Securing APIs

> Token-based Authentication
>
> The standard for authenticating users with RESTful APIs.
>
> - Client sends a request to the `/users` endpoint
> - The server captures the essential information and creates a user
> - Client sends a request with the users credentials to the `/auth` app to login
> - The server validates the information
>   - If valid = server returns Token
>   - If not valid = server returns Error
>
> ---
>
> This Token is like a temporary key we give to the client to access protected resources. The client is going to store this locally and each time it needs to access protected resources, it's going to send it to the server.

> Adding the Authentication Endpoints
>
> As you saw earlier, Django has a full authentication system built-in. But it does not have an API layer. it's just models and tables. Technically we can build the Authentication API ourselves but it's repetitive and unnecessary. That's where we use **Djoser**.
>
> Djoser is the RESTful implementation of Django Authentication System. It provides a bunch of views for user registration, login, logout, password reset and ...
>
> Look at Djoser Docs for introduction and installation.
>
> ---

Authentication Engine

> Djoser itself is an API layer that contains Models, Serializers and Routes. But the actual Authentication Process needs to be handled by an Authentication Engine. With Djoser we have two choices:
>
> - Token Based Authentication (built in Django)
>   - Keeps tokens in a table which means, per each request there will be a database call
> - JSON Web Token Authentication (third party library)
>   - No database table required. validation is done through digital signatures and token structure

---

> After installation we can see that the endpoint are not accessible. This is because we need to pass a JWT in the request header.

> Registering Users
>
> The `GET` request is not allowed for Anonymous users but the `POST` method is open to everybody for registering.
>
> First and Last name are not in the registration form by default. so let's add them. We know that these fields are based off a serializer. This serializer is defined in **Djoser**. so if we want to capture additional fields, we need a custom serializer.
>
> > Look at Djoser Documentation

Creating the custom `serializer`

create a `serializers.py` module in the core app

```
from djoser.serializers import UserCreateSerializer as
BaseUserCreateSerializer


class UserCreateSerializer(BaseUserCreateSerializer):
    class Meta(BaseUserCreateSerializer.Meta):
```

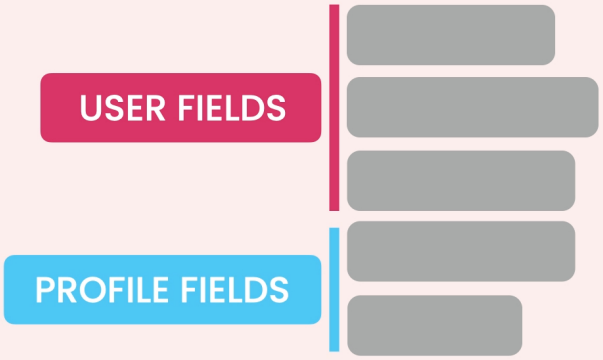
```
        fields = ['id', 'username', 'password', 'email',
'first_name', 'last_name']
```

Registering the serializer

in the settings module

```
DJOSER = {
    'SERIALIZERS': {
        'user_create':
'core.serializers.UserCreateSerializer'
    }
}
```

Adding profile data to the serializer



So the client app needs to call two different endpoints from the same form

> Every Component should have a **Single** responsibility

Building the Profile API

Build the customer API

- `Serializer`

- `ViewSet`
- `Route`

Logging in

- `/jwt/create` - Create a new token (**LOGIN ENDPOINT**)
- `/jwt/refresh` - Refresh an expired token

```
{
        // a long lived token used for regenerating an
access token
        "refresh": "token",
        // a short lived token used for calling secured
endpoints
        "access": "token"
}
```

Token Life-span

- Refresh - 1 day by default
- Access - 5 minutes by defaults

> These values are perfect for a real application. but for ease in development we change them to something longer

Overriding Life-span

look at `django-rest-framework-simplejwt` docs.

> These tokens get stored in the local storage by the client apps.

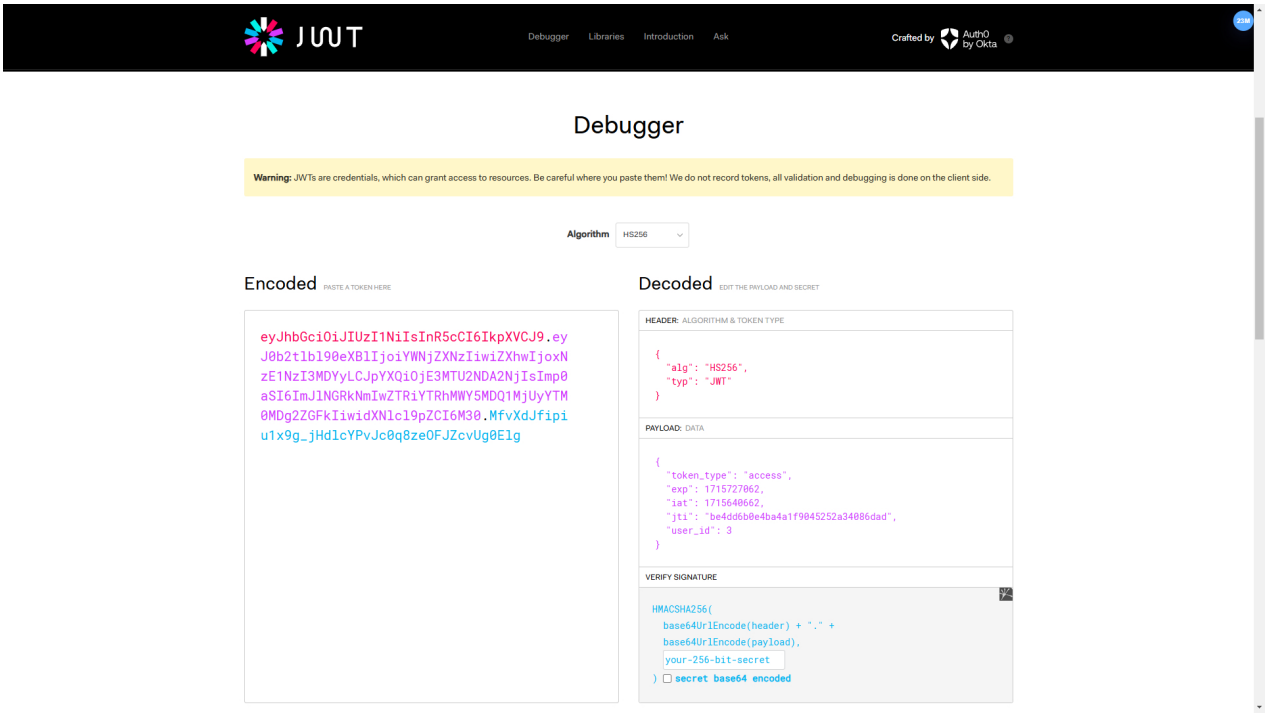
Logout

All we have to do is to delete the stored token from the client

Inspecting a JSON Web Token

[JWT Website]

Using the debugger in this website, we can provide a JWT and Decode it



Refreshing Tokens

If the access token is expired, the client needs to send its `refresh_token` to `/auth/jwt/refresh` to get a new access token.

Getting the Current User

We can get the current user by sending a `GET` request to `auth/users/me`. Right now we get 401, because we don't have our authentication credentials in the header (**JWT**) so we must use an extension called `modheader` to attach the token to the request object.

```
{
        "Authorization": "JWT {Access Token}"
}
```

Now we can hit `/auth/users/me` and we get the results.

Adding Profile fields to `/users/me`

> We need a custom serializer. Do this as an exercise

> Getting Current User's Profile
>
> Available at `/store/customers/me`.
> In order to do this, first we need to create a custom action in `CustomerViewSet`
>
> Support `GET` / `PUT`
>
> If the User Does not have a corresponding customer profile, Create it

> Applying Permissions
>
> See DRF docs and head over to permissions section.
> Some built-in permission classes:
>
> - `IsAuthenticated`
> - `IsAdminUser`
> - `IsAuthenticatedOrReadOnly`
>
> We can also define our own permission classes

Usage

- we can add them globally at `REST_FRAMEWORK` settings with `DEFAULT_PERMISSION_CLASSES: ['restframework.permissions.AllowAny']`
- on view sets with `permission_classes`

```
permission_classes = [IsAdmin, IsAuthenticated]
```

> Close `CustomerViewSet` to Anonymous users.

> Just Like `get_queryset` and `get_serializer_class`, we have `get_permissions` which we can use to change permissions at runtime.

> Applying Custom Permissions
>
> Anyone can **view** products, but only admins can **add** / **modify** / **delete** products.
>
> Let's create `IsAdminOrReadOnly`
>
> - create `store/permissions.py`
> - import `BasePermission`

```
class IsAdminOrReadOnly(BasePermission):
    def has_permission(self, request, view):
        return bool(
            request.method in SAFE_METHODS
            or request.user and request.user.is_staff
        )
```

> Applying Model Permissions
>
> With this feature we can use permissions that are defined in the permissions and groups tables.
>
> > In many cases this is over-engineering but if you want very specifically defined authorization, this is the way to go.