Django pt. 2: Database, ORM, and Auth

Kianoosh Abbasi

CSC309 Fall 2022

So far

- How web works, HTML, CSS, JS
- Backend vs Frontend development
- Django
  Setup, simple views, forms, templates
- MVC design patterns

This week

- Working with a database
  ORM and models
- Authentication
  User model, sessions, login
- Class-based views
- Admin panel

Signup form cont'd

- We have not stored/read data so far!
  - Every web application needs a persistent storage

- Many different databases are around
  - Relational: Postgres, mySQL
  - Non-relational: Cassandra, MongoDB

- Django supports various database backends

Do we need Django's support?

- Technically, we can make a connection to any database and run queries

- But this is a terrible idea!

  WHY?

- How can the framework/language help us out?

Object Relational Mapper

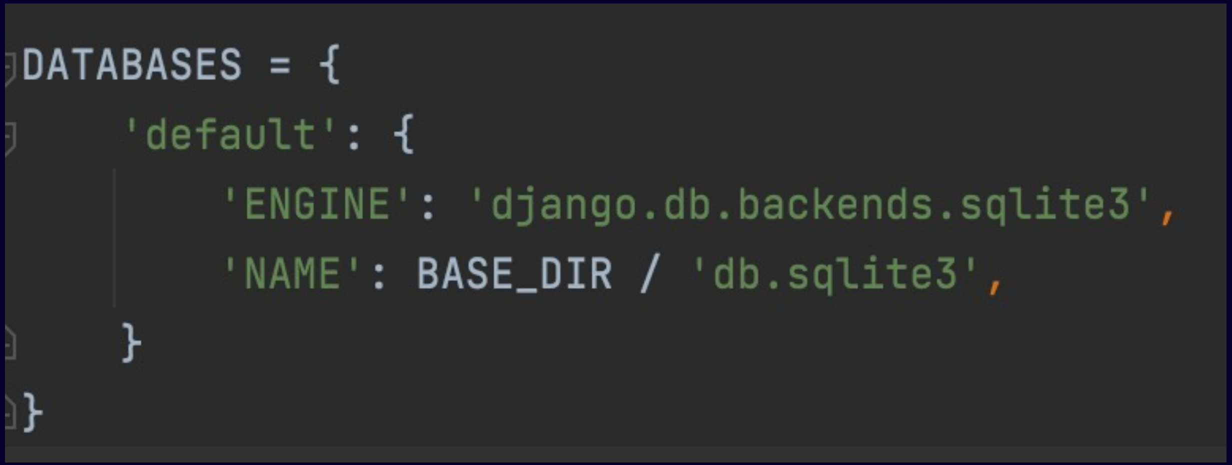
- Provides an abstraction over the underlying database queries
- Method/attribute accesses are translated to queries
- Results are wrapped by objects/attributes

Object Relational Mapper

- Simplicity: No need to use SQL syntax
- Consistency: Everything is in the same language (Python)
- Can switch database backend easily
- Enables Object Oriented Programming
- Runs a secure efficient query
  - SQL injection, atomicity, etc.
- But, for super-efficient queries, you might still need to run raw queries

SQLite

- Django's default database backend
- Light-weight database that stores everything in one single file
- Follows standard SQL syntax
- Great option for development: no setup/installation required
- For production, switch to a more sophisticated database



```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Models

- Represents, stores, and manages application’s data
  The M from MVC

- Typically, a table in the database

- Thanks to ORM, models can be defined as classes

- Django has some pre-defined models

- User: Django’s default model for authentication and authorization

Authentication vs Authorization

- Authentication:
  + Who's calling?
  - This is Daniel Zingaro
  + Is it really Daniel Zingaro?
- Obtains user information from user/pass, session, API key, etc.

- Authorization:
  Does Daniel Zingaro have enough access and permissions (aka authorized) to make this request?
- Checks user's properties and permissions

User

- Pre-defined fields:
  username, password, first_name, last_name, email, etc

- Raw passwords must never be saved to database

- Considerations:
  Username is case-sensitive!
  Emails don’t have to be unique!

Creating a user

- Initially, database is empty and has no table
  - Even Django’s pre-defined tables
- To add/updates tables, you must migrate the database
  - Run `python3 manage.py migrate`
- More on migrations next session

Creating a user

- Create a user via ORM

```
User.objects.create_user(username='dan1995', password='123',
first_name='Daniel', last_name='Zingaro')
```

- Access user(s)

```
users = User.objects.all()
dan = User.objects.get(username='dan1995’)
active_users = User.objects.filter(is_active=True)
```

- Delete user(s)

```
User.objects.all().delete()
dan.delete()
```

Working with the ORM

- Every models has an objects attribute

  Handles database queries

- filter and get both run select statements

- get returns exactly one object

  Throws an exception if zero, two, or more objects are returned

- filter returns a list of objects (more precisely, a queryset)

Querysets

- Evaluated lazily

  Queries not run until really needed

- This example only runs one query:

```
users = User.objects.all()
users2 = users.filter(is_active=True)
users3 = users2.filter(username__contains='test')
user = users3.get()
user.get_full_name()
```

Update queries

- Update a queryset

```
User.objects.filter(is_active=True).update(is_active=False)
```

- Update a single instance

```
dan = User.objects.get(first_name='Daniel')
dan.first_name = 'Dan'
dan.save()
```

- Attributes are locally cached values

  To refresh: `dan.refresh_from_db()`

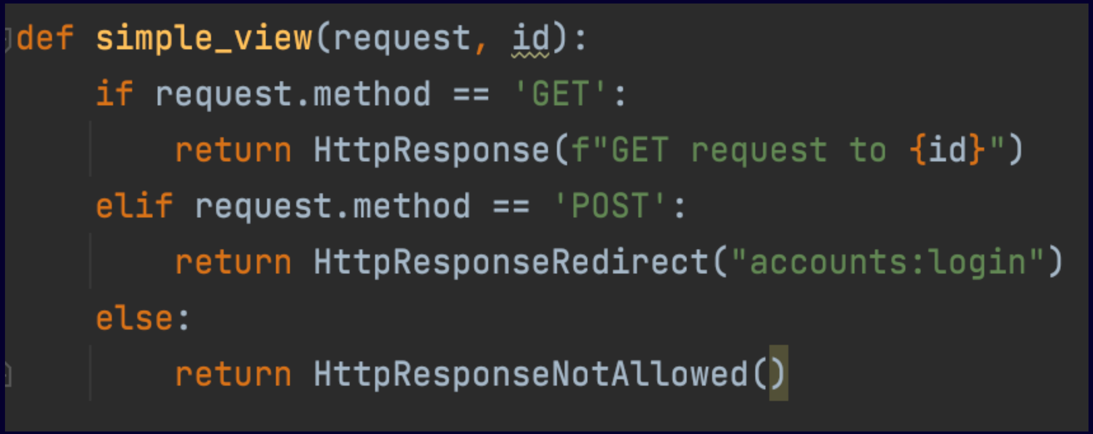
Exercise: Extend the signup form to actually create a user

Advanced Views

Class-based views

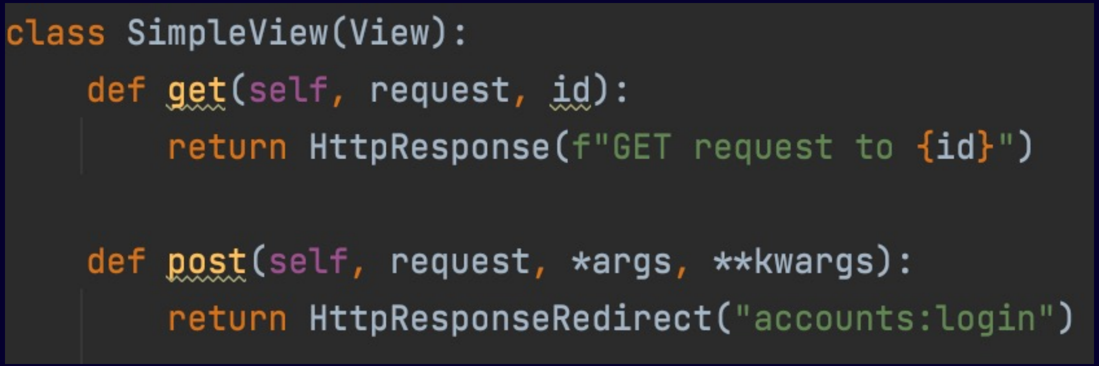
- Views can be numerous and big
- Class-based views: standard for medium/big projects
  A new instance created at every request: NO shared self object
- Create views directory and have each view in a separate file
- Subclass `django.views.View` and create a method (function) for each HTTP method

Function based



```python
def simple_view(request, id):
    if request.method == 'GET':
        return HttpResponse(f"GET request to {id}")
    elif request.method == 'POST':
        return HttpResponseRedirect("accounts:login")
    else:
        return HttpResponseNotAllowed()
```

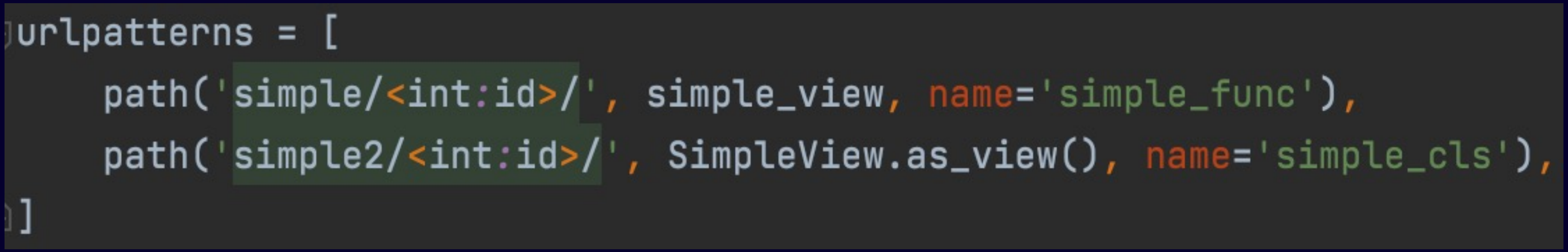
Class based



```python
class SimpleView(View):
    def get(self, request, id):
        return HttpResponse(f"GET request to {id}")

    def post(self, request, *args, **kwargs):
        return HttpResponseRedirect("accounts:login")
```

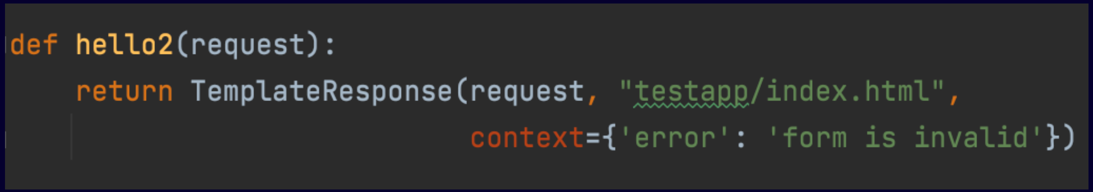
URLs



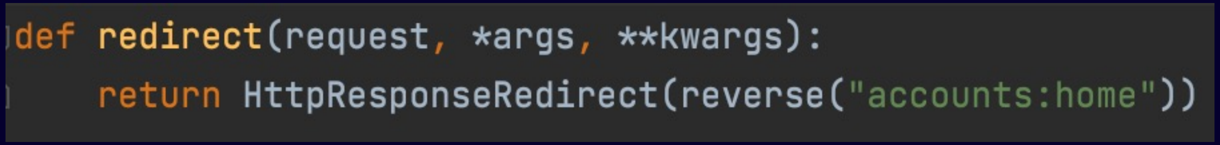
```python
urlpatterns = [
    path('simple/<int:id>/', simple_view, name='simple_func'),
    path('simple2/<int:id>/', SimpleView.as_view(), name='simple_cls'),
]
```

TemplateView

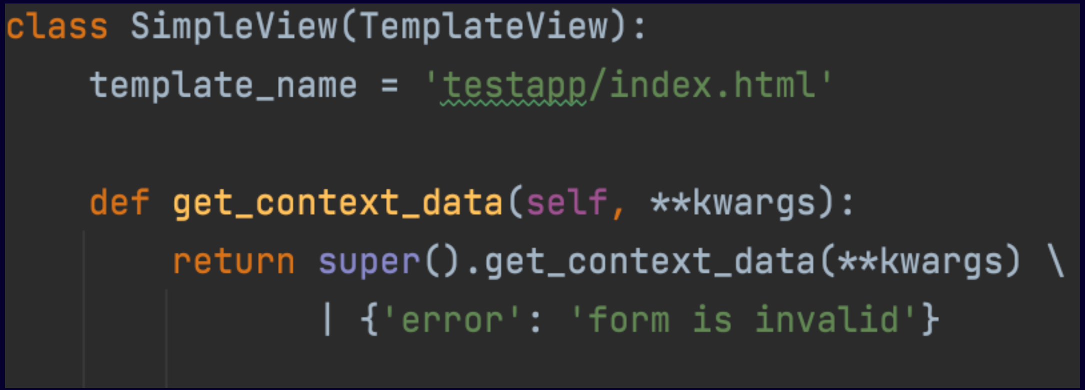
RedirectView



```python
def hello2(request):
    return TemplateResponse(request, "testapp/index.html",
                            context={'error': 'form is invalid'})
```

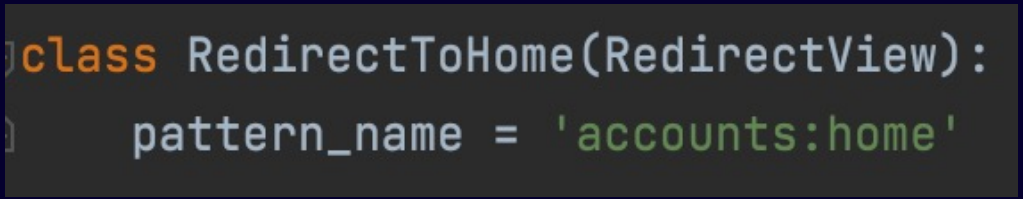


```python
def redirect(request, *args, **kwargs):
    return HttpResponseRedirect(reverse("accounts:home"))
```



```python
class SimpleView(TemplateView):
    template_name = 'testapp/index.html'

    def get_context_data(self, **kwargs):
        return super().get_context_data(**kwargs) \
            | {'error': 'form is invalid'}
```



```python
class RedirectToHome(RedirectView):
    pattern_name = 'accounts:home'
```

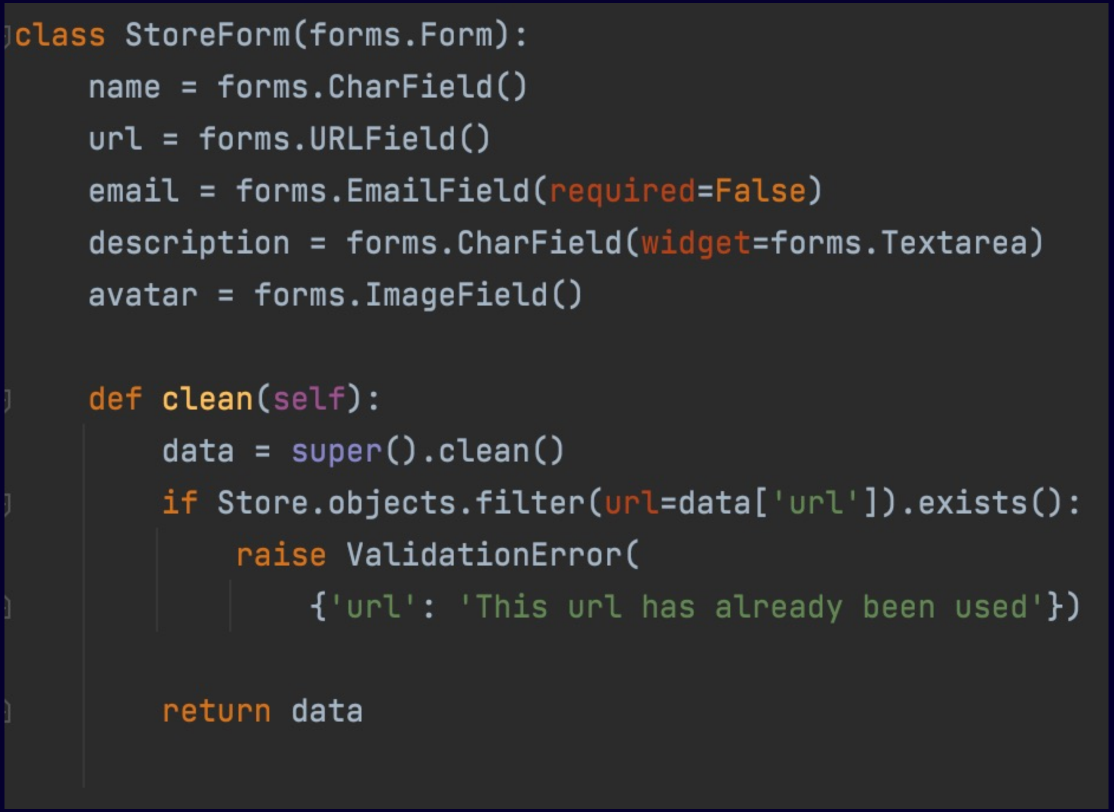
FormView

- A way to organize forms
  Separates the form logic from the view logic

- Form class:
  Define fields one by one
  Define a clean method for validation

- FormView:
  Specify `form_class` attribute
  Specify `success_url` or `get_success_url`
  Override `form_valid` to apply the changes

forms/store_form.py

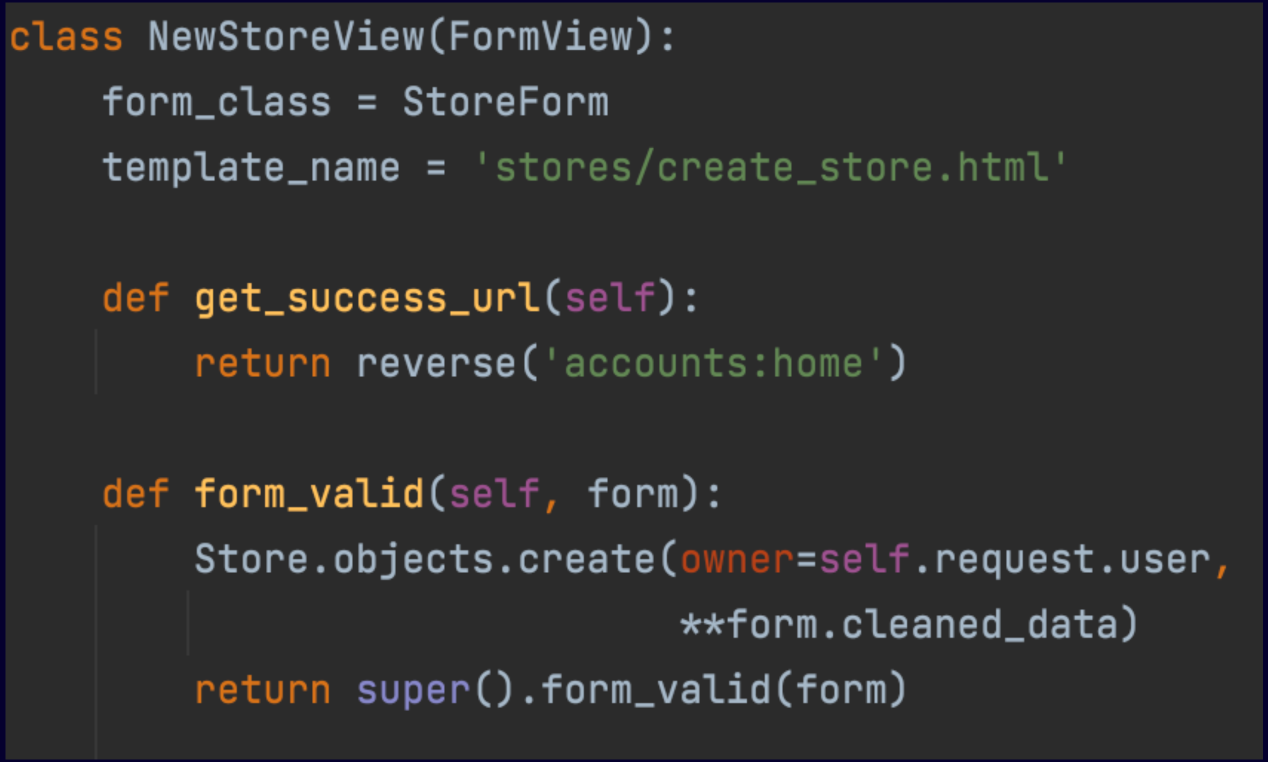


```python
class StoreForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    email = forms.EmailField(required=False)
    description = forms.CharField(widget=forms.Textarea)
    avatar = forms.ImageField()

    def clean(self):
        data = super().clean()
        if Store.objects.filter(url=data['url']).exists():
            raise ValidationError(
                {'url': 'This url has already been used'})

        return data
```

views/new_store.py



```python
class NewStoreView(FormView):
    form_class = StoreForm
    template_name = 'stores/create_store.html'

    def get_success_url(self):
        return reverse('accounts:home')

    def form_valid(self, form):
        Store.objects.create(owner=self.request.user,
                             **form.cleaned_data)
        return super().form_valid(form)
```

Template

- The form instance is being created at every request
- GET request: `{{ form }}` sent to template context
- Up to developer to use it at all!
  You can stick with your html input tags (recommended)
- Reason: MVC’s view should be separated from controller

- POST request: form instance created and populated with POST data

- Goes through validation and calls `form_valid` (redirect) or `form_invalid`

- `form_invalid` renders the template (like GET)

  This time, the `{{ form }}` instance has values and errors

- Use values and errors at template

  ```
  {{ form.name }} {{ form.name.error }}
  {{ form.non_field_errors }}
  ```

Authentication

- Client should tell us who they are
- Via Authorization header in HTTP
- Several authentication methods
  - Password auth
  - Session auth
  - Token auth

Basic (password) auth

- Simply sends username and password at every request
- No concept of login and logout
- Unencrypted base64 strings
- So insecure: transfers raw password this many times

Session auth

- Client sends user/pass at login
- If successful, server creates and stores a session id
  - Mapped to user
- Session id returned in the response
  - Browser saves it in cookies
- Browsers sends the same session id at next requests
  - Incognito tab: browser does not send the same session id

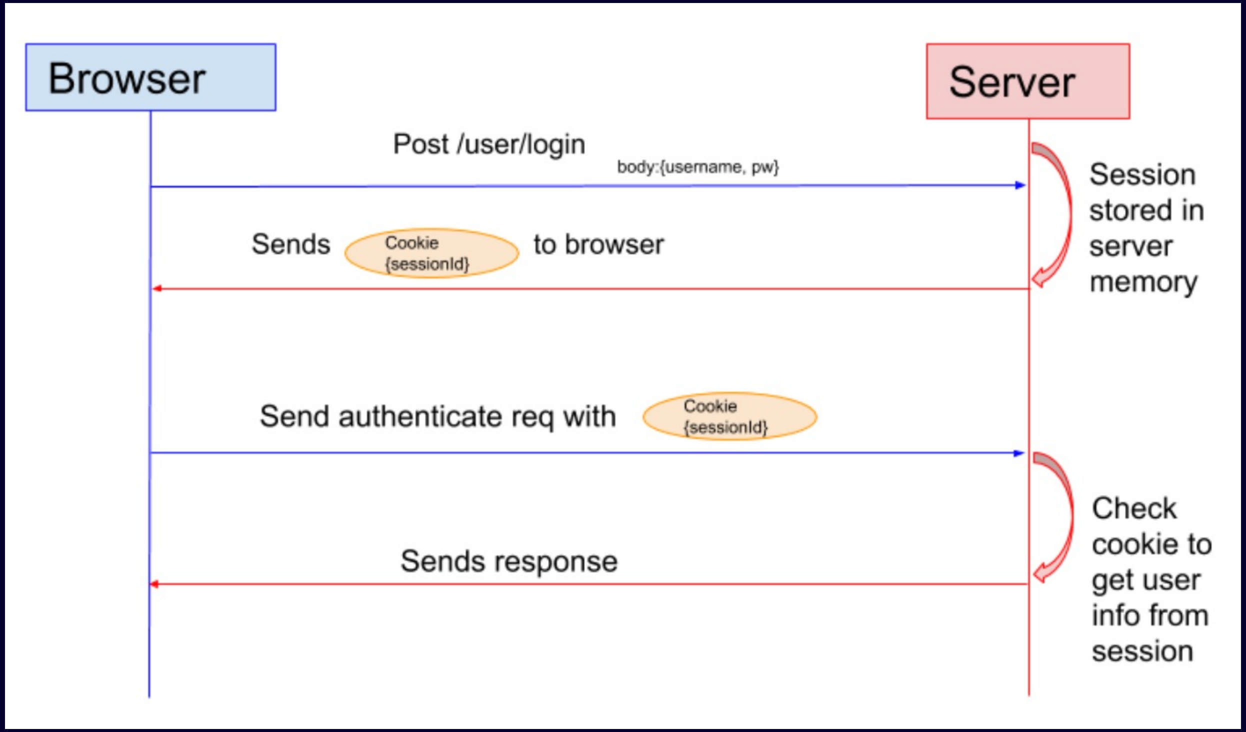
Token auth

- Storing every single session could be an overhead
  - Limits the scalability of the application

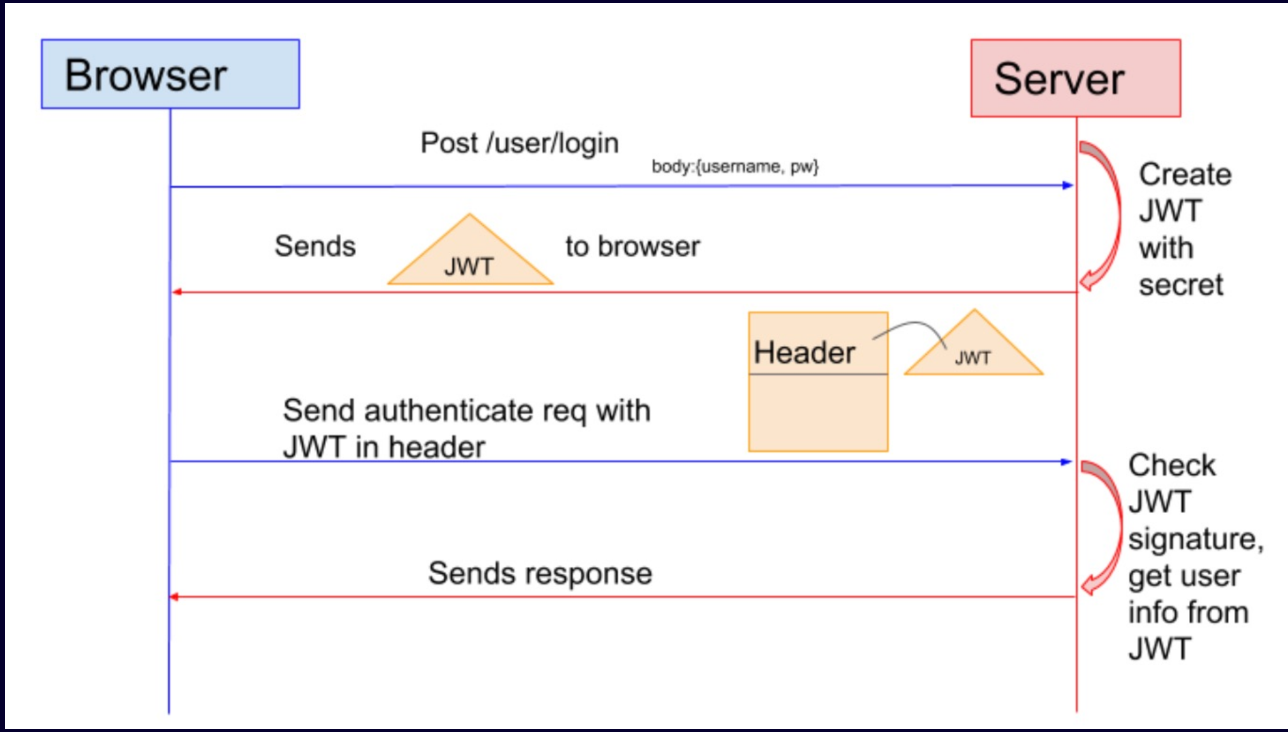
- Instead of a random session id, the token can contain information about the user

- Must be signed by the server to avoid attacks

Session auth



Token auth



Source: https://sherryhsu.medium.com/session-vs-token-based-authentication-11a6c5ac45e4

Django's session auth

- Check user/pass combination is right
  ```
  user = authenticate(username='john', password='secret')
  ```
- Django's login function: attaches user to the current session
  ```
  login(request, user)
  ```
- Django does the session id lookup itself
  User object accessible at `request.user`
  `AnonymousUser` if unauthenticated
- logout function: removes session data

Exercise: a class-based login view

Admin panel

- A very convenient medium to see/change database records
  - Instead of running raw queries or python code at `python3 manage.py shell`
- The admin url at urls.py
- Needs an active user with `is_superuser` and `is_staff` `True`
  - Can be created manually through the shell
  - Or via command: `python3 manage.py createsuperuser`

This week

- Working with a database
  ORM and models
- Authentication
  User model, sessions, login
- Class-based views
- Admin panel

Next week

- Custom models

  Fields, relations, queries

- CRUD views

- Advanced admin panel

- Migrations