
Piccolo Admin

Release 1.3.3

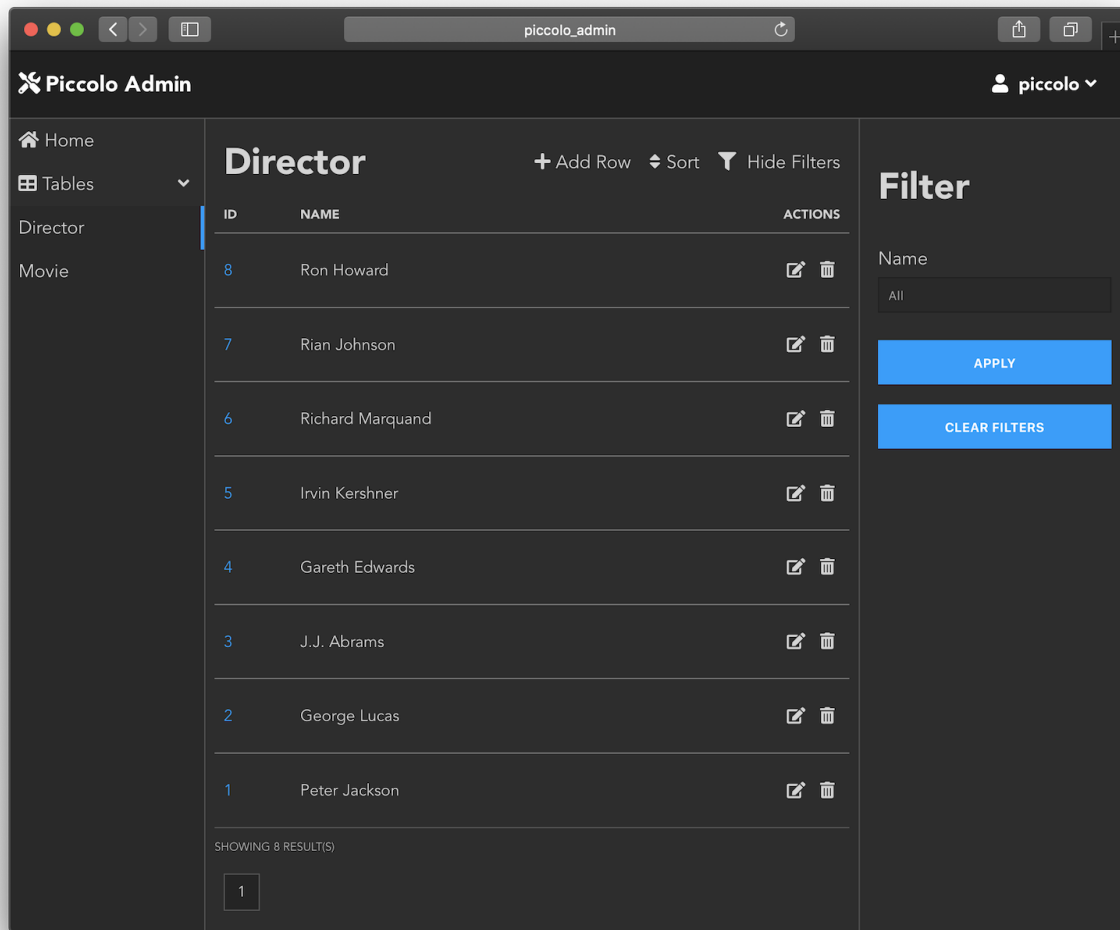
Daniel Townsend

May 02, 2024

CONTENTS

1	Demo	3
2	Table of Contents	5
	Index	67

Piccolo Admin is a powerful admin interface / content management system for Python.



It was created at a design agency to serve the needs of customers who demand a high quality, beautiful admin interface for their websites. It's a modern alternative to tools like Wordpress and Django Admin.

It's built using the latest technologies, with Vue.js on the front end, and a powerful REST backend.

Some of it's standout features:

- Powerful data filtering
- Builtin security
- Media support, both locally and in S3 compatible services
- Dark mode support
- CSV exports
- Easily create custom forms
- Works on mobile and desktop
- Use standalone, or integrate it easily with ASGI apps like FastAPI, and Starlette
- Multilingual out of box

- Bulk actions, like updating and deleting data
- Flexible UI - only show the columns you want your users to see

CHAPTER ONE

DEMO

Try it [online](#) in read-only mode (username: piccolo, password: piccolo123).

TABLE OF CONTENTS

2.1 Installation

Python 3.8 or above is required.

Install `piccolo_admin`, ideally inside a `virtualenv`.

```
pip install piccolo_admin
```

2.1.1 Local demo

To run a demo locally:

```
admin_demo
```

And then just launch `localhost:8000` in your browser. Login using `username: piccolo`, `password: piccolo123`.

To see what happens behind the scenes, see `piccolo_admin/example.py`.

In a few lines of code we are able to:

- Define our tables
- Setup a database
- Create a REST API
- Setup a web server and admin interface

Next we'll look at integrating the Piccolo admin properly into your own project.

2.2 ASGI

Since the admin is an `ASGI app`, you can either run it standalone like in the demo, or integrate it with a larger ASGI app.

Hint: Piccolo can help you `create a new ASGI app` using `piccolo asgi new`.

For example, using Starlette routes:

```
import uvicorn
from movies.endpoints import HomeEndpoint
from movies.tables import Director, Movie
from starlette.routing import Mount, Route, Router

from piccolo_admin.endpoints import create_admin

# The `allowed_hosts` argument is required when running under HTTPS. It's
# used for additional CSRF defence.
admin = create_admin([Director, Movie], allowed_hosts=["my_site.com"])

router = Router(
    [
        Route(path="/", endpoint=HomeEndpoint),
        Mount(path="/admin/", app=admin),
    ]
)

if __name__ == "__main__":
    uvicorn.run(router)
```

2.2.1 FastAPI example

Here's a complete example of a FastAPI app using Piccolo admin.

```
# app.py
from fastapi import FastAPI
from fastapi.routing import Mount
from movies.tables import Director, Movie
from piccolo.engine import engine_finder

from piccolo_admin.endpoints import create_admin

app = FastAPI(
    routes=[
        Mount(
            path="/admin/",
            app=create_admin(
                tables=[Director, Movie],
                # Specify a different site name in the
                # admin UI (default Piccolo Admin):
                site_name="My Site Admin",
                # Required when running under HTTPS:
                # allowed_hosts=["my_site.com"],
            ),
        ),
    ],
)
```

(continues on next page)

(continued from previous page)

```

@app.on_event("startup")
async def open_database_connection_pool():
    engine = engine_finder()
    await engine.start_connection_pool()

@app.on_event("shutdown")
async def close_database_connection_pool():
    engine = engine_finder()
    await engine.close_connection_pool()

```

To run `app.py` use:

```
uvicorn app:app --port 8000 --host 0.0.0.0
```

Now you can go to `localhost:8000/admin` and log in as an admin user (see [Authentication](#) for how to create users).

2.2.2 Source

```

class piccolo_admin.endpoints.create_admin(
    tables: Sequence[Union[Type[Table], TableConfig]], forms:
        List[FormConfig] = [], auth_table:
            Optional[Type[BaseUser]] = None, session_table:
            Optional[Type[SessionsBase]] = None, session_expiry:
            timedelta = timedelta(hours=1), max_session_expiry:
            timedelta = timedelta(days=7), increase_expiry:
            Optional[timedelta] = timedelta(minutes=20), page_size:
            int = 15, read_only: bool = False, rate_limit_provider:
            Optional[RateLimitProvider] = None, production: bool =
            False, site_name: str = 'Piccolo Admin',
            default_language_code: str = 'auto', translations:
            Optional[List[Translation]] = None, auto_include_related:
            bool = True, allowed_hosts: Sequence[str] = [], debug:
            bool = False, sidebar_links: Dict[str, str] = {})

```

Parameters

- **tables** – Each of the tables will be added to the admin.
- **forms** – For each `FormConfig` specified, a form will automatically be rendered in the user interface, accessible via the sidebar.
- **auth_table** – Either a `BaseUser`, or `BaseUser` subclass table, which is used for fetching users. Defaults to `BaseUser` if none is specified.
- **session_table** – Either a `SessionsBase`, or `SessionsBase` subclass table, which is used for storing and querying session tokens. Defaults to `SessionsBase` if none is specified.
- **session_expiry** – How long a session is valid for.
- **max_session_expiry** – The maximum time a session is valid for, taking into account any refreshes using `increase_expiry`.

- **increase_expiry** – If set, the `session_expiry` will be increased by this amount if it's close to expiry.
- **page_size** – The admin API paginates content - this sets the default number of results on each page.
- **read_only** – If True, all non auth endpoints only respond to GET requests - the admin can still be viewed, and the data can be filtered. Useful for creating online demos.
- **rate_limit_provider** – Rate limiting middleware is used to protect the login endpoint against brute force attack. If not set, an `InMemoryLimitProvider` will be configured with reasonable defaults.
- **production** – If True, the admin will enforce stronger security - for example, the cookies used will be secure, meaning they are only sent over HTTPS.
- **site_name** – Specify a different site name in the admin UI (default 'Piccolo Admin').
- **default_language_code** – Specify the default language used in the admin UI. The value should be an [IETF language tag](#), for example 'en' for English. To see available values see `piccolo_admin/translations/data.py`. The UI will be automatically translated into this language. If a value of 'auto' is specified, then we check the user's browser for the language they prefer, using the `navigator.language` JavaScript API.
- **translations** – Specify which translations are available. By default, we use every translation in `piccolo_admin/translations/data.py`.

Here's an example - if we know our users only speak English or Croatian, we can specify that only those translations are visible in the language selector in the UI:

```
from piccolo.translations.data import ENGLISH, CROATIAN

create_admin(
    tables=[TableA, TableB],
    default_language_code='hr',
    translations=[ENGLISH, CROATIAN]
)
```

You can also use this to provide your own translations, if there's a language we don't currently support (though please open a PR to add it!):

```
from piccolo.translations.models import Translation
from piccolo.translations.data import ENGLISH

MY_LANGUAGE = Translation(
    language_code='xx',
    language_name='My Language',
    translations={
        'Welcome': 'XXXXX',
        ...
    }
)

create_admin(
    tables=[TableA, TableB],
    default_language_code='xx',
    translations=[ENGLISH, MY_LANGUAGE]
)
```

- **auto_include_related** – If a table has foreign keys to other tables, those tables will also be included in the admin by default, if not already specified. Otherwise the admin won't work as expected.
- **allowed_hosts** – This is used by the [CSRFMiddleware](#) as an additional layer of protection when the admin is run under HTTPS. It must be a sequence of strings, such as ['my_site.com'].
- **debug** – If True, debug mode is enabled. Any unhandled exceptions will return a stack trace, rather than a generic 500 error. Don't use this in production!
- **sidebar_links** – Custom links in the navigation sidebar. Example uses cases:
 - Providing a quick way to get to specific pages with pre-applied filters/sorting.
 - Linking to relative external websites.

Here's a full example:

```
from piccolo_admin.endpoints import create_admin

create_admin(
    tables=[Movie, Director],
    sidebar_links={
        "Top Movies": "/admin/#/movie?__order=-box_office",
        "Google": "https://google.com"
    },
)
```

2.3 Authentication

2.3.1 Session table

Piccolo admin uses [session auth](#), which requires a Session database table.

Add `piccolo_admin.piccolo_app` to the `APP_REGISTRY` in your `piccolo_conf.py` project file, then run migrations:

```
piccolo migrations forwards session_auth
```

To learn more about the Piccolo project files, check out the [Piccolo ORM docs](#).

2.3.2 Creating users

`BaseUser` is a `Table` you can use to store and authenticate your users. You need this table to be able to create users with admin privileges. `BaseUser` is shipped out of the box with Piccolo and you just need to run the migrations.

Run the migrations:

```
piccolo migrations forwards user
```

Create a new user.

```
piccolo user create
```

You will be prompted to enter a username, email address and password (you will be asked to enter your password twice for confirmation). Make sure you enter y when asked if it's an admin user, otherwise the user won't be able to login to the Piccolo admin GUI.

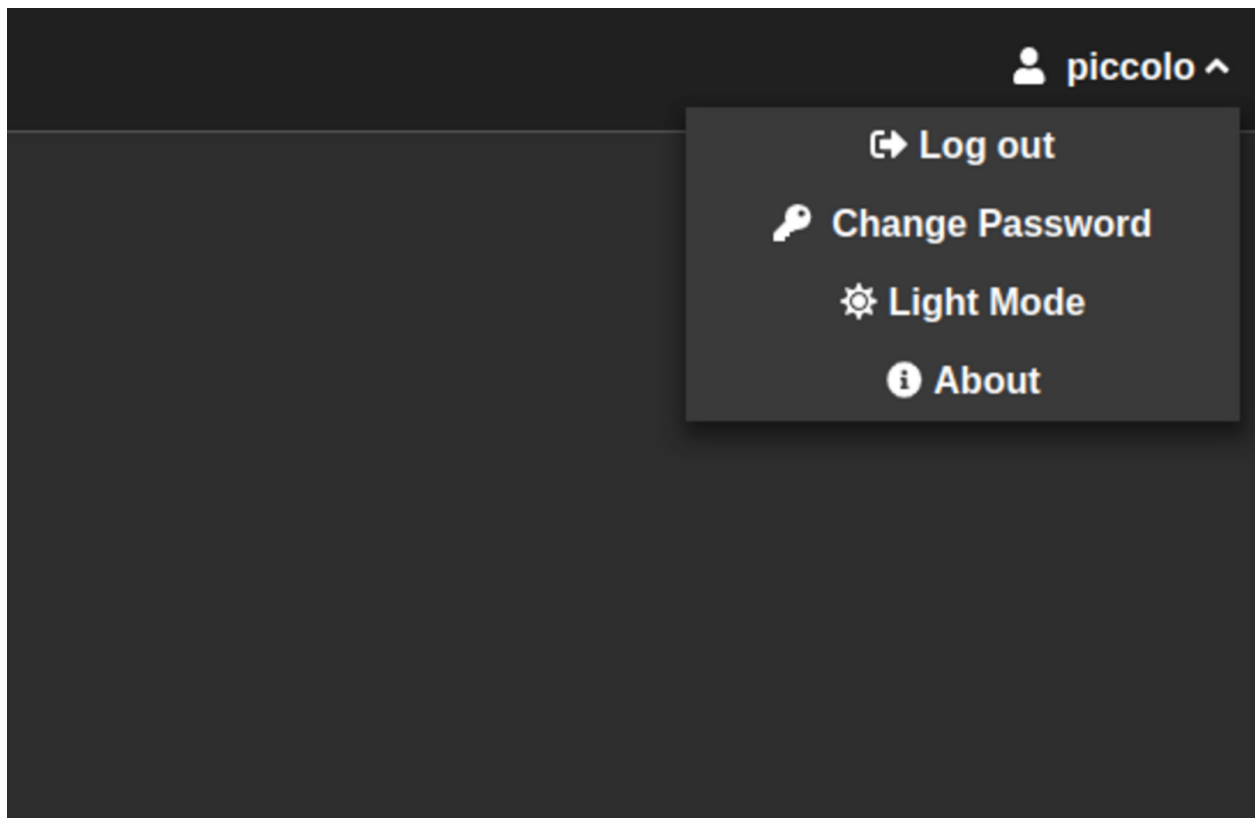
Warning: Non-admin users can't login to the Piccolo admin GUI.

You can also change a user's password:

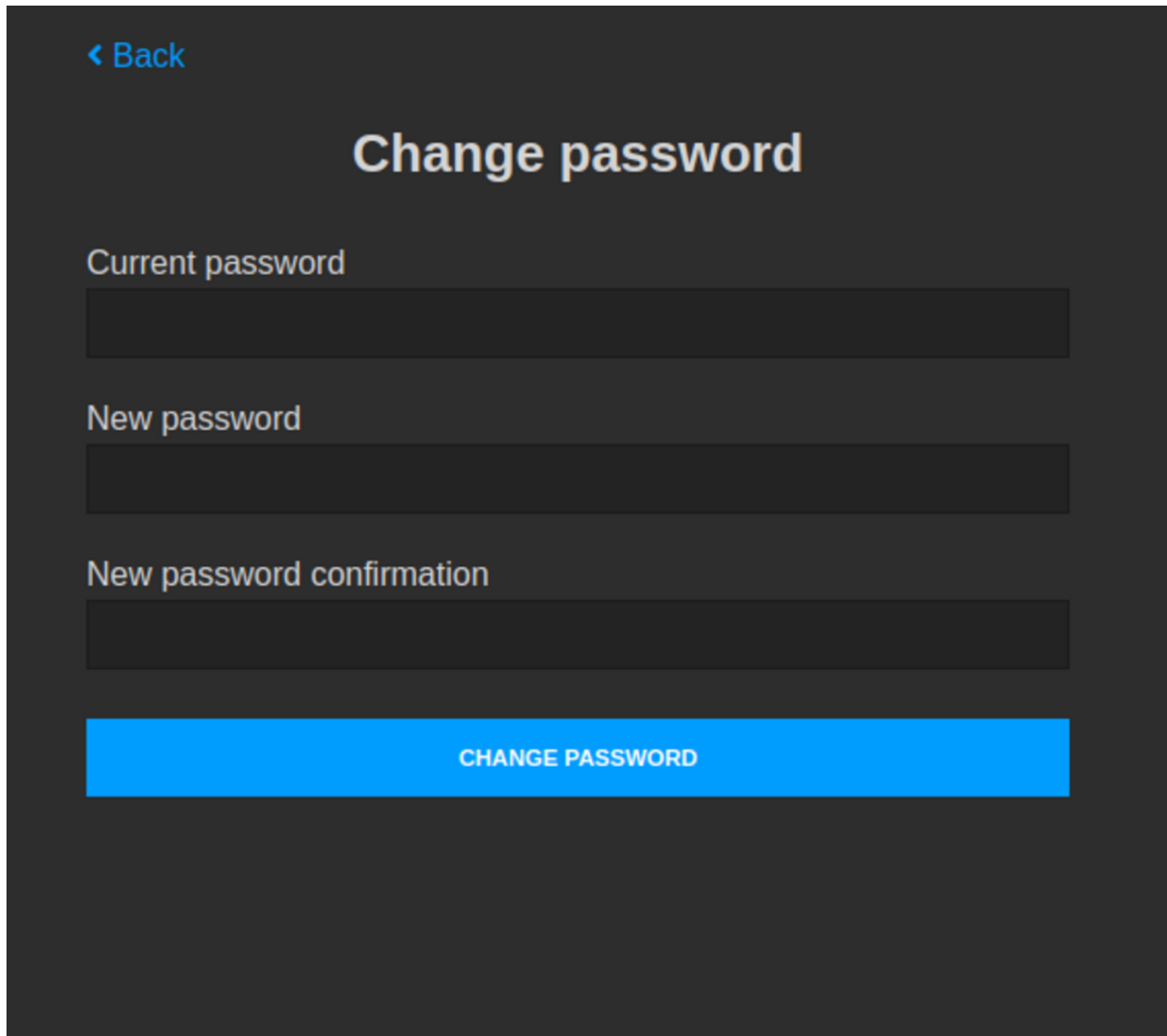
```
piccolo user change_password
```

2.3.3 Change admin password in the UI

The admin user also has the option to change their password in the Piccolo Admin UI. This option can be selected from the user dropdown menu.



After that, you will be shown a form in which you can change your administrator password.



[< Back](#)

Change password

Current password

New password

New password confirmation

CHANGE PASSWORD

2.4 Help Text

2.4.1 Columns

Sometimes you will want to provide hints to your users about what the form fields mean.

Using this table as an example:

```
from piccolo.table import Table
from piccolo.columns import Varchar, Numeric

class Movie(Table):
    name = Varchar(length=300)
    box_office = Numeric(digits=(5, 1))
```

It isn't immediately clear what `box_office` means. We can add a `help_text` attribute to the column:

```
from piccolo.table import Table
from piccolo.columns import Varchar, Numeric

class Movie(Table):
    name = Varchar(length=300)
    box_office = Numeric(
        digits=(5, 1),
        help_text="In millions of US dollars."
    )
```

Piccolo Admin will then show a tooltip next to this field.

2.4.2 Tables

You can also provide help text about the table itself.

```
from piccolo.table import Table
from piccolo.columns import Varchar, Numeric

class Movie(Table, help_text="Movies which were released in the cinema."):
    name = Varchar(length=300)
    box_office = Numeric(
        digits=(5, 1),
        help_text="In millions of US dollars."
    )
```

Piccolo Admin will then show a tooltip next to the table name in the row listing page.

2.5 TableConfig

When using `create_admin`, you can pass in normal `Table` classes:

```
from piccolo_admin.endpoints import create_admin

create_admin([Director, Movie])
```

Alternatively, you can pass in `TableConfig` instances (or mix and match them).

By passing in a `TableConfig` you have extra control over how the UI behaves for that table. This is particularly useful when you have a `Table` with lots of columns.

In the future, `TableConfig` will be extended to allow finer grained control over the UI.

2.5.1 visible_columns

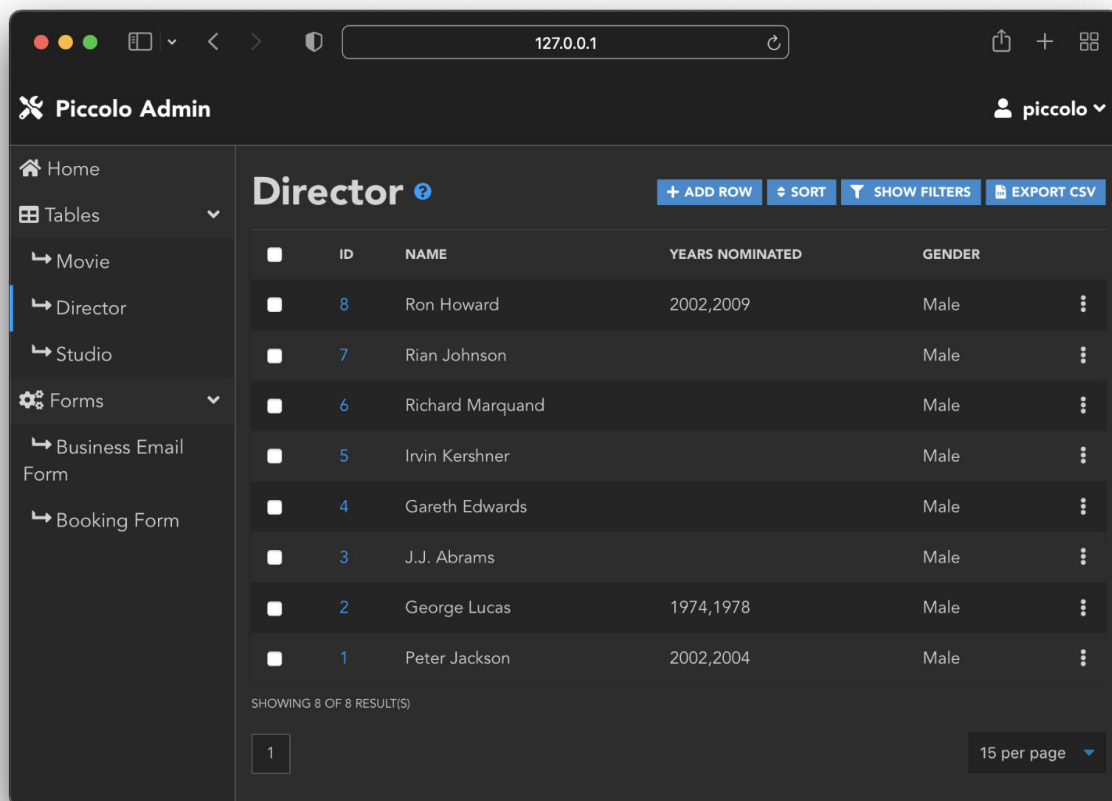
We can set which columns are visible in the list view:

```
from piccolo_admin.endpoints import TableConfig

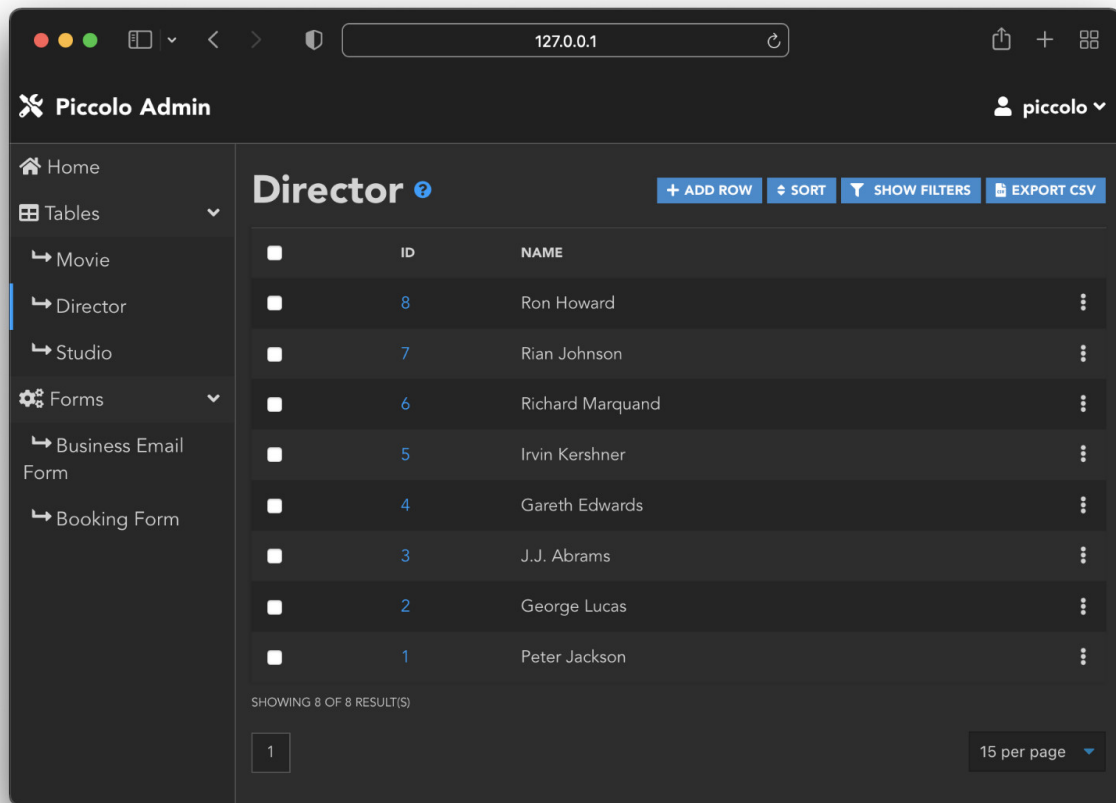
movie_config = TableConfig(Movie, visible_columns=[Movie.id, Movie.name])

create_admin([Director, movie_config])
```

Here is the UI when just passing in a Table:



Here is the UI when just passing in a TableConfig instance instead (fewer columns are visible):



2.5.2 visible_filters

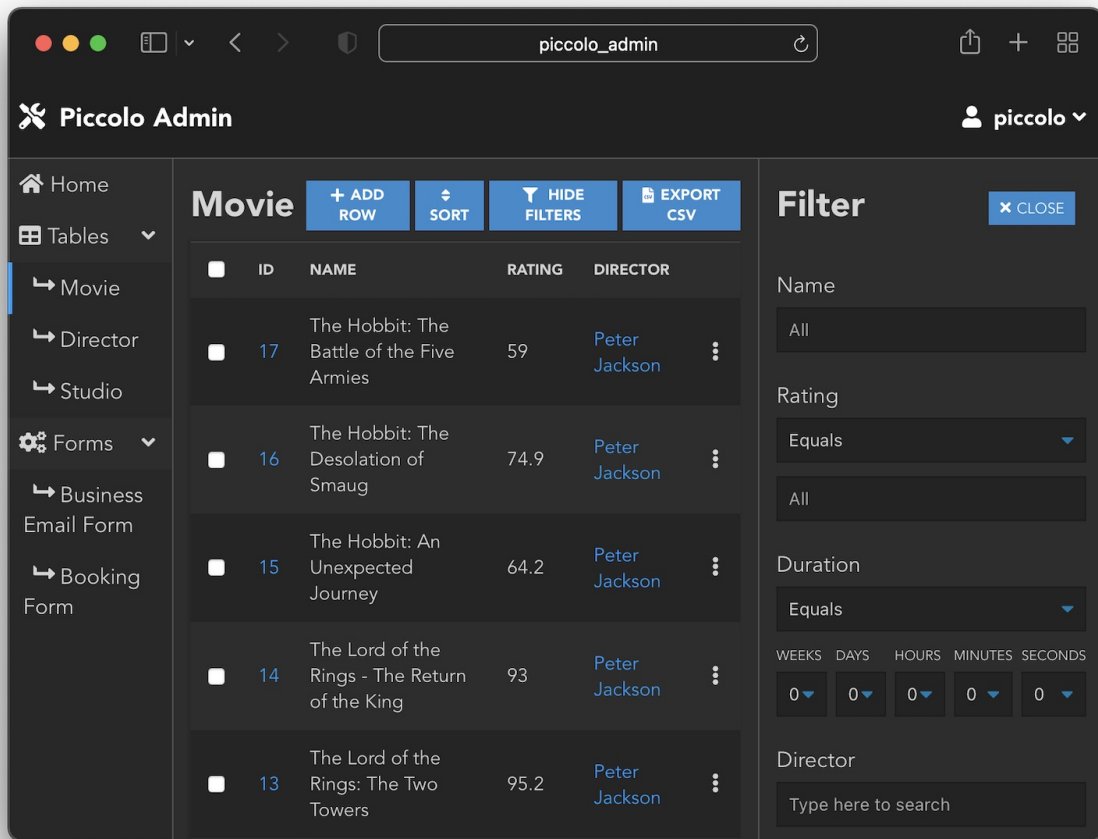
We can set which columns are visible in the filter sidebar:

```
from piccolo_admin.endpoints import TableConfig

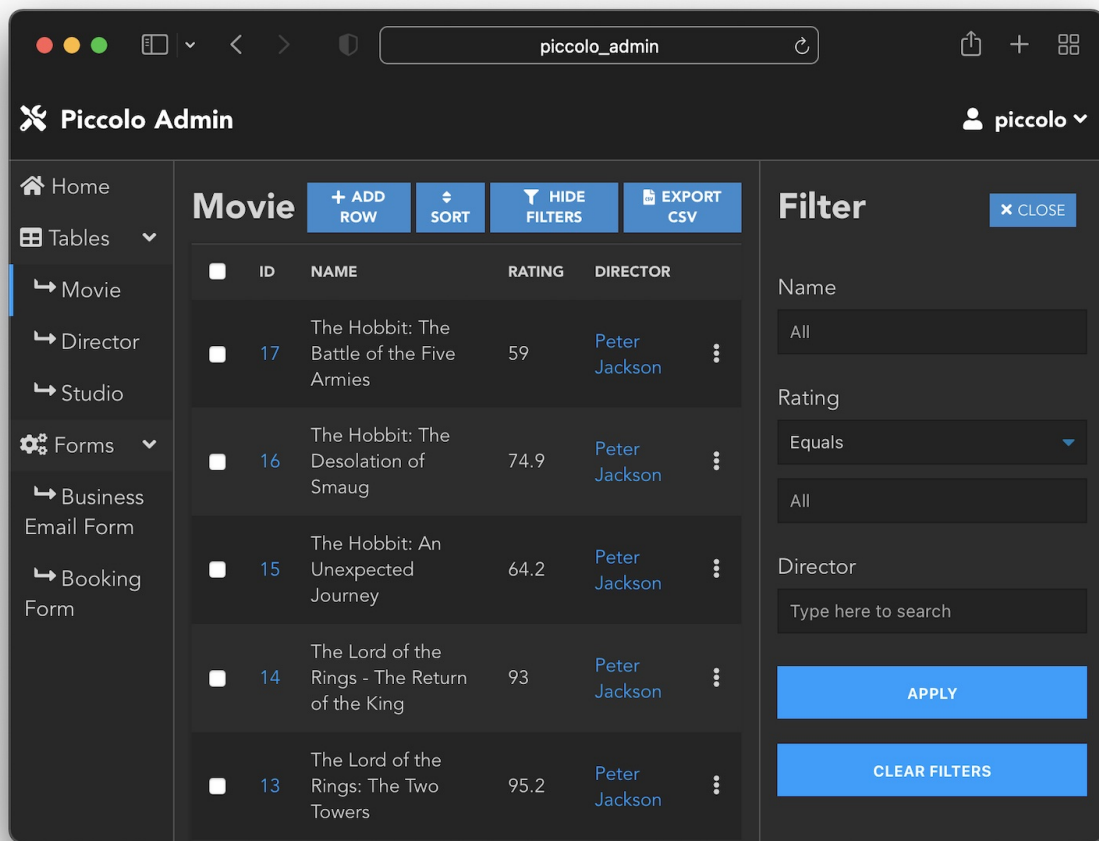
movie_config = TableConfig(
    Movie,
    visible_filters=[
        Movie.name, Movie.rating, Movie.director,
    ]
)

create_admin([Director, movie_config])
```

Here is the UI when just passing in a Table:



Here is the UI when just passing in a `TableConfig` instance instead (fewer filters are visible in the sidebar):



2.5.3 rich_text_columns

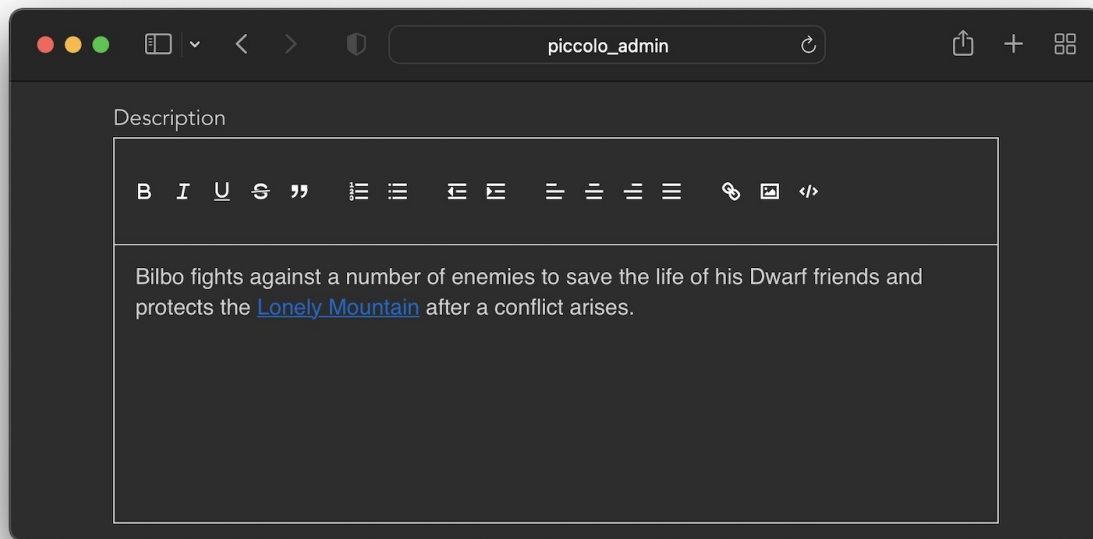
We can specify which Text columns will use a rich text editor.

```
from piccolo_admin.endpoints import TableConfig

movie_config = TableConfig(
    Movie,
    rich_text_columns=[
        Movie.description
    ]
)

create_admin([Director, movie_config])
```

This allows the user to add hyperlinks, and basic formatting to their content, without having to write HTML.



2.5.4 hooks

Can be used to run custom logic when a row is created, modified, or deleted.

```
from piccolo_admin.endpoints import TableConfig
from piccolo_api.crud.hooks import Hook, HookType

async def my_save_hook(row: Movie):
    # Insert custom logic here
    return row

movie_config = TableConfig(
    Movie,
    hooks=[
        Hook(hook_type=HookType.pre_save, callable=my_save_hook)
    ]
)

create_admin([Director, movie_config])
```

To learn more about hooks, see the [Hook](#) docs in Piccolo API.

2.5.5 media_storage

Allows you store files (video / media / audio etc) in certain columns. See [Media Storage](#).

2.5.6 validators

Allows fine grained access control over each API endpoint. See [TableConfig](#) for more information.

2.5.7 menu_group

We can set groups of tables in the table navigation sidebar. This is useful when we have many tables and in this way we can organize the tables into groups for better visibility:

```
from piccolo_admin.endpoints import TableConfig

movie_config = TableConfig(
    Movie,
    menu_group="Movies"
)

director_config = TableConfig(
    Director,
    menu_group="Movies"
)

create_admin([director_config, movie_config])
```

2.5.8 link_column

We use the primary key as the link to the edit page, but we can specify an alternative column to use as a link. If we specify the foreign key column as link Piccolo Admin will raise `ValueError`:

```
from piccolo_admin.endpoints import TableConfig

movie_config = TableConfig(
    Movie,
    link_column=Movie.name
)

create_admin([movie_config])
```

2.5.9 order_by

By default, the primary key is used to order the results, but we can specify one or more columns to order by instead. Here we use the `rating` column in descending order:

```
from piccolo_admin.endpoints import TableConfig, OrderBy

movie_config = TableConfig(
    Movie,
    order_by=[OrderBy(Movie.rating, ascending=False)]
)

create_admin([movie_config])
```

Here we order by the `rating` and `title` columns:

```
movie_config = TableConfig(
    Movie,
    order_by=[
        OrderBy(Movie.rating, ascending=False),
        OrderBy(Movie.title, ascending=True)
    ]
)
```

This means that the results are first ordered by `rating`, and any rows with an equal rating are then sorted by `title`. Note you can still override the order in the UI.

2.5.10 Source

```
class piccolo_admin.endpoints.TableConfig(table_class: Type[Table], visible_columns:
    Optional[List[Column]] = None, exclude_visible_columns:
    Optional[List[Column]] = None, visible_filters:
    Optional[List[Column]] = None, exclude_visible_filters:
    Optional[List[Column]] = None, rich_text_columns:
    Optional[List[Column]] = None, hooks:
    Optional[List[Hook]] = None, media_storage:
    Optional[Sequence[MediaStorage]] = None, validators:
    Optional[Validators] = None, menu_group: Optional[str] =
    None, link_column: Optional[Column] = None, order_by:
    Optional[List[OrderBy]] = None, time_resolution:
    Optional[Dict[Union[Timestamp, Timestampz, Time],
    Union[float, int]]] = None)
```

Gives the user more control over how a `Table` appears in the UI.

Parameters

- **table_class** – The `Table` class to configure.
- **visible_columns** – If specified, only these columns will be shown in the list view of the UI. This is useful when you have a lot of columns.

- **exclude_visible_columns** – You can specify this instead of `visible_columns`, in which case all of the Table columns except the ones specified will be shown in the list view.
- **visible_filters** – If specified, only these columns will be shown in the filter sidebar of the UI. This is useful when you have a lot of columns.
- **exclude_visible_filters** – You can specify this instead of `visible_filters`, in which case all of the Table columns except the ones specified will be shown in the filter sidebar.
- **rich_text_columns** – You can specify `rich_text_columns` if you want a WYSIWYG editor on certain Piccolo `Text` columns. Any columns not specified will use a standard HTML textarea tag in the UI.
- **hooks** – These are passed directly to `PiccoloCRUD`, which powers Piccolo Admin under the hood. It allows you to run custom logic when a row is modified.
- **media_storage** – These columns will be used to store media. We don't directly store the media in the database, but instead store a string, which is a unique identifier, and can be used to retrieve a URL for accessing the file. Piccolo Admin automatically renders a file upload widget for each media column in the UI.
- **validators** – The `Validators` are passed directly to `PiccoloCRUD`, which powers Piccolo Admin under the hood. It allows fine grained access control over each API endpoint. For example, limiting which users can POST data:

```
from piccolo_api.crud.endpoints import PiccoloCRUD
from starlette.exceptions import HTTPException
from starlette.requests import Request

async def manager_only(
    piccolo_crud: PiccoloCRUD,
    request: Request
):
    # The Piccolo `BaseUser` can be accessed from the request.
    user = request.user.user

    # Assuming we have another database table where we record
    # users with certain permissions.
    manager = await Manager.exists().where(manager.user == user)

    if not manager:
        # Raise a Starlette exception if we want to reject the
        # request.
        raise HTTPException(
            status_code=403,
            detail="Only managers are allowed to do this"
        )

admin = create_admin(
    tables=TableConfig(
        Movie,
        validators=Validators(post_single=[manager_only])
    )
)
```


- **menu_group** – If specified, tables can be divided into groups in the table menu. This is useful when you have many tables that you can organize into groups for better visibility.
- **link_column** – In the list view of Piccolo Admin, we use the primary key to link to the edit page. However, if the primary key column is hidden, due to `visible_columns` or `exclude_visible_columns`, then we need to specify an alternative column to use as the link.
- **order_by** – If specified, the rows are sorted by `order_by`, otherwise the default `primary_key` column is used to sort the rows.
- **time_resolution** – Controls the resolution of Time columns, and the time component of Timestamp / Timestamptz columns. The units are given in seconds. Some examples:
 - 0.001 - the max resolution is 1 millisecond (this is the minimum currently allowed by HTML input fields)
 - 1 - the max resolution is 1 second (the default)
 - 60 - the max resolution is 1 minute

2.6 Custom Forms

Piccolo Admin has the ability to turn a Pydantic model into a form in the UI, without writing any frontend code.

Here's an example of a form which sends email, using FastAPI:

```
import smtplib

from fastapi import FastAPI
from fastapi.routing import Mount
from home.tables import Task
from pydantic import BaseModel

from piccolo_admin.endpoints import FormConfig, create_admin

# Pydantic model for form
class EmailFormModel(BaseModel):
    email: str
    title: str
    content: str

# Send email handler
def email_endpoint(request, data):
    sender = "info@example.com"
    receivers = data.email

    message = f"""From: Sender <info@example.com>
To: Receiver <{data.email}>
Subject: {data.title}
{data.content}
"""
```

(continues on next page)

(continued from previous page)

```

try:
    smtpObj = smtplib.SMTP("localhost:1025")
    smtpObj.sendmail(sender, receivers, message)
    print("Successfully sent email")
except smtplib.SMTPException:
    print("Error: unable to send email")

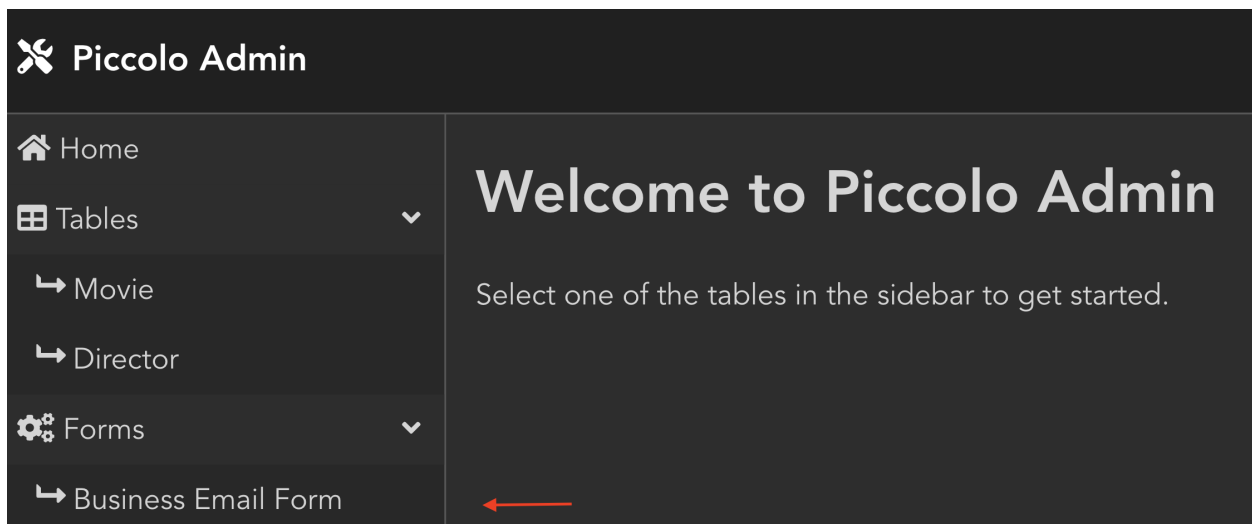
return "Email sent"

app = FastAPI(
    routes=[
        Mount(
            "/admin/",
            create_admin(
                tables=[Task],
                forms=[
                    FormConfig(
                        name="Business Email Form",
                        pydantic_model=EmailFormModel,
                        endpoint=email_endpoint,
                    ),
                ],
            ),
        ),
    ],
)

# For Starlette it is identical, just `app = Starlette(...)`

```

Piccolo Admin will then show a custom form in the UI.



The screenshot shows the Piccolo Admin interface for a 'Business Email Form'. At the top, there's a header with the Piccolo Admin logo and a user profile 'piccolo'. Below the header, there's a 'Back' link. The main heading is 'Business Email Form'. The form consists of three input fields: 'Email', 'Title', and 'Content'. At the bottom of the form is a prominent blue 'SUBMIT' button.

To process the form, you only need to specify the Pydantic model and function which processes your custom form and Piccolo Admin will do the rest, like in the above example.

2.6.1 Source

```
class piccolo_admin.endpoints.FormConfig(name: str, pydantic_model: Type[PydanticModel], endpoint:
    Callable[[Request, PydanticModel], Union[str, None,
    Coroutine]], description: Optional[str] = None)
```

Used to specify forms, which are passed into `create_admin`.

Parameters

- **name** – This will be displayed in the UI in the sidebar.
- **pydantic_model** – This determines which fields to display in the form, and is used to de-serialise the responses.
- **endpoint** – Your custom handler, which accepts two arguments - the FastAPI / Starlette request object, in case you want to access the cookies / headers / logged in user (via `request.user`). And secondly an instance of the Pydantic model. If it returns a string, it will be shown to the user in the UI as the success message. For example 'Successfully sent email'. The endpoint can be a normal function or async function.
- **description** – An optional description which is shown in the UI to explain to the user what the form is for.

Here's a full example:

```
class MyModel(pydantic.BaseModel):
    message: str = "hello world"

def my_endpoint(request: Request, data: MyModel):
    print(f"I received {data.message}")

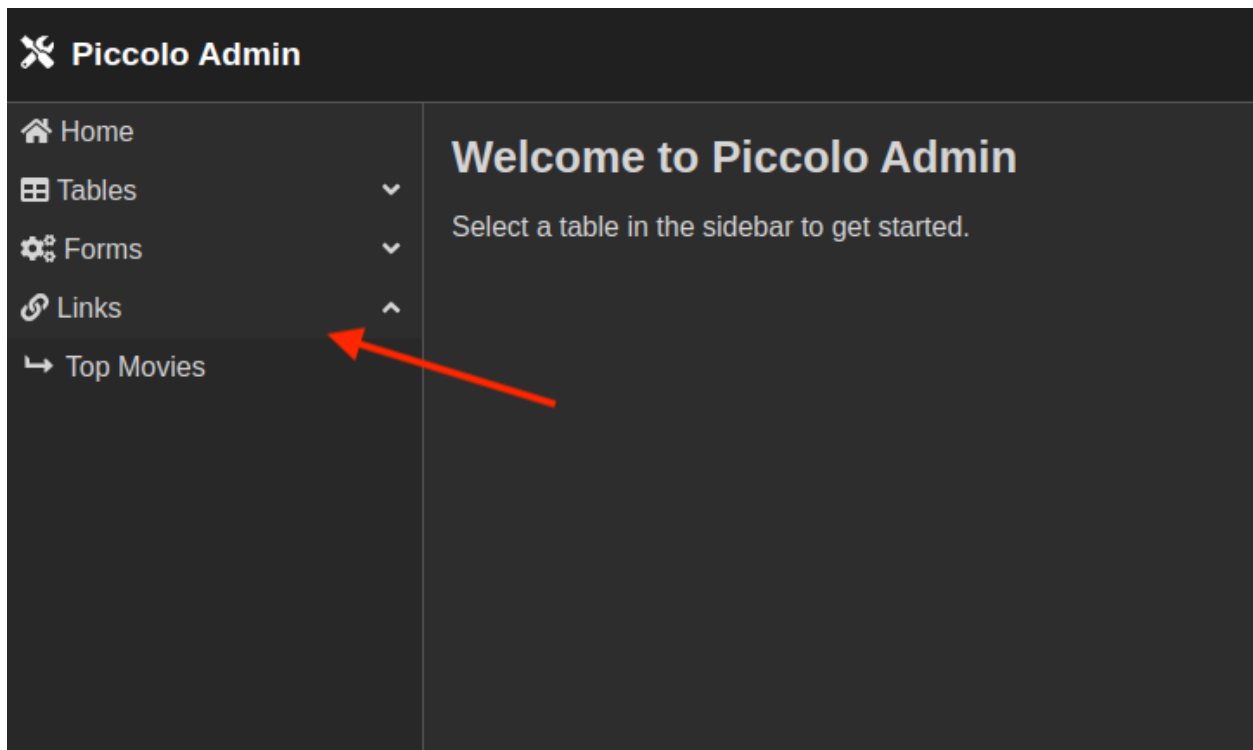
    # If we're not happy with the data raise a ValueError
    # The message inside the exception will be displayed in the UI.
    raise ValueError("We were unable to process the form.")

    # If we're happy with the data, just return a string, which
    # will be displayed in the UI.
    return "Successful."

config = FormConfig(
    name="My Form",
    pydantic_model=MyModel,
    endpoint=my_endpoint
)
```

2.7 Sidebar Links

We can specify custom links in the navigation sidebar. This feature is useful if we want a quick way to get to specific pages with pre-applied filters/sorts, or to any external websites.



For example:

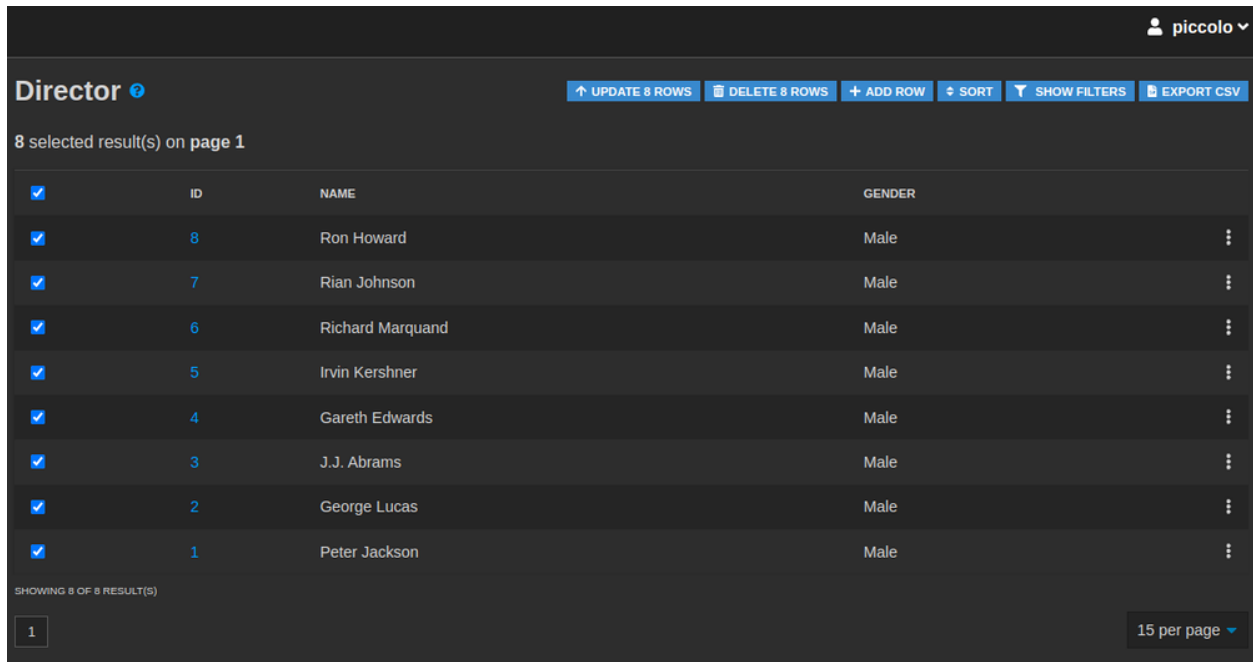
```
# app.py
from fastapi import FastAPI
from fastapi.routing import Mount
from piccolo_admin.endpoints import create_admin

from movies.tables import Director, Movie

app = FastAPI(
    routes=[
        Mount(
            path="/admin/",
            app=create_admin(
                tables=[Director, Movie],
                sidebar_links={
                    "Top Movies": "/admin/#/movie?__order=-box_office",
                    "Google": "https://google.com"
                },
            ),
        ),
    ],
)
```

2.8 Actions

Piccolo Admin allows us to perform bulk actions, by ticking the checkboxes next to the rows.



The screenshot shows the Piccolo Admin interface for the 'Director' table. At the top, there's a user profile 'piccolo' and a dropdown arrow. Below that, the table title 'Director' is followed by a help icon and a row of action buttons: 'UPDATE 8 ROWS', 'DELETE 8 ROWS', 'ADD ROW', 'SORT', 'SHOW FILTERS', and 'EXPORT CSV'. A status bar indicates '8 selected result(s) on page 1'. The table has columns for 'ID', 'NAME', and 'GENDER'. All 8 rows are selected, indicated by blue checkmarks in the first column. Each row also has a vertical ellipsis menu icon on the right. The data in the table is as follows:

ID	NAME	GENDER
8	Ron Howard	Male
7	Rian Johnson	Male
6	Richard Marquand	Male
5	Irvin Kershner	Male
4	Gareth Edwards	Male
3	J.J. Abrams	Male
2	George Lucas	Male
1	Peter Jackson	Male

At the bottom left, it says 'SHOWING 8 OF 8 RESULT(S)' and there's a page number '1' in a box. At the bottom right, there's a '15 per page' dropdown menu.

2.8.1 Bulk delete

Ability to delete multiple rows in one action. Select which rows you want to delete by checking the checkboxes and pressing the delete button.

2.8.2 Bulk update

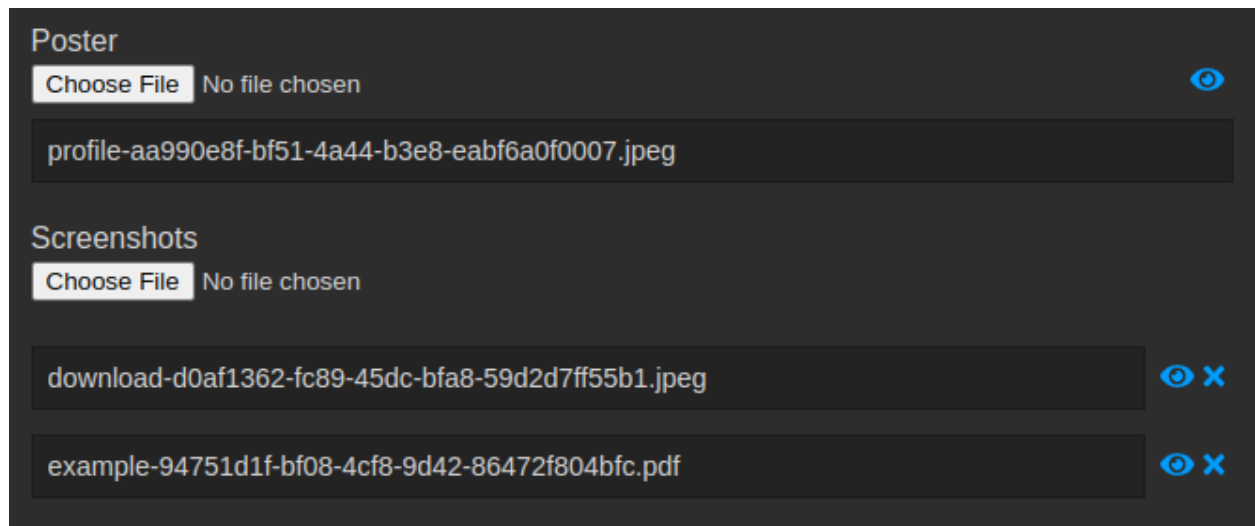
Ability to update multiple rows in one action. Select which rows you want to update by checking the checkboxes. By pressing the update button you will be asked to pass the column name and the value which you want to update in bulk.

2.8.3 Export to CSV

Export data to a CSV document by pressing the export CSV button.

2.9 Media Storage

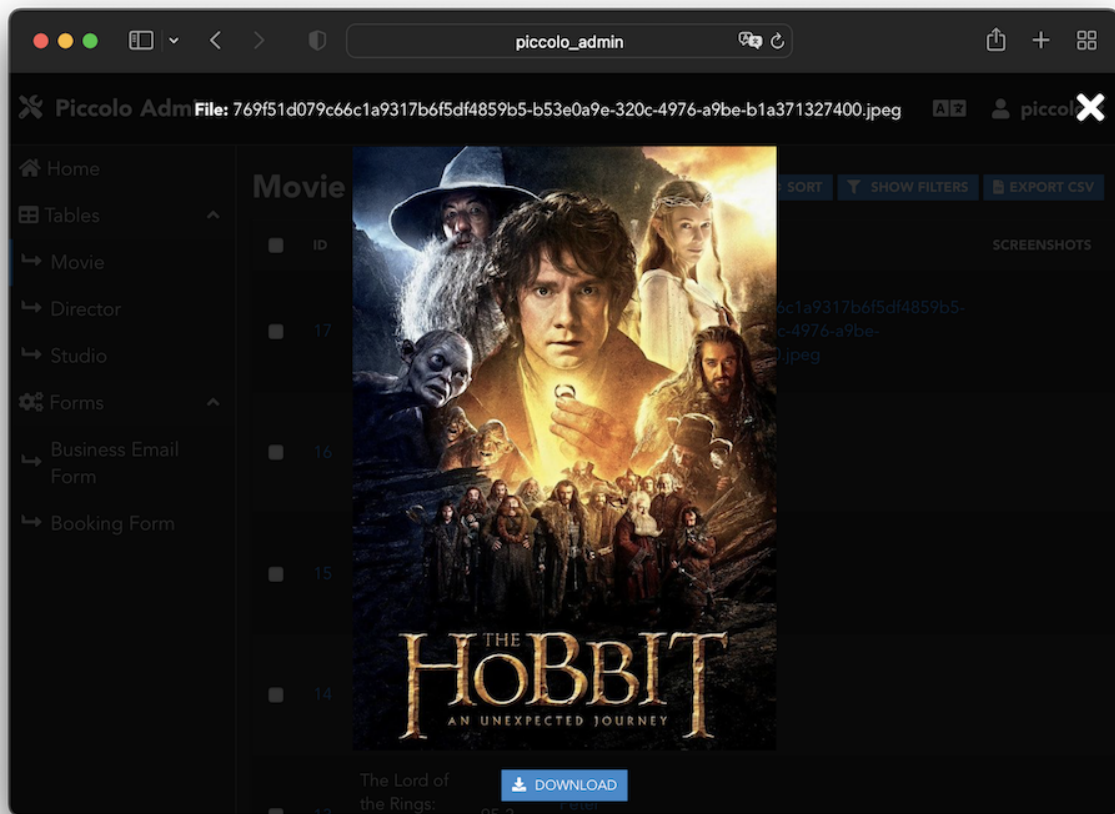
Piccolo Admin has excellent support for managing media files (images, audio, video, and more). The files can be stored locally on the server, or in S3 compatible storage.



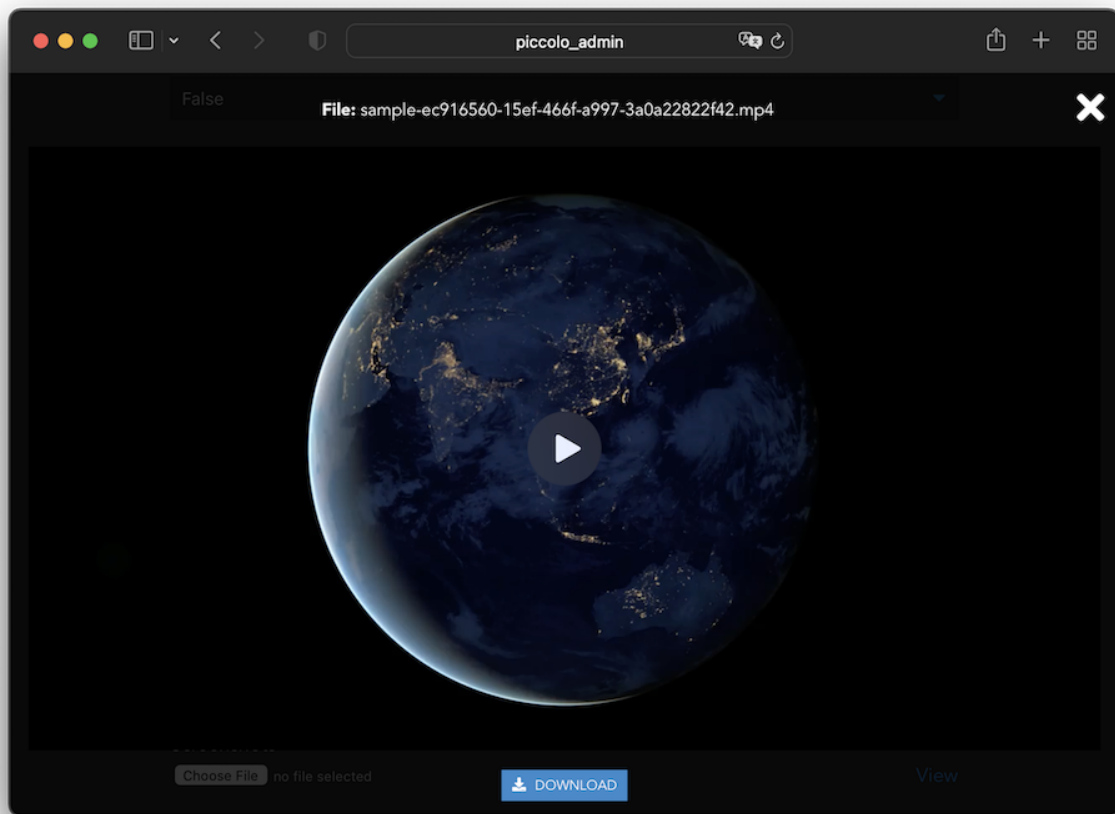
2.9.1 User Interface

You're able to view many file types within the user interface - images, videos, audio, documents, and more.

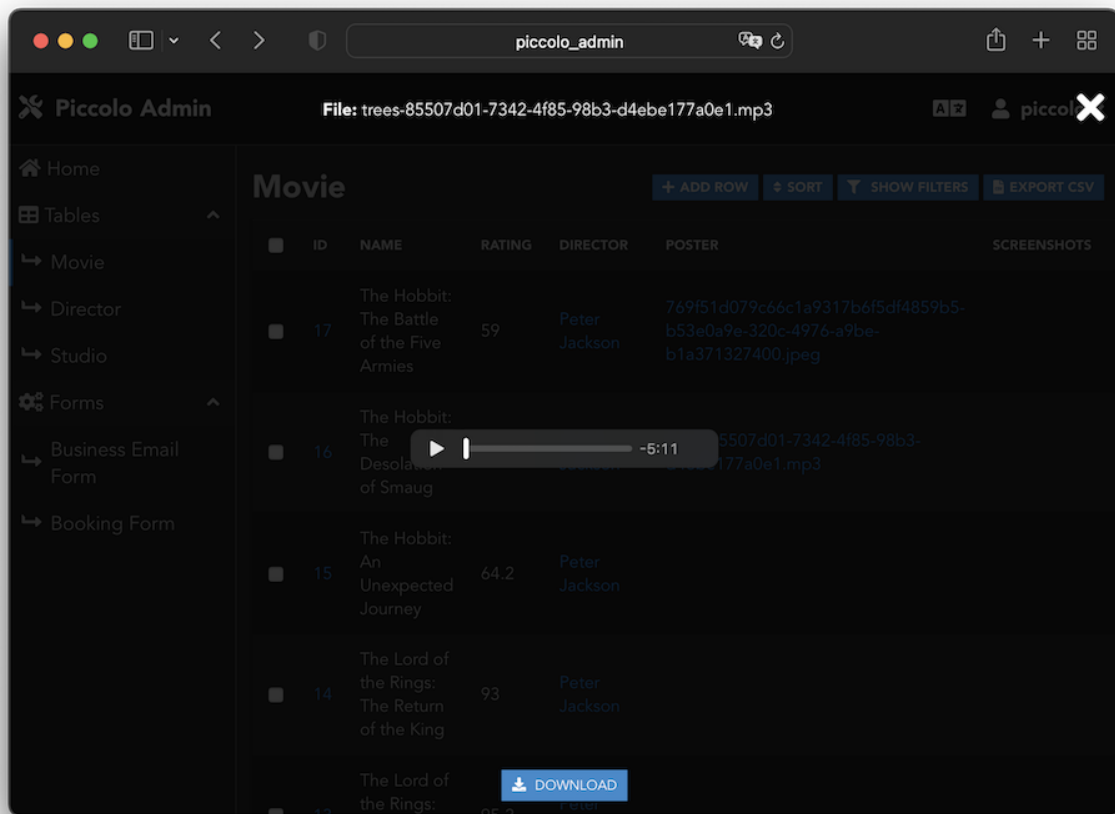
Viewing images:



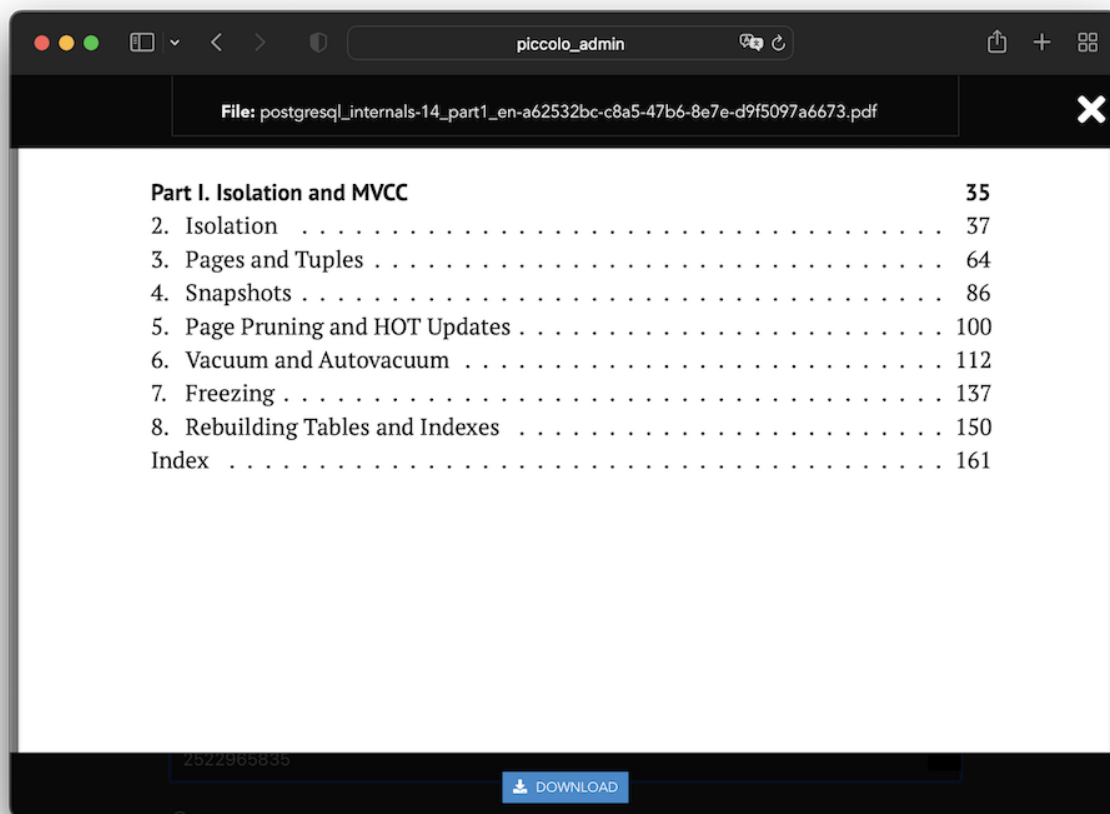
Viewing video:



Viewing audio:



Viewing PDFs:



2.9.2 Supported columns

We store the files in a folder on the server, or in a S3 bucket, and store unique references to those files in the database.

Note: We don't store the files directly in the database, because database storage is typically much more expensive than block / object storage.

An example file reference is `my-file-abc-123.jpeg`. Since it's a string, we can only store it in a database column which stores strings.

Varchar

`Varchar` is a good choice for storing file references. For example:

```
from piccolo.table import Table
from piccolo.column.column_types import Varchar

class Movie(Table):
    poster = Varchar()
```

Text

`Text` can also be used. For example:

```
from piccolo.table import Table
from piccolo.column.column_types import Text

class Movie(Table):
    poster = Text()
```

Array

We also support `Array`, but only when the `base_column` is either `Varchar` or `Text`. For example:

```
from piccolo.table import Table
from piccolo.column.column_types import Array, Varchar

class Movie(Table):
    screenshots = Array(base_column=Varchar())
```

This allows us to store multiple file references in a single column.

2.9.3 MediaStorage

For each column we want to use for media storage, we associate it with a `MediaStorage` instance.

Out of the box we have two subclasses - `LocalMediaStorage` and `S3MediaStorage`. You can also create your own subclass of `MediaStorage` to implement your own storage backend.

2.9.4 LocalMediaStorage

This stores media in a folder on the server.

Example

In order to associate a column with `LocalMediaStorage`, we do the following:

```
import os

from piccolo.columns import Array, Varchar
from piccolo_admin.endpoints import (
    TableConfig,
    create_admin
)
from piccolo_api.media.local import LocalMediaStorage

class Movie(Table):
    title = Varchar()
```

(continues on next page)

(continued from previous page)

```

poster = Varchar()
screenshots = Array(base_column=Varchar())

MEDIA_ROOT = '/srv/piccolo-admin/'

MOVIE_POSTER_MEDIA = LocalMediaStorage(
    column=Movie.poster,
    media_path=os.path.join(MEDIA_ROOT, 'movie_poster'),
    allowed_extensions=['jpg', 'jpeg', 'png']
)

MOVIE_SCREENSHOTS_MEDIA = LocalMediaStorage(
    column=Movie.screenshots,
    media_path=os.path.join(MEDIA_ROOT, 'movie_screenshots'),
    allowed_extensions=['jpg', 'jpeg', 'png']
)

movie_config = TableConfig(
    table_class=Movie,
    media_storage=[MOVIE_POSTER_MEDIA, MOVIE_SCREENSHOTS_MEDIA],
)

APP = create_admin([movie_config])

```

Some things to be aware of:

- By specifying `allowed_extensions`, we make sure that only images can be uploaded.
- We store the files for posters and screenshots in **separate folders** - this is important when *cleaning up files*.

Source

```

class piccolo_api.media.local.LocalMediaStorage(column: t.Union[Text, Varchar, Array], media_path:
    str, executor: t.Optional[Executor] = None,
    allowed_extensions: t.Optional[t.Sequence[str]] =
        ALLOWED_EXTENSIONS, allowed_characters:
        t.Optional[t.Sequence[str]] =
        ALLOWED_CHARACTERS, file_permissions:
        t.Optional[int] = 0o600)

```

Stores media files on a local path. This is good for simple applications, where you're happy with the media files being stored on a single server.

Parameters

- **column** – The Piccolo Column which the storage is for.
- **media_path** – This is the local folder where the media files will be stored. It should be an absolute path. For example, `'/srv/piccolo-media/poster/'`.

- **executor** – An executor, which file save operations are run in, to avoid blocking the event loop. If not specified, we use a sensibly configured `ThreadPoolExecutor`.
- **allowed_extensions** – Which file extensions are allowed. If `None`, then all extensions are allowed (not recommended unless the users are trusted).
- **allowed_characters** – Which characters are allowed in the file name. By default, it's very strict. If set to `None` then all characters are allowed.
- **file_permissions** – If set to a value other than `None`, then all uploaded files are given these file permissions.

2.9.5 S3MediaStorage

This allows us to store files in a private S3 bucket. When we need to access a file, a signed URL is generated, so the file can be viewed securely.

Example

In order to associate a column with `S3MediaStorage`, we do the following:

```
import os

from piccolo.columns import Array, Varchar
from piccolo_admin.endpoints import (
    TableConfig,
    create_admin
)
from piccolo_api.media.s3 import S3MediaStorage

class Movie(Table):
    title = Varchar()
    poster = Varchar()
    screenshots = Array(base_column=Varchar())

# Note - don't store credentials in source code if possible.
# It's safer to read them from environment variables.
# A tool like `python-dotenv` can help with this.
S3_CONNECTION_KWARGS = {
    "aws_access_key_id": os.environ.get("AWS_ACCESS_KEY_ID"),
    "aws_secret_access_key": os.environ.get("AWS_SECRET_ACCESS_KEY"),
}

MOVIE_POSTER_MEDIA = S3MediaStorage(
    column=Movie.poster,
    bucket_name="bucket123"
    folder_name="movie_poster"
    connection_kwargs=S3_CONNECTION_KWARGS,
    allowed_extensions=['jpg', 'jpeg', 'png']
)
```

(continues on next page)

(continued from previous page)

```

MOVIE_SCREENSHOTS_MEDIA = S3MediaStorage(
    column=Movie.screenshots,
    bucket_name="bucket123"
    folder_name="movie_screenshots"
    connection_kwargs=S3_CONNECTION_KWARGS,
    allowed_extensions=['jpg', 'jpeg', 'png']
)

movie_config = TableConfig(
    table_class=Movie,
    media_storage=[MOVIE_POSTER_MEDIA, MOVIE_SCREENSHOTS_MEDIA],
)

APP = create_admin([movie_config])

```

Some things to be aware of:

- By specifying `allowed_extensions`, we make sure that only images can be uploaded.
- We store the files for posters and screenshots in **separate folders within the bucket** - this is important when *cleaning up files*. You could even store them in separate buckets if you prefer.

Source

```

class piccolo_api.media.s3.S3MediaStorage(column: t.Union[Text, Varchar, Array], bucket_name: str,
    folder_name: t.Optional[str] = None, connection_kwargs:
    t.Optional[t.Dict[str, t.Any]] = None, sign_urls: bool = True,
    signed_url_expiry: int = 3600, upload_metadata:
    t.Optional[t.Dict[str, t.Any]] = None, executor:
    t.Optional[Executor] = None, allowed_extensions:
    t.Optional[t.Sequence[str]] = ALLOWED_EXTENSIONS,
    allowed_characters: t.Optional[t.Sequence[str]] =
    ALLOWED_CHARACTERS)

```

Stores media files in S3 compatible storage. This is a good option when you have lots of files to store, and don't want them stored locally on a server. Many cloud providers provide S3 compatible storage, besides from Amazon Web Services.

Parameters

- **column** – The Piccolo `Column` which the storage is for.
- **bucket_name** – Which S3 bucket the files are stored in.
- **folder_name** – The files will be stored in this folder within the bucket. S3 buckets don't really have folders, but if folder is 'movie_screenshots', then we store the file at 'movie_screenshots/my-file-abc-123.jpeg', to simulate it being in a folder.
- **connection_kwargs** – These kwargs are passed directly to the boto3 client. For example:

```

S3MediaStorage(
    ...,

```

(continues on next page)

(continued from previous page)

```

connection_kwargs={
    'aws_access_key_id': 'abc123',
    'aws_secret_access_key': 'xyz789',
    'endpoint_url': 's3.cloudprovider.com',
    'region_name': 'uk'
}
)

```

- **sign_urls** – Whether to sign the URLs - by default this is True, as it's highly recommended that your store your files in a private bucket.
- **signed_url_expiry** – Files are accessed via signed URLs, which are only valid for this number of seconds.
- **upload_metadata** – You can provide additional metadata to the uploaded files. To see all available options see `S3Transfer.ALLOWED_UPLOAD_ARGS`. Below we show examples of common use cases.

To set the ACL:

```

S3MediaStorage(
    ...,
    upload_metadata={'ACL': 'my_acl'}
)

```

To set the content disposition (how the file behaves when opened - is it downloaded, or shown in the browser):

```

S3MediaStorage(
    ...,
    # Shows the file within the browser:
    upload_metadata={'ContentDisposition': 'inline'}
)

```

To attach user defined metadata to the file:

```

S3MediaStorage(
    ...,
    upload_metadata={'Metadata': {'myfield': 'abc123'}}
)

```

To specify how long browsers should cache the file for:

```

S3MediaStorage(
    ...,
    # Cache the file for 24 hours:
    upload_metadata={'CacheControl': 'max-age=86400'}
)

```

Note: We automatically add the `ContentType` field based on the file type.

- **executor** – An executor, which file save operations are run in, to avoid blocking the event loop. If not specified, we use a sensibly configured `ThreadPoolExecutor`.
- **allowed_extensions** – Which file extensions are allowed. If `None`, then all extensions are allowed (not recommended unless the users are trusted).

- **allowed_characters** – Which characters are allowed in the file name. By default, it's very strict. If set to `None` then all characters are allowed.

2.9.6 Integrating it with your wider app

If you're using Piccolo Admin as part of a larger application, you can easily gain access to the stored files, and use them within your own code.

For example:

```
# We can fetch a file from storage
file = await MOVIE_POSTER_MEDIA.get_file('some-file-key.jpeg')

# We can delete files from storage
await MOVIE_POSTER_MEDIA.delete_file('some-file-key.jpeg')
```

To see all of the methods available, look at [MediaStorage](#).

2.9.7 Cleaning up files

If we delete a row from the database which references a stored file, the file isn't automatically deleted. This is common practice, as it gives a bit more safety against accidentally deleting files.

We can periodically delete any files which are no longer referenced in the database.

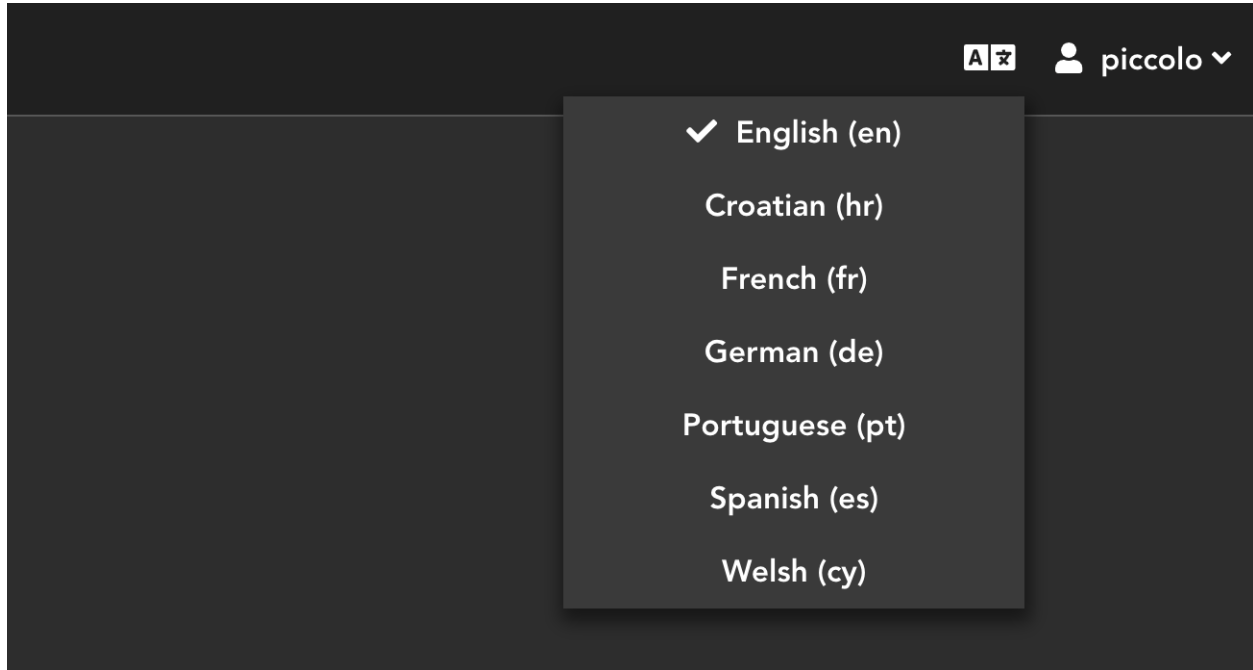
```
await MOVIE_POSTER_MEDIA.delete_unused_files()
```

Warning: It's very important that each column stores files in its own folder or S3 bucket. If multiple columns share the same folder, when we run `delete_unused_files` we may delete files needed by another column.

2.10 Internationalization

Piccolo Admin has built-in language translations for easy internationalization.

By clicking on the icon in the menu, the user can change their language, and UI will be translated accordingly.



2.10.1 Default language

By default, the user's language is detected from the web browser using the `navigator.language` JavaScript API.

You can explicitly set the default instead. For example, if we know all of our users are French speakers, we can explicitly set French as the default.

```
admin = create_admin(..., default_language_code="fr")
```

2.10.2 Available languages

To see all of the languages that we support, see `piccolo_admin/translations/data.py`.

Hint: More languages will be supported in the future. Pull requests are welcome.

You can create your own translations using the `Translation` class.

By default, all of the languages we have translations for are made available in the dropdown in the UI. However you can reduce the number of options, if you know your users only require certain languages.

```
from piccolo_admin.translations.data import ENGLISH, FRENCH

admin = create_admin(..., translations=[ENGLISH, FRENCH])
```

2.11 REST API Documentation

Piccolo Admin is powered by a rich REST API.

After logging into Piccolo Admin, you can go to `/api/docs/` to see the [Swagger docs](#), for the API.

2.12 Debugging

2.12.1 Logging

If Piccolo Admin encounters a `500` error, for example:

- The database is unavailable
- A table doesn't exist

Then Piccolo Admin will use Python's `exception logger` to log the exception. If running the app via `uvicorn`, you will then see the exception traceback in the terminal.

2.12.2 Debug mode

During development, you can run Piccolo Admin in debug mode.

```
app = create_admin(tables=[MyTable], debug=True)
```

When a `500` error is returned via the API, a stack trace is included.

This is useful if you want to inspect any `500` errors from the API in the browser.

Warning: DO NOT USE THIS IN PRODUCTION - the stack traces are for the developer's eyes only.

2.13 Help

To get support please [create an issue](#), or start a [new discussion on GitHub](#).

2.14 API Reference

2.14.1 Translations

```
class piccolo_admin.translations.models.Translation(*, language_name: str, language_code: str,
                                                    translations: Dict[str, str])
```

Used to provide translations in the UI.

Parameters

- **language_name** – A human readable representation of the language. For example 'English'.

- **language_code** – The [IETF language tag](#). For English it is 'en'. However, it also allows us to specify dialects like 'en-US' for American English, or 'en-GB' for British English.
- **translations** – A mapping of English words / phrases to their translated form. For example:

```
Translation(
    language_name='Portuguese',
    language_code='pt',
    translations={
        'Welcome to': 'Bem-vindo ao',
        ...
    }
)
```

2.14.2 MediaStorage

```
class piccolo_api.media.base.MediaStorage(column: Union[Text, Varchar, Array], allowed_extensions:
    Optional[Sequence[str]] = ALLOWED_EXTENSIONS,
    allowed_characters: Optional[Sequence[str]] =
    ALLOWED_CHARACTERS)
```

If you want to implement your own custom storage backend, create a subclass of this class. Override each abstract method.

Typically, just use `LocalMediaStorage` or `S3MediaStorage` instead.

abstract async delete_file(file_key: str)

Deletes the file object matching the file_key.

async delete_unused_files(number_shown: int = 10, auto: bool = False)

Over time, you will end up with files stored which are no longer needed. For example, if a row is deleted in the database, which referenced a stored file.

By periodically running this method, it will clean up these unused files.

It's important that each column uses its own folder for storing files. If multiple columns store data in the same folder, then we could delete some files which are needed by another column.

Parameters

- **number_shown** – This number of unused file names are printed out, so you can be sure nothing strange is going on.
- **auto** – If True, no confirmation is required before deletion takes place.

generate_file_key(file_name: str, user: Optional[BaseUser] = None) → str

Generates a unique file ID. If you have your own strategy for naming files, you can override this method.

By default we add a UUID to the filename, to make it unique:

```
>>> self.generate_file_key(file_name='my-poster.jpg')
my-poster-3beac950-7721-46c9-9e7d-5e908ef51011.jpg
```

Raises

ValueError – If the file_name is invalid.

abstract async generate_file_url(file_key: *str*, root_url: *str*, user: *Optional[BaseUser]* = None)

This retrieves an absolute URL for the file. It might be a signed URL, if using S3 for storage.

Parameters

- **file_key** – Get the URL for a file with this file_key.
- **root_url** – The URL the media is usually served from. The sub class might ignore this argument entirely, if it's fetching the data from an external source like S3.
- **user** – The Piccolo BaseUser who requested this.

abstract async get_file(file_key: *str*) → *Optional[IO]*

Returns the file object matching the file_key.

abstract async get_file_keys() → *List[str]*

Returns the file key for each file we have stored.

async get_file_keys_from_db() → *List[str]*

Returns the file key for each file we have in the database.

async get_unused_file_keys() → *List[str]*

Compares the file keys we have stored, vs what's in the database.

abstract async store_file(file_name: *str*, file: *IO*, user: *Optional[BaseUser]* = None) → *str*

Stores the file in whichever storage you're using, and returns a key which uniquely identifies the file.

Parameters

- **file_name** – The file name with which the file will be stored.
- **file** – The file to store.
- **user** – The Piccolo BaseUser who requested this.

validate_file_name(file_name: *str*)

Raises

ValueError – If the file name is invalid.

```
piccolo_api.media.base.ALLOWED_EXTENSIONS = ('mp3', 'wav', 'csv', 'tsv', 'pdf', 'gif',  
'jpeg', 'jpg', 'png', 'svg', 'tif', 'webp', 'rtf', 'txt', 'mov', 'mp4', 'webm')
```

These are the extensions which are allowed by default.

```
piccolo_api.media.base.AUDIO_EXTENSIONS = ('mp3', 'wav')
```

Pass into allowed_characters to just allow audio files.

```
piccolo_api.media.base.DATA_EXTENSIONS = ('csv', 'tsv')
```

Pass into allowed_characters to just allow data files.

```
piccolo_api.media.base.DOCUMENT_EXTENSIONS = ('pdf',)
```

Pass into allowed_characters to just allow document files.

```
piccolo_api.media.base.IMAGE_EXTENSIONS = ('gif', 'jpeg', 'jpg', 'png', 'svg', 'tif',  
'webp')
```

Pass into allowed_characters to just allow image files.

```
piccolo_api.media.base.TEXT_EXTENSIONS = ('rtf', 'txt')
```

Pass into allowed_characters to just allow text files.

```
piccolo_api.media.base.VIDEO_EXTENSIONS = ('mov', 'mp4', 'webm')
```

Pass into `allowed_characters` to just allow video files.

```
piccolo_api.media.base.ALLOWED_CHARACTERS = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A',
'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ',
'-', '_', '.', '(', ')')
```

These are the characters allowed in the file name by default.

2.15 Contributing

The backend is just vanilla Python.

The front end is built using Vue.js. To make modifications, clone the repo, and `cd` into the `admin_ui` directory.

Install the npm dependencies:

```
npm install
```

And then you can launch the admin as follows:

```
npm run dev
```

It will auto refresh the UI as you make changes to the source files.

The UI needs an API to interact with - the easiest way to do this is to use the demo app.

```
admin_demo

# Or alternatively
python -m piccolo_admin.example

# You can also populate lots of test data
python -m piccolo_admin.example --inflate=10000

# To find out all available options:
python -m piccolo_admin.example --help
```

2.15.1 Code style

Python

Piccolo uses [Black](#) for formatting, preferably with a max line length of 79, to keep it consistent with [PEP8](#).

You can configure [VSCode](#) by modifying `settings.json` as follows:

```
{
  "python.linting.enabled": true,
  "python.formatting.provider": "black",
  "python.formatting.blackArgs": [
```

(continues on next page)

(continued from previous page)

```
    "--line-length",
    "79"
  ],
  "editor.formatOnSave": true
}
```

Type hints are used throughout the project.

Front end

We use VSCode for front end development, as it has the excellent Vetur extension.

Add the following to your VSCode `settings.json`:

```
{
  // Standalone LESS and Typescript files
  "prettier.semi": false,
  "prettier.singleQuote": false,
  "prettier.tabWidth": 4,
  "prettier.trailingComma": "none",

  // Vetur - handles Vue files
  "vetur.format.enable": true,
  "vetur.format.options.tabSize": 4,
  "vetur.format.options.useTabs": false,
  "vetur.format.defaultFormatterOptions": {
    "prettier": {
      "semi": false,
      "singleQuote": false,
      "trailingComma": "none",
    },
  },
}
```

2.15.2 Storybook

The project uses [Storybook JS](#), which is a nice tool for viewing UI components in isolation. To launch it:

```
npm run storybook
```

Note: This was temporarily removed in v1, but we will try and add it back.

2.15.3 Playwright

Playwright is a tool for running end to end tests. It enables us to check that the entire application is working as expected.

Run all tests

From within the root folder, use the following command to run all of the Playwright tests with dev server in parallel:

```
./scripts/run-e2e-test.sh
```

2.15.4 Translations

The Piccolo Admin UI supports translations for several languages. If you would like to contribute a new language, it would be very appreciated.

Look in `piccolo_admin/translations/data.py`. Use one of the existing translations as a foundation.

We have a script which checks if any translations are missing, which you can use if you like:

```
python scripts/get_translations.py validate
```

2.16 Changes

2.16.1 1.3.3

Fixed a bug with array inputs in custom forms (thanks to @sinisaos for this).

2.16.2 1.3.2

Added CSP (Content Security Policy) middleware to stop malicious SVG files from executing JavaScript. This was possible if:

- Local media storage was enabled
- SVG uploads were allowed from untrusted sources
- When viewing an uploaded SVG in Piccolo Admin, if you open the SVG in a new tab then it's possible for JavaScript embedded in the SVG file to run.

It's recommended that you upgrade to this version. Thanks to @Skelmis for this.

2.16.3 1.3.1

Fixed a bug with the bulk update button not being translated.

Thanks to @jrycw for reporting the issue, and @sinisaos for the fix.

2.16.4 1.3.0

Added translations for Traditional Chinese (thanks to @jrycw for this).

2.16.5 1.2.2

Fixed a bug with filtering `Array` columns when choices are defined. Thanks to @sinisaos for discovering the solution.

2.16.6 1.2.1

Fixed a bug with `Array` columns which have choices defined. Both a `select` and `input` widget were being shown.

2.16.7 1.2.0

Added Python 3.12 support.

When filtering `Varchar` and `Text` columns, you can now specify the `match`. Previously, it always defaulted to `contains`, but now you can specify `starts`, `ends` and `exact`. For example, you can now filter for a movie with a name starting with `Star Wars`.

When filtering numeric / date / time columns, you can now specify the `not equals` operator. For example, give me all the movie tickets which aren't on a certain day.

Fixed some minor bugs, and added additional Playwright tests.

2.16.8 1.1.3

Improved CSV downloads - the user now has the option of using commas or semicolons as delimiters. In Piccolo Admin v1 we had changed to using semicolons by default, which was causing confusion.

2.16.9 1.1.2

The sidebar styles were improved (see [this issue](#) for more info).

2.16.10 1.1.1

Fixed a regression in Piccolo Admin v1, where nullable boolean fields defaulted to `null` instead of `all` in the filter sidebar. This was caused by changes in Pydantic v2, where the JSON schema changed.

2.16.11 1.1.0

Big improvements to `Timestamptz` columns:

- Piccolo Admin now displays the timezone in the UI.
 - The resolution of the widget can be specified using `TableConfig.time_resolution`, so you can decide if the user can pick seconds / milliseconds.
-

2.16.12 1.0.0

Updated to work with Piccolo v1 (which uses Pydantic v2).

The front end code has also been substantially upgraded to Vue v3.

2.16.13 0.58.0

The default rate limiting is now more aggressive. This can be overridden using the `rate_limit_provider` argument of `create_admin`.

2.16.14 0.57.0

Improved the handling of nullable `Varchar` / `Text` columns in the UI.

2.16.15 0.56.0

Improved the handling of nullable `JSON` / `JSONB` columns in the UI.

2.16.16 0.55.0

When deleting a row, if a problem is encountered then an error message is now shown in the UI.

This is useful if we have constraints on the table (for example `ON DELETE RESTRICT`).

Support for Python 3.7 was dropped as it's now end of life.

2.16.17 0.54.0

Improved the behaviour of the *referencing tables* links on the detail page.

2.16.18 0.53.0

Improved the UI for JSON fields (the cursor would sometimes jump to the bottom).

2.16.19 0.52.0

Version pinning Pydantic to v1, as v2 has breaking changes.

We will add support for Pydantic v2 in a future release.

Thanks to @sinisaos for helping with this.

2.16.20 0.51.0

Improved the UI for password inputs (e.g. on the change password page). Thanks to @sinisaos for this.

Fixed a bug with nullable date fields.

2.16.21 0.50.0

Improved handling of nullable email fields.

Thanks to @sinisaos for adding this.

2.16.22 0.49.0

Custom links can now be added to the sidebar. This allows quick navigation to specific pages in the admin, or to external websites. For example:

```
create_admin(
  sidebar_links={
    "Top Movies": "/admin/#/movies/?__order=-popularity",
  }
  ...
)
```

Thanks to @sinisaos for adding this.

2.16.23 0.48.0

- Improved the type annotations for `FormConfig`.
- Fixed a bug with array fields in custom forms (thanks to @sinisaos for fixing this).

2.16.24 0.47.0

Multiple columns can now be used for sorting the rows in the UI.

Setting the default order for a table is now possible. For example, if we want to order movies by rating:

```
create_admin(
  tables=[
    TableConfig(
      Movie,
      order_by=[
        OrderBy(Movie.rating, ascending=False)
      ]
    )
  ]
)
```

Thanks to @sinisaos and @sumitsharansatsangi for their help with this.

2.16.25 0.46.0

Added Turkish translations (thanks to @omerucel for this).

2.16.26 0.45.2

Nullable UUID fields now work correctly.

2.16.27 0.45.1

Add back JSON formatting in list view which was removed by accident.

2.16.28 0.45.0

Nullable number fields now work correctly.

2.16.29 0.44.0

Fixed a bug with nullable Boolean columns - filtering wasn't working in the sidebar.

2.16.30 0.43.0

Added the `link_column` option to `TableConfig`. By default, the primary key is used in the list view of Piccolo Admin to link to the edit page. Using `link_column` you can specify a different column (for example, if you hid the primary key using `visible_columns`). Thanks to @sinisaos for helping with this.

2.16.31 0.42.0

Tables can now be grouped in the sidebar - this is helpful if you have lots of tables. To do this, use the `menu_group` argument of `TableConfig`.

Thanks to @sinisaos and @sumitsharansatsangi for their help with this.

2.16.32 0.41.0

A fix to make Piccolo Admin work with `fastapi>=0.89.0`.

2.16.33 0.40.0

- Improved German translations (thanks to @hblunck for this).
 - When submitting a form, scroll to the top of the page if an error occurs so the error box is visible (thanks to @sinisaos for this).
 - If a custom BaseUser table is used for authentication, which uses a UUID as the primary key, it now works.
-

2.16.34 0.39.0

If an Array column has choices specified, then Piccolo Admin will show dropdowns, so the user can pick one of the choices.

2.16.35 0.38.0

Fixed a bug with TableConfig and exclude_visible_columns. Thanks to @web-maker for this fix.

2.16.36 0.37.0

- Python 3.11 is now officially supported.
 - Added debug mode: `create_admin(tables=[MyTable], debug=True)`.
 - Logging exceptions for 500 errors.
 - Fixed a typo in the docs about how to use validators (thanks to @sinisaos for reporting this).
 - Updated the tests for Starlette / FastAPI's new test client. This means that `fastapi==0.87.0` / `starlette==0.21.0` are now the minimum versions supported. Thanks to @sinisaos for this.
-

2.16.37 0.36.0

Lots of small enhancements.

- Fixed bugs with the foreign key selector. Sometimes the edit button didn't work. Also, sometimes the value shown in the input box wasn't refreshing when navigating to a new page.
 - The HTML title now matches the `site_name` parameter in `create_admin` (thanks to @sinisaos for this).
 - Updated Vue to the latest version.
 - Internal code refactoring.
-

2.16.38 0.35.0

Validators can now be specified in TableConfig.

This allows fine grained access control - for example, only allowing some users to send POST requests to certain API endpoints:

```
from piccolo_api.crud.endpoints import PiccoloCRUD
from starlette.exceptions import HTTPException
from starlette.requests import Request

async def manager_only(
    piccolo_crud: PiccoloCRUD,
    request: Request
):
    # The Piccolo `BaseUser` can be accessed from the request.
    user = request.user.user

    # Assuming we have another database table where we record
    # users with certain permissions.
    manager = await Manager.exists().where(manager.user == user)

    if not manager:
        # Raise a Starlette exception if we want to reject the
        # request.
        raise HTTPException(
            status_code=403,
            detail="Only managers are allowed to do this"
        )

admin = create_admin(
    tables=TableConfig(
        Movie,
        validators=Validators(post_single=[manager_only])
    )
)
```

2.16.39 0.34.0

Updated the date / datetime / time picker.

2.16.40 0.33.1

Fixed an issue with installing `piccolo_admin` in editable mode with `pip`.

Thanks to @peterschutt for reporting this issue.

2.16.41 0.33.0

Improved the UI for error messages. Thanks to @sinisaos for adding this.

2.16.42 0.32.0

Camelcase column names could break parts of Piccolo Admin. It now works as expected:

```
class Person(Table):  
    # This now works:  
    firstName = Varchar()
```

Even though camelcase is unusual in Python, a user may be using an existing database, so it makes sense to support it. Thanks to @sumitsharansatsangi for reporting this issue.

2.16.43 0.31.2

When `piccolo_admin` is installed, an `admin_demo` script is made available on the command line, which launches a Piccolo Admin demo.

It wasn't working due to a missing folder, which has now been fixed.

2.16.44 0.31.1

Fixed a bug with custom forms - under some situations they would fail to render. Thanks to @sinisaos for discovering this issue. See [PR 208](#) for more info.

2.16.45 0.31.0

Improved the French translations (courtesy @LeMeteore).

2.16.46 0.30.0

Added translations for simplified Chinese characters (courtesy @mnixry).

2.16.47 0.29.1

The media endpoints now obey the `read_only` option of `create_admin`. Read only mode is used for online demos. Thanks to @sinisaos for adding this.

2.16.48 0.29.0

Added media upload support - to both a local folder, and S3. Images, videos, PDFs, and audio files can be viewed within the UI. This is the one of the biggest updates we've ever made! Thanks to @sinisaos for all of the help.

2.16.49 0.28.0

Added Ukrainian translations (courtesy @ruslan-rv-ua).

2.16.50 0.27.0

Added Russian translations (courtesy @northpowered).

2.16.51 0.26.1

Modified the release process, so it works on GitHub (courtesy @olliglorioso).

2.16.52 0.26.0

Added Finnish translations (courtesy @olliglorioso).

2.16.53 0.25.0

Added translations, to make the UI more accessible in a variety of languages (thanks to @sinisaos for helping with this).

2.16.54 0.24.0

TableConfig now has a `hooks` argument - so custom logic can be run when a row is added / deleted / modified. Thanks to @Anton-Karpenko for suggesting this feature.

2.16.55 0.23.0

The WYSIWYG editor we use for `rich_text_columns` has been modified, so the user can now create HTML headings. Thanks to @tigerline86 for suggesting this feature and @sinisaos for implementing it.

Rows can now be bulk modified - for example, if you have 100 blog posts which need converting to `draft=False`, it can now be easily done using the Piccolo Admin GUI in a single operation (courtesy @sinisaos).

2.16.56 0.22.2

More sandbox fixes.

2.16.57 0.22.1

Fixed a bug with the sandbox.

2.16.58 0.22.0

The user can now change their password in the Piccolo Admin UI (courtesy @sinisaos).

After submitting a custom form with Piccolo Admin, the UI used to show the response message in a popup at the bottom of the screen. It now shows a success page instead, which is better if the response message is long, as it's easier for the user to read. Thanks to @ethagnawl for reporting this issue.

2.16.59 0.21.0

Added a warning if a Piccolo Table column is both `secret=True` and `required=True`, as it's unsupported by Piccolo admin (courtesy @ethagnawl).

2.16.60 0.20.0

You can now use a rich text editor for Text columns (courtesy @sinisaos).

```
from piccolo_admin.endpoints import TableConfig

from movies.tables import Movie

movie_config = TableConfig(
    Movie,
    rich_text_columns=[
        Movie.description
    ]
)

create_admin(movie_config)
```

This is useful when using Piccolo Admin for authoring content in blogs etc.

2.16.61 0.19.6

Fixes for Table classes which have custom primary key columns.

2.16.62 0.19.5

More z-index refinements (thanks @sinisaos).

2.16.63 0.19.4

Fixed a bug with the z-index of the sidebar on mobile. Thanks to @sinisaos for discovering this issue.

2.16.64 0.19.3

Improved the UI when the network is slow (courtesy @sinisaos).

With `FormConfig`, if the Pydantic model has a default value provided, this is rendered in the form UI (thanks to @simplynail for this idea).

2.16.65 0.19.2

The `textarea` and `button` elements were using the browser's default font, instead of our custom font.

Improved the docstring for `create_admin`.

2.16.66 0.19.1

Fixed a bug where a filter for a column with choices defined would default to `Null` instead of `All`.

2.16.67 0.19.0

Added new UI for the foreign key selector.

2.16.68 0.18.2

Fixed a bug where resetting the filters in the sidebar would set them to `less than`. Now they reset to `equals`. Courtesy @sinisaos.

2.16.69 0.18.1

Fixed a bug where a filter for a column with choices would default to `'Null'` instead of `'All'`.

2.16.70 0.18.0

Added a `visible_filters` option to `TableConfig`, allowing the user to specify which filters are shown in the filter sidebar. This is useful if you have a lot of columns. Courtesy @sinisaos.

Improved the navigation sidebar UI - each section can now be hidden, and the appearance has been improved when table names are very long. Courtesy @sinisaos.

Added docs for Javascript formatting to help new contributors.

2.16.71 0.17.0

Added `TableConfig`, which allows more fine grained control over how the UI behaves for a given `Table`. Currently it allows you to specify which columns are visible on the list page, but more options will be added in the future. Courtesy @sinisaos.

2.16.72 0.16.1

Fixed bugs with nullable `ForeignKey` and `Timestamp` columns - the UI would try sending back an empty string, instead of a null value. Courtesy @sinisaos.

2.16.73 0.16.0

JSON values are now displayed in a nicer format in the UI (courtesy @sinisaos).

The popup banner displayed at the bottom of the page will now turn red when showing an error (it was already green in the past). Courtesy @sinisaos.

2.16.74 0.15.2

`FormConfig.endpoint` now works with async functions.

2.16.75 0.15.1

Fixing a bug where setting `FormConfig.description` to `None` caused a serialisation error.

2.16.76 0.15.0

Added custom forms (courtesy @sinisaos).

It's very easy to use - just provide a Pydantic model, and a function for handling posted data. Piccolo Admin will then auto generate all of the UI necessary.

2.16.77 0.14.0

Using the `swagger_ui` endpoint from Piccolo API for the Swagger docs, so it works with the CSRF middleware.

2.16.78 0.13.2

Rewrote *admin_demo* command to expose configuration options on the command line.

2.16.79 0.13.1

- Bumped Node dependencies with security warnings.
 - Slightly changed light mode styles (blue-grey sidebar instead of grey).
 - Fixed the *admin_demo* command which is installed by *setup.py* - the path was wrong.
-

2.16.80 0.13.0

Modified the UI to support columns with a *choices* attribute set. A select input element is shown.

2.16.81 0.12.1

Fixed issue with *BigInt* values being displayed incorrectly.

2.16.82 0.12.0

Added support for *Array* column type.

2.16.83 0.11.13

Exposing the site name on the login page, courtesy of *sinisaos*.

2.16.84 0.11.12

Added tooltips using the *help_text* attribute on *Table*.

2.16.85 0.11.11

Added tooltips using the `help_text` attribute on `Column`.

2.16.86 0.11.10

- The foreign key selector in the add and edit row forms now use the search based UI, courtesy of sinisaos.
 - Fixing a Vue JS warning about a route parameter being undefined.
-

2.16.87 0.11.9

- Exposed the `host` and `port` options directly in the sandbox CLI.
 - Fixing a bug with read only mode. Was raising a 500 with disallowed HTTPS methods
-

2.16.88 0.11.8

- The foreign key selector in the sidebar is now search based, rather than a select element, courtesy of sinisaos. This makes the admin work better with very large data sets.
 - Fixed a bug with nullable foreign keys. The value can now be set to null without a validation error.
-

2.16.89 0.11.7

Added an `--inflate` option to the CLI in `example.py`. This allows lots of dummy data to be added during development.

2.16.90 0.11.6

Fixing a bug with the date time picker on mobile devices - thanks sinisaos!

2.16.91 0.11.5

Fixing a bug where clearing the filters wasn't clearing the duration widget's value, as it uses a hidden input - thanks sinisaos!

2.16.92 0.11.4

Added missing trailing slash to table detail endpoints.

2.16.93 0.11.3

Fixing auth API URL - thanks sinisaos!

2.16.94 0.11.2

requirements.txt fixes

2.16.95 0.11.1

Updated Node dependencies, and fixed requirements clash with FastAPI and Starlette.

2.16.96 0.11.0

- Refactored `AdminRouter` to use `FastAPI`. This means the API is fully documented - courtesy of sinisaos.
 - Moved auth endpoints from `/api/` to `/auth/`, to separate auth from the main API.
-

2.16.97 0.10.9

Fixing a bug with fetching meta information from the API (Piccolo version, site name etc). When a user isn't logged in, it would fail. It now calls the API again after a successful login - courtesy of sinisaos.

2.16.98 0.10.8

- Can override the nav bar title (defaults to *Piccolo Admin*) - courtesy of sinisaos.
 - Other nav bar improvements, such as truncating long usernames.
-

2.16.99 0.10.7

- Added page size selector - courtesy of sinisaos.
 - Minor fixes
-

2.16.100 0.10.6

Added bulk deletion, and a custom widget for *timedelta* - courtesy of sinisaos.

2.16.101 0.10.5

Added a CSV export button to the row listing - courtesy of sinisaos.

2.16.102 0.10.4

- Removed dependency number for `uvicorn` and `Hypercorn` - only the very high level API is being used, which is unlikely to change, and was causing issues for some users when installing via Poetry.
 - Bumped node dependencies.
-

2.16.103 0.10.3

Fixing packaging issues - add Python 3.8 classifier, and missing `index.html` file.

2.16.104 0.10.2

Subtle UI fixes - page selector, and `setTimeout` typo.

2.16.105 0.10.1

Added `allowed_hosts` argument to `create_admin` - otherwise CSRF middleware will block requests when running under HTTPS.

2.16.106 0.10.0

Using latest piccolo, and piccolo_api.

2.16.107 0.9.2

- Improved pagination when there's lots of data.
 - Bumped node dependencies.
-

2.16.108 0.9.1

Bumped node requirements because of security warning.

2.16.109 0.9.0

Bumped node and pip requirements.

2.16.110 0.8.1

Bumped node and pip requirements.

2.16.111 0.8.0

Added support for Numeric and Real column types in Piccolo.

2.16.112 0.7.0

Exposing more configuration options for session auth.

2.16.113 0.6.6

Disabling redirect on session auth.

2.16.114 0.6.5

Loosening requirements for Piccolo projects.

2.16.115 0.6.4

Bumped requirements.

2.16.116 0.6.3

Bumped requirements and added apps to piccolo_app migration dependencies.

2.16.117 0.6.2

Converted into a Piccolo app.

2.16.118 0.6.1

Bumped requirements.

2.16.119 0.6.0

Supporting piccolo 0.10.0.

2.16.120 0.5.1

Updated requirements.

2.16.121 0.5.0

Updated dependencies, and vendored remaining Javascript.

2.16.122 0.4.1

Using rate limit middleware on login endpoint. Auto including related tables. Using PATCH instead of PUT when editing a row. UI improvements.

2.16.123 0.4.0

Using textarea for Text database fields, using new API schema format, and various UI improvements.

2.16.124 0.3.8

Updated piccolo_api requirements.

2.16.125 0.3.7

UI improvements, and catching 404 errors.

2.16.126 0.3.6

Added 'about' modal to UI.

2.16.127 0.3.5

Updated sandbox - populates data.

2.16.128 0.3.4

Added sandbox, for deploying demo version online.

2.16.129 0.3.3

UI improvements, including light mode. Support for pagination, and operators in filters.

2.16.130 0.3.2

Fixed typo - missing trailing slash.

2.16.131 0.3.1

Improved auth error handling, and adding defaults automatically when adding a new row.

2.16.132 0.3.0

Login is working, and various UI improvements.

2.16.133 0.2.0

Updated to work with Piccolo API code layout changes.

2.16.134 0.1.4

Making edit row work.

2.16.135 0.1.3

Added missing assets.

2.16.136 0.1.2

Added missing assets.

2.16.137 0.1.1

Fixing filters.

2.16.138 0.1.0

Initial release.

INDEX

A

ALLOWED_CHARACTERS (in module piccolo_api.media.base), 41
ALLOWED_EXTENSIONS (in module piccolo_api.media.base), 40
AUDIO_EXTENSIONS (in module piccolo_api.media.base), 40

C

create_admin (class in piccolo_admin.endpoints), 7

D

DATA_EXTENSIONS (in module piccolo_api.media.base), 40
delete_file() (piccolo_api.media.base.MediaStorage method), 39
delete_unused_files() (piccolo_api.media.base.MediaStorage method), 39
DOCUMENT_EXTENSIONS (in module piccolo_api.media.base), 40

F

FormConfig (class in piccolo_admin.endpoints), 23

G

generate_file_key() (piccolo_api.media.base.MediaStorage method), 39
generate_file_url() (piccolo_api.media.base.MediaStorage method), 39
get_file() (piccolo_api.media.base.MediaStorage method), 40
get_file_keys() (piccolo_api.media.base.MediaStorage method), 40
get_file_keys_from_db() (piccolo_api.media.base.MediaStorage method), 40

get_unused_file_keys() (piccolo_api.media.base.MediaStorage method), 40

I

IMAGE_EXTENSIONS (in module piccolo_api.media.base), 40

L

LocalMediaStorage (class in piccolo_api.media.local), 32

M

MediaStorage (class in piccolo_api.media.base), 39

S

S3MediaStorage (class in piccolo_api.media.s3), 34
store_file() (piccolo_api.media.base.MediaStorage method), 40

T

TableConfig (class in piccolo_admin.endpoints), 19
TEXT_EXTENSIONS (in module piccolo_api.media.base), 40
Translation (class in piccolo_admin.translations.models), 38

V

validate_file_name() (piccolo_api.media.base.MediaStorage method), 40
VIDEO_EXTENSIONS (in module piccolo_api.media.base), 40