CSC309H1S

Programming on the Web

Winter 2023

Lecture 5: Django Templates and Models

Instructor: Kuei (Jack) Sun

Department of Computer Science
University of Toronto

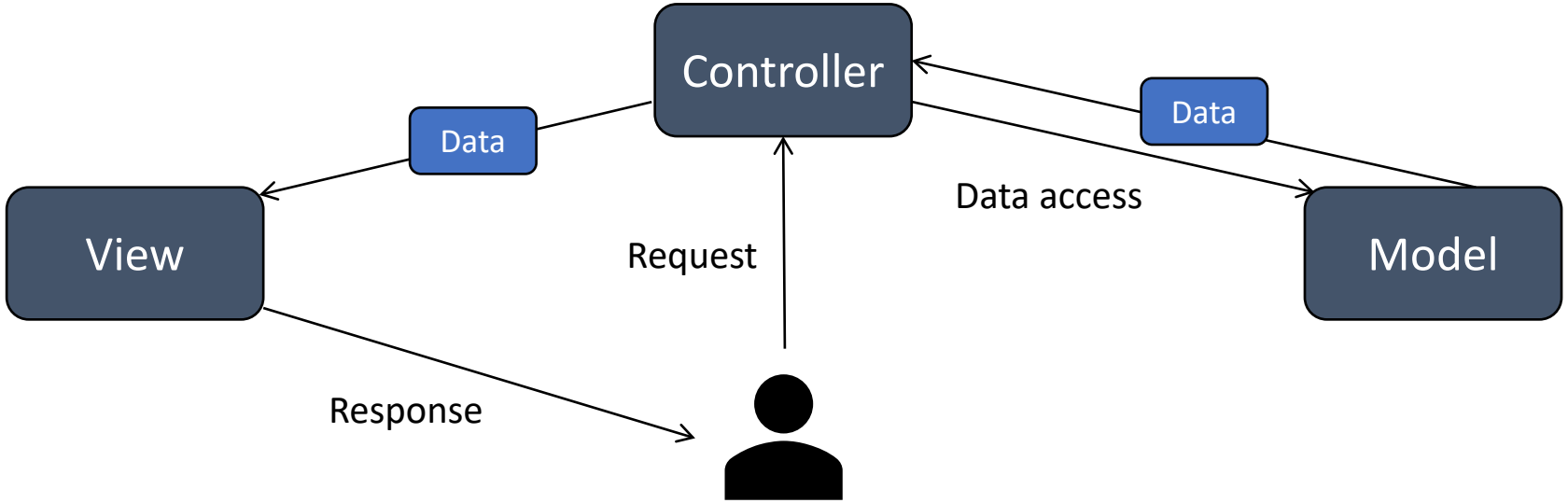
v1.01

Architectural Design Patterns

- General approach to recurring problems in software architectural design
- Helps clarify and define components in a large application
- Frequently used terms:
  - Model: handles data storage and forms logical structure of the application.
    - Can include business logic that handles, modifies, or processes data
  - View: the presentation layer that handles user interface.
- Frequently used patterns for web applications (with UI)
  1. MVC/MVT (model-view-controller/model-view-template)
  2. MVP (model-view-presenter)
  3. *Optional reading*: MVVM (model-view-view-model)

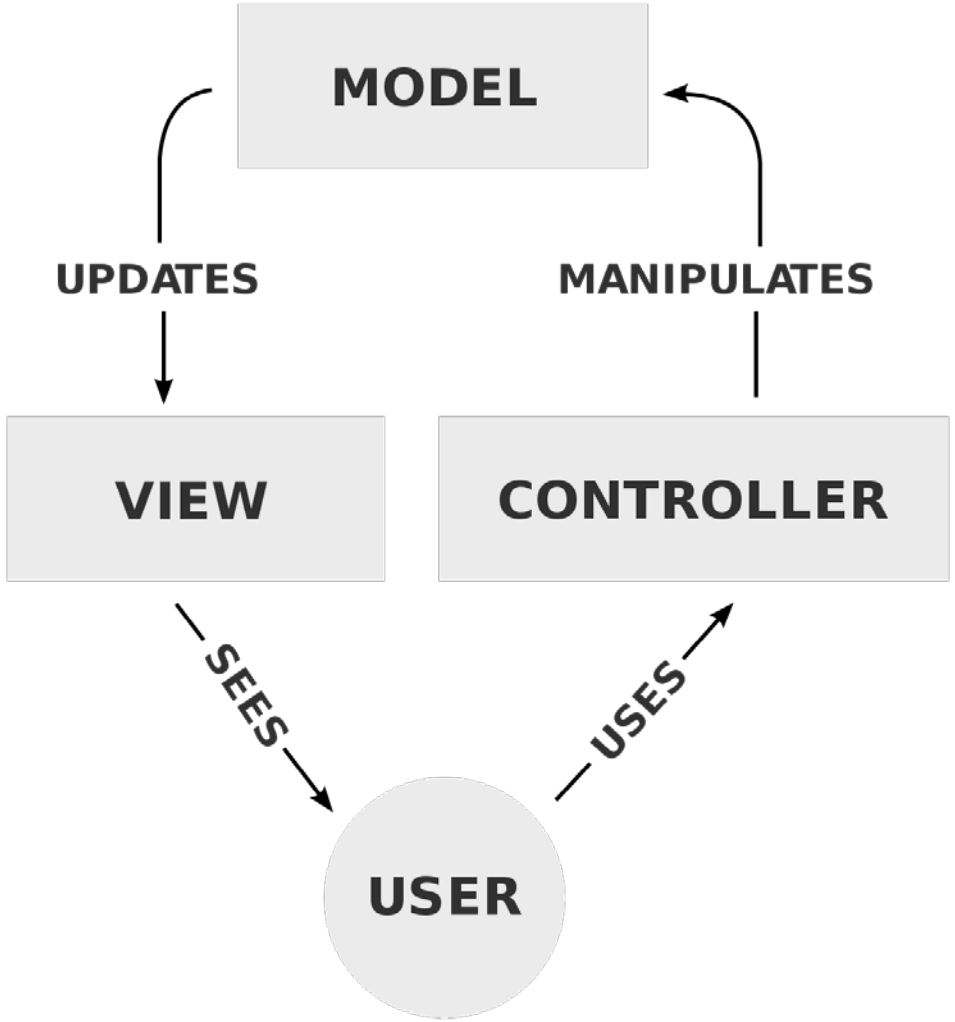
Model-View-Controller (MVC)

- Focuses the separation of appearance, data, and business logic.
- View depends on model (data)
- Controller manipulates model and connects it to relevant view
- Easy to switch out presentation or data source (e.g., database)



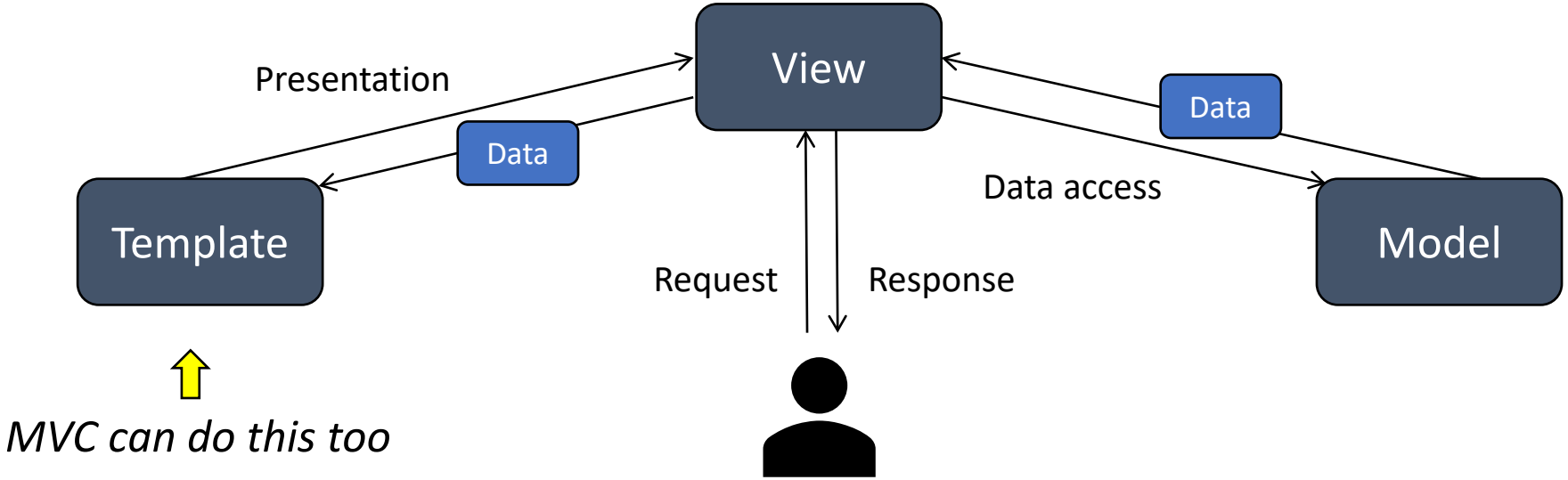
MVC Pattern

- Alternative interaction
  - Model *notifies* view of update.
- Both controller and model can handle business logic.
- Rule of thumb
  - *Fat model* and *skinny controller*
  - Model should handle domain-specific knowledge, e.g., account management.
  - Controller should handle application logic only. e.g., ask for password.



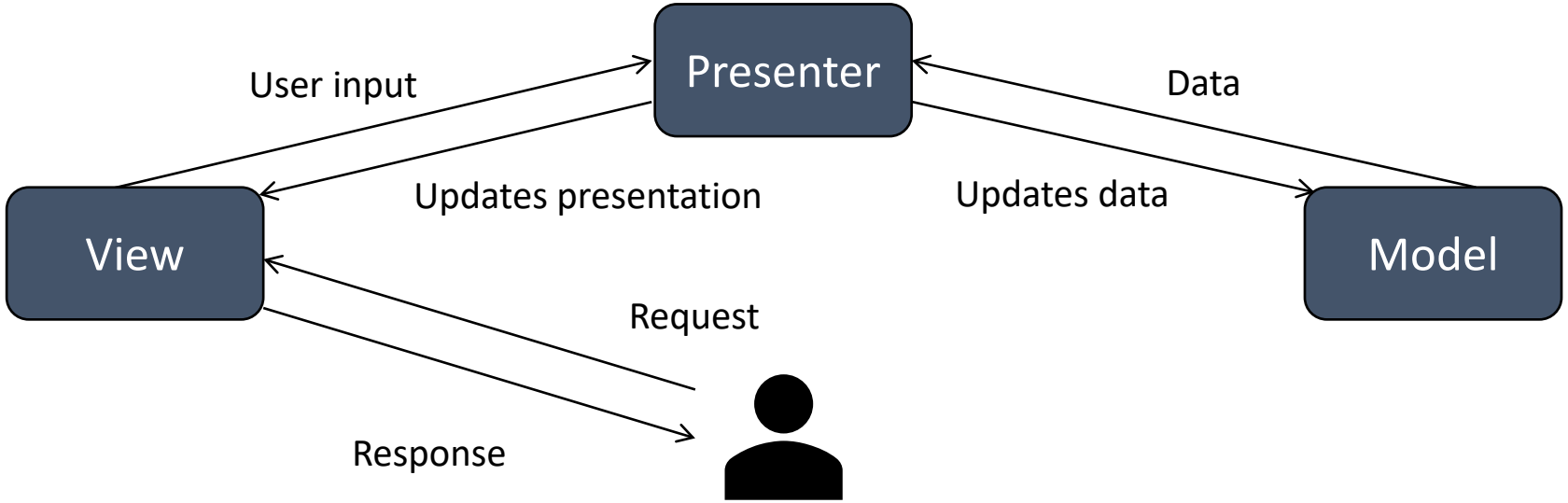
Model-View-Template (MVT)

- Same as MVC, except it uses Django's terminology
- Django view = MVC controller, Django template = MVC view
- URL dispatcher is part of MVC controller
  - Classical controller does not have one, *but moderns one do* (e.g., Spring MVC)

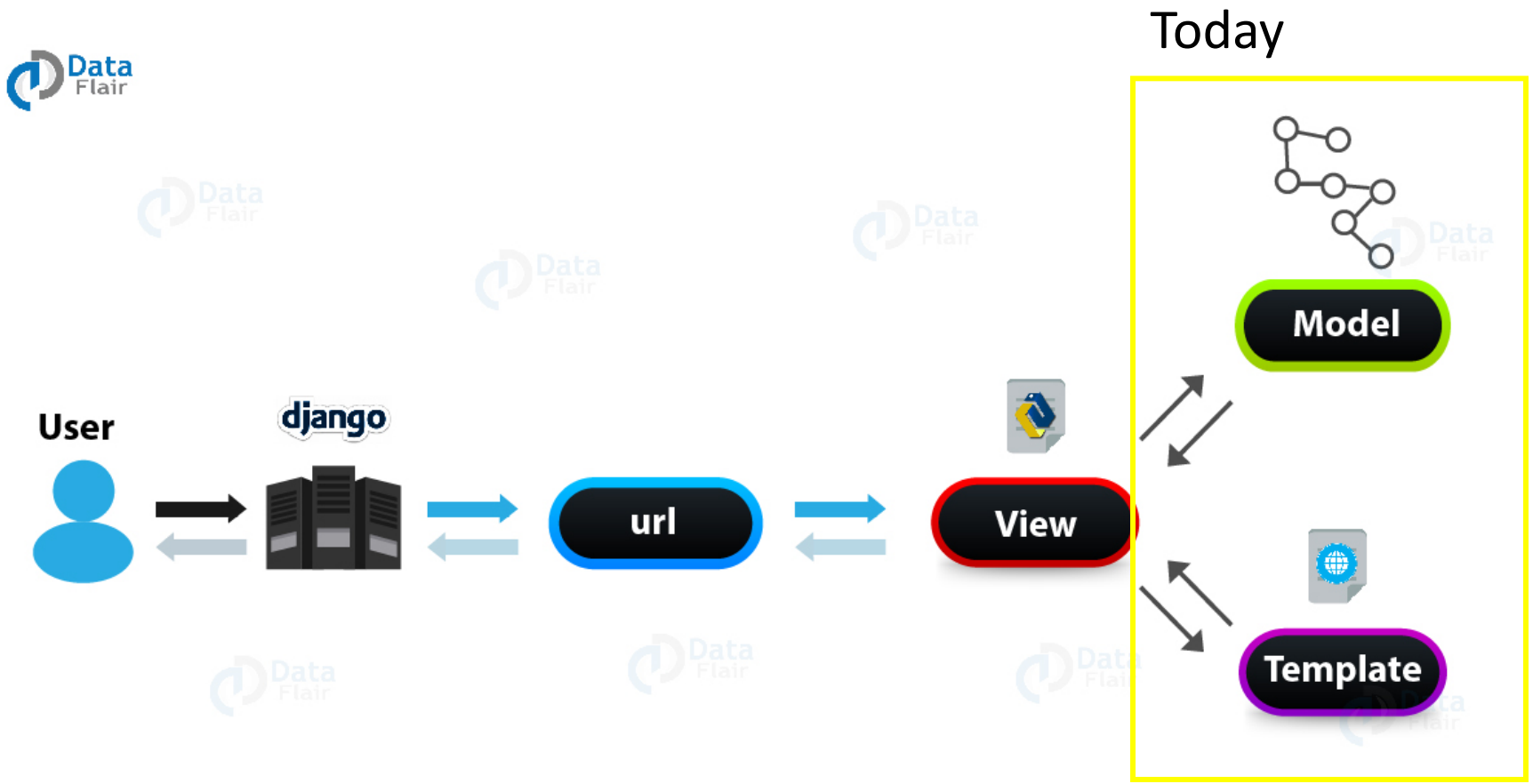


Model-View-Presenter (MVP)

- User communicates with the view.
- Presenter receives input from view and process data with model.
- Presenter updates view through an interface, e.g., `show_products`
- Breaks dependency of model from view by acting as "middleman"



Django's Architecture



Source: https://data-flair.training/blogs/django-architecture/

Django Template Language

- Review
  - DTL supports imperative programming features
  - Data passed from view to template via context.
  - Variables are replaced by data from context
  - Tags control the logic of the template
- Control flow tags
  - for loop

```
<ul>
{% for athlete in athlete_list %}
  <li>{{ forloop.counter }}:
    {{ athlete.name }}</li>
{% endfor %}
</ul>
```

  - if, elif, else

```
{% if ticket_unavailable %}
<p>Tickets are not available.</p>
{% elif tickets %}
<p>Number of tickets:
   {{ tickets }}.</p>
{% else %}
<p>Tickets are sold out.</p>
{% endif %}
```

Django Template Tips

- {% url 'namespace:name' %}
  - Same as Django's `reverse` function, to map named URL to user URL
- if tags can take relational operators
  - E.g., {% **if** myvar == "x“ %}
  - E.g., {% **if** user **in** user_list %}
  - E.g., {% **if** myval > 500 %}
- Members variable, dictionary lookup, index access all use dot operator
  - E.g., {{ user_list.0 }}
  - E.g., {{ request.POST.username }}
- Django template comment
  - E.g., {# hello, this is a comment #}

Django Template Filters

- Like a function that modifies a variable for display
- Syntax: pipe character followed by filter name, i.e., {{ *var* | *filter* }}
- List of tags and filters:
  - https://docs.djangoproject.com/en/4.1/ref/templates/builtins/
- `length`
  - Same as Python len()
  - E.g., {{ my_list | length }}
- `lower`
  - Same as str.lower()
  - E.g., {{ title | lower }}
- `time`
  - Formats time object
  - E.g., {{ value | time:"H:i" }}
    - 10:05
- Filters can be chained
  - {{ value | first | upper }}

Django Template Inheritance

- Parent templates define *blocks* that child templates can override.

```
<head>
  {% block staticfiles %}
  <link rel="stylesheet" href="{% static '/css/boostrap.css' %}">
  {% endblock %}
  <title>{% block title %}My amazing site{% endblock %}</title>
</head>
<body>
  <div id="sidebar">
    <ul>
    {% block sidebar %}
      <li><a href="/">Home</a></li>
    {% endblock %}
    </ul>
  </div>
  <div id="content">{% block content %}{% endblock %}</div>
</body>
```

Django Template Inheritance

- Example child template

```
{% extends 'parent.html' %}
{% load static %}

{% block staticfiles %}
  {{ block.super }}
  <link rel="stylesheet" href="{% static '/css/child.css' %}">
{% endblock %}

{% block title %}My Child{% endblock %}

{% block sidebar %}
  {{ block.super }}
  <li><a href="{% url 'child' %}">Child Page</a></li>
{% endblock %}

{% block content %} <h1>Child Page</h1> <p>{% lorem %}</p> {% endblock %}
```

`extends` must be on the first line

child template does not inherit tags loaded in parent.

adds parent's code in block

Note: code outside of block tags are ignored

Root Template Folder

- Typically, each template belongs to only one view
  - These templates should be placed inside the app folders.
- Some templates have common components across apps
  - E.g., navigation bar, footer, form elements, etc.
- Reusable templates can be placed in a root template directory.

```
TEMPLATES = [
  {
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [ BASE_DIR / "templates" ],
    'APP_DIRS': True,
    'OPTIONS': {
        …
    },
  },
]
```

adds this line

Django include tag

- Render a sub-template and include the result
- {% **include** template_name %}
  - Can be a variable, absolute path, or relative path (`extends` supports this too)
- Sub-template is rendered with the current context.
- Can pass additional context
  - {% **include** "greeting.html" **with** person="Bob" %}
- Can restrict context to only ones explicitly passed in
  - {% **include** "greeting.html" **with** person="Bob" user=user **only** %}
- Exercise
  - Do Question 4 on Quercus

Database

- Most web applications need a persistent storage
- Database
  - Collection of data organized for fast storage and retrieval on a computer.
- Choices for primary database
  - Relational: MySQL, PostgresSQL
  - Non-relational (NoSQL): Cassandra, MongoDB
- Django supports various database backends
  - Transparently through an Object Relational Mapper
  - See list of built-in support
    - https://docs.djangoproject.com/en/4.1/ref/settings/#databases

Object Relational Mapper

- Provides an *abstraction* for accessing the underlying database
- Separates application from database implementation
  - We could connect to a specific database using its client, e.g., MySQLdb
  - However, it would couple our application to the database of choice
- Method calls and attribute accesses are translated to queries
- Query results are encapsulated in objects and their attributes
- Django has a built-in ORM layer
- Other ORM frameworks:
  - SQLAlchemy (Python), Hibernate (Java), Sequelize (JavaScript)

Object Relational Mapper

- Advantages
  - Simplicity: no need to learn SQL or other database languages
  - Consistency: everything is in the same language (e.g., Python)
    - Enables object-oriented programming
  - Flexibility: can switch database backend easily
  - Security: runs secure queries that can prevent attacks like SQL injection
  - Seamless conversion from in-memory types to storage types, and vice versa
    - E.g., storing datetime as an integer in database
- Disadvantage
  - Additional layer of abstraction reduces overall performance
  - Hiding implementation detail may result in poorly designed database schema

SQLite

- Django's default database backend
- Lightweight database that stores everything in a file

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

- Follows standard SQL syntax
- Excellent option for development; no setup or installation required
- For production, a more scalable database is required

Django Models

- Represents, stores, and manages application data
- Typically implemented as one or more tables in database
  - Some models require database joins
- The ORM layer enables defining models with classes
- Django has a set of predefined models for convenience
- Example:
  - User: Django's default model for authentication and authorization
  - Permissions: what a user can or cannot do
  - Session: stores data on *server-side* about current site visitor

Django Security Model

- Authentication
  - Verifies identity of a user or service
  - Typically done through verification of correct username and password
    - Other methods include API key, session token, certificate, etc.
  - Two-factor authentication
    - Provides additional layer of protection by asking additional information
      - E.g., one-time passcode sent to email or phone
- Authorization
  - Determines a user's access right
  - Checks user's privilege level (group) and permissions

User Authentication in Django

- https://docs.djangoproject.com/en/4.1/topics/auth/
- User
  - Derived class of AbstractUser
  - Contains predefined fields
    - username, firstname, lastname, email, etc.
  - Passwords are *hashed* before they are stored
    - Storing raw passwords can result in identity theft if database is hacked
  - Passwords are also *salted* before hashing
    - Rainbow attack
      - Uses a table of known hashes values to revert to original plaintext
    - Salt is a random value that is added to the password

Setting Up Database Tables

- Initially, the database is empty with no tables
  - Same with Django's predefined tables
- To add/update tables, run the migrate command
  - `python3 manage.py migrate`
  - The ORM layer will create or update database tables for you
- Django shell
  - Provides interactive Python shell within Django environment
  - Helps you test models without running a web server
  - `python3 manage.py shell`

Working with ORM Objects

- Create an object
  - `User.objects.create_user(username='dan1995', password='123', \ first_name='Daniel', last_name='Zingaro’)`
  - Some fields are optional, e.g., first_name, last_name
    - Preview: the field is defined with blank=True
- Get all objects of the same type
  - `users = User.objects.all()`
- Get just one object based on exact match
  - `dan = User.objects.get(username='dan1995’)` ← can return not found or not unique.
- Delete object(s)
  - `User.objects.all().delete()`
  - `dan.delete()`

Working with ORM

- Every model (Python class) has an objects class attribute
  - E.g., User.objects
  - Handles database queries, such as SELECT statements
  - all(), get(), and filter() returns a QuerySet
- objects.all() retrieves all objects, objects.get() retrieves exactly one
- objects.filter()
  - Returns a list of objects based on one or more field lookups
  - Syntax: filter(fieldname__lookup=value, …)
    - Exception: exact match does not require a lookup
  - E.g., User.objects.filter(last_name="Smith", age__gt=19)

QuerySets

- Evaluated lazily
  - Queries are not run until field of object is accessed
- In this example, only **one** query is run:

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

- A lot of methods and field lookups!
  - https://docs.djangoproject.com/en/4.1/ref/models/querysets/
  - Methods: exclude(), order_by(), annotate(), etc.
  - Lookups: in, iexact (case-insensitive match), isnull, etc.

Update Queries

- Update a single instance

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

- Update everything in a QuerySet

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

- Attributes are locally cached values
  - To refresh
    - `dan.refresh_from_db()`
- Exercise
  - Do Question 5 on Quercus

Authentication

- Clients should tell the server who they are
- Can use Authorization header in HTTP
- Several authentication methods available

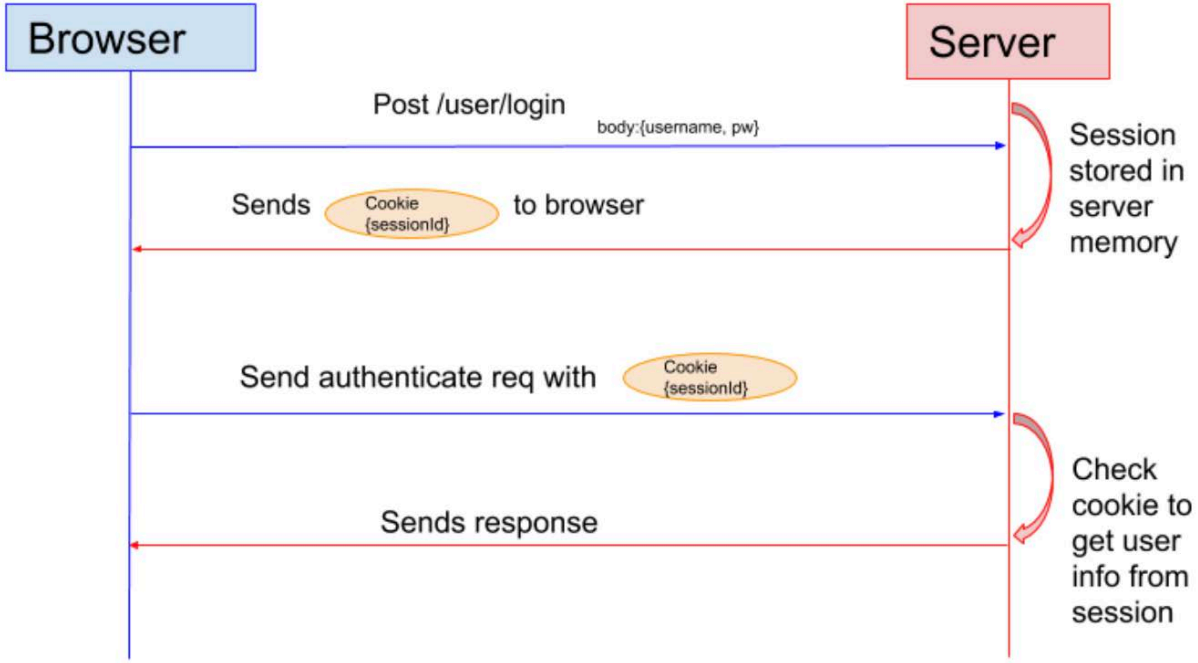
1. (Basic) password authentication
   - Sends username and password for every request
     - No concept of login and logout
   - User information is unencrypted and encoded in base64
     - Insecure without HTTPS

Session Authentication

2. Session authentication

- Client only sends username and password at login
- If successful, server creates and stores a session id
  - Session id is mapped to the specific user
- Session id is returned in the response
  - Browser saves it in cookies
  - Session data is saved on server, and not saved in cookie!
- Browser sends the same session id for subsequent requests
  - Incognito tab: browser does not send cookie so session id is not sent

Session Authentication



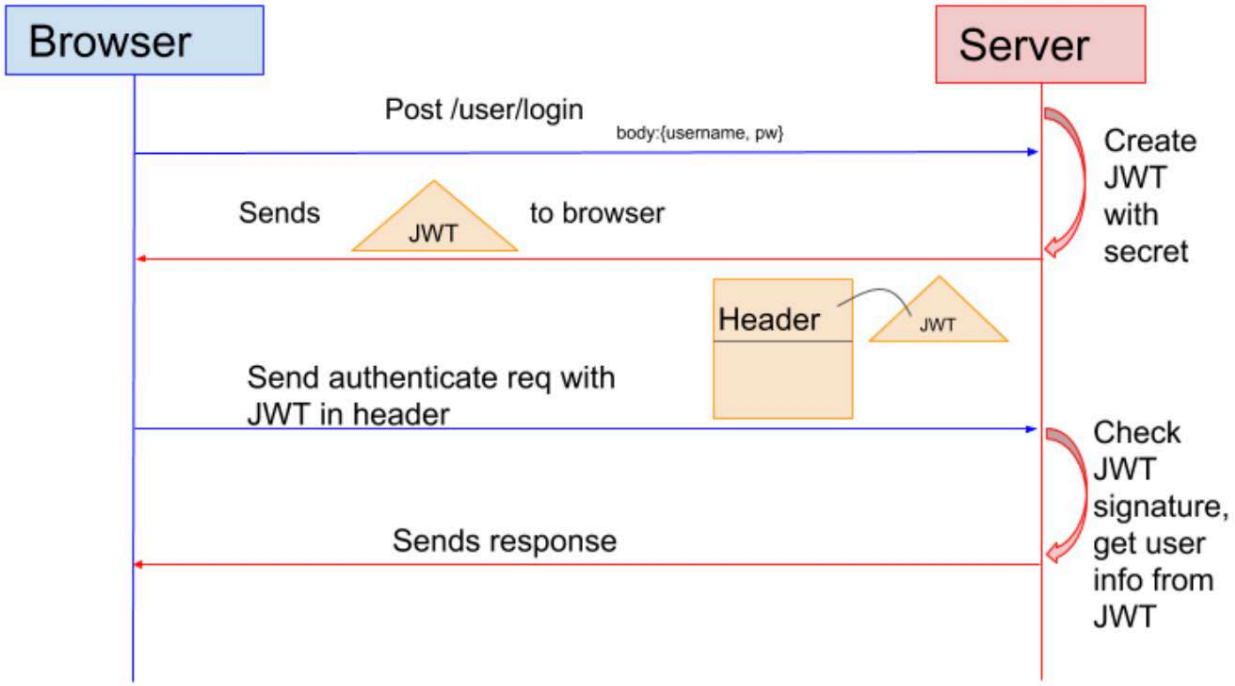
Source: https://sherryhsu.medium.com/session-vs-token-based-authentication-11a6c5ac45e4

Token Authentication

3. Token authentication

- Token is signed by server to avoid attacks
  - Can be used to identify the client and their permissions
  - Analogy: using your driver's license to go into a bar/club
    - Doorman checks its security features (signed) and your age (permission)
- Much faster than session because no database query is needed
- JSON web token (JWT)
  - Industry standard method for securely representing claims
- Claims
  - Can contain your information, including identity and/or permissions

Token Authentication



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

Django Session Authentication

- Checks that username/password combination is correct
  - `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 internally
  - User object is attached to the request object
    - request.user
  - User type is AnonymousUser if current visitor is not authenticated
- logout() function
  - Removes session data

Admin Panel

- A convenient service provided by Django to manage database records
  - Allows developer to see and update records
  - More user-friendly than running database queries or Python code
- The admin panel is installed by default
  - See the global urls.py
- Go to localhost:8000/admin to see it
- Requires an active user with is_superuser and is_staff field set to True
  - Can be created manually through the shell
  - Or created via the admin panel itself
  - Lastly, via command: `./manage.py createsuperuser`

Exercise

- Optional
- User authentication
  - Build a login page and logout page
- Making a query
  - Once logged in, display all users registered to the site
  - Add a filter feature to find users by username
- Django session
  - Create a landing page
  - Depending on which page you came from, i.e., via redirect, display a different message