

Contents

Contents	i
1 Metadata Visualization	1
1.1 MONTRA	1
1.1.1 Communities	2
1.1.2 Questionnaires	3
1.1.3 Fingerprints	4
1.1.4 Import Questionnaires - Excel	12
1.1.5 Data Models	17
1.2 Refactoring	21
1.2.1 Data Models	21
1.2.2 Views	23
1.2.3 API	24
1.2.4 Excel	27
2 Automatic Metadata Extraction and Update	33
2.1 Requirement Analysis	33
2.2 Extraction	33
2.2.1 ACHILLES	33
2.2.2 Asynchronous Message Systems	33
2.2.3 Kafka Source Connectors	33
2.3 Publishing	33
2.4 Network Manager	34
2.4.1 Pipelines Workers	34
2.4.2 Sender	34
2.4.3 Orchestrator	34
2.4.4 Admin Dashboard	34
2.5 Network Data Flow	34

Metadata Visualization

Among the presented tools in chapter ??, MONTRA was our go-to mainly because of its data source-centric approach. Also, no real data is shown here, only metadata is handled within the platform, however, it still allows to impose access permissions at several levels.

This chapter will detail the key concepts of the MONTRA framework and its internal data model. After this, there will be presented a refactoring process that was done to the platform, with its improvements and flaws fixed.

1.1 MONTRA

Originally, the MONTRA framework was developed by the Bioinformatics team of the Institute of Electronics and Informatics Engineering of Aveiro associated with the European Medical Information Framework (EMIF) project with the goal to develop a common patient health information framework with emphasis on the research topics of Obesity and its metabolic complications and Markers for the development of Alzheimer's disease and other dementias. The code is publicly available on github¹, whereby Django 1.4, a high-level Python Web framework[1], was used as a framework to develop the entire system. This framework allows to develop complete web applications, in a faster and easier way. It contains a model layer that allows specifying database tables through python classes called models, following a Object-Relational Mapping (ORM) approach, allowing to perform database queries through python code instead of Structured Query Language (SQL) queries. Django can then check if there are new models or if existing ones have changed. Such changes are then expressed through database migration files which will apply them to the database tables. Next, the developer can set custom URL patterns so specific requests are handled by a specific function of the view layer, where the business logic will be implemented. Finally, Django contains a template layer that allows building dynamic HyperText Markup Language (HTML) pages without requiring to have a separate javascript framework to do such. The developer builds the main static structure of the page and then uses a special syntax that describes how Django

¹<https://github.com/bioinformatics-ua/montra>

should display the data received from the view layer. Django also has an important form feature that allows to easily create a set of pages that allow performing the usual Create, Read, Update and Delete (CRUD) operation over the database model.

MONTRA's development started at the beginning of 2013 and ended in the middle of 2018. After this date, at the end of 2018, the framework started being used by the European Health Data and Evidence Network (EHDEN) project, to develop a portal to allow discovery and analysis of health data of a federated network of data sources standardized to the Observational Medical Outcomes Partnership (OMOP) common data model in Europe. Currently, the project is being developed in a private repository but has intentions to make it public after the code base is more robust and well documented.

1.1.1 Communities

In the first versions of the framework, MONTRA allowed only one level of organization related to data sources, in which they could only be separated according to their skeleton that describe their original data, which will be more detailed in the next subsection. Newer versions created the concept of a Community. This allows having multiple networks of data sources on the same portal, where originally the only option was to have different installations of the framework.

These communities can be created in several ways:

1. the admin can create it through Django's admin console
2. a user can request a community to be created through a form and then the admin will receive an admin with such request
3. the MONTRA installation can be deployed in a single community mode where only one community exists, which is created on the first setup, giving the idea that there is no community concept on the platform

They also can have different access levels:

- Open: Does not require membership. Any user can access the data sources of this community
- Public: Does not require a membership however, the user needs to accept a set of terms and conditions before being able to access the data sources
- Moderated: A user has to request the community managers for approval
- Invitation: Users can only access and see the community by invite and subsequent approval

Plugins

The concept of Community also allows customization at that level, affecting only sections within that given community. One example of such customization is plugins, a way to extend the functionalities of the framework without having to deal with the base code. These plugins can either be full web applications, with different functionalities, that are linked to MONTRA through the community navigation menu or extensions that provide extra data services such as a dashboard about the data of a data source.

1.1.2 Questionnaires

As the data from different data sources is highly heterogeneous, MONTRA ensures that the data inserted within a given community follow a common structure. This structure is called a skeleton, which is represented in a form of a questionnaire with a set of questions, which can then be grouped in sections called Question Sets. It represents a set of metadata that better describes the original data of data sources that need to remain private. The skeleton schema can easily be defined through a spreadsheet, which will be more detailed on subsection 1.1.4.

Next is presented the available question types which can be used to build a questionnaire for a community

Type	Description
choice	Single choice (radio box)
choice-freeform	Single choice with an open text fields
choice-multiple	Multiple choice (checkbox)
choice-multiple-freeform	Multiple choice with and open text fields
choice-multiple-freeform-options	Multiple choice with and open text fields
choice-tabular	Creates a table with single, multiple choices or text by row
choice-yesno	Single choice with yea and no choices
choice-yesnodontknow	Single choice with yes, no and don't know choices
comment	Used to separate groups of questions
custom	Mirrors another question by its type
datepicker	Field with date picker widget
email	Text input with email validation
numeric	Numeric input
open	Text field with no validation
open-button	Text input with backend validation
open-location	Text input with autocomplete sugestions
open-multiple	Allows to record an history of a value overtime
open-multiple-composition	Allows to record an history of several values overtime
open-textfield	Same as open but a textarea html tag is used
open-upload-image	Image upload
open-validated	Text input with a regex validation
publication	A custom widget that allows to attach a set of publications
sameas	Mirrors another question by its number
timeperiod	Numeric input + select to choose numeric unit
url	Text input with url validation

Table 1.1: All available question types that can be used to build a questionnaire

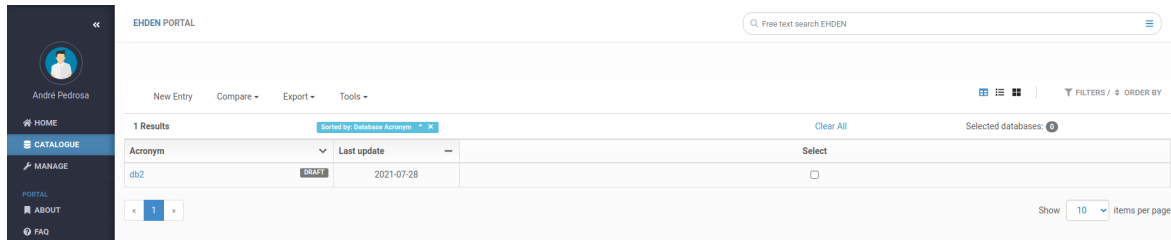


Figure 1.1: The user interface displayed after selecting a community, which shows the list of the fingerprints of the chosen community

1.1.3 Fingerprints

A fingerprint is a name given to the set of answers to the questions of a questionnaire. In other words, it is the metadata that better describes the original data of the associated data source. Data owners can start to answer the questions to build the profile of their data sources once there are communities on the platform and, these have at least a questionnaire associated.

On the list presented in figure 1.1, we can notice that the only fingerprint present is marked as draft. This is a state of the fingerprint which prevents from non-ready or non-approved fingerprints won't show to the regular community users. With this, for such users the list showed above would be empty. Fingerprints can be published depending on the chosen settings for the community. After the data source owners request to publish a fingerprint the framework allows to either automatically accept or require a community manager to accept it.

Views

In figure 1.3 it is presented the user interface where a data owner can answer the questions of a questionnaire. Each question of the questionnaire is placed under a container that can be collapsed, as is shown for question *Institution name*. However, if multiple questions are grouped, the collapsible container will affect all questions of the group. Besides the input widget where the data owner can insert its answers, the user also has some additional control buttons:

1. Allows to Hide or Show Questions that have been answered or that are empty. Besides being an interesting feature is important to note that on the last version it does not work, as clicking on the presented options will result in no visual effect
2. Allows to collapse or expand all questions or question groups containers
3. Allows the data owners to set permissions at the question set level
 - Visibility: Let plugins have access to answers data
 - Allow printing: On the fingerprints list page, showed in figure 1.1, there is a dropdown with tools, being the only one the "Print" tool (1.2). However, this feature is not correctly implemented since it calls the browser's built-int printing function on the fingerprint list page, so no actual fingerprint data will be printed. This permission ends up being useless. Additionally, if a user calls the browser's print function (e.g. hitting Ctrl+P) when viewing the data of a fingerprint, the platform will not block the action.

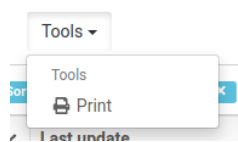


Figure 1.2: Available tools on the fingerprint selection

Figure 1.3: User interface to create a fingerprint

- Allow indexing: If the data owner allows indexing of the answers, which will allow for other users to find fingerprints based on the answers to a question of this specific question set.
 - Allow exporting: If the answers to this question set can be included on the export file of a fingerprint.
4. Enable navigation along the question sets of the current questionnaire
 5. Permits to save or cancel all the changes made to the current question set

Once the fingerprint is filled and published, a regular user can consult the metadata, which will be displayed by default in detailed view. Similar to the create view, it is presented with some control buttons:

1. Database level plugins associated with this community
2. Statistics of this fingerprint
 - Progress bar + Filled: How many questions of the questionnaire were answered
 - Hits: Number of times this fingerprint showed up on search queries
 - Unique Views: Number of users that visited this fingerprint
3. Question set controls
 - Summary: Allows to switch to the summary view (Figure 1.5)
 - Collapse & Show: The same role as mentioned for the create view
4. Fingerprint control buttons

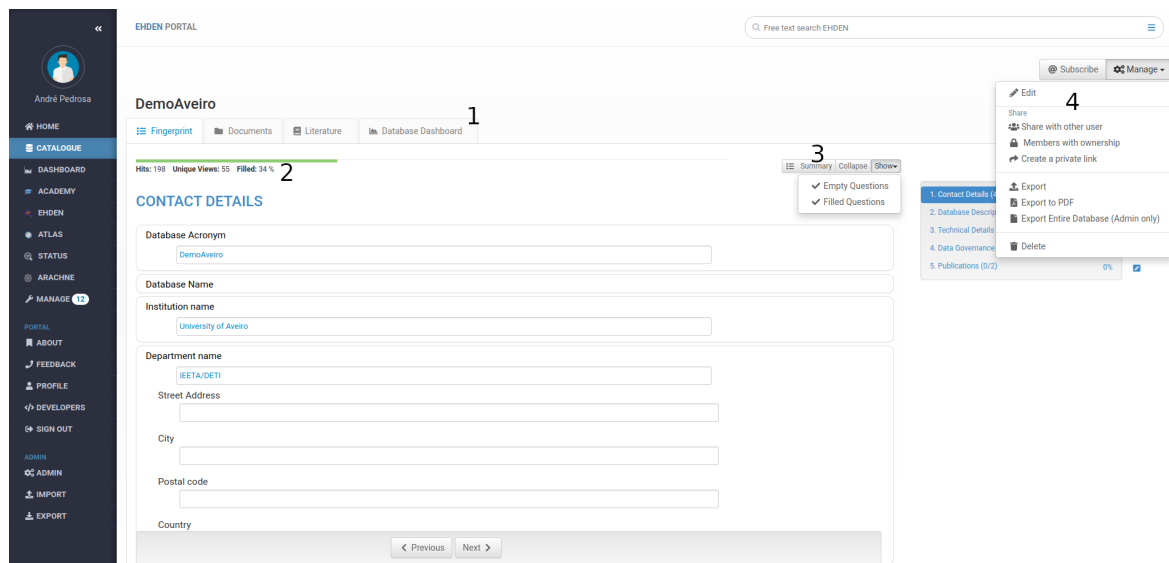


Figure 1.4: A detailed version user interface to view and analyze fingerprints

- Subscribe: Receive notifications whenever changes are made to the fingerprint answers
- Manage: Several fingerprint operations
 - Edit: Enter the edit mode
 - Share: Allows to add other users as owners of the fingerprint and also to create links that enable anonymous users to consult the fingerprint
 - Export: Different forms of export. CSV, PDF, and MONTRA format to import on other installations of the MONTRA framework
 - Delete: Remove the fingerprint from the community

On the show view, if the user enters the summary view (Figure 1.5), the user is presented with a table with three columns where each row contains the question number, name and the answer given. On this view, by hovering over an empty answer container the user can send a request to the database owner to answer the specific question.

The MONTRA framework also offers an “Advanced Search” feature, allowing to perform searches for fingerprints. The overall interface used is identical to the ones presented above, where the user answers the questions of the specific questionnaire of the community. The main difference to the other versions of the fingerprint view is that the only control buttons available are just the navigation ones, and all the questions don’t have any validation. Additionally, at the bottom of the page, it is provided to the user a way to customize the search query, allowing to create complex search criteria through a drag and drop interface, as is presented in figure 1.6.

Despite all these variations of the same view having a similar look, as some components appear on several variants, all views have a separate Django template, so there is some duplicated code across the different views. If some changes are made to a shared component, such as the side question set menu bar, those changes have to be applied to all different

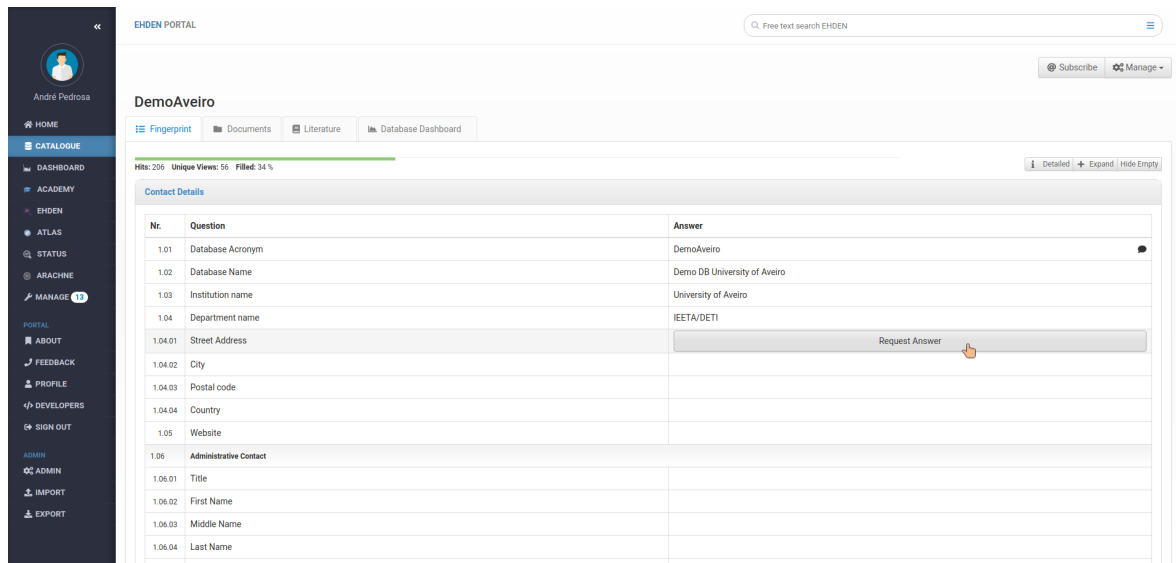


Figure 1.5: A summary version user interface to view and analyze fingerprints

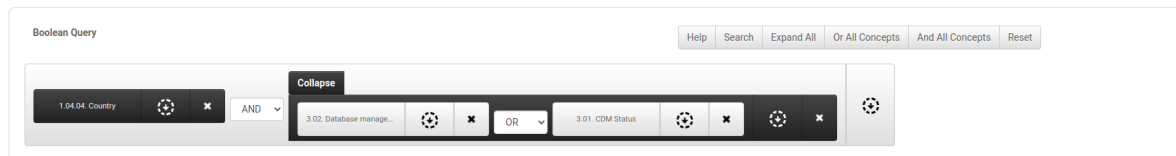


Figure 1.6: Interface to customize the fingerprint search query

templates. This can be avoided since the Django template system allows to both include a shared component into other templates and also supports conditional rendering, for example, the permissions dropdown for a question set of a fingerprint should only be rendered when the user is editing or creating a fingerprint.

These different view versions provide a valid workflow to perform CRUD operations over the metadata related to a data source, however, views that are used to insert or edit metadata of a fingerprint present several flaws related to how the inputs are validated and how they are presented to the user.

First, the validation of the user input is done on the client-side through javascript code that runs after a user submits a question set form. Subsequently, the data is sent to the backend through Application Programmable Interface (API) calls. However, if one would use the API directly to update metadata of the fingerprint, the validation could be skipped and invalid data could be stored on the database. In some cases there might exist some validation code on the server-side, however, this brings the necessity to maintain two separate code files.

Second, there is no escaping procedure done to the user's input when it is fetched from the database which allows that Cross-Site Scripting (XSS) attacks can be easily performed and put users that consult a compromised fingerprint at risk. As an example, on figure 1.7 on the Database Name field, I added a *script* tag with code that shows a popup and also a valid database name after. Once a user opens the fingerprint to check the data, the code will execute but the dummy name will render on the Database Name field. This can be

Contact Details

1.01. Database Acronym

 Database Name Available.

1.02. Database Name

1.03. Institution name

Figure 1.7: Example of how an XSS attack could be done on MONTRA

mednat.ieeta.pt:8624 says
 You were Hacked :/ OK

EMIF CATALOGUE / European Network

Nothing selected

db

Fingerprint Documents

Hits: 5 Unique Views: 1 Filled: 1.7 %

CONTACT DETAILS

1.01. Database Acronym

1.02. Database Name

Figure 1.8: Victim of a simple XSS attack

further exploited, where the user visiting the fingerprint will not notice that malicious code was executed.

All these problems exist because such views were developed from scratch without using the provided features that Django has out of the box for form validation and security. For client-side validation, Django takes advantage of HTML5 form validation features [2]. This allows imposing restrictions and validations on the user's input without writing any additional javascript code.

To show these features I wrote a simple HTML file with a simple form that expects a number under 100 and an email.

```
<html>
<body>
  <form>
    <label for="num">Number:</label>
    <input id="num" type="number" max="100">

    <label for="mail">Email:</label>
    <input id="mail" type="email">

    <button type="submit">Submit</button>
```



Figure 1.9: Error messages that appear after submitting the form on a chromium based browser

```

    </form>
  </body>
</html>

```

If I try to submit the form with invalid values, error messages are presented as shown in figure 1.9.

Additionally, if the data is sent directly through an API call to the backend, Django forms framework ensures invalid data is rejected. With this, when building a form in Django the developer only needs to specify what fields are present and their type and restrictions and Django will validate all this before data is stored on the database.

Finally, XSS attacks are prevented because Django escapes the values that were previously provided by users when filling the input tags, e.g. the character “<” is transformed in “<”, avoiding the browser to interpret user’s input as HTML code.

API

It was mentioned several times that some checks are not enforced through the API. It will be detailed now what calls are performed by MONTRA’s fingerprint views.

Note that there are two different APIs available here. Fingerprint views use one specific set of API requests that return answers in HTML format, ready to present to the user, and others to send the user’s data. For a regular user to use, there are other set o API endpoints that are more human friendly, which is documented and has a How to Use page, however, an advanced user can see what requests are made on the fingerprint views through browser’s console and perform the requests themselves.

First, we will go over the API used by the fingerprint views. But before showing the available endpoints, let’s go over how the fingerprint views are organized and how they show only the questions of a certain question set at a time.

Let use figure 1.10 as example. According to the right side menu, there are six question sets and currently the question set “Contact Details” is being presented. In the middle of the page, we then see the content of the question set: questions, title, control buttons, and permissions. Note that there is a scroll bar so the question set contains more questions. Also the previous, next, cancel and save buttons are not tied to the question set. The other question sets are also present on the page, however, are hidden. Every question set has its container, so whenever the user clicks on the previous or next buttons or selects another question set on the

The screenshot shows the EMIF CATALOGUE interface. The main content area displays the 'Contact Details' form, which is highlighted with a red rectangle. The form contains several input fields for contact information, including 'Database Acronym', 'Database Name', 'Institution name', 'Department name', 'Street Address', 'City', 'Postal code', and 'Country'. Each field has a checkbox to its right. The form is titled 'Contact Details' and has a 'Show' button and a 'Collapse all' button. To the right of the form, there is a sidebar titled 'EHEN Questionnaire Database' which lists five question sets with their completion status: 1. Contact Details (0/21) 0%, 2. Database Description (0/11) 0%, 3. Technical Details CDM (0/5) 0%, 4. Data Governance and Ethics (0/9) 0%, and 5. Publications (0/2) 0%.

Figure 1.10: Each question set has its container. Here the current container is presented within the red rectangle. The other existing question sets are represented through the blue lines, which currently are hidden.

question set side menu bar, the current question set container is hidden and the new is shown. For questionnaires with a high number of question sets, rendering all available question sets could not be necessary. For that, a question set is loaded only when accessed the first time and following accesses will not require a load.

The fingerprint view can be used for four different use cases:

- Create a new fingerprint: The questions are presented with clear and editable inputs
- Edit an existing fingerprint: All questions are editable but the previously answered questions are filled
- View the answers of a fingerprint: A read-only version of the fingerprint's answers
- Search for fingerprints: Same as creating a new fingerprint, however, no validations are performed on the user's input

For these use cases, the following endpoints are used:

- Create

```
GET [base url]/c/[community slug]/addqs/[fingerprint hash]/[questionnaire id]/[question set id]/
POST [base url]/c/[community slug]/addPost/[questionnaire id]/[question set id]/[save id]
```

- Edit

```
GET [base url]/editqs/[fingerprint hash]/[questionnaire id]/[question set]/
POST [base url]/c/[community slug]/addPost/[questionnaire id]/[question set id]/[save id]
```

- View

```
GET [base url]/detailedqs/[fingerprint hash]/[questionnaire id]/[question set id]/
```

- Search

```
GET [base url]/c/[community slug]/searchqs/[questionnaire id]/[question set id]/
```

For both cases where new data is stored, besides the usual GET that is used to load the question set, there is also a POST request that saves the progress done to a specific question set. The data for these requests are gathered natively by javascript once each question set container contains a *form* tag englobing all the questions inputs. This way there is no need to iterate over the questions of a question set and append each response to the request's data. The *save id* field of the endpoints tells to which question set the data is related to, however, there is already a *question set* field which always has the value of 1, so one of these fields could be removed from the endpoint.

The GET request is then used to load the HTML to present the questions of a question set. Note that the returned data is just the HTML data to be placed on the respective question set container and not the whole fingerprint page.

Moving now to the set of API endpoints available to the users to both read and update answer data the following are available:

```
GET /api/fingerprints/[fingerprint hash]/answers/
GET /api/fingerprints/[fingerprint hash]/answers/[question slug]
PUT /api/fingerprints/[fingerprint hash]/answers/[question slug]
```

The first two GETs are used to retrieve answers data, which is returned as the JavaScript Object Notation (JSON) object presented next, the only difference being the first one returns an array of the mentioned object instead of just one.

```
{
  "question": "patients_count",
  "data": ""
}
```

With the PUT request, a fingerprint owner can update data of specific answers where a JSON object should be sent in the body of the request with the field “data”.

```
{
  "data": 10000
}
```

Existing two different types of endpoints to retrieve answers data makes sense since the returned format is returned in a way that is easier to handle data by the target entity that will consume that endpoint. If the fingerprint pages receive the data in HTML format they can just put the HTML in the determined container instead of having to build the entire container and insert the data on each input. Accordingly, if data is returned in a JSON format it can easily access the data and perform some data processing avoiding going transverse the HTML and retrieve the data from each input. However, having two different endpoints

to update data doesn't make that much sense since current HyperText Transfer Protocol (HTTP) requests libraries offer an easy way to build a request in the required format as the one's browsers automatically build whenever a form is submitted.

Draft

We will also go over the API endpoints that the front end code uses to request to change the fingerprint state from draft to published. Associated with this feature, there are two endpoints available:

POST [base url]/api/pending/[fingerprint hash]

POST [base url]/api/draft/[fingerprint hash]

The existence of two endpoints for the same purpose is because Communities have a setting that allows to auto-accept requests to publish a fingerprint. For that, whenever the auto-accept setting is off, the first endpoint must be used, which will send a request to the community owners to publish the given fingerprint. The second must be used otherwise.

The problem with this approach is that the code that decides what endpoint to use is on the client-side and there is no server-side check if the correct endpoint is being used. If the auto-accept setting is off and the second endpoint is used, a user can publish his fingerprint without requiring the approval of the community owners.

1.1.4 Import Questionnaires - Excel

As mentioned before, to define a skeleton of the metadata that describes a data source for a given community, a spreadsheet file has to be submitted. There is already a template with some instructions and columns where a community manager only has to fill the necessary rows to then get the wanted result.

The column defined in the template are the following:

- Type: the type of the specific row. Here are allowed 3 different values:
 - QuestionSet: allows dividing the questionnaire into several sections
 - Question: a question specification
 - Category: allows to create a group of questions inside a question set
- Text/Question: label/name given to the item being defined
- Level/Number: use as a level for questions and categories and number otherwise. As level allows to create groups of questions inside a question set. As number defines the number, and subsequently the order, of the question sets.
- Data type: used only for rows of type Question, specifying the question type
- Value list: used to indicate extra information to build the question
- Help text/Description: a small text that will be displayed along with the item being defined
- Tooltip: Yes if the Help text/Description should be displayed as a tooltip or No otherwise
- Slug: internal identifier

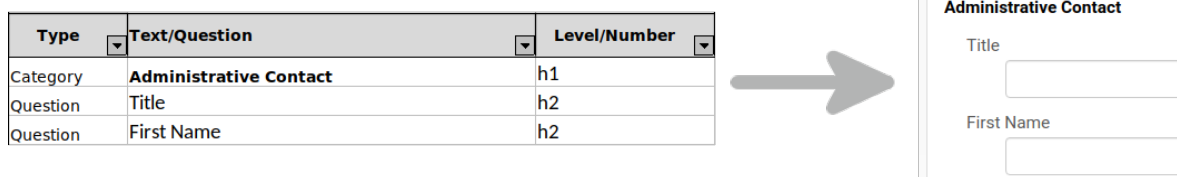


Figure 1.11: Create a group of questions with a title

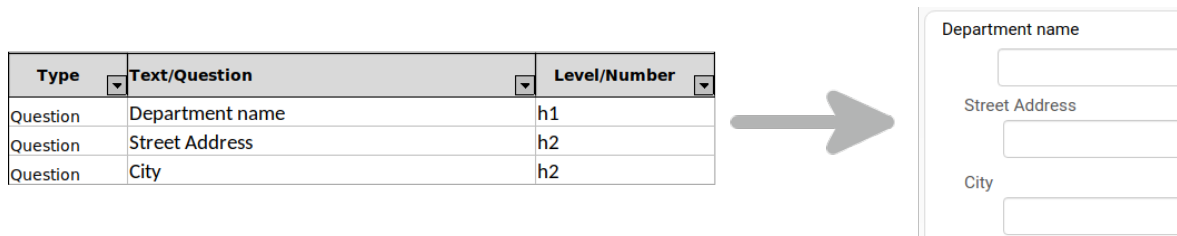


Figure 1.12: Create a group of questions using a question's text as the title

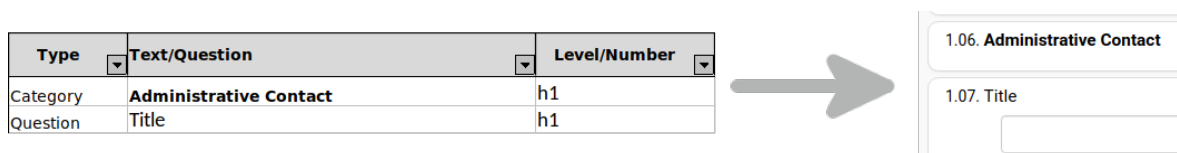


Figure 1.13: Example to show that the category way to create question groups also depends on the values that are set on the level column

- Dependencies: used to tell that a question or group of questions can only be answered if a specific choice of a choice-based question was selected
- Stats, Comments Stats and Disposition: Columns that were used for old features but are still present on the spreadsheet
- Include in Advanced Search: if the answers of the question can be used to search fingerprints

Question Groups

From the previous list, question groups were mentioned both when the type column has the Category value and on the Level part of the Level/Number column. The former is used to add a title with no question associated, resulting in what is presented in figure 1.11.


On the latter, the text of the question in the most upper level is used as the title of the questions group, resulting in an output similar to the one in figure 1.12.

It's important to highlight that the category way to create question groups is not independent of levels. If both a category row and a question are on the most upper level, MONTRA will render two separate collapsable containers, where the first one will be empty, as is shown in figure 1.13.

Value list

To avoid having a spreadsheet with several columns that will not be used for all types of rows, the value list column expects some extra information required for some types of

Type	Text/Question	Data type	Value list
Question	If you have a documented data dictionary, is the data dictionary a document (paper or electronic) or structured (spread sheet, database, XML, ISO 11179 etc.)	choice-multiple	Paper{...} Word (unstructured electronic){...} Spread sheet, Database{...} XML{...} ISO 11179{...}



8.02. If you have a documented data dic

☐ Paper

☐ Word (unstructured electronic)

☐ Spread sheet, Database

☐ XML

☐ ISO 11179

Figure 1.14: Multiple choice question with some extra text fields associated with the several choices

questions.

Choices

For choice-based questions, the value list column is used to define the possible choices and to add an extra text field associated with a given choice. Choices are separated by a “|” character and the extra text field can be set by appending “{...}” after the target choice’s text.

Although the framework allows the extensibility to have an additional text field, these don’t support other types of input and also have no validation. Also, if the number of choices is high and the text of them is long, there starts to exist some clutter on the spreadsheet. With this, the person creating the spreadsheet will have problems perform edits and check if there is something wrong or missing.

To facilitate the job of who is filling the spreadsheet, some question types are shortcuts. For example, the choice-yesnodontknow is a question of type choice with three possible options: Yes, No, Don’t Know. Choice variants with freeform on the name, besides the usual choices, always have an additional text field, associated with the question instead of a choice.

Open Multiple Composition

Open multiple questions allow showing a history of a given value. For that the simple version of the question is represented in a table of two columns: Date and the value, so no input is expected on the value list column. The composition version of the question type allows having several values, instead of just one. In a way, the simpler version can be used as a shortcut question type, since it can be reproduced with the composition variant.

To render this question type, a third-party widget called Tabulator² is being used. The configuration of such widget, expects an array of JSON objects, each specifying some configuration of each column. To configure the open multiple composition question type, the value

²<http://tabulator.info/>

Type	Text/Question	Data type	Value list
Question	Clinical test parameters results.	open-multiple-composition	{title:"Weight X", field:"msurex", editor:"input"}, {title:"Height", field:"msurey", editor:"input"}, {title:"Msure Z", field:"msurez", editor:"input"}

↓

12.08. Clinical test parameters results.

Please capture the results for the variables of the clinical test B.

	Date	Weight X	Height	Msure Z
	30/07/2021	10	10	

Figure 1.15: How an open-multiple-composition question type is specified on the spreadsheet and how it is rendered

list column expects these JSON objects, where the date column is implicit. The MONTRA framework will then put the provided objects on the configuration array that the Tabulator widgets expects, adding the configuration for the date column.

Choice Tabular

This type of question allows reusing the same choices across several answering items. There are three variations of this questions types, where the difference between them is what and how is the information connected between answering items and choices. There are two versions where the user can select one (single choice) or more (multiple-choice) choices for each answering item. The other version allows the user to write text for each choice within each answering item. The question type is rendered as a table where in the columns are displayed the several choices and on the rows the different answering items are presented. Additionally, there can be a “More” choice column, where the user can insert any text information for a specific answering item since is displayed with a textarea HTML element.

The value list column of a choice tabular question type expects a three-component value. Each component is separated by the characters “\\”. Within each component, items are separated by the “|” character. The components are the following:

- choices (columns)
- answering items (rows)
- type of the information: available values are choice, multiple-choice and text

Figure 1.18: The questions with dependencies are not rendered if the specific choice is not selected

Figure 1.19: Once the dependency of a specific question is met it will be rendered

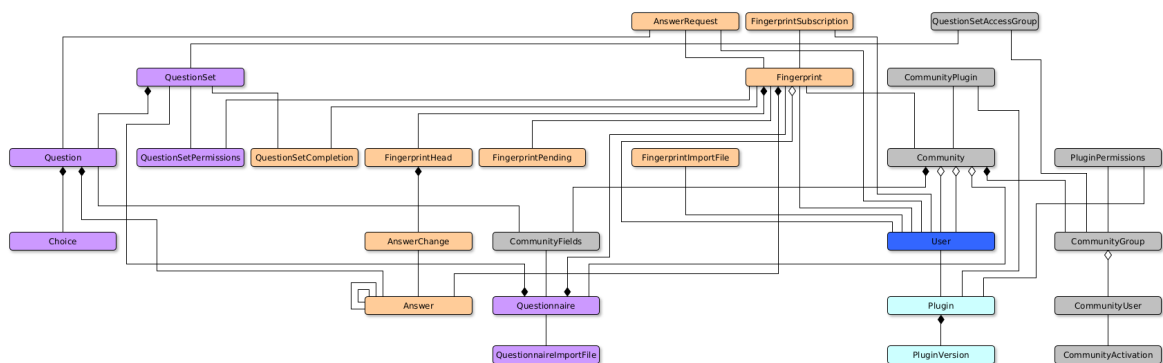


Figure 1.20: Class diagram of MONTRA's Model classes. Each color has a Django application associated. Gray: Community; Purple: Questionnaire; Orange: Fingerprint; Light Blue: Plugin; Blue: Django Auth. MONTRA's data model is much more complex, however, it is just presented the ones that impact the features and/or concepts mentioned previously.

1.1.5 Data Models

This section will be presented how the previous concepts are mapped to database models. Taking advantage of Django's ORM feature, MONTRA's data models are defined as python classes that extend Django's base Model class. These Model classes belong to different applications, which are associated with distinct aspects of the platform.

In figure 1.20 is presented the class diagram of the classes that are related to features and/or concepts that were mentioned in previous sections. Classes with the same color belong to the same Django application, dividing them into specific purposes.

- Blue - Django's built-in authentication system³.
- Orange - Fingerprint application: Answers to questionnaires questions and other models associated with features that were mentioned on the fingerprint views section, such as AnswerRequest and FingerprintSubscription. MONTRA also keeps a record of all the

³<https://docs.djangoproject.com/en/1.11/ref/contrib/auth/>

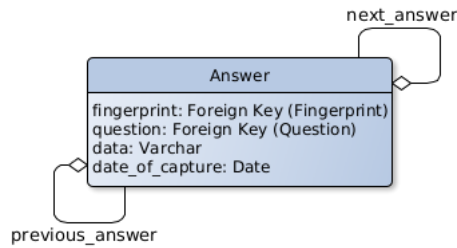


Figure 1.21: Detailed information of the Answer model of the Fingerprint application

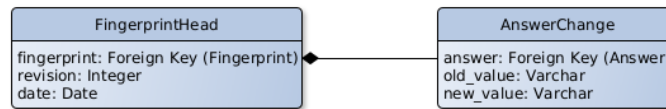


Figure 1.22: Models that store the changes to answers of fingerprint

states of a given Fingerprint. For that, it uses the FingerprintHead model, which maps to a set of AnswerChanges.

- Purple - Questionnaire application: Questionnaires structure information such as question sets, questions, and choices, and import spreadsheets logs. Additionally, associated with each fingerprint, contains a model with the allowed permissions on each question set.
- Light Blue - Developer application: Allows adding customizations to different MONTRA's installations through plugins.
- Gray - Community application: Community's groups, users and access permissions, fields to be presented on the fingerprint list page for each questionnaire and plugins.

Going into more detail on some models we can see some poor design decisions.

Regarding fingerprint answers to a questionnaire, all the data is stored in the Answer model on the data field, which is a variable-length string field type. Although this approach is much simpler in terms of data models, it leads to two problems:

1. this is not the most optimized way to store all the data types. Some question types expect numeric values and other date values, which could use built-in field types of a relational BDMS.
2. for complex question types which the answer contains several fields, before and after storing such data in the database, some processing has to be made to convert the data to the necessary format. One example of this is multiple choice questions, where the value of the several selected choice is joined in a single string to then be stored on the database. Every time the answer needs to be displayed to a user, it is necessary to split that string by its separator. For this situation instead of storing the choice's value, the models of the questionnaire application could be used as a foreign key.

Related to MONTRA itself, when a user provides no data to a specific question, an empty string will still be sent for that question on the submission of the answers to a question set, which will create unnecessary records on the database.

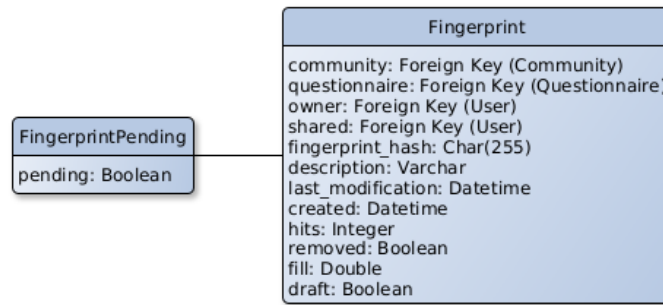


Figure 1.23: The Model that stores the information telling if a fingerprint is waiting to be approved to be published

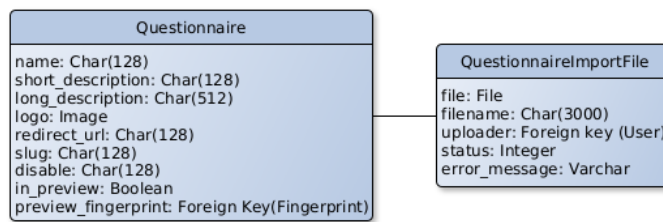


Figure 1.24: Questionnaire model and the model where questionnaire imports information is stored

As mentioned previously, MONTRA records the history of submissions for a specific fingerprint. Each submission has an associated FingerprintHead record, which its name might have been inspired by the HEAD concept of Git⁴, a version control system. For every FingerprintHead there is a set of answers that suffer changes which are then recorded on the AnswerChange model. In this model, we can get the answers data duplicated three times since it is already stored on the Answer model and can also be stored again as text in both *old_value* and *new_value*.

As mentioned previously, a fingerprint is not immediately available to all regular users, since it first enters on a draft state. To transit to the published state, a request needs to be made to the community owner. Such requests are stored on the FingerprintPending table, where the pending value will be *true*. Once a request is rejected or accepted, the value of the pending value is changed to *false*.

On the Fingerprint model, there is also a *draft* value that indicates if the fingerprint is published or not. Once a fingerprint is published, the value of the *draft* will be *true*, and on the FingerprintPending table, the associated record will have the pending value at *false*. However, this value is not required to be stored on the database since after a fingerprint is published, the fingerprint will not be pending therefore, the associated FingerprintPending record could be deleted.

Whenever a questionnaire is imported, a QuestionnaireImportFile record is created containing the information of the uploaded file and the user that uploaded it. Also contains a status field to give some feedback to the user. Associated with every import will also be

⁴<https://git-scm.com/>

Question
questionset: Foreign Key (QuestionSet)
number: Char(255)
text: Varchar
type: Char(32)
extra: Char(128)
checks: Char(128)
footer: Varchar
slug: Char(128)
slug_fk: Foreign Key (Slugs)
help_text: Char(2255)
stats: Boolean
category: Boolean
tooltip: Boolean
visible_default: Boolean
mlt_ignore: Boolean
disposition: Integer
metadata: Varchar
show_advanced: Boolean

Figure 1.25: Question model and its extensive number of fields

created a Questionnaire record. It contains an `in_preview` variable that is set to `true` whenever a questionnaire is in the import process. If it fails to import, only the `QuestionnaireImportFile` record will be kept, which the status will change to *Failed* and an error message will also be attached to the import record.

If the questionnaire is valid, the user will be redirected to a page where he can preview the result and can accept or reject it. It is important to point out that for the preview process a new Fingerprint object is created so API calls can be performed.

Strangely on the Questionnaire model, the disable column uses a character type column instead of a boolean one since it expects only the value of “False” and “True”. If there are some user input errors, it could lead to unexpected behavior or internal server error.

Associated with how the structure of a questionnaire is stored, there are only four models to do so: Questionnaire, QuestionSet, Question, and Choice. All are associated with a specific concept of the questionnaire, although other concepts are not represented. First, there’s the category, used to create question groups within a question set. It is represented as with a question record, where the “category” field of the question models has the value “true”. For more complex question types that require extra configuration such as choice tabular, which requires information about the name of the rows and column, and open multiple, which expects the configuration of the columns, such data is stored in a metadata field of the associated question model. Question types that do not make use of these fields, will have an empty value, leading to some database space being wasted.

When the questionnaire’s spreadsheet was explained, some fields were related to some old deprecated features. The question model contains the same fields that map to the questionnaire’s spreadsheet deprecated fields: `stats`, `visible_default`, and `disposition`. There is also a `slug_fk` field that points to a Slugs model that replicates the question’s slug and text, which is was used to yet another deprecated feature.

Finally, a question supports for the user to define additional checks which are defined on

the “checks” field however, this feature is not exposed through the questionnaire’s spreadsheet.

1.2 REFACTORING

Until now it was shown how the MONTRA framework was designed around databases, their metadata, and how that metadata can be shared among the platform users. For regular users the framework provides a good user experience, however, for power users, that make use of APIs, might make some errors which the framework will not prevent, leading to inaccurate data to be stored and presented. Also for developers, such flaws and bad design choices make the maintainability process of MONTRA instance a demanding and tedious process.

Next, we will propose several changes to be applied to the framework, beginning from the data models, regarding how fingerprint answers data is stored, how questionnaires structure is stored and some other fixes for some defects explained previously. Such refactoring will imply changes on other components, one of them being the rendering of questionnaires and how input validation is done. Finally, several clutter problems were mentioned when describing the spreadsheet to define a questionnaire structure, so the spreadsheet will also undergo a refactor.

1.2.1 Data Models

Since the MONTRA framework uses some outdated software and there are active installations, the refactoring process can not be simply designing new models and views, implementing them, and replacing old ones is not a viable option. The approach taken was to first decide what components would go through a refactoring process and within each, until what level we would apply it.

Taking into account figure 1.20, since both the Community and Developer (Plugins) Django applications target functionalities not so fingerprint-centric and are more related to features of the framework as a whole, we decided not to perform any changes on the models of these applications. This leaves us with both the Fingerprint and Questionnaire applications. Regarding the Fingerprint application, the fingerprint model is one of the main models of the framework, affecting several features of the framework, so we intend to perform minimal changes on it. However, the remaining models associated with the answers of a fingerprint and submissions will have a new models design. Respecting the Questionnaire application both the Questionnaire and QuestionSet models are represent well-defined concepts and don’t require any changes, yet, the remaining models, mainly the models storing the structure of the questions, their dependencies, choices, etc. will also have a new design.

In Figure 1.26 it is presented the new models in light green. The decided approach to implement these models was to create a new Django application and create all these new models on that application. This way, the migration process is an iterative process where the framework is always operational since the previous models still exist. Then with the help of an Integrated Development Environment (IDE), we can search for usages of the old models and migrate each feature at the time.

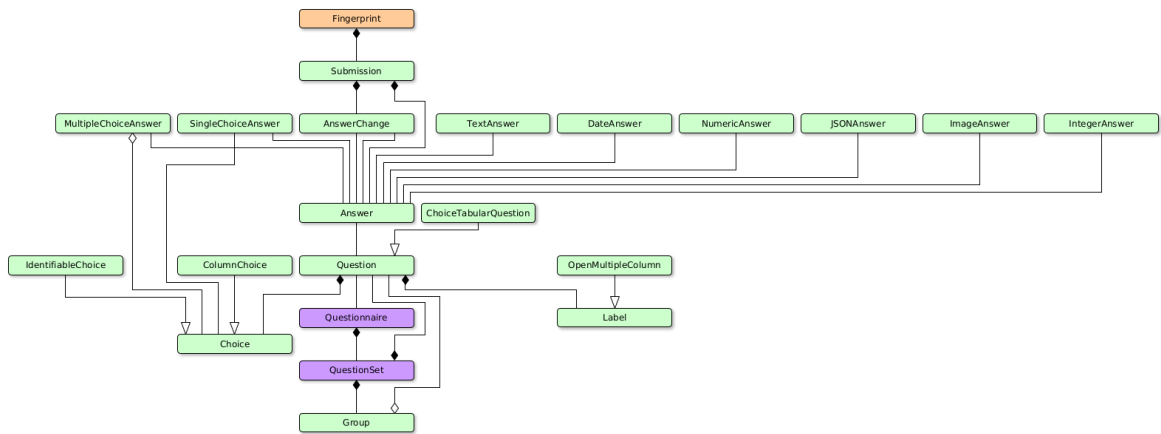


Figure 1.26: Model Diagram of the new models

Let's start with the fingerprint-related changes. Now there is a new Answer model that doesn't hold directly the content of answers, instead the content is stored on data-specific models such as IntegerAnswer and DateAnswer. For choice-based questions, instead of storing the value of the selected choice(s), a single or multiple choice answer contains a foreign key(s) for the selected choices. To establish the connection between the main Answer model and the data-specific model, the main model contains a *type* field that indicates the data type of the answer. Then the primary keys of the data-specific answer model are the same as the associated record of the Answer model, which allows fetching the data-specific answers of an Answer. It is important to note that the number of different answer types is not the same as the number of question types. Answers of different questions types can be stored in the same answer type models, for example, open text and email questions both can be stored as text in the database. Previously fingerprint submissions were being represented with the model FingerprintHead, now we created a Submission model which contains the same relationships as before, a set of answers, a set of answer changes, all this associated with a fingerprint. The AnswerChange model now does not contain the data of the previous and current answers, instead, it contains the foreign keys for the answer model. In cases where the change was from or to an empty answer, either the previous answer or the next answer field are filled with the NULL value.

Moving now to the models related to the Questionnaire application, there is a new Question model mainly because the previous one had too many fields. An example of these extra fields is the *metadata*, which is used to store the information of the column of the open-multiple question type and the choices and answering items of a choice tabular question type. This field was replaced by the addition of new models (Label) which will be explained in more detail in the next Excel subsection. Another example was the category field, which indicated if a specific question record described a category in the questionnaire which was being used to create subgroups of questions. With the new models, a new Group model was created which has a set of questions associated. Previously this relationship did not exist, so MONTRA would create a group based only on the question's order and level. Additionally, with the

addition of this Group model, there is no need to have a level field on the question type, since nested levels are represented through Groups, which can have a parent group. A group with no parent is represented in the root level of the questionnaire. The remaining models (Label, Choice, and their descendants) will be explained in more depth in the next Excel subsection since they were created mainly because of how questions are represented in the new spreadsheet format.

Meanwhile, there are previously existing models that have undergone changes to fix some design flaws mentioned before.

- **QuestionnaireImportFile:** To provide feedback to the user this model only contained an *error_message* field. However several times there are some errors with the questionnaire, but the framework can continue. In these situations, a warning could be sent to the user just so he is aware that a part of the spreadsheet has some errors and the result could not be his real end goal. Then the new model contains two new fields, *errors*, replacing the old *error_message*, and *warnings* fields, which are stored in JSON, where the keys are the lines where the errors or warnings were found and the values are an array of messages associated with that line.
- **Questionnaire:** associated with the questionnaire model we mentioned that it had a specific fingerprint attached that was used to show a preview of the result of the questionnaire. This field is not required since the preview mode of a questionnaire is viewed on read-only, so no answers are associated with these preview fingerprints.
- **QuestionSetPermissions:** From this model, we removed the useless permission of “Allow Printing”, considering it is impossible to restrict users from printing the page since a screen capture software can be used as a replacement for the browser’s print function. Also, there was a record associated with each question set of every fingerprint, even if the values of each permission were the default ones. The new approach taken is to only create a record if any permission value is different from the default one. Otherwise, an object with the default values is returned.
- **Fingerprint:** It has a new “*submission_token*” used for the new API endpoints explained next.

1.2.2 Views

Deixo isto mais para o fim pois ainda vou fazer alterações nas views

- Agora todas as variantes da fingerprint view usa o mesmo template. Uma alteração implica apenas uma alteração
- Falar nos vários componentes da pagina de fingerprint E em que variantes da view eles aparecem:
 - draft checkbox
 - Question sets menu (vai ser detalhado na secção do API)
 - inputs das questões
 - botões de questionario (summary, detailed, collapse, show)

- botões de question set (show, collapse, permissions)
- progress bar
- hitcount stats
- falar nas versão de apenas uma e duas colunas. Criar imagens com retangulos com transparencia

1.2.3 API

When we went over how question sets were rendered, we saw that each question set had its container and they are only rendered when is accessed for the first time. When it needs to be rendered, API endpoints are used to fetch the HTML of a specific question set. On this HTML returned, each question has attached their client-side validation code. The goal is to remove all existing code of client-side validation, making use of only simple built-in HTML validations on the client-side and use Django's forms framework to have all remaining validations on the server-side.

Before going into more details let's go over Django's built-in form system and some of its components. A form in HTML is represented through the form tag and fields are represented through input tags. Each input tag has a type attribute that defines how the data will be inserted by the users, for example, to insert a date the user will be presented with a calendar where it can click on the desired date. These different ways to insert data are called widgets. To work with forms, Django only makes use of two HTTP request methods: GET and POST. GET is used by browsers to fetch the page/HTML of an URL and it can also be used to fetch information. POST on the other side is used to send information to a server. On the backend side, Django has three main classes that handle the data around forms. The main one is the *Form* class that defines how it works and how it will be presented to the user. Then there is the *Field* class that describes a field of a form, which is in charge of performing field-specific validations. Django already has some implementations of this class for the most common field types such as number, text, choice, ... Finally, the *Widget* class is in charge of processing and transforming the raw data received and also preparing and restructure data to be presented to the user, and again Django already has some implementations of this class. Each *Field* class has a widget class associated. The common development flow is to create a child class of Django's *Form* class and then define its fields with *Field* classes.

However, in the case of MONTRA, the number and type of the fields are dynamic, since both can be customizable by a community manager when he is building a questionnaire. A Submission form was created, child of Django's *Form* class, where its constructor was overridden so the fields of the requested Question Set of a Questionnaire are defined in the form class. Additionally, several question types lead to the need of having to create and associate new *Field* and *Widget* child classes since they were too complex to be represented or validated with the ones that Django already has implemented. To build the widgets of such question types, existing implementations were ported to an associated Django template to then be used in its new widget class.

Considering this, below are presented the endpoints used by the different variations to fetch the HTML of a question set of a specific questionnaire:

```
GET [base url]/questions/[questionnaire id]/[section index]/preview/

GET [base url]/questions/[community slug]/[questionnaire id]/[section index]/search/

GET [base url]/questions/[community slug]/[questionnaire id]/[section index]/ % create
GET [base url]/questions/[questionnaire id]/[section index]/ % create

GET [base url]/questions/[fingerprint hash]/[section index]/ % edit

GET [base url]/questions/[fingerprint hash]/[section index]/show/
```

There are two different endpoints for the create variant, because certain installations are a single community, for that, the community is implicit, but for multi-community installations, the community slug is required, since the same questionnaire can be used on different communities.

For edit and create variants, there needs to be additional endpoints that save the progress when the user changes from one question set to another. Bellow are such:

Submissions Management:

```
POST [base url]/api/submission/save/[community slug]/[questionnaire id]/[section index]/
POST [base url]/api/submission/save/[questionnaire id]/[section index]/
POST [base url]/api/submission/save/[fingerprint hash]/[section index]/
```

Questionnaire Information:

```
GET [base url]/api/questionnaire/info/[community slug]/[questionnaire id]/[section index]/
GET [base url]/api/questionnaire/info/[questionnaire id]/[section index]/
```

Related to the Submissions Management endpoints, which are used to save answers data, the first two endpoints follow the same idea as the ones presented before, which on single community installations there is no need to provide the community slug. However, there is a third alternative. The first two should be used when there is not yet a fingerprint created, and which will return both the fingerprint hash of the created fingerprint and the current submission token. The fingerprint hash is the identifier of a fingerprint, the submission token is used to identify if different calls to the save endpoints are related to the same changes. With that, several changes can be grouped in the same Submission, instead of the previous implementation that would create a FingerprintHead record for each save of a question set.

This also enforces that if the user performs several changes to the same question set with the same submission token, only the last changes will be kept, replacing (figure 1.27), or even deleting (figure 1.28), old answer values sent in the same submission.

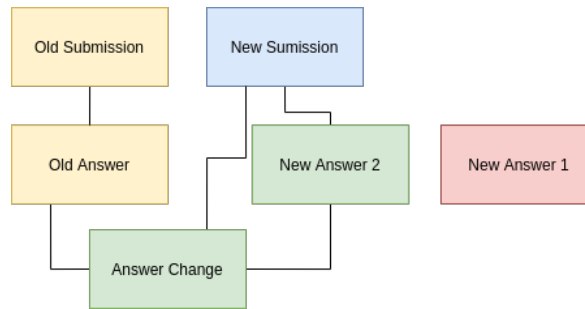


Figure 1.27: When an answer change is submitted and there were already changes made to that question, then the previous one is deleted (New Answer 1) and a new one (New Answer 2) is associated with both the current Submission and the related Answer Change.



Figure 1.28: If the user either provides an empty value or no value to an answer that previously had a value on the current submission, associated Answer and Answer Change records are deleted.

If no submission token or one that does not match the current one is provided, then it is assumed that the changes submitted are related to a new submission, thus the most recent submission is closed and the new provided answers are added to a new submission.

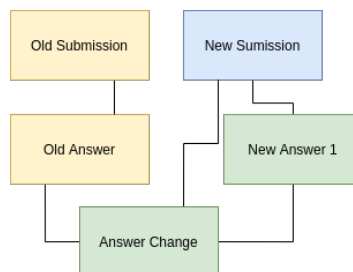


Figure 1.29: Whenever new changes to the answer of a question in a specific submission are submitted, a new Answer model is created, and also an Answer Change record is created connecting with the previous answer. In the case the provided submission token does not match the current one stored in the fingerprint model, then a Submission record will also be created, and changes will be associated with this new Submission.

In such requests, data must be sent on the body encoded in a multipart/form-data format. Such requests are not so trivial to create by hand, however, there are tools and libraries such as Postman⁵ or python's request library⁶ that aid in the process of creating such requests.

As mentioned previously, choice-based answers are stored as a foreign key to the choice(s) selected. This implies that the answer's data sent on the Submissions Management endpoints

⁵<https://www.postman.com/>

⁶<https://docs.python-requests.org/en/master/index.html>

for such question types must be these foreign keys, avoiding misspellings and non-existing choices. However, one does not know the keys for each choice and also the available choice if the web view is not used. For that, the Questionnaire Information endpoints are used to get information of choice-based questions, so that the correct data is sent along with the Submissions Management endpoints.

Draft

There were also some modifications to the endpoints to change the publish status of a fingerprint. Instead of having two different endpoints for different community settings (to auto-accept or not fingerprint publish requests), but for the same purpose, now only one exists.

POST [base url]/api/draft/[fingerprint hash]

This endpoint expects the same data in the same format, though takes different actions according to the community of the fingerprint at issue. Also, on the previous versions of publishing endpoints, there was a possibility to publish fingerprints that weren't complete, not having answers to required answers. Now, a validation is implemented when the user wants to publish its fingerprint, not completing the change on the publish status if the fingerprint is not complete. This avoids notifications being sent to the community manager telling that a user wants to publish a fingerprint, although such fingerprints might be incomplete.

1.2.4 Excel

The previously presented version of the spreadsheet used to define a questionnaire, had some problems in terms of clutter due to the overuse of the "Value list" column to define the extra configuration of specific questions types. The column was used to define all choices and their extra information of choice-based questions, to define columns and their settings of open multiple question and columns, rows, and type of choice tabular questions.

To fix those clutter problems, improvements were done to the spreadsheet by deleting some useless columns that were related to deprecated features and adding new types of rows that define the extra configurations for those questions types that require.

Starting by the columns of the actual spreadsheet now are the following:

- Type
- Text/Question
- Data Type
- Help Text/Description
- Slug
- Dependencies
- Constraints
- Tooltip
- Include in Advanced Search

Most of the columns are already known from the previous version, “Constraints” being the only one that was added to this new version. The columns that were removed are Stats, Comments Stats, Disposition (all associated with deprecated features), Level/Number (the number is now automatically retrieved according to the order of how the rows are defined), and Value List (brought the clutter problem to the table).

Type

On this column, the only possible values available were QuestionSet, Question, and Category. The new version has now eleven possible types of rows for the spreadsheet:

Type	Description
IntroSection	These represent the same concept as QuestionSet. In the previous version, it was possible to add hidden question sets to both the beginning of the end of the questionnaire. This was achieved by setting the Level/Number column to either 0 or 99 respectively. Since the Number portion of that column was removed, such Question Sets must be represented with the IntroSection and CloseSection rows.
Section	
CloseSection	
Group	This is a replacement to the category row type plus the comment question type.
Question	The old and unchanged Question row type
Choice	This row indicates a choice of the previously defined Question
ChoiceInfo	Allows adding an extra question that can be answered if a specific choice is selected. Such a question will be rendered right below the choice. Note that this is different from the dependencies column.
TabularChoice	Row types to define the configuration of choice tabular question types
TabularRow	
Column	Row types to define the configuration of the open multiple question types
ColumnChoice	

Table 1.2: Available values to use on the “Type” Column of the new version of the spreadsheet to define a questionnaire.

Using the new row types described in table 1.2, the spreadsheet itself will be longer, however, it will be easier to read. In figures 1.30, 1.31 and 1.32, it’s presented three before and after of how questions that require extra configuration were and are now defined with the new spreadsheet version.

It was considered to also remove the Level part of the Level/Number column, adding an EndGroup row type. Then whenever the user wanted to create a subgroup of questions after a question would create a Group with empty text and the framework would move the question of that group to a deeper level with no group label. This however would make the spreadsheet even longer for certain questionnaires that make use of categories to group questions that have some dependency on another question. Figure 1.33 shows how the change would affect the spreadsheet.

Even with colors, the user editing has to look several rows above to know at each level it is so he can match the “EndGroud” rows with the respective “Group” rows. This is the same

Type	Text/Question	Data type	Value list
Question	If your database contains vaccine data, please indicate the completeness of recording in the target population (in the database) with respect to each vaccine	choice-tabular	None Partially Complete Complete Don't Know BCG Diphtheria Haemophilus influenzae Hepatitis A Hepatitis B HPV Influenza Measles Meningococcal Mumps Pertussis Pneumococcal Polio Poliovirus Rubella Shingles Tetanus Tick born encephalitis Typhoid Varicella choice

↓

Type	Text/Question	Data type
Question	If your database contains vaccine data, please indicate the completeness of recording in the target population (in the database) with respect to each vaccine	choice tabular single
TabularChoice	None	
TabularChoice	Partially Complete	
TabularChoice	Complete	
TabularChoice	Don't Know	
TabularRow	BCG	
TabularRow	Diphtheria	
TabularRow	Haemophilus influenzae	
TabularRow	Hepatitis A	
TabularRow	Hepatitis B	
TabularRow	HPV	

Figure 1.30: Differences between how a choice tabular question was versus how is now defined on the questionnaire spreadsheet.

Type	Text/Question	Data type	Value list
Question	Clinical test parameters results.	open-multiple-composition	{title:"Weight X", field:"msurex", editor:"input"}, {title:"Height", field:"msurey", editor:"input"}, {title:"Msure Z", field:"msurez", editor:"input"}

↓

Type	Text/Question	Data type
Question	Clinical test parameters results.	open multiple
Column	Weight X	short text
Column	Height	short text
Column	Msure Z	short text

Figure 1.31: Differences between how a open multiple question was versus how is now defined on the questionnaire spreadsheet.


Type	Text/Question	Data type	Value list
Question	Does this take place in participants' homes or at a central location?	choice-multiple-freeform	Home{...} Central{...}

↓

Type	Text/Question	Data type
Question	Does this take place in participants' homes or at a central location?	multiple choice
Choice	Home	
ChoiceInfo		short text
Choice	Central	
ChoiceInfo		short text

Figure 1.32: Differences between how a open multiple question was versus how is now defined on the questionnaire spreadsheet. Note that now it is possible to specify the data type of the extra information that is attached to the extra information of a choice.

Type	Text/Question	Level/Number
Category	Treatments	h1
Question	Relapse Therapy	h2
Question	What is the requirement of this data?	h3
Question	Number of registrations	h3
Category	Disease Modifying Treatments (DMTs)	h2
Question	Past disease modifying therapies	h3
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Start and end dates of past treatments	h4
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Current disease modifying therapies	h3
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Start date of current treatment	h4
Question	What is the requirement of this data?	h4
Question	Number of registrations	h4
Question	Reasons for discontinuation of DMTs	h3



Type	Text/Question
Group	Treatments
Question	Relapse Therapy
Group	
Question	What is the requirement of this data?
Question	Number of registrations
EndGroup	
Group	Disease Modifying Treatments (DMTs)
Question	Past disease modifying therapies
Group	
Question	What is the requirement of this data?
Question	Number of registrations
Question	Start and end dates of past treatments
Question	What is the requirement of this data?
Question	Number of registrations
EndGroup	
Question	Current disease modifying therapies
Group	
Question	What is the requirement of this data?
Question	Number of registrations
Question	Start date of current treatment
Question	What is the requirement of this data?

Figure 1.33: A possible solution to avoid relying on the Level column to create subgroups of questions. On the new solution proposed it's much difficult to know at which level the specific question is located, without looking at the rows above.

problem as matching if open and closing brackets on C-like programming languages. Such a problem is alleviated since indentation is allowed, which can't be achieved on a spreadsheet. For those reasons, the Level column was kept, however, when a questionnaire is imported, whenever there are nested levels the framework will create groups with empty text.

Data Type

As it was possible to see in figures 1.30, 1.31 and 1.32, the Data Type column is used to detail the type of data that will be stored and how it will be requested to the user in terms of HTML widgets.

Table 1.1 contains all the previously allowed question types, a total of twenty-five different types. On the new version, this list was reduced, where some questions can now be achieved using simple question types in conjunction with the new row types that extend their base functionality.

Next is the list of question types available after the refactoring:

- short text: Old open. It replaces the open-validated field by making use of the new Constraints column;
- long text: Old open-textfield
- single choice: Used to achieved any old single-choice question type. The extra information input can now be achieved with the ChoiceInfo row type;
- multiple choice: Used to achieved any old multiple-choice question type. The extra information input can now be achieved with the ChoiceInfo row type;
- integer: Old integer;
- date: Old datepicker;
- email: Old email;
- url: Old url;
- numeric: Old numeric;
- publication: Old publication;

Question Type(s)	Constraint
any	required
short text long text	min_length
	max_length
	regex
integer numeric	min_value
	max_value
date	min_date
	max_date
	format

Table 1.3: Newly available constraints to apply to a question

- image: Old open-upload-image;
- choice tabular: Old choice-tabular;
- open multiple: Can achieve both the old open-multiple and open-multiple composition

Open-location and timeperiod were removed since they weren’t being used on any of the active installations of the MONTRA framework. The same happened with sameas and custom as additionally, they were shortcut type questions.

Slug and Dependencies

Previously the dependencies between questions were detailed by providing the id of the target question and the order of the choice that was needed to be selected, but since now the choices are described by row, the user can also detail an id for them, which such id can then be used on the Dependencies column of the questions that depend on such choice.

Constraints

The old Question model allowed to add custom checks, however, this feature was not possible to make use of through the spreadsheet, it was only possible to define them once the questionnaire was imported. This “Constraints” column intends to enable configuring such additional checks. Checks were previously stored in a custom format, where each check was separated by a space, defined as *name = "value"*. To avoid having to parse a string every time we want to load the constraints, they are now stored in a JSON field. In terms of the spreadsheet, the format chosen was YAML Ain’t Markup Language™ (YAML) [3] since the previously used format deals with values in strings, which makes the validation of the import process much harder. JSON was also considered, however it was too much verbose to put on a spreadsheet cell, emerging the clutter problem. Besides YAML being more user-friendly, it easily maps to a JSON object to them be stored along with the new Question model.

Currently, the supported constraints are the following:

Automatic Metadata Extraction and Update

2.1 REQUIREMENT ANALYSIS

2.2 EXTRACTION

- a ideia geral em mente
- diagrama high-level do data flow

2.2.1 ACHILLES

- organização interna
- implementado em R
- diferentes maneiras de exportação (json, csv ou diretamente para db)
- a query for each analysis
- Catalogue Export

2.2.2 Asynchronous Message Systems

RabbitMQ

Kafka

2.2.3 Kafka Source Connectors

- fetch from files
- fetch from tables/sql

2.3 PUBLISHING

- need to send custom data on a CUSTOM FORMAT to a custom REST endpoint
- Kafka Sink Connectors (The target REST API is not customizable, such as the data format) - need for a sender application
- No known end on kafka topics/streams - require some management on top of kafka

2.4 NETWORK MANAGER

- centralized entity
- transforms the data to the required format
- sends to the specific application's endpoint
- 4 components

2.4.1 Pipelines Workers

- select and filter the data coming from the databases

2.4.2 Sender

- sends/publishes the data resulting from the pipelines, to the application's endpoints

2.4.3 Orchestrator

- To ensure multiple records of multiple databases are not processes at the same time, this component redirects the data from databases to the pipelines workers preventing the previous problem

2.4.4 Admin Dashboard

- Allows to create new pipelines and add applications

2.5 NETWORK DATA FLOW

- Evaluation?
- step by step
- number of topics involved

References

- [1] *Django project*, <https://www.djangoproject.com/>, Accessed on Jul. 2021.
- [2] *Client-side form validation*, https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation, Accessed on Aug. 2021.
- [3] *The official yaml web site*, <https://yaml.org/>, Accessed on Sep. 2021.