Mathesar

Google Summer of Code 2023

Support importing Excel and JSON files.

Anshuman Maurya

Synopsis

Mathesar is an open-source, web-based database management tool that is easy to use, flexible, and scalable. It provides users a simple, intuitive interface for managing their databases, including creating and editing tables, defining relationships between tables, and querying data.

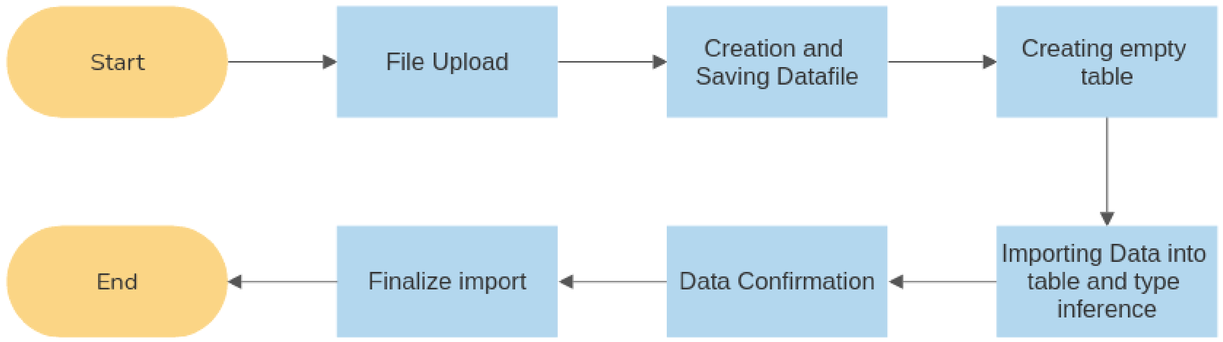
Mathesar UI allows users to import data from CSV and TSV files. We aim to expand this functionality into importing JSON and Excel files too. The project will allow users to import JSON and Excel files and create a table from the imported data. The user can preview the table, remove/rename columns, change data types, and more. If the imported file is not in a suitable format for creating a table, we will develop algorithms to convert the file into a suitable format. If the algorithm doesn't work, the system will refuse to import the file and give the user a readable error.

The project will also include a column data type guessing functionality during the import process (will reuse and refactor existing functions to achieve the same). This will help users save time and effort manually assigning column data types.

Implementation

Current Workflow

The current flow for importing data from a file:


Start
File Upload
Creation and Saving Datafile
Creating empty table
Importing Data into table and type inference
Data Confirmation
Finalize import
End

1. Upload the file/download URL.

2. If the download URL has been provided, download the file. Create the Datafile object providing the following parameters:
    a. file (raw file)
    b. base_name (name of the file)
    c. created_from (whether the file has been downloaded or directly uploaded)
    d. header
    e. delimiter
    f. escapechar
    g. quotechar

    Save the data file.

3. Create the table using the '**create_table_from_datafile()**' method in the '*mathesar/utils/tables.py*' file.

4. Import data into the created table using the '**insert_records_from_json()**' method in the '*db/records/operations/insert.py*' file. The method uses SQLAlchemy code to implement **COPY** command in Postgres to read data from

the CSV file into the Postgres table.

Type Inference Logic

The table initially created has all the columns with the data type **PostgresType.TEXT**. However, we infer the data types of the columns from the data added, the logic for which is present in the *'db/columns/operations/infer_types.py'* file. We have defined a TYPE_INFERENCE_DAG dict that contains the sequence of possible data types to which a particular data type can be inferred. We recursively iterate through the sequence of types trying to alter the column's type to them.

Importing JSON files

A table in JSON can be structured in the following ways. For example,

- **Array of Objects**
  A table can be represented as an array of objects, where each object in the array represents a row in the table, and the keys in the object represent the column names. For example:

```
[
 { "name": "John", "age": 25, "city": "New York" },
 { "name": "Jane", "age": 30, "city": "Chicago" },
 { "name": "Bob", "age": 40, "city": "San Francisco" }
]
```

- **Object with Arrays**
  A table can also be represented as an object with arrays as values, where each array represents a column in the table.
  For example:

```
{
 "name": ["John", "Jane", "Bob"],
 "age": [25, 30, 40],
 "city": ["New York", "Chicago", "San Francisco"]
}
```

- **Array of Arrays**
  A table can be represented as an array of arrays, where each inner array represents a row in the table. Each element in the inner array represents a value in the row.
  For example:

```
[
 ["name", "age", "city"],
 ["John", 25, "New York"],
 ["Jane", 30, "Chicago"],
 ["Bob", 40, "San Francisco"]
]
```

- **Object with Objects**
  A table can also be represented as an object with objects as values, where each key in the outer object represents a row in the table and each inner object represents the column values for that row.
  For example:

```
{
 "row1": { "name": "John", "age": 25, "city": "New York" },
 "row2": { "name": "Jane", "age": 30, "city": "Chicago" },
 "row3": { "name": "Bob", "age": 40, "city": "San Francisco" }
}
```

A table structured as **Array of Objects** can be easily inserted into Postgres table using the solution described below.

Tables structured in other ways must be processed before we import them into Postgres tables. We can easily write such utility functions in Python to convert data from other JSON structures to **Array of Objects** form. We'll also carry out the validation and normalization step (explained below) to make sure JSON objects are consistent.

However, JSON files structured in some forms might not be correctly imported:

- An invalid JSON file that is not correctly structured. This will be caught when we use the method json.loads() which raises ValueError and JSONDecodeError, respectively. An example of such file:

```
[{ "name": "John", "age": 25, "city": "New York"]
```

- A JSON file containing objects with duplicate keys. Though we validate and normalize JSON objects (explained below), JSON objects having duplicate keys would result in overwriting of data (and the overwritten value would be set to null due to the normalization procedure). An example of such file:

```
[
 { "name": "Xyz", "age": 21, "city": "London" },
 { "name": "Abc", "name": 21, "city": "Mumbai" }
]
```

- A JSON file with a table structured in '**Array of Arrays**' form, but missing the first array with column names. In this case, we might name columns with data items to be present in the first row. An example of such file:

```
[
 ["John", 25, "New York"],
 ["Jane", 30, "Chicago"],
 ["Bob", 40, "San Francisco"]
]
```

- A JSON file with an invalid escape character (like "\x") in any of its data item. This will be caught by the method json.loads(), which raises JSONDecodeError. An example of such file:

```
[
 { "name": "Abc", "age": 21, "city": "Vns" },
 { "name": "Abc\x", "age": 21, "city": "Vns" }
]
```

The proposed solution to import JSON files makes changes in the following steps:

Creation and Saving Datafile

Following changes are to be made while creating and saving the Datafile:

- Update the **DataFile** model in the *'mathesar/models/base.py'*
  Add an attribute 'type' with signature

  ```
  CharField(max_length=10, default='CSV', blank=True)
  ```

  This attribute will be used to know which type of file we are dealing with (example: CSV, TSV, JSON, etc.)

- Create a '**json.py**' file in the '*mathesar/imports*' directory.
  This file will store all utility functions that import data into a table from a JSON file. This will also store the functions that convert JSON files with complex data structures into JSON files with data structured as an **Array of Objects.**

- Update the '**create_datafile()**' method in the *'mathesar/utils/datafiles.py'* file.
  We will first check the file type before creating the Datafile object. We can use the following line of code to get the file extension:

  ```
  os.path.splitext(raw_file.name)[1]
  ```

  For files ending with the extension ".csv" or ".tsv," we will follow the current workflow and create the DataFile object.

  Regarding JSON files, we will first identify how the table is structured in JSON using simple utility methods. Code link. We'll then convert complex JSON files into standard JSON files (Array of objects form) using methods defined in '*mathesar/imports/json.py.*' For example, code to convert a table structured as "Object of Arrays" in JSON file to "Array of Objects" is given below. [Link]

  Next, we will deal with the JSON files with inconsistent JSON objects. For example, consider the following JSON file:

```
[
   {
       "first_name":"Matt",
       "last_name":"Murdock",
       "gender":"Male"
   },
   {
       "first_name":"John",
       "last_name":"Doe",
       "email":"jd@example.org",
       "gender":"Male"
   },
   {
       "first_name":"Frank",
       "last_name":"Castle",
       "gender":"Male"
   }
]
```

The second JSON object has field “email” which is not defined in other two objects. Before creating a table, we want to create columns for all the fields and set it to null wherever required. We will use the following code to validate and normalize JSON files against such inconsistencies.

Once we have the file normalized, we will create the DataFile object setting the ‘type’ attribute of the DataFile object according to the file's extension.

Creating an Empty table

Following changes are to be made while creating an empty table:

- Define utility methods in the ‘*mathesar/imports/json.py*’ file. These methods serve a similar purpose as their counterparts already defined for CSV files.
    - get_column_names_from_json()

This method will extract all keys in the JSON objects to be used as table columns. Code Link

  - create_db_table_from_json_data_file()
    This method creates an empty table with all column names with TEXT data type and calls methods for importing data into the table.
    Code link

  - create_table_from_json()
    This method calls the methods for creating the Postgres table from the JSON data file and saves it. Code link

- Update '**create_table_from_datafile()**' method in '*mathesar/utils/tables.py*' file. The method will now check if the file is of type "JSON," "XLSX," or "CSV/TSV" and call respective utility methods. Code link

Importing data into table

Following changes are to be made while importing data from a JSON file into a table:

- Create the '**insert_records_from_json()**' method in the '*db/records/operations/insert.py*' file. Code link.
  This method ensures that if the item's data type is dict or list, it is stringified first and then inserted in TEXT form. This way, our type inference logic kicks in later on and gets converted to **MathesarCustomType.MATHESAR_JSON_OBJECT** and **MathesarCustomType.MATHESAR_JSON_ARRAY** later on.
  This method gets called by the utility methods defined in the '*mathesar/imports/json.py*' file. The method '**insert_record_or_records**' is already defined in the '*db/records/operations/insert.py*' file.

Testing the functionality

The tests for testing datafile API are in the '*mathesar/tests/api/test_data_file_api.py*' file. We'll update the existing tests to cover the cases when we import JSON data files. A new file, **'test_json.py',** will be created in the '*mathesar/tests/imports*' directory. The

tests for utility functions for validating and importing JSON files will be present in this file. They will directly call the functions and check if they perform as expected with the test data.
We'll create the test dataset in the '*mathesar/tests/data/json_parsing*' folder. We'll try to include files:

- that use every form of table structuring discussed above (Array of objects, Object of Arrays, etc.).
- that cover all possible edge cases discussed above (poorly structured JSON, containing invalid escape characters, etc.).
- that have inconsistent objects to ensure our validation and normalization code works perfectly.

The tests for type inference from **PostgresType.TEXT** to **MathesarCustomType.MATHESAR_JSON_OBJECT** and **MathesarCustomType.MATHESAR_JSON_ARRAY** have already been defined in the file '*db/tests/tables/operations/test_infer_types.py*.'

Importing EXCEL files

Working with Excel files is easier than importing JSON because of the functions provided by the **pandas** library.
Our proposed solution uses **pandas.read_excel()** method that supports .xls, .xlsx, .xlsm, .xlsb, .odf, ods, and .odt file extensions.

The proposed solution makes changes in the following steps:

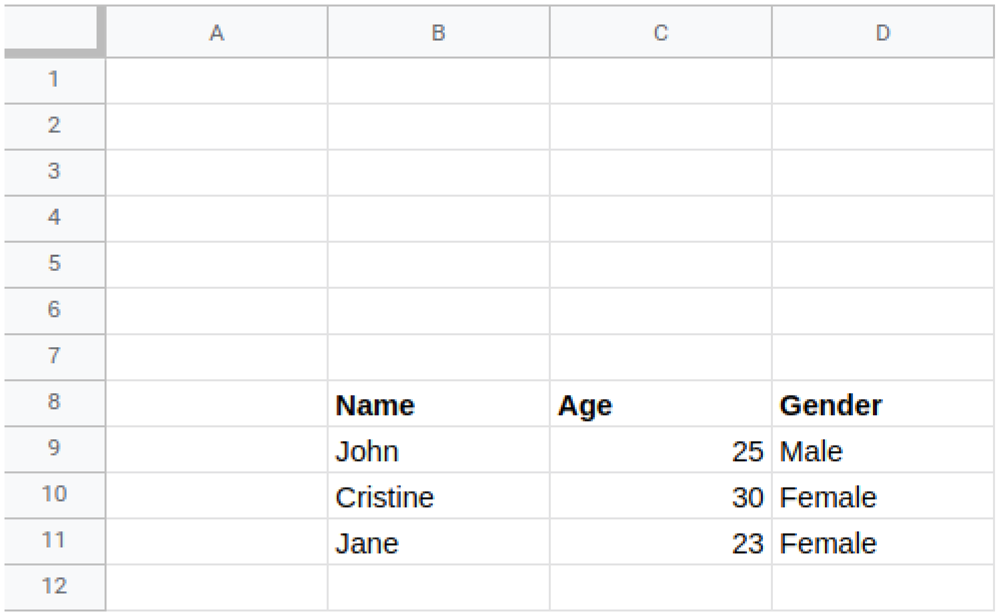
Creation and Saving Datafile

Following changes are to be made while creating and saving the Datafile:

- Create an '**excel.py**' file in the '*mathesar/imports*' directory.

This file will store all utility functions used to import data into a table from an Excel file.

- Update '**create_datafile()**' method in '*mathesar/utils/datafiles.py*' file.
  We'll first check the type of file using the code discussed above. For files of type "CSV/TSV", we will follow the current workflow. For the files of the type "JSON", we will follow the workflow discussed above.
  Regarding "EXCEL" files, we will first check if the data is not present at the beginning of the file. For example:



	A	B	C	D
1				
2				
3				
4				
5				
6				
7				
8		**Name**	**Age**	**Gender**
9		John	25	Male
10		Cristine	30	Female
11		Jane	23	Female
12				

To handle these cases, we can use the code below:

```
df = pd.read_excel(raw_file.file)

if df.iloc[0].isna().any():

   # drop rows with all NaN values
   df.dropna(how='all', inplace=True)

   # drop columns with all NaN values
   df.dropna(axis=1, how='all', inplace=True)

   df.columns = df.iloc[0]
```

```
    df = df[1:]
    df.to_excel(target_file, index=False, header=True)
```

Once we get the normalized excel file, with the column names in the first row, and data records starting from the second row, we can easily use pandas to import data into a table later.

Creating an Empty table

Following changes are to be made while creating an empty table:

- Define utility methods in the '*mathesar/imports/excel.py*' file. These methods serve a similar purpose as their counterparts already defined for CSV files.
    - get_column_names_from_excel()
      This method will extract all column names from the Excel file. Code link

    - create_db_table_from_excel_data_file()
      This method creates an empty table with all column names with TEXT data type and calls methods for importing data into the table.
      Code link

    - create_table_from_excel()
      This method calls the methods for creating the Postgres table from the Excel data file and saves it.
      Code link

- Update '**create_table_from_datafile()**' method in '*mathesar/utils/tables.py*' file. The method will now check if the file is of type "JSON," "XLSX," or "CSV/TSV" and call respective utility methods. Code link

Importing data into table

Following changes are to be made while importing data from an Excel file into a table:

- Create the '**insert_records_from_excel()**' method in the '*db/records/operations/insert.py*' file. Code link

  This method gets called by the utility methods defined in the '*mathesar/imports/excel.py*' file. The method '**insert_record_or_records**' is already defined in the '*db/records/operations/insert.py*' file.

Testing the functionality

The tests for testing datafile API are in the *'mathesar/tests/api/test_data_file_api.py'* file. We'll update the existing tests to cover the cases when we import Excel data files. A new file, **'test_excel.py',** will be created in the '*mathesar/tests/imports*' directory. The tests for utility functions for validating and importing Excel files will be present in this file. They will directly call the functions and check if they perform as expected with the test data.
We'll create the test dataset in the '*mathesar/tests/data/excel_parsing*' folder. We'll try to include files:

- with tables present in the center of the file and not at the start of the file to make sure our logic for the same works perfectly.
- with multiple tables present in a single sheet.
- with messy tables having inconsistent rows.

API spec

We won't require any new API endpoints for importing JSON and Excel files. We will simply tweak some already-defined functions and create new ones that will not change the current API structure.

UX Design

Only UI change in this project will show users that they can import JSON and Excel files along with CSV/TSV files.

Architectural / UX problems

All the problems and their solutions have been discussed above.

External Dependencies

For parsing excel files, we'll require the following library:

- pandas

Research and References

Regarding Importing JSON files

- I went through Konbert's tutorial to import JSON into Postgres using COPY command [Link] as suggested in the Project wiki [Link]. However, the wiki involves us using the COPY command with the syntax:

  ```
  \COPY temp (data) FROM 'my_data.json';
  ```

  When I wrote the SQLAlchemy code for the same and tried to execute it, it showed an error saying that Postgres did not have access to the dynamically saved file (which we received as form input from the User). This might be the reason we do not use `COPY {relation} FROM <filename.csv>;` syntax while importing CSV / TSV files and use the `COPY {relation} FROM STDIN ...` syntax instead.

- I investigated other ways to use COPY command to import JSON directly into the Postgres table in the official documentation [Link].
  To use the `COPY {relation} FROM STDIN ...` syntax to import JSON as we do while importing CSV, I designed the insert function in this way:

  ```
  def insert_records_from_json(
          table, engine, json_filepath, column_names
  ```

```
):
   with open(json_filepath, "r") as json_file:
       with engine.begin() as conn:
           cursor = conn.connection.cursor()

           relation = sql.SQL(".").join(
               sql.Identifier(part) for part in (
                   table.schema, table.name
           ))
           formatted_columns = sql.SQL(",").join(
               sql.Identifier(column_name)
               for column_name in column_names
           )
           copy_sql = sql.SQL(
               "COPY {relation} ({formatted_columns}) FROM STDIN
(FORMAT JSON)"
           ).format(
               relation=relation,
               formatted_columns=formatted_columns
           )
           cursor.copy_expert(copy_sql, json_file)
```

However, I got an error that FORMAT JSON was not supported with STDIN in the current Postgres version.

I also tried to get a workaround and used the *cursor.copy_from()* method but was still hitting same errors:

```
table_name = f'"{table.schema}.{table.name}"'
with open(json_filepath, 'r') as f:
     cursor.copy_from(f, table_name, format='json')
```

- Other useful resources:
  - JSON module documentation [Link]
  - Stackoverflow [Link]

Regarding Importing Excel files

Some useful resources:

- Pandas library official documentation [Link]
- Tutorial on using Pandas to read messy Excel files [Link]

Timeline and Deliverables

	TIMELINE	DELIVERABLES
MILESTONE 1	Week 1 May 29, 2023 - Jun 4, 2023	Write conversion functions for all complex JSON structures into standard JSON (Array of objects form).
	Week 2 Jun 5, 2023 - Jun 11, 2023	Write tests for all the conversion functions written in week one to ensure that code coverage does not regress.
	Week 3 Jun 12, 2023 - Jun 18, 2023	Write functions for validating and normalizing JSON files. Write functions to import data from standard JSON to Postgres table.
	Week 4 Jun 19, 2023 - Jun 25, 2023	Write tests for all the validation and import functions written in week three to ensure that code coverage does not regress.
	Week 5 Jun 26, 2023 - Jul 2, 2023	Update tests written for existing APIs (for saving datafile, table preview, and data confirmation) to test JSON import too.
	Week 6 Jul 3, 2023 - Jul 9, 2023	Update the UI to allow users to import JSON files. Update the UI to show error messages to the user on JSON import failure.
	Week 7 Jul 10, 2023 - Jul 16, 2023	Check the complete JSON import feature end-to-end and remove any bugs.

	Week 8 Jul 17, 2023 - Jul 23, 2023	Buffer
MILESTONE 2	Week 9 Jul 24, 2023 - Jul 30, 2023	Write functions for preprocessing Excel files. Write functions to import data from Excel files to Postgres table.
	Week 10 Jul 31, 2023 - Aug 6, 2023	Write tests for all the import functions written in week eight to ensure that code coverage does not regress.
	Week 11 Aug 7, 2023 - Aug 13, 2023	Update tests written for existing APIs (for saving datafile, table preview, and data confirmation) to test Excel import too.
	Week 12 Aug 14, 2023 - Aug 20, 2023	Update the UI to allow users to import Excel files. Update the UI to show error messages to the user on Excel import failure.
	Week 13 Aug 21, 2023 - Aug 27, 2023	Check the complete Excel import feature end-to-end and remove any bugs.
	Week 14 Aug 28, 2023 - Sep 3, 2023	Buffer

Questionnaire

- Why are you interested in working on Mathesar?

  Mathesar is an open-source project with the aim of making data management and collaboration more accessible and easier for everyone. This is an important goal, as many people struggle with managing and sharing data effectively, and a tool like Mathesar could help solve this problem.

  I love to contribute to open-source projects and I appreciate the organization's commitment to open-source software and its values of collaboration and community. I found the community very welcoming and I am happy to be part of a team that shares equal enthusiasm and motivation toward the project. Moreover, it would also help me grow my software development skills and understand how production-grade software is actually developed.

- Why are you interested in working on this project idea?

  While glancing through the project ideas, I found this one quite engaging. The project promises an amazing feature addition to Mathesar, and allowing users to directly import JSON and Excel into the Postgres table would definitely be a great help to them. Moreover, this project seems to be an excellent opportunity to develop my software development skills and I would be thrilled if I could contribute to this.

- What about your skills and experience makes you well-suited to take on this project?

  I started my development journey in 2021 and have learned quite a few technologies since then. I am familiar with Python and Typescript and various frontend and backend frameworks like Svelte, Vue, and Django. I also attended a DBMS course in my last semester where I studied efficient database design, mostly dealing with SQL databases.

  I am also an active open-source contributor and was a member of Oppia for about a year. I successfully merged 34 PRs into the codebase adding some 13k+ lines in Oppia's codebase. The list of my PRs at Oppia can be found here. I have been contributing to Mathesar for the past month and solved some of the issues, gaining experience and familiarity with the codebase.

  Along with this, I also worked on many group and personal projects which can be viewed on my GitHub profile.

- Do you have any other commitments during the program period? Provide dates, such as holidays, when you will not be available.

  I do not have any other plans and commitments this summer except GSoC. I'll have my mid-semester exams somewhere in the month of July, the dates of which have not been announced yet. However, I have allotted a week of buffer period in each half of the GSoC coding period to meet these commitments.

- If your native language is not English, are you comfortable working closely with a mentor in English?

  I have Hindi as a native language but I am comfortable conversing in English.

- Have you worked on a project remotely and/or with people in other time zones before? If you have, please provide details.

  Yes, I participated in GSoC 2022 where my mentor worked in Pacific Standard Time while I spent my time in Indian Standard Time. We had a time difference of approximately 13.5 hours but we were able to schedule meetings and carry out conversations seamlessly.

- Are you interested in contributing to Mathesar after the program is complete?

  I'd love to!

General Information

About Me

I am Anshuman Maurya, a 3rd-year undergraduate student pursuing Computer Science and Engineering from the Indian Institute of Technology BHU Varanasi, India. I am a software developer with experience in building web applications using various programming languages and frameworks. I am passionate about creating solutions that are scalable, efficient, and user-friendly.

I have a strong foundation in software engineering principles, and I enjoy exploring new technologies and approaches to improve my skills. Apart from this, I enjoy reading books (currently migrating from fiction to self-help) and I write poems in my native language. I believe that my skills and creative approach to problems make me a good fit for this project, and I am excited about the opportunity to contribute to its success.

Contact Information

Full name -	Anshuman Maurya
Email address -	anshumanmaurya111@gmail.com
Matrix username -	iamezio [Display name: Anshuman Maurya]

GitHub username -	IamEzio
Phone number -	+91 8318904105
Emergency contact information -	anshuman.maurya.cd.cse20@itbhu.ac.in +91 8953247217

Education

Institute -	Indian Institute of Technology BHU Varanasi
Degree -	B. Tech
Major -	Computer Science and Engineering
Graduation year -	2024
Courses taken -	Database Management Systems, Operating Systems, Data Structures and Algorithms, Computer Architecture and Artificial Intelligence.

Skills

Skill name	Proficiency (1-5)	Where I've used this skill
Python	4	As a part of the academic course curriculum and many personal projects.
Python Unit Testing	4	While working at Oppia for GSoC'22. [Project Link]
SQL	4	As a part of the academic curriculum and some personal projects.
Vue	5	Many of my personal projects. [Link to my repositories]
Django	4	Some of my personal projects and Mathesar.
Git	4	Collaborating on various open-source projects.

Experience

Brief Description	Relevant Links	Additional notes
Internship: Google Summer of Code 2022	[Project Link]	Project was to achieve 100% Per-File Branch and Line Coverage for the Frontend and the Backend. Gained much experience in software testing.
Open Source: User Checkpoints project @Oppia	#15213 Backend changes for logged-in users. #15482 Frontend changes for logged-out users.	Worked in a team of 3 people to introduce checkpoints in Oppia explorations. The feature allowed users to save their exploration progress and continue later on. Successfully completed the project and received positive feedback from org admins.
Open Source: SVG sanitizer @Oppia	#14876 Made SVG sanitizer for image inputs	SVGs given as input to forms may contain invalid tags and malicious scripts. Made an SVG sanitizer for Oppia which stripped SVGs of such invalid tags and displayed the info to the user.
Club project: Software Development Group website	Merged PRs	Contributed to developing the official website of the Software Development Group (part of the Club of Programmers at IIT BHU Varanasi).

Contributions to Mathesar

Issue Title	Link to issue and/or PR	Additional notes
Add client-side validations for email, username, password length	Issue Link PR link	**[Merged]**
Show appropriate breadcrumbs for /administration/users/* pages	Issue Link PR link	**[Merged]**
When loading the Schema Page, show the correct number of loading skeletons	Issue Link PR link	**[Merged]**
Add more space at the bottom of the Schema Page	Issue Link PR link	**[Merged]**

Code Snippets

Code to identify JSON structure and convert it into Array of objects form

```
import json

with open(json_filepath, 'r') as f:
   data = json.load(f)
```

```
# check if data is an array of objects
if isinstance(data, list) and all(isinstance(item, dict) for item in
data):
   # Follow the normal workflow
   next()

# check if data is an object of arrays
elif isinstance(data, dict) and all(isinstance(value, list) for value in
data.values()):
   # Convert JSON into Array of objects structure
   next()

# check if data is an array of arrays
elif isinstance(data, list) and all(isinstance(item, list) for item in
data):
   # Convert JSON into Array of objects structure
   next()

# check if data is an object of objects
elif isinstance(data, dict) and all(isinstance(value, dict) for value in
data.values()):
   # Convert JSON into Array of objects structure
   next()

# otherwise, structure could not be determined
else:
   raise InvalidJSONError()
```

Code to convert “Object of Arrays” JSON to “Array of Objects”

```
import json

def convert_object_of_arrays_to_array_of_objects(json_filepath):
   with open(json_filepath, 'r') as f:
       data = json.load(f)

   # Get the number of rows in the table (assumes all arrays have the same
length)
   num_rows = len(list(data.values())[0])
   # Initialize an empty list to hold the dictionaries
   array_of_dicts = []

   # Loop through each row
   for i in range(num_rows):
       # Initialize an empty dictionary for the row
       row_dict = {}

       # Loop through each column
       for column_name in data:
           # Get the value of the current column for the current row
           column_value = data[column_name][i]

           # Add the value to the row dictionary with the column name as
the key
           row_dict[column_name] = column_value

       # Add the row dictionary to the array of dictionaries
       array_of_dicts.append(row_dict)

   return array_of_dicts
```

Code to validate and normalize JSON files

```
import json

def validate_and_normalize_json_file(filename):
   with open(filename, 'r') as json_file:
       # Load the JSON data from the file
       data = json.load(json_file)

       # Get the set of all keys in the JSON objects
       all_keys = set()
       for obj in data:
           all_keys |= set(obj.keys())
```

```
        # Add any missing keys to each object with an empty value
        for obj in data:
            for key in all_keys:
                if key not in obj:
                    obj[key] = ""

    # Write the normalized data back to the file
    with open(filename, 'w') as json_file:
        json.dump(data, json_file)
```

Code to get column names from JSON

```
def get_column_names_from_json(data_file):
    with open(data_file, 'r') as f:
        data = json.load(f)

    if isinstance(data, list):
        return list(data[0].keys())
    return list(data.keys())
```

Code to create DB table from JSON

```
def create_db_table_from_json_data_file(
        data_file, name, schema, comment=None
):
    db_name = schema.database.name
    engine = create_mathesar_engine(db_name)
    json_filepath = data_file.file.path
    header = data_file.header
    column_names = [
        column_name.strip() for column_name in
        get_column_names_from_json(data_file.file.path)
    ]
```

```python
column_names = [
    f"{COLUMN_NAME_TEMPLATE}{i}" if name == '' else name
    for i, name in enumerate(column_names)
]
column_names_alt = [
    fieldname if fieldname != ID else ID_ORIGINAL
    for fieldname in column_names
]
table = create_string_column_table(
    name=name,
    schema=schema.name,
    column_names=column_names,
    engine=engine,
    comment=comment,
)
try:
    insert_records_from_json(
        table,
        engine,
        json_filepath
    )
    update_pk_sequence_to_latest(engine, table)
except (IntegrityError, DataError):
    drop_table(
        name=name, schema=schema.name, engine=engine)
    table = create_string_column_table(
        name=name,
        schema=schema.name,
        column_names=column_names_alt,
        engine=engine,
        comment=comment,
    )
    insert_records_from_json(
        table,
        engine,
        json_filepath
    )
reset_reflection(db_name=db_name)
return table
```

Code to create and save the table from JSON

```
def create_table_from_json(
       data_file, name, schema, comment=None
):
   engine = create_mathesar_engine(schema.database.name)
   db_table = create_db_table_from_json_data_file(
       data_file, name, schema, comment=comment)
   db_table_oid = get_oid_from_table(
       db_table.name, db_table.schema, engine)

   table = Table.current_objects.get(
       oid=db_table_oid,
       schema=schema,
   )
   table.import_verified = False
   table.save()
   data_file.table_imported_to = table
   data_file.save()
   return table
```

Code for updated ‘create_table_from_datafile’ method

```
def create_table_from_datafile(
       data_files, name, schema, comment=None
):
   data_file = data_files[0]
   if data_file.type == "JSON":
       table = create_table_from_json(
           data_file, name, schema, comment=comment)
   elif data_file.type == "XLSX":
       table = create_table_from_json(
           data_file, name, schema, comment=comment)
   else:
       table = create_table_from_csv(
           data_file, name, schema, comment=comment)
   return table
```

Code to insert data from JSON into table

```
def insert_records_from_json(table, engine, json_filepath):
    with open(json_filepath, 'r') as json_file:
        data = json.load(json_file)

    for i, row in enumerate(data):
        data[i] = {
            k: json.dumps(v)
            if (isinstance(v, dict) or isinstance(v, list))
            else v
            for k, v in row.items()
        }
    insert_record_or_records(table, engine, data)
```

Code to get column names from Excel file

```
def get_column_names_from_excel(data_file):
    df = pd.read_excel(data_file)
    return list(df.columns)
```

Code to create DB table from Excel file

```
def create_db_table_from_excel_data_file(
        data_file, name, schema, comment=None
):
    db_name = schema.database.name
    engine = create_mathesar_engine(db_name)
    json_filepath = data_file.file.path
    header = data_file.header
    column_names = [
        column_name.strip() for column_name in
        get_column_names_from_excel(data_file.file.path)
    ]
    column_names = [
        f"{COLUMN_NAME_TEMPLATE}{i}" if name == '' else name
        for i, name in enumerate(column_names)
    ]
    column_names_alt = [
        fieldname if fieldname != ID else ID_ORIGINAL
        for fieldname in column_names
    ]
    table = create_string_column_table(
        name=name,
        schema=schema.name,
        column_names=column_names,
        engine=engine,
        comment=comment,
    )
    try:
        insert_records_from_excel(
            table,
            engine,
            json_filepath
        )
        update_pk_sequence_to_latest(engine, table)
    except (IntegrityError, DataError):
        drop_table(
            name=name, schema=schema.name, engine=engine)
        table = create_string_column_table(
            name=name,
            schema=schema.name,
            column_names=column_names_alt,
            engine=engine,
            comment=comment,
        )
        insert_records_from_excel(
            table,
            engine,
            json_filepath
        )
    reset_reflection(db_name=db_name)
    return table
```

Code to create and save table from Excel

```
def create_table_from_excel(
        data_file, name, schema, comment=None
):
    engine = create_mathesar_engine(schema.database.name)
    db_table = create_db_table_from_excel_data_file(
        data_file, name, schema, comment=comment)
    db_table_oid = get_oid_from_table(
        db_table.name, db_table.schema, engine)

    table = Table.current_objects.get(
        oid=db_table_oid,
        schema=schema,
    )
    table.import_verified = False
    table.save()
    data_file.table_imported_to = table
    data_file.save()
    return table
```

Code to insert data from Excel file

```
def insert_records_from_excel(table, engine, excel_filepath):
    df = pd.read_excel(excel_filepath)
    records = json.loads(df.to_json(orient='records'))
    insert_record_or_records(table, engine, records)
```