Piccolo API

Release 1.11.0

Daniel Townsend

Aug 07, 2026

CRUD

Piccolo API makes it easy to turn your Piccolo ORM tables into a working REST API, using ASGI.

It also includes a bunch of essential middleware for building a production ASGI app, covering authentication, security, and more.

CHAPTER

ONE

CRUD

The following make it easy to create an ASGI web application which turns your Piccolo tables into a powerful REST API.

1.1 PiccoloCRUD

This creates an ASGI app, which exposes the usual CRUD methods on your Piccolo table, as well as some extras, via a REST API.

1.1.1 Endpoints

Path	Methods	Description
/	GET, POST, DELETE	Get all rows, post a new row, or delete all matching rows.
/<id>/	GET, PUT, DELETE, PATCH	Get, update or delete a single row.
/schema	GET	Returns a JSON schema for the table. This allows clients to auto generate forms.
/ids/	GET	Returns a mapping of all row ids to a description of the row.
/count/	GET	Returns the number of matching rows.
/new/	GET	Returns all of the default values for a new row - can be used to dynamically generate forms by the client.

1.1.2 Creating an ASGI app

Using it is as simple as this:

```
# app.py
from piccolo_api.crud.endpoints import PiccoloCRUD

from movies.tables import Movie


# If we just want to expose the GET endpoints:
app = PiccoloCRUD(table=Movie)

# This will expose all the endpoints:
app = PiccoloCRUD(table=Movie, read_only=False)
```

To expose several CRUD endpoints in our app, we use Starlette's Router.

```
# app.py
from piccolo_api.crud.endpoints import PiccoloCRUD
from starlette.routing import Mount, Router

from movies.tables import Movie, Director


app = Router([
    Mount(
        path='/movie',
        app=PiccoloCRUD(table=Movie),
    ),
    Mount(
        path='/director',
        app=PiccoloCRUD(table=Director),
    ),
])
```

You can then run it using an ASGI server such as Uvicorn:

```
uvicorn app:app
```

1.1.3 Filters

Example schema

`PiccoloCRUD` makes filtering your data really easy using query parameters in your HTTP request. Using the following Piccolo schema as an example:

```
# tables.py
from piccolo.table import Table
from piccolo.columns import (
    Varchar,
    Integer,
    ForeignKey,
    Boolean,
    Text,
    Timestamp,
    Numeric,
    Real
)


class Director(Table):
    name = Varchar(length=300, null=False)


class Movie(Table):
    name = Varchar(length=300)
    rating = Real()
```

(continues on next page)

(continued from previous page)

```
    duration = Integer()
    director = ForeignKey(references=Director)
    won_oscar = Boolean()
    description = Text()
    release_date = Timestamp()
    box_office = Numeric(digits=(5, 1))
```

Basic queries

Get all movies with 'star wars' in the name:

```
GET /movie/?name=star%20wars
```

> **Hint**
>
> You can try these queries for yourself, but first login at https://demo1.piccolo-orm.com/ using username: `piccolo`, password: `piccolo123`. Then prefix `https://demo1.piccolo-orm.com/api/tables` to all your queries.

Operators

As shown above you can specify which operator to use. For numeric, and date / time fields the following operators are allowed:

- `lt`: Less Than
- `lte`: Less Than or Equal
- `gt`: Greater Than
- `gte`: Greater Than or Equal
- `e`: Equal (default)
- `ne`: Not Equal

To specify which operator to use, pass a query parameter like `field__operator=operator_name`. For example `duration__operator=gte`.

Here's a query which fetches all movies lasting more than 200 minutes:

```
GET /movie/?duration=200&duration__operator=gte
```

`is_null` / `not_null`

All field types also support the `is_null` and `not_null` operators.

For example:

```
# Get all rows with a null duration
GET /movie/duration__operator=is_null

# Get all rows without a null duration
GET /movie/duration__operator=not_null
```

Match type

When querying text fields (like `Varchar` and `Text`), you can specify the kind of match you're looking for.

- contains
- exact
- starts
- ends

To specify which match type to use, pass a query parameter like `field__match=match_type`. For example `name__match=starts`.

A query which fetches all movies whose name begins with 'star wars':

```
GET /movie/?name=star%20wars&name__match=starts
```

Order

To specify which field to sort by, pass a query parameter like `__order=field`. For example `__order=name`.

A query which fetches all movies, sorted by duration:

```
GET /movie/?__order=duration
```

You can reverse the sort by prepending `'-'` to the field. For example:

```
GET /movie/?__order=-duration
```

Multiple columns can be used for the sort. Just separate them with a comma:

```
GET /movie/?__order=-duration,name
```

Visible fields

You can request a subset of columns from the `GET` endpoint. It means we're not overfetching data when we're only interested in some of it.

For example `__visible_fields=id,name` will only fetch the values for `id` and `name` from the `Movie` table.

```
GET /movie/?__visible_fields=id,name
```

```
{
    "rows": [
        {
            "id": 17,
            "name": "The Hobbit: The Battle of the Five Armies"
        },
        ...
    ]
}
```

It can even work with joins. However, you need to enable this by setting the `max_joins` parameter of `PiccoloCRUD`. Notice how we pass in `director.name`:

```
GET /movie/?__visible_fields=id,name,director.name
```

```
{
    "rows": [
        {
            "id": 17,
            "name": "The Hobbit: The Battle of the Five Armies",
            "director": {
                "name": "Peter Jackson"
            }
        },
        ...
    ]
}
```

Pagination

You can specify how many results to return, and which page to return, using the `__page` and `__page_size` query parameters.

For example, to return results 11 to 20:

```
GET /movie/?__page=2&__page_size=10
```

1.1.4 Readable

As foreign keys are just integers in Piccolo, they aren't very descriptive about what is being pointed to. To get around this, each `Table` subclass can specify a 'readable' representation, which is more descriptive, and readable for humans. See the Piccolo docs for more details.

If you'd like to retrieve the readable representations for each foreign key in the queried table, you can do so by appending the *__readable=true* parameter to your GET requests.

```
GET /movie/?__readable=true
```

Which returns something like this:

```
{
    "rows": [
        {
            "id": 17,
            "name": "The Hobbit: The Battle of the Five Armies",
            "rating": 59,
            "duration": 164,
            "director": 1,
            "director_readable": "Peter Jackson",  // <- Note
            "won_oscar": false,
            "description": "Bilbo fights against a number of enemies to save the life of␣
↪his Dwarf friends and protects the Lonely Mountain after a conflict arises.",
            "release_date": "2014-12-01T00:00:00",
            "box_office": 956
        },
        ...
    ]
}
```

You can also use this on GET requests when retrieving a single row, for example:

```
GET /movie/1/?__readable=true
```

1.1.5 Content-Range header

In some applications it can be practical to get information about the total number of records without invoking a separate call to the `count` endpoint. Piccolo API will supply this information in the `Content-Range` response header if the `__range_header` GET parameter is set to `true`.

You can use the `__range_header_name` GET parameter to configure the "plural name" used in the `Content-Range` response header.

The contents of the `Content-Range` header might look something like this for the "Movie" table: `movie 0-9/100`.

Example usage:

```
GET /movie/?__page=2&__page_size=10&__range_header=true
```

1.1.6 Source

PiccoloCRUD

class piccolo_api.crud.endpoints.**PiccoloCRUD**(*table: type[Table]*, *read_only: bool = True*, *allow_bulk_delete: bool = False*, *page_size: int = 15*, *exclude_secrets: bool = True*, *validators: Validators | None = None*, *schema_extra: dict[str, Any] | None = None*, *max_joins: int = 0*, *hooks: list[Hook] | None = None*)

Wraps a Piccolo table with CRUD methods for use in a REST API.

Parameters

- **table** – The Piccolo `Table` to expose CRUD methods for.
- **read_only** – If `True`, only the GET method is allowed.
- **allow_bulk_delete** – If `True`, allows a delete request to the root to delete all matching records. It is dangerous, so is disabled by default.
- **page_size** – The number of results shown on each page by default.
- **exclude_secrets** – Any Piccolo columns with `secret=True` will be omitted from the response.
- **validators** – Used to provide extra validation on certain endpoints - can be easier than subclassing.
- **schema_extra** – Additional information included in the Pydantic schema.
- **max_joins** – Determines whether a query can request data from related tables using joins. For example `/movie/?__visible_fields=name,director.name`, which would return:

```
{
    'rows': [
        {
            'name': 'Star Wars',
            'director': {
                'name': 'George Lucas'
            }
        }
    ]
}
```

This is a very powerful feature, but before enabling it, bear in mind the following:

- If set too high, it could be used maliciously to craft slow queries which contain lots of joins, which could slow down your site.
- Don't enable it if sensitive data is contained in related tables, as this feature can be used to retrieve that data.

It's best used when the data in related tables is not of a sensitive nature and the client is highly trusted. Consider using it with `exclude_secrets=True`.

To see which fields can be filtered in this way, you can check the `visible_fields_options` value returned by the `/schema` endpoint.

Validators

class piccolo_api.crud.endpoints.**Validators**(*every: list[ValidatorFunction] = []*, *get_single: list[ValidatorFunction] = []*, *put_single: list[ValidatorFunction] = []*, *patch_single: list[ValidatorFunction] = []*, *delete_single: list[ValidatorFunction] = []*, *post_single: list[ValidatorFunction] = []*, *get_all: list[ValidatorFunction] = []*, *delete_all: list[ValidatorFunction] = []*, *get_references: list[ValidatorFunction] = []*, *get_ids: list[ValidatorFunction] = []*, *get_new: list[ValidatorFunction] = []*, *get_schema: list[ValidatorFunction] = []*, *get_count: list[ValidatorFunction] = []*, *extra_context: dict[str, Any] = {}*)

These validators are run by the corresponding method on *PiccoloCRUD*.

The validator function is given the `PiccoloCRUD` instance, and the Starlette `Request` instance, and should raise a Starlette `HTTPException` if there is a problem.

Async functions are also supported. Here are some examples:

```
def validator_1(piccolo_crud: PiccoloCRUD, request: Request):
    if not request.user.user.superuser:
        raise HTTPException(
            status_code=403,
            "Only a superuser can do this"
        )

async def validator_2(piccolo_crud: PiccoloCRUD, request: Request):
```

(continues on next page)

(continued from previous page)

```
    if not await my_check_user_function(request.user.user):
        raise HTTPException(
            status_code=403,
            "The user can't do this."
        )
```

1.2 Serializers

PiccoloCRUD uses `create_pydantic_model` internally to serialize and deserialize data.

`create_pydantic_model` is very useful when integrating Piccolo with a framework such as FastAPI, which also uses Pydantic for serialisation. It automatically creates Pydantic models for you.

1.3 Hooks

Hooks allow executing custom code as part of processing a CRUD request. You can use this to validate data, call another custom API, place messages on queues and many other things.

1.3.1 Enabling a hook

Define a method, and register it with *PiccoloCRUD*:

```
# app.py
from piccolo_api.crud.endpoints import PiccoloCRUD

from movies.tables import Movie


# set movie rating to 10 before saving
async def set_movie_rating_10(row: Movie):
    row.rating = 10
    return row


# set movie rating to 20 before saving
async def set_movie_rating_20(row: Movie):
    row.rating = 20
    return row


async def pre_delete(row_id):
    pass


# Register one or multiple hooks
app = PiccoloCRUD(
    table=Movie,
    read_only=False,
    hooks=[
```

(continues on next page)

(continued from previous page)

```
        Hook(hook_type=HookType.pre_save, callable=set_movie_rating_10),
        Hook(hook_type=HookType.pre_save, callable=set_movie_rating_20),
        Hook(hook_type=HookType.pre_delete, callable=pre_delete)
    ]
)
```

You can specify multiple hooks (also per `hook_type`). Hooks are executed in order. You can use either async or regular functions.

1.3.2 Hook types

There are different hook types, and each type takes a slightly different set of inputs.

It's also important to return the expected data from your hook.

pre_save

This hook runs during POST requests, prior to inserting data into the database. It takes a single parameter, `row`, and should return the row:

```
async def set_movie_rating_10(row: Movie):
    row.rating = 10
    return row


app = PiccoloCRUD(table=Movie, read_only=False, hooks=[
    Hook(hook_type=HookType.pre_save, callable=set_movie_rating_10)
    ]
)
```

pre_patch

This hook runs during PATCH requests, prior to changing the specified row in the database.

It takes two parameters, `row_id` which is the id of the row to be changed, and `values` which is a dictionary of incoming values.

Each function must return a dictionary which represent the data to be modified.

```
async def reset_name(row_id: int, values: dict):
    current_db_row = await Movie.objects().get(Movie.id==row_id)
    if values.get("name"):
        values["name"] = values["name"].replace(" ", "")
    return values


app = PiccoloCRUD(
    table=Movie,
    read_only=False,
    hooks=[
        Hook(hook_type=HookType.pre_patch, callable=reset_name)
    ]
)
```

pre_delete

This hook runs during DELETE requests, prior to deleting the specified row in the database.

It takes one parameter, `row_id` which is the id of the row to be deleted.

`pre_delete` hooks should not return data.

```
async def pre_delete(row_id: int):
    pass


app = PiccoloCRUD(
    table=Movie,
    read_only=False,
    hooks=[
        Hook(hook_type=HookType.pre_delete, callable=pre_delete)
    ]
)
```

Dependency injection

Each hook can optionally receive the `Starlette` request object. Just add `request` as an argument in your hook, and it'll be injected automatically.

```
async def set_movie_rating_10(row: Movie, request: Request):
    ...
```

1.3.3 Source

HookType

class piccolo_api.crud.hooks.**HookType**(*value*)

When the hook should be applied.

pre_delete = 'pre_delete'

pre_patch = 'pre_patch'

pre_save = 'pre_save'

Hook

class piccolo_api.crud.hooks.**Hook**(*hook_type:* HookType, *callable:* *Callable*)

CHAPTER

TWO

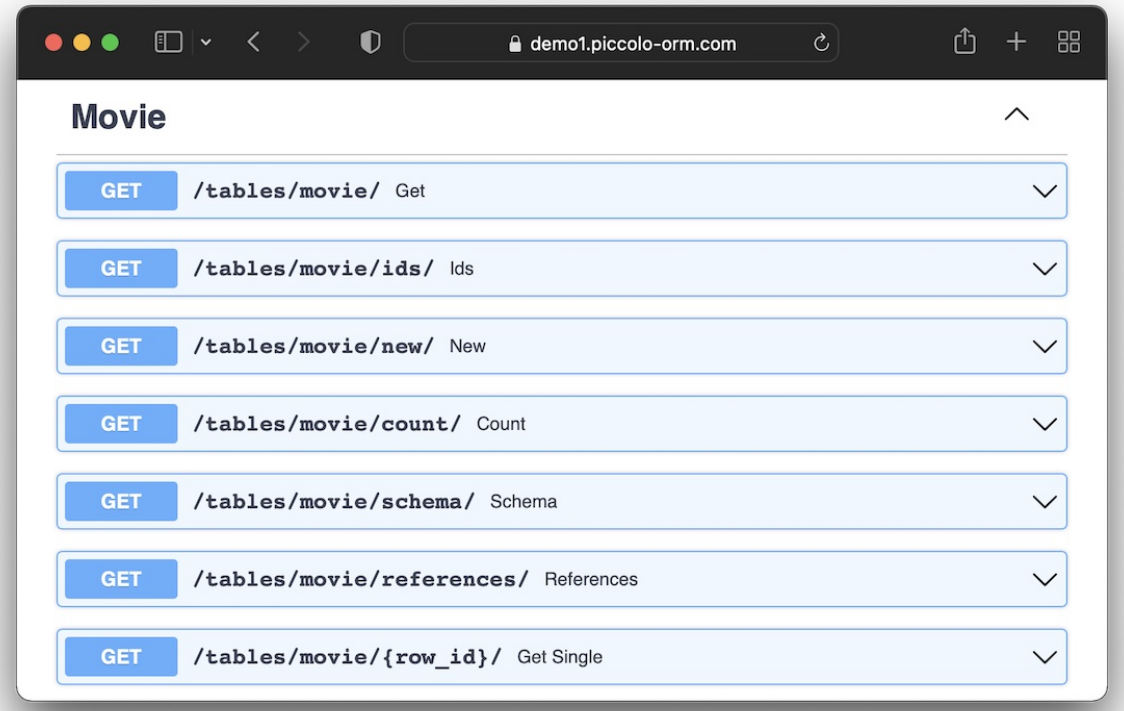
FASTAPI

FastAPI is a powerful ASGI web framework, built on top of Starlette, which lets you build an API very easily, with interactive docs.

It does this by making heavy use of type annotations.

By using `FastAPIWrapper` we can annotate our `PiccoloCRUD` endpoints so FastAPI can automatically document them for us. It's an incredibly productive way of building an API.

We're able to create all of these endpoints, and more, with very little code:



2.1 Example

In this example we're building the API for a movie app. Assuming you have defined a Piccolo `Table` called `Movie`:

```
# app.py

from fastapi import FastAPI
from piccolo_api.fastapi.endpoints import FastAPIWrapper
from piccolo_api.crud.endpoints import PiccoloCRUD

from movies.tables import Movie


app = FastAPI()


FastAPIWrapper(
    root_url="/movie/",
    fastapi_app=app,
    piccolo_crud=PiccoloCRUD(
        table=Movie,
        read_only=False,
    )
)
```

We can now run this app using an ASGI server such as uvicorn.

```
uvicorn app:app
```

Then try out the following:

- OpenAPI docs: http://127.0.0.1:8000/docs/
- API endpoint: http://127.0.0.1:8000/movie/

To see full examples of FastAPI apps built this way, take a look at:

- PyMDb - a movie database.
- A headless blog.

2.2 Configuring the endpoints

If you want more control over the endpoints, you can do so using *`FastAPIKwargs`*, which allows you to specify additional arguments to pass into `FastAPIApp.add_api_route`.

2.2.1 Example

In the following example, we add tags to the endpoints, which separates them in the docs UI.

We also mark one of the endpoints as deprecated.

```
from fastapi import FastAPI
from piccolo_api.fastapi.endpoints import FastAPIWrapper, FastAPIKwargs
```

(continues on next page)

(continued from previous page)

```
from piccolo_api.crud.endpoints import PiccoloCRUD

from my_app.tables import Movie


app = FastAPI()


FastAPIWrapper(
    root_url="/movie/",
    fastapi_app=app,
    piccolo_crud=PiccoloCRUD(
        table=Movie,
        read_only=False,
    ),
    fastapi_kwargs=FastAPIKwargs(
        all_routes={"tags": ["Movie"]},  # Added to all endpoints
        get={"deprecated": True},  # Just added to the 'get' endpoint
    )
)
```

2.3 Authentication

You can wrap your FastAPI app with any of our authentication middleware, such as *SessionAuthMiddleware*.

If you want to use FastAPI's builtin OAuth2 features to protect some endpoints, you can do so as follows:

2.3.1 OAuth2 example

In the following example we pass dependencies into `FastAPIKwargs` to protect endpoints with unsafe HTTP methods.

```
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer
from piccolo_api.fastapi.endpoints import FastAPIWrapper, FastAPIKwargs
from piccolo_api.crud.endpoints import PiccoloCRUD

from my_app.tables import Movie

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")

FastAPIWrapper(
    root_url="/movie/",
    fastapi_app=app,
    piccolo_crud=PiccoloCRUD(
        table=Movie,
        read_only=False,
    ),
    fastapi_kwargs=FastAPIKwargs(
```

(continues on next page)

(continued from previous page)

```
        all_routes={"tags": ["Movie"]},  # Added to all endpoints
        post={"dependencies": [Depends(oauth2_scheme)]}, # protected route
        put={"dependencies": [Depends(oauth2_scheme)]},  # protected route
        patch={"dependencies": [Depends(oauth2_scheme)]},  # protected route
        delete_single={"dependencies": [Depends(oauth2_scheme)]},  # protected route
    )
)
```

See this example project for more details.

2.4 Source

2.4.1 FastAPIWrapper

class piccolo_api.fastapi.endpoints.**FastAPIWrapper**(*root_url: str*, *fastapi_app: FastAPI | APIRouter*, *piccolo_crud:* PiccoloCRUD, *fastapi_kwargs:* FastAPIKwargs | *None* = *None*)

Wraps `PiccoloCRUD` so it can easily be integrated into FastAPI. `PiccoloCRUD` can be used with any ASGI framework, but this way you get some of the benefits of FastAPI - namely, the OpenAPI integration. You get more control by building your own endpoints by hand, but `FastAPIWrapper` works great for getting endpoints up and running very quickly, and reducing boilerplate code.

Parameters

- **root_url** – The URL to mount the endpoint at - e.g. `'/movies/'`.
- **fastapi_app** – The `FastAPI` instance you want to attach the endpoints to.
- **piccolo_crud** – The `PiccoloCRUD` instance to wrap. `FastAPIWrapper` will obey the arguments passed into `PiccoloCRUD`, for example `ready_only` and `allow_bulk_delete`.
- **fastapi_kwargs** – Specifies the extra kwargs to pass to FastAPI's `add_api_route`.

2.4.2 FastAPIKwargs

class piccolo_api.fastapi.endpoints.**FastAPIKwargs**(*all_routes: dict[str, Any] = {}*, *get: dict[str, Any] = {}*, *delete: dict[str, Any] = {}*, *post: dict[str, Any] = {}*, *put: dict[str, Any] = {}*, *patch: dict[str, Any] = {}*, *get_single: dict[str, Any] = {}*, *delete_single: dict[str, Any] = {}*)

Allows kwargs to be passed into `FastAPIApp.add_api_route`.

CHAPTER

THREE

OPENAPI

Piccolo API has great support for viewing OpenAPI schemas using SwaggerUI.

3.1 Source

piccolo_api.openapi.endpoints.**swagger_ui**(*schema_url: str = '/openapi.json'*, *swagger_ui_title: str = 'Piccolo Swagger UI'*, *csrf_cookie_name: str | None = DEFAULT_COOKIE_NAME*, *csrf_header_name: str | None = DEFAULT_HEADER_NAME*, *swagger_ui_version: str = '5'*)

Even though ASGI frameworks such as FastAPI and BlackSheep have endpoints for viewing OpenAPI / Swagger docs, out of the box they don't work well with some Piccolo middleware (namely CSRF middleware, which requires Swagger UI to add the CSRF token to the header of each API call).

By using this endpoint instead, it will work correctly with CSRF.

FastAPI example

```
from fastapi import FastAPI
from piccolo_api.openapi.endpoints import swagger_ui

# By setting these values to None, we disable the builtin endpoints.
app = FastAPI(docs_url=None, redoc_url=None)

app.mount('/docs', swagger_ui())
```

Parameters

- **schema_url** – The URL to the OpenAPI schema.
- **csrf_cookie_name** – The name of the CSRF cookie.
- **csrf_header_name** – The HTTP header name which the CSRF cookie value will be added to.
- **swagger_ui_version** – Which version of Swagger UI to use.

CHAPTER

FOUR

CSP

CSP (Content Security Policy) middleware signals to a browser to only execute scripts which have come from the same domain. This provides some defence against cross site scripting.

4.1 Usage

```
from piccolo_api.csp.middleware import CSPMiddleware

app = CSPMiddleware(my_asgi_app)
```

4.2 Source

4.2.1 CSPConfig

class piccolo_api.csp.middleware.**CSPConfig**(*report_uri: 'Optional[bytes]' = None*, *default_src: 'str' = 'self'*)

4.2.2 CSPMiddleware

class piccolo_api.csp.middleware.**CSPMiddleware**(*app: ASGIApp*, *config:* CSPConfig *= CSPConfig()*)

Adds Content Security Policy headers to the response.

CHAPTER

FIVE

CSRF

5.1 Introduction

CSRF (Cross Site Request Forgery) is a serious security vulnerability for websites which are relying on cookies for authentication.

Browsers will send cookies belonging to a domain, no matter which website the request was made from. So if you're authenticated on foo.com, an attacker can make malicious HTTP requests to foo.com on your behalf even when you're browsing a completely different website.

5.1.1 Make sure safe methods are safe!

The CSRF protection only works on unsafe methods - DELETE, POST, PUT etc.

Make sure that safe methods (e.g. GET) are indeed safe, meaning they do not cause state to change (i.e. adding data to the database, or other side effects).

5.2 Prevention Measures

There are many approaches to preventing CSRF. The security measures implemented by this middleware are:

5.2.1 Double submit cookie

A cookie is set on the user's browser, containing a random token.

When submitting unsafe requests (POST, DELETE, PUT etc.), this token also needs to be added to the request using the `X-CSRFToken` header. The backend verifies that the token in the header matches the token in the cookie.

The Same Origin Policy means that cookies from one domain can't be read or set from another domain. This means an attacker doesn't know what the token value is.

> **Warning**
>
> The limitation of this approach is sub domains can set cookies on the root domain. For example, a.foo.com can create cookies on foo.com. If you have a secure website on foo.com, be aware of this attack vector. It's highly recommended not to give access to sub domains to any untrusted third parties.

Also, the Same Origin Policy prevents websites from setting custom headers when making AJAX requests to other domains. In theory just adding a custom header should be sufficient without the need for a token, but some people have been able to circumvent it in the past. A developer could also accidentally disable this protection using CORS, by allowing custom headers from other domains, which is why a token is also required.

Most popular AJAX libraries make setting this custom header very straight forward.

5.2.2 Referer checking

When running under HTTPS - which you 100% should be doing in production, then most browsers send `origin` and / or `referer` headers (source).

This is used to detect if requests are coming from a different domain.

5.2.3 Same Origin Cookies

This is a relatively new feature designed to prevent CSRF attacks, which is already supported by most evergreen browsers. If all browsers supported it, there would be no need for any of the other CSRF preventions outlined above, but until then we employ a defence in depth strategy, and use a combination of prevention strategies to protect older browsers.

The session cookies provided by Piccolo API are same origin.

5.3 Usage

5.3.1 Setup

Using the middleware is straightforward. You can wrap your ASGI app in it:

```
from piccolo_api.csrf.middleware import CSRFMiddleware

app = CSRFMiddleware(my_asgi_app, allowed_hosts=["foo.com"])
```

Or you can pass it to the middleware argument of your ASGI app. For example, with FastAPI:

```
from fastapi import FastAPI
from starlette.middleware import Middleware

app = FastAPI(middleware=[Middleware(CSRFMiddleware)])
```

Or Starlette:

```
from starlette import Starlette
from starlette.middleware import Middleware

app = Starlette(middleware=[Middleware(CSRFMiddleware)])
```

5.3.2 How it works

When the user makes a request, the middleware makes sure that a CSRF cookie is set on their device. This cookie contains a random token.

When a request is made with a non-safe method (e.g. POST), then the middleware checks for the cookie, and the token contained within the cookie either in a HTTP header or form field. If the token values don't match, the request is rejected.

Note: You have to explicitly tell the middleware to look for the token in a form field:

```
app = CSRFMiddleware(my_asgi_app, allow_form_param=True)
```

It isn't enabled by default, as adding it to the header is preferable (see below).

5.3.3 Accessing the CSRF token in HTML

As mentioned, you need to add the token contained within the CSRF cookie as a HTTP header or form field. There are two ways of accessing this value.

Template variable

Firstly, we can get the `csrftoken` value from the requests' ASGI scope, and then insert it into the template context:

```
def my_endpoint(request: Request):
    csrftoken = request.scope.get('csrftoken')
    csrf_cookie_name = request.scope.get('csrf_cookie_name')

    template = ENVIRONMENT.get_template("example.html.jinja")
    content = template.render(
        csrftoken=csrftoken,
        csrf_cookie_name=csrf_cookie_name
    )

    return HTMLResponse(content)
```

And then within the HTML template:

```
<form method="POST">
    <label>Username</label>
    <input type="text" name="username" />
    <label>Password</label>
    <input type="password" name="password" />

    {% if csrftoken and csrf_cookie_name %}
        <input type="hidden" name="{{ csrf_cookie_name }}">{{ csrtoken }}</input>
    {% endif %}

    <button>Login</button>
</form>
```

Reading from the cookie

Rather than injecting the CSRF token into the template, we can read it directly from the cookie, using something like js-cookie.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/js-cookie/2.2.1/js.cookie.min.js"></
↪script>

<script>
var csrftoken = Cookies.get('csrftoken');
</script>
```

Full example

In the the example below, we get the token from the cookie, and add it to the request header.

```
<form method="POST" onsubmit="login(event,this)">
    <label>Username</label>
    <input type="text" name="username" />
    <label>Password</label>
    <input type="password" name="password" />

    <button>Login</button>
</form>

<script src="https://cdnjs.cloudflare.com/ajax/libs/js-cookie/2.2.1/js.cookie.min.js"></
↪script>

<script>
    function login(event, form) {
        const csrftoken = Cookies.get('csrftoken');
        event.preventDefault()
        fetch("/login/", {
            method: "POST",
            credentials: "same-origin",
            headers: {
                "X-CSRFToken": csrftoken,
                "Accept": "application/json",
                "Content-Type": "application/json"
            },
            body: JSON.stringify({ username: form.username.value, password: form.
↪password.value })
        })
    }
</script>
```

5.3.4 Should I embed the token in the form, or add it as a HTTP header?

Setting the cookie in the header is preferable as:

- It makes caching easier, as CSRF tokens aren't embedded in HTML forms.
- We no longer have to worry about BREACH attacks.

However, you can embed the CSRF token in the form if you want.

```
app = CSRFMiddleware(my_asgi_app, allow_form_param=True)
```

To guard against BREACH attacks, you can use rate limiting middleware on that endpoint, or just disable HTTP compression for your website.

5.3.5 Source

class piccolo_api.csrf.middleware.**CSRFMiddleware**(*app: ASGIApp*, *allowed_hosts: Sequence[str] = []*, *cookie_name: str = DEFAULT_COOKIE_NAME*, *header_name: str = DEFAULT_HEADER_NAME*, *max_age: int = ONE_YEAR*, *allow_header_param: bool = True*, *allow_form_param: bool = False*, ***kwargs*)

For GET requests, set a random token as a cookie. For unsafe HTTP methods, require a HTTP header to match the cookie value, otherwise the request is rejected.

This uses the Double Submit Cookie style of CSRF prevention. For more information see the OWASP cheatsheet (double submit cookie and customer request headers).

By default, the CSRF token needs to be added to the request header. By setting `allow_form_param` to `True`, it will also work if added as a form parameter.

Parameters

- **app** – The ASGI app you want to wrap.
- **allowed_hosts** – If using this middleware with HTTPS, you need to set this value, for example `['example.com']`.
- **cookie_name** – You can specify a custom name for the cookie. There should be no need to change it, unless in the rare situation where the name clashes with another cookie.
- **header_name** – You can tell the middleware to look for the CSRF token in a different HTTP header.
- **max_age** – The max age of the cookie, in seconds.
- **allow_header_param** – Whether to look for the CSRF token in the HTTP headers.
- **allow_form_param** – Whether to look for the CSRF token in a form field with the same name as the cookie. By default, it's not enabled.

CHAPTER

SIX

ENCRYPTION

6.1 Introduction

Piccolo API provides some wrappers around popular encryption libraries.

These are current used by *Multifactor Authentication*.

6.2 Providers

6.2.1 EncryptionProvider

class piccolo_api.encryption.providers.**EncryptionProvider**(*prefix: str*)

Base class for encryption providers. Don't use it directly, it must be subclassed.

6.2.2 FernetProvider

class piccolo_api.encryption.providers.**FernetProvider**(*encryption_key: bytes*)

Uses the Fernet algorithm for encryption.

Parameters

encryption_key – This can be generated using `FernetEncryption.get_new_key()`.

6.2.3 PlainTextProvider

class piccolo_api.encryption.providers.**PlainTextProvider**

The values aren't encrypted - can be useful for testing.

6.2.4 XChaCha20Provider

class piccolo_api.encryption.providers.**XChaCha20Provider**(*encryption_key: bytes*)

Uses the XChaCha20-Poly1305 algorithm for encryption.

This is more secure than `FernetProvider`.

Parameters

encryption_key – This can be generated using `XChaCha20Provider.get_new_key()`.

6.2.5 Dependencies

When first using some of the providers, you will be prompted to install the underlying encryption library.

For example, with `XChaCha20Provider`, you need to install `pynacl` as follows:

```
pip install piccolo_api[pynacl]
```

6.2.6 Example usage

All of the providers work the same (except their parameters may be different).

Here's an example using `XChaCha20Provider`:

```
>>> from piccolo_api.encryption.providers import XChaCha20Provider

>>> encryption_key = XChaCha20Provider.get_new_key()
>>> provider = XChaCha20Provider(encryption_key=encryption_key)

>>> encrypted = provider.encrypt("hello world")
>>> print(provider.decrypt(encrypted))
"hello world"
```

6.2.7 Which provider to use?

`XChaCha20Provider` is the most secure.

You may decide to use `FernetProvider` if you already have the Python `cryptography` library as a dependency in your project.

6.2.8 Passing in encryption keys via environment variables

A common way of passing sensitive information to an app is via environment variables.

The encryption keys for `XChaCha20Provider` and `FernetProvider` are in bytes. You can still pass them in as environment variables though.

One approach (using `XChaCha20Provider` as an example), is to convert the bytes to a hex string:

```
>>> key = XChaCha20Provider.get_new_key()
>>> key.hex()
'25d49a31af520fd4c24553890f154deeead1fb61a409e6ea3df7b62ed4b8925d'
```

You can then use the hex string as the environment variable. To convert it back into bytes:

```
>>> key = bytes.fromhex('25d49a31af520fd4c24553890f154deeead1fb61a409e6ea3df7b62ed4b8925d
↪')
```

CHAPTER

SEVEN

RATE LIMITING

7.1 Introduction

Rate limiting is useful for certain public API endpoints. Examples are:

- Login endpoints - reducing the speed with which passwords can be brute forced.
- Computationally intensive endpoints, which could lead to a DOS attack.

7.2 RateLimitingMiddleware

Usage is simple - just wrap your ASGI app:

```
from piccolo_api.rate_limiting.middleware import RateLimitingMiddleware

app = RateLimitingMiddleware(my_asgi_app)
```

7.2.1 Source

class piccolo_api.rate_limiting.middleware.**RateLimitingMiddleware**(*app: ASGIApp*, *provider:* RateLimitProvider | *None* = *None*)

Blocks clients who exceed a given number of requests in a given time period.

Parameters

- **app** – The ASGI app to wrap.
- **provider** – Provides the logic around rate limiting. If not specified, it will default to a *InMemoryLimitProvider*.

7.3 Providers

The middleware can work with different *Providers*, which are responsible for storing traffic data, and signalling when a rate limit has been exceeded.

By default `InMemoryLimitProvider` is used.

7.4 InMemoryLimitProvider

Stores the traffic data in memory. You can customise it as follows:

```
app = RateLimitingMiddleware(
    my_asgi_app,
    provider=InMemoryLimitProvider(
        limit=1000,
        timespan=300
    ),
)
```

The `limit` is the number of requests needed by a client within the `timespan` (measured in seconds) to trigger a 429 error (too many requests).

If you want a blocked client to be allowed access again after a time period, specify this using the `block_duration` argument:

```
app = RateLimitingMiddleware(
    my_asgi_app,
    provider=InMemoryLimitProvider(
        limit=1000,
        timespan=300,
        block_duration=300  # Blocked for 5 minutes
    ),
)
```

7.4.1 Source

class piccolo_api.rate_limiting.middleware.**InMemoryLimitProvider**(*timespan: int*, *limit: int = 1000*, *block_duration: int | None = None*)

A very simple rate limiting provider - works fine when running a single application instance.

Time values are given in seconds, rather than a timedelta, for improved performance.

Parameters

- **timespan** – The time in seconds between resetting the number of requests. Beware setting it too high, because memory usage will increase.
- **limit** – The number of requests in the timespan, before getting blocked.
- **block_duration** – If set, the number of seconds before a client is no longer blocked. Otherwise, they're only removed when the app is restarted.

7.5 Custom Providers

Making a provider is simple, if the built-in ones don't meet your needs. A provider needs to implement a simple interface - see *RateLimitProvider*.

7.5.1 Source

class piccolo_api.rate_limiting.middleware.**RateLimitProvider**

An abstract base class which all rate limit providers should inherit from.

abstractmethod increment(*identifier: str*)

Parameters
identifier – A unique identifier for the client (for example, IP address).

Raises
RateLimitError – If too many requests are received from a client with this identifier.

CHAPTER

EIGHT

WHICH AUTH TO USE?

Piccolo API provides easy-to-use middleware and endpoints for implementing authentication in your ASGI applications.

To learn more about how ASGI works, see the Introduction to ASGI article on the Piccolo blog. FastAPI and Starlette are examples of ASGI frameworks.

For most web apps, we recommend using *Session Auth*. It is robust, and well understood. Piccolo API has a very complete implementation with endpoints for logging in, logging out, changing password, and more.

Token Auth is useful when authenticating mobile apps, or machine to machine communication.

JWT Auth has emerged in recent years as an alternative to session auth. Rather than storing a session in a database and using cookies, it uses signed tokens instead. If you application requires JWT, then we have basic support for it, but we recommend *Session Auth* for most applications.

CHAPTER

NINE

JWT AUTH

9.1 Introduction

JSON Web Token (JWT) is a token format, often used for authentication. For more information see the jwt.io website.

9.2 Endpoints

An endpoint is provided for JWT login, and is designed to integrate with an ASGI app, such as Starlette or FastAPI.

9.2.1 jwt_login

This creates an endpoint for logging in, and getting a JSON Web Token (JWT).

```
from piccolo_api.jwt_auth.endpoints import jwt_login
from starlette import Starlette
from starlette.routing import Route, Router


app = Starlette(
    routes=[
        Route(
            path="/login/",
            endpoint=jwt_login(
                secret='mysecret123'
            )
        ),
    ]
)
```

secret

This is used for signing the JWT.

expiry

An optional argument, which allows you to control when a token expires. By default it's set to 1 day.

```
from datetime import timedelta

jwt_login(
    secret='mysecret123',
    expiry=timedelta(minutes=10)
)
```

Hint

You generally want short expiry tokens for web applications, and longer expiry times for mobile applications.

Hint

See *JWTMiddleware* for how to protect your endpoints.

9.2.2 Usage

You can use any HTTP client to get the JWT token. In our example we use `curl`.

To get a JWT token:

```
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"username": "piccolo", "password": "piccolo123"}' \
    http://localhost:8000/login/
```

To get data from a protected endpoint:

```
curl -H "Authorization: Bearer your-JWT-token" \
    http://localhost:8000/private/movies/
```

Hint

You can use all `HTTP` methods by passing a valid JWT token in the `Authorization` header.

9.2.3 Source

piccolo_api.jwt_auth.endpoints.**jwt_login**(*secret: str*, *auth_table: type[BaseUser] = BaseUser*, *expiry: timedelta = timedelta(days=1)*) → type[JWTLoginBase]

Create an endpoint for generating JWT tokens.

Parameters

- **secret** – Used to sign the the JWT tokens.
- **auth_table** – Which Piccolo table to use to authenticate the user.
- **expiry** – How long before the JWT token expires.

9.3 Middleware

This middleware protects endpoints using JWT tokens.

9.3.1 Setup

`JWTMiddleware` wraps an ASGI app, and ensures a valid token is passed in the header. Otherwise a 401 error is returned. If the token is valid, the corresponding `user_id` is added to the ASGI `scope`.

blacklist

Optionally, you can pass in a `blacklist` argument, which is a subclass of `JWTBlacklist`. The implementation of the `in_blacklist` method is up to the user - the data could come from a database, a file, a Python list, or anywhere else.

```
# An example blacklist.

BLACKLISTED_TOKENS = ['abc123', 'def456']


class MyBlacklist(JWTBlacklist):
    async def in_blacklist(self, token: str) -> bool:
        return token in BLACKLISTED_TOKENS


asgi_app = JWTMiddleware(
    my_endpoint,
    secret='mysecret123',
    blacklist=MyBlacklist()
)
```

Hint

Blacklists are important if you have tokens with a long expiry date.

allow_unauthenticated

By default, if the JWT token is invalid then the HTTP request is rejected. However, by setting `allow_unauthenticated=True` the request will be allowed to continue instead. The following will be added to the ASGI scope:

- `user_id`, which is set to `None`
- `jwt_error`, explaining why the token is invalid (see *JWTError*)

It is then up to the endpoints in your application to check whether `user_id` is `None` and then reject the request.

This is useful when the middleware is wrapping lots of endpoints, and you want more control over which ones are protected, or want to take additional actions when an error occurs before rejecting the request.

9.3.2 Source

JWTMiddleware

class piccolo_api.jwt_auth.middleware.**JWTMiddleware**(*asgi: ASGIApp*, *secret: str*, *auth_table: type[BaseUser] = BaseUser*, *blacklist: JWTBlacklist = JWTBlacklist()*, *allow_unauthenticated: bool = False*)

Protects ASGI endpoints - only allows access if a JWT token is present in the `authorization` HTTP header.

Parameters

- **asgi** – The ASGI app to protect.
- **secret** – The secret used to decode the JWT token.
- **auth_table** – The Piccolo table containing users - either `BaseUser` or a subclass.
- **blacklist** – Any tokens in this list will be rejected.
- **allow_unauthenticated** – By default the middleware rejects any requests with an invalid token.

JWTBlacklist

class piccolo_api.jwt_auth.middleware.**JWTBlacklist**

Inherit from this class, and override *in_blacklist()*. Used in conjunction with *JWTMiddleware*. An example is *StaticJWTBlacklist*.

async **in_blacklist**(*token: str*) → bool

Checks whether the token is in the blacklist.

StaticJWTBlacklist

class piccolo_api.jwt_auth.middleware.**StaticJWTBlacklist**(*blacklist: list[str]*)

A simple implementation of *JWTBlacklist*, which rejects a token if it's in the given list.

JWTError

class piccolo_api.jwt_auth.middleware.**JWTError**(*value*)

This enum contains all of the possible errors which can be returned by *JWTMiddleware*. If `allow_unauthenticated=True` then these errors will be added to the ASGI scope instead under `jwt_error`.

token_expired = 'Token has expired'

token_invalid = 'Token is invalid'

token_not_found = 'Token not found'

token_revoked = 'Token revoked'

user_not_found = 'User not found'

9.4 Full Example

Let's combine all of previous examples into a complete app.

9.4.1 FastAPI

```
from datetime import timedelta

from fastapi import FastAPI, Request
from piccolo_admin.endpoints import create_admin
from piccolo_api.crud.endpoints import PiccoloCRUD
from piccolo_api.fastapi.endpoints import FastAPIKwargs, FastAPIWrapper
from piccolo_api.jwt_auth.endpoints import jwt_login
from piccolo_api.jwt_auth.middleware import JWTBlacklist, JWTMiddleware
from piccolo.apps.user.tables import BaseUser
from starlette.routing import Mount, Route

from home.tables import Movie  # An example Table


public_app = FastAPI(
    routes=[
        Mount(
            "/admin/",
            create_admin(tables=[Movie]),
        ),
        Route(
            path="/login/",
            endpoint=jwt_login(
                secret="mysecret123",
                expiry=timedelta(minutes=60),  # default is 1 day
            ),
        ),
    ],
)


BLACKLISTED_TOKENS = []


class MyBlacklist(JWTBlacklist):
    async def in_blacklist(self, token: str) -> bool:
        return token in BLACKLISTED_TOKENS


private_app = FastAPI()

protected_app = JWTMiddleware(
    private_app,
    auth_table=BaseUser,
    secret="mysecret123",
    blacklist=MyBlacklist(),
)

FastAPIWrapper(
    "/movies/",
    fastapi_app=private_app,
```

(continues on next page)

(continued from previous page)

```
    piccolo_crud=PiccoloCRUD(Movie, read_only=False),
    fastapi_kwargs=FastAPIKwargs(
        all_routes={"tags": ["Movies"]},
    ),
)

public_app.mount("/private", protected_app)


# This is optional if you want to provide a logout endpoint
# in your application. By adding a token to the token blacklist,
# you are invalidating the token and need to login again to get
# new valid token
@private_app.get("/logout/")
async def logout(request: Request) -> None:
    BLACKLISTED_TOKENS.append(
        request.headers.get("authorization").split(" ")[-1]
    )
```

9.4.2 Starlette

Is almost identical to the FastAPI example - just replace `FastAPI` with `Starlette`, and use Starlette's `HTTPEndpoint` for your endpoints. You also need to write your own crud endpoints because `Starlette` can't use `FastAPIWrapper`. An example is in SessionAuth.

CHAPTER

TEN

SESSION AUTH

10.1 Introduction

Session auth is the classic approach to authentication on the web. When a user logs in, a session cookie is set on their browser, which contains a unique session ID. This session ID is also stored by the server in a database. Each time the user makes a request, the session ID stored in the cookie is compared with the ones stored in the database, to check if the user has a valid session.

There are several advantages to session auth:

- A session can be invalidated at any time, by deleting a session from the database.
- HTTP-only cookies are immune to tampering with JavaScript.

10.2 Tables

We store the session tokens in *SessionsBase* table, and the user credentials in `BaseUser` table.

10.2.1 Migrations

We recommend creating these tables using migrations.

You can add `piccolo_api.session_auth.piccolo_app` to the `apps` arguments of the `AppRegistry` in `piccolo_conf.py`.

```
APP_REGISTRY = AppRegistry(
    apps=[
        ...
        "piccolo_api.session_auth.piccolo_app",
        "piccolo.apps.user.piccolo_app",
        ...
    ]
)
```

To learn more about Piccolo apps, see the Piccolo docs.

To run the migrations and create the tables, run:

```
piccolo migrations forwards user
piccolo migrations forwards session_auth
```

10.2.2 Creating them manually

If you prefer not to use migrations, and want to create them manually, you can do this instead:

```
from piccolo_api.session_auth.tables import SessionsBase
from piccolo.apps.user.tables import BaseUser
from piccolo.tables import create_tables

create_tables(BaseUser, SessionsBase, if_not_exists=True)
```

10.2.3 Source

SessionsBase

class piccolo_api.session_auth.tables.**SessionsBase**(*_data: dict[Column, Any] | None = None*, *_ignore_missing: bool = False*, *_exists_in_db: bool = False*, ***kwargs*)

Use this table, or inherit from it, to create a session store.

async classmethod create_session(*user_id: int*, *expiry_date: datetime | None = None*, *max_expiry_date: datetime | None = None*) → *SessionsBase*

Creates a session in the database.

classmethod create_session_sync(*user_id: int*, *expiry_date: datetime | None = None*) → *SessionsBase*

A sync equivalent of *create_session()*.

async classmethod get_user_id(*token: str*, *increase_expiry: timedelta | None = None*) → int | None

Returns the `user_id` if the given token is valid, otherwise `None`.

Parameters

increase_expiry – If set, the `expiry_date` will be increased by the given amount if it's close to expiring. If it has already expired, nothing happens. The `max_expiry_date` remains the same, so there's a hard limit on how long a session can be used for.

classmethod get_user_id_sync(*token: str*) → int | None

A sync wrapper around *get_user_id()*.

async classmethod remove_session(*token: str*)

Deletes a matching session from the database.

classmethod remove_session_sync(*token: str*)

A sync wrapper around *remove_session()*.

expiry_date

Stores the expiry date for this session.

id: Serial

An alias to an autoincrementing integer column in Postgres.

max_expiry_date

We set a hard limit on the expiry date - it can keep on getting extended up until this value, after which it's best to invalidate it, and either require login again, or just create a new session token.

token

Stores the session token.

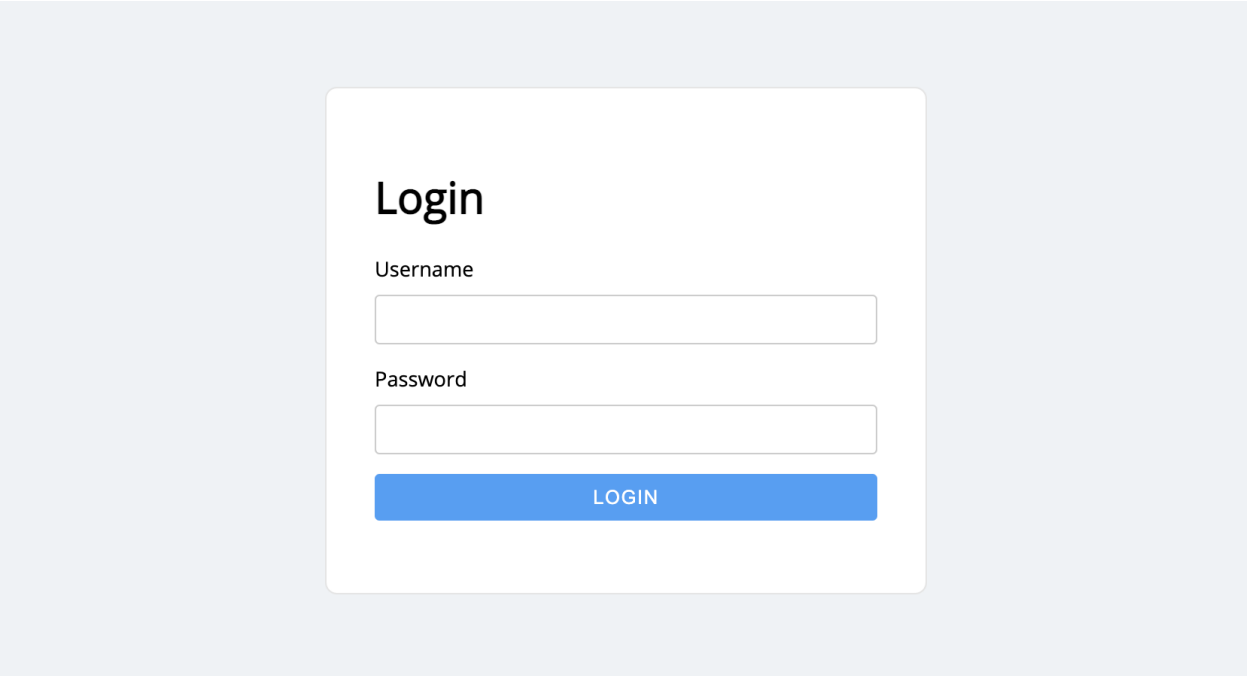
`user_id`
: Stores the user ID.

10.3 Endpoints

Endpoints are provided for session login and logout. They are designed to integrate with an ASGI app, such as Starlette or FastAPI.

10.3.1 session_login

An endpoint for creating a user session. If you send a GET request to this endpoint, a simple login form is rendered, where a user can manually login.



Hint

You can use a custom login template, which matches the look and feel of your application. See the `template_path` parameter.

Alternatively, you can login programatically by sending a POST request to this endpoint (passing in `username` and `password` parameters as JSON, or as form data).

When login is successful, the user is redirected. The destination can be configured using the `redirect_to` parameter.

Additional security

It's recommended to protect any login endpoints with *rate limiting middleware*, to help slow down any brute force attacks.

You can also add a CAPTCHA if you prefer. The approach is very similar to the *register endpoint*.

Examples

Here's a Starlette example:

```
from piccolo_api.session_auth.endpoints import session_login
from starlette import Starlette

app = Starlette()

app.mount("/login/", session_login())
```

Here's a FastAPI example:

```
from piccolo_api.session_auth.endpoints import session_login
from fastapi import FastAPI

app = FastAPI()

app.mount("/login/", session_login())
```

Source

piccolo_api.session_auth.endpoints.**session_login**(*auth_table: type[BaseUser] = BaseUser*, *session_table: type[SessionsBase] = SessionsBase*, *session_expiry: timedelta = timedelta(hours=1)*, *max_session_expiry: timedelta = timedelta(days=7)*, *redirect_to: str | None = '/'*, *production: bool = False*, *cookie_name: str = 'id'*, *template_path: str | None = None*, *hooks: LoginHooks | None = None*, *captcha: Captcha | None = None*, *styles: Styles | None = None*, *mfa_providers: Sequence[MFAProvider] | None = None*) → type[SessionLoginEndpoint]

An endpoint for creating a user session.

Parameters

- **auth_table** – Which table to authenticate the username and password with. It defaults to `BaseUser`.
- **session_table** – Which table to store the session in. If defaults to *SessionsBase*.
- **session_expiry** – How long the session will last.
- **max_session_expiry** – If the session is refreshed (see the `increase_expiry` parameter for *SessionsAuthBackend*), it can only be refreshed up to a certain limit, after which the session is void.
- **redirect_to** – Where to redirect to after successful login.
- **production** – Adds additional security measures. Use this in production, when serving your app over HTTPS.
- **cookie_name** – The name of the cookie used to store the session token. Only override this if the name of the cookie clashes with other cookies.
- **template_path** – If you want to override the default login HTML template, you can do so by specifying the absolute path to a custom template. For example `'/`

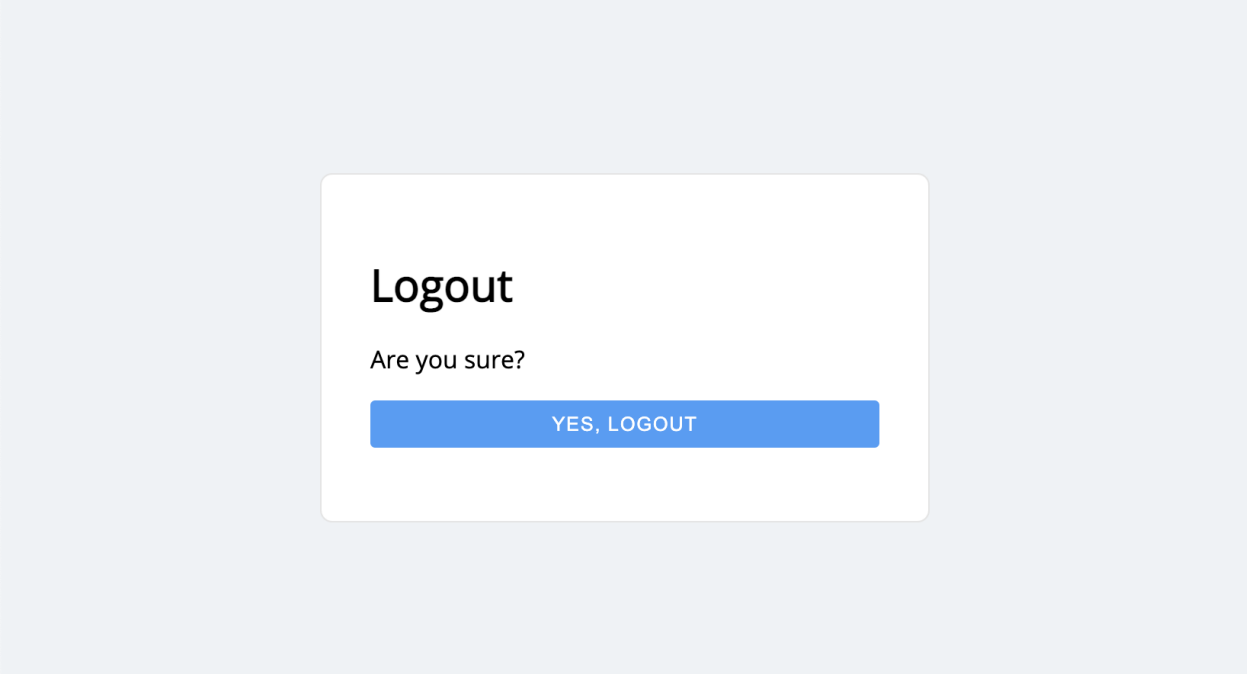
`some_directory/login.html'`. Refer to the default template at `piccolo_api/templates/session_login.html` as a basis for your custom template.

- **`hooks`** – Allows you to run custom logic at various points in the login process. See *`LoginHooks`*.
- **`captcha`** – Integrate a CAPTCHA service, to provide protection against bots. See *`Captcha`*.
- **`styles`** – Modify the appearance of the HTML template using CSS.
- **`mfa_providers`** – Add additional security to the login process using Multi-Factor Authentication.

10.3.2 session_logout

This unsets the cookie value, and invalidates the session in the database, if you send a POST request.

If you send a GET request, a simple logout form is rendered, where a user can manually logout.



Hint

You can use a custom logout template, which matches the look and feel of your application. See the `template_path` parameter.

Examples

Here's a Starlette example:

```
from piccolo_api.session_auth.endpoints import session_logout
from starlette import Starlette

app = Starlette()
```

(continues on next page)

(continued from previous page)

```
app.mount("/logout/", session_logout())
```

Here's a FastAPI example:

```
from piccolo_api.session_auth.endpoints import session_logout
from fastapi import FastAPI

app = FastAPI()

app.mount("/logout/", session_logout())
```

Source

piccolo_api.session_auth.endpoints.**session_logout**(*session_table: type[SessionsBase] = SessionsBase*, *cookie_name: str = 'id'*, *redirect_to: str | None = None*, *template_path: str | None = None*, *styles: Styles | None = None*) → type[SessionLogoutEndpoint]

An endpoint for clearing a user session.

Parameters

- **session_table** – Which table to store the session in. It defaults to *SessionsBase*.
- **cookie_name** – The name of the cookie used to store the session token. Only override this if the name of the cookie clashes with other cookies.
- **redirect_to** – Where to redirect to after logging out.
- **template_path** – If you want to override the default logout HTML template, you can do so by specifying the absolute path to a custom template. For example `'/some_directory/logout.html'`. Refer to the default template at `piccolo_api/templates/logout.html` as a basis for your custom template.
- **styles** – Modify the appearance of the HTML template using CSS.

10.4 Middleware

This middleware protects endpoints using session cookies.

10.4.1 Setup

The middleware wraps your ASGI app.

Here's a Starlette example:

```
from starlette import Starlette
from starlette.middleware.authentication import AuthenticationMiddleware
from piccolo_api.session_auth.middleware import SessionsAuthBackend

my_asgi_app = Starlette()
```

(continues on next page)

(continued from previous page)

```
app = AuthenticationMiddleware(
    my_asgi_app,
    backend=SessionsAuthBackend(),
)
```

Here's a FastAPI example:

```
from fastapi import FastAPI
from starlette.middleware.authentication import AuthenticationMiddleware
from piccolo_api.session_auth.middleware import SessionsAuthBackend

my_asgi_app = FastAPI()

app = AuthenticationMiddleware(
    my_asgi_app,
    backend=SessionsAuthBackend(),
)
```

On each request, the middleware looks for a session token held in a cookie, and whether a corresponding user is found in the database. If so, a `user` object is added to the ASGI scope.

10.4.2 Accessing user

In Starlette you can access the user in your endpoint using `request.user` or `request.scope['user']`.

```
from starlette.requests import Request
from starlette.endpoints import HTTPEndpoint
from starlette.responses import JSONResponse

class PostsEndpoint(HTTPEndpoint):
    async def get(self, request: Request):
        piccolo_user = request.user.user
        posts = await Post.select().where(
            Post.author.id == piccolo_user.id
        ).run()
        return JSONResponse(posts)
```

In FastAPI, you can also access the user from the request scope, as follows:

```
from fastapi.requests import Request
from fastapi.responses import JSONResponse

@app.get('/posts/')
def get_posts(request: Request):
    piccolo_user = request.user.user
    posts = await Post.select().where(
        Post.author.id == piccolo_user.id
    ).run()
    return JSONResponse(posts)
```

10.4.3 excluded_paths

This works identically to token auth - see *excluded_paths*.

10.4.4 Source

SessionsAuthBackend

class piccolo_api.session_auth.middleware.**SessionsAuthBackend**(*auth_table: type[BaseUser] = PiccoloBaseUser*, *session_table: type[SessionsBase] = SessionsBase*, *cookie_name: str = 'id'*, *admin_only: bool = True*, *superuser_only: bool = False*, *active_only: bool = True*, *increase_expiry: timedelta | None = None*, *allow_unauthenticated: bool = False*, *excluded_paths: Sequence[str] | None = None*)

Authenticaion middleware which uses session cookies.

Parameters

- **auth_table** – The Piccolo table used for authenticating users. It defaults to `BaseUser`.
- **session_table** – The Piccolo table used for storing sessions. If defaults to `SessionsBase`.
- **cookie_name** – The name of the session cookie. Override this if it clashes with other cookies in your application.
- **admin_only** – If `True`, users which aren't admins will be rejected.
- **superuser_only** – If `True`, users which aren't superusers will be rejected.
- **active_only** – If `True`, users which aren't active will be rejected.
- **increase_expiry** – If set, the session expiry will be increased by this amount on each request, if it's close to expiry. This allows sessions to have a short expiry date, whilst also providing a good user experience.
- **allow_unauthenticated** – If `True`, when a matching user session can't be found, the request still continues, but an unauthenticated user is added to the scope. It's then up to the application's endpoints to check if a user is authenticated or not using `request.user.is_authenticated`. If `False`, the request is automatically rejected if a user session can't be found.
- **excluded_paths** – These paths don't require a session cookie - useful if you want to exclude a few URLs, such as docs.

10.5 Commands

If you've registered the `session_auth` app in your `piccolo_conf.py` file (see the *migrations docs*), it gives you access to a custom command.

10.5.1 clean

If you run the following on the command line, it will delete any old sessions from the database.

```
piccolo session_auth clean
```

10.6 Full Example

Let's combine all of previous examples into a complete app.

10.6.1 FastAPI

```
import datetime
from typing import Any

from fastapi import FastAPI
from fastapi.responses import JSONResponse
from home.tables import Movie  # An example Table
from piccolo.engine import engine_finder
from piccolo_admin.endpoints import create_admin
from starlette.middleware import Middleware
from starlette.middleware.authentication import AuthenticationMiddleware
from starlette.routing import Mount, Route

from piccolo_api.crud.serializers import create_pydantic_model
from piccolo_api.csrf.middleware import CSRFMiddleware
from piccolo_api.openapi.endpoints import swagger_ui
from piccolo_api.session_auth.endpoints import session_login, session_logout
from piccolo_api.session_auth.middleware import SessionsAuthBackend

# The main app with public endpoints, which will be served by Uvicorn
app = FastAPI(
    routes=[
        # If we want to use Piccolo admin:
        Mount(
            "/admin/",
            create_admin(
                tables=[Movie],
                # Required when running under HTTPS:
                # allowed_hosts=['my_site.com']
            ),
        ),
        # Session Auth login:
        Mount(
            "/login/",
            session_login(redirect_to="/private/docs/"),
        ),
    ],
    docs_url=None,
    redoc_url=None,
)
```

(continues on next page)

(continued from previous page)

```
# The private app with SessionAuth protected endpoints
private_app = FastAPI(
    routes=[
        Route("/logout/", session_logout(redirect_to="/")),
        # We use a custom Swagger docs endpoint instead of the default FastAPI
        # one, because this one supports CSRF middleware:
        Mount("/docs/", swagger_ui(schema_url="/private/openapi.json")),
    ],
    middleware=[
        Middleware(
            AuthenticationMiddleware,
            backend=SessionsAuthBackend(
                increase_expiry=datetime.timedelta(minutes=30)
            ),
        ),
        # CSRF middleware provides additional protection for older browsers, as
        # we're using cookies.
        Middleware(CSRFMiddleware, allow_form_param=True),
    ],
    # We disable the default Swagger docs, as we have a custom one (see above).
    docs_url=None,
    redoc_url=None,
)


# Mount our private app within the main app.
app.mount("/private/", private_app)


################################################################################
# Example FastAPI endpoints and Pydantic models.


MovieModelIn: Any = create_pydantic_model(
    table=Movie, model_name="MovieModelIn"
)

MovieModelOut: Any = create_pydantic_model(
    table=Movie, include_default_columns=True, model_name="MovieModelOut"
)


@private_app.get("/movies/", response_model=list[MovieModelOut])
async def movies():
    return await Movie.select().order_by(Movie._meta.primary_key)


@private_app.post("/movies/", response_model=MovieModelOut)
async def create_movie(movie_model: MovieModelIn):
    movie = Movie(**movie_model.dict())
    await movie.save()
```

(continues on next page)

(continued from previous page)

```
    return MovieModelOut(**movie.to_dict())


@private_app.put("/movies/{movie_id}/", response_model=MovieModelOut)
async def update_movie(movie_id: int, movie_model: MovieModelIn):
    movie = await Movie.objects().get(Movie._meta.primary_key == movie_id)
    if not movie:
        return JSONResponse({}, status_code=404)

    for key, value in movie_model.dict().items():
        setattr(movie, key, value)

    await movie.save().run()

    return MovieModelOut(**movie.to_dict())


@private_app.delete("/movies/{movie_id}/")
async def delete_movie(movie_id: int):
    movie = (
        await Movie.objects()
        .where(Movie._meta.primary_key == movie_id)
        .first()
    )
    if not movie:
        return JSONResponse({}, status_code=404)

    await movie.remove()

    return JSONResponse({})


###############################################################################
# This is optional - it's for creating a connection pool.


@app.on_event("startup")
async def open_database_connection_pool():
    try:
        engine = engine_finder()
        await engine.start_connection_pool()
    except Exception:
        print("Unable to connect to the database")


@app.on_event("shutdown")
async def close_database_connection_pool():
    try:
        engine = engine_finder()
        await engine.close_connection_pool()
    except Exception:
        print("Unable to connect to the database")
```

(continues on next page)

(continued from previous page)

```
###############################################################################

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app)
```

10.6.2 Starlette

Is almost identical to the FastAPI example - just replace `FastAPI` with `Starlette`, and use Starlette's `HTTPEndpoint` for your endpoints.

CHAPTER

ELEVEN

MULTI-FACTOR AUTHENTICATION

11.1 Introduction

11.1.1 What is Multi-Factor Authentication (MFA)?

MFA provides additional security to *Session Auth*.

As well as needing a username and password to login, the user must provide an additional piece of information.

One of the most popular ways of doing this is by providing a code generated by an authenticator app on the user's phone.

11.2 Endpoints

You must mount these ASGI endpoints in your app.

11.2.1 `mfa_setup`

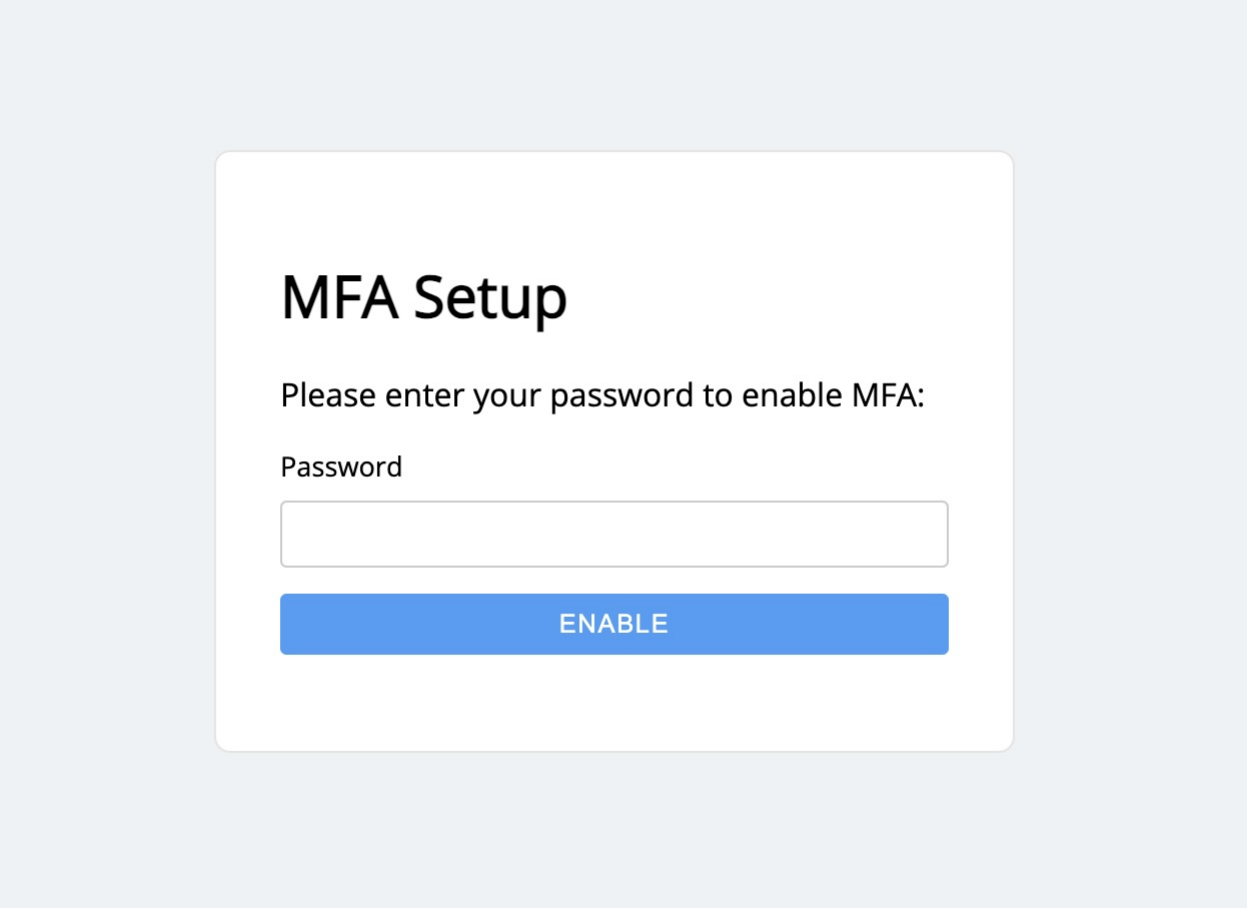
`piccolo_api.mfa.endpoints.`**`mfa_setup`**(*provider:* MFAProvider, *auth_table:* *type[BaseUser]* = *BaseUser*, *styles:* Styles | *None* = *None*) → type[HTTPEndpoint]

This endpoint needs to be protected `SessionAuthMiddleware`, ensuring that only logged in users can access it.

We also recommend protecting it with `RateLimitingMiddleware`, because:

- Some of the forms accept a password, and we want to protect against brute forcing.
- Generating secrets and refresh tokens is somewhat expensive, so we want to protect against abuse.

Users can setup and manage their MFA setup using this endpoint.



11.2.2 session_login

Make sure you pass the `mfa_providers` argument to *session_login*, so it knows to look for an MFA token.

11.3 Providers

Most of the MFA code is fairly generic, but `Providers` implement the logic which is specific to its particular authentication mechanism.

For example, `AuthenticatorProvider` knows how to authenticate tokens which come from an authenticator app on a user's phone, and knows how to generate new secrets which allow users to enable MFA.

11.3.1 MFAProvider

class piccolo_api.mfa.provider.**MFAProvider**(*name: str = 'MFA Code'*)

This is the base class which all providers must inherit from. Use it to build your own custom providers. If you use it directly, it won't do anything. See *AuthenticatorProvider* for a concrete implementation.

Parameters

token_name – Each provider should specify a unique `token_name`, so when a token is passed to the login endpoint, we know which `MFAProvider` it belongs to.

11.3.2 AuthenticatorProvider

class piccolo_api.mfa.authenticator.provider.**AuthenticatorProvider**(*encryption_provider:* EncryptionProvider, *recovery_code_count:* *int* = *8*, *secret_table:* *type[AuthenticatorSecret]* = *AuthenticatorSecret*, *issuer_name:* *str* = *'Piccolo-MFA'*, *register_template_path:* *str* | *None* = *None*, *styles:* Styles | *None* = *None*, *valid_window:* *int* = *0*)

Allows authentication using an authenticator app on the user's phone, like Google Authenticator.

Parameters

- **encryption_provider** – The shared secrets can be encrypted in the database. We recommend using *XChaCha20Provider*. Use *PlainTextProvider* to store the secrets as plain text.
- **recovery_code_count** – How many recovery codes should be generated.
- **secret_table** – This is the table used to store secrets. You shouldn't have to override this, unless you subclassed the default `AuthenticatorSecret` table for some reason.
- **issuer_name** – This is how it will be identified in the user's authenticator app.
- **register_template_path** – You can override the HTML template if you want. Try using the `styles` param instead though if possible if you just want basic visual changes.
- **styles** – Modify the appearance of the HTML template using CSS.
- **valid_window** – Extends the validity to this many counter ticks before and after the current one. Increasing it is more convenient for users, but is less secure.

11.4 Tables

11.4.1 AuthenticatorSecret

This is required by *AuthenticatorProvider*.

To create this table, you can using Piccolo's migrations.

Add `piccolo_api.mfa.authenticator.piccolo_app` to `APP_REGISTRY` in `piccolo_conf.py`:

```
APP_REGISTRY = AppRegistry(
    apps=[
        "piccolo_api.mfa.authenticator.piccolo_app",
        ...
    ]
)
```

Then run the migrations:

```
piccolo migrations forwards mfa_authenticator
```

Alternatively, if not using Piccolo migrations, you can create the table manually:

```
>>> from piccolo_api.mfa.authenticator.table import AuthenticatorProvider
>>> AuthenticatorProvider.create_table().run_sync()
```

11.5 Full Example

Let's look at what an entire app looks like, which uses session auth, along with MFA (using the Authenticator provider).

11.5.1 Starlette

```
import os

from jinja2 import Environment, FileSystemLoader
from starlette.applications import Starlette
from starlette.endpoints import HTTPEndpoint
from starlette.middleware import Middleware
from starlette.middleware.authentication import AuthenticationMiddleware
from starlette.requests import Request
from starlette.responses import HTMLResponse, RedirectResponse
from starlette.routing import Mount, Route

from piccolo_api.csrf.middleware import CSRFMiddleware
from piccolo_api.encryption.providers import XChaCha20Provider
from piccolo_api.mfa.authenticator.provider import AuthenticatorProvider
from piccolo_api.mfa.endpoints import mfa_setup
from piccolo_api.register.endpoints import register
from piccolo_api.session_auth.endpoints import session_login, session_logout
from piccolo_api.session_auth.middleware import SessionsAuthBackend

EXAMPLE_DB_ENCRYPTION_KEY = b"W\x8b&E[\x8elr\xba\xb7\x19g\n\xd5`g\xea!Q#\x97\xcf\xed\
↪xdd+\xc7\x0e\xf7P\x82\xdf\x86"  # noqa: E501


environment = Environment(
    loader=FileSystemLoader(
        os.path.join(os.path.dirname(__file__), "templates"),
    ),
    autoescape=True,
)


class HomeEndpoint(HTTPEndpoint):
    async def get(self, request):
        home_template = environment.get_template("home.html")

        return HTMLResponse(content=home_template.render())


class PrivateEndpoint(HTTPEndpoint):
    async def get(self, request):
        return HTMLResponse(
```

(continues on next page)

(continued from previous page)

```
            content=(
                "<style>body{font-family: sans-serif;}</style>"
                "<h1>Private page</h1>"
            )
        )


def on_auth_error(request: Request, exc: Exception):
    return RedirectResponse("/login/")


private_app = Starlette(
    routes=[
        Route("/", PrivateEndpoint),
        Route("/logout/", session_logout(redirect_to="/")),
        Route(
            "/mfa-setup/",
            mfa_setup(
                provider=AuthenticatorProvider(
                    encryption_provider=XChaCha20Provider(
                        encryption_key=EXAMPLE_DB_ENCRYPTION_KEY
                    )
                )
            ),
        ),
    ],
    middleware=[
        Middleware(
            AuthenticationMiddleware,
            on_error=on_auth_error,
            backend=SessionsAuthBackend(admin_only=False),
        ),
    ],
    debug=True,
)


app = Starlette(
    routes=[
        Route("/", HomeEndpoint),
        Route(
            "/login/",
            session_login(
                mfa_providers=[
                    AuthenticatorProvider(
                        encryption_provider=XChaCha20Provider(
                            encryption_key=EXAMPLE_DB_ENCRYPTION_KEY
                        )
                    )
                ]
            ),
        ),
```

(continues on next page)

(continued from previous page)

```
        Route(
            "/register/",
            register(redirect_to="/login/", user_defaults={"active": True}),
        ),
        Mount("/private/", private_app),
    ],
    middleware=[
        Middleware(CSRFMiddleware, allow_form_param=True),
    ],
)
```

CHAPTER

TWELVE

TOKEN AUTH

12.1 Introduction

Token auth is a simple approach to authentication, which is most suitable for mobile apps and embedded systems.

Each user / client has a token generated for them. The token is just a random string - no information is embedded within it, as is the case with JWT.

When a client makes a request, the token needs to be added as a header. The user object associated with this token is then retrieved from a `TokenAuthProvider`. By default, this is a Piccolo table, but you can implement your own token provider if you so choose.

The token doesn't expire. It's suitable for mobile apps and other systems where tokens can be securely stored on the device. The client logic is simple to implement, as you don't have to worry about refreshing your token.

It's not recommended to use this type of authentication with web apps, because you can't securely store the token using JavaScript, which makes it susceptible to exposure using a XSS attack.

12.1.1 Header format

The client has to make a request which includes the `Authorization` HTTP header, with a value of `Bearer SOMETOKEN`.

12.2 Tables

We store the tokens in *TokenAuth* table, and the user credentials in `BaseUser` table.

12.2.1 Migrations

We recommend creating these tables using migrations.

You can add `piccolo_api.token_auth.piccolo_app` to the `apps` arguments of the `AppRegistry` in `piccolo_conf.py`.

```
APP_REGISTRY = AppRegistry(
    apps=[
        ...
        "piccolo_api.token_auth.piccolo_app",
        "piccolo.apps.user.piccolo_app",
        ...
```

(continues on next page)

(continued from previous page)

```
    ]
)
```

To learn more about Piccolo apps, see the Piccolo docs.

To run the migrations and create the tables, run:

```
piccolo migrations forwards user
piccolo migrations forwards token_auth
```

12.2.2 Creating them manually

If you prefer not to use migrations, and want to create them manually, you can do this instead:

```
from piccolo_api.token_auth.tables import TokenAuth
from piccolo.apps.user.tables import BaseUser
from piccolo.tables import create_tables

create_tables(BaseUser, TokenAuth, if_not_exists=True)
```

12.2.3 Source

TokenAuth

class piccolo_api.token_auth.tables.**TokenAuth**(*_data: dict[Column, Any] | None = None*, *_ignore_missing: bool = False*, *_exists_in_db: bool = False*, ***kwargs*)

Holds randomly generated tokens.

Useful for mobile authentication, IOT etc. Session auth is recommended for web usage.

async classmethod authenticate(*token: str*) → dict | None

classmethod authenticate_sync(*token: str*) → dict | None

async classmethod create_token(*user_id: int*, *one_per_user: bool = True*) → str

Create a new token.

Parameters

one_per_user – If True, a ValueError is raised if a token already exists for that user.

classmethod create_token_sync(*user_id: int*) → str

async classmethod get_user_id(*token: str*) → int | None

Returns the user_id if the given token is valid, otherwise None.

12.3 Endpoints

If using Piccolo as a backend, there is an endpoint for logging in, which will return a token.

12.3.1 token_login

This creates an endpoint for logging in, and getting a token.

```
from piccolo_api.token_auth.endpoints import token_login
from starlette import Starlette
from starlette.routing import Route, Router


app = Starlette(
    routes=[
        Route("/login/", token_login()),
    ]
)
```

For the user to login you **have to create a token for the user** in the `TokenAuth` table (the easiest way is to create a token in `Piccolo Admin`).

Usage

You can use any HTTP client to get the token. In our example we use `curl`.

To get a token:

```
curl -X POST \
    -H "Content-Type: application/json"
    -d '{"username": "piccolo", "password": "piccolo123"}' \
    http://localhost:8000/login/
```

To get data from a protected endpoint:

```
curl -H "Authorization: Bearer your-token" \
    http://localhost:8000/private/movies/
```

Hint

You can use all HTTP methods by passing a valid token to the `Authorization` header.

Source

class piccolo_api.token_auth.endpoints.**token_login**(*provider:* TokenProvider = *PiccoloTokenProvider()*)

Create an endpoint for logging using tokens.

Parameters

token_provider – Used to check if a token is valid.

class piccolo_api.token_auth.endpoints.**PiccoloTokenProvider**

Retrieves a token from a Piccolo table.

class piccolo_api.token_auth.endpoints.**TokenProvider**

Subclass this to provide your own custom token provider.

12.4 Middleware

The middleware builds upon Starlette's `AuthenticationMiddleware` (see the docs).

TokenAuthBackend is used to extract the token from the request. If the token is present and correct, then the request is accepted and the corresponding user is added to the scope, otherwise it is rejected.

`TokenAuthBackend` can work with several different *TokenAuthProvider* subclasses. The following are provided by default, but custom ones can be written by creating your own `TokenAuthProvider` subclasses.

12.4.1 SecretTokenAuthProvider

This provider checks whether the token provided by the client matches a list of predefined tokens.

```
from starlette.middleware.authentication import AuthenticationMiddleware

from piccolo_api.token_auth.middleware import (
    TokenAuthBackend,
    SecretTokenAuthProvider,
)

app = AuthenticationMiddleware(
    my_asgi_app,
    backend=TokenAuthBackend(SecretTokenAuthProvider(tokens=["abc123"])),
)
```

If successful a user called `secret_token_user` is added to the scope.

This provider is useful for protecting internal services, where the client is trusted not to leak the tokens.

It is also useful for protecting login endpoints when accessed from native apps. The client provides the token to be able to access the login endpoint, after which they obtain a unique token, which is used to authenticate with other endpoints.

12.4.2 PiccoloTokenAuthProvider

This provider checks a Piccolo database table for a corresponding token, and retrieves a matching user ID. It is the default provider.

```
from starlette.middleware.authentication import AuthenticationMiddleware

from piccolo_api.token_auth.middleware import (
    TokenAuthBackend,
    PiccoloTokenAuthProvider,
)

app = AuthenticationMiddleware(
    my_asgi_app,
    backend=TokenAuthBackend(PiccoloTokenAuthProvider()),
)
```

You'll have to run the migrations for this to work correctly.

12.4.3 TokenAuthBackend

excluded_paths

By default, the middleware protects all of the endpoints it is wrapping (i.e. if a token isn't present in the header then the request is rejected).

However, we may want to exclude certain endpoints - for example, if there's a Swagger docs endpoint, and allow access to them without a token. This is possible using `excluded_paths`.

Paths can be specified explicitly, or using wildcards:

```
TokenAuthBackend(
    PiccoloTokenAuthProvider(),
    excluded_paths=["/docs", "/openapi.json", "/foo/*"],
)
```

Note

In the above example `/foo/*` matches `/foo/`, `/foo/a`, `/foo/b`, `/foo/b/1` etc.

This is useful when using Swagger docs as they can be viewed in a browser, but they are still token protected.

FastAPI example

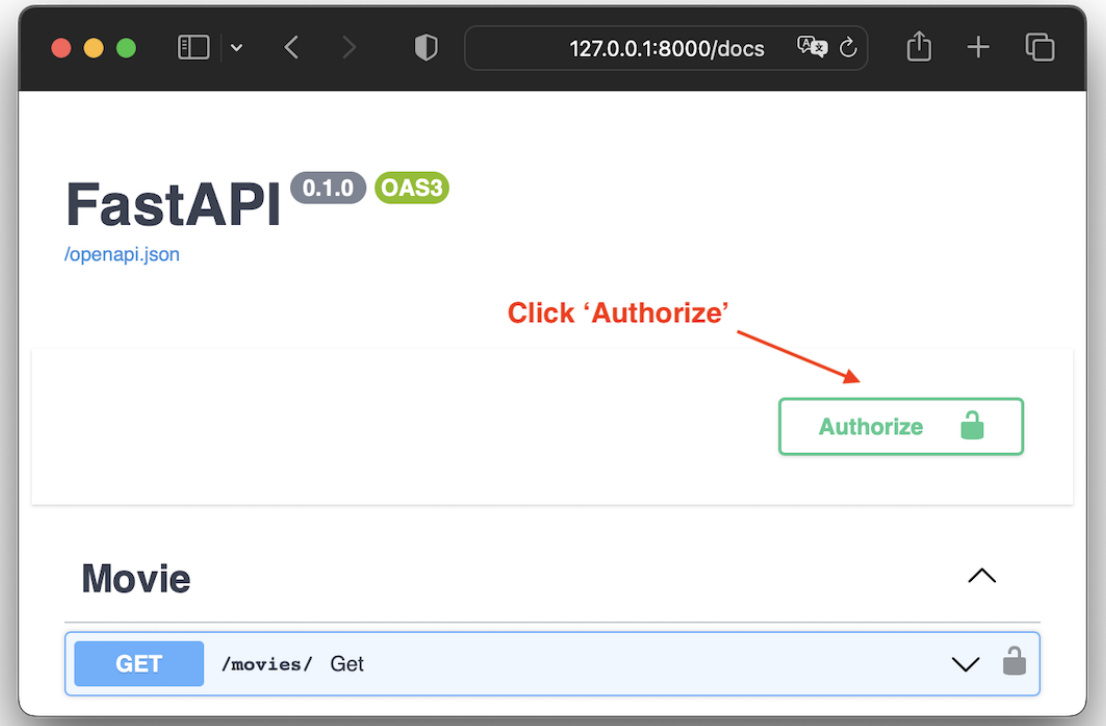
If we want to communicate with the API endpoints via the Swagger docs, we need to set FastAPI APIKeyHeader as a dependency. After that we can authorise the user with a valid token as in the example below.

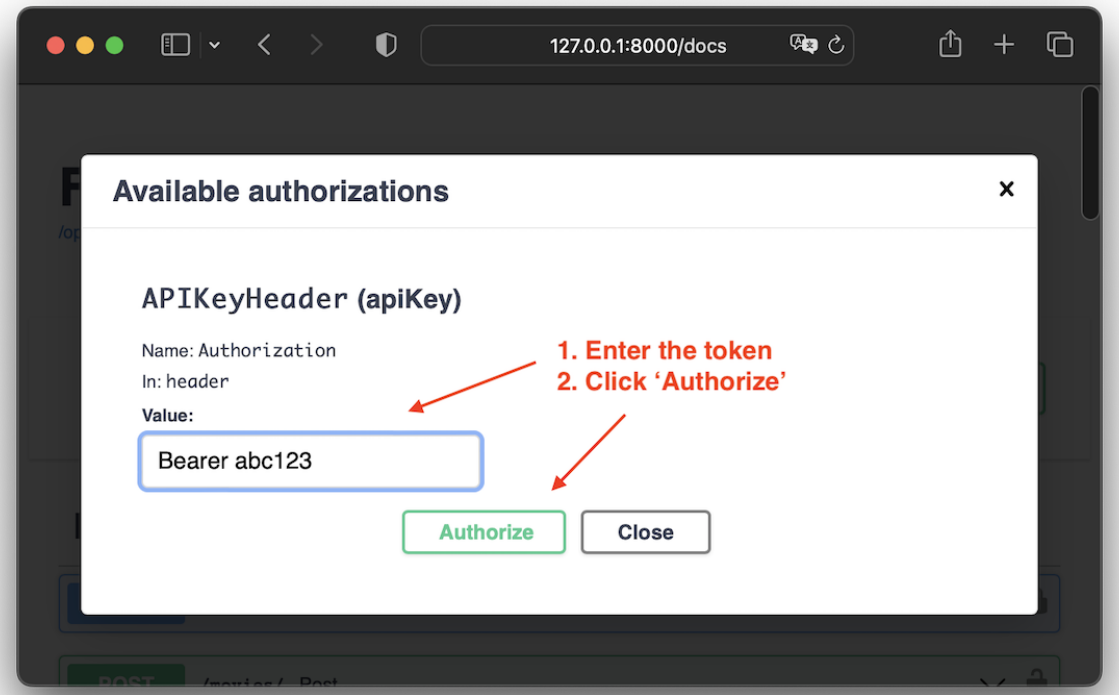
```
"""
An example usage of excluded_paths.
"""

from fastapi import Depends, FastAPI
from fastapi.middleware import Middleware
from fastapi.security.api_key import APIKeyHeader
from starlette.middleware.authentication import AuthenticationMiddleware
from tables import Movie  # An example Table

from piccolo_api.crud.endpoints import PiccoloCRUD
from piccolo_api.fastapi.endpoints import FastAPIKwargs, FastAPIWrapper
from piccolo_api.token_auth.middleware import (
    SecretTokenAuthProvider,
    TokenAuthBackend,
)

app = FastAPI(
    dependencies=[Depends(APIKeyHeader(name="Authorization"))],
    middleware=[
        Middleware(
            AuthenticationMiddleware,
            backend=TokenAuthBackend(
                SecretTokenAuthProvider(tokens=["abc123"]),
                excluded_paths=["/docs", "/openapi.json"],
            ),
```

(continues on next page)

(continued from previous page)

```
        )
    ],
)


# This is a quick way of building FastAPI endpoints using Piccolo, but isn't
# required:
FastAPIWrapper(
    "/movies/",
    fastapi_app=app,
    piccolo_crud=PiccoloCRUD(Movie, read_only=False),
    fastapi_kwargs=FastAPIKwargs(
        all_routes={"tags": ["Movie"]},
    ),
)
```





The user can then use the Swagger docs to interact with the API.

Note

The full source code is available on GitHub. Find the source for this documentation page, and look at the `examples` folder.

12.4.4 Source

class piccolo_api.token_auth.middleware.**TokenAuthBackend**(*token_auth_provider: TokenAuthProvider = DEFAULT_PROVIDER*, *excluded_paths: Sequence[str] | None = None*)

Parameters

- **token_auth_provider** – Used to verify that a token is correct.
- **excluded_paths** – These paths don't require a token - useful if you want to exclude a few URLs, such as docs.

class piccolo_api.token_auth.middleware.**TokenAuthProvider**

Subclass to create your own token provider.

class piccolo_api.token_auth.middleware.**PiccoloTokenAuthProvider**(*auth_table: type[BaseUser] = BaseUserTable*, *token_table: type[TokenAuth] = TokenAuth*)

Use this when the token is stored in a Piccolo database table.

class piccolo_api.token_auth.middleware.**SecretTokenAuthProvider**(*tokens: Sequence[str]*)

Checks that the token belongs to a predefined list of tokens. This is useful for very simple authentication use cases - such as internal microservices, where the client is trusted.

12.5 Full Example

Let's combine all of previous examples into a complete app.

12.5.1 FastAPI

```
from fastapi import FastAPI
from home.tables import Movie  # An example Table
from piccolo_admin.endpoints import create_admin
from starlette.middleware.authentication import AuthenticationMiddleware
from starlette.routing import Mount, Route

from piccolo_api.crud.endpoints import PiccoloCRUD
from piccolo_api.fastapi.endpoints import FastAPIKwargs, FastAPIWrapper
from piccolo_api.token_auth.endpoints import token_login
from piccolo_api.token_auth.middleware import (
    PiccoloTokenAuthProvider,
    TokenAuthBackend,
)
from piccolo_api.token_auth.tables import TokenAuth

public_app = FastAPI(
    routes=[
        Mount(
            "/admin/",
            create_admin(tables=[Movie, TokenAuth]),
        ),
        Route("/login/", token_login()),
    ],
)

private_app = FastAPI()

protected_app = AuthenticationMiddleware(
    private_app,
    backend=TokenAuthBackend(PiccoloTokenAuthProvider()),
)

FastAPIWrapper(
    "/movies/",
    fastapi_app=private_app,
    piccolo_crud=PiccoloCRUD(Movie, read_only=False),
    fastapi_kwargs=FastAPIKwargs(
        all_routes={"tags": ["Movie"]},
```

(continues on next page)

(continued from previous page)

```
    ),
)

public_app.mount("/private", protected_app)
```

12.5.2 Starlette

Is almost identical to the FastAPI example - just replace `FastAPI` with `Starlette`, and use Starlette's `HTTPEndpoint` for your endpoints. You also need to write your own crud endpoints because `Starlette` can't use `FastAPIWrapper`. An example is in SessionAuth.

CHAPTER

THIRTEEN

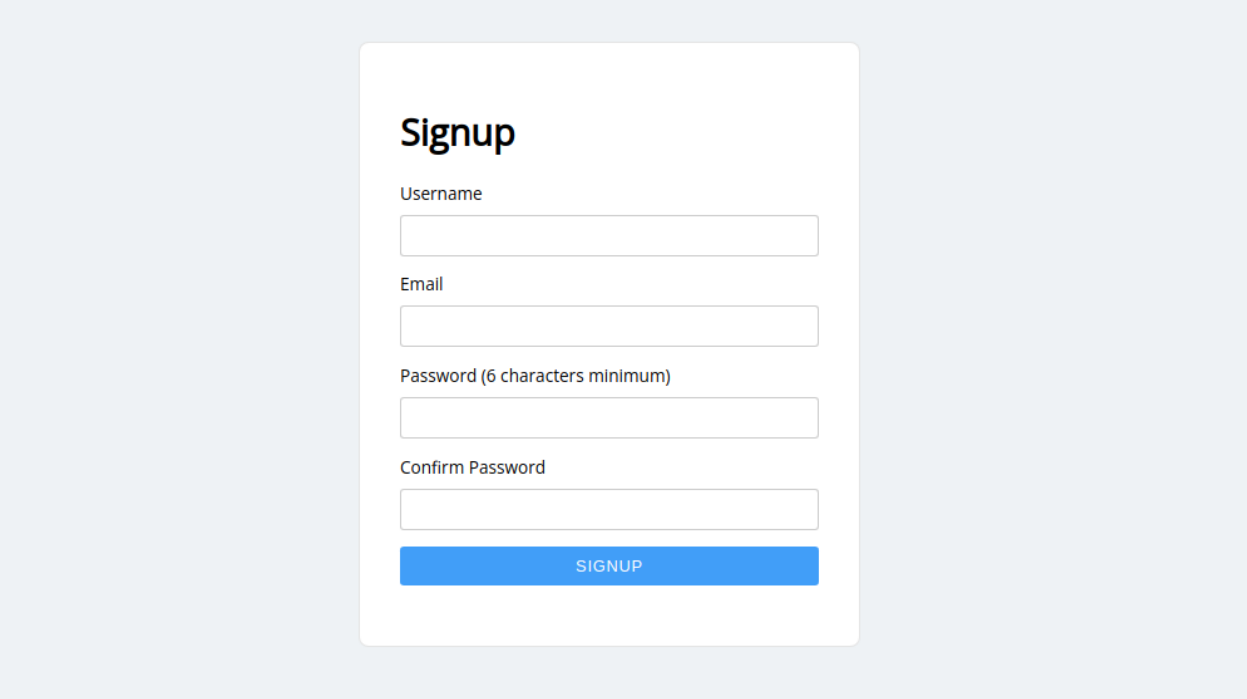
REGISTER

An important part of any application is being able to register new users.

13.1 Endpoints

13.1.1 register

An endpoint for registering a user. If you send a GET request to this endpoint, a simple registration form is rendered, where a user can manually sign up.



Hint

You can use a custom template, which matches the look and feel of your application. See the `template_path` parameter.

Alternatively, you can register a user programatically by sending a POST request to this endpoint (passing in `username`, `email`, `password` and `confirm_password` parameters as JSON, or as form data).

When registration is successful, the user can be redirected to a login endpoint. The destination can be configured using the `redirect_to` parameter.

Examples

Here's a Starlette example:

```
from piccolo_api.register.endpoints import register
from starlette import Starlette

app = Starlette()

app.mount("/register/", register(redirect_to="/login/"))
```

Here's a FastAPI example:

```
from fastapi import FastAPI
from piccolo_api.register.endpoints import register

app = FastAPI()

app.mount("/register/", register(redirect_to="/login/"))
```

Security

The endpoint is fairly simple, and works well for building a quick prototype, or internal application. If it's being used on the public internet, then extra precautions are required.

Rate limiting

One option is to apply rate limiting to this endpoint. This can be done using *RateLimitingMiddleware*. Modifying the FastAPI example above:

```
from fastapi import FastAPI
from piccolo_api.rate_limiting.middleware import (
    RateLimitingMiddleware, InMemoryLimitProvider
)
from piccolo_api.register.endpoints import register

app = FastAPI()

app.mount(
    "/register/",
    RateLimitingMiddleware(
        register(redirect_to="/login/"),
        InMemoryLimitProvider(
            timespan=3600,  # 1 hour
            limit=20,
            block_duration=86400  # 24 hours
        )
    )
)
```

We have used quite aggressive rate limiting here - there is no reason for a user to visit a registration page a large number of times.

CAPTCHA

Alternatively, we can easily integrate a CAPTCHA service. Sign up for an account with hCaptcha or reCAPTCHA, and then do the following:

```
from fastapi import FastAPI
from piccolo_api.register.endpoints import register
from piccolo_api.shared.auth.captcha import hcaptcha, recaptcha_v2

app = FastAPI()

# To use hCaptcha:
app.mount(
    "/register/",
    register(
        redirect_to="/login/",
        captcha=hcaptcha(
            site_key="my-site-key",
            secret_key="my-secret-key",
        )
    )
)

# To use reCAPTCHA:
app.mount(
    "/register/",
    register(
        redirect_to="/login/",
        captcha=recaptcha_v2(
            site_key="my-site-key",
            secret_key="my-secret-key",
        )
    )
)
```

For a complete example app, see here.

Building your own

There is no one-size-fits-all registration solution. You can use this endpoint as a basis for your own solution, which fits the needs of your application. For example, you can add extra registration fields.

Source

piccolo_api.register.endpoints.**register**(*auth_table: type[BaseUser] = BaseUser*, *redirect_to: str | URL = '/login/'*, *template_path: str | None = None*, *user_defaults: dict[str, Any] | None = None*, *captcha: Captcha | None = None*, *styles: Styles | None = None*, *read_only: bool = False*) → type[RegisterEndpoint]

An endpoint for register user.

Parameters

- **auth_table** – Which `Table` to create the user in. It defaults to `BaseUser`.
- **redirect_to** – Where to redirect to after successful registration.

- **template_path** – If you want to override the default register HTML template, you can do so by specifying the absolute path to a custom template. For example `'/some_directory/register.html'`. Refer to the default template at `piccolo_api/templates/register.html` as a basis for your custom template.
- **user_defaults** – These values are assigned to the new user. An example use case is setting `active = True` on each new user, so they can immediately login (not recommended for production, as it's better to verify their email address first, but OK for a prototype app):

```
register(user_defaults={'active': True})
```

- **captcha** – Integrate a CAPTCHA service, to provide protection against bots. See *Captcha*.
- **styles** – Modify the appearance of the HTML template using CSS.

Read_only

If `True`, the endpoint only responds to GET requests. It's not commonly needed, except when running demos.

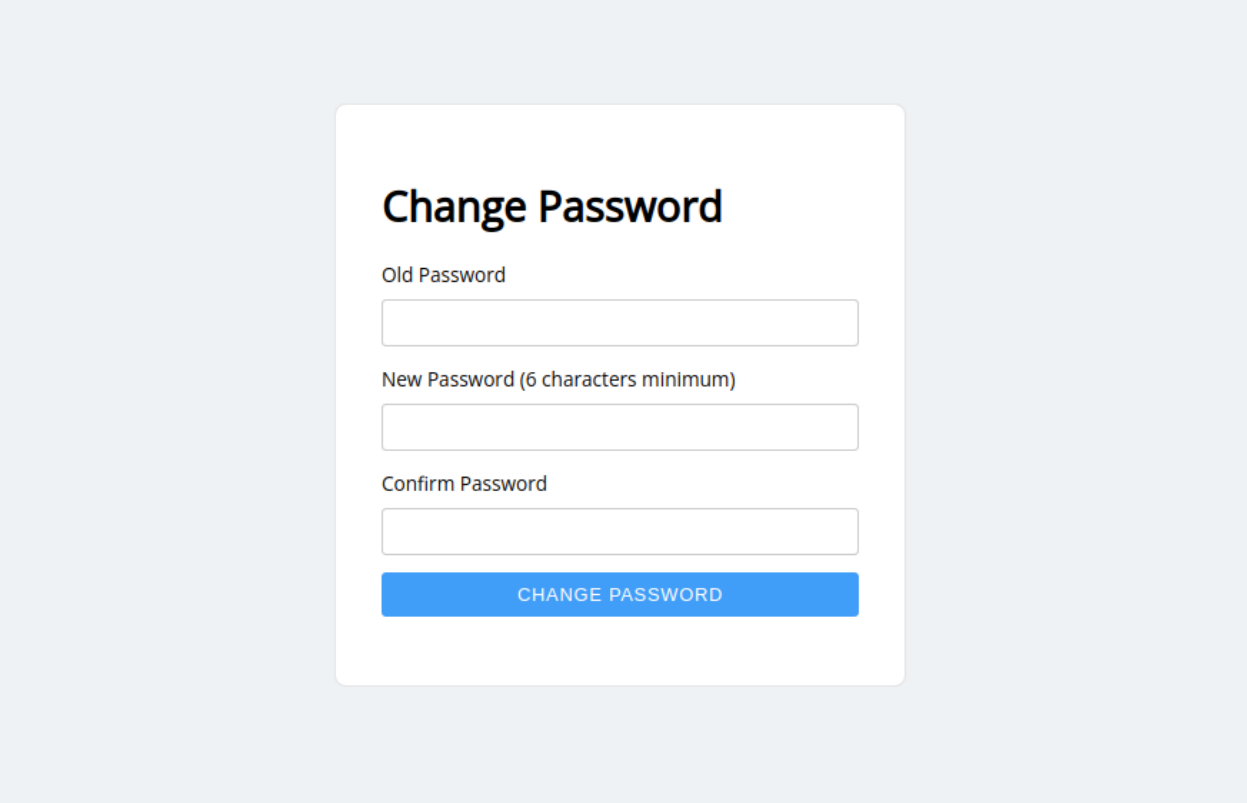
CHAPTER

FOURTEEN

CHANGE PASSWORD

14.1 change_password

An endpoint where the authenticated user can change their password. If you send a GET request to this endpoint, a simple form is shown in which the user can change their password manually.



Hint

You can use a custom template, which matches the look and feel of your application. See the `template_path` parameter. Or specify custom CSS styles using the `styles` parameter.

Alternatively, you can change the password programatically by sending a POST request to this endpoint (passing in `old_password`, `new_password` and `confirm_new_password` parameters as JSON, or as form data).

When the password change is successful, we invalidate the session cookie, and redirect the user to the login endpoint.

> **Warning**
>
> Only authenticated users can change their passwords!

14.1.1 Example

In this example we show how the endpoint integrates with a Starlette app using session auth.

For the complete source code, see the demo project on GitHub.

```
from starlette.applications import Starlette
from starlette.endpoints import HTTPEndpoint
from starlette.middleware import Middleware
from starlette.middleware.authentication import AuthenticationMiddleware
from starlette.requests import Request
from starlette.responses import HTMLResponse, RedirectResponse
from starlette.routing import Mount, Route

from piccolo_api.change_password.endpoints import change_password
from piccolo_api.csrf.middleware import CSRFMiddleware
from piccolo_api.register.endpoints import register
from piccolo_api.session_auth.endpoints import session_login
from piccolo_api.session_auth.middleware import SessionsAuthBackend


class HomeEndpoint(HTTPEndpoint):
    async def get(self, request):
        return HTMLResponse(
            content=(
                "<style>body{font-family: sans-serif;}</style>"
                "<h1>Change Password Demo</h1>"
                '<p>First <a href="/register/">register</a></p>'  # noqa: E501
                '<p>Then <a href="/login/">login</a></p>'  # noqa: E501
                '<p>Then try <a href="/private/change-password/">changing your password</
↪a></p>'  # noqa: E501
            )
        )


def on_auth_error(request: Request, exc: Exception):
    return RedirectResponse("/login/")


private_app = Starlette(
    routes=[
        Route(
            "/change-password/",
            change_password(),
        ),
    ],
    middleware=[
        Middleware(
            AuthenticationMiddleware,
```

(continues on next page)

(continued from previous page)

```
            on_error=on_auth_error,
            backend=SessionsAuthBackend(admin_only=False),
        ),
    ],
)


app = Starlette(
    routes=[
        Route("/", HomeEndpoint),
        Route("/login/", session_login()),
        Route(
            "/register/",
            register(redirect_to="/login/", user_defaults={"active": True}),
        ),
        Mount("/private/", private_app),
    ],
    middleware=[
        Middleware(CSRFMiddleware, allow_form_param=True),
    ],
)
```

If you want to use FastAPI instead, just make the following minor changes:

- Change the imports from `starlette` -> `fastapi`
- Change `Starlette` -> `FastAPI`

14.1.2 Source

piccolo_api.change_password.endpoints.**change_password**(*login_url: str = '/login/'*, *session_table: type[SessionsBase] | None = SessionsBase*, *session_cookie_name: str | None = 'id'*, *template_path: str | None = None*, *styles: Styles | None = None*, *read_only: bool = False*) → type[ChangePasswordEndpoint]

An endpoint for changing passwords.

Parameters

- **login_url** – Where to redirect the user to after successfully changing their password.
- **session_table** – If provided, when the password is changed, the sessions for the user will be invalidated in the database.
- **session_cookie_name** – If provided, when the password is changed, the session cookie with this name will be deleted.
- **template_path** – If you want to override the default change password HTML template, you can do so by specifying the absolute path to a custom template. For example `'/some_directory/change_password.html'`. Refer to the default template at `piccolo_api/templates/change_password.html` as a basis for your custom template.
- **styles** – Modify the appearance of the HTML template using CSS.

Read_only

If `True`, the endpoint only responds to GET requests. It's not commonly needed, except when

running demos.

CHAPTER

FIFTEEN

ADVANCED AUTH

15.1 Multiple Auth Backends

Sometimes you'll want to use multiple auth backends to protect the same endpoints. An example is using *Session Auth* for web users and *Token Auth* for mobile app users.

You can do this using `AuthenticationBackendJunction` which wraps multiple `AuthenticationBackend` subclasses, and tries each in turn. If none of them successfully authenticate, then authentication fails.

```
from piccolo_api.session_auth.middleware import SessionsAuthBackend
from piccolo_api.shared.auth.junction import AuthenticationBackendJunction
from piccolo_api.token_auth.middleware import (
    TokenAuthBackend,
    SecretTokenAuthProvider,
)
from starlette.middleware.authentication import AuthenticationMiddleware


app = AuthenticationMiddleware(
    my_asgi_app,
    backend=AuthenticationBackendJunction(
        backends=[
            SessionsAuthBackend(),
            TokenAuthBackend(
                SecretTokenAuthProvider(tokens=["abc123"])
            )
        ],
    )
)
```

CHAPTER

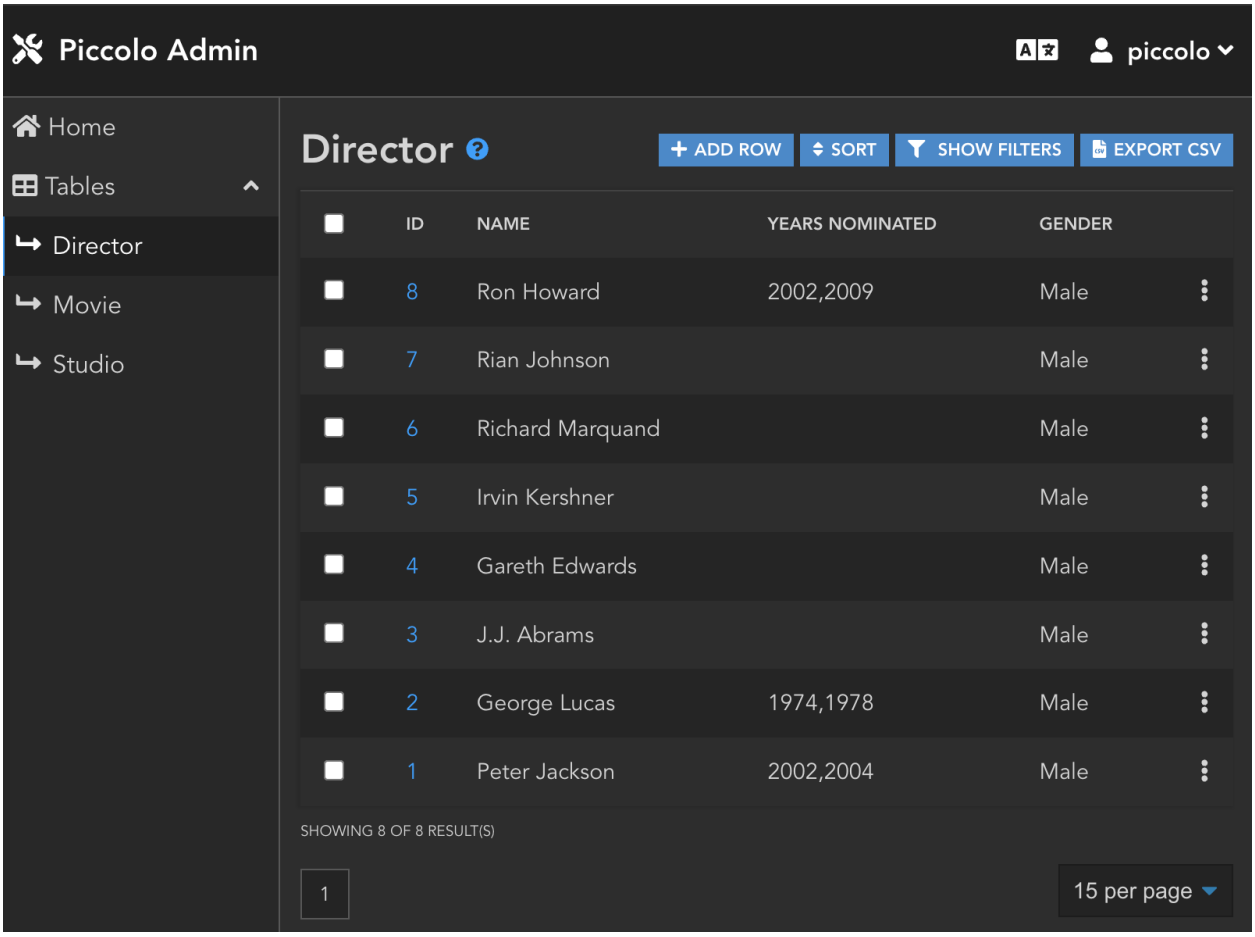
SIXTEEN

PICCOLO ADMIN

Piccolo Admin is a powerful web GUI which can be used to manage your data.

It's similar to tools like Django Admin and WordPress, but uses a rich Vue.js front end, with a modern REST backend.

Piccolo API powers Piccolo Admin. If you want to see how the technologies outlined here all work together, it's recommended to look at the Piccolo Admin source code.



CHAPTER

SEVENTEEN

API REFERENCE

17.1 LoginHooks

class piccolo_api.shared.auth.hooks.**LoginHooks**(*pre_login: list[TypeAliasForwardRef('PreLoginHook')] | None = None*, *login_success: list[TypeAliasForwardRef('LoginSuccessHook')] | None = None*, *login_failure: list[TypeAliasForwardRef('LoginFailureHook')] | None = None*)

Allows you to run custom logic during login. A hook can be a function or coroutine.

Here's an example using *session_login*:

```
def check_ban_list(username: str, **kwargs):
    '''
    An example `pre_login` hook.
    '''
    if username in ('nuisance', 'pest'):
        return 'This account has been temporarily suspended'.


async def log_success(username: str, user_id: int, **kwargs):
    '''
    An example `login_success` hook.
    '''
    await my_logging_service.record(
        f'{username} just logged in'
    )


async def log_failure(username: str, **kwargs):
    '''
    An example `login_failure` hook.
    '''
    await my_logging_service.record(f'{username} could not login')
    return (
        'To reset your password go <a href="/password-reset/">here</a>.'
    )


login_endpoint = session_login(
```

(continues on next page)

(continued from previous page)

```
    hooks=LoginHooks(
        pre_login=[check_ban_list],
        login_success=[log_success],
        login_failure=[log_failure],
    )
)
```

If any of the hooks return a string, the login process is aborted, and the login template is shown again, containing the string as a warning message. The string can contain HTML such as links, and it will be rendered correctly.

All of the example hooks above accept `**kwargs` - this is recommended just in case more data is passed to the hooks in future Piccolo API versions.

Parameters

- **pre_login** – A list of function and / or coroutines, which accept the username as a string.
- **login_success** – A list of function and / or coroutines, which accept the username as a string, and the user ID as an integer. If a string is returned, the login process stops before a session is created.
- **login_failure** – A list of function and / or coroutines, which accept the username as a string.

17.2 OrderBy

class piccolo_api.crud.endpoints.**OrderBy**(*column: Column*, *ascending: bool = True*)

17.3 CAPTCHA

class piccolo_api.shared.auth.captcha.**Captcha**(*form_html: str*, *token_field: str*, *validator: Callable[[str], str | None] | Callable[[str], Awaitable[str | None]]*)

Used to create CAPTCHAs for adding bot protection to endpoints. This is a generic class for implementing your own CAPTCHA. Out of the box support is provided for *hcaptcha* and *recaptcha_v2*, so you should only need to use this class directly if doing something custom.

Parameters

- **form_html** – Any HTML which needs inserting into the form to make the CAPTCHA work.
- **validator** – A callback (either an async or normal function), which is passed the CAPTCHA token, and is used to verify with the CAPTCHA provider's API that the token is valid. To indicate that validation has failed, return a string containing an error message which will be shown to the user.

piccolo_api.shared.auth.captcha.**hcaptcha**(*site_key: str*, *secret_key: str*) → *Captcha*

Can be used along with Piccolo endpoints to incorporate hCaptcha.

Parameters

- **site_key** – Provided by hCaptcha.
- **secret_key** – Provided by hCaptcha.

piccolo_api.shared.auth.captcha.**recaptcha_v2**(*site_key: str*, *secret_key: str*) → *Captcha*

Can be used along with Piccolo endpoints to incorporate reCAPTCHA.

Parameters

- **site_key** – Provided by reCAPTCHA.
- **secret_key** – Provided by reCAPTCHA.

17.4 Styles

class piccolo_api.shared.auth.styles.**Styles**(*background_color: str = '#EEF2F5'*, *foreground_color: str = 'white'*, *text_color: str = 'black'*, *error_text_color: str = 'red'*, *button_color: str = '#419EF8'*, *button_text_color: str = 'white'*, *link_color: str = '#419EF8'*, *border_color: str = 'rgba(0, 0, 0, 0.2)'*)

Used to set CSS styles in endpoints such as `session_login`, `session_logout`, and `register`.

Each of the values must be valid CSS.

17.5 Media

class piccolo_api.media.local.**LocalMediaStorage**(*column: Text | Varchar | Array*, *media_path: str*, *executor: Executor | None = None*, *allowed_extensions: Sequence[str] | None = ALLOWED_EXTENSIONS*, *allowed_characters: Sequence[str] | None = ALLOWED_CHARACTERS*, *file_permissions: int | None = 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.

class piccolo_api.media.s3.**S3MediaStorage**(*column: Text | Varchar | Array*, *bucket_name: str*, *folder_name: str | None = None*, *connection_kwargs: dict[str, Any] | None = None*, *sign_urls: bool = True*, *signed_url_expiry: int = 3600*, *upload_metadata: dict[str, Any] | None = None*, *executor: Executor | None = None*, *allowed_extensions: Sequence[str] | None = ALLOWED_EXTENSIONS*, *allowed_characters: Sequence[str] | None = 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(
    ...,
    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(
    ...,
```

(continues on next page)

(continued from previous page)

```
    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.

CHAPTER

EIGHTEEN

CONTRIBUTING

If you want to explore the interior of Piccolo API more deeply and help with the project, follow these instructions.

18.1 Get the tests running

- Create a new virtualenv
- Clone the Git repo
- `cd piccolo_api`
- Install default dependencies: `pip install -r requirements/requirements.txt`
- Install development dependencies: `pip install -r requirements/dev-requirements.txt`
- Install test dependencies: `pip install -r requirements/test-requirements.txt`
- Setup Postgres
- Run the automated code linting/formatting tools: `./scripts/lint.sh`
- Run the test suite with Postgres: `./scripts/test-postgres.sh`
- Run the test suite with Sqlite: `./scripts/test-sqlite.sh`

18.2 Contributing to the docs

The docs are written using Sphinx. To get them running locally:

- Install the requirements: `pip install -r requirements/doc-requirements.txt`
- `cd docs`
- Do an initial build of the docs: `make html`
- `cd ..`
- Serve the docs: `./scripts/run-docs.sh`
- The docs will auto rebuild as you make changes.

18.3 Code style

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.linting.mypyEnabled": true,
    "python.formatting.provider": "black",
    "python.formatting.blackArgs": [
        "--line-length",
        "79"
    ],
    "editor.formatOnSave": true
}
```

Type hints are used throughout the project.

CHAPTER

NINETEEN

CHANGES

19.1 1.11.0

Handle check constraint errors gracefully.

Bumped `cryptography` dependency (used for MFA).

19.2 1.10.0

Bumped `cryptography` dependency (used for MFA), and made the `SessionsBase.token` column `secret=True`.

19.3 1.9.0

Bumped `cryptography` dependency (used for MFA).

19.4 1.8.0

Bumped `pynacl` dependency (used for MFA).

Officially support Python 3.14 (thanks to @sinisaos for this).

19.5 1.7.0

Modernised the type hints across the codebase (e.g. replacing `typing.List` with `list`). Thanks to @sinisaos for this.

Added a missing import to the `jwt` code example. Thanks to @andrewshaodev for this.

19.6 1.6.0

Improved HTTP error codes in JWT middleware - thanks to @sinisaos for this.

Added official Python 3.13 support, and dropped Python 3.8 - thanks to @sinisaos for this.

Updated dependencies - thanks to @sinisaos for this.

Fixed type annotations for `TokenAuth`.

Improved docs for MFA - passing in encryption keys using environment variables.

19.7 1.5.2

Fix an import from *pynacl* - it's an optional dependency.

19.8 1.5.1

Fixed optional requirements for MFA - thanks to @sinisaos for reporting this issue.

19.9 1.5.0

Added support for Multi Factor Authentication, using an authenticator app.

This will soon be implemented in Piccolo Admin.

Huge thanks to @Skelmis and @sinisaos for their help with this.

19.10 1.4.1

Fix for a breaking change in Pydantic 2.8.0.

Upgraded setuptools.

19.11 1.4.0

Added `excluded_paths` argument to `SessionsAuthBackend`.

19.12 1.3.1

Fixed typo in `CSPMiddleware` (thanks to @Skelmis for fixing this).

19.13 1.3.0

`default-src` can now be set in `CSPMiddleware`.

19.14 1.2.0

Fixed bugs with filtering `Email` and multi-dimensional `Array` columns.

Fixed bugs with bulk updating the `BaseUser` table.

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

19.15 1.1.0

Added Python 3.12 support.

Added `ne` (not equals) operator to `PiccoloCRUD`.

19.16 1.0.0

Works with Piccolo v1, Piccolo Admin v1, and Pydantic v2.

19.17 1.0a3

Using the new `json_schema_extra` argument for `create_pydantic_model`.

19.18 1.0a2

Fixed a bug with extracting the type from an optional type. Thanks to @sinisaos for discovering this issue.

19.19 1.0a1

Pydantic v2 support - many thanks to @sinisaos for this.

19.20 0.58.0

Upgraded the version of Swagger UI used in the `swagger_ui` endpoint (see `piccolo_api.openapi.endpoints.swagger_ui`). Thanks to @sinisaos for this.

19.21 0.57.0

`PiccoloCRUD` now handles foreign key violation errors gracefully.

For example, if we have tables like this:

```
class Director(Table):
    name = Varchar()

class Movie(Table):
    name = Varchar()
    director = ForeignKey(Director, on_delete=OnDelete.restrict)
```

The `ON DELETE RESTRICT` constraint means we're not allowed to delete a director if a movie has a foreign key to it.

We now get a `422` error response, with an error message which we can display in Piccolo Admin.

Support for Python 3.7 has also been dropped, as it's end of life.

19.22 0.56.0

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.

19.23 0.55.0

Added the `excluded_paths` argument to `TokenAuthBackend`. This means you can wrap an entire ASGI application in this middleware, and exclude certain paths, such as the Swagger docs. Thanks to @sinisaos for this.

```
app = FastAPI(
    dependencies=[Depends(APIKeyHeader(name="Authorization"))],
    middleware=[
        Middleware(
            AuthenticationMiddleware,
            backend=TokenAuthBackend(
                SecretTokenAuthProvider(tokens=["abc123"]),
                excluded_paths=["/docs", "/openapi.json"],  # <- Note
            ),
        )
    ],
)
```

19.24 0.54.0

Added `allow_unauthenticated` option to `JWTMiddleware`.

By default, `JWTMiddleware` rejects any request with an invalid JWT token, but with this option we allow the user to reject the request instead within their endpoints.

19.25 0.53.0

Added `token_login` endpoint, which is more convenient than `TokenAuthLoginEndpoint`.

Improved the docs for token auth and JWT auth (thanks to @sinisaos).

Modified the `OrderBy` class, to add some functionality needed by Piccolo Admin.

19.26 0.52.0

`PiccoloCRUD` now lets you specify multiple columns in the `__order` GET param.

For example, with this schema:

```
class Movie(Table):
    name = Varchar()
    rating = Integer()
```

To order the results by descending `rating` and ascending `name`:

```
GET /?__order=-rating,name
```

19.27 0.51.0

You can now get all rows with a null / not-null value in `PiccoloCRUD`.

For example, if we have a nullable column called `score`:

```
GET /?score__operator=is_null
```

Likewise, to get all rows whose score is not null:

```
GET /?score__operator=not_null
```

19.28 0.50.0

Catching more database errors in `PiccoloCRUD`, and returning useful API responses instead of 500 errors.

Implemented GitHub's CodeQL suggestions - this now means `LocalMediaStorage` uses `600` instead of `640` as the default file permissions for uploaded files (thanks to @sinisaos for this).

19.29 0.49.0

- Added Python 3.11 support.
- `PiccoloCRUD` validators can now be async.
- Improved logging.
- The minimum version of FastAPI is now `0.87.0`. The reason for this is Starlette made a fairly large change in version `0.21.0`, which meant we had to refactor a lot of our tests, which makes it challenging to support older versions.

19.30 0.48.1

Improving type annotations:

- Adding `id: Serial` for `SessionsBase` and `TokenAuth`.
- Fixed type annotations for latest version of Starlette (thanks to @sinisaos for this).

19.31 0.48.0

If `BaseUser` is used with `PiccoloCRUD`, passwords are handled properly. Thanks to @sinisaos for making this change.

19.32 0.47.0

`PiccoloCRUD` now handles database exceptions better. If a query fails due to a unique constraint, a 422 response code is returned, along with information about the error.

This means Piccolo Admin will show more useful debugging information when a query fails.

Thanks to @ethagnawl for reporting this issue, and @sinisaos for help prototyping a solution.

19.33 0.46.0

Fixed a bug with `Email` columns and `PiccoloCRUD.get_new`. Thanks to @Tar8117 for reporting this bug.

19.34 0.45.0

Previously you had to provide `folder_name` as an argument to `S3MediaStorage`.

It's now optional, as some users may choose to store their files in a bucket without a folder.

19.35 0.44.0

When uploading files to S3, we try and correctly set the content type. This now works correctly for `.jpg` files (previously only `.jpeg` worked for JPEGs). Thanks to @sumitsharansatsangi for adding this.

19.36 0.43.0

Fixed a bug with `MediaStorage.delete_unused_files` - it was raising an exception when used with `Array` columns. Thanks to @sumitsharansatsangi for reporting this issue.

When using `S3MediaStorage` you can now specify additional arguments when files are uploaded (using the `upload_metadata` argument), for example, setting the cache settings, and much more. Thanks to @sumitsharansatsangi, and @sinisaos for help reviewing.

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

19.37 0.42.0

Added dependency injection to `PiccoloCrud` hooks - the Starlette request object will now be passed in if requested. For example:

```
def my_hook(row_id, request):
  ...
```

Thanks to @AnthonyArmour and @destos for this.

19.38 0.41.0

Added support for file storage in a local folder and in S3. This was added for Piccolo Admin, but is useful for all Piccolo apps. Thanks to @sinisaos for assisting with this.

19.39 0.40.0

Make Piccolo API work with Piccolo >= 0.82.0. `Table` used to accept a parameter called `ignore_missing`. This was renamed to `_ignore_missing`. Thanks to @sinisaos for this fix.

19.40 0.39.0

Improved the HTTP status codes returned by the `change_password`, `register` and `session_login` endpoints. They now return a 422 status code if a validation error occurs. This is required by Piccolo Admin, to better determine why a request failed.

19.41 0.38.0

Added `read_only` option to `change_password` and `register` endpoints.

This is for Piccolo Admin's `read_only` mode.

19.42 0.37.2

Changed a parameter name used in the `change_password` endpoint to be less ambiguous (`old_password` -> `current_password`).

19.43 0.37.1

Changed a parameter name used in the `change_password` endpoint to be less ambiguous (`confirm_password` -> `confirm_new_password`).

19.44 0.37.0

Added a `change_password` endpoint (courtesy @sinisaos).

See the demo project for a full example.

19.45 0.36.0

The `session_login`, `session_logout`, and `register` endpoints can now have their CSS styles easily customised, to make them match the rest of the application.

```
from fastapi import FastAPI
from piccolo_api.session_auth.endpoints import register
from piccolo_api.shared.auth.styles import Styles

app = FastAPI()

app.mount(
    '/register/',
    register(
        styles=Styles(background_color='black')
    )
)
```

19.46 0.35.0

It is now trivially easy to add CAPTCHA support to the `register` and `session_login` endpoints, to provide protection against bots. Just sign up for an account with hCaptcha or reCAPTCHA, and do the following:

```
from fastapi import FastAPI
from piccolo_api.session_auth.endpoints import register
from piccolo_api.shared.auth.captcha import hcaptcha

app = FastAPI()

# To use hCaptcha:
app.mount(
    '/register/',
    register(
        captcha=hcaptcha(
            site_key='my-site-key',
            secret_key='my-secret-key',
        )
    )
)
```

19.47 0.34.0

Added a `register` endpoint, which is great for quickly prototyping a sign up process (courtesy @sinisaos).

Added hooks to the `session_login` endpoint, allowing additional logic to be triggered before and after login.

19.48 0.33.1

Fixing the `ids` endpoint of `PiccoloCRUD` when a custom primary key column is used with a name other than `id`.

19.49 0.33.0

The schema endpoint of `PiccoloCRUD` now returns the primary key name. This means we'll be able to support tables with a custom primary key name in Piccolo Admin.

19.50 0.32.3

Make sure tables with a custom primary key column work with `PiccoloCRUD`.

19.51 0.32.2

Fixed a bug with `PiccoloCRUD`, where a PATCH request returned a string instead of a JSON object. Thanks to @trondhindenes for discovering and fixing this issue.

19.52 0.32.1

Fixed bug with `__range_header=false`.

19.53 0.32.0

Added support for the `Content-Range` HTTP header in the GET endpoint of `PiccoloCRUD`. This means the API client can fetch the number of available rows, without doing a separate API call to the `count` endpoint.

```
GET /?__range_header=true
```

If the page size is 10, then the response header then looks something like:

```
Content-Range: movie 0-9/100
```

The feature was created to make Piccolo APIs work better with front ends like React Admin.

Thanks to @trondhindenes for adding this feature, and @sinisaos for help reviewing.

19.54 0.31.0

Added hooks to `PiccoloCRUD`. This allows the user to add their own logic before a save / patch / delete (courtesy @trondhindenes).

For example:

```
# Normal functions and async functions are supported:
def pre_save_hook(movie):
    movie.rating = 90
    return movie

PiccoloCRUD(
    table=Movie,
    read_only=False,
    hooks=[
        Hook(hook_type=HookType.pre_save, callable=pre_save_hook)
    ]
)
```

19.55 0.30.1

- Streamlined the `CSRFMiddleware` code, and added missing type annotations.
- If using the `__visible_fields` parameter with `PiccoloCRUD`, and the field name is unrecognised, the error response will list the correct field names.
- Improved test coverage (courtesy @sinisaos).

19.56 0.30.0

We recently added the `__visible_fields` GET parameter to `PiccoloCRUD`, which allows the user to determine which fields are returned by the API.

However, there was no way of the user knowing which fields were supported. This is now possible by visiting the `/schema` endpoint, which has a `visible_fields_options` field which lists the columns available on the table and related tables (courtesy @sinisaos).

19.57 0.29.2

Fixed a bug with the OpenAPI docs when using `Array` columns. Thanks to @gmos for reporting this issue, and @sinisaos for fixing it.

19.58 0.29.1

The `__visible_fields` filter on `PiccoloCRUD` now works on the detail endpoint (courtesy @sinisaos). For example:

```
GET /1/?__visible_fields=id,name,director.name
```

We also modified a type annotation in `FastAPIWrapper`, so you can use it with FastAPI's `APIRouter` without getting a type warning. Thanks to @gmos for reporting this issue.

19.59 0.29.0

Added a `__visible_fields` filter to `PiccoloCRUD`. It's a very powerful feature which lets us specify which fields we want the API to return from a GET request (courtesy @sinisaos).

It can even support joins, but we must supply a `max_joins` parameter:

```
app = PiccoloCRUD(Movie, max_joins=1)
uvicorn(app)
```

Then we can do:

```
GET /?__visible_fields=id,name,director.name
```

Which will return:

```
{
  "rows": [
      {
          "id": 17,
          "name": "The Hobbit: The Battle of the Five Armies",
          "director": {
              "name": "Peter Jackson"
          }
      },
      ...
  ]
}
```

By specifying exactly which data we want returned, it is much more efficient, especially when fetching large numbers of rows, or with tables with lots of columns.

19.60 0.28.1

Fixed a bug with the delete endpoint of `PiccoloCRUD`. It was returning a 204 response with a body (this isn't allowed, and could cause an exception to be raised in the web server). Thanks to @trondhindenes for reporting this issue.

Updated Swagger UI to the latest version.

19.61 0.28.0

Modified the `get_ids` endpoint of `PiccoloCRUD`, so it accepts an `offset` query parameter. It already supported `limit`.

19.62 0.27.0

You can now pass a `schema_extra` argument to `PiccoloCRUD`, which is added to the underlying Pydantic schema.

19.63 0.26.0

`create_pydantic_model` is now imported from the main Piccolo repo.

19.64 0.25.1

- Added examples to CSRF docs (courtesy @sinisaos).
- Improved `SessionAuthBackend` - it was too aggressive at rejecting requests when `allow_unauthenticated=True` (thanks to @Bakz for reporting this).

19.65 0.25.0

If you send a GET request to the `session_logout` endpoint, it will now render a simple logout form. This makes it work much nicer out of the box. Thanks to @sinisaos for adding this.

19.66 0.24.1

When using the `nested` argument in ``create_pydantic_model``, more of the other arguments are passed to the nested models. For example, if `include_default_columns` is `True`, both the parent and child models will include their default columns.

19.67 0.24.0

Added support for nested models in `create_pydantic_model`. For each `ForeignKey` in the Piccolo table, the Pydantic model will contain a sub model for the related table.

For example:

```
class Manager(Table):
    name = Varchar()

class Band(Table):
    name = Varchar()
    manager = ForeignKey(Manager)

BandModel = create_pydantic_model(Band, nested=True)
```

If we were to write `BandModel` by hand instead, it would look like this:

```
class ManagerModel(BaseModel):
    name: str

class BandModel(BaseModel):
    name: str
    manager: ManagerModel
```

This feature is designed to work with the new `nested` output option in Piccolo >= 0.40.0, which returns the data in the correct format to pass directly to the nested Pydantic model.

```
band = Band.select(
    Band.id,
    Band.name,
    *Band.manager.all_columns()
).first(
).output(
    nested=True
).run_sync()
>>> print(band)
{'id': 1, 'name': 'Pythonistas', 'manager': {'id': 1, 'name': 'Guido'}}

BandModel(**band)
```

Courtesy @aminalaee.

19.68 0.23.1

Make sure `asyncpg` gets installed, as Piccolo API currently has a hard requirement on it (we hope to fix this in the future).

19.69 0.23.0

- Fixed MyPy errors (courtesy @sinisaos).
- Simplification of JWT authentication - it no longer needlessly checks expiry, as PyJWT already does this (courtesy @aminalaee).
- Substantial increase in code coverage (courtesy @aminalaee and @sinisaos).
- Increased the minimum PyJWT version, as versions > 2.0.0 return the JWT as a string instead of bytes.
- Added an option to exclude columns when using `create_pydantic_model` (courtesy @kucera-lukas).

19.70 0.22.0

Updating `PiccoloCRUD` so it works better with the custom primary key feature added in Piccolo.

19.71 0.21.1

Minor changes to the custom login template logic. More complex Jinja templates are now supported (which are extended from other Jinja templates).

19.72 0.21.0

Session auth improvements:

- The default login template is much nicer now.
- The login template can be overridden with a custom one, to match the look and feel of the application.
- The `session_logout` endpoint can now redirect after successfully logging out.

19.73 0.20.0

When using the `swagger_ui` endpoint, the title can now be customised - courtesy @heliumbrain.

19.74 0.19.0

- Added an `allow_unauthenticated` option to `SessionsAuthBackend`, which will add an `UnauthenticatedUser` to the scope, instead of rejecting the request. The app's endpoints are then responsible for checking `request.user.is_authenticated`.
- Improved the docs for Session Auth.
- If `deserialize_json` is False on `create_pydantic_model`, it will still provide some JSON validation.

19.75 0.18.0

Added a `deserialize_json` option to `create_pydantic_model`, which will convert JSON strings to objects - courtesy @heliumbrain.

19.76 0.17.1

Added the OAuth redirect endpoint to `swagger_ui`.

19.77 0.17.0

Added a `swagger_ui` endpoint which works with Piccolo's `CSRFMiddleware`.

19.78 0.16.0

Modified the auth middleware to add the Piccolo `BaseUser` instance for the authenticated user to Starlette's `BaseUser`.

19.79 0.15.1

Add missing *login.html* template.

19.80 0.15.0

Added support for `choices` argument in Piccolo `Column` instances. The choices are output in the schema endpoint of `PiccoloCRUD`.

19.81 0.14.1

Added `validators` and `exclude_secrets` arguments to `PiccoloCRUD`.

19.82 0.14.0

Added `superuser_only` and `active_only` options to `SessionsAuthBackend`.

19.83 0.13.0

Added support for `Array` column types.

19.84 0.12.13

Added `py.typed` file, for MyPy.

19.85 0.12.12

Exposing the `help_text` value for `Table` in the Pydantic schema.

19.86 0.12.11

Exposing the `help_text` value for `Column` in the Pydantic schema.

19.87 0.12.10

Fixing a bug with `ids` endpoint when there's a limit but no search.

19.88 0.12.9

Fixing `ids` endpoint in `PiccoloCRUD` with Postgres - search wasn't working.

19.89 0.12.8

The `ids` endpoint in `PiccoloCRUD` now accepts a limit parameter.

19.90 0.12.7

Added additional validation to Pydantic serialisers - for example, `Varchar` max length, and `Decimal` / `Numeric` precision and scale.

19.91 0.12.6

The `ids` endpoint in `PiccoloCRUD` is now searchable.

19.92 0.12.5

Added missing `new` endpoint to `FastAPIWrapper` - courtesy @sinisaos.

19.93 0.12.4

Made FastAPI a requirements, instead of an optional requirement.

19.94 0.12.3

- Added ids and references endpoints to `FastAPIWrapper`.
- Increase compatibility of `SessionLoginEndpoint` and `CSRFMiddleware` - adding a CSRF token as a form field should now work.

19.95 0.12.2

- Added docstrings to FastAPI endpoints in `FastAPIWrapper`.
- Exposing count and schema endpoints in `FastAPIWrapper`.

19.96 0.12.1

- Added docs for `__page` and `__page_size` query parameters for `PiccoloCRUD`.
- Implemented `max_page_size` to prevent excessive server load - courtesy @sinisaos.

19.97 0.12.0

Renaming migrations which were problematic for Windows users.

19.98 0.11.4

Using Pydantic to serialise the `PiccoloCRUD.new` response. Fixes a bug with serialising some values, such as `decimal.Decimal`.

19.99 0.11.3

- Using Piccolo's `run_sync` instead of asgiref.
- Loosened dependencies.
- `create_pydantic_model` now supports lazy references in `ForeignKey` columns.
- MyPy fixes.

19.100 0.11.2

- `PiccoloCRUD` now supports the *__readable* query parameter for detail endpoints - i.e. */api/movie/1/?__readable=true*. Thanks to sinisaos for the initial prototype.
- Improving type hints.

19.101 0.11.1

Bumped requirements.

19.102 0.11.0

Using `Column._meta.required` for Pydantic schema.

19.103 0.10.1

Can pass more configuration options to FastAPI via `FastAPIWrapper`.

19.104 0.10.0

Updated for Piccolo 0.12.

19.105 0.9.2

- Added `FastAPIWrapper`, which makes building a FastAPI endpoint really simple.
- Improved the handling of malformed queries better in `PiccoloCRUD` - catching unrecognised column names, and returning a 400 response.

19.106 0.9.1

`create_pydantic_model` now accepts an optional *model_name* argument.

19.107 0.9.0

Bumped requirements, to support Piccolo `Numeric` and `Real` column types.

19.108 0.8.0

Improved session auth - can increase the expiry automatically, which improves the user experience.

19.109 0.7.6

Can choose to not redirect after a successful session auth login - this is preferred when calling the endpoint via AJAX.

19.110 0.7.5

Loosening requirements for Piccolo projects.

19.111 0.7.4

Bumped requirements.

19.112 0.7.3

Bumped requirements.

19.113 0.7.2

Can configure where `CSRFMiddleware` looks for tokens, and bug fixes.

19.114 0.7.1

CSRF tokens can now be passed as form values.

19.115 0.7.0

Supporting Piccolo 0.10.0.

19.116 0.6.1

Adding missing __init__.py file - was messing up release.

19.117 0.6.0

New style migrations.

19.118 0.5.1

Added support for PATCH queries, and specifying text filter types, to `PiccoloCRUD`.

19.119 0.5.0

Changed schema format.

19.120 0.4.4

`PiccoloCRUD` 'new' endpoint works in readonly mode - doesn't save any data.

19.121 0.4.3

Supporting order by, pagination, and filter operators in `PiccoloCRUD`.

19.122 0.4.2

Added 'new' endpoint to `PiccoloCRUD`.

19.123 0.4.1

Added missing `__init__.py` files.

19.124 0.4.0

Added token auth and rate limiting middleware.

19.125 0.3.2

Updated Piccolo import paths.

19.126 0.3.1

Updated Piccolo syntax.

19.127 0.3.0

Improved code layout.

19.128 0.2.0

Updating to work with Piccolo > 0.5.

19.129 0.1.3

Added validation to PUT requests.

19.130 0.1.2

Added foreign key support to schema.

19.131 0.1.1

Changed import paths.

PYTHON MODULE INDEX

p

INDEX

L

M

O

P

R

S

T

U

V

X