CSC309 Notes

Jenci Wei

Winter 2023

Contents

1 Introduction

End User Perspective

1. User enters a web address inside a browser
2. Browser sends a request to the server
3. Server processes the request and responds with a web page

World Wide Web (the Web)

- A collection of information and services that can be accessed on local devices through the **Internet**
- Internet
  - An interconnected network of computers
  - Can communicate with each other through standardized protocols
- TCP/IP
  - Protocols that provide reliable end-to-end communication between two applications on different computers
  - IP (Internet Protocol)
    - Identifies computers on the network by assigning a unique **IP address**, e.g. 192.168.7.41
    - Knows how to route data from/to the destination computer
  - TCP (Transmission Control Protocol)
    - Allows multiple virtual connections to share a single physical IP address
    - Each connection is identified by a unique **port number**, e.g. 80
    - Deals with the unreliable nature of data transmission over network
- HTTP
  - Protocol for delivery of contents from the Web

Domains

- IP addresses are hard to remember, and it is possible to move websites elsewhere
- Some websites may be hosted on multiple physical machines (i.e. CDN: content delivery network)
- Want addresses that are easy to remember, and also easy to remap to different IP addresses
- **Domain Name**
  - Maps an easy-to-remember name to IP addresses
  - E.g. www.google.com maps to 142.251.41.78
  - Clients must *resolve* the domain before making a connection

Domain Name System (DNS)

- A collection of mappings from domain names to their IP addresses
- Want to find the DNS server
- DNS manually assigned by system administrator
  - E.g. 8.8.8.8 is Google's public DNS

- DNS automatically configured when computer connects to Internet
    - Computer sends a broadcast message to everyone on the local network
    - The DHCP server is responsible for assigning an IP address to the computer, and it also provides the IP address of a DNS server

Hypertext Transfer Protocal (HTTP)

- A protocol for distributing and accessing hypertext documents, where hypertext is text displayed on electronic devices
- Built on top of TCP/IP
- Human readable protocol
- HTTP servers typically listen on port 80
- HTTPS (HTTP) secure
    - Messages are encrypted for security purposes, which protects against eavesdropping and tampering
    - Used by 81.3% off all public websites

Stateless Protocol

- Does *not* remember previous interaction with their clients
- HTTP is a stateless protocol

Statefulness

- A stateful service reacts *differently* to the same input
- Server must track the states of all open connections
- E.g. a website might want to know whether a user is logged in
    - A stateful server remembers this on the server-side
    - A stateless server gives the client a **cookie** to be passed back later, where the client *reminds* the server of the previous step

Stateful Service	**Stateless Service**
Requires server to keep information about a session (interaction with client)	Does not require server to remember session states
More complicated to design and implement	Simple to design and implement
Server crash or power outage would result in loss of session states	Server outage does not result in loss of session states
Difficult to scale (i.e. work smoothly with increased number of users)	Easier to scale and optimize, e.g. by caching responses

HTTP Message

- Components of an HTTP request:
    1. Method: describes what we want to do
    2. Path: specifies which resource we want to
    3. Header: describes various settings and client environment
    4. Body: additional data to be sent to server
- Components of an HTTP response:

1. Response code: describes the outcome of the request
2. Header: describes various settings and server environment
3. Body: data from the server (usually the hypertext of the web page)

HTTP Methods

- POST: create a new resource
- GET: read information about a resource (most used)
- PUT: replace a resource
- PATCH: modify a resource
- DELETE: delete a resource

Response Code

- Success: 200–299
  - 200: OK
  - 201: Created
- Redirection: 300–399
  - Instructs user to check out a different web address
  - 301: Moved Permanently
- Client error: 400–499
  - 404: Not Found
  - 400: Bad Request
  - 403: Permission Denied
- Server error: 500–599
  - 500: Internal Server Error
  - 502: Bad Gateway

Uniform Resource Locator (URL)

- A string to reference a web resource and how to retrieve it
- Format of a **hyperlink** for navigating through hypertext documents
- E.g. $\underbrace{\text{https}}_{\text{protocol}}\text{://}\underbrace{\text{www.utoronto.ca}}_{\text{domain name}}/\underbrace{\text{current-students}}_{\text{resource name}}$
- URL encoding
  - Some characters are not safe in documents where URLs may be used
  - Escaped using *percent encoding*, e.g. space is converted to %20

Web Browser

- A client-side application that takes an URL and retrieves a web page (using the HTTP/HTTPS protocol over TCP/IP)
- Web pages are typically written in HTML (Hypertext Markup Language)
- A web browser *renders* the hypertext to display formatted context

2 Hypertext Markup Language (HTML)

Markup Language

- Language that provides control over organization of document content
- Allows specification of its structure and various components (e.g. headings, paragraphs, etc.)
- Can help with formatting of text or multimedia components (e.g. Markdown)
- Extensible Markup Language (XML):
  - Provides a standard for storage and transmission of arbitrary data
  - Labels, categorizes, and organizes information
  - Commonly used for interchange of data over the Internet

Hypertext Markup Language (HTML)

- A special form of XML for interchange of web documents
- Browser *renders* an HTML file to display a web page
- Emphasis on *structural semantics* of **elements**
- Elements
  - Building blocks of a web page
  - Wide variety of elements are supported (e.g. images, videos, embedded PDFs, interactive objects)
  - Declared using a **`tag`**

Two Types of Tags

1. Regular tag
   - Element can have nested elements or text
   - Requires a closing tag
   - E.g. section tag
2. Inline tag
   - Cannot have nested elements
   - Does not require closing tag
   - E.g. `img`

Basic HTML Tags

- `<html>` tag: the root element which contains all other elements
- `<head>` tag: the "invisible" part of an HTML document that specifies various information about the document
- `<body>` tag: the "visible" part of an HTML document
- `<header>, <main>, <footer>` tags: analogous to the header/main content/footer of a printed document
- `<h1>, ..., <h6>`: headings that are typically used to name a section of the document
  - h1 is the largest and h6 the smallest by default

- `<p>`: paragraph, which is a block of text
- `<a>`: anchor that defines a hyperlink

HTML Attributes

- Identifiers
  - `id`: specifies name of a unique element in the document
    - Hyperlink can include id to jump to that element (i.e. example_page#id)
  - `class`: specifies name(s) and class(es) in which elements with the same class share the same style and/or behaviour
    - Used extensively by CSS to style elements
- Some tags have required attributes (e.g. `<img>` requires the src attibute to specify the URL of the image)

Whitespaces in HTML

- Whitespaces in HTML files are *ignored*
- Can force a line break using `<br>` (which is rarely used)

Preformatted Text

- `<pre>` tag asks the browser to not ignore whitespaces inside the element
- The text may use characters that have a special meaning in HTML
- HTML entities represent characters reserved by the HTML language (e.g. `>` becomes `&gt;` and `<` becomes `&lt;`)

Organizing Elements

- Division tag `<div>`
  - Block level element, which changs to a new line wherever defined
  - Used extensively for organizing and styling elements
  - Can be styled to provide spacing and alignment of child elements
- Span tag `<span>`
  - Inline element
  - Allows styling of an inline element or text (i.e. using the `class` identifier)
  - Alternatives include `<em>` (emphasis) and `<strong>`

HTML Table

- `<table>` tag creates a table of rows and columns
- Each row is specified by the `<tr>` tag
- Each cell is specified by the `<td>` (table data) or `<th>` (table header) tag
- Can use `colspan` or `rowspan` to enable a cell to span multiple columns or rows

HTML Lists

- Unordered list `<ul>`: each list element is prefixed with a symbol (e.g. bullet)

- Ordered list `<ol>`: each list element is prefixed with an ordinal value (e.g. number)
- List item `<li>`: an item inside a list
- Description list `<dl>`: a list of key value pairs, where child elements must alternate between term `<dt>` and description `<dd>`

HTML Forms

- Primary way to send user data to server
- Specified by `<form>` tag
  - `action` attribute defines the URL of the HTTP request
  - `method` attribute defines which HTTP method will be used (see section 1)
- Input elements `<input>`:
  - E.g.

    ```
            <input class="mystyle" type="text" name="first_name" size="60" required>
    ```

  - Many other types
    - Text-based: `password, email`
    - File upload: `file`
    - Button-like: `radio, checkbox, submit`
  - `textarea` tag for multiline input
- Submit button that sends a request when pressed
  - Form data consists of key-value pairs of input name and their values

HTML Validation

- Helps identify issues with HTML code, including
  - Syntax error (e.g. missing a closing tag)
  - Semantic error (e.g. missing required elements)
  - Warnings (e.g. image tag without `alt` attribute)

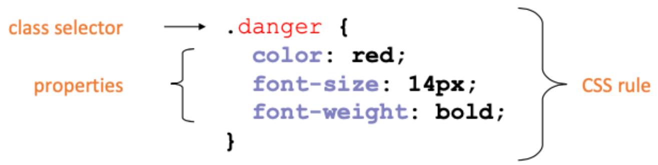
3 Cascading Style Sheet (CSS)

Web Standards Model

- Separation of content (HTML) and appearance (CSS)
  - Modern websites are dynamic – content is frequently updated
  - Modern websites strive for consistency in apperaance
  - this allows content and appearance to be updated independent of each other
- Other benefits
  - Accessibility – simplifies the work of screen readers
  - Device compatibility – web page can be rendered nicely on different devices
  - Search engine – helps search engine with parsing the web pages and avoiding misclassification

Cascading Style Sheet

- Describes the presentation of a document written in markup languages
- CSS **property**:
  - Defines the style of behaviour of an element
  - Syntax: `property name : property value(s);`
- Specifying properties
  1. Inline style
     - An attribute named "style"
     - Style only applies to this element
     - Syntax: `<div style="...">`
  2. CSS **rule**
     - One or more properties that apply to one or more elements
     - CSS **selector** determines which elements are targeted
     - Syntax: `selector {properties}`



```
.danger {
  color: red;
  font-size: 14px;
  font-weight: bold;
}
```

- Can write CSS rules in
  1. `<style>` tag
     - Needs to be inside the `<head>` element
     - Not recommended
  2. CSS file
     - Needs to be "imported" in the HTML file:

```
<link rel="stylesheet" href="style.css">
```

     - Can import multiple CSS files

    - `<link>` tag should go in the `<head>` element

CSS Cascade

- **Cascading**: multiple rules can affect the same element
- When two rules override the same property, a less specific rule is overridden by a more specific rule (1: least specific, 6: most specific)
  1. Order of appearanec (later is more specific)
  2. Elements and pseudo-elements (e.g. `p`, `h1`, `::before`)
  3. Classes, pseudo-classes, and attribute selectors (e.g. `.danger`, `:hover`, `[name]`)
  4. IDs (e.g. `#footer`)
  5. Inline style
  6. !important rule (e.g. `font-size:  1rem !important;`)

CSS Selector

- Element selector
  - Syntax: `tagname`
- Class selector
  - Syntax: `.classname`
- ID selector
  - Syntax: `#idname`
- Pseudo-class selector
  - Syntax: `:pseudoclass`
  - E.g. `:hover`
- Attribute selector
  - Syntax: `[attrname]`
  - E.g. `[href]`
- Pseudo-element selector
  - Syntax: `::pseudoelement`
  - E.g. `::first-letter`

Combining CSS Selectors

- AND condition
  - Syntax: join multiple selectors without space in between
  - E.g. `p.danger`
- OR condition
  - Syntax: join multiple selectors with comma in between
  - E.g. `h1, h2, p`
- Descendant condition

- Subsequent selector must be child or descendant of current selector
- Syntax: join multiple selectors with space in between
- E.g. `header nav a`

- Immediate child condition
  - E.g. `form > input`
- Adjacent sibling condition
  - E.g. `div + p` (selects `p` after a `div`)

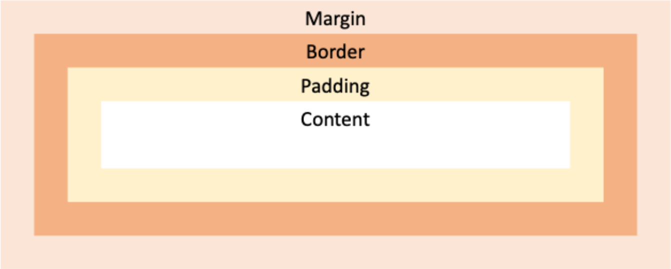
Font Properties

- `color`: selects text colour
- `font-family`: select one or more fonts, in that order (in case former one is unavailable)
- `font-style`: either `normal` or `italic`
- `font-weight`: `normal` and `bold` are the most commonly used
- `text-decoration`: `none, underline, overline, line-through`
- `text-align`: `left, right, center, justify`
  - Justify means all lines of text are same width
  - May want to use `hyphens` property with justify
- `text-size`: set size of text

Units

- Used by any property that specifies size or length
- Absolute units: `cm, in, px`
- Relative units:
  - `rem`: root element's font-size (default 16px)
  - `em`: parent element's font-size
  - `vh, vw`: current screen's (viewport) height/width
  - `%`: a percent relative to the size of the parent element
  - `fr`: fraction of the available space
- `calc` funcction allows arithmetic on different units (e.g. `width: calc(100% - 100px);`)

Box Model



- A set of boxes that wraps around every visible HTML elements
- The width and height of an element *includes* border, padding, and content, but *not* margin

Spacing Properties

- To specify margin/padding/border, use `margin/padding/border-width`
- To specify all edges:

```
        border-width: 1px 2px 3px 4px; (top right bottom left)
        margin: 0; (all edges)
        padding: 1rem 2px; (top/bottom left/right)
```

- Can specify specific edges using `top/right/bottom/left`, e.g.

```
        border-top-width: 1px;
        margin-bottom: 1rem;
```

- Margin can be negative to pull other elements closer

Border Properties

- `border-style`: `none, solid, dotted`
- `border-color`
- `border-radius`
  - Adds rounded edges to element
  - Can create a circle when radius is exactly half the width and height

Position Property

- Specifies how to position an element
- `static`: default behaviour, `top, bottom, left, right` properties are *ignored*
- `relative`: relative to its static position (i.e. where it would have been), makes it "positioned"
- `fixed`: relative to the *viewport*, i.e. stays in the same place on the screen
- `absolute`: relative to the nearest "positioned" ancestor, i.e. not static
- `sticky`: relative to the user's scroll position

Display Property

- Specifies how to render an element and/or its children
- Some display behaviours affects how child elements are placed
- `inline`: display the element as if it were an inline element
- `block`: display the element as if it were a block-level element
- `none`: do not display the element, as if it were removed
- `inline-block`: same as `inline`, but `width` and `height` properties are allowed

Responsive Design

- A web design approach where pages adjust themselves to “look good” on all screen sizes
- Viewport setup: required to ensure viewport adjusts to the current screen size:

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

- Responsive image: set width property to 100%
- Responsive text size
  - Set font size to a percentage or viewport width
  - Use clamp function, e.g. `font-size:  clamp(1rem, 10vw, 2rem);`
    - The three values are respectively the minimum, preferred, and maximum font size

Responsive Layout

- Allows elements to be placed flexibly depending on screen size
- `float` property
  - Useful for creating magazine-like layout
  - Should *only* be used to place an element to the side of a container, while allowing othe elements to flow around it
  - Should *not* be used to design page layout in modern days

Flexbox

```
display: flex;
```

- Flexibly places items inside the parent element, aka a *container*
- Container size automatically adjusts to size of child elements
- `justify-content`: `right, left, space-evenly, space-between, space-around`
  - Default is `left`
- `flex-wrap`: `wrap, nowrap`
  - Default is `nowrap`
  - `wrap` allows items to wrap to the next line
- `flex-direction`: `column, column-reverse, row, row-reverse`
  - `column` means items are stacked top-down
  - `column-reverse` means items are stacked bottom-up
- `align-items`: where to place child elements in a large container
  - `center, flex-start, flex-end`
- `align-content`: determines where to place each flex line (row/column) in a large container
  - `center, flex-start, flex-end, stretch, space-between`
  - Default is `stretch`

- To achieve perfet centering, use:

  ```
  .container {
     display: flex;
     justify-content: center;
     align-items: center;
  }
  ```

Flex Item

- **Flex item**: the direct child elements of a flex container
- `flex-grow, flex-shrink`: how much a flex item will grow or shrink relative to other items
  - Default is 0 (no growth/shrinkage)
- `flex-basis`: initial length of the flex item (width if row, height if column)
- `align-self`: overrides the container's `align-items` property

Grid Layout

```
display: grid;
```

- Grid supports 2-dimensional layout, similar to table
  - Use grid for layout and use table for tabular data
- `grid-template-columns`: for each column, specify a size value
  - E.g.

    ```
    grid-template-columns: auto auto auto;
    ```

- `grid-template-areas`: uses named grid items to specify rows and columns
  - E.g.

    ```
    grid-template-areas: 'menu top top'
                         'menu bot bot';
    ```

- `gap, row-gap, column-gap`: space between rows and/or columns

Grid Item

- **Grid item**: the direct child of a grid container
- `grid-column-start, grid-column-end`: allows an item to span multiple grid columns
  - Similar to `colspan` for `<td>`
  - E.g.

    ```
    .item1 {
       grid-column-start: 1;
       grid-column-end: 3;
    }
    ```

- `grid-row-start, grid-row-end`: allows an item to span multiple grid rows
  - Similar to `rowspan` for `<td>`
- `grid-area`: specify which named area this grid item belongs to
  - E.g. `.item1 { grid-area:  menu; }`

Media Query

- Checks the capability of the deviec before applying CSS rules
- Can completely change layout based on device
- Syntax:

```
@media <type> and (<expressions>) {
    <CSS rules>
}
```

- `type` is one of `screen, printer, speech`
- Mostly likely used expressions `min-width, max-width`
- E.g.

```
@media screen and (min-width: 480px) {
    ...
}
```

Browser Support

- Not all browsers support the same CSS properties
- E.g. range syntax in media queries

```
@media (100px <= width <= 1900px)
```

  instead of

```
@media (min-width: 100px) and (max-width: 1900px)
```

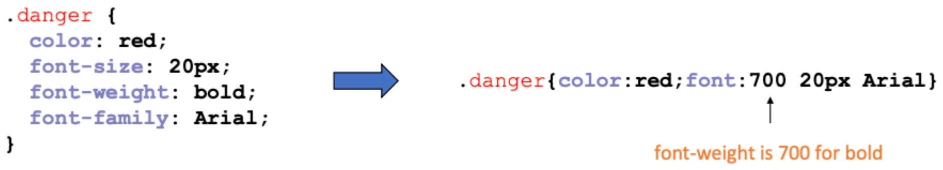
  is supported on Chrome and Firefox but not on Safari (as of Jan. 2023)

CSS Framework

- Ready-to-use CSS libraries (may include JavaScript)
- E.g. Bootstrap, Tailwind CSS, Bulma
- Provides basic and advanced interface components
- Suggested for most web development project
  - Easy to use and maintain consistent style
  - Speeds up development cycle
  - Browser compatibility is (mostly) handled by the framework
  - CSS optimization is already done

CSS Tools

- Minifier
  - CSS files are “compressed” to reduce file size and save bandwidth
  - Removes extra spaces, new lines, comments, etc.
  - Optimize for shorthands



```
.danger {
  color: red;
  font-size: 20px;
  font-weight: bold;
  font-family: Arial;
}
```

```
.danger{color:red;font:700 20px Arial}
```

font-weight is 700 for bold

- Linter
  - Performs syntax validation (like a compiler)
  - Performs style and formatting analysis

CSS Functions

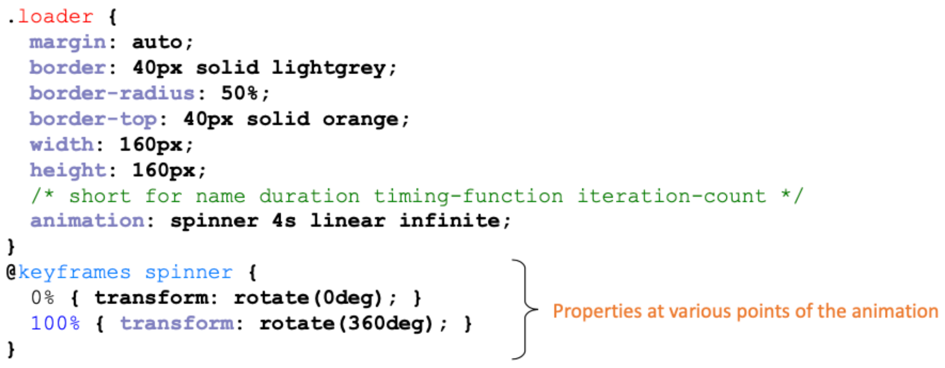
- `var()`
  - Uses a custom defined variable in plaec of a property value
  - E.g.

```
          :root { --main-bg-color: pink; }
          body { background-color: var(--main-bg-color); }
```

- `url()`
  - Use for properties that references a file (usually image)
  - E.g. `background-image: url("star.gif");`
- `max(), min()`
  - Selects the maximum or minimum of a set of values
  - E.g. `width: max(20vw, 400px);`

CSS Animations

- Allows animating HTML elements natively (without JS or Flash)
- E.g.

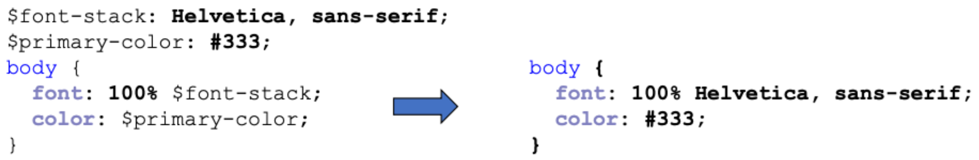


```
.loader {
  margin: auto;
  border: 40px solid lightgrey;
  border-radius: 50%;
  border-top: 40px solid orange;
  width: 160px;
  height: 160px;
  /* short for name duration timing-function iteration-count */
  animation: spinner 4s linear infinite;
}
@keyframes spinner {
  0% { transform: rotate(0deg); }
  100% { transform: rotate(360deg); }
}
```

Properties at various points of the animation

CSS Preprocessor

- A language on top of CSS that provides imperative programming features
- E.g. Sass, Less, Stylus
- Can declare variables, create loops, support inheritance
  - Helps with writing concise and maintainable CSS
  - Useful for making animations
- Interpreter (or compiler) translates preprocessor script into CSS
  - Done automatically as soon as the script is updated
- E.g.



```
$font-stack: Helvetica, sans-serif;
$primary-color: #333;
body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

4 Backend

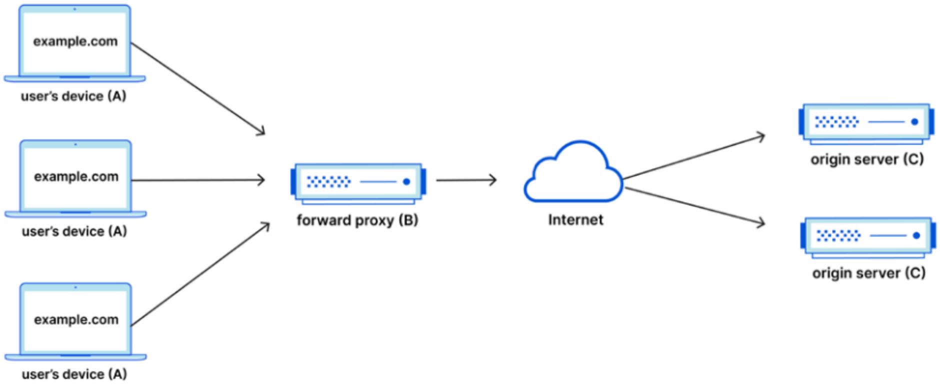
Web Development

- **Frontend**:
  - Focuses on *presentation*
  - Part of the *client*
  - Faces the *end-user*
  - Provides user-friendly interface
- **Backend**:
  - Focuses on *data access*
  - Part of the *server* (though server can do some frontend work)
  - Data storage and business logic

Web Server

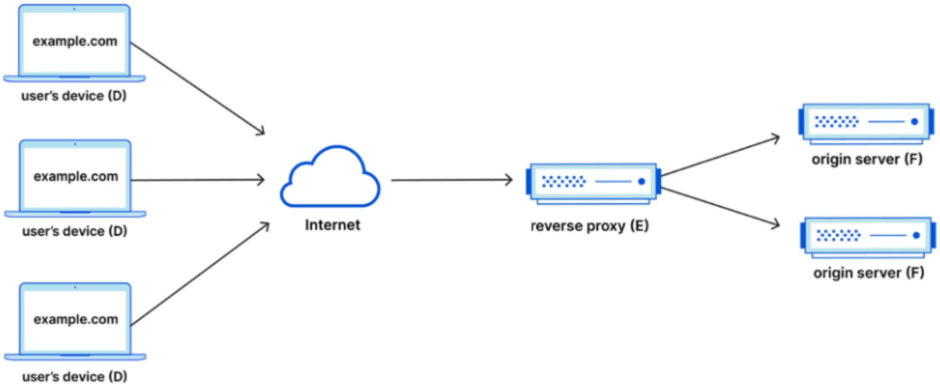
- Listens on specific port(s) for HTTP/HTTPS requests
- E.g. Apache, Nginx
- Handles incoming connections
  - Generates a response (dynamic content)
  - Fetches a file (static content)
    - ∗ Can be cached in memory for faster subsequent access
  - To act as a proxy between the client and the origin server
    - ∗ Forward proxy: sits in front of client devices, before Internet access
    - ∗ Reverse proxy: sits in front of origin server, after Internet access
    - ∗ This only works for HTTP requests (unlike VPN)

Forward Proxy



- Block or monitor access to certain content (e.g. on a school network)
- Improves security and anonymity by hiding user's IP address
- Can sometimes circumvent regional restrictions

Reverse Proxy



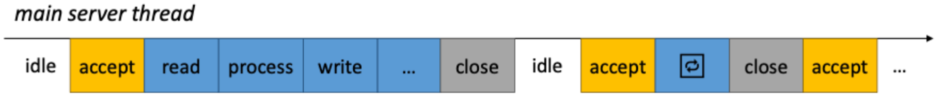
- Caches content for geographically distant web server
- Acts as a front for security purposes, e.g. encryption, prevent DDoS attack
- Provides *load balancing*

Load Balancer

- Popular websites can serve millions of concurrent requests
- **Load balancer** distributes incoming requests among backend servers to ensure all servers have similar utilization
- Allows adding/removing servers based on current demand, which reduces energy consumption

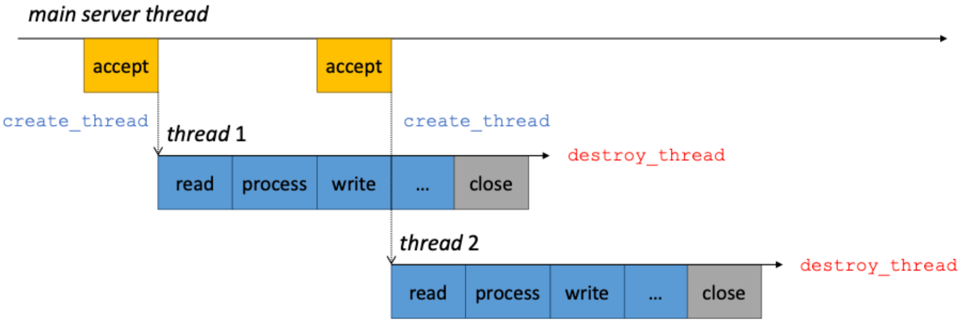
Web Server Architeture

- Single-threaded server



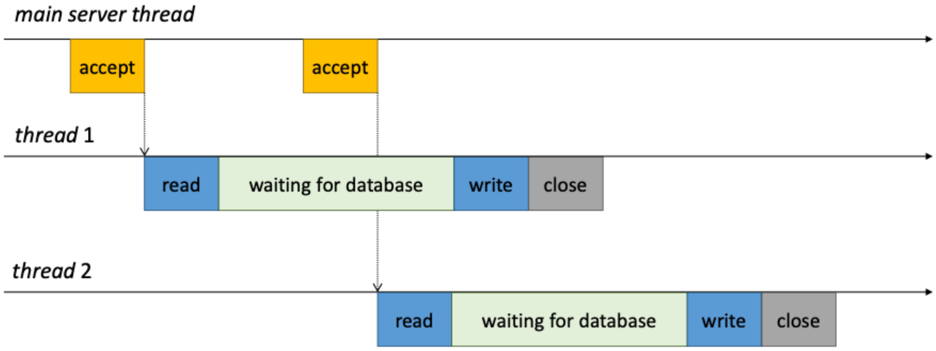
  - Problem: can only handle one connection at a time

- Multi-threaded server



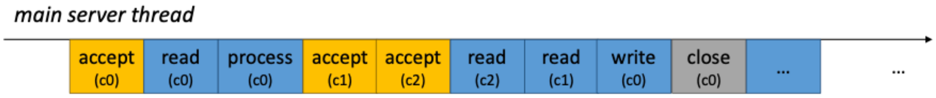
  - Problem: creating threads is expensive and HTTP request are short-lived

- Multi-threaded server with **thread pool**



  - Problem: threads are frequently blocked waiting for IO

- Event-driven web server



  - Events are queued and executed in order
  - An HTTP request can be broken up into *states*
  - Each state transition is an *event*, processed asynchronously
  - No overhead of switching between threads
  - Can be combined with thread pool to utilize more physical CPUs

Common Gateway Interface

- Allows web server to run an external program to process requests
- Separates web server from web application
  - Any web application can use any web server to generate dynamic content
  - Web application can be compiled or interpreted program
- Care is required due to execution of arbitrary code
- Many similar standards for CGI
  - WSGI (Web Server Gateway Interface): used by Python programs
  - Rack: used by Ruby programs
  - JSGI (JavaSript Gateway Interface): used by JavaScript programs

Programming Languages

- Any language can be used in the backend – it just needs a library that understands HTTP protocol
- E.g. Python, Java, JavaScript, php, Ruby
- Popular web programming languages are mostly *interpreted*
  - Portable: can run on many operating systems
  - Flexible: does not require compilation
  - Quick and easy to make changes on the fly

  - Bottleneck of most servers is *network*, not code execution

Runtime Environment

- Hardward and software infrastructure for running code
- It is *not* the programming language itself
- E.g.
  - CPython: interpreter that runs Python
  - Node.js: runs on V8 JavaScript engine, favorite among web developers since can write both frontend and backend with just 1 language
  - PHP interpreter: runs PHP

Backend Frameworks

- Libraries on the server-side that helps build a web application
- Avoids doing everything from scratch (e.g. listen on port, process HTTP request, retrieve data from storage, process data, create HTTP responses, etc.)
- PHP: Laravel, CodeIgniter
- Python: Django, Flask, FastAPI
- JavaScript: ExpressJS, Spring
- Ruby: Ruby on Rails

Python Project

- Require use of external packages
- Python's package manager: pip
  - Helps install and manage software packages
  - Automatically handles *dependencies* (i.e. other packages and their versions that are required to use a package)
  - E.g. `pip3 install Django`

Virtual Environment

- Manages separate package installations for different projects
- An *isolated* environment with its own version of everything (Python interpreter, pip, and packages)
- virtualenv provides lightweight encapsulation of Python dependencies
  - Lightweight: does not encapsulate the operating system
- Create a new virtual environment (with a specific Python version):

```
        virtualenv -p /usr/bin/python3.9 venv
```

- To activate the virtual environment:

```
        source venv/bin/activate
```

    - venv is the folder where the virtual environment is created

- To deactivate the virtual environment:

```
deactivate
```

- Packages will *not* be installed globally via pip
- To remove a virtual environment, delete the folder venv
- Can keep a text file that includes all the required package
    - So that when we recreate the virtual environment, we can install those packages by running

    ```
    pip install -r packages.txt
    ```

Start a Django Project

1. Create the project folder
2. Set up virtual environment and install Django
3. Run the command `django-admin startproject <name> .`
    - The dot means create the project in the current directory
    - This creates the skeleton code for the project
4. The following files are created:
    - manage.py: a command-line utility
    - A folder with the same name as the project, which contains project-wide settings

Development Server

- Used for testing and development only – *not* suitable for deployment
- To start the development server:

```
python3 manage.py runserver
```

- The website is accessible at http://localhost:8000
    - localhost: domain name for the current machine
    - 8000: port number

Django Apps

- Django is intended for big projects (i.e. hundreds of web pages, each with different URL)
- Project is organized into *apps*
- An **app** is a set of related concepts and functionalities
    - E.g. an app to manage accounts, another app to anage products
- To create a new app and its folder:

```
./manage.py startapp <name>
```

  - Contains views.py, migration folder, models.py, admins.py

- Need to add the app name to INSTALLED_APPS in settings.py, otherwise the app won't be loaded

Django View

- Code that runs when a specific endpoint (i.e. URL) is requested
  - Can be any callable object (i.e. function, calss that implements `__call__`)
- E.g.

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("hello")
```

- Funtion should take an argument, usually named request
- It should return an `HttpResponse` object

From URL to View

1. Create a file (preferably named urls.py) in the app folder

```
from django.urls import path
from . import views
urlpatterns = [ path('hello', views.hello), ]
```

2. Modify the project's urls.py (i.e. add an entry to the list named `urlpatterns`)

```
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    path('test/', include('testapp.urls')), # Add this line
    path('admin/', admin.site.urls),
]
```

- View is now accessible through URL /test/hello

URL Dispatcher

- Attempts to match URL from top to bottom of `urlpatterns`
- Can capture values from an URL and pass them to the view
- E.g.

```
path('hello/<str:name>/<int:age>', hello)
```

- The corresponding view function now takes 2 extra arguments:

```
def hello(request, name, age):
```

HTTP Request Data

- `request.method`: tells us which HTTP method was used to access this view
- `request.GET`: a dictionary of key-value pairs from query parameters (or URL parameter)
- `request.POST`: a dictionary of key-value pairs from POST requests
- `request.headers`: the HTTP headers of the request

Sanitization and Validation

- Sanitization: modifies input to ensure it is syntactically valid (e.g. escape characters that are dangerous to HTML)
- Validation: checks if input meets a set of criteria (e.g. check that passwords match and username is not blank)
- Should be checked at frontend for faster error feedback
- Should **always** be checked at backend as well (since users can bypass front-end restrictions)

Processing POST Request

- Validation error: return a 400-level error code if data is invalid
  - 400: Bad Request
  - 401: Unauthorized
  - 404: Not Found
  - 405: Method Not Allowed
- On success, a redirect is usually returned (e.g. redirect to profile page/index page after logging in)
- Use `HttpResponseRedirect` for redirect (from `django.shortcuts`) (e.g. `redirect(’/some/url’)`)

Named URL Patterns

- Django separates URLs that users see from the URLs developers use
- Developers should use **named URLs** instead of user URLs
  - User URLs may change, causing the redirects to break
- Add **name** or **namespace** argument to the path object
- E.g.
  - project’s urls.py

    ```
            path('accounts/', include('testapp.urls', namespace='accounts'))
    ```

  - account’s urls.py

    ```
            app_name = 'accounts'
            urlpatterns = [ path('', hello, name='hello'), ]
    ```

  - To redirect:

    ```
            reverse('accounts:hello')
    ```

Django Template Language

- Adds imperative programming features to making HTML files
  - Similar to PHP, where we run PHP code inside `<?php ...code... ?>`
- Variables: surrounded by {{ and }}
  - E.g.

```
<p>Hello, {{ username }}.</p>
```

- Tags: surrounded by {% and %}
  - Provides arbitrary logic in the rendering process
  - E.g.

```
{% if has_error %} <p class="error">Bad!</p> {% endif %}
```

Template Response

- Create a `templates` folder in the app's folder
- Convention: create subfolder with app name and put HTML files inside
  - E.g. the template path would be `<app_name>/hello.html`
- Use the `render` shortcut function to use Django templates
  - E.g.

```
from django.shortcuts import render

def signup(request):
    error = None
    code = 200 # succcess
    ...
    return render(request, 'accounts/hello.html', { # template path
        "error" : error, # passing template arguments
        "username" : username,
    }, status=code)
```

Cross-Site Request Forgery

- Unauthorized commands from trusted users
  - Can be transmitted by maliciously crafted forms, images, and JavaScript
  - Can work without the user's knowledge
  - E.g. hacking a user's browser to visit their bank account
- Prevention: use CSRF tokens
  - Add `{% csrf_token %}` to the form
  - The following will be generated:

```
<input type="hidden" name="csrfmiddlewaretoken" value="somerandomstringtoken">
```

  - Token value is unique each time the web page is generated

  - Attack becomes unable to authenticate the request without knowing the token

Static Files

- Django can manage static files (e.g. images, CSS, JavaScript)
- To use a static file, create a folder named `static` (recommended), and put the static files inside (or inside its subfolders)
- Add the following to settings.py:

```
STATICFILES_DIRS = [ BASE_DIR / "static", ]
```

- In the HTML file, can specify a static file as follows:

```
{% load static %} <!-- load this template tag -->
<!DOCTYPE html>
...
<img src="{% static 'me.jpg' %}" alt="me">
```

- Django development can serve static files (not suitable for production use)
- Add the URLs of the static file to `urlpatterns` in urls.py

```
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

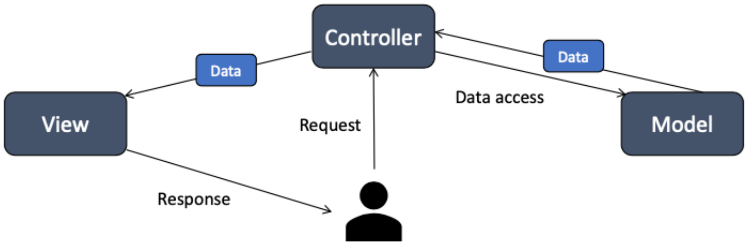
5 Django Templates and Models

Architectural Design Patterns

- 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)
  - **MVC/MVT**: model-view-controller/model-view-template
  - **MVP**: model-view-presenter
  - **MVVM**: model-view-viewmodel

Model-View-Controller (MVC)

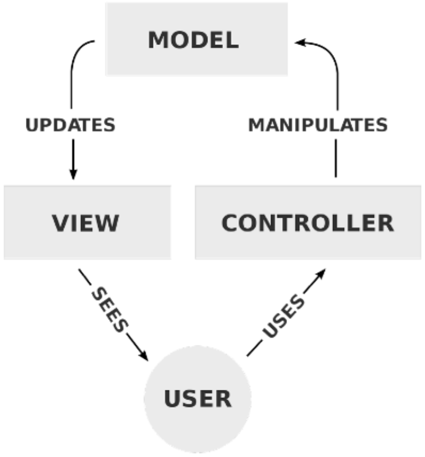
- Focuses on the separation of appearance (view), data (model), and business logic (controller)



- Easy to switch out presentation or data source

Alternative MVC Pattern

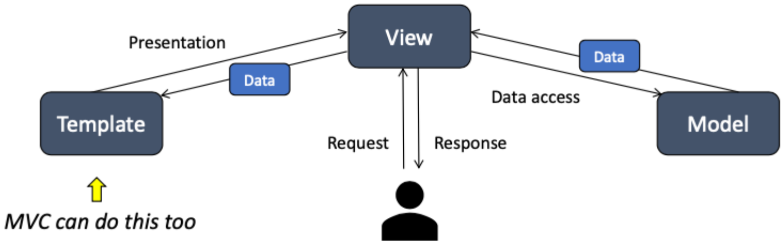
- Model notifies view of update



- Both controller and model can handle business logic
- Fat model and skinny controller
- Model should handle domain-specific knowledge (e.g. acccount management)
- Controller should handle application logic only (e.g. ask for password)

Model-View-Template (MVT)

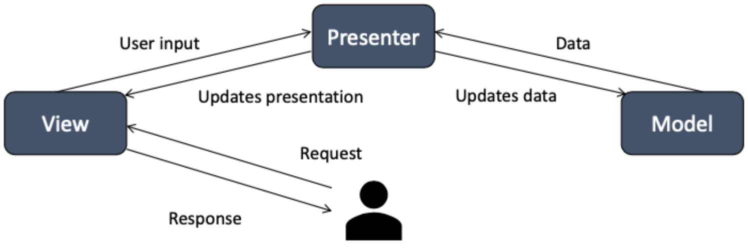
- Same as MVC, except it uses Django's terminology



- Django view = MVC controller
- Django template = MVC view
- URL dispatcher is part of MVC controller
  - Classical controller does not have one, but modern ones do (e.g. Spring MVC)

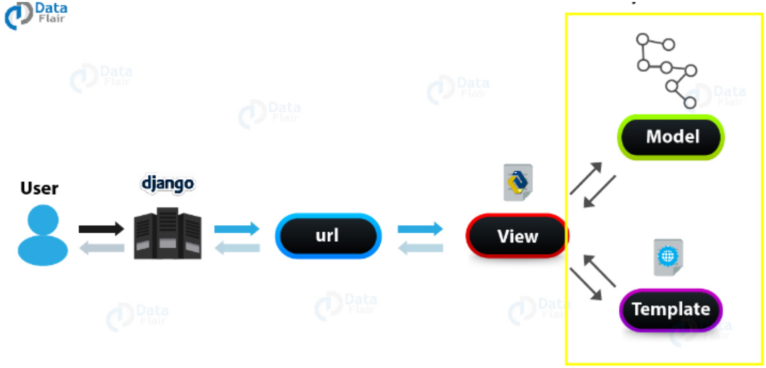
Model-View-Presenter (MVP)

- User communicates with the view



- Breaks dependency of model from view

Django's Architecture



Django Template Language

- For loop

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

- If statement

```
        {% 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 %}
```

- To map named URL to user URL (same as `reverse` function):

```
        {% url 'namespace:name' %}
```

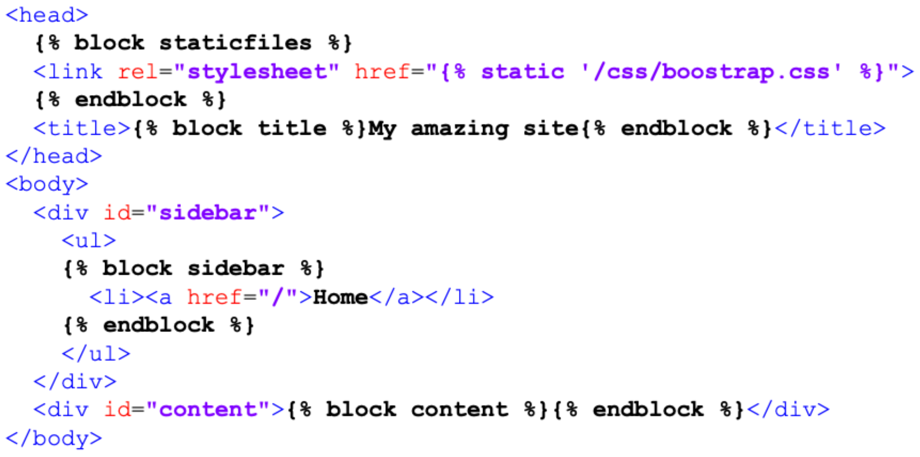
- If tags can take relational operators (e.g. `==`, `>`, `in`)
- Members variable, ditionary lookup, index access all use dot operator (e.g. `user_list.0`, `request.POST.username`)
- Django template comment: `{# comment #}`

Django Template Filters

- Similar to a function that modifies a variable for display
- Syntax: pipe character followed by filter name, i.e. `{{ var | filter }}`
- `length`: same as Python `len()`
- `lower`: same as Python `str.lower()`
- `time`: formats time object
  - E.g. `{{ value | time:"H:i"}}` $\implies 10:05$
- Filters can be chained, e.g. `{{ value | first | upper }}`

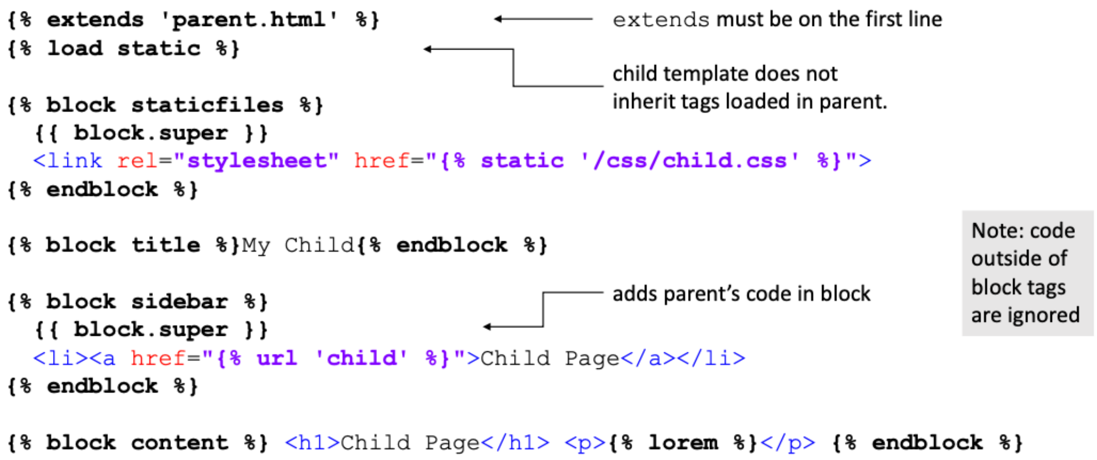
Django Template Inheritance

- Parent templates define *blocks* that child templates can override
- E.g.
  - Parent:



```
<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>
```

- Child:



```
{% 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 belong to only 1 view (which should be placed inside the app folders)
- Some templates have common components across apps (e.g. navagation bar, footer)
- Reusable templates can be placed in a **root template directory**:

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

Django Include Tag

- Render a subtemplate and include the result
- `{% include template_name %}`
  - Can be a variable, absolute path, or relative path
- Subtemplate is rendered with the current context
- Can pass additional context, e.g. `{% include "greeting.html" with person="Bob" %}`
- Can restrict context to only ones explicitly passed in, e.g. `{% include "greeting.html" with person="Bob" user=user only %}`

Database

- Most web applications need a persistent storage
- Database: collection of data organized for fast storage and retrieval on a computer
- Choice for primary database:

  - Relational: MySQL, PostgreSQL
  - Non-relational (NoSQL): Cassandra, MongoDB

- Django supports various database backends transparently through an **object relational mapper**

Object Relational Mapper (ORM)

- Provides an *abstraction* for accessing the underlying database
- Separates application from database implementation
  - If we connect to a specific database using its client (e.g. MySQLdb), 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 ha a built-in ORM layer
- Other ORM frameworks: SQLAlchemy (Python), Hibernate (Java), Sequelize (JavaScript)
- Advantages
  - Simplicity: no need to learn SQL or other database languages
  - Consistency: everything is in the same language (e.g. Python), which 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 vie versa
- Disadvantages
  - 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
- Follows standard SQL syntax
- No setup or installation required, good for development
- However, 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
- The ORM layer enables defining models with *classes*
- Django has a set of predefined models for convenience
  - User: default model for authentication and authorization
  - Permissions: what a use 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 current 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

- `User` is a derived class of `AbstractUser`
- Contains preefined fields, such as 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 hash values to revert the 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
- To add/update tables, run the `migrate` command

```
python3 manage.py migrate
```

  - The ORM layer will create or update the database tables
- Django shell
  - Provides interactive Python shell within Django environment
  - Helps to test models without running a web server

```
python3 manage.py shell
```

Working With ORM Objects

- Create an object:

```
User.objects.create_user(
    username='js',
    password='123',
    first_name='Jack',
    last_name='Sun'
)
```

    - Some fields are optional (e.g. `first_name`, `last_name`)

- Get all objects of the same type:

```
        users = User.objects.all()
```

- Get just one object based on exact match:

```
        jack = User.objects.get(username='js')
```

    - Can return "not found" or "not unique"

- Delete object(s)

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

Working With ORM

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

QuerySets

- Evaluated lazily, i.e. queries are not run until field of object is accessed
- Can define query sets sequentially (without them being actually ran)

Update Queries

- Update a single instance:

```
        js = User.objects.get(first\_name="Jack")
        js.first\_name = "Kuei"
        js.save()
```

- Update everything in a `QuerySet`

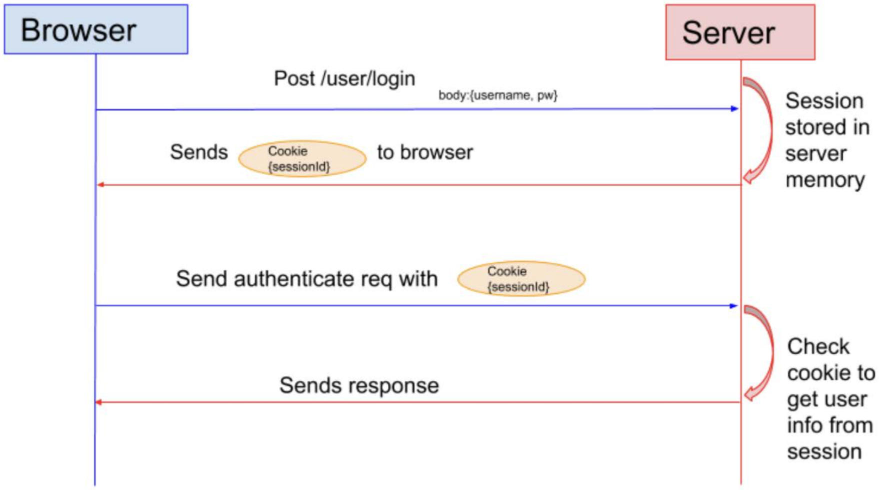
```
        User.objects.filter(is\_active=True).update(is\_active=False)
```

- Attributes are locally cached values
  - To refresh, use

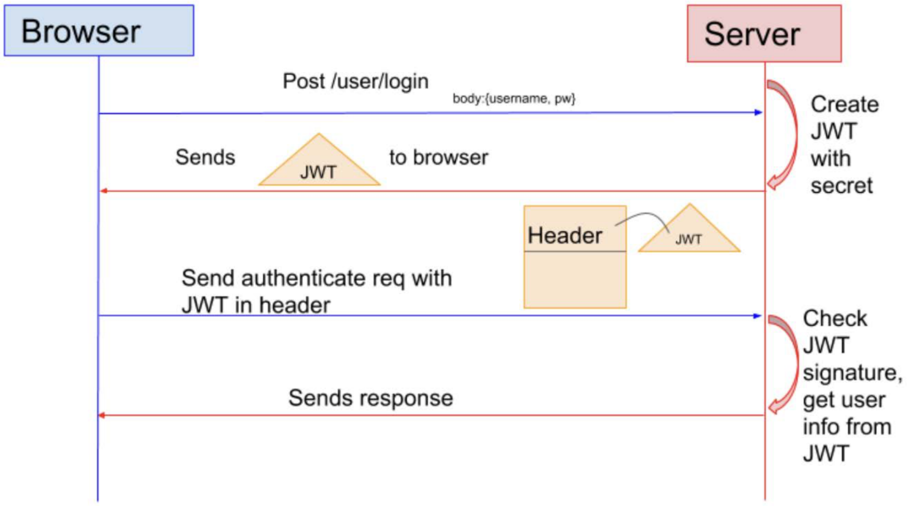
```
            js.refresh_from_db()
```

Authentication

- Clients should tell the server who they are
- Can use Authorization header in HTTP
- Authentication methods:
  1. Basic password authentication
     - Sends username and password for every request (no concept of login/logout)
     - User information is unencrypted, which is insecure without HTTPS
  2. Session authentication
     - Client only sends usernamd and password at login
     - If successful, server creates and stores a session ID (which is mapped to to the specific user)
     - Session ID is returned in the response, which the browser saves in cookies
     - For subsequent requests, browser sends the session ID
       - ∗ Does not work for Incognito, since browser does not send cookie



  3. Token authentication
     - Token is *signed* by server to avoid attacks (which is used to identify the client and their permissions)
     - Much faster than session because no database query is needed
     - JSON web token (JWT): industry standard method for securely representing *claims*
     - Claims: can contain the user's information, including identity and/or permissions



Django Session Authentication

- Cheks that username/password combination is correct

```
        user = authenticate(username='js', password='secret)
```

- Django's `login` function

```
        login(request, user)
```

  attaches user to current session

- Django does the session ID lookup internally
  - User object is attached to the request object (i.e. `request.user`)
  - User type is `AnonymousUser` if current visitor is not authenticated
- `logout()` function removes session data

Admin Panel

- A convenient Django service to manage database records
- Installed by default (see global `urls.py`)
- Can be found in `http://localhost:8000/admin`
- Requires an active user with `is_superuser` and `is_staff` field set to True
  - Can be created manually through the shell:

```
            ./manage.py createsuperuser
```

6 Django Models

Designing Models

- Models involve representing and storing user data
- Modelling should be done *before* coding
- Changing models become more difficult as the project grows
- Can be done independent of programming language or framework (e.g. UML, ER diagrams)

Creating Models

- In `models.py`, add subclasses of `django.db.models.Model`
- Add *fields* from the diagram to each model

```python
from django.db import models
class Store(models.Model):
    name = models.CharField(max_length=255)
    email = models.EmailField()
```

  - Each field is mapped to a database column by the ORM layer
- Convention: create a `models` directory and put eacfh model in a separate file
  - Add `__init__.py` and import each model

Django Fields

- Each field type maps to a primitive type in the database
  - Strings
    - `CharField` for small amount of text
    - `EmailField, URLField` checks for valid format
    - `TextField` for large amount of text
  - Files
    - Need to specify where to save the files
    - `FileField, ImageField`
  - Numbers
    - `IntegerField`
    - `BigIntegerField` for large numbers (i.e. at least 64-bit)
    - `FloatField, DecimalField` maps to Python `float` and `decimal`
  - Time
    - `DateField, DateTimeField`
  - True/False
    - `BooleanField`

Django Field Options

- Every field can be restricted/checked in some ways
- `null: bool = False` allows the lack of value

- `blank: bool = False` allows the field to be unspecified
- `unique: bool = False` requires value to be unique (throughout database table), otherwise throws `IntegrityError`
- `choices: [(Any, Any), ...]` restricts the field to a set of values displayed to user
  - A list of key-value pairs
  - Keys should be an abbreviation
- `max_length: int` limits the number of characters (only for `CharField` and its subclasses)
- `default: Any` sets the default value for the field

Foreign Key

```
models.ForeignKey(to, on_delete, related_name, ...)
```

- Used for many-to-one and one-to-many relations
  - Defined at the "many" end as part of the foreign key
  - Stores only the primary key in the database column
- `on_delete` determines the behaviour when referenced object is deleted
  - `CASCADE`: delete everything that references the deleted object
  - `SET_NULL`: set reference to NULL
- `related_name` provides alternative name for reverse traversal by a field
  - Defualt is `<model_name>_set`, e.g. `prof.class_set.all()`

Models to Tables

- Every time the model changes, we must create and run *migrations*

```
./manage.py makemigrations
./manage.py migrate
```

- By default, Django ccreates an `AutoField` named `id`
  - Configurable in `<app_name>/apps.py`
  - Used as the primary key of the table
  - Can be overridden with `primary_key=True` for another column

Admin Panel

- Can register model in the admin panel
  - Add `admin.site.register(Store)` to `admin.py`
- Field options for admin panel (and also form)
  - `help_text` adds help text in tooltip
  - `verbose_name` gives alternative name for the field
- `__str__()` is called then typecasting Python object to `str`

  - Admin panel does this when displaying a list of objects

Model Inheritance

- `OneToOneField` defines a one-to-one relationship
  - Not frequently used, same thing can be done with *inheritance*
- Model inheritance: creates an additional table with a pointer to the base class

```python
class Product(models.Model):
    name = modelsCharField(max_length=255)
    price = models.FloatField(default=0.,)
    store = models.ForeignKey(Store, on_delete=models.CASCADE)

class Produce(Product):
    expiry_date = models.DateField()
```

Many-to-Many Relationship

- `ManyToMany` field defines a many-to-many relationship (e.g. classes and students)
- By default, an intermediary join table is used to represent the relationship
- Supports recursive relationship (`ForeignKey` can do the same)

```python
class Student(models.Model):
    friends = models.ManyToManyField("self") # Symmetrical relationship
```

- Can specify the intermediary table manually

```python
class Student(models.Model):
    classes = models.ManyToManyField(Class, through='Enroll')

class Enroll(models.Model):
    student = models.ForeignKey(Student)
    klass = models.ForeignKey(Class)
    grade = models.CharField(max_length=3)
```

Working With Relationships

- `add` method associates two objects in a one-to-many or many-to-many relationship

```python
Store.objects.create(name='Apple', url='apple.com')
apple = Store.objects.filter(name__contains='Apple').first()
user = User.objects.get(username='js')
apple.users.add(apple)
```

- Can access foreign object(s) through current object
  - May require multiple database queries, which could impact performance

```python
apple.refresh_from_db()
apple.owner.first_name = 'Kuei'
apple.owner.save()
```

  - `select_related(fieldname)`: grab related object in a single query

    ```
            Store.objects.select_related('owner').get(id=1)
    ```

File Upload

- Only the file's *local path* is stored
- In `settings.py`, create the media root folder and its URL

  ```
          MEDIA_ROOT = BASE_DIR / 'media'
          MEDIA_URL = 'media/'
  ```

  - By default, the `upload_to` folder is created in the project directory
- Browser sends a separate request to access the file (where Django translate request to a file access)
- For images, must install the `pillow` package
- To access uploaded files, must register with URL dispatcher

  ```
          static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
  ```

7 Migrations

ORM Layer

- Assumption: the state of database tables is the same as the definitions in model classes
- Reality: the two can become out-of-sync whenever the model changes
- ORM must apply *current* application schema to the database (which requires running DDL queries)
- Whenever the model changes, database should *migrate* to the new state
- Django does not perform migration automatically (to avoid data loss)
- Django does not monitor potential model changes (i.e. if there is a mismatch, database exception will occur when executing queries)
- Two stetps to perform migration
  1. Make migration
  2. Migrate

Make Migration

- Generates a list of operations needed to migrate to new state
- History of changes is stored in the `migrations` folder for each app (each migration also tracks a list of *dependencies*)
- To generate a migration, run

```
        ./manage.py makemigrations
```

- Builds a temporary model state from previous migrations (by replaying all migrations in order, e.g. from 0001 to current)
- No data operations are executed (since migrations are written in a database-neutral way)
- *Temporary* model state is compared with *current* model state
- From the differences, a list of operations is generated (i.e. a new migration file is created with a `Migration` class defined)
- For the command to work, `__init__.py` must be present in the `migrations` folder

Applying Migrations

```
    ./manage.py migrate
```

- DDL queries are generated from each migration file, then, the are executed
- Django knows which migration has not been applied
  - Migration information is stored in the database (in the table `django_migrations`)
  - Table only includes metadata (e.g. name, app, time applied, etc.)
  - Actual operation is stored in the migration files
  - The `migrate` command only applies migrations not in the table

Migration Error

- We never need to manipulate migration files/table
- Lots of assumptions are in the implementation of migrations (e.g. deleting a migration file may impact future migrations)
- Migrate errors can be difficult to solve
- Possible solutions: *unapply* or *fake* a migration

Unapply Migration

```
./manage.py migrate <app> <last_migration_name>
```

- Rolls back all changes to a previous migration state
  - Data loss is possible
  - Can create a full backup of the database in a JSON file:

    ```
              ./manage.py dumpdata > db.json
    ```

  - Can load the database with data from backup file:

    ```
              ./manage.py loaddata db.json
    ```

- The corresponding row in `django_migrations` is deleted
  - Can delete the migration file that was unapplied
  - Do *not* delete a migration file before it is unapplied

Fake Migration

```
./manage.py migrate --fake
```

- Only creates a row in `django_migrations` without executing any database queries
- Use when the database state is already consistent with the models

Full Reset

1. Delete the enttire database
2. Delete all migrations files to start over from fresh

Class-Based Views

- Function-based views can become too big

8 Advanced Views

Class-Based Views

- Function-based views can become too big, since one view may need to support multiple HTTP methods
- Class-based view
  - A subclass of `django.views.View`
  - HTTP requests are routed to methods of the respective names (e.g. HTTP GET request will call `view.get(request, ...)`)
  - A new instance of the object is created for every request
- Convention: create a `views` directory and put each view in a separate file
  - Add `__init__.py` and import each view
  - In `urls.py`, each class-based view must call the `as_view()` method

Comparison Between Views

```
def simple_view(request, id):
    if request.method == "GET":
        return HttpResponse(
                f"My ID is {id}")
    elif request.method == "POST":
        return redirect("accounts:login")
    else:
        return HttpResponseNotAllowed()
```

```
from django.views import View
class SimpleView(View):
    def get(self, request, id):
        return HttpResponse(
                f"My ID is {id}")

    def post(self, request, *a, **k):
        return redirect("accounts:login")
```

- In `urls.py`:

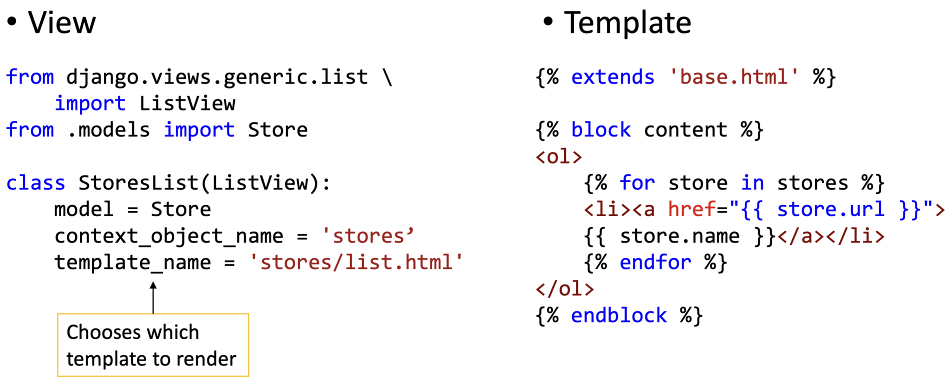
```
urlpatterns = [
    path('func/<int:id>/', simple_view, name='simple_func'),
    path('cls/<int:id>/', SimpleView.as_view(), name='simple_cls'),
]
```

CRUD Views

- Create-Read-Update-Delete
- Most views fall under one of these categories
- Django provides CRUD base classes for these views
- Generic display views: designed to display data
  - `DetailView, ListView`
- Generic editing views: designed to create, update, or delete data
  - `CreateView, UpdateView, DeleteView, FormView`

List View

- A page that displays a list of objects



- View

```
from django.views.generic.list \
    import ListView
from .models import Store

class StoresList(ListView):
    model = Store
    context_object_name = 'stores’
    template_name = 'stores/list.html'
```

Chooses which template to render

- Template

```
{% extends 'base.html' %}

{% block content %}
<ol>
    {% for store in stores %}
    <li><a href="{{ store.url }}">
    {{ store.name }}</a></li>
    {% endfor %}
</ol>
{% endblock %}
```

Display View Attributes

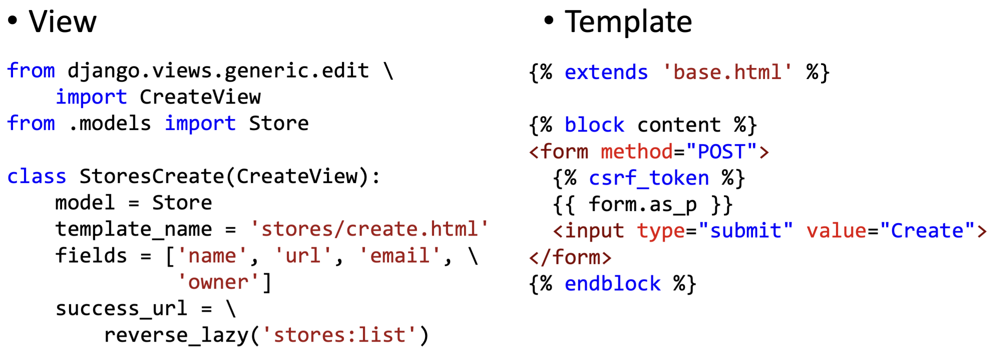
- models: the model of the generic view, assumes query set is entire table
- context_object_name: is the object_list or object (DetailView) by default, allows alternative context name
- queryset: same as model, but allows specifying a subset or ordering

```
       Store.objects.filter(is_active=True)
```

- get_context_data(self, **kwargs): override to add extra context
- get_queryset(self): override to customize query set
- get_object(self): override to retrieve object (DetailedView only)
- URL arguments: stored under self.kwargs (e.g. self.kwargs['pk'])
- Request object: stored under self.request

Create View

- A page that allows for creating objects
- On GET request, returns blank form
- On POST request, redirect on success, redisplay form upon error



- View

```
from django.views.generic.edit \
    import CreateView
from .models import Store

class StoresCreate(CreateView):
    model = Store
    template_name = 'stores/create.html'
    fields = ['name', 'url', 'email', \
              'owner']
    success_url = \
        reverse_lazy('stores:list')
```

- Template

```
{% extends 'base.html' %}

{% block content %}
<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Create">
</form>
{% endblock %}
```

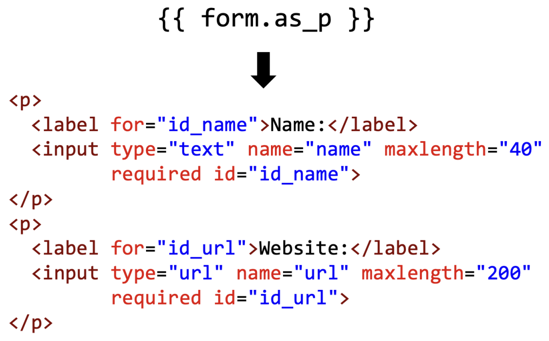
Editing View Attributes

- fields: a list of fields in the model to edit (does *not* have to be every field)

- `success_url`: redirect URL on success
  - Must use `reverse_lazy` here
- `get_success_url(self)`: needed if `reverse` needs argument, e.g.

```
        reverse('stores:detail', kwargs={'pk': self.kwargs['pk']})
```

- Django form: helps with all aspect of form (e.g. render HTML, validation, update associated model)
- All edit views have a `form` context



```
{{ form.as_p }}
```

```
<p>
  <label for="id_name">Name:</label>
  <input type="text" name="name" maxlength="40"
         required id="id_name">
</p>
<p>
  <label for="id_url">Website:</label>
  <input type="url" name="url" maxlength="200"
         required id="id_url">
</p>
```

9 Django Forms

Django Forms

- An abstraction for working with HTML forms
  - Frontend: renders form, converts Django fields to HTML input elements
  - Backend: sanitizes and validates form data
- `Form` class: similar to Django model class

```
from django import forms
class NameForm(forms.Form):
    name = forms.CharField(label='Your name', max_length=100)
```

⬇

```
<label for="id_name">Your name:</label>
<input type="text" name="name" maxlength="100" required id="id_name">
```

Making Django Form

- Convention for large projects: create a `forms` directoryt and put each form class in a separate file
  - Add `__init__.py` and import each form class
- `clean` method: performs *validation* (sanitization has been done already)

```
    def clean(self):
        data = super().clean()
        user = authenticate(username=data['username'], password=data['password'])
        if user:
            data['user'] = user
            return data
        raise ValidationError({'username': 'Invalid username or password'})
```

  - Override to add custom logic

Model Form

- Form that maps closely to Django model

```
    class ArticcleForm(forms.ModelForm):
        class Meta:
            model = Article
            fields = ['title', 'content', 'image']
```

- `Meta` inner class defines the associated model and the fields that appear in the form
- `save` method create or update the associated Model object

```
    f = ArticleForm(request.POST)
    article = f.save()
```

Using Django Form

- With a function-based view:

```python
def get_name(request):
    if request.method == 'POST':
        form = NameForm(request.POST)
        if form.is_valid():
            return HttpResponseRedirect('/thanks/')
    else:
        form = NameForm()
    return render(request, 'name.html', {'form': form})
```

- With a class-based view:

```python
class NameView(FormView):
    form_class = NameForm
    template_name = 'name.html'
    success_url = '/thanks/'
```

Form Widgets

- Forms can be passed into template and rendered
- Some form fields can be rendered differently
  - E.g. a `CharField` can be rendered as text input, password input, textarea, etc.
  - Specify a widget to custommize the rendering

```python
class LoginForm(forms.Form):
    usename = forms.CharField(max_length=150)
    password = forms.CharField(widget=forms.PasswordInput())
```

- Not recommended for large projects since view should be separate fromm controller in MVC pattern

Form View

- One of Django's generic editing view
- Similar to other CRUD views, but more customizable
- `form_valid` method is called when form is valid (i.e. the POST request contains valid data)

```python
class LoginView(FormView):
    form_class = LoginForm
    template_name = 'accounts/login.html'
    success_url = reverse_lazy('accounts:admin')
    def form_valid(self, form):
        login_user(self.request, form.cleaned_data['user'])
        return super().form_valid(form)
```

- `form_invalid` method overrides to custom handle invalid data

`CreateView` and `UpdateView`

- `CreateView` class: a subclass of `FormView` whose `form_class` is a `ModelForm`
- `UpdateView` class: a subclass of `CreateView` that implemments the `get_object` method

- A default form_valid method is implemented that saves the object (can return super().form_valid(form) to save the model object)

Authenticacted Views

- Simplifies views where user must be logged in
- Function-based views:

```python
from django.contrib.auto.decorators import login_required
@login_required(login_url=reverse_lazy('accounts:login'))
def admin(request):
    return render(request, 'accounts/admin.html', {})
```

- Class-based views: (requires login_url to be specified for redirect)

```python
from django.contrib.auto.mixins import LoginRequiredMixin
class DeleteUserView(LoginRequiredMixini, DeleteView):
    model = User
    login_url = reverse_lazy('accounts:login')
    success_url = reverse_lazy('accounts:admin')
```

10 REST APIs

Full Stack Framework

- Django is a **full-stack framework**, i.e. library that do both backend and frontend work
- Server responsible for serving static files and handling business logic
- Design *couples* backend and frontend
  - Poor separation of duties
  - Can't use a dedicated frontend framework (like *React*)
  - Restricts and/or complicates other types of *rendering pattern*
- Rendering pattern: the way HTML is rendered on the web
  - Django primarily supports *server-side rendering*

Separating Frontend and Backend

- Enables one backend and *multiple frontends*
- Improves *modularity*, i.e. changes in frontend will not affect backend and vice versa

Web API

- Different services and/or applications talk to each other (with a preestablished protocol)
- API (application programming interface): the way in which applications communicate with each other
- Web applications typically communicate via HTTP requests
- Backend views are responsible for data retrieval and manipulation
  - Should *not* care about how data is presented

JavaScript Object Notation (JSON)

```
[
  {
    "_id": "63ea43564bfe5fbf662a2e76",
    "index": 0,
    "guid": null,
    "isActive": false,
    "balance": "$3,863.93",
    "picture": "http://placehold.it/img",
    "age": 20,
    "name": "Duffy Sanchez",
    "friends": [
      { "id": 0, "name": "Rosie Crell" },
      { "id": 1, "name": "Eaton Mars" },
    ],
    "favoriteFruit": "strawberry"
  }
]
```

- Popular standard for backend responses
- Derived from JavaScript syntax for defining objects (which simplifies use in a browser since naitively support JavaScript)
- Advantages

  - Easy to read, easy to use, fast
  - Many programming languages have built-in parser and support

- Primitives types: number, string, boolean, null
- Array: ordered collection of elements
- Object: key-value pairs (key must always be a string)
- Array elements and object values can be of *any* type

Web APIs

- REST (Representation State Transfer)
  - A particular architectural style with a set of constraints and principles
  - Goal is to create a scalable, maintainable, and flexible system

  1. Uses HTTP verbs to make requests (e.g. GET, POST, PUT, etc.), and resources should be identified through URIs
  2. Requires stateless client-server communication
  3. Responses should be clearly labelled as cacheable or non-cacheable
  4. Client should only interact with the API and not server directly

- SOAP (Simple Object Access Protocol)
  - XML-based protocol with standardized format for data transfer
  - Less popular comparing to REST

11 Django REST Framework (DRF)

Django REST Framework

- Helps with writing RESTful APIs
- Provides JSON parser, CRUD views, permissions, and serializers
- Only uses Django's backend
  - Models and URLs are unchanged
  - Views are subclasses of DRF views
- Installation

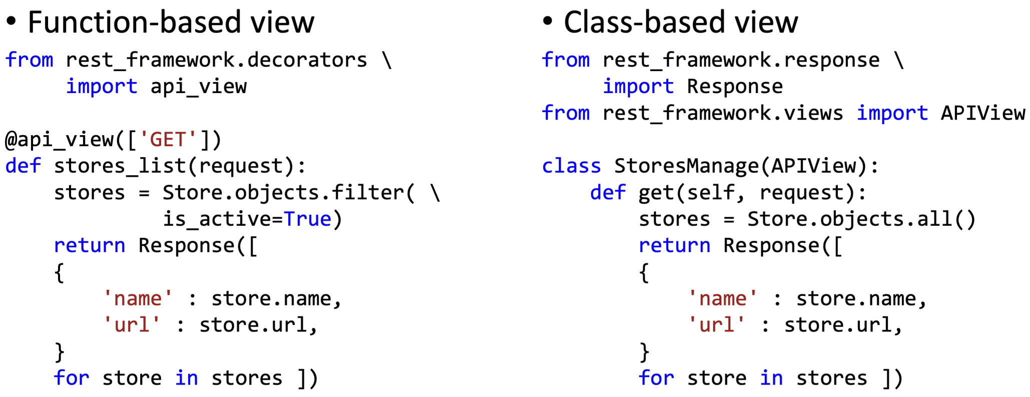
```
pip3 install djangorestframework
```

  - Add `rest_framework` to `INSTALLED_APPS` in `settings.py` and the following:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny'
    ]  # No authentication required, do not use
}
```

REST Views

- Returns a REST `Response` class
  - Takes a list or a dictionary, and converts it to an HTTP JSON response



- Function-based view

```
from rest_framework.decorators \
     import api_view

@api_view(['GET'])
def stores_list(request):
    stores = Store.objects.filter( \
             is_active=True)
    return Response([
    {
        'name' : store.name,
        'url' : store.url,
    }
    for store in stores ])
```

- Class-based view

```
from rest_framework.response \
     import Response
from rest_framework.views import APIView

class StoresManage(APIView):
    def get(self, request):
        stores = Store.objects.all()
        return Response([
        {
            'name' : store.name,
            'url' : store.url,
        }
        for store in stores ])
```

Model Serializer

- Model instances need to be *serialized* and *deserialized* for client
- Object represented in formmat that can be *transferred* and *reconstructed*
- DRF provides JSON serializer
  - Very similar to Django's `ModelForm`
  - Plain serializer (not mapped to a model) is also available
- Create a `serializer.py` or a `serializer` directory in the app

```python
from rest_framework.serializers import ModelSerializer

class StoreSerializer(ModelSerializer):
    class Meta:
        model = Store
        fields = ['name', 'url', 'email', 'is_active']
```

REST CRUD Views

- Requires a model serializer
- `CreateAPIView`: overrides `create` method
  - Returns 201 Created on success
  - Accepts HTTP POST
- `RetrieveAPIView, ListAPIView`: overrides `retrieve` method
  - Returns 200 OK on success
  - Accepts HTTP GET
- `UpdateAPIView`: overrides `update` method
  - Returns 200 OK on success
  - Provides HTTP PUT and PATCH method handlers
- `DestroyAPIView`: overrides `destroy` method
  - Returns 204 No Content on success
  - Provides HTTP DELETE method handler
- `ListAPIView`: reqquires `queryset` attribute or `get_queryset` method
- `RetrieveAPIView, UpdateAPIView, DeleteAPIView`: requires `get_object` method
- `CreateAPIView`: does not require any addition method or attribute
- Can mix multiple views in one class (i.e. multiple inheritance) as long as each view uses a different HTTP method
- Can use same serializer across different views

```python
from django.shortcuts import get_object_or_404
from rest_framework.generics import RetrieveAPIView

class StoresRetrieve(RetrieveAPIView):
    serializer_class = StoreSerializer
    def get_object(self):
        # Returns 404 NOT FOUND if the object is not found
        return get_object_or_404(Store, pk=self.kwargs['pk'])
```

- Can perform testing using Postman or DRF's built-in browsable APIs

Serialization Fields

- Fields have similar options to Django's model field

    - Exceptions:
        - null: `allow_null`
        - blank: `allow_blank`
    - `read_only`: makes a field non-writable

- Field validations are done automatically

- Foreign key: serializes to ID of referenced object by default

- Custom fields: can create new fields or override existing fields

```
class StoreSerializer(ModelSerializer):
    owner_username = CharField(read_only=True, source='owner.username',
        allow_null=True)
    ...
```

12 Token-Based Authentication

REST Authentication

- DRF's browsable API works with session auth, however, REST APIs must be stateless
- REST APIs use *token-based* authentication
- JWT (JSON Web Token) package: `simplejwt`
- Installation:

```
pip3 install djangorestframework-simplejwt
```

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
}
```

Setting Up `simplejwt`

- Create login view:

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [
    path('api/token', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
]
```

- Token is short-lived
  - 5 minutes by default, but can be changed to other durations
  - A **refresh** token can be used to extend its duration

REST Permissions

- A set of permissions can be applied to `APIViews`
  - E.g. `IsAuthenticated` requires the user to be logged in, e.g. via token
- Can specify a list of permissions for a view

```
from rest_framework.permissions import IsAuthenticated
class StoresOwned(ListAPIView):
    permission_classes = [IsAuthenticated]
    serializer_class = StoreSerializer
    def get_queryset(self):
        return Store.objects.filter(owner=self.request.user)
```

- Can create custom permissions
  - Subclass `BasePermission` and implement `has_permission` method

13 JavaScript

JavaScript (JS)

- A programming language that the browser understands
- Where JSON is derived from
- Used in both frontend and backend (Node.js) development
- High-level, runtime interpreted
- TypeScript: a strict superset of JavaScript
  - Strongly typed language

Declare a Variable

```
var x = 5;
let y = 4;
z = 3;
```

- `var`: creates a variable in global or *function* scope
- `let`: creates a variable in global or *block* scope
- Third way is not recommended, since it is hard to know if we are declaring or modifying the variable
- Variables can be reassigned to different types
- Use `const` to create constants, e.g. `const pi = 3.14`

Scope

- JavaScript has 3 types of scopes
  1. Global scope
     - Variables outside of any function
     - Variables can be accessed from anywhere in the program
  2. Function scope
     - Variables defined anywhere inside the function are local to that function
     - Cannot be used outside the function
  3. Block scope
     - Variable is only accessible inside the block it is declared in (e.g. if-statement blocks, loops)
- `var` and `let` are identical when used in global scope
- Global variables are discouraged
  - Convention: code should only be run inside functions

Data Types

- Number: integer or floating point
- String: same as Python
- Boolean: `true` or `false`, same as Java

- Function: can be created anywhere (locally or globally)
- To see what the data type of a value is, use `typeof`

Function

- Syntax

```
        function foobar(param1, param2, param3) {
            // ...
            return 0;
        }
```

- Can be declared anonymously and assigned to a variable

```
        var fun = function(a, b, c) {
            // ...
        }
```

- Can accept *any* number of arguments without error
  - Missing arguments are given the value `undefined`
- Without a return statement, function returns `undefined`

Object

- Syntax: similar to JSON, but key does not need to be a string
- `null`: denotes "no object", `typeof(null)` is object
- `undefined`: denotes "lack of value", `typeof(undefined)` is undefined
- Attributes (called properties in JS) can be modified in two ways:
  1. `person.firstName = "Joe";`
  2. `person['lastName'] = "Jordan";`

Array

- Syntax is same as Python list
- Arrays are objects
- Objects and arrays are mutable
  - Other data types are immutable

Method

- When object has function as a property, the functio becomes a method
- Method can access instance variable via `this` keyword
- E.g.

```
        var fruits = ["Banana", "Orange", "Apple"];
        fruits.clear = function() {
            this.length = 0;
        };

        fruits.clear();
```

Class

- Class is a special function that creates an object
- Requires a *constructor* method
- Supports *inheritance*
- E.g.

```
class Car {
    constructor(name, year) {
        this.name = name;
        this.year = year;
    }
    age() {
        let date = new Date();
        return date.getFullYear() - this.year;
    }
}
```

Condition

- Typically used in if statements
- == vs. ===
  - == performs implicit typecasting to satisfy the comparison
  - === does not perform typecasting
  - Avoid ==

Loops

- C-like loop

```
for (var i = 0; i < 10; i++) {
    ...
}
```

- While loop

```
var cars [];
while (cars.length > 0) {
    ...
}
```

- For ... of loop

```
var cars = []
for (var car of cars) {
    ...
}
```

- For ... in loop
  - Loops through *properties*

  - Similar to looping through keys of a dictionary
- `Array.forEach` method
  - Takes a function as argument

Switch

- Same a Java/C
- Accepts any mixture of data types for cases

```
        switch (new Date().getDay()) {
            case 1:
                ...
                break;
            case 2:
                ...
                break;
            ...
            default:
                ...
        }
```

14 Document Object Model (DOM)

JavaScript can be placed into HTML in three ways:

1. Inline JS

```
<script>console.log('hello')</script>
```

2. JS file

```
<script src="hello.js"></script>
```

3. As an event attribute

```
<form onsubmit="return validate(this)">...</form>
```

Document Object Model

- Browser creates the **DOM tree** of the page
- Each element is a **DOM node**
- `<html>` is the root node
- Each element can have zero or more child nodes
- Scripts access DOM elements through `document`
- `document` is a global object containing various methods

Accessing Elements

- Basic element getters

```
document.getElementById("st-2");
document.getElementsByClassName("ne-share-buttons");
document.getElementsByTagName("ul");
document.documentElement; // the root element <html>
document.body; // the body element <body>
```

- Query selector: uses CSS selector to specify elements

```
document.querySelector("#submit-btn");
document.querySelectorAll(".col-md-12");
```

DOM Object

- Each DOM node has properties to access related nodes:
  - `parentNode`
  - `firstChild`
  - `lastChild`
  - `childNodes`
  - `nextSibling`

  - `previousSibling`
- Can be combined with the element getters or query selectors

Manipulating Elements

- Element properties can be changed
  - E.g. `style`, `getAttribute(name)`, `setAttribute(name, value)`
- `innerHTML`: accepts HTML tags, typically preferred over `innerText`
  - E.g.

```
let body = document.body;
body.innerHTML = "<h3>Hello</h3>";
h3 = document.getElementsByTagName("h3");
h3.style.color = "green";
h3.setAttribute("class", "title");
```

Event

- JS supports event-driven paradigm
- Document events
  - Occurs to the entire page
  - E.g. onload, onkeydown, onkeyup
  - Convention: script should only be run after the onload event (to ensure all contents have been loaded)
- Element events
  - Occurs to a specific element, typically a specific type of element
  - E.g. onclick, onmouseover, ondrag, oncopy, onfocus, onselect, onsubmit

Two ways to add an event listener function:

1. Set the event property of a DOM element to a function

```
h1 = document.getElementsById("page-title");
h1.onclick = function() {
    this.innerHTML = "clicked";
};
```

2. Set the event attribute of an HTML element to a function

```
<script>
    function h3click(h3) {
        h3.style.color = "blue";
    }
</script>
<h3 onclick="h3click(this)">Hello</h3>
```

15 Asynchronous Requests

Requests

- Upon entering URL or submitting a form, one main request is made to the server
- Response is rendered
  - Additional requests are made to fetch static data, e.g. JS files, CSS files, images, fonts, etc.
- Server-side rendering
  - A full reload is needed for every URL request
  - High load time
  - Django's full stack framework does this

Asynchronous Request

- Ajax: Asynchronous JavaScript and XML
- Browser sends background request
  - Main thread is not blocked
  - Webpage still interactive
- Response handled by series of events and callbacks
  - Allows for further changes to the document
- Basis for single page application (e.g. React)

Sending Ajax Request

1. Instantiate a new Ajax request object

```
        let req = new XMLHttpRequest();
```

2. Define a handler for onreadystatechange

```
        req.onreadystatechange = function() {
            ...
        };
```

3. Set method and endpoint and send request

```
        req.open("GET", "http://localhost:8000/accounts/update/");
        req.send();
```

4. The event handler will trigger when response is received

- Very verbose, rarely used

16 jQuery

jQuery

- One of the most popular JS libraries
- Simplifies HTML DOM tree traversal and manipulation
- Helps with event handling and Ajax requests
- Could be replaced by React

jQuery Basics

- Syntax: everything is done through the $ function, based on *query selectors*

```
$("p").hide();
$(document).ready(function() {...});
```

- A wrapper around plain JS
- jQuery objects have different methods/properties

```
document.querySelector("#title").innerHTML = "<h1>Hello</h1>"; // plain JS
$("#title").html("<h1>Hello</h1>"); // jQuery
```

  - Designed to support *chaining*

Common jQuery Methods

- `val([value])`: get or set input value
- `attr(k [, value])`: get or set attribute with name k
- `css(p [, value])`: get or set CSS property with name p
- `html([value])`: get or set arbitrary HTML
- `click(function)`: register onclick event
- `parent(), children()`: get parent or children
- `next(), prev()`: get next or previous sibling
- `addClass(), removeClass()`: add or remove class(es)

Ajax With jQuery

- Use method `$.ajax(url [, settings])`
- Can speciyfy URL, method, etc.
- Accepts handler for success or error

```
$.ajax("/user/", {
    method : 'PATCH',
    data : {
        username : $('#username-input').val(),
    },
    headers : {
        'X-CSRFToken’ :
        $('input[name=csrfmiddlewaretoken]').val(),
    },
    success : function() {
        $('.show-modal').hide();
    },
    error : function(xhr) {
        if (xhr.status === 400) {
            var resp = xhr.responseJSON;
            if (resp['username']) {
                var message = resp['username'][0];
                $error.html(message).show();
            }
        }
    }
});
```

17 Advanced JavaScript

Built-in Functions/Methods

- `parseInt(x, [, base])`: attempts to convert string to integer, returns NaN on failure
- `isNaN(x)` checks if `x` is NaN
  - `NaN === NaN` is false
- `parseFloat(x)`: attempts to convert string to float, returns NaN on failure
- `String.padStart(n, c)`: pad `n` characters with character `c`
- `setTimeout(code, time)`: execute `code` after `time` milliseconds
- `setInterval(code, time)`: execute `code` every `time` milliseconds
- `String.trim()`: remove leading/trailing space
- `escape(x)`: convert to URL encoding
- `unescape(x)`: convert from URL encoding

Sessions

- Session-based authentication: browser already stores/sends *cookie* header
- Token-based authentication: we are responsible for storing/using the token, using `localStorage` global variable

  ```
        localStorage.setItem('access_token', access_token);
        localStorage.getItem('access_token');
  ```

- Set *authorization* header with the token value
  - E.g. in jQuery Ajax request settings:

    ```
            beforeSend: function (xhr) {
                xhr.setRequestHeader("Authorization", "Bearer " + access_token);
            }
    ```

Closures

- Functions can be defined inside a function and be returned
- **Closures**: nesting of functions where inner function has access to local variables in the outer functions
- Inner function **captures** local variable(s) from outer function
  - Captured variables can be referenced by inner function, where each invocation of outer function creates new copies of outer variables
- Can capture function arguments as well

For Loop and Closures

- In a for loop, `var` and `let` behaves differently
  - `var` declares a variable once and updates its value

  - `let` redeclares the variable multiple times with different values

- Since `var` only creates one variable, all closures created in the invocation of function references same variable
  - Results in aliasing among different closures
- E.g.

```
function outer() {
    let a = [];
    for (var i = 1; i <= 5; i++) {
        a.push(function() {
            return i;
        });
    }
    return a;
}
```

- Solution 1: force a copy by using immediately invoked function expression

```
for (var i = 1; i <= 5; i++) {
    a.push((function(i) {
        return function() { return i; };
    })(i));
}
```

- Solution 2: use `let` to declare loop variable, which creates one variable per iteration

```
for (let i = 1; i <= 5; i++) {
    a.push(function() {
        return i;
    });
}
```

Arrow Function

- Similar to lambda function in Python
- Allows for a code block on the left side of the arrow
- Syntax: `(args) => expression or body`
- The following are equivalent:

```
function regular(a, b) { return a + b; }
const arrow = (a, b) => { return a + b; };
const concise = (a, b) => a + b;
```

- Arrow function does *not* have their own `this` value
  - Do not use as event listeners or object methods
  - But arrow functions can capture `this` in a closure

Functional Programming

- Arrow function is used often in functional programming paradigm
- `forEach`: apply function to each element in array

```
names.forEach((item, index) => {
   console.log(item, index);
});
```

- `map`: for each element, modify it in some way and return new array

```
let upper = names.map(item => item.toUpperCase());
```

- `filter`: returns a new array, keeping elements that satisfies the condition

```
let jays = names.filter(item => item.name === "Jay");
```

- `reduce`: returns an aggregate value after processing the array, the accumulator takes an initial value

```
let longest = names.reduce((acc, cur) => Math.max(cur.length, acc),
    Number.NEGATIVE_INFINITY);
```

Destructuring

- Can unpack values from arrays or objects into local variable
- To destructure an object, the variable names has to be same as property names

```
const hero = { alias: "Batman", name: "Bruce Wayne" };
const {alias, name} = hero;
```

- Can also partially deconstruct and place the rest into a subobject

```
const {alias, ...rest} = hero;
```

- To destructure an array, can use any variable names

18 Event Loop and Promises

Event Loop

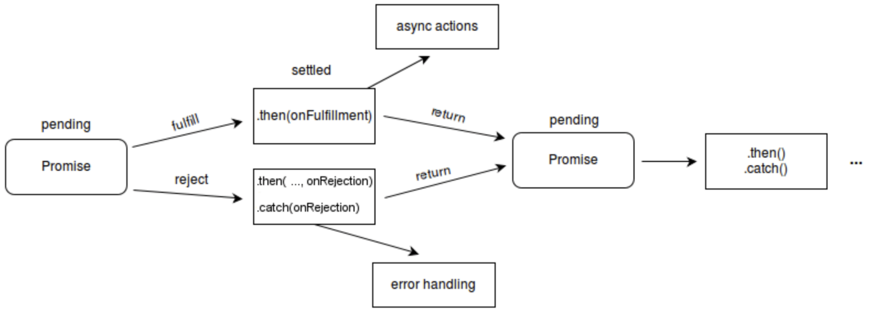
- JavaScript code is run in a single thread
  - Scripts are executed at load time, the rest are all *events*
- **Event loop** provides the illusion of multiple threads
- Events (e.g. ready, click, ajax, setTimeout) are pushed to the **event queue**
- Event loop constantly checks for new events and execute their callback (happens synchronously)

Promises

- Callbacks can make code hard to understand
- **Promise**

```
let test = new Promise(function(resolve, reject) {
    resolve("resolved");
});
test.then(msg => console.log(msg));
```

  - An alternative to using callbacks
  - Code inside the promise is executed immediately
  - Calling **resolve** or **reject** pushes events to the event queue (asynchronous), which can later be handled by the methods **then** or **catch**



Fetch API

- Returns a Promise object

```
let request = fetch('/account/login/', {
    method: 'POST',
    data: {username: "jack", password: "123"},
});
request.then(response => response.text()).then(text => console.log(text));
```

- Callback is specified in the `then` method instead of Ajax object
- Promise states:
  - *Pending*: the initial state

    - *Resolved*: happens when the resolve function is called
    - *Rejected*: happens when the reject function is called

Chaining Promises

- `then/catch` will get called even if promise is already settled
- Multiple callbaccks can be added by calling `then` several times
- Return value in `then` is wrapped in Promise

Async Function

- **Await** operator: waits for a Promise to be fulfilled before continuing code
- Error handling can be done through try/catch
- `await` can only be used inside `async` functions

19 React

Single Page Application (SPA)

- Executed in the browser's built-in JavaScript engine
- Only requires 1 hard URL reload
- Subsequent request/rendering can be done through Ajax and background
- Benefits
  - Seamless user experience: performing an action does not reset the page
  - Efficiency: only relevant parts of page are updated, not entire page
  - Improves load time: initial load (when nothing is there) takes less time
- Frontend frameworks: React, Angular, Vue

React JS

- A JS library for building interactive UI
- React takes charge of re-rendering when something change (no need to manually manipulate elements)
- **Virtual DOM**
  - Representation of UI kept in memory and synced with real DOM (handled by library ReactDOM)
  - When something changes, it compares new and old DOMs
    - Find what has been updated
    - Update only those elements in the browser's DOM

JSX

- A variation of JS that React uses
- Short for JavaScript XML, which merges HTML and JS into one language, e.g.

```
        const element = <h1>Hello</h1>;
```

- Browser does not understand JSX natively
- Babel JS: a JS compiler that can translate JSX code into pure JS code

React Components

- React **components**: functions that return a JSX element, or classes that extend `React.Component` and implement the *render* method
- Allows us to make elements reusable
- Void tags must always end with `/>`
- Component name must be capitalized
  - To distinguish from built-in HTML elements, which are always lowercase
- A JSX element must be wrapped in *one* enclosing tag
- React fragment: `<></>`

Components and Props

- Can put any JS expression inside curly braces in JSX
- **Props**
  - Read-only arguments passed into React components via a dictionary

```
function Text(props) {
    return <p>{props.value}</p>;
}
```

  - Can pass arguments like specifying HTML attributes in JSX

```
root.render(<Text value="Hello world" />);
```

Styles and Classes

- Styles and classes uses JavaScript names, not CSS/HTML names
- Styles must be placed inside a dictionary, i.e. `style={{fontSize: size}}`
- Do not need to add quotation marks around attribute values

Loop Generated Elements

- Elements created in a loop must have a unique `key` prop, which identifies which item has changed, is added, or is removed
- Otherwise, React will have to re-render the whole list whenever something changes
- Only affects the virtual DOM, no visible difference in the real DOM

Paired Tag

- Components can be wriite as paired tags
- Elements inside the tags are passed as the `children` props

```
function Box({children}) {
    return <div className="box">{children}</div>;
}
const mybox = (
    <Box>
        <List title="Cats" values={["Felix", "Oscar", "Fluffy"]} />
    </Box>
);
root.render(mybox);
```

Class Components

- To define a component, we can extend `React.Component` base class and implement the `render` method
- Can have *states*
  - In contrast, functional components are “stateless”
- Props are passed to constructor, can access through `this.props`

```
class Welcome extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}</h1>;
    }
}
```

Component State

- Class components have a built-in state whose default value is `null`
- Can override constructor to change the initial state
- State values can be accessed via `this.state` in the render method
- Whenever the state changes, the component re-renders

Updating State

- React states should never be mutated directly, except in the constructor
  - Otherwise re-rendering will not be triggered
- Use the `setState` method, which updates the state and triggers re-rendering

Events

- React has the same set of events as vanilla JS
- Syntax differences:
  - React events are written in camelCase
  - the actions must be a function, not just an expression
- Defining event handlers with component method doesn't work

Instance Binding

- A regular function binds to instance when called
- The object that calls the event handler is *not* the component
- Solutions
  1. Use the `bind` method, which enables early binding (not recommended since it is unrelated to application logic)

     ```
     constructor() {
         this.onClick = this.conClick.bind(this)
     }
     ```

  2. Use arrow function in class definition
     - Arrow function capture `this` from outer scope, which is the class body

     ```
     increment = () => {
         this.setState({counter: this.state.counter + 1});
     }
     ```

Event Handling

- `event.target` is the element that triggered the event

```
<input type="text" onChange={event => this.setState({message: event.target.value})} />
```

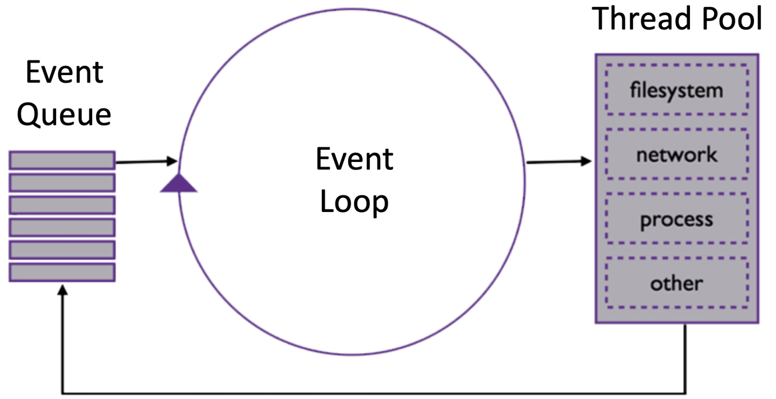
20 React Project

React Project

- JSX code are translated to JS every time page is loaded, which is very slow
- Alternative: **React Project**
  - **Frontend** server that returns appropriate files per request
  - A precompiled and bundled build for production
- Node.js: a runtime environment for running JS on server-side

Node.js Processing Model

- JS code still run in a single thread, but hidden threads exist
- I/O requests can be handled asynchronously without blocking main thread



Node Console

- Can be opened with the `node` command
- Allows executing inline JS code
- No `window` or `document` global object since we are not inside a browser
- Can execute scripts

```
        node <filename>
```

Installing Modules

- Node Package Manager (npm): similar to Python pip
- Install packages via `npm install <package_name>`
  - Packages are stored in the `node_modules` directory, similar to venv in Python
- Automatically generates and maintains a file named `package.json`, similar to requirement.txt in Python
- Node Package eXecute (npx)
  - Allows executing JS package

Creating React Project

- Create react project: `npx create-react-app <name>`
- Run development server (default port is 3000): `npm start`
- Make a production build: `npm build` (i.e. project contains the same code but more organized)
- Important files:
  - public/index.html
    * Contains base HTML code
    * DOM is rendered inside a div with id `root`
  - src/index.js
    * Invokes ReactDOM.createRoot
    * By default, renders `<App />`
  - src/App.js
    * Placeholder App component

Exports

- In JS, each file is a module
- By default, all definitions in a module are *not exported*, i.e. they cannot be imported into another module
- `export` keyword: allows variable/class/function to be exported

```
const var1 = 3, var2 = (x) => x + 1;
export { var1, var2 };
// or
export const var1 = 3, var2 = (x) => x + 1;
```

- `import` statement:

```
import { var1 } from './App';
```

Default Export

- Each module can have one default export (usually the component defined within the module)

```
export default App;
```

- Can import the default export

```
import App from './App';
```

- Importing default export *does not* require matching name, e.g.

```
import OldApp from './App';
```

File Structure

- Put almost everything in the `src` folder

    - If not used by any React component, then place in `public` folder

- Image, fonts, and other static files
    - Create a `src/assets` folder and place them there
    - Import them directly into JS module to use them

```
import logo from './assets/logo.svg';

// in render method
<img src={logo} />
```

- Do *not* import anything to the HTML
    - All static file imports, including JS and CSS, are handled automatically by the browser

Organizing Components

- All components should be placed in `src/components` folder
- Each component should be in its own folder
    - Name of folder should be same as the component
    - JavaScript file should be named `index.jsx`
    - CSS file should be in the same subfolder, usually named `style.css`
- Import local CSS file as the following:

```
import './style.css';
```

- Import other commponents as the following:

```
import Counter from './components/Counter';
```

- Components should be small, i.e. < 100 lines of code
    - Large components should be split into small, nested child components

21 Hooks

Hooks

- Introduced in React 16.8 (2019)
- Make functional components much more versatile
- Used to write clear and concise components
- **Hooks**: a set of functions that we can call inside a functional component
- E.g. `useState(initialState)`
  - Defines a single state variable within the component
  - Returns the variable and its update function
    - By convention, should be stored using *array destructuring*

      ```
      [variable, update] = useState(initialState);
      ```

  - Component re-rendered when `update` is called to change the variable

Using Hooks

- Only call hooks at the top level (which ensures deterministic call ordering)
- Only call hooks from React functions
- Benefits of using hooks
  - Supports multiple state variables
  - Easy to share state(s) with child components
  - Easier to use compared to class components

Lifecycle

- So far, we only run code when `render` is called (for both class and functional components)
- We don't want to run expensive operation on every re-render (e.g. sending ajax request only when component is first loaded)
- Lifecycle methods: executes when somthing happens to a component
  - `componentWillMMount()`: before loading a commponent
  - `componentDidMount()`: after loading a component
  - `componentDidUpdate()`: after updating a component (except initial load)
  - `componentWillUnmount()`: before unloading a component

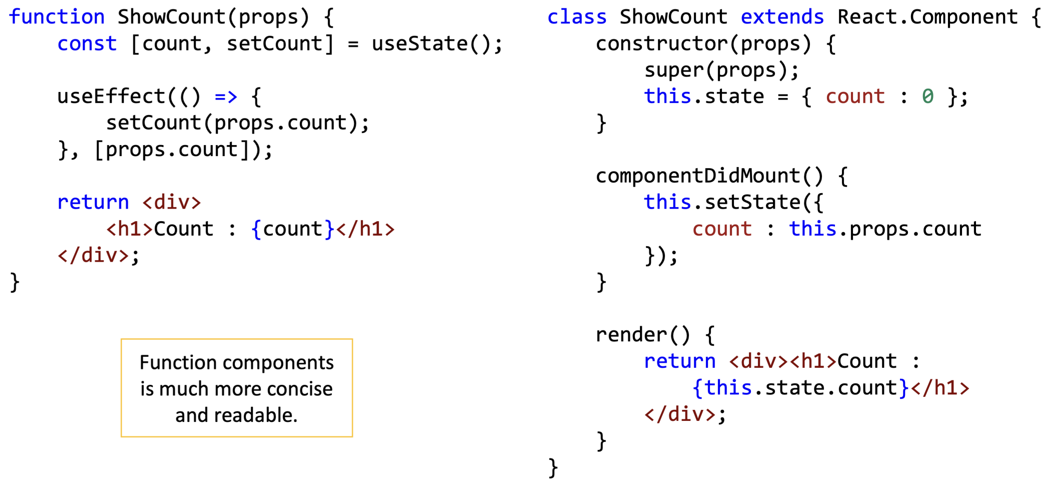
useEffect

- Takes two parameters, a function and an array of *dependencies*
  - If dependency is empty, callback only occurs on load
  - Otherwise, callback occurs whenever a dependency changes
- E.g.

```
    useEffect(() => {
        console.log("This is called when component mounts");
    }, []);
    useEffect(() => {
        console.log("prop s size or status has changed");
    }, [status, props.length]);
```

- If dependency is missing, effect would run at every re-render (typically not used this way)
- Dependency array should include *all* variables used in the effect
  - Otherwise it might use *stale* values at re-render (since React sometimes caches values for optimization)

Function vs. Class Component



```
function ShowCount(props) {
    const [count, setCount] = useState();

    useEffect(() => {
        setCount(props.count);
    }, [props.count]);

    return <div>
        <h1>Count : {count}</h1>
    </div>;
}
```

```
class ShowCount extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count : 0 };
    }

    componentDidMount() {
        this.setState({
            count : this.props.count
        });
    }

    render() {
        return <div><h1>Count :
            {this.state.count}</h1>
        </div>;
    }
}
```

Function components is much more concise and readable.

22 Global State

Global State

- **Prop drilling**: passing state(s) down to descendants components
  - Could be cumbersome
- A **global state** is accessible everywhere
- Do not use global state for everything, which would make code hard to understand and make components hard to reuse
- **Context**: React's solution to support global state
  - Create a state variable and its setter, and put them in a *context*
  - Everything inside the context is accessible within its *provider*
- Context enables handling API data easily
  - For some data (e.g. username), many components need to access them
- For each Django app, create a *context* in React
  - Then, write a function that sets up relevant values and their setters (namme of this function should start with "use")

Context

- Convention: create a `contexts` folder under `src`, and put all context files inside
- `createContext`: creates a context that can be later used

```
export const APIContext = createContext({
    players: [],
    setPlayers: () => {},
});
```

- Put default initial values for every variable that we will include in the context

Provider

- Creates an environment where the context is available

1. With `useState`, create the state(s) and their setters
2. Put a provider around the parent component and initialize it

```
function App() {
    const [players, setPlayers] = useState([]);

    return <APIContext.Provider value={{players, setPlayers}}>
        <Players />
    </APIContext.Provider>;
}
```

3. Any descendant components can access the context with `useContext`

```
const { players } = useContext(APIContext);
```

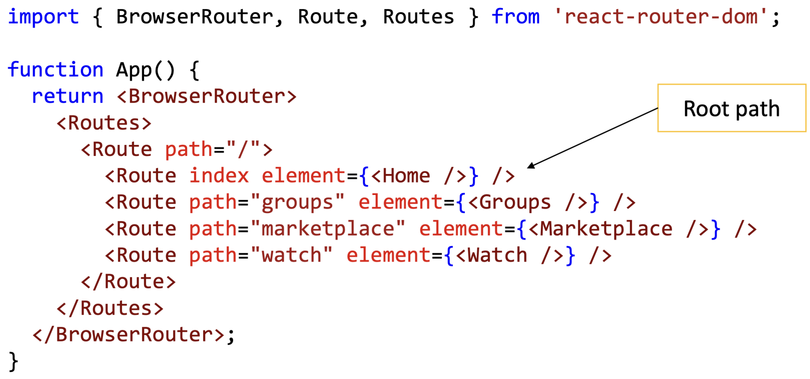
23 Multi-Page React App

Router

- We want each page to have its own URL, but no browser reload when switching between pages
- Installation: `npm install react-router-dom`
- Convention: craete a `pages` folder inside `src`, and put each page's component in a separate file/directory

Routes and Links

- We can set up the **routes** in App.js
  - Similar to setting up `urls.py` in Django



```
import { BrowserRouter, Route, Routes } from 'react-router-dom';

function App() {
  return <BrowserRouter>
    <Routes>
      <Route path="/">
        <Route index element={<Home />} />
        <Route path="groups" element={<Groups />} />
        <Route path="marketplace" element={<Marketplace />} />
        <Route path="watch" element={<Watch />} />
      </Route>
    </Routes>
  </BrowserRouter>;
}
```

Root path

Link

- Similar to `<a>`, but without a browser reload

```
<Link to="/watch">Watch</Link>
```

- URL arguments
  - Specified as part of the route definition, using : before parameter name

```
<Route path="groups/:groupID" element={<Groups />} />
```

  - Can be accessed via a hook

```
const { groupID } = useParams();
```

  - Same way to link to the page:

```
<Link to="/groups/42">Groups</Link>
```

Query Parameters

- Can be accessed via another hook

```
const [searchParams, _setSearchParams] = useSearchParams();
```

- To extract a specific key:

```
searchParams.get('name');
```

- Use query parameters in an URL:

```
<Link to="/groups/42?name=kia">Groups</Link>
```

Navigation

- Sometimes, we need a URL change via code (e.g. when response is 401, redirect to the login page)
- Vanilla Javascript: `window.location.replace("/login");` which causes a browser reload
- React Router

```
let navigate = useNavigate();
navigate("/login");
```

Outlet

- We need a *navbar* to navigate through pages
  - Bad idea to copy it to all the pages
- When we specify an element for root URL, only that element will be rendered and all child elements will be ignored
- In nested routes, React renders the first coponent that *partially matches* the URL and has an element
- It continues matching the remaining URL and returns the *matching child* commponents as `<Outlet />`
- Convention: root element is used to specify layout, child components are rendered within

```
// App.js
<Route path="/" element={<Layout />}>
    <Route index element={<Home />} />
    ...

// Layout.jsx
const layout = () => {
    return <>
        <header>
            <Link to="/watch">Watch</Link>
            <Link to="/groups/88/?name=joe">Groups</Link>
            <Link to="/marketplace">Marketplace</Link>
        </header>
        <Outlet />
    </>;
}
```

24 Web Deployent

Development vs. Production

Development	Production
Lightweight Database (e.g. SQLite)	Real Database (e.g. MySQL)
Runs on a development server	Runs a real webserver (e.g. Nginx)
Hosts on local machine	Hosts on public machine/cloud platform
Hosts on a local IP address	Hosts on a static IP address
No domain name	Has domain name
Upon error, shows stack trace	Upon error, returns 500 or 404
Security is not a concern	Needs to be secure and robust
Cannot handle high traffic	Can handle high traffic

IP Address

- Most IPv4 addresses havee almost been used up
- Transition to IPv6 has been slow
- Static IP address
  - Fixed IP address for the machine
  - Does not change over time, even if the machine restarts
- Dynamic IP address
  - IP address assigned by DHCP server
  - Can change the next time we connect to the Internet
- Production server typically uses static IP address(es)

Domain Name

- Websites need a domain name, otherwise, users must use the IP address directly
- Domain Name Registrar
  - Handles reservation of domain names
  - Assigns IP addresses to those domain names
  - E.g. GoDaddy, Namecheap, etc.
- We would need to buy a domain namme from a registrar

Web Hosting

- Dedicated hosting
  - Entire physical server to the website
  - Poor utilization, scalability, and availability
- Cloud hosting
  - Running application using combined computer resource
  - IaaS (Infrastructure-as-a-Service): user manages OS and above
  - PaaS (Platform-as-a-Service): user manages application and data
  - SaaS (Software-as-a-Service): user only manages data

25 Deploying Django Project

Deployment Options

- Can directly run development server on HTTP port 80

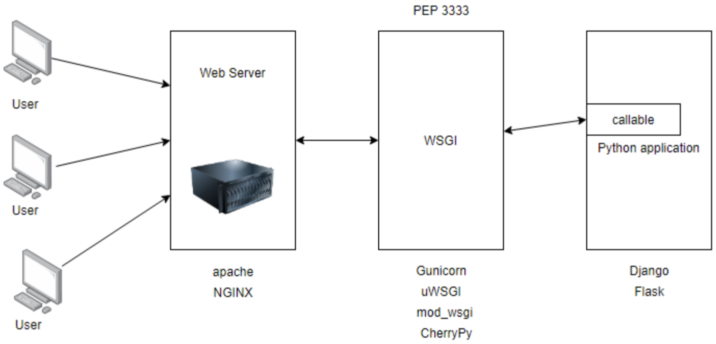
```
        sudo python3 manage.py runserver 0.0.0.0:80
```

  - `sudo` is required because using port 80 is root privilege
  - Not recommended

- **Gunicorn** (Green Unicorn)
  - Fast ad lightweight WSGI HTTP server
  - Installation: `sudo pip3 install gunicorn`
  - Start in terminal
    * Run `gunicorn --bind 0.0.0.0:80 <project>.wsgi` in the Django root folder
    * Does not serve static file
    * `<project>.wsgi` is the name of the new file

WSGI

- Requests are forwarded to the Python application via WSGI
- Works for any webserver and any Python backend framework

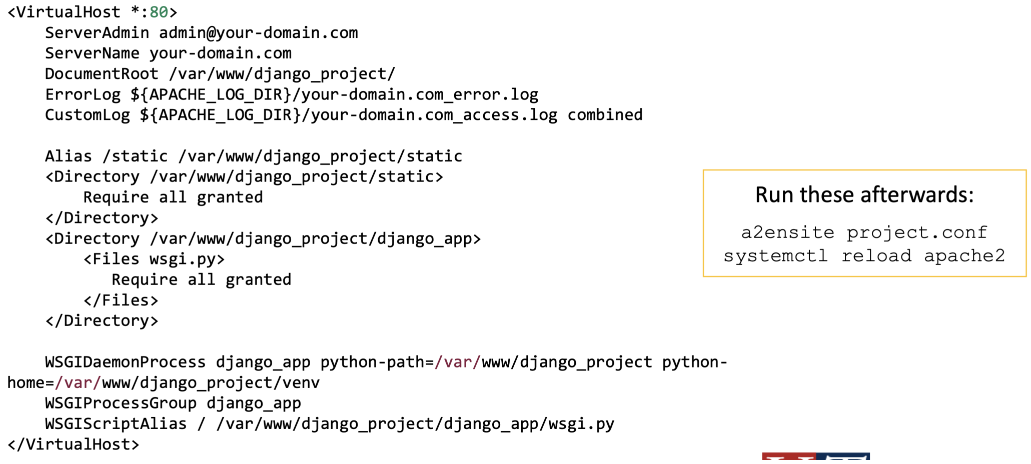


Apache Webserver

- Django's development server is not meant to handle high loads, withstand attacks, etc.
- Installation of Apache
  - Apache webserver: `sudo apt-get install apache2 apache2-dev apache2-utils`
  - mod-wsgi (for Apache): `sudo apt-get install libapache2-mod-wsgi-py3`
  - mod-wsgi (for Django): `sudo pip3 install mod-wsgi`
  - Copy the entire project folder into `/var/www/`
    * We may need to change owner/group to `www-data`: `sudo chown -R www-data:www-data /var/www/<project>`

Set Up Apache Conf File

- Create /etc/apache2/sites-available/<project>.conf



```
<VirtualHost *:80>
    ServerAdmin admin@your-domain.com
    ServerName your-domain.com
    DocumentRoot /var/www/django_project/
    ErrorLog ${APACHE_LOG_DIR}/your-domain.com_error.log
    CustomLog ${APACHE_LOG_DIR}/your-domain.com_access.log combined

    Alias /static /var/www/django_project/static
    <Directory /var/www/django_project/static>
        Require all granted
    </Directory>
    <Directory /var/www/django_project/django_app>
        <Files wsgi.py>
           Require all granted
        </Files>
    </Directory>

    WSGIDaemonProcess django_app python-path=/var/www/django_project python-
home=/var/www/django_project/venv
    WSGIProcessGroup django_app
    WSGIScriptAlias / /var/www/django_project/django_app/wsgi.py
</VirtualHost>
```

Run these afterwards:

```
a2ensite project.conf
systemctl reload apache2
```

Set Up Django

- Set DEBUG to False
- Use a secure secret key for SECRET_KEY
- Add the domain name for ALLOWED_HOSTS
- Use a real database (not SQLite) for DATABASE
- Do not push settings.py to repository (since database password is stored)

Production Settings

- Django loads settings from the environment variable DJANGO_SETTINGS_MODULE
  - Can create another file that imports from settings.py and override some options:

```
            export DJANGO_SETTINGS_MODULE=project.production_settings
```

- Use if DEBUG to separate debug/production settings
- .env file: load settings from an environment file
  - Use one for local and one for production
  - Load it on startup (python-dotenv package)

Static Files

- **Static files**: file/directory access granted to the webserver
- In Django project, they are *scattered* in many places, i.e.
  - app/static folders
  - Global static folder
  - Django contrib package, e.g. admin panel
- Django can *collect* all of the into the STATIC_ROOT folder
  - Required for security and performance reasons
  - Typically served by the webserver (i.e. should not go through URL dispatcher)

  - Command: `python3 manage.py collectstatic`

Advanced Setup

- Combine multiple webservers
  - Each has a dedicated task (e.g. serving dynamic content, static files, etc.)
  - Can exist on a different physical machine or in a CDN
- Gunicorn
  - Currently stops when we close the terminal
  - Can start using `nohup`, which will not shutdown when terminal closes
  - However, this does not restart if the machine reboots
- Can make it a **service**, which runs forever, restarts upon error, and runs on startup

Service

- To register a service, create a new file under the following directory:

```
        /etc/systemd/system/<service_name>.service
```

- Can manage service via the following commands:

```
        sudo service <name> restart
        sudo service <name> status
        sudo service <name> stop
```

- If we change the service file, we might need to reload it:

```
        sudo systemct1 daemon-reload
```

- A service binds to a socket, not `0.0.0.0:8000`
  - We want a real webserver to serve our application on port 80
  - However we cannot let gunicorn take port 80
- A real webserver forwards dynamic requests to gunicorn
  - Gunicorn services the backend project through a local socket

Deploy with Nginx

- Create config files in `/etc/nginx/sites-available/<project_name>`
- Make a symbolic link to that file, at `/etc/nginx/sites-enabled/`

26 Deploy React Project

Deploy React Project

1. Build React project: `npm run build`
2. Configure webservet to serve the appropriate files

Nginx

- Route all requests to the build folder