



**Universidade Federal de Pernambuco**

Centro de Informática

Curso de Ciência da Computação

**Django Datatable Builder: An Improved Approach for Implementing Dynamic Tables in Django**

Trabalho de Conclusão de Curso de Graduação

por

Romero Ramsés Cartaxo Bizarria

Orientador: Prof. Dra. Paola Rodrigues de Godoy Accioly

Recife, Agosto / 2025

Romero Ramsés Cartaxo Bizarria

**Django Datatable Builder: An Improved Approach for Implementing Dynamic Tables in Django**

Monografia apresentada ao Curso de Ciência da Computação,  
como requisito parcial para a obtenção do Título de Bacharel em  
Ciência da Computação, Centro de Informática da Universidade  
Federal de Pernambuco.

Orientadora: Prof. Dra. Paola Rodrigues de Godoy Accioly

Recife  
2025

Ficha de identificação da obra elaborada pelo autor,  
através do programa de geração automática do SIB/UFPE

Bizarria, Romero Ramsés Cartaxo.

Django Datatable Builder: An Improved Approach for Implementing  
Dynamic Tables in Django / Romero Ramsés Cartaxo Bizarria. - Recife, 2025.  
44 p. : il., tab.

Orientador(a): Paola Rodrigues de Godoy Paola Rodrigues de Godoy  
Accioly

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de  
Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado,  
2025.

Inclui referências.

1. Desenvolvimento web. 2. Tabelas dinâmicas. 3. Padrão de Design  
Builder. 4. Arquitetura de software. I. Paola Rodrigues de Godoy Accioly,  
Paola Rodrigues de Godoy. (Orientação). II. Título.

600 CDD (22.ed.)

Romero Ramsés Cartaxo Bizarria

## **Django Datatable Builder: An Improved Approach for Implementing Dynamic Tables in Django**

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Aprovado em: 05 de Agosto de 2025

### **Banca Examinadora**

---

Prof. Dra. Paola Rodrigues de Godoy Accioly (Orientadora)  
Universidade Federal de Pernambuco

---

Prof. Dr. Kiev Santos da Gama (Examinador Interno)  
Universidade Federal de Pernambuco

# Django Datatable Builder: An Improved Approach for Implementing Dynamic Tables in Django

ROMERO CARTAXO, Universidade Federal de Pernambuco, Brazil

Dynamic tables serve as essential components in modern web applications, but integrating rich client-side JavaScript libraries with server-side frameworks like Django presents significant challenges. Existing integration libraries often suffer from being unmaintained, incomplete, or architecturally restrictive. This paper presents **Django Datatable Builder**, a Python library designed to solve this problem by encapsulating the complexity of DataTables.js integration using the Builder design pattern. The library provides a unified, Pythonic API that handles both client and server-side logic, abstracting away repetitive tasks like request parsing, queryset filtering, and response formatting, while adding high-level features such as custom actions and extensible filters. Its effectiveness is demonstrated through the implementation of a sample college management system, which showcases the library's ability to reduce development time, improve code maintainability, and enforce security best practices across various real-world scenarios. The result is a powerful tool that enables developers to build secure, consistent, and feature-rich dynamic tables with minimal code.

CCS Concepts: • **Software and its engineering** → *Object oriented development*; **Software prototyping**; *Software design techniques*; **Application specific development environments**.

Additional Key Words and Phrases: Django, DataTables, server-side processing, dynamic tables, Builder pattern, web framework, Python, interactive UI components, code reusability, Web development

## 1 Introduction

Modern web applications frequently rely on the interactive presentation of data to deliver a high-quality user experience. Dynamic tables that support features like sorting, searching, filtering, and pagination have become a standard expectation for users to explore and manipulate complex datasets effectively. While server-side frameworks like Django provide robust tools for managing data and business logic, and JavaScript libraries like DataTables.js excel at creating interactive front-end components, the integration of these two technologies is a well-known challenge.

The integration challenge manifests as a two-fold problem requiring parallel implementation efforts. On the client side, developers must configure DataTables instances, manage column definitions, handle event listeners and create table templates for each table created. Simultaneously, on the server side, they must parse and validate DataTables-specific request parameters, construct appropriate database queries with filtering and pagination, and format responses according to DataTables' expected JSON structure. This dual implementation often results in tightly coupled code, repeated boilerplate across multiple tables, and increased potential for security vulnerabilities if improperly handled.

To address this, several third-party libraries have been developed to simplify the integration between Django and DataTables. Projects like `django-datatable-view`, `django-ajax-datatable`, and `djangoestframework-datatables` have offered various solutions, from custom class-based views to integrations with Django REST Framework. However, these existing solutions exhibit notable limitations. Many are no longer actively maintained, leaving them incompatible with modern versions of Django or riddled with unresolved issues. Furthermore, these libraries mostly address the server-side aspect of the integration, forcing developers to still write significant JavaScript boilerplate.

This paper introduces **Django Datatable Builder**, a new library that provides a comprehensive and flexible solution by encapsulating the entire integration process within a unified, Pythonic interface. Our approach is centered on the Builder design pattern, which allows for the step-by-step construction of complex table objects through a fluent

API. This method elegantly handles both the client-side JavaScript generation and the server-side data processing, abstracting the underlying complexity away from the developer.

The effectiveness of our library is measured through a detailed report of its abstractions and a practical demonstration. We provide a summary of the numerous implementation steps that the library automates, such as request validation, parameter parsing, queryset slicing, and JSON response formatting, along with the high-level features it introduces, including a system for creating custom actions, mass operations, and extensible filters. To validate these capabilities in a real-world context, we developed the "Greendale College Management System", a case study application that showcases how the library manages complex data models with varied requirements, such as conditional logic, custom data formatting, and relational data filtering.

The results demonstrate that Django Datatable Builder significantly reduces the code and effort required to implement sophisticated dynamic tables. The library promotes code reusability, enforces security best practices like XSS protection, and ensures a consistent look and feel across an entire application, empowering developers to focus on business logic rather than on repetitive integration tasks.

## 2 Key concepts

### 2.1 Model-View-Controller (MVC) Architecture

The Model-View-Controller (MVC) is a software architectural pattern that separates an application into three interconnected components, promoting separation of concerns and facilitating maintainability and scalability in web applications [2].

**2.1.1 Model.** The Model component represents the application's data structure and business logic. It manages data, logic, and rules of the application independently of the user interface. In web frameworks, models typically interact with the database, validate data, and enforce business rules. The model notifies its associated views and controllers when there has been a change in its state, allowing the view to update its presentation and the controller to change the available set of commands.

**2.1.2 View.** The View component handles the presentation layer of the application. It renders the model's data to the user in a specific format and provides the user interface through which users interact with the application. Views retrieve data from the model and determine how that data should be presented. In web applications, views typically generate HTML, JSON, or other output formats that can be rendered by web browsers or consumed by client applications.

**2.1.3 Controller.** The Controller component acts as an intermediary between the Model and View components. It processes user input, manipulates the model based on that input, and determines which view should be rendered in response. Controllers handle request routing, invoke appropriate business logic on models, and select views for rendering responses. They orchestrate the flow of data between the model and view while keeping them decoupled from each other.

Django implements a variation of the MVC pattern, often referred to as Model-Template-View (MTV), where Django's "views" correspond to controllers in traditional MVC, and Django's "templates" correspond to views in traditional MVC. This architectural pattern provides the foundation for Django's design philosophy and influences how developers structure their applications using the framework.

## 2.2 Django

Django serves as a high-level Python web framework [1] that facilitates the development of robust and scalable web applications.

**2.2.1 Models.** A model in Django is a Python class that represents a table in the application's database. It contains the table's fields as attributes and can retrieve data from this table by creating querysets.

**2.2.2 Querysets.** A queryset in Django is an object that represents a database query that can be evaluated to retrieve a set of objects from the database. It uses lazy evaluation, meaning the database query is not executed until the queryset is evaluated.

**2.2.3 Views.** A Django function view is a Python function that receives an HTTP request and returns an HTTP response. It usually links to an endpoint of the application in order to receive the user's requests, perform the necessary logic, and return an HTTP response to the user.

**2.2.4 Class based views.** A class-based view has the same purpose as a function view in Django, but as the name suggests, it is a Python class instead of a function. This allows the user to take advantage of Object-Oriented Programming techniques while preventing code duplication.

**2.2.5 Django Template Language (DTL).** DTL is Django's native template system. Templates in this context are HTML files with additional syntax designed to make the content of the HTML more dynamic. DTL, for example, supports loops, conditionals, variables, custom tags, among other features.

**2.2.6 Template tags.** Functions defined in Python that are executed and evaluated within a Django template.

## 2.3 DataTables

DataTables is a prominent JavaScript library [7] for creating dynamic and interactive tables in web applications. It offers robust and flexible features such as sorting, searching, filtering, and pagination. There are two approaches to providing the data to be displayed with DataTables: client-side rendering and server-side rendering.

**2.3.1 Client-side rendering.** It is the simplest approach to create a dynamic table. The developer needs to create an HTML table with all the data and then initialize a DataTable instance by passing the identifier of the newly created table. The DataTables library will now handle the sorting, pagination, filtering, and data export of the table.

```

1  <table id="example" class="display">
2  <thead>
3  <tr>
4  <th>Name</th>
5  <th>Position</th>
6  <th>Office</th>
7  <th>Age</th>
8  <th>Start date</th>
9  </tr>
10 </thead>
11 <tbody>
12 <tr>
13 <td>Tiger Nixon</td>
14 <td>System Architect</td>
15 <td>Edinburgh</td>
16 <td>61</td>
17 <td>2011/04/25</td>
18 </tr>
19 <tr>
20 <td>Garrett Winters</td>
21 <td>Accountant</td>
22 <td>Tokyo</td>
23 <td>63</td>
24 <td>2011/07/25</td>
25 </tr>
26 <tr>
27 <td>Ashton Cox</td>
28 <td>Junior Technical Author</td>
29 <td>San Francisco</td>
30 <td>66</td>
31 <td>2009/01/12</td>
32 </tr>
33 <tr>
34 <td>Cedric Kelly</td>
35 <td>Senior JavaScript Developer</td>
36 <td>Edinburgh</td>
37 <td>22</td>
38 <td>2012/03/29</td>
39 </tr>
40 </tbody>
41 </table>

```

Listing 1. DataTables client-side HTML initialization



```

1 $(document).ready(function() {
2   $('#example').DataTable();
3 });

```

Listing 2. DataTables client-side JavaScript initialization

This example will render the following table in the browser:

10 ▾ entries per page

Search:

Name	Position	Office	Age	Start date
Ashton Cox	Junior Technical Author	San Francisco	66	2009/01/12
Cedric Kelly	Senior Javascript Developer	Edinburgh	22	2012/03/29
Garrett Winters	Accountant	Tokyo	63	2011/07/25
Tiger Nixon	System Architect	Edinburgh	61	2011/04/25

Showing 1 to 4 of 4 entries

« < 1 > »

Fig. 1. DataTables client-side rendering example

**2.3.2 Server-side rendering.** Server-side rendering becomes essential when dealing with large datasets that would be impractical to load entirely into the browser. Instead of sending all data to the client, the server processes and returns only the data needed for the current table state. This approach significantly improves performance and reduces memory usage for tables containing thousands or millions of records.

With server-side processing, DataTables delegates all operations like pagination, searching, sorting, and filtering to the server. Each user interaction triggers an asynchronous request containing parameters about the current table state. The server processes these parameters, queries the database with the appropriate filters and limits, and returns only the required subset of data.

Since the server handles the data, there is no need to populate the table with data on the client.

To enable server-side processing, developers must set two main configuration options: `serverSide: true` and an `ajax` endpoint. The basic initialization looks like this:

```

1 $('#example').DataTable({
2   serverSide: true,
3   processing: true,
4   ajax: {
5     url: '/api/data',
6     type: 'POST'
7   }
8 });

```

Listing 3. DataTables server-side initialization

DataTables sends several parameters with each request, including the current page position (start and length), search term (search[value]), and sorting information (order array). The server must parse these parameters and construct the appropriate database query.

The server response must follow a specific JSON format:

```

1 {
2   "draw": 1,
3   "recordsTotal": 4,
4   "recordsFiltered": 2,
5   "data": [
6     {
7       "name": "Tiger_Nixon",
8       "position": "System_Architect",
9       "office": "Edinburgh",
10      "age": 61,
11      "start_date": "2011/04/25"
12    },
13    {
14      "name": "Garrett_Winters",
15      "position": "Accountant",
16      "office": "Tokyo",
17      "age": 63,
18      "start_date": "2011/07/25"
19    }
20  ]
21 }
```

Listing 4. DataTables server response JSON format

The draw parameter prevents out-of-order responses, recordsTotal indicates the total dataset size, recordsFiltered shows records after filtering, and data contains the actual table rows for display.

Server-side processing offers crucial advantages for enterprise applications: it handles datasets of any size without browser limitations, reduces initial page load time, minimizes bandwidth usage, and enables real-time data updates. The trade-off is increased server load and the need to implement the server-side logic for data processing.

This project will focus on server-side processing.

## 2.4 Builder Design Pattern

The Builder pattern is a creational design pattern that separates the construction of a complex object from its representation so that the same construction process can create different representations. [3]. It provides a flexible solution for constructing complex objects step by step.

In the context of Django applications, the Builder pattern proves valuable for constructing complex UI components like dynamic tables, where various configurations (columns, filters, actions, export options) need to be assembled in a flexible and reusable manner.

## 3 Motivation

While integrating DataTables with Django can improve the creation of interactive tables, it often introduces significant complexity, particularly for developers unfamiliar with both frameworks. This complexity requires a detailed study to

learn optimal practices and implementation techniques. Developers must also implement this integration across both the client (front-end) and server (back-end), ensuring proper data manipulation and responsive user interfaces.

On the client side, the developer needs to:

- Create a table element with its header and necessary attributes
- Instantiate a DataTable object
- Specify the columns' names
- Specify the order of records
- Specify which columns can be sorted
- Configure the initial page for pagination
- Create event listeners for events like rendering, filtering, sorting, and searching if necessary
- Ensure that the table data is handled correctly to avoid security vulnerabilities

On the server side, the developer needs to:

- Validate and parse DataTables requests to the server
- Keep track of the columns in the table, ensuring the columns have the same name and order as defined in the client
- Handle ordering of the table
- Handle filtering and searching on the table
- Ensure that the table data is handled correctly to avoid security vulnerabilities

Furthermore, repeatedly implementing this integration for multiple dynamic tables leads to a process prone to errors and inconsistencies. A more efficient approach would involve implementing an architecture that encapsulates this integration, allowing code reusability and ensuring standardization across various tables.

Therefore, the primary motivation behind this project is to develop a library that provides such an encapsulated architecture. This aims to free developers from the burden of implementing a standardized, customizable, secure, and thoroughly tested integration, allowing them instead to focus on creating tables aligned with the application's specific business rules.

#### 4 Basic Usage

When using this integration for the first time, developers should create a class that inherits from the `DatatableBuilderBase` class in order to set the options that all of the tables should have. The integration already has a basic table template written in HTML and CSS, but since each project will have its own styling libraries and design principles, the developer can create a custom template and provide the template's path to the inherited class.

After the initial setup, to create a new table, the developer needs to:

- Create a class that inherits from the `Datatable` class initially created. This class needs to have a unique identifier and the model the table will query
- Add the `register_table` decorator to this class
- Register the url that will listen for the requests
- Override the method `build_table` and call the builder methods provided by the integration inside of this method.
- Place the `{% datatable %}` template tag with the id of the datatable inside the template where the table should be rendered.

```

1 class CustomDatatableBuilder(DatatableBuilderBase):
2     table_template = "custom_table.html"

```

Listing 5. Example of a base table class

Given the following Student model:

```

1 class Student(models.Model):
2     full_name = models.CharField(max_length=255)
3     date_of_birth = models.DateField()
4     enroll_date = models.DateField()
5     grade = models.PositiveSmallIntegerField(null=True, blank=True)

```

Listing 6. Example of a Django model

One can create the following table that queries the Student model:

```

1 from django_datatable_builder.datatable_builder_base import DatatableBuilderBase
2 from django_datatable_builder.registry import register_table
3
4 @register_table
5 class StudentTable(CustomDatatableBuilder):
6     id = "student_table"
7     model = Student
8
9     def build_table(self):
10         self.add_column(field_name='enroll_date', title='Enrollment_Date')
11         self.add_column(field_name='date_of_birth', title='Birth_Date')
12         self.add_column(field_name='full_name', title='Full_Name', init_order='asc')
13         self.add_column(field_name='grade', title='Grade', formatter=lambda item: item.grade or '-',
                           orderable=False)

```

Listing 7. Example of a table class

The next step is to create the endpoint that will listen to the requests for this table, along with the other registered urls:

```

1 urlpatterns = [
2     ...,
3     path('datatable/student/', StudentTable.as_view(), name=StudentTable.url_name),
4 ]

```

Listing 8. Example of a url pattern



## 5 Advanced usage

Even though it is not necessary for the developer to create JavaScript files to handle a table created with this library, sometimes developers might need to change the table's behavior directly on the client side through JavaScript. Some examples where this would be necessary are:

- Override default options for the DataTables instance
- Specify additional options on the DataTables instance
- Create a custom filter type
- Perform specific actions after table initialization
- Perform specific actions before or after table updates
- Change how table-related modals are opened and closed

To address this necessity, this library provides a global configuration object on JavaScript. The developer can specify the path to a JavaScript file on the DatatableBuilder class through the `config_script` attribute. In this file, developers can override the default configuration accessing the configuration object keys:

```
1 const datatableConfig = window.datatableBuilderConfig
2 datatableConfig.firstOptionExample = "Example_Value"
3 datatableConfig.secondOptionExample = {"key1": "value1", "key2": "value2"}
```

Listing 10. Example of configuration script defined by the user

We chose this pattern for its simplicity and flexibility. The main drawback of this pattern is the possibility of namespace collisions with other libraries or scripts. However, it is very unlikely that there will be a variable with the name `datatableBuilderConfig` in the global namespace. As a precaution, if this collision happens, the developer can change the name of this variable inside the DatatableBuilder class through the `js_config_name` attribute. This approach also allows different client-side configuration on different tables, even if the tables are rendered on the same page.

While there are more modern ways to implement this customizable configuration without the namespace collision problem, the implementation of these alternatives would vary depending on how and where the application's static files are hosted. Since this is a Django library that can be used in projects that host static files with different approaches, the safest option is to use the global configuration object that is guaranteed to work irrespective of the methods used to host static files.

## 5.1 Available options in configuration object

Table 1. Client-side configuration options

Option	Type	Description	Default
datatableOptions	Object	Additional options to be passed in the DataTables object instantiation	-
customFilters	Object	Specification of custom filter types	-
makeActionRequest	Function	Makes the request to perform an action	Asynchronous request to the action view url
onInvalidAction	Function	Triggered after error executing action	Alert popup
onActionSuccess	Function	Triggered after successful action execution	Updates table and resets action checkboxes
onInit	Function	Triggered after when the table is first rendered	-
onPreDraw	Function	Triggered before the table is updated	-
onDraw	Function	Triggered after the table is updated	-

## 6 Architecture

While this library is implemented in Django, its architecture could be adapted for other web frameworks. However, the framework needs to meet certain requirements that are essential to the correct implementation of this architecture:

- **Support to the MVC pattern**
- **Template Engine**
- **Dynamic injection of partial templates into a template:** In Django, this is supported via template tags. With template tags, One can provide a template path to render it with custom context variables that can also be defined based on the parameters provided when calling the template tag.

## 6.1 Implementation

To implement this integration we will expand on the classic MVC pattern architecture:

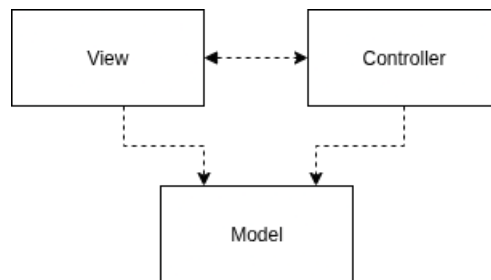


Fig. 3. MVC architecture diagram

Firstly, we will create another controller that will retrieve data from the same model, the **Datatable controller**. This controller will define the table structure and how it should be rendered. It will also be responsible for listening to DataTables.net requests, process it and return the appropriate responses. In addition, this controller needs to have a unique identifier.

Then, we need to create the **template tag** (or the framework's equivalent). This template tag will receive the Datatable Controller identifier to instantiate it and retrieve all the necessary information to render the table. The controller will hold the path to the template used for rendering the table. It will also have a method to return all of the context variables needed by the template. With that information, the template tag can render all of the html, styles and scripts necessary for the requested table.

It is also recommended to have a base template that the developer can inherit to create a custom table template. This base template should primarily have the necessary scripts for the DataTables.net object instantiation on the client-side. Additionally, this base template could also have an html structure and styles that would be necessary for every table, even if they are customized. Then, the developer can inherit this template and customize the relevant parts of the table, eliminating the need to copy and paste code boilerplate from the default template.

The last entities of our implementation are the **Datatable registry** and the **Register decorator**. The Datatable registry is a simple structure that maps the Datatable controller ids to the corresponding controller while the Register decorator is a class decorator [3] that we can use to register the Datatable controller to the Datatable registry. This enables the template tag to retrieve the Datatable controller from its id.

The Register decorator is not necessary for the architecture as there are other ways to register the controller to the registry, but it is a very convenient approach for the developer as it usually means to just add one line above the controller's definition. In some frameworks, it might also be possible to use metaprogramming to add the Datatable controllers to the registry without any explicit declaration. However, this makes it harder for the developer to understand the behavior of the integration and might lead to unexpected behaviors difficult to diagnose.



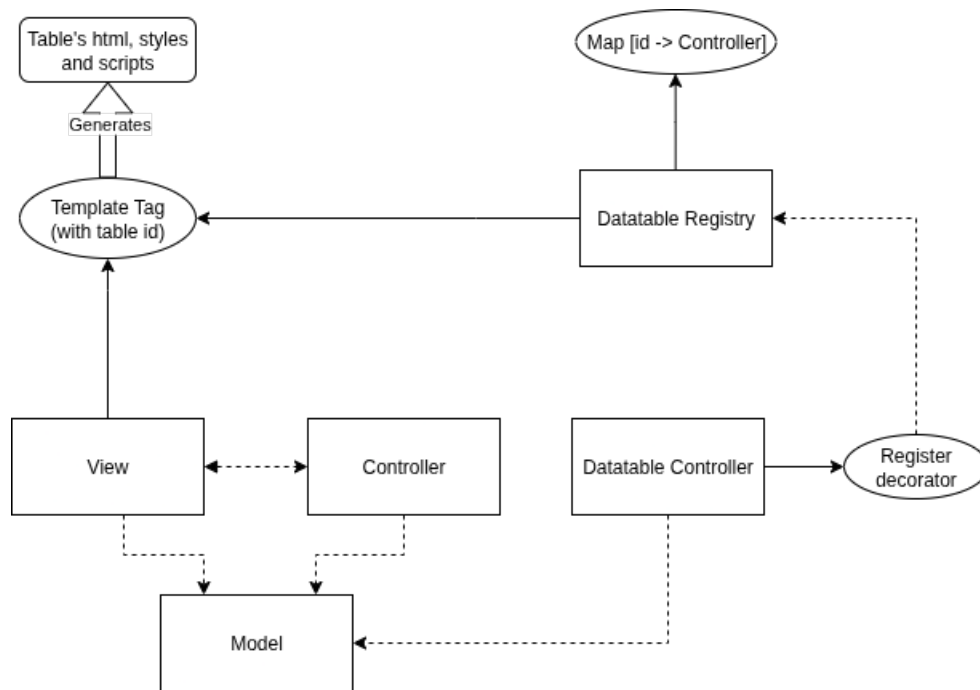


Fig. 4. Datatable builder architecture diagram

With this approach, the architecture allows the rendering of multiple tables on the same view. The developer just needs to call the template tag multiple times with different Datatable controller ids on the appropriate locations inside the view.

## 6.2 Alternative

It is also possible to implement an architecture without using the Datatable registry. Instead of using a registry and registering the controllers to it, we could initialize Datatable controller inside the MVC controller and create context variables to pass the table information to the view so it can be used by the template tag. The downside of this simpler approach is that it is transferring the responsibility of initiating the Datatable controller and handling its data to the developer when a table needs to be created. Besides that, if there are multiple tables in the same view the developer would also need to structure the data in a way that avoid conflicts between multiple tables within the same view.

Since one of the goals of this library is to avoid code duplication and boilerplate code on the developer's side, this alternative implementation is less suitable for this project, but it might be a better alternative if the framework used has limitations that prevent the creation of the registry or make its implementation unfeasible.

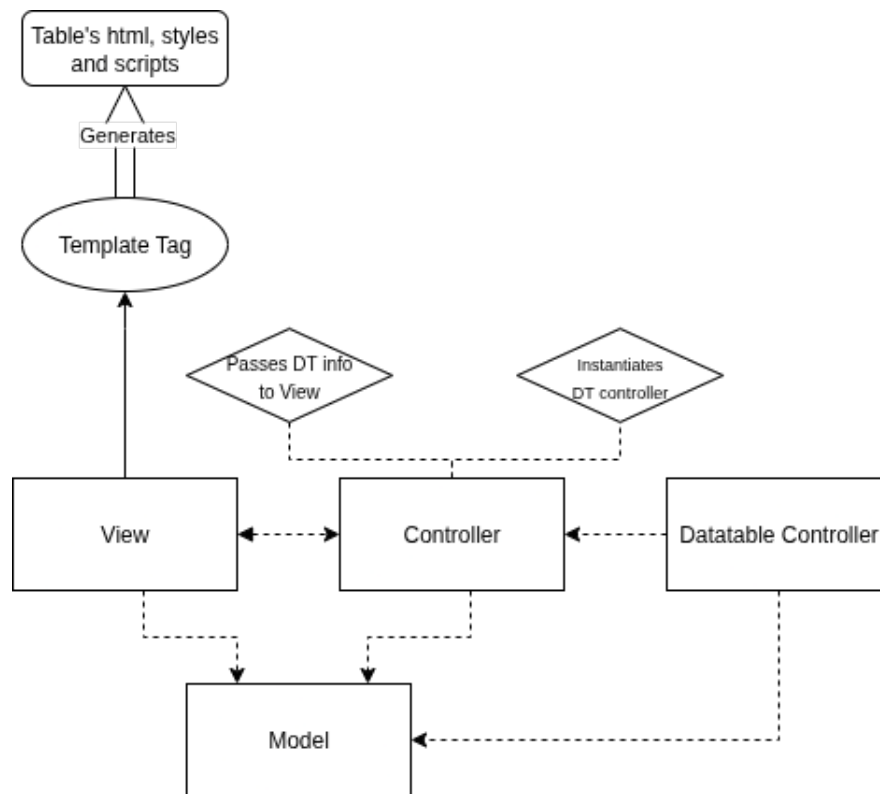


Fig. 5. Alternate architecture diagram

## 7 Django Datatable Builder - Server side

The main component of the library is a Django class-based view called `DatatableBuilderBase`. This is the Datatable controller from our architecture. It has the following roles:

- (1) Handle DataTables requests to the server.
- (2) Store the information necessary to render the datatable's components correctly and transmit them to the client.
- (3) Provide an intuitive interface for the developer to implement the datatable's components.

### 7.1 Handling requests

DataTables will send a request to the server on two occasions: when the table is first rendered and when the content displayed in the table should change, for example, when the user goes to the first page, filters, or sorts the data. When a table needs to be rendered for the first time or updated, DataTables sends an HTTP request to the specified URL specifying the state of the table. DataTables updates a table whenever the user changes pages, applies a filter, performs a search, changes the sort order, or makes any operations that change what the table should present. This request can be either a GET or a POST, depending on what the user specifies.

The most significant parameters in the request are:

- **draw**: The draw parameter ensures that DataTables renders the table in the correct order, even if the responses do not arrive in the same order.
- **start**: The start parameter calculates the offset of the data to be returned. For example, if the user is on the second page and the length of the table is 10 records, the start parameter will be 10.
- **length**: The length parameter calculates the number of records to be returned. For example, if the user is on the second page and the length of the table is 10 records, the length parameter will be 10.
- **order[i][name]**: Sort the table by a specific column. For example, if the user sorts the table by the column "Name", the order parameter will be "Name".
- **order[i][column]**: The same as `order[i][name]`, but it has the column index instead of the column name.
- **order[i][dir]**: Specifies if the table should be sorted in ascending or descending order. For example, if the user sorts the table in descending order, the dir parameter will be "desc".
- **search[value]**: The search parameter searches for a specific value in the table. For example, if the user searches for the value "John", the search parameter will be "John".
- **columns[i][search][value]**: Search value for a specific column. This project uses it to implement filters.

When the server receives the request, it needs to build the table's components, generate the initial queryset, and process the data in the following order:

- (1) Count the number of records in the queryset
- (2) Apply the filters defined by the user depending on the column search parameters
- (3) Filter the queryset based on the search parameter
- (4) Count the number of records in the filtered queryset
- (5) Order the queryset
- (6) Select the records from the queryset to display in the current page (pagination)
- (7) Transform the queryset data into a list of dictionaries that can be read by DataTables to display the records

The order of the steps is important for database performance because Django evaluates the queryset 3 times:

- (1) To count the number of records
- (2) To count the number of filtered records
- (3) To select the subset of records to be displayed in the table

Therefore, we should avoid ordering the queryset before the second evaluation, because it would add a potentially expensive order operation that is not necessary for counting the records on the first two evaluations. Additionally, by ordering the queryset as late as possible, the database will only have to order the filtered records which is faster than ordering all of the initial records.

```

1      def handle_datatables_request(self, params):
2          # Defined by the user to build the table components (columns, filters, actions, etc)
3          self.build_datatable()
4
5          try:
6              draw_parsed = int(params.get('draw'))
7          except ValueError:
8              return JsonResponse({"error": "Bad_draw_parameter"}, status=400)
9
10         # Initial queryset defined by the user
11         qs = self.get_base_queryset()
12
13         # Count the total number of records
14         records_total = qs.count()
15
16         # Apply filters defined by the user
17         for function in self.filter_functions:
18             qs = function(qs)
19
20         # Filter by the search value
21         search = params.get('search[value]', None)
22         if search:
23             qs = self.generic_filter(qs, search)
24
25         # Count the number of records after filtering
26         records_filtered = qs.count()
27
28         qs = self.order_qs(qs, params)
29
30         # Select the subset of records to be displayed
31         qs = self.paginate_qs(qs, params)
32
33         response = {
34             'draw': draw_parsed,
35             'recordsTotal': records_total,
36             'recordsFiltered': records_filtered,
37             'data': self.process_qs(qs)
38         }
39         return JsonResponse(response)

```

Listing 11. Method responsible for handling DataTables requests

## 7.2 Datatable components

This integration provides three components for the user to create a table: columns, filters and actions.

**7.2.1 Column.** The main component of the table. Through the column component, the user can define everything about the table's columns, the main options are:

Table 2. Columns options

Option	Description	Required	Default
field_name	Name of the field (preferably the name of the model's field)	Yes	-
title	Name of the column's header	Yes	-
orderable	Whether the column can be ordered	No	True
init_order	Initial ordering of the column	No	None
formatter	Function to format the value of the column	No	Value of the model's field defined on field_name
escape_html	Escape the column's html to avoid XSS attacks	No	True

All of the options are used to create the columns, however, the required options need to be explicitly defined by the user. The non-required options will be used with their default values.

**7.2.2 Filter.** By default, the only type of filter available is a dropdown filter. However, the developer can create custom filters that suit their needs.





The dropdown filter allows the developer to define a list of values that users can select and a function that will filter the queryset based on the chosen value.

Table 3. Dropdown filter options

Option	Description	Required	Default
column_name	Field name of the column used for the filter	Yes	-
options	List of values that users can select	Yes	-
filter_function	Function used to filter the queryset	Yes	-

In addition, the developer can also define a custom template for the dropdown filter at the table level, so that all of the filters can have the same style as the rest of the application.




**7.2.3 Action.** An action component allows the developer to define actions that users can perform on a table entry. The developer can add an action column where the system displays all actions with icons.

Full Name	E-mail	Field	Salary	Status	Classes	
Benjamin Chang	benchang@greendale.com	Spanish	\$ 1000.00	Terminated	<a href="#">See classes</a>	 
Eustice Whitman	whitman@greendale.com	Sociology	\$ 1500.00	Active	<a href="#">See classes</a>	 
Michelle Slater	slater@greendale.com	Statistics	\$ 1500.00	Active	<a href="#">See classes</a>	 

Showing 1 to 3 of 3 entries

Fig. 6. Example of an action column

The developer can also decide if the action should be a mass action. In this case, users can select table entries and the action can be performed across all of the selected entries.

Students list							
<div> <div>Actions</div> <div> <div>Actions</div> <div>Transfer student</div> </div> </div>		page		Search: <input type="text"/>			
<input type="checkbox"/>	Full Name	Grade	Status	Enrollment Date	Birth Date	Transferred	
<input checked="" type="checkbox"/>	Jeffrey Winger	4	enrolled	2025-02-17	2014-11-20	False	
<input checked="" type="checkbox"/>	Pierce Hawthorne	-	egressed	2025-02-17	2013-08-10	False	
<input checked="" type="checkbox"/>	Troy Barnes	-	egressed	2025-02-17	2009-12-04	False	
<input type="checkbox"/>	Annie Eddison	8	enrolled	2025-02-17	2010-12-19	True	
<input type="checkbox"/>	Britta Perry	8	enrolled	2025-04-11	2010-10-19	True	

Showing 1 to 5 of 5 entries

Fig. 7. Example of a mass action

This library provides a `DatatableActionView` class that the developer can inherit to specify what the action should do. This view handles the retrieval of the records from the database, and possible errors that might occur. When the developer creates an action class, the only necessary thing to do is to override the method `modify_selected_item` and describe what should be done to all the entries selected for this action.

```

1      class TransferStudentView(DatatableActionView):
2          model = Student
3
4          def modify_selected_item(self, request, item):
5              if item.transfer_student:
6                  item.transfer_student = True
7                  item.save()

```

Listing 12. Example of an action class

Table 4. Actions options

Option	Description	Required	Default
action_id	Unique identifier of the action	Yes	-
name	Name of the action	Yes	-
icon_class	Icon that the action should have on the table	Yes	-
url	Url to which the action request should be sent	Yes	-
condition	Function that checks if the action should be displayed on each entry	No	None
mass_action	Allows the action to be executed on several entries at once	No	False
confirm	Shows a confirmation dialog before executing the action	No	True
confirm_mass_action	Shows a confirmation dialog before executing the mass action	No	True

Besides the options described in Table 4, the developer can also define at the table level, which templates to use for the mass action dropdown and for the confirmation dialogs to make sure it has the same style as the rest of the application.

### 7.3 Table state

The state attribute in the DatatableBuilder can specify which parts of the table state to monitor and populate the url parameters of the page. This can share or save urls of a with the table in a specific state. For example, a user could share a link of a table page with a specific filter applied, so when the link is opened, the table will automatically apply the filters.

The system can save the following table information url parameters:

- Page
- Records per page
- Initial ordering
- Filter values

## 8 Django Datatable Builder - Client side

The main component of the client-side integration of the library is a JavaScript method called `createTable` that is responsible for:

- (1) Initializing the `DataTables` instance with the appropriate options.
- (2) Adding table event listeners
- (3) Adding action event listeners
- (4) Adding filters event listeners

### 8.1 Initializing `DataTables` instance

To initialize the `DataTables` instance we need to perform the following steps:

- (1) Retrieve the table data from the template [Listing 13]
- (2) Read the url parameters to load the table state [Listing 14]
- (3) Sanitize the `page` and `page_length` parameters if the state is enabled [Listing 15]
- (4) Calculate the `displayStart` `DataTables` option from the page parameters [Listing 16]
- (5) Change the `page_length` url parameter to the sanitized parameter if the state is enabled [Listing 17]
- (6) Sanitize the filter options and generate the `searchCols` option to define the initial filtering of the table [Listing 18]
- (7) Parse the order url parameters to create the order `DataTables` options [Listing 19]
- (8) If there are no order parameters or the order state is disabled, create the order option based on the columns initial ordering defined on the server [Listing 19]
- (9) Create the `DataTable` options object and merge it with the custom options that may be provided by the user [Listing 20]



```
1 const dtData = getVariableFromDjango("dt_data");
```

Listing 13. DataTable initialization - Retrieve table data from the template

```
1 const urlParams = new URLSearchParams(params)
```

Listing 14. DataTable initialization - Read the url parameters

```
1 function parsePageLength(value, dtData) {
2   value = parseInt(value) || 1
3   const outOfBounds = value < dtData.min_page_length || value > dtData.max_page_length
4   const invalidChoice = dtData.page_length_strict &&
5     !dtData.page_length_choices.includes(value)
6   if (outOfBounds || invalidChoice) {
7     value = dtData.default_page_length
8   }
9   return value
10 }
11
12 function parsePage(page, dtData) {
13   page = parseInt(page);
14   if (isNaN(page) || page < 1 || page > dtData.totalPages) {
15     page = 1;
16   }
17   return page;
18 }
19
20 let pageParsed = 1
21 let pageLengthParsed = dtData.default_page_length
22
23 if (dtData.states.page) {
24   const pageParam = urlParams.get('page')
25   pageParsed = parsePage(pageParam, dtData)
26 }
27 if (dtData.states.page_length) {
28   const pageLengthParam = urlParams.get('page_length')
29   pageLengthParsed = parsePageLength(pageLengthParam, dtData)
30 }
```

Listing 15. DataTable initialization - Sanitize page parameters

```
1 const displayStart = (pageParsed - 1) * pageLengthParsed
```

Listing 16. Calculate DataTables displayStart option

```
1 if (dtData.states.page_length) {
2   changeUrl('page_length', pageLengthParsed)
3 }
```

Listing 17. Update page length url parameter

```

1      let searchCols = dtData.columns.map(column => {
2          const filter = dtData.filters.find(filter => filter.column_name === column.field_name)
3          if (!filter || !dtData.states.filter) {
4              return null
5          }
6          const filterUrlParam = urlParams.get(filter.id)
7          if (filterUrlParam) {
8              changeUrl(filter.id, filterUrlParam)
9              updateFilterElement(filter, filterUrlParam)
10             return {
11                 'search': filterUrlParam,
12             }
13         }
14         const selectedOption = filter.options.find(option => option.selected)
15         const selectedOptionValue = selectedOption ? selectedOption.value : null
16         if (selectedOption) {
17             changeUrl(filter.id, selectedOptionValue)
18             updateFilterElement(filter, selectedOptionValue)
19         }
20         return selectedOption
21     })

```

Listing 18. Generate DataTables searchCols option to define initial filters

```

1      let orderList = []
2      if (dtData.states.order) {
3          urlParams.forEach((value, key) => {
4              // The parameters for order have the format o_{index}={asc/desc}
5              if (key.startsWith('o_')) {
6                  const orderIndex = parseInt(key.split('_')[1])
7                  const column = columnsList[orderIndex]
8                  if (column && column.orderable && ['asc', 'desc'].includes(value)) {
9                      orderList.push([orderIndex, value])
10                 }
11             } else {
12                 // Remove invalid order parameter
13                 removeFromUrl(key);
14             }
15         })
16     }
17 }
18
19 if (!orderList.length) {
20     // Get the order of the columns from the server
21     orderList = dtData.order;
22     for (let [pos, dir] of orderList) {
23         changeUrl(`o_${pos}`, dir);
24     }
25 }

```

Listing 19. Generate DataTables order option

```

1      const datatableDefaultOptions = {
2          displayStart: displayStart,
3          pageLength: pageLengthParsed,
4          lengthMenu: dtData.page_length_choices,
5          serverSide: true,
6          ajax: {
7              url: dtData.url,
8              type: 'GET',
9          },
10         columns: columnsList,
11         order: orderList,
12         searchCols: searchCols,
13     }
14
15     const table = tableElement.DataTable({
16         ...datatableDefaultOptions,
17         ...dtConfig.datatableOptions
18     });

```

Listing 20. Merge default options with user options and instantiate DataTables object

## 8.2 Table events

The DataTables library offers various events that we can listen to perform specific actions based on what happens to the table. In this library we mainly use the `init` event to properly set the page url parameter, the `page` event (fired when the table's page is changed) to update the page url parameter when the user changes the table's page, and the `length` event (fired when the user changes the number of records per page) to update the `page_length` and page url parameters accordingly. The library also allows the developer to add custom functions inside the `init`, `draw`, and `preDraw` events. The `init` event is fired when the table is rendered for the first time, the `preDraw` event is fired right before a table update happens, and the `draw` event is fired right after a table update happens.

```

1      table.on('init', function(e, settings, json) {
2          // custom init function
3          dtConfig.onInit(e, settings, json, dtData)
4          if (!dtData.states.page) return; // checks if page state is enabled
5
6          const pageInfo = getPageInfo(table);
7          const { pages } = pageInfo;
8          // Goes to the first page if the page sent is invalid
9          if (dtData.page > pages) {
10             table.page(0).draw('page')
11             changeUrl('page', 1);
12         }
13         else {
14             changeUrl('page', pageInfo.page1);
15         }
16     })
17
18     table.on('page', function(e, settings) {
19         // Updates page url parameter if page state is enabled
20         if (!dtData.states.page) return;
21         const pageInfo = getPageInfo(table)
22         changeUrl('page', pageInfo.page1);
23     })
24
25     table.on('length', function(e, settings, len) {
26         const pageInfo = getPageInfo(table)
27
28         // Updates page url parameter if page state is enabled
29         if (dtData.states.page) {
30             changeUrl('page', pageInfo.page1);
31         }
32
33         // Updates page length url parameter if page_length state is enabled
34         if (dtData.states.page_length) {
35             changeUrl('page_length', len);
36         }
37     })
38
39     table.on('preDraw', function(e, settings) {
40         // Custom preDraw function
41         dtConfig.onPreDraw(e, settings, dtData)
42     })
43
44     table.on('draw', function(e, settings) {
45         // Custom draw function
46         dtConfig.onDraw(e, settings, dtData)
47     })

```

Listing 21. Table event listeners

### 8.3 Actions

The table action is a very dynamic component that performs a lot of changes on the page's interface. When an action is clicked, if the action has a confirm flag, a confirmation modal will appear, otherwise, the request will be directly sent to the server.

When dealing with mass actions, we need to keep track of which checkboxes are active, add an event listener to enable or disable all checkboxes if the header checkbox is enabled or disabled. We also need to create an event listener on every checkbox to determine if the mass action selector element should be enabled or disabled. Finally, we need to create an event listener to send the requested entries to the server when an action is selected on the mass action selector.

Because we need to listen to events on a lot of similar objects (the checkboxes), we will apply the event delegation technique, which creates one event listener that handles several objects by leveraging JavaScript's event bubbling [5].

```

1 // Click event listener function on action icons
2 function actionIconClickEvent(table, actionElement, actions, modal) {
3   const actionItemId = actionElement.getAttribute("data-item-id")
4   const actionId = actionElement.getAttribute("data-id")
5   const actionName = actionElement.getAttribute("data-name")
6   const action = getCurrentAction(actions, actionId)
7   if (action.url) {
8     if (action.confirm) {
9       dtConfig.actionModalOpen(modal, actionName, actionItemId, actionId) // Populates
10        action data to modal and opens it
11     }
12     else {
13       dtConfig.makeActionRequest(table, action, actionItemId, masterCheckbox,
14         massActionButton)
15     }
16   }
17 }
18
19 // Change event listener function on header checkbox
20 function massActionCheckboxChangeEvent(checkboxElement, actionCheckboxes){
21   if (checkboxElement.checked) {
22     actionCheckboxes.forEach(actionNode => actionNode.checked = true)
23   }
24   else {
25     actionCheckboxes.forEach(actionNode => actionNode.checked = false)
26   }
27 }
28
29 // Change event listener function on mass action selector
30 function massActionDropdownChangeEvent(table, action, actionCheckboxes, massActionButton)
31 {
32   const firstOption = massActionButton.options[0]
33   const firstOptionText = firstOption.innerText
34   firstOption.innerText = "Loading..."
35   massActionButton.selectedIndex = 0
36   const massActionString = getMassActionString(actionCheckboxes)
37   dtConfig.makeActionRequest(table, action, massActionString, masterCheckbox,
38     massActionButton).finally(() =>
39     firstOption.innerText = firstOptionText
40   )

```

```

37     }
38
39
40     // Make request when user clicks on confirm button of the confirmation modal
41     confirmButton.addEventListener('click', function (event) {
42         const actionItemId = modal.getAttribute('data-action-item-id')
43         const actionId = modal.getAttribute("data-action-id")
44         const action = getCurrentAction(actions, actionId)
45         dtConfig.makeActionRequest(table, action, actionItemId, masterCheckbox,
46             massActionButton).finally(() => dtConfig.closeModal(modal))
47     })
48
49     // Close modal if user cancels confirmation modal
50     cancelButton.addEventListener('click', function (event) {
51         dtConfig.closeModal(modal)
52     })
53
54     tableNode.addEventListener('click', function (event) {
55         // Only call function if element clicked is an action icon
56         if (event.target.classList.contains('action-icon')) {
57             actionIconClickEvent(table, event.target, actions, modal)
58         }
59     })
60
61     dtContainer.addEventListener('change', function(event) {
62         const actionCheckboxes = tableNode.querySelectorAll('.mass-action-input-container_>_
63             input')
64         // Only call function if element changed is the header checkbox
65         if
66             (event.target.parentElement.classList.contains('mass-action-header-input-container'))
67             {
68                 massActionCheckboxChangeEvent(event.target, actionCheckboxes)
69                 // Enable or disable mass action selector depending on header button state
70                 if (event.target.checked) {
71                     massActionButton.disabled = false
72                 }
73                 else {
74                     massActionButton.disabled = true
75                 }
76             }
77
78         // Only call function if element changed is the row checkbox
79         if (event.target.parentElement.classList.contains('mass-action-input-container')) {
80             //Checks/Unchecks header checkbox and enable/disables mass action based on the
81             checkboxes status
82             masterCheckbox.checked = allChecked(actionCheckboxes)
83             massActionButton.disabled = !someChecked(actionCheckboxes)
84         }
85
86         // Only call function if element changed is the the mass action selector
87         if (event.target.getAttribute('name') === "mass_action_dropdown") {
88             const option = event.target.options[event.target.selectedIndex]
89             const actionId = option.value
90             if (!actionId) {
91                 return
92             }

```

```

88     const action = getCurrentAction(actions, actionId)
89     massActionDropdownChangeEvent(table, action, actionCheckboxes, massActionButton)
90   }
91 })

```

Listing 22. Actions event listeners

## 8.4 Filters

The filters are designed to be an extensible component that the developer can create filters that suit their needs. Currently there is only one default filter, the dropdown filter, where the user can select a set of predefined options and filter the table based on the chosen option. As mentioned in Table 1, the filters can be defined through the global configuration option.

There are 2 filter entries on the configuration objects, the `defaultFilters`, which defines the filters types provided by the library, and the `customFilters`, which is supposed to be used by the developer to create new filter types. Both entries have the same structure, and the developer can use the same approach to create custom filters that the library does to create the default filters.

The filter structure consists in a key that will be the name of the filter type and a value, the object defining how the filter works.

These are the default options for a filter:

```

1     const filterDefaults = {
2       // Creates the event listener function that will trigger when the filter is applied
3       listenerFunction: function(listenerEvent, filter, table) {
4         const filterElement = this.getFilterElement(filter, table);
5         filterElement.addEventListener(listenerEvent, () => {
6           if (this.validationFunction(filter, table)) {
7             const filterValue = this.getFilterValue(filter, table)
8             this.filterFunction(filterValue, filter, table);
9             if (saveFilterState) {
10               changeUrl(filter.id, filterValue);
11             }
12             this.onActiveChange(this.isActive())
13           }
14         })
15       },
16
17       // Function called when registering filters (Only function that is called outside of
18       // the filter configuration object)
19       onChange: function(filter, table) {
20         this.listenerFunction('change', filter, table);
21       },
22
23       // Retrieves the filter value selected (Needs to be specified on each filter type)
24       getSelectedValue: function(filter, table) {
25         return false
26       },
27
28       // Function that performs the filter request through DataTables api
29       filterFunction: function(value, filter, table) {
30         changeUrl(filter.id, value)
31         changeUrl('page', 1)

```

```

31         table.column(`${filter.column_name}:name`).search(value).draw()
32     },
33
34     // Validate if the filter value is valid
35     validationFunction: function(filter, table) {
36         return true
37     },
38
39     // Retrieve the filter value to be applied
40     getFilterValue: function(filter, table) {
41         return this.getFilterElement(filter, table).value
42     },
43
44     // Check if the filter is being applied (Recommended to be specified on each filter
45     // type)
46     isActive: function(filter, table) {
47         return false;
48     },
49
50     // Triggered when a trigger starts or stops being applied
51     onActiveChange: function(filter, isActive) {
52         const filterElement = this.getFilterElement(filter, table);
53         if (isActive) {
54             filterElement.classList.add('active')
55         }
56         else {
57             filterElement.classList.remove('active')
58         }
59     },
60
61     // Helper function to retrieve the filter html element
62     getFilterElement: function(filter, table) {
63         return document.getElementById(filter.id)
64     },
65 },

```

Listing 23. Default values for filter configuration

As indicated by Listing 23, the only field that needs to be defined when creating a custom filter is `getSelectedValue`, although developers should define the `validationFunction` to make sure the server is always receiving expected values. Listing [reflst:dropdown-filter-example-client](#) shows how the default dropdown is defined. The user can define a custom filter in the same manner using the `customFilters` attribute instead of `defaultFilters`, to avoid overriding the built in filters.



```

1      window[jsConfigName].defaultFilters = {
2          dropdown: {
3              validationFunction: function(filter, table) {
4                  const possibleValues = filter.options.map(option => option.value)
5                  return possibleValues.includes(this.getFilterValue(filter, table));
6              },
7
8              isActive: function(filter, isActive) {
9                  const default_option = filter.options.find(option => option.default)
10                 return default_option.value !== this.getFilterValue(filter, table);
11             },
12
13             getSelectedValue: function(filter, table) {
14                 const selected_option = filter.options.find(option => option.selected)
15                 return selected_option.value;
16             },
17         }
18     }

```

Listing 24. Client side implementation of dropdown filter

The developer would also need to define an `add_<custom>_filter` method to the `DatatableBuilder` to make sure that the dropdown can be created using the default builder pattern. There is a helper method `add_filter` that the developer can use on their custom method to automatically handle the internal logic of passing the correct data to the client. The helper method abstracts the filter creation complexity, ensuring that the developer only needs to worry about the specific features of the custom filter. Listing 25 shows how the `add_dropdown_filter` method is defined.

```

1      def add_dropdown_filter(self, column_name, options, filter_function, template=None):
2          if template is None:
3              template = self.default_filter_template
4
5          selected = False
6          for option in options:
7              # Ensures that the option items have the structure {'name': 'option_name', 'value':
              # 'option_value', 'selected': True/False}
8              # The selected key is optional and will be considered false if not specified
9              expected_key_count = 2
10             if not isinstance(option, dict):
11                 raise TypeError('Dropdown options must be dictionaries')
12             if 'name' not in option:
13                 raise KeyError('Dropdown options must have a "name" key')
14             if 'value' not in option:
15                 raise KeyError('Dropdown options must have a "value" key')
16             if 'selected' in option:
17                 expected_key_count += 1
18                 selected = True
19             if 'default' in option:
20                 expected_key_count += 1
21
22             if len(option) != expected_key_count:
23                 raise ValueError(f'Options can only have the keys "name", "value", and "selected"')
24
25             if not selected:
26                 options[0]['selected'] = True
27
28             context = {'options': options}
29
30             # Specifies the filter name, the template that will be used and the necessary variables
31             # to be sent to the client through the context
32             self.add_filter(column_name, 'dropdown', filter_function, template, context)

```

Listing 25. Server side implementation of dropdown filter

## 9 Django Datatable Builder - Summary

The Django Datatable Builder library creates several abstractions both on the server and client that greatly reduce the DataTables integration complexity while still providing a high degree of flexibility. Without these abstractions, these steps would have to be done manually for each dynamic table created with DataTables. This significantly impacts developer productivity, as developers spend more time writing and debugging repetitive code. In addition, if the developer needs to spend less time implementing dynamic tables, this time can be allocated to creating more meaningful tables for the application.

It also creates additional features and components that are used frequently on dynamic tables but can be challenging to implement manually. With these features, the developer can create more complex and useful tables with elements designed to facilitate data visualization and manipulation.

The table presents a summary of the abstracted implementation steps and features provided by the library.

Table 5. Summary of abstractions and features implemented by Django Datatable Builder

Abstractions	Features
Initialization of DataTables instace	Interface to define custom DataTables options
Centralized table html	Interface to define table event listeners
Centralized table components html	Storage of table state on url parameters
Request validation	Custom actions
Parsing of filter parameters	Mass actions
Parsing of pagination parameters	Conditional actions
Parsing of ordering parameters	Standardized filter creation
Parsing of search parameter	Flexible and safe column rendering
Queryset slicing based on pagination parameters	Filter extensibility
Queryset filtering based on filter and search parameters	
Queryset ordering based on ordering parameters	
Server response formatting	

## 10 Demonstration: Greendale College Management System

To demonstrate the practical application and effectiveness of Django Datatable Builder, we developed a comprehensive college management system called "Greendale College Management System." This case study showcases how the library can be applied to real-world scenarios and its capabilities across different data models and use cases typical in educational administration.

### 10.1 Technology Stack

The demonstration application utilizes a modern technology stack that reflects current industry practices:

#### Backend Technologies:

- Django 5.1.6 as the primary web framework
- SQLite database for data persistence (suitable for demonstration purposes)
- Django Datatable Builder library for dynamic table functionality
- Python 3.10.x runtime environment

#### Frontend Technologies:

- CoreUI 5.3.0 Bootstrap admin theme for consistent UI design
- SASS preprocessing for maintainable stylesheets
- DataTables.js 2.2.2 for client-side table interactions
- jQuery 3.7.1 for DOM manipulation and AJAX requests

The application demonstrates how Django Datatable Builder integrates with modern admin themes and maintains compatibility with existing JavaScript ecosystems while providing a Pythonic development experience.

### 10.2 Data Models and Domain

The Greendale College Management System is built around three core Django models that represent the essential entities in college administration. These models demonstrate how Django Datatable Builder handles various data types, relationships, and business logic patterns commonly found in real-world applications.

**10.2.1 Student Model.** The Student model represents enrolled students with demographic and academic information:

```

1  class Student(models.Model):
2      SITUATION_CHOICES = (
3          ('enrolled', 'Enrolled'),
4          ('egressed', 'Graduated'),
5          ('expelled', 'Expelled'),
6      )
7
8      full_name = models.CharField(max_length=255)
9      date_of_birth = models.DateField()
10     grade = models.PositiveSmallIntegerField(null=True, blank=True)
11     situation = models.CharField(max_length=20, choices=SITUATION_CHOICES)
12     enroll_date = models.DateField()
13     transfer_student = models.BooleanField(default=False)
14     phone = models.CharField(max_length=20, null=True, blank=True)
15     email = models.EmailField(null=True, blank=True)
16     address = models.TextField(null=True, blank=True)
17     classes = models.ManyToManyField('Class', related_name='students', blank=True)
18
19     created_at = models.DateField(auto_now_add=True)
20     updated_at = models.DateField(auto_now_add=True)

```

Listing 26. Student Model Definition

**10.2.2 Teacher Model.** The Teacher model manages faculty information including employment status and salary:

```

1  class Teacher(models.Model):
2      STATUS_CHOICES = (
3          ('active', 'Active'),
4          ('inactive', 'Inactive'),
5          ('terminated', 'Terminated'),
6      )
7
8      full_name = models.CharField(max_length=255)
9      date_of_birth = models.DateField()
10     hire_date = models.DateField()
11     phone = models.CharField(max_length=20)
12     email = models.EmailField()
13     subject_specialty = models.CharField(max_length=100)
14     employee_id = models.CharField(max_length=50, unique=True)
15     salary = models.DecimalField(max_digits=10, decimal_places=2)
16     status = models.CharField(max_length=20, choices=STATUS_CHOICES, default='active')
17
18     created_at = models.DateTimeField(auto_now_add=True)
19     updated_at = models.DateTimeField(auto_now=True)

```

Listing 27. Teacher Model Definition

**10.2.3 Class Model.** The Class model represents academic courses with capacity management and which teacher lectures the class:

```

1 class Class(models.Model):
2     name = models.CharField(max_length=100)
3     academic_year = models.PositiveSmallIntegerField()
4     teacher = models.ForeignKey(Teacher, on_delete=models.CASCADE, related_name='classes')
5     subject = models.CharField(max_length=100)
6     capacity = models.PositiveSmallIntegerField()
7
8     created_at = models.DateTimeField(auto_now_add=True)
9     updated_at = models.DateTimeField(auto_now=True)

```

Listing 28. Class Model Definition

### 10.3 Application Screens and Datatable Features

The Greendale College Management System consists of three main administrative screens, each showcasing different aspects of Django Datatable Builder's functionality. Each screen demonstrates specific features including filtering, actions, custom formatting, and user interface integration patterns.

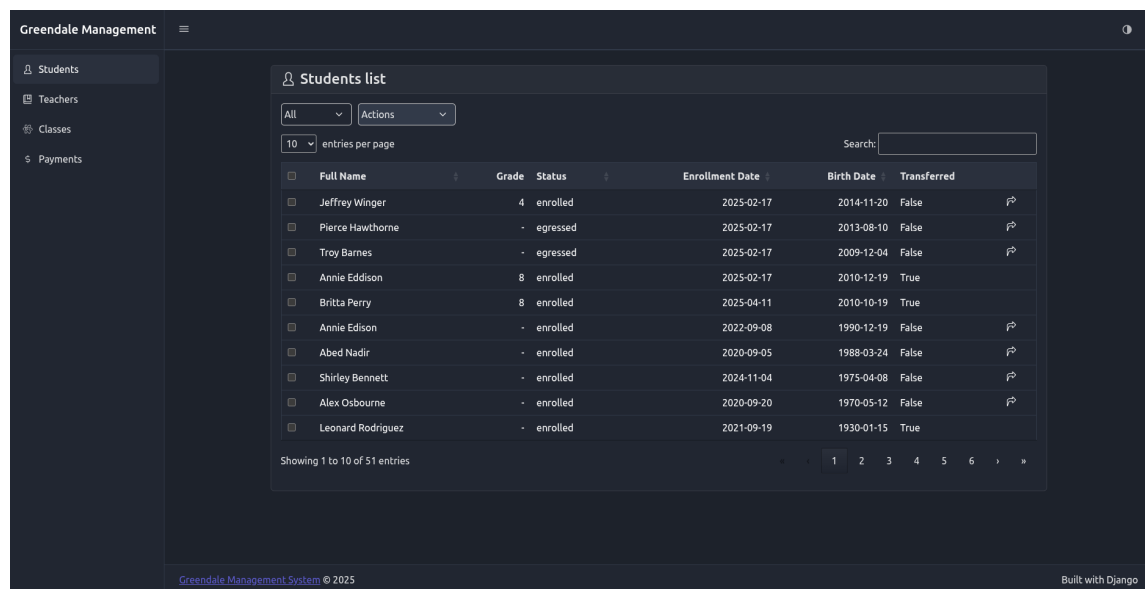


Fig. 8. Students management screen

**10.3.1 Students Management Screen.** The Students Management screen serves as the primary interface for managing student records and demonstrates several key features of Django Datatable Builder:

- **Mass Selection:** Checkbox column enabling bulk operations on multiple student records
- **Conditional Actions:** Transfer Student action appears only for non-transfer students
- **Dropdown Filtering:** Status-based filtering with options for All, Graduated, and Enrolled students

- **Custom Formatting:** The grade field displays as dash (-) when null, demonstrating conditional display logic
- **Global Search:** Full-text search across student names and key fields
- **Date Handling:** Proper formatting and display of enrollment and birth dates

### Implementation Details:

The StudentDatatable class demonstrates the builder pattern with method chaining for table construction:

```

1      @register_table
2      class StudentDatatable(CustomDatatableBuilder):
3          id = "students-table"
4          model = Student
5          export_enabled = False
6          url_name = 'datatable_students'
7          search_fields = ['full_name', 'situation', 'enroll_date', 'grade', 'transfer_student']
8
9          def build_table(self):
10             self.add_mass_action_column()
11             self.add_column(field_name='full_name', title='Full_Name')
12             self.add_column(field_name='grade', title='Grade',
13                             formatter=lambda item: item.grade or '-', orderable=False)
14             self.add_column('situation', 'Status', orderable=True,
15                             formatter=lambda item: item.situation)
16             self.add_column(field_name='enroll_date', title='Enrollment_Date')
17             self.add_column(field_name='date_of_birth', title='Birth_Date')
18             self.add_column('transfer_student', 'Transferred', orderable=False)
19
20             self.add_action('transfer_student', 'Transfer_student', 'cil-share',
21                             url=reverse('edit_student'),
22                             condition=lambda item: not item.transfer_student,
23                             mass_action=True)
24             self.add_actions_column()
25
26             options = [
27                 {'name': "All", "value": "-1", "default": True},
28                 {'name': "Graduated", "value": "egressed"},
29                 {'name': "Enrolled", "value": "enrolled"},
30             ]
31             self.add_dropdown_filter('situation', options, self.filter_situation)
32
33         @staticmethod
34         def filter_situation(qs, value):
35             if value and value != "-1":
36                 qs = qs.filter(situation=value)
37             return qs

```

Listing 29. Student Datatable Implementation

Django's class-based views implement action handling, providing clean separation of concerns:

```

1  class EditActionView(DataTableActionView):
2      model = Student
3
4      def modify_selected_item(self, request, item):
5          if item.transfer_student:
6              item.transfer_student = True
7              item.save()

```

Listing 30. Student Action View Implementation

Full Name	E-mail	Field	Salary	Status	Classes
Ben Chang	ben.chang@greendale.edu	Spanish	\$ 58035.00	Active	<a href="#">See classes</a>
Benjamin Chang	benchang@greendale.com	Spanish	\$ 1000.00	Terminated	<a href="#">See classes</a>
Buzz Hickey	buzz.hickey@greendale.edu	Criminology	\$ 37281.00	Active	<a href="#">See classes</a>
Carl Bladt	carl.bladt@greendale.edu	Business	\$ 59332.00	Active	<a href="#">See classes</a>
Cory Radison	cory.radison@greendale.edu	Music	\$ 54688.00	Active	<a href="#">See classes</a>
Dr. Escodera	dr.escodera@greendale.edu	Medicine	\$ 48546.00	Active	<a href="#">See classes</a>
Eustice Whitman	whitman@greendale.com	Sociology	\$ 1500.00	Active	<a href="#">See classes</a>
Herbert Bogner	herbert.bogner@greendale.edu	English	\$ 59157.00	Active	<a href="#">See classes</a>
Ian Duncan	ian.duncan@greendale.edu	Psychology	\$ 57123.00	Inactive	<a href="#">See classes</a>
Jason Chapman	jason.chapman@greendale.edu	Physical Education	\$ 48417.00	Active	<a href="#">See classes</a>

Fig. 9. Teachers management screen

### 10.3.2 Teachers Management Screen

#### Features Demonstrated:

- **Conditional Actions:** Three distinct actions (Fire, Suspend, Hire) that appear based on current teacher employment status
- **Custom HTML Rendering:** Underlined headers and linked navigation elements
- **Relational Data Links:** Dynamic links to related class information filtered by teacher

### Implementation Details:

```

1      @register_table
2      class TeacherDatatable(CustomDatatableBuilder):
3          id = "teachers-table"
4          model = Teacher
5          url_name = "datatable-teachers"
6
7          @staticmethod
8          def classes_html(item):
9              return f"""
10             <a href="{reverse('classes') + "?teacher_id=" + str(item.id)}">See_classes</a>
11             """
12
13          @staticmethod
14          def filter_status(qs, value):
15              if value in {'active', 'inactive', 'terminated'}:
16                  qs = qs.filter(status=value)
17              return qs
18
19          def build_table(self):
20              self.add_column('full_name', '<u>Full_Name</u>',
21                             formatter=lambda item: item.full_name,
22                             init_order=[1, 'asc'], escape_header_html=False)
23              self.add_column('email', 'E-mail', formatter=lambda item: item.email)
24              self.add_column('subject_specialty', 'Field',
25                             formatter=lambda item: item.subject_specialty)
26              self.add_column('salary', 'Salary', escape_html=False,
27                             formatter=lambda item: self.no_wrap_item(self.format_money(item.salary)))
28              self.add_column('status', 'Status',
29                             formatter=lambda item: item.status.capitalize())
30              self.add_column('classes', "Classes", formatter=self.classes_html,
31                             escape_html=False, orderable=False, include_in_export=False)
32              self.add_actions_column()
33
34              options = [
35                  {'name': "All", "value": "-1", "default": True},
36                  {'name': "Active", "value": "active"},
37                  {'name': "Inactive", "value": "inactive"},
38                  {'name': "Terminated", "value": "terminated"},
39              ]
40              self.add_dropdown_filter('status', options, self.filter_status, default_value='-1')
41
42              self.add_action('fire_teacher', 'Fire_teacher', 'cil-fire',
43                             url=reverse('fire_teacher'), confirm=False,
44                             condition=lambda item: item.status != 'terminated')
45              self.add_action('suspend_teacher', 'Suspend_teacher', 'cil-ban',
46                             url=reverse('suspend_teacher'), confirm=False,
47                             condition=lambda item: item.status != 'inactive')
48              self.add_action('hire_teacher', 'Hire_teacher', 'cil-user-plus',
49                             url=reverse('hire_teacher'), confirm=False,
50                             condition=lambda item: item.status != 'active')

```

Listing 31. Teacher Datatable Implementation



The actions are handled through dedicated view classes:

```

1      class FireTeacherView(DatatableActionView):
2          model = Teacher
3
4          def modify_selected_item(self, request, item):
5              item.status = 'terminated'
6              item.save()
7
8      class HireTeacherView(DatatableActionView):
9          model = Teacher
10
11         def modify_selected_item(self, request, item):
12             item.status = 'active'
13             item.save()
14
15         class SuspendTeacherView(DatatableActionView):
16             model = Teacher
17
18         def modify_selected_item(self, request, item):
19             item.status = 'inactive'
20             item.save()

```

Listing 32. Teacher Action Views Implementation

### 10.3.3 Classes Management Screen

#### . Features Demonstrated:

- **Calculated Fields:** Real-time student enrollment counts calculated from many-to-many relationships
- **Foreign Key Display:** Rendering of related teacher information
- **Foreign Key Filter:** Dynamic filtering of related teacher information.
- **Relational Queries:** Database queries handling foreign key relationships

**Implementation Details:**

```

1      @register_table
2      class ClassDatatable(CustomDatatableBuilder):
3          id = "classes-datable"
4          model = Class
5          url_name = "datatable-classes"
6
7          @staticmethod
8          def students_enrolled_html(item):
9              return Student.objects.filter(classes=item).count()
10
11         @staticmethod
12         def filter_teachers(qs, value):
13             try:
14                 value = int(value)
15             except ValueError:
16                 return qs
17
18             if value == -1:
19                 return qs
20             return qs.filter(teacher_id=value)
21
22         def build_table(self):
23             teachers = Teacher.objects.all()
24             self.add_column('name', 'Class', formatter=lambda item: item.name)
25             self.add_column('teacher', 'Teacher',
26                             formatter=lambda item: item.teacher.full_name if item.teacher else "N/A")
27             self.add_column('capacity', 'Capacity',
28                             formatter=lambda item: f'{item.capacity}_students')
29             self.add_column('students_enrolled', 'Students_enrolled',
30                             formatter=self.students_enrolled_html,
31                             orderable=False, include_in_export=False)
32
33             teacher_options = [{'name': teacher.full_name, 'value': str(teacher.id)}
34                                for teacher in teachers]
35             options = [
36                 {'name': "All", "value": "-1", "default": True},
37                 *teacher_options
38             ]
39
40             self.add_dropdown_filter('teacher', options, self.filter_teachers,
41                                     default_value="-1")

```

Listing 33. Class Datatable Implementation

Class	Teacher	Capacity	Students_enrolled
Sociology	Eustice Whitman	50 students	13
Spanish 101	Benjamin Chang	50 students	15
Statistics	Michelle Slater	40 students	12
Spanish 102	Benjamin Chang	30 students	15
Sociology 101	Eustice Whitman	28 students	12
Statistics Fundamentals	Michelle Slater	23 students	10
Introduction to Law	Jeff Winger	25 students	9
Spanish 101	Ben Chang	29 students	18
Introduction to Psychology	Ian Duncan	28 students	17
Criminology Basics	Buzz Hickey	31 students	10

Fig. 10. Classes management screen

#### 10.4 Technical Implementation Highlights

The Greendale College Management System demonstrates several key technical aspects of Django Datatable Builder that make it particularly suitable for enterprise applications and complex data management scenarios.

**10.4.1 Custom Base Class Architecture.** The system implements a custom base class CustomDatatableBuilder that extends the library's core functionality while maintaining consistency across all tables:

```

1 class CustomDatatableBuilder(DatatableBuilderBase):
2     table_template = "coreui/datatables_builder/datatables.html"
3     default_filter_template = "coreui/datatables_builder/dropdown_filter.html"
4     mass_action_dropdown_template = "coreui/datatables_builder/mass_action_dropdown.html"
5     action_confirmation_modal_template =
6         "coreui/datatables_builder/action_modal_confirmation.html"
7     config_script = "js/datatables_config.mjs"

```

Listing 34. Custom Datatable Builder Base Class

**10.4.2 Security and Data Integrity.** Django Datatable Builder implements several security measures demonstrated throughout the case study:

- **HTML Escaping:** All user data is automatically escaped unless explicitly disabled with `escape_html=False`
- **XSS Protection:** Template rendering includes proper sanitization of user-generated content
- **CSRF Protection:** All action requests utilize Django's CSRF token system
- **SQL Injection Prevention:** ORM-based queries prevent SQL injection attacks
- **Permission Integration:** Table access and action conditions can incorporate Django's built-in permission system

**10.4.3 Template System Integration.** The case study demonstrates sophisticated template integration that is easy to use, maintains separation of concerns and provides flexibility:

```

1      {% extends 'coreui/base.html' %}
2      {% load datatable %}
3      {% load static %}
4
5      {% block content %}
6          <div class="card">
7              <div class="card-header">
8                  <h4 class="card-title mb-0 d-flex align-items-center">
9                      <i class="cil-user me-2"></i>Students list
10                 </h4>
11             </div>
12             <div class="card-body">
13                 {% datatable 'students-table' %}
14             </div>
15             </div>
16         {% endblock %}

```

Listing 35. Template Usage Example

This approach provides:

- **Template Tag Simplicity:** Single template tag for complete datatable rendering
- **Theme Integration:** Seamless integration with existing template hierarchies
- **Asset Management:** Automatic inclusion of required CSS and JavaScript files
- **Context Preservation:** Maintains template context for proper inheritance

## 11 Similar projects

There are other libraries that provide an integration with DataTables for Django. Currently, there are 3 prominent libraries for this purpose:

### 11.1 django-datatable-view [8]

A Django class-based view that provides an integration with jQuery DataTables, offering a drop-in replacement for Django's ListView with server-side processing capabilities. It uses a ModelForm-like declarative approach that makes table creation intuitive for Django developers familiar with existing Django patterns. The project does not seem to be actively maintained (last commit on November 12, 2024 and last issue responded on July 8, 2021 on their GitHub repository).

Table 6. Overview of django-datatable-view

Approach	Features	Limitations
Overrides Django's ListView	Multiple datatables support	Not actively maintained
Syntax similar to ModelForm	Search and sort support	Incomplete documentation
	Bootstrap integration	Hard to customize

11.2 `django-ajax-datatable` [6]

A Django app specifically designed for server-side processing mode with DataTables, providing an `AjaxDatatableView` base class that handles complex frontend-backend interactions. It focuses on minimal configuration while automatically optimizing database queries for improved performance. The project does not seem to be actively maintained (last commit on April 4, 2024 and only other users respond to issues and not maintainers).

Table 7. Overview of `django-ajax-datatable`

Approach	Features	Limitations
Base class handling configurations and requests Inheritance of base class to define tables	Customizable rendering of rows Search and sort support  Pagination support Support for date range searching Extensive documentation	Not actively maintained Only handles the server side of the integration

11.3 `djangoestframework-datatables` [4]

A library that bridges Django REST Framework(DRF) and DataTables by enabling any existing DRF API to work with DataTables by simply adding `?format=datatables` to requests. It provides an integration through custom renderers, filters, and pagination components, requiring minimal configuration while maintaining full API functionality for both regular clients and DataTables. The project had two releases in 2024 (version 0.7.1 on March 6, 2024 and version 0.7.2 on June 14, 2024). Current maintenance activity appears limited with the last commit around June 2024 and most recent issues opened in August 2024.

Table 8. Overview of `djangoestframework-datatables`

Approach	Features	Limitations
Zero-code integration via URL parameter Uses DRF renderers, filters, and pagination	Maintains existing API functionality Search and sort support	Not actively maintained Requires DRF Limited to DRF architecture

## 12 Django Datatable Builder Overview

By understanding the current state and limitations of these libraries, Django Datatable Builder aims to create an integration that mitigates those limitations to provide a complete solution for developers.

Table 9. Overview of Django Datatable Builder

Approach	Features	Limitations
Base class handling configurations and requests	Client and server integration	Early development stage
Inheritance of base class to define tables	Customizable templates	Complex initial configuration
Builder pattern	Modular components	
	Easily extensible	

## 13 Limitations

While this library addresses some limitations from similar projects, it is still in an early development stage and lacks some features that these more mature projects have, for example a caching system for the tables querysets.

This library also does not currently handle concurrency problems that might happen if several users are modifying the same table.

The architecture of this library relies heavily on the MVC pattern and on Django's template tag system. Therefore, it cannot be implemented on frameworks or languages with different UI design patterns and might be challenging to be implemented on frameworks that don't support a robust template engine.

Furthermore, this library requires more initial configuration than the other libraries mentioned, especially if the developer needs to create complex templates to integrate with the application's current design system or create custom table components. However, this initial complexity is necessary to ensure a robust communication between client and server and that all tables created have a coherent interface and can be easily maintained and modified.

## 14 Future work and enhancements

The project is still in its early stages, and there are several areas that require further development and improvement. Some of the key areas to focus on include:

### 14.1 Export functionality

This involves allowing the user to choose the export format and select which columns to include. Additionally, the user should have the option to apply the current filters to the exported data.

### 14.2 Caching support

Caching the data on these tables can significantly improve performance, especially on tables that contain large amounts of data and are rarely modified. Since the developer has access to the querysets associated with the table, developers can create basic caching using Django's own caching framework. However the library should provide a cache schema that allows the user to easily configure how and which tables should be cached to allow the developer to easily set up an efficient caching system.

### 14.3 Built-in filter types

Currently the library supports only the dropdown filter. And although it enables the developer to create their own custom filters depending on the project's necessity, adding more built-in filter types that are common in tables can make the creation of tables even easier.

## 15 Conclusion

This paper presented Django Datatable Builder, a comprehensive library that addresses the complexity of integrating DataTables.js with Django applications. By employing the Builder design pattern, we have created a solution that significantly reduces the implementation burden on developers while maintaining the flexibility required for real-world applications.

The library successfully achieves its primary objectives by providing a unified interface that handles both client and server-side aspects of dynamic table creation. Through the encapsulation of DataTables integration complexity, developers can now create feature-rich tables with minimal code, focusing on business logic rather than implementation details. The builder pattern API ensures code reusability across multiple tables while maintaining consistency throughout the application.

Our demonstration of the Greendale College Management System shows the library's practical applicability across diverse data models and business requirements. The implementation showcases how Django Datatable Builder handles complex scenarios including relational data, conditional actions, custom formatting, and filtering mechanisms while maintaining security best practices through built-in XSS protection, CSRF token integration, and proper data sanitization.

When compared to existing solutions, Django Datatable Builder distinguishes itself through its comprehensive approach that combines ease of use with extensive customization capabilities. Unlike libraries that require significant modifications to existing code or those limited to specific use cases, our solution provides a balanced approach that integrates seamlessly with Django's existing patterns while offering full control over table behavior and appearance.

The impact of this library extends beyond individual developer productivity. By standardizing the implementation of dynamic tables across Django projects, it promotes best practices, reduces potential security vulnerabilities, and ensures maintainable code. Organizations can achieve faster development cycles and more consistent user interfaces, ultimately leading to improved user experience and reduced maintenance costs.

While future enhancements such as export functionality, caching support, and additional filter types will further expand the library's capabilities, the current implementation already provides substantial value to the Django development community.

## References

- [1] Django Software Foundation. 2025. Django: The Web Framework for Perfectionists with Deadlines. <https://www.djangoproject.com/> Accessed: 2025-07-23.
- [2] Martin Fowler. 2002. *Patterns of enterprise application architecture*. Addison-Wesley.
- [3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [4] David Jean Louis. 2025. djangoestframework-datatables: Django REST Framework integration with DataTables. <https://pypi.org/project/djangoestframework-datatables/> Accessed: 2025-07-30.
- [5] Mozilla Developer Network. 2025. Event bubbling. [https://developer.mozilla.org/en-US/docs/Learn\\_web\\_development/Core/Scripting/Event\\_bubbling/](https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Event_bubbling/) Accessed: 2025-08-16.
- [6] Mario Orlandi. 2025. django-ajax-datatable: Server-side Django DataTables with Ajax. <https://pypi.org/project/django-ajax-datatable/> Accessed: 2025-07-30.
- [7] Allan Rosen. 2025. DataTables: Table Plug-in for jQuery. <https://datatables.net/> Accessed: 2025-07-23.
- [8] Pivotal Energy Solutions. 2025. django-datatable-view: Server-side processing for DataTables in Django. <https://pypi.org/project/django-datatable-view/> Accessed: 2025-07-30.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009