associate the sub-event to each of those elements

```
switch (ev.event) {
  case EVENTS.onChanged:
    if (this.track.onChanged.includes(ev.field)) return;
    this._setEvents(form_element, list_element, ev.field, 'change',
ref.dispatchChanged, 'value')
    this.track.onChanged.push(ev.field);
    break;
```

where:

```
_setEvents(form_element, list_element, field, event, callback=null,
value) {
  if (form_element) {
    form_element.addEventListener(event, function(e) {
      if (callback) callback(field, e.target[value]);
    })
  }
  if (list_element) {
    list_element.addEventListener(event, function(e) {
      if (callback) callback(field, e.target[value]);
    })
  }
}
```

where the callback function is basically a signal dispatch for the sub-events defined in the next section, to trigger the evaluation of the full event's logical expression. Ex:

```
dispatchChanged = (field, value) => {
  this.context.signals.onEventChanged.dispatch(field, EVENTS.on-
  Changed, true);
}
```

4.12.5.1 Events

Since the *Event* can be a composite of events (subevents), which are tracked:

```
data.forEach(ev => {
  this.tracker.push([ev.field, ev.logic, ev.event, false]);
  switch (ev.event) {
    case EVENTS.onChanged:
      context.signals.onEventChanged.add(this.setStatus);
      break;
  }
  ...
}
```

it was just a question of turning these into a logic expression and evaluate it (not with *eval*) everytime a subevent is triggered that changes its logic value:

```
setStatus = (element_id, event, operation = true) => {
  this.tracker.forEach(item => {
    if (item[0] === element_id && item[2] === event) {
      item[3] = operation;
      return;
    }
  })
  this.check();
}
```

```
check() {
  let expression = '(((((';
  this.tracker.forEach(subevent => {
    expression += subevent[1]?subevent[1]:' ';
    expression += ' ' + subevent[3] + ' ';
  })
  expression += '))))';
  expression = expression.replaceAll("and", ") && (");
  expression = expression.replaceAll("or", ") || (");
  const result = looseJsonParse(expression);
  if (result) {
    this.action();
    this.reset();
  }
  return result
}
```

4.12.5.2 Actions

The Action is create by the *ActionFactory*, which is basically a switch:

```
export function ActionsFactory(context, ea_type, ea_id, actions_data)
{
  let action = null;
  switch (ea_type) {
    case EA_TYPE.APPEND.name:
      action = new AppendAction(context, ea_id, actions_data)
      break;
    case EA_TYPE.SELECTION.name:
      action = new SelectionAction(context, ea_id, actions_data)
      break;
    ...
  }
}
```

All actions descend from *Action* class, which only defines a single `execute()` function. For examples, in the Append action, it gets the contents of all elements to be appended and puts it in the targeted element. Naturally with all the checks.


```

execute = () => {
  super.execute();
  if (this.target_field_formview && this.target_field_listview) {
    let appended_str = '';
    this.data.origin_fields.forEach((field, index) => {
      const field_dom = document.getElementById(field + ID_FORMVIEW_APPEND);
      if (field_dom) {
        const target_type_un = field_dom.dataset.type;
        switch (target_type_un) {
          case 'DROPDOWN':
            if (field_dom.value !== '')
              appended_str += field_dom.options[field_dom.selectedIndex].text;
            break;
          default:
            appended_str += field_dom.value;
            if (index < this.data.origin_fields.length-1)
              appended_str += this.connector;
        }
      } else {
        console.error("[AppendAction::execute] - field_dom error");
      }
    })
    this.target_field_formview.value = appended_str;
    this.target_field_listview.value = appended_str;

    // dispatch an event that this element changed
    // required for example for chained events
    var event = new Event('change');
    target_field_formview.dispatchEvent(event);

  } else {
    console.error("[AppendAction::execute] - target_field_X error");
  }
}

```

4.12.6 Load / Save

BaseElement setups the parent `save/restore` functions:

```

save() {
  this.status.id = this.real_id;
  this.status.visible = $(this.dom).css('display') !== 'none';
  this.status.crossed = this.hasClass('cross');
}

async restore(data) {
  this.status = data;
  if (this.status.crossed) this.addClass('cross');
  if (this.status.visible)
    $(this.dom).show();
  else
    $(this.dom).hide();
}

```

Both saving and restoring the inputs of an operation are asynchronous operations (**FormViewBuilder**):

```

save: async function () {
  const data = {elements:[], extras: this.extras, annexes:
this.annexes};
  for (let ele in this.values) {
    //await this.values[ele].save().then(data_2_save => data.ele-
ments.push(data_2_save));
    const data_2_save = await this.values[ele].save();
    data.elements.push(data_2_save);
  }
  //console.warn(data);
  return data;
},

restore: async function (data) {
  if (!data || !data.elements) return;
  await [...Array(data.elements.length)].reduce( (p, _, i) =>
p.then(
  () => {
    return this.values[data.elements[i].id].restore(data.ele-
ments[i]);
  })
, Promise.resolve());

  this.extras = data.extras;
  this.annexes = data.annexes;
},

```

However, only when saving and not restoring a form/operation we have in account this synchronicity. Therefore, then opening a form or operation, certain items, like images, may not be fully loaded yet. This was considered satisfactory, since it doesn't affect neither the operational flux of the system as well the user experience (as much).

Since both forms are synchronized, we only need to save the inputs of one of the views, in this case the *FormsView* was selected. The complexity of the save/restore functions of each element is dependent on their inputs. *TextElement* types as the simplest:

```
save() {
  super.save();
  return new Promise((resolve) => resolve(this.status));
}

async restore(data) {
  super.restore(data);
  return Promise.resolve();
}
```

while elements like *BarcodeImageElement* and *TableElement* are more complicated, due to their complex inputs.

4.13 Validations

Validation are done both by the server and by the client. The validations are divided in *Errors* and *Warnings*. *Errors* happen when a validation condition fails. *Warnings* are not errors but just an attention call, such as, “this element is empty, but it's not required to input any value”.

In the client side, the inputs are validate through the *Validate* and *OpFormData* class. The process is quite simple.

1. In *OpFormData* we get the validations available to a specific element type.

```
const validations = ELEMENTS_TYPE[type].validations;
```

2. Checks which of those respective validations were selected for that specific element in the properties and creates an object containing them.

```
bj.validations = {};
if (validations & 32)
  obj.validations['max-length'] = _element.props[PROPERTIES_ID.-
MAXLENGTHPROPERTY];
if (validations & 16)
  obj.validations['max'] = _element.props[PROPERTIES_ID.MAXPROP-
ERTY];
if (validations & 8)
  obj.validations['min'] = _element.props[PROPERTIES_ID.MINPROP-
ERTY];
if (validations & 4)
  obj.validations['uppercase'] = _element.props[PROPERTIES_ID.UP-
PERCASEPROPERTY]== 'yes'?true:false;
if (validations & 2)
  obj.validations['required'] = _element.props[PROPERTIES_ID.RE-
QUIREDPROPERTY]== 'yes'?true:false;
if (validations & 1)
  obj.validations['pattern'] = _element.props[PROPERTIES_ID.PATTERN-
PROPERTY]== 'Default'?null:_element.props['properties-pattern'];
```

3. Once the validations for each element have being determined, then use the *Validate* class to check which validations are fulfilled. For example for the uppercase, if the element has that validation active, checks if the input value is all uppercase. If not, then adds a corresponding error message.

```
if (validations.hasOwnProperty('uppercase') && validations.upper-
case) { // === 'yes' } {
  if (_value && _value !== _value.toUpperCase()) {
    error_messages.push(Translator.translate(MESSAGE_NO_UPPER-
CASE));
    this.has_errors = true;
  }
}
```

4. Once all elements are validated, the validation modal displays the results (fig. 13).

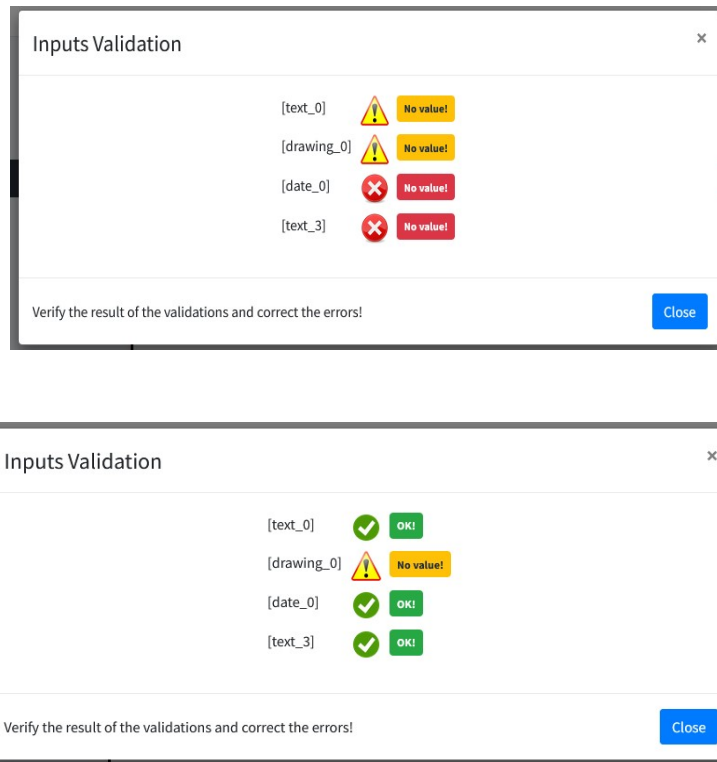


Fig. 13: Inputs validation

For checkboxes and radios, the validations process is different but easier, since it only checks for “*required*” and on a group level.

On the server side, the process is similar. The validation functions are located in *operations/services.py*.

4.14 Printing / Exporting to PDF

The printing process starts in *FormViewBuilder.js* with `printForm()`. It then creates a frame and copies the form’s contents into it. Since some elements required some changes before printing, the elements in the *iframe* are processed by `prepare2Print()`. Despite some changes are done through *css*, some of them cannot be pro-

cessed through that. These include, replacing canvas and map elements with images and setting the values of elements (fig. 14). Example:

```
case ELEMENTS_TYPE.CHECKBOX:
case ELEMENTS_TYPE.RADIO:
  // simple input check
  destination_element.checked = original_element.input.dom.checked;
  resolve();
  break;
```

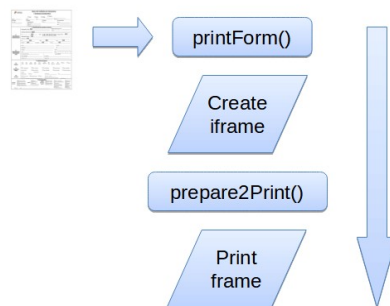


Fig. 14: The printing process

The PDF export works exactly the same, except it turns the *iframe* contents into a canvas, then to an image and finally to a PDF document.

4.15 Offline System

One of the most important requirement was the ability the use any form while off-line. For that end, we turned the Inspector's APP into a PWA application.

4.15.1 Technologies

To ensure the offline functionality we used the *IndexedDB*¹ through the *Dexie.js*² wrapper, which makes the usage of this storage as easy as using a MySQL database. *IndexedDB* is an easy to use *NoSQL* large scale storage system. It allows the storage of any type of data in the browser itself (in *json* format), having as its global limit³ 50% of the disk's free space.

4.15.2 Development

The *SForm*'s offline system consists of three files:

Modal	Description
<i>ServiceWorker.js</i>	Sets the system and captures url requests
<i>offline/offlinedb.js</i>	Deals with <i>IndexedDB</i>
<i>offline/utills.js</i>	Some useful functions

Tab. 30: Elements fields

We used *django-pwa*⁴, which is a Django app that includes a *manifest.json* and Service Worker instance to enable progressive web app behaviour. It was set in *setting-s.py*:

¹ <https://w3c.github.io/IndexedDB/>

² <https://dexie.org/>

³ https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria#storage_limits

⁴ <https://pypi.org/project/django-pwa/>

```

PWA_SERVICE_WORKER_PATH = BASE_DIR / 'sform/static/sform/js/Ser-
viceWorker.js'
PWA_APP_NAME = 'Forms'
PWA_APP_DESCRIPTION = "Forms PWA"
PWA_APP_THEME_COLOR = '#000000'
PWA_APP_BACKGROUND_COLOR = '#ffffff'
PWA_APP_DISPLAY = 'standalone'
PWA_APP_SCOPE = '/sform/'
PWA_APP_ORIENTATION = 'any'
PWA_APP_START_URL = '/sform/'
PWA_APP_STATUS_BAR_COLOR = 'default'
PWA_APP_ICONS = [
    {
        'src': '../static/images/icons/android-chrome-192x192.png',
        'sizes': '192x192'
    }
]
PWA_APP_ICONS_APPLE = [
    {
        'src': '../static/images/icons/apple-touch-icon.png',
        'sizes': '180x180'
    }
]
PWA_APP_SPLASH_SCREEN = [
    {
        'src': '../static/images/icons/android-chrome-192x192.png',
        'media': '(device-width: 320px) and (device-height: 568px) and
(-webkit-device-pixel-ratio: 2)'
    }
]
PWA_APP_DIR = 'ltr'
PWA_APP_LANG = 'en-US'

```

We also had to add the `{% load pwa %}` tag to the *SForm* template file.

4.15.2.1 Stores

We used four stores to save local data, as illustrated in figure 15.



Fig. 15: IndexedDB Stores

Store	Indices	Fields
Forms	id isDeprecated	id name description form isDeprecated
Forms Assets	id form_id asset	id form_id name asset data
Operations	id form status_name isPendingDeletion [status_name+isPendingDeletion]	id name description form form_name data_creation status_name operation_data isPendingDeletion date_updated updated_by
Operations As-sets	id operation_id asset	id operations name asset date type type_name is_annex data

Tab. 31: Stores

4.15.2.2 App and ServiceWorker comms

The service worker runs independently in the browser background and intercepts all relevant network requests.

```
// Serve from Cache
self.addEventListener('fetch', event => {
  // Prevent the default, and handle the request ourselves.
  event.respondWith(async function() {

    // Try to get the response from a cache.
    const cachedResponse = await caches.match(event.request);
    if (cachedResponse) {
      return cachedResponse;
    }
    // --- ONLINE ---
    try {
      if (event.request.url.includes(URL_LIST_IN_USE_FORMS)) {

        ...
      }
    } catch (e) {
      // --- OFFLINE ---
      if (event.request.url.includes(URL_LIST_IN_USE_FORMS)) {

        ...
      }
    }
  })
}
```

It's also where the list of files to be cached are defined and cached.

```
// Cache on install
self.addEventListener("install", event => {
  this.skipWaiting();
  event.waitUntil(
    caches.open(staticCacheName)
      .then(cache => {
        return cache.addAll(filesToCache);
      })
  )
});
```

A broadcast channel was set to allow direct communication between the app and the service worker. The app uses this channel to transmit the security token (required for all posts and gets):

```
const broadcast = new BroadcastChannel('sync-channel');
broadcast.postMessage({msg: 'token', value: getCookie('csrftoken')});
```

and to trigger a full synchronization:

```
broadcast.postMessage('sync');
```

in the *serviceworker*:

```
broadcast.addEventListener("message", async function(event) {
  // sync the databases
  if (event.data && event.data === 'sync') {
    try {
      broadcast.postMessage('sync_started');
      await offline.fullOperationsSynchronization();
      await offline.fullFormsSynchronization();
      broadcast.postMessage('sync_finished');
    } catch {
      broadcast.postMessage('sync_finished');
    }
  }
  // get and set the CSRF token for the POST calls
  else if (event.data && event.data.msg && event.data.msg === 'token') {
    offline.setToken(event.data.value);
  }
});
```

4.15.2.3 Synchronization

For the app to operate when there's not network, the app should keep synchronized with the server as much as possible. When the user opens the app our visits the main screen, it will automatically attempt to fully synchronize with the server. In this case a visual indication will be presented to the user on this fact (fig. 16).

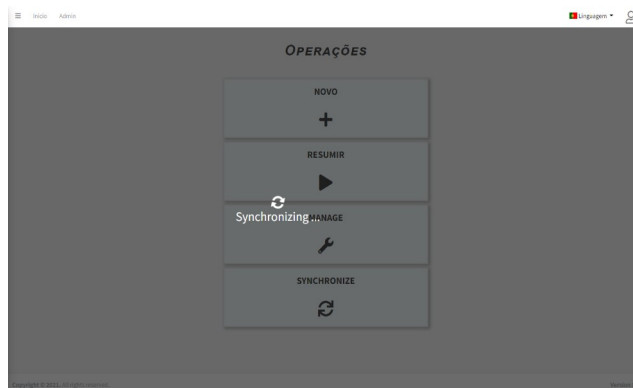


Fig. 16: App synchronizing

The user also has the option by manually triggering this synchronization by clicking the “Synchronization” button in the main screen. If the App is online, then it will sync, otherwise, it will present a warning message to the user (fig. 17).

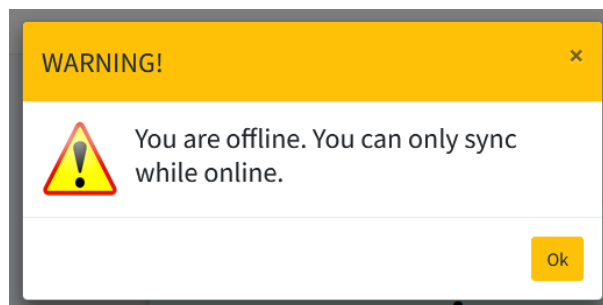


Fig. 17: Offline warning

If online, the service worker will forward the request and will update the stores accordingly. For example, if online and the user requests a list of all forms, proceeds with the request and updates the local storage with the response:

```
// LIST OF FORMS
// if network and user fetching list of IN USE forms
// then synchronize remote with local
if (event.request.url.includes(URL_LIST_IN_USE_FORMS)) {
  const response = await fetch(event.request);
  const forms = await response.clone().json();
  await offline.syncForms(forms);

  return response;
}
```

If on the other hand, the user is offline, creates a response based on the data stored locally.

```
// LIST OF FORMS
if (event.request.url.includes(URL_LIST_IN_USE_FORMS)) {
  const forms = await offline.getListOfAllValidLocalForms();
  const fakeResponse = new Response(JSON.stringify(forms));
  return fakeResponse; // .clone(); // clone????
}
```

In a global synchronization, both the forms and operations are synchronized. The synchronization is mainly based on dates.

```
await offline.fullOperationsSynchronization();
await offline.fullFormsSynchronization();
```

Operations were synchronized first, in order to allow the elimination of deprecated forms. This is a requirement, since, a form cannot be eliminated from local storage if it's being used by a non-complete operation.

```
async function fullOperationsSynchronization(remote_operations =
null) {
  // delete all local ops marked for deletion, and their assets
  await deleteLocalPendingDeletionOperations();
  // cases 5
  await syncFullLocalOperations();
  // cases 1,2,3,4
  return await syncOperations(remote_operations);
}
```

```
/**
 * Fetches all IN USE forms and synchronizes local forms and their
assets.
 * One side synchronization.
 */

async function fullFormsSynchronization() {
  const response = await fetch(URL_LIST_IN_USE_FORMS);
  if (response.ok) {
    const remote_forms = await response.json();
    await syncForms(remote_forms);
  }
}
```

When offline, new operations and new assets, will have a negative ID. We have 5 possible cases do consider when synchronizing an operation, which are listed in table 32.

Case	Remote	Local	Action	Description
1	OP	OP	-	Fetches remote operations (non-completed). Local orphan operations (local operations with ID > 0 that don't exist in server) are deleted. Compares each remote operation with its local, by its last update date. Dates are equal, so no action required.
2	OP+	OP	$R \rightarrow L$	Fetches remote operations (non-completed). Local orphan operations (local operations with ID > 0 that don't exist in server) are deleted. Compares each remote operation with its local, by its last update date. Remote more recent than local \Rightarrow update local operation with remote data and its assets.
3	OP	OP+	$L \rightarrow R \rightarrow L$	Fetches remote operations (non-completed). Local orphan operations (local operations with ID > 0 that don't exist in server) are deleted. Compares each remote operation with its local, by its last update date. Local more recent than remote \Rightarrow update remote operation with local data and update remote assets. All assets with negative IDs, are then updated with their new IDs.
4	OP	-	$R \rightarrow L$	Fetches remote operations (non-completed). Local orphan operations (local operations with ID > 0 that don't exist in server) are deleted. Compares each remote operation with its local, by its last update date. Remote doesn't exist locally \Rightarrow create local operation and all its assets.
5	-	OP	$L \rightarrow R \rightarrow L$	The operation was created entirely locally. Operation ID < 0 and assets ID < 0. Both the operation and assets will be saved on the server and their local IDs updated with the new IDs.

Tab. 32: Cases when synchronizing operations

Synchronizing forms is simpler, since synchronization is only done from remote to local. It follows a 3 step process:

1. fetch all “*IN USE*” forms and their assets not yet in local storage;
2. all non *Deprecated* forms that exists locally but not remotely, means that they are not longer in use and they are marked as *Deprecated*, and if they are not in use by any operation, then deletes them and their assets;
3. eliminate all forms marked as *Deprecated* and their assets.

Notes:

- orphan assets in the server are not a problem, since, these are automatically delete every time an operation is saved/updated.

4.16 Barcode Reader

The barcode reader is capable of reading a barcode either from a picture or from a camera in real-time. Note, that this real time capability is only available in PCs. The barcode modal is presented in figure 18.

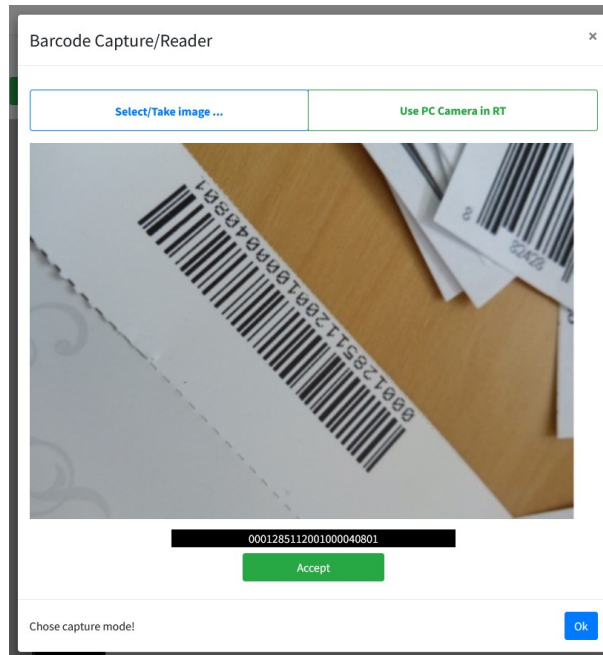


Fig. 18: Barcode Modal

The detection works independently of the direction and inclination of the barcode. It's however dependent on the lighting and the uniqueness of the barcode in the image, i.e., only one barcode should be visible in the image.

By default it's set to read the following code formats:

- EAN
- CODE 128

- CODE 39
- UPC
- I2of5
- 2of5
- CODE 93
- CODABAR

These are set in *BarcodeDecoder.js*:

```
decoder: {
  readers : [{
    format: "code_128_reader",
    config: {}
  }, {
    format: "ean_reader",
    config: {}
  }, {
    format: "code_39_reader",
    config: {}
  }, {
    format: "upc_reader",
    config: {}
  }, {
    format: "codabar_reader",
    config: {}
  }, {
    format: "i2of5_reader",
    config: {}
  }, {
    format: "2of5_reader",
    config: {}
  }, {
    format: "code_93_reader",
    config: {}
  }]
}
```

The library allows more, however there are some caveats that need to be considered:

- more decoders, means more clashes and false positives;
- the order is also important;
- EAN-2 and EAN-5 require their activation and if they are, then reading EAN-13 is not longer possible.

```

decoder: {
  readers: [{
    format: "ean_reader",
    config: {
      supplements: [
        'ean_5_reader', 'ean_2_reader'
      ]
    }
  }]
}
}

```

4.17 Smart Card Signature

The signing modal allows a PDF of the operation to be signed by multiple signatures and also to select the page and location of the signature (see fig. 19).

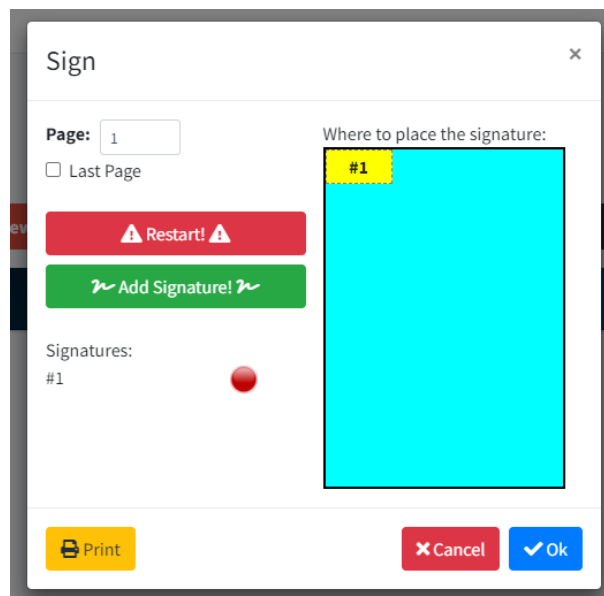


Fig. 19: Signing Modal

The signing process ends up by creating a signed PDF file, named **DOC_SIGNED.pdf**. If another signature process starts, it will keep this file and create a new one named **DOC_SIGNED_*.pdf**, where *, represents a random alphanumeric sequence.

4.17.1 Desktop/Windows

To sign under windows, a Chrome extension and a c++ application⁵ was developed. This application serves as the communication bridge between the SDK from *Autenticação.gov*⁶ and the extension.

There are several risks en allowing a web page to communicate outside the browser. In order to minimize those risks, Google imposed several limitations on the native messaging system⁷.

One of those limitations is the max size of a message (1MB) sent by the native application to the browser. This limitation was easily overcome by dividing any message in blocks of inferior size. All communication is done in *json*, and since only text is allowed, the exchange of PDF data is done through *base64*.

The native application receives the message (json format), which has a limit of 4GB and saves the data in a PDF temporary file:

```
// save file in computer
string result = base64_decode(data, false);
ofstream wf((temp_path + FILENAME_2_SIGN).c_str(), ios::binary);
wf << result;
wf.close();
```

Uses the SDK to sign that temporary file:

⁵ <https://github.com/tiago1856/CCPDFSign>

⁶ <https://www.autenticacao.gov.pt/web/guest/cc-aplicacao>

⁷ <https://developer.chrome.com/docs/apps/nativeMessaging/>

```

LOG("Initing EID SDK ...");
eIDMW::PTEID_InitSDK();
LOG("EID SDK initiated!");
sendMessage("status", "SDK READY");
try {
    LOG("Signing file ...");
    sendMessage("status", "SIGNING");
    signPDFFile(1, pos_x, pos_y, "", "", (temp_path +
FILENAME_2_SIGN).c_str(), (temp_path + FILENAME_SIGNED).c_str());
    LOG("File signed!");
    sendMessage("status", "SIGNED");
}
catch (eIDMW::PTEID_Exception ex) {
    std::string error_msg = errorMessage(ex.GetError(), ex.GetMes-
sage());
    LOG("Error: " + error_msg);
    sendMessage("error", error_msg);
    error = true;
}
catch (...)
{
    sendMessage("error", "UNKNOWN ERROR");
    LOG("UNKNOWN ERROR");
}
LOG("Releasing EID SDK ...");
eIDMW::PTEID_ReleaseSDK();
LOG("EID SDK released!");

```

Once signed, reads the file and encodes it into base64:

```

ifstream in((temp_path + FILENAME_SIGNED).c_str(), ios::binary);
in.seekg(0, ios::end);
std::streamoff iSize = in.tellg();
in.seekg(0, ios::beg);

char* pBuff = new char[iSize];
memset(pBuff, 0, sizeof(pBuff));
in.read(pBuff, iSize);
in.close();

std::string xxx(pBuff, pBuff + iSize);
string data2 = base64_encode(xxx, false);

```

Then send chunks of *base64* data to the browser:

```

size_t total_size = data2.length();
// total number of blocks
size_t blocks = (total_size / MAX_BLOCK_SIZE) + 1;
size_t desc = 0;
// divide the message into blocks
for (unsigned int i = 0; i < blocks; i++)
{
    // block size
    size_t delta = (i < blocks - 1) ? MAX_BLOCK_SIZE : total_size -
MAX_BLOCK_SIZE * (blocks - 1);
    // data to send
    string block_message = data2.substr(desc, delta);
    // create json with the data
    string message_2_send = "{\"data\": \"" + block_message + "\"}";
    desc += delta;
    //sendMessage("info", to_string(delta));
    size_t block_size = message_2_send.length();
    //send the 4 bytes of length information
    cout << char(((block_size >> 0) & 0xFF))
        << char(((block_size >> 8) & 0xFF))
        << char(((block_size >> 16) & 0xFF))
        << char(((block_size >> 24) & 0xFF));
    // output our message
    cout << message_2_send;
}

sendMessage("status", "DONE");

```

The Chrome extension is composed by two javascript files: *background.js* and *contentscript.js*. In simpler terms, while the *background* is used to trigger the execution and to communicate with the native application, the *contentscript* is used to communicate between the background and our *webpage*. All communications are done through messaging. For example, *contentscript* \rightarrow *webpage*:

```

// send to js app
chrome.runtime.onMessage.addListener((message, sender,
sendResponse) => {
    window.postMessage({ origin: "NATIVE", content: message }, "*");
    return true;
});

```

and in the webpage:

```

window.addEventListener("message", (event) => {
  // We only accept messages from ourselves and from native
  // here NATIVE also implies native related errors, such as does
  not exist
  if (event.source !== window || event.data.origin !== 'NATIVE') {
    return;
  }
  const key = event.data.content.key;
  const data = event.data.content.data;
  if (key === 'test') {
    // test && error => no native app installed
    if ( event.data.content.result && event.data.content.result
    === "ERROR") {
      self.unlock();
      self.changeStatus(self.current_status, SM_STATUS.RED);
      console.error("ERROR > ", key, event.data.content.message);
      self.context.signals.onWarning.dispatch(Translator.trans-
      late(event.data.content.message));
    }
  }
  else if (key === 'status') {
    console.warn("SIGNING STATUS > ", key, data);
  } else if (key === 'error') {
    self.unlock();
    self.changeStatus(self.current_status, SM_STATUS.RED);
    console.error("ERROR > ", key, data);
    self.context.signals.onWarning.dispatch(data);
  } else if (key === 'result') {
    self.unlock();
    console.log("RESULT > ", key); //, data);
    self.pdf_data = data;
    self.messageOk();
    clearTimeout(self.sign_timeout);
  }
}, false);

```

In the App, the PDF conversion to base64, follows the same logic as the export to PDF, since it uses the same library, where instead of exporting to a file (in *Print.js*):

```

const pdfBytes = await pdfDoc.save();
var file = new File([pdfBytes], filename, {
  type: "application/pdf;charset=utf-8",
});
saveAs(file);

```

it exports to a base64 string:

```

const base64 = await pdfDoc.saveAsBase64();
return [base64, pdfDoc.getPageCount()];

```

4.17.2 Android

4.18 Multi-language support

Since 90% of all applications content are dynamically created, the translation procedure had to be divided into 2 systems: one is done through *Django* tools and another through *JavaScript*.

4.18.1 By Django

Django offers tools that allows to easily insert internationalization and location to applications. This is done through the following steps:

1. In *settings.py*:

```
LOCALE_PATHS = (BASE_DIR / 'locale',)

USE_I18N = True

LANGUAGE_CODE = 'en'

LANGUAGES = [
    ('en', 'English'),
    ('pt', 'Portuguese'),
    ('hr', 'Croatian'),
]
```

2. In the template files, we add the following tag at the top:

```
{% load i18n %}
```

and we use the tag `trans` to indicate all terms to be translated:

```
{% trans "SECTIONS" %}
```

3. Compilation:

```
django-admin makemessages --all --ignore env
```

This step creates *.po* files, containing all terms.

4. Manual translation of the terms in the *.po* files

5. Messages compilation:

```
django-admin compilemessages --ignore=cache --  
ignore=outdated/*/locale
```

The language selection is done through a simple dropdown menu (fig. 20), which is available to all applications.

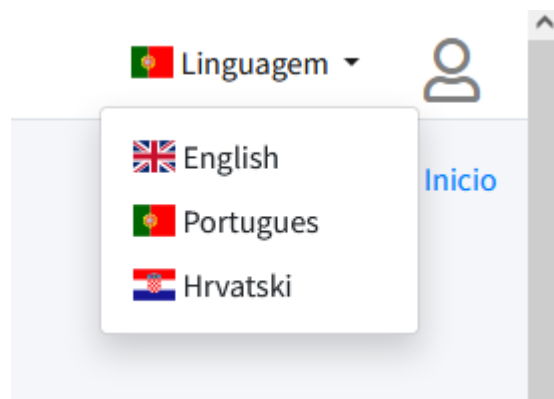


Fig. 20: Language selection

Code:

```
<form id="lang-select-form" action="{% url 'set_language' %}"
method="post">
  {% csrf_token %}
  <input name="next" type="hidden" value="{{ redirect_to }}" />
  <input id="selected-lang-form" name="language" type="hidden"
value="{{ LANGUAGE_CODE }}" />
</form>
```

```
function setFlag(language_code=null) {
  if (!language_code) language_code = $('#selected-lang-
form').val();
  $('#selected-language').addClass('flag-icon');
  $('#selected-language').addClass('flag-icon-' + language_code);
}

$('.language-select').click(function(e) {
  const sprache = $(this).attr('data-lang');
  if (sprache === $('#selected-lang-form').val()) return false;
  context.signals.onAYS.dispatch(Translator.translate("Selecting a
new language will cause the editor to reload. Continue?"), () => {
    setFlag(sprache);
    $('#selected-lang-form').val(sprache);
    $('#lang-select-form').submit();
  });
});
```

4.18.2 By Javascript

The translation is done through the `Translator` class, which offers 2 static methods: `setLanguage` and `translate`.

The class is basically a dictionary composed by all available languages:

```
static dictionary = {
  pt: {
    'file': 'Ficheiro',
    'save': 'Guardar',
    ...
  },
  es: {
    ...
  },
  ...
}
```

The translation is done directly in code thorough:

```
Translator.translate('Adding Elements')
```

The language selection is done in the main script of each application:

```
Translator.setLanguage($('#selected-lang-form').val());
```

where `$('#selected-lang-form').val()` corresponds to the value selected in the language dropdown.

4.19 Template Theme

To accelerate the development and not waist time in the design and implementation of a general style form the applications, a template⁸ was selected. Fortunately, the chosen template has a Django implementation⁹ which made the integration seamless. All it was necessary was the package installation and adding two lines in *settings.py*.

```
INSTALLED_APPS = [  
    'adminlte3',  
    'adminlte3_theme',  
    ...
```

4.20 Adding a New Element Type

There are several steps and considerations when adding a new element X. Here, is presented the most important steps and files to consider. The addition of a new element will directly affect two applications: the *Designer* and the *SForm*.

⁸ <https://adminlte.io/themes/AdminLTE/>

⁹ <https://pypi.org/project/django-adminlte-3/>

4.20.1 In Designer

- 1 Define the element characteristics in `ELEMENT_TYPE` (*elements/constants.js*);

```
X : {
  name: "X",
  dimensions:{},
  visibility: {groups: 0x?, location: 0x?, field:0x?, display:0x?},
  availability: {form: false, list: false, groups: false, ea: false, transfer: false, repeatable: false, lv_header: false},
  validations: 0x?,
},
```

- 2 Define the UI item in *ui/UIFormElementMenu.js*;

```
FFG_Editables.add(new UIFormElementsItem(context, Translator.translate("X"), ELEMENTS_TYPE.X));
```

- 3 If the element can't be represented by any of the elements presented in *elements/elements*, then create one with `BaseElement` as the parent;
- 4 In the `BaseElement`:

- 4.1 If the element can't move or resize, make sure to specify that;

```
if (type !== ELEMENTS_TYPE.TABLE && type !== ELEMENTS_TYPE.STAT-
ICTABLE && type !== ELEMENTS_TYPE.X) this.makeResizable();
```

- 4.2 If the name of element should only be presented when a name is valid, add the element to `setText`;

- 5 Add the element to `ElementsManager::createElement`:

```
case ELEMENTS_TYPE.X:
  new_element = new XElement(type, props, parent, this.context,
left, top, id, DEFAULTBACKGROUNDCOLOR);
  break;
```

- 6 In `PropertyTabActions::nameProp`, add the element if necessary,

- 7 In `ea/constants.js`, add, if necessary, the element to `EVENTS_ELEMENTS_ALLOWED` and `EA_TYPE`.

4.20.2 In *SForm*

- 1 Define the element class both for the Form (`views/formview/elements/`) and List (`views/listview/elements/`) views;
- 2 Add the element to `FormViewBuilder` and `ListViewBuilder`:

```
case ELEMENTS_TYPE.X:  
  element = new XElement(context, data.elements[i].props, id);  
  break;
```

- 3 Add the element to `FormViewBuilder::processElement` and set its print/export output;

5 Security

Besides the usage of frameworks, libraries, plugins and tools with a good dependable and reliable history, we also consider:

- tokens and *CSRF*;
- permissions
- POST for any operation that affects the system.

5.1 Accounts

The default Django signup system was overwritten by the *Accounts* app. It defines a single view:

```
class SignUpView(generic.CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'registration/signup.html'
```

and a simple filter:

```
@register.filter(name='has_group')
def has_group(user, group_name):
    """Filter to be used in templates, to check if a user is in a
    specific group.
    Ex: {% if request.user|has_group:"manager" %}"""
    return user.groups.filter(name=group_name).exists()
```

which is used throughout the template files to limit the access to users:

```
{% if request.user.is_authenticated and request.user|
has_group:"manager" %}
...
{% endif %}
```

5.2 Tokens

All API endpoints available by the Rest API app, both GET and POST requires a security token. This is provided by the *rest_framework.auth_token* app. The generation is unique for each user. Successive calls for the token doesn't changes it.

```
class CustomAuthToken(ObtainAuthToken):
    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data, context={'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        if user.groups.filter(name = 'manager').exists():
            token, created = Token.objects.get_or_create(user=user)
            return Response({
                'token': token.key,
                'user_id': user.pk,
                'email': user.email
            }, status=status.HTTP_200_OK)
        else:
            return Response ({'message': 'You do not have permission to generate a token'}, status=status.HTTP_401_UNAUTHORIZED)
```

The token should be provided on every header:

```
const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8',
  },
  body: JSON.stringify({
    "username": "someuser",
    "password": "somepassword"
  }),
}
fetch('http://34.134.170.7/rest/api/api-token-auth/', options)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error))
```

Unlike all the other applications, all views are class based. Security is set by the authentication class:

```
class ListOperations(generics.ListCreateAPIView):
    authentication_classes = [TokenAuthentication]
    #permission_classes = [IsAuthenticated]
    queryset = Operation.objects.all()
    serializer_class = OperationSerializer
    http_method_names = ['get']
```

Was also added to the rest framework settings, in settings.py:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.TokenAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    )
}
```

5.3 Permissions

Three levels of permissions or groups were defined:

- administrator
- manager
- inspector

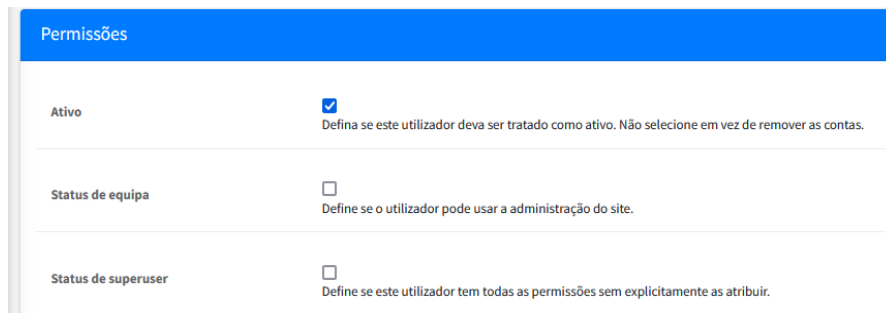
Their access is not only defined in the template files, through the use of tags:

```
{% if request.user.is_authenticated and request.user|
has_group:"manager" %}
...
{% endif %}
```

```
{% if request.user|has_group:"manager" %}
...
{% endif %}
```

```
{% if request.user.is_authenticated %}
    {% trans 'MAIN' %}
{% endif %}
```

but also in the administration back-office (see fig. 21).



Permissões	
Ativo	<input checked="" type="checkbox"/> Define se este utilizador deva ser tratado como ativo. Não selecione em vez de remover as contas.
Status de equipa	<input type="checkbox"/> Define se o utilizador pode usar a administração do site.
Status de superuser	<input type="checkbox"/> Define se este utilizador tem todas as permissões sem explicitamente as atribuir.

Fig. 21: Setting permissions in the administrator page

6 Administration

There are 15 tables available to an administrator. These are described in table 33.

<i>Application</i>	<i>Table</i>	<i>Description</i>
Accounts	<i>User</i>	All users and privileges (see section 5.1)
Auth Token	<i>Tokens</i>	All generated tokens per user (see section 5.2)
Authentication and Authorization	<i>Groups</i>	All groups that are used to define access privileges: admin / manager / ...
Files	<i>Files</i>	Configuration files, data files, application and programs. (see section 6.1)
Designer	<i>Asset types</i>	All assets type: PDF / IMAGE / ...
	<i>Form assets</i>	Assets description, their location, dates and associated forms
	<i>Forms</i>	Forms details and data
	<i>Queries</i>	Queries used in the database query E/A s
	<i>Status</i>	Lists all possible states a form, operation and queries can be: EDITABLE / CLOSED / ...
Django Q	<i>Failed tasks</i>	These tables are related with the automatic tasks django performs every n interval: delete all temps forms / ... (see section 6.2)
	<i>Queued tasks</i>	
	<i>Scheduled tasks</i>	
	<i>Successful tasks</i>	
Operations	<i>Operation assets</i>	Assets description, their location, dates and associated operation
	<i>Operations</i>	Operations details
	<i>Operations Data</i>	Operations inputs

Tab. 33: Back-office tables

6.1 Files

The Files application allows the administrator to keep data and config files updated, by uploading new versions of the files.

Unlike in the other applications, uploading files with the same name will not trigger the automatic name change, but it will replace the current file with the new one. This is done by providing our own `get_available_name` function:

```
class OverwriteStorage(FileSystemStorage):
    def get_available_name(self, name, max_length=None):
        # If the filename already exists, remove it as if it was a true
file system
        if self.exists(name):
            os.remove(os.path.join(settings.MEDIA_ROOT, name))
        return name
```

which is set in the *FileField* field:

```
file = models.FileField(upload_to='files/', max_length=256, stor-
age=OverwriteStorage())
```

6.2 Automatic Tasks

Three automatic tasks were schedule to run once every day:

- *remove all temporary forms*
 - closing the Designer, while the form is in a “*TEMPORARY*” state, it triggers its deletion. However there are case where this doesn’t happens, for example, when the computer crashes.
- *remove all empty assets directories (forms and operations)*
 - as soon as the first asset is added to an operation or a form, a directory is created. However, deleting all assets, doesn’t delete the directory.

These tasks are done using the *django-q*¹⁰ application. The cluster was set in *settings.py*:

¹⁰ <https://pypi.org/project/django-q/>

```
Q_CLUSTER = {
    'name': 'ers_ia',
    'workers': env.int('N_WORKERS', default=4),
    'recycle': 500,
    'retry': 1800,
    'timeout': 60,
    'compress': True,
    'save_limit': 0,
    'cpu_affinity': 1,
    'label': 'Django Q',
    'catch_up': False,
    'orm': 'default'
}
```

and the tasks were created in the *apps.py* files. For example

```
class OperationsConfig(AppConfig):
    name = 'operations'

    def ready(self):
        from django_q.models import Schedule
        try:
            Schedule.objects.get_or_create(
                name='remove-all-empty-dirs-operations',
                defaults={'func': 'operations.services.removeEmptyAssets-
Dirs', 'name': 'remove-all-empty-dirs-operations', 'schedule_type': 'D',
                })
        except Exception as e:
            print('Unable to schedule automatic tasks! ', e)
```

7 APIs

All APIs are listed in tables 34, 35, 36 and 37.

Modal	Description
api/list_databases/	Returns a list of all databases the server is connected to and some info about them.
api/db_tables/	Returns a json with all databases elements that can be created from and their respective tables.
api/db_table_columns/	Returns the columns and their properties (type, size, ...) of a specific table and database.
api/change/	Updates the form's status, name and description.
api/check_name/	Checks if a form with a specific name already exists.
api/new_form/	Creates a new Form and set its status as <i>TEMPORARY</i> .
api/save/	Saves a form and change its status as well, if necessary, to <i>EDITABLE</i> .
api/editable_forms/	Lists all <i>EDITABLE</i> forms (status = <i>EDITABLE</i> <i>TEMPORARY</i>).
api/upload_form_asset/	Uploads an asset to the media directory.
api/get_form/<int:pk>/	Gets a specific form.
api/get_editable_form/<int:pk>/	Gets a specific editable form (status = <i>EDITABLE</i> <i>TEMPORARY</i>).
api/list_form_assets/<int:pk>/	Lists all assets or all assets of a specific type of a specific form.
api/list_form_assets/<int:pk>/<str:name>/	Lists all assets or all assets of a specific type of a specific form.
api/form_content/<int:pk>/	Returns the form contents (no id, name, description, ...).
api/delete_temp/	Deletes a temporary form, deletes all its assets and their directory.
api/remove_form_asset/	Removes form assets, identified by an array of IDs or full relative names.
api/download/<int:pk>/	Downloads an asset.
api/get_table/<int:form>/<str:table>/	Gets the contents of a <i>CSV</i> file as a json object.
api/list_table_columns/<int:form>/<str:table>/	Given the id of an table (asset) returns all columns of the table.
api/get_table_column/<int:form>/<str:table>/<str:column>/	Gets a list of all values of a specific column of a <i>CSV</i> table.
api/list_dbs/	Returns a list with the name of all databases

	that elements can be created from, as defined in <i>settings.py</i> .
api/list_db_tables/<str:db>/	Returns a list of all tables in a specific database where elements and data can be extracted from.
api/list_db_table_cols/<str:db>/<str:table>/	Returns a list of all columns/fields of a specific table of a specific database where elements and data can be extracted from.
api/get_db_table_col/<str:db>/<str:table>/<str:column>/	Returns all the values of a columns of table of database where elements and data can be extracted from.
api/get_db_table_col_unique/<str:db>/<str:table>/<str:column>/	Returns all the unique values of a columns of table of database where elements and data can be extracted from.
api/list_queries/	Returns all <i>LOCKED</i> queries.

Tab. 34: Designer APIs

Modal	Description
api/use/	Locks a form, by setting its status to <i>IN USE</i> .
api/disable/	Disables a form, by setting its status to <i>DISABLED</i> .
api/delete/	Deletes a form.
api/delete_temps/	Deletes all temporary forms.
api/clone/	Clones a specific form.
api/forms/	Returns a list of all the forms.

Tab. 35: FormsManager APIs

Modal	Description
api/list_in_use_forms/	Returns all in use forms but not disabled.
api/exec_query/	Executes a query and returns the results.
api/new_operation/	Creates a new operation and sets its status as <i>OPEN</i> .
api/delete_operation/	Deletes a non <i>COMPLETED</i> operation, all its assets and their directory.
api/update_operation/	Updates a non <i>COMPLETED</i> operation.
api/new_operation_complete/	Saves an operation.
api/get_operation/<int:pk>/	Returns a specific operation.
api/list_operations/<str:stats>/	Returns all operations with a given status for the current user.
api/list_all_operations/	Returns a list of all operations, regardless of the status and user.

api/list_non_completed_operations/	Returns all non <i>CLOSED</i> operations, so all (<i>OPEN</i> <i>CLOSED</i>) for the current user.
api/list_all_non_completed_operations/	Returns all non <i>COMPLETED</i> operations, so all (<i>OPEN</i> <i>CLOSED</i>).
api/get_operation_data/<int:pk>/	Returns the data of a specific operation.
api/set_operation_status/	Changes the status of a non <i>COMPLETED</i> operation to (<i>OPEN</i> <i>CLOSED</i> <i>COMPLETED</i>) and update its assets.
api/get_operation_form_data/<int:pk>/	Returns all the data of a specific operation and its respective form (only id, name and form).
api/upload_operation_asset/	Uploads an asset and associates it to a non <i>COMPLETED</i> operation.
api/upload_operation_asset_complete/	Uploads an asset and associates it to a non <i>COMPLETED</i> operation.
api/remove_operation_asset/	Removes operation assets from a non <i>COMPLETED</i> operation, identified by an array of IDs or full relative names.
api/validate_operation_inputs/<int:pk>/	Validates the inputs of an operation.
api/get_op_form_data/<int:pk>/	Returns all relevant data from the form (id, name, label, database field, type) and operations (input data, status, visibility) and includes all validations associated with each field.
api/list_operation_assets/<int:pk>/	Lists all assets or all assets of a specific type of a specific operation.
api/list_operation_assets/<int:pk>/<str:name>/	Lists all annexes of a specific type of a specific operation.
api/list_operation_annexes/<int:pk>/	Lists all annexes of a specific type of a specific operation.
api/download_asset/<int:pk>/	Returns an asset.
api/clean_operation_assets/	Removes all unused assets of a non <i>COMPLETED</i> operation by saving the operation data of a specific operation.
api/is_operation_signed/<int:pk>/	Checks whether or not an operation is signed.

Tab. 36: Operations APIs

Modal	Description
api/api-token-auth/	Generate an access token.
api/list_operations/	Returns a list of all operations, regardless of the status and user.
api/list_non_completed_operations/	Returns all non <i>CLOSED</i> operations, so all (<i>OPEN</i> , <i>CLOSED</i>) for the current user.
api/get_operation/<int:pk>/	Returns a specific operation.

api/get_operation_data/<int:pk>/	Returns the data of a specific operation.
api/get_operation_form_data/<int:pk>/	Returns all the data of a specific operation and its respective form (only id, name and form).
api/validate_operation_inputs/<int:pk>/	Validates the inputs of an operation.
api/get_operation_data_val/<int:pk>/	Returns all relevant data from the form (id, name, label, database field, type) and operations (input data, status, visibility) and includes all validations associated with each field.
api/list_operation_assets/<int:pk>/	Returns a list of all annexes of a certain operation.
api/delete_operation/	Deletes a non <i>COMPLETED</i> operation.
api/set_operation_status/	Sets the operation status of a non <i>COMPLETED</i> operation.

Tab. 37: Rest APIs

8 TODO

Apart from optimization requirements, we now present the list of some features yet to be implemented:

- **Designer:**
 - Richtext element
 - Validations for the FoodEx2 element
 - Import / Export - Import and export forms to doc format
 - Undo
 - Rest API E/A - An E/A that make an API call and to fills fields
 - SQL parser to avoid having to use markers in E/A Database query
 - Chain events
 - Graphical interface for the E/A system
 - Selection of multiple background images in one selection
 - PDF selection as background image, also capable to automatically add additional pages if required
- **App:**
 - Smart cart signature in mobile devices
 - Starting actions so that an action is executed if its event are already fulfilled at start
- **Global:**
 - Form's Table assets - Keep synchronized with an external source
 - Teams management system