

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 1: Introduction to the Web

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# Course Instructor

- Kuei (Jack) Sun
- Contact Information
  - Use Piazza
    - <https://piazza.com/utoronto.ca/winter2023/csc309>
    - Sign up to the course to get access
  - By E-mail
    - Personal: [sunk@cs.toronto.edu](mailto:sunk@cs.toronto.edu)
    - Course Related: [csc309-2023-01@cs.toronto.edu](mailto:csc309-2023-01@cs.toronto.edu)
      - Request for lecture-related office hour and special consideration
  - By Calendly
    - <https://calendly.com/csc309-2023s>
    - Request for project/assignment related office hours and project grading



# Course Information

---

- Course Website on Quercus
  - <https://q.utoronto.ca/courses/293527>
  - Syllabus
  - Lecture slides/videos/exercises
  - Assignment handouts
  - Project handout
  - Grade posting
- Piazza
  - Course announcement, course discussion
  - Assignment discussion
    - Lab TAs will read and answer relevant posts periodically

# Don't Copy!

- Academic Integrity: Plagiarism and cheating
  - Very serious academic offences
    - All potential cases will be investigated fully
  - All assignments and exercises are to be completed individually
  - Do not submit code for grading that is not your own.
    - If you re-use any code, document the source
      - E.g., hash function from CSC209 A3 starter code, Fall 2019
      - Do not look at others' code, and do not share your code
      - Do not search for solutions on the web, or use AI-assisted tools, e.g., GitHub Copilot
    - Ask (and answer) questions on Piazza, but don't add details about your solution
- Exception: term project
  - You may use open-source packages, but they must be clearly referenced

# Join or Lead an RSG



- Meet weekly with up to 8 classmates online
- Review and discuss course material
- Prepare for tests and exams
- Get student advice from upper year mentors

Last year, over 3000 students joined a Recognized Study Group (RSG) where they met friends and reached their study goals.

Plan for success this term by joining your RSG today.

---

Join an RSG today: [uoft.me/recognizedstudygroups](http://uoft.me/recognizedstudygroups)

---

SIDNEY SMITH COMMONS



@sidneysmithcommons



# What we will cover

- How web works
  - Client-server model, Internet, HTTP, browsers
- Static web pages
  - HTML and CSS
- Dynamic website
  - Backend framework, i.e., Django (Python)
- Interactive pages
  - Frontend framework, i.e., React (JavaScript)
- System Administration
  - Deployment (website in production environment)

# Term Project

---

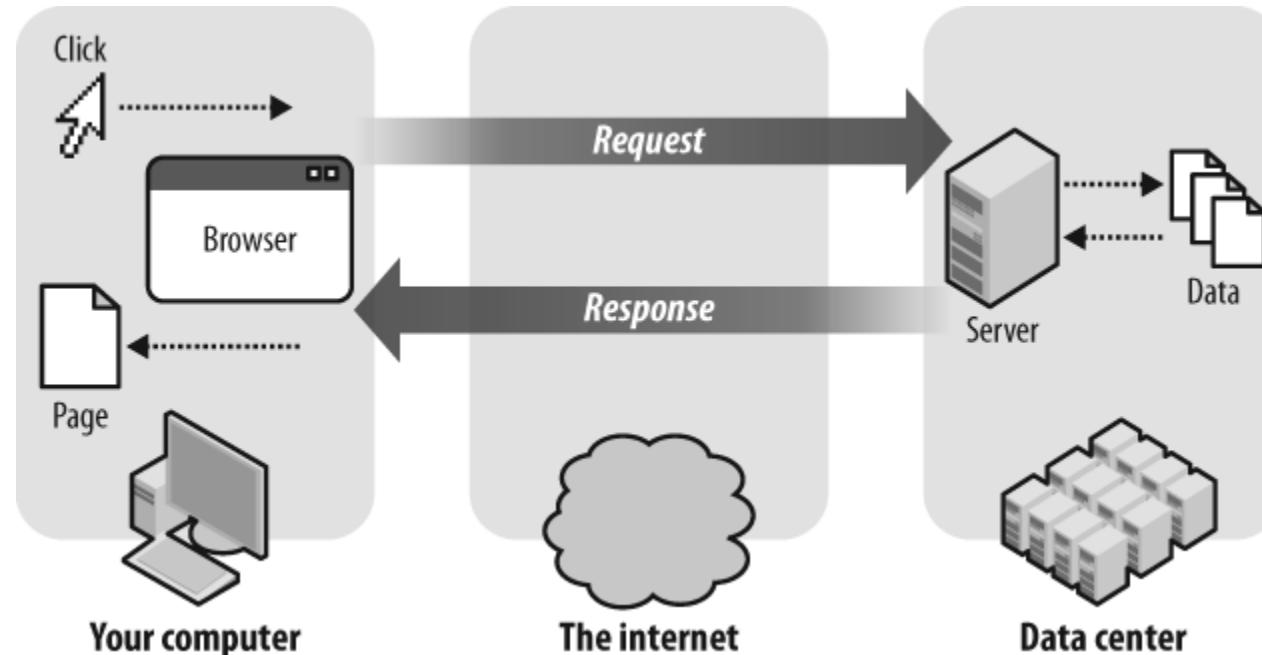
- Group project with up to 3 team members
- Restify
  - Simplified version of Airbnb
  - See project description for detail
- Split into 3 phases
  1. Static Design
  2. Backend API implementation with Django
  3. Frontend implementation with React
- For each milestone, book a grading session with a TA *before deadline*
- Form a group **ASAP** on MarkUs to start planning

# Disclaimer

- A lot of material is covered over 12 weeks
- Lectures and tests are focused on knowledge and concepts
  - With some simple coding exercises
- Project requires self-motivated learning
  - Lecture itself is not sufficient to teach every detail of web programming
  - Consult reference manuals
  - Search for answers online
  - Do the assignments for practice
  - Go to mentoring sessions

# End User Perspective

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



<https://medium.com/@lokeschhinni123>

# World Wide Web

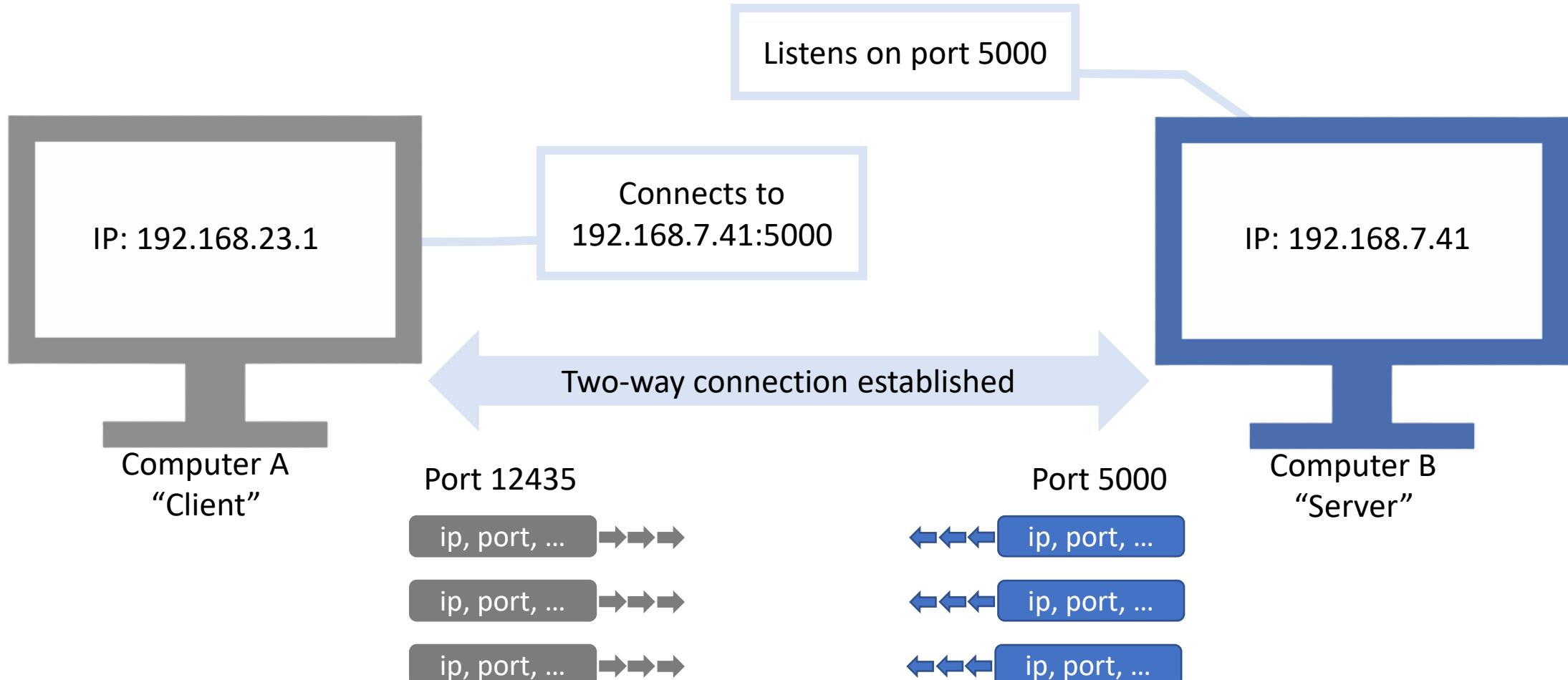
---

- Also just called “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
- HTTP
  - Protocol for delivery of contents from the Web.

# TCP/IP

- 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., port 80
  - Deals with unreliable nature of data transmission over network

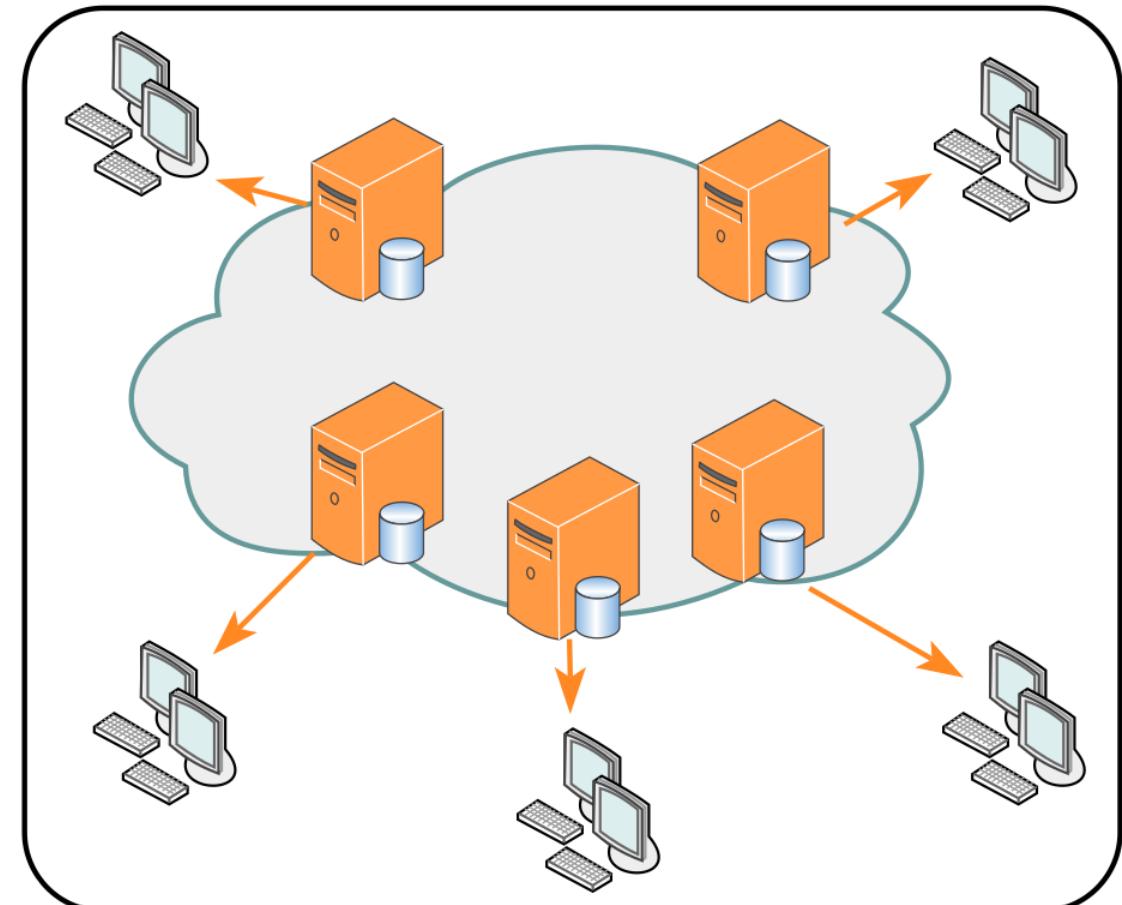
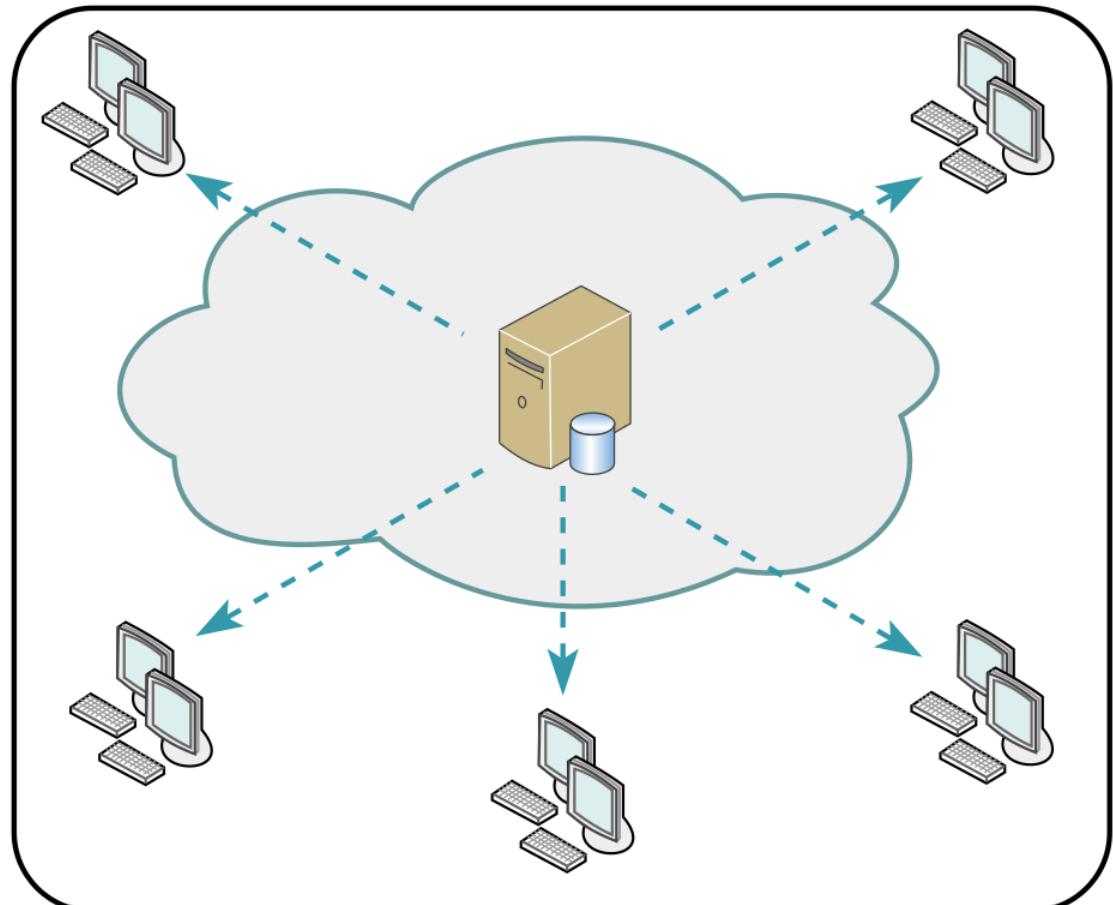
# How computers talk to each other?



# Domains

- IP addresses are hard to remember
- It is possible to move websites elsewhere
- Some websites may be hosted on multiple physical machines
  - Content delivery network (CDN)
- We want an easier way to remember addresses
  - Also want an easy way to remap websites to different IP addresses
- Domain Name
  - Maps an easy-to-remember name to IP address(es)
  - `www.google.com` → 142.251.41.78
  - Clients must *resolve* the domain before making a connection

# Content Delivery Network



# Domain Name System

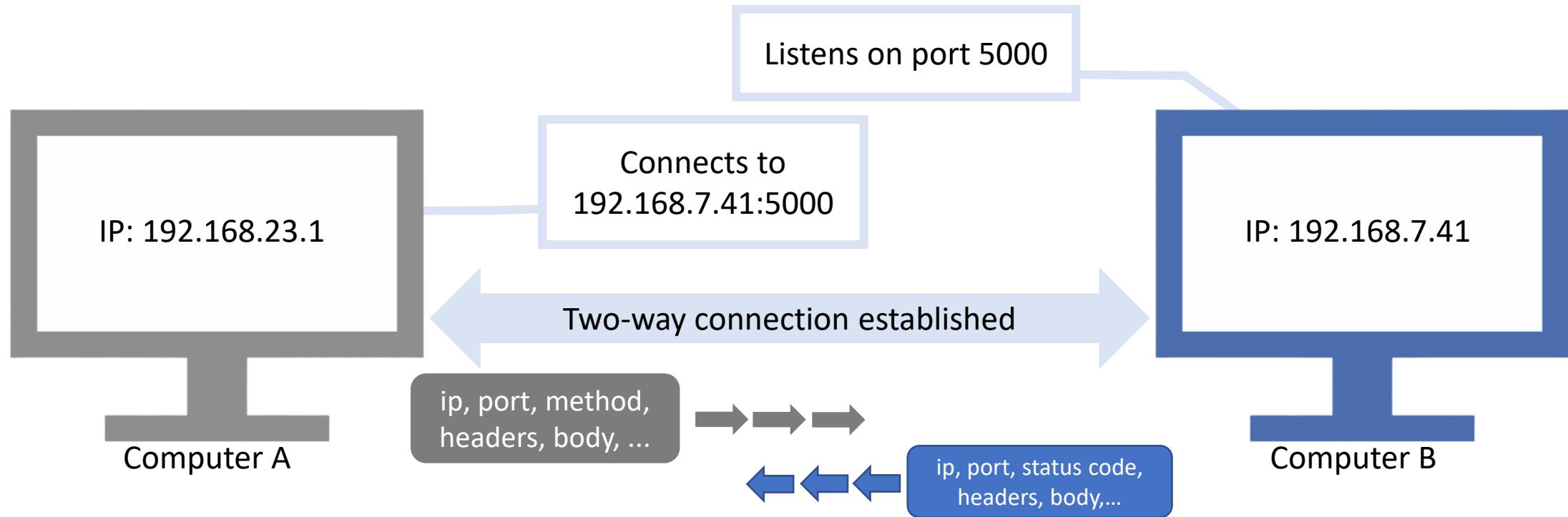
- A collection of mappings from domain names to their IP addresses
  - Analogy: Phone book
- Problem: how do we find the DNS server?
- Manually assigned by system administrator
  - E.g., 8.8.8.8 (Google's Public DNS)
- Automatically configured when computer connects to Internet
  - Computer sends a broadcast message to everyone on the local network
  - The DHCP server is responsible for assigning an IP address to the computer
    - It would also respond with the IP address of a DNS server

# Hypertext Transfer Protocol

- A protocol for distributing and accessing hypertext documents
  - Hypertext is text displayed on electronic devices, e.g., monitor
- 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
    - Protects against eavesdropping and tampering
  - Used by 81.3% of all public websites

# Stateless Protocol

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



# Statefulness

- A stateful service reacts *differently* to the same input
- Server must track the states all open connections
- Example: money transfer
  1. Enter account password
  2. Enter amount and recipient
  3. Confirm transfer
- Online banking service requires knowing that step 1 was successful
  - A stateful server remembers this on the server-side (the bank)
  - A stateless server gives the client a [cookie](#) to be passed back later
    - The client *reminds* the server of the previous step

# Statefulness

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

# HTTP Message

- Components of an HTTP Request
  - Method: describes what you want to do
  - Path: specifies which resource you want to access
  - Header: describes various settings and client environment
  - Body: additional data to be sent to server
- Components of an HTTP Response
  - Response code: describes the outcome of the request
  - Header: describes various settings and server environment
  - Body: data from the server (usually the hypertext of the web page)

# HTTP Message

## Requests

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;... )... Firefox/51.0
Accept: text/html,application/xhtml+xml,...,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

## Responses

```
HTTP/1.1 403 Forbidden
Server: Apache
Content-Type: text/html; charset=iso-8859-1
Date: Wed, 10 Aug 2016 09:23:25 GMT
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
Age: 3464
Date: Wed, 10 Aug 2016 09:46:25 GMT
X-Cache-Info: caching
Content-Length: 220

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
(more data)
```

# HTTP Methods

- POST
  - Create a new resource
- GET (most used)
  - Read information about a resource
- 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

- A string to reference a web resource and how to retrieve it
- Format of a [hyperlink](#) for navigating through hypertext documents
- Example:
  - `https://www.utoronto.ca/current-students`

The diagram illustrates the structure of a URL with three components: protocol to use, domain name, and resource name. The URL is shown as `https://www.utoronto.ca/current-students`. Three arrows point upwards from the text labels below to the corresponding parts of the URL above. The first arrow points to the prefix `https://`, labeled "protocol to use". The second arrow points to the domain name `www.utoronto.ca`, labeled "domain name". The third arrow points to the path `/current-students`, labeled "resource name".

protocol to use      domain name      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 content
- Popular modern browsers



**Safari**

Apple



**Firefox**

Mozilla



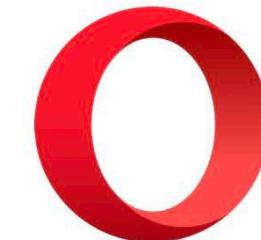
**Chrome**

Google



**Edge** new

Microsoft

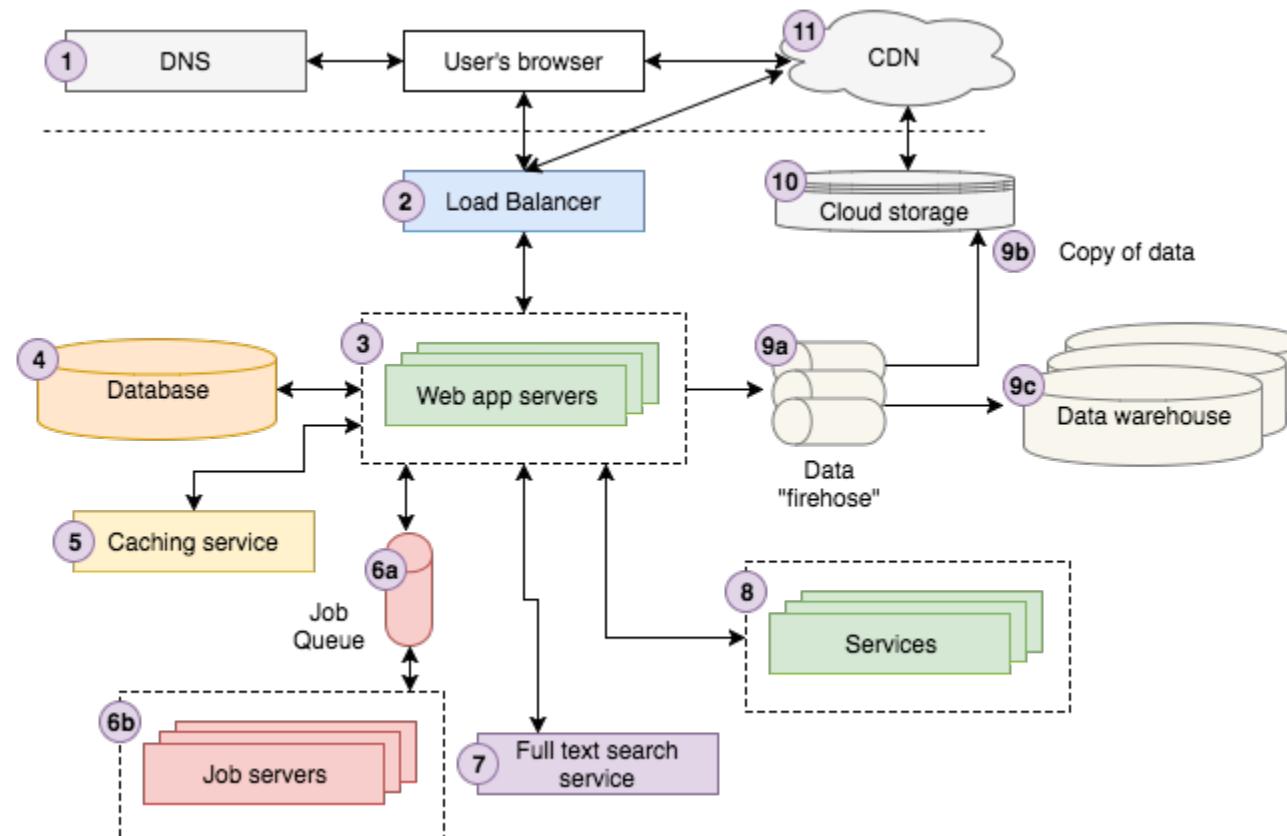


**Opera**

Opera Software

# Modern Web Architecture

- Contains many components, each used for different purposes



<https://medium.com/storyblocks-engineering/web-architecture-101-a3224e126947>

# Summary

---

- Web server **listens** on a specific port
  - Client(s) connect to IP address and port number
- DNS translates domain names to IP addresses
  - Users can refer to websites by domain name rather than IP address
- HTTP protocol
  - Stateless
  - Client sends a **request** and server replies with a **response**
- HTTP response body is usually in **HTML** format
  - Browsers understand this format and renders accordingly

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 2: Hypertext Markup Language

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# 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
  - Now used commonly for interchange of data over the Internet

# Hypertext Markup Language

- 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
    - E.g., image tag
    - ``



The diagram illustrates the structure of an HTML tag. It shows a blue `<img` tag with attributes `src="portrait.jpg"` and `alt="me"`. An arrow labeled "tag" points to the opening angle bracket of the tag. Another arrow labeled "attribute name" points to the word "src". A third arrow labeled "attribute value" points to the string "portrait.jpg".

```

```

↑ tag  
↑ attribute name  
attribute value

# HTML Syntax

- Two types of elements

## 1. Regular elements

- Element can have nested elements or text
- Most HTML tags are in this category
- Requires a closing tag
- E.g., section tag
  - `<section>` ← *opening tag*
  - `<p>first paragraph</p>`
  - `<p>second paragraph</p>`
  - `</section>` ← *closing tag*

The diagram illustrates the structure of a section element. It shows the opening tag `<section>` with a blue arrow pointing to it labeled "opening tag". Inside the section, there are two paragraphs: `<p>first paragraph</p>` and `<p>second paragraph</p>`. Below the section tag is the closing tag `</section>` with a blue arrow pointing to it labeled "closing tag". A large curly brace on the right side groups all these elements together and is labeled "section element".

```
<section>
  <p>first paragraph</p>
  <p>second paragraph</p>
</section>
```

## 2. Void elements

- Cannot have nested elements; does not require closing tag. E.g., `img`

# An Example HTML file

```
<!-- My first HTML file --> ← This is a comment  
<!DOCTYPE html> ← Not HTML. It declares the format of the document  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <title>My first HTML file</title>  
</head>  
<body>  
    <header>Hello World</header>  
    <main>  
        <h1>Welcome!</h1>  
        <p>I am Mishu.</p>  
          
    </main>  
    <footer>Bye</footer>  
</body>  
</html>
```

# Basic HTML tags

---

- <html>
  - The root element. All other tags must be declared within <html>
- <head>
  - The “invisible” part of an HTML document
  - Specifies various information about the document
    - E.g., <title> goes in here to specify the title of the document
- <body>
  - The “visible” part of an HTML document
- <header>, <main>, <footer>
  - Analogous to the header and footer of a printed document
  - main is the “meat” of the content

# Basic HTML tags

- Visit [www.w3schools.com/html/](http://www.w3schools.com/html/) for a full list
- Headings: <h1> to <h6>
  - Typically used to name a section of the document
  - Headings usually have larger font size
  - By default, h1 is largest, h6 is smallest
- Paragraph: <p>
  - A block of text
- Anchor: <a>
  - Defines a hyperlink
  - <a href="<https://www.google.com>">Go to Google</a>

# HTML Attributes

- Identifiers
  - id: specifies name of a unique element in the document
  - class: specifies name(s) of class(es) in which elements with same class share the same style and/or behavior
    - Used extensively by CSS to style elements
  - Hyperlink can include id to jump to that element
  - Example:
    - [https://en.wikipedia.org/wiki/HTML#HTML\\_5](https://en.wikipedia.org/wiki/HTML#HTML_5)
- Some tags have required attributes
  - E.g., <img> requires the src attribute to specify the URL of the image
  - You can “make your own” attributes, which are ignored by renderer

# Whitespaces in HTML

- Whitespaces in HTML files are **ignored**.

- Example:

```
<p>Hello
```

World

```
</p>
```

- when rendered, becomes:

Hello World

- forced line break

- Seldom used. Why?

# Preformatted Text

- Prefomatted text: <pre>
  - Asks browser to *not ignore* whitespaces inside the element
  - Often used to show source code
  - Problem: what if the text uses the greater than or less than character?
- HTML entities
  - Represent characters reserved by the HTML language
  - Starts with ampersand, followed by entity name or number
  - These 4 ASCII characters must be escaped
    - Greater than > becomes &gt;
    - Less than < becomes &lt;
    - Ampersand & becomes &amp;
    - double quote " becomes &quot;
  - Other entities: [https://www.w3schools.com/html/html\\_entities.asp](https://www.w3schools.com/html/html_entities.asp)

# Organizing Elements

- Division <div> tag
  - Block level element
    - Changes 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 <span> tag
  - Inline element
  - Allows styling of an inline element or text
  - `<p>Hello <span class="red">World</span>!</p>`
  - Alternatives include <em> (emphasis) and <strong>
    - Do not use: strike through <strike>, bold\* <b>, italic\* <i>

# 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> or <th> tag
  - <td>: table data
  - <th>: table header
- Number of cells in row should match expected number of columns
- colspan and rowspan attributes
  - Allows a cell to span multiple columns or rows
- More rigid than <div> and <span>

# 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 a ordinal value, e.g., Arabic number
- List item: <li>
  - An item inside the list. Must be a child of either <ul> or <ol>
- Description list: <dl>
  - A list of key value pairs
  - Child elements must alternate between term <dt> and description <dd>
  - Hint: very good for designing forms

# HTML Form

- Primary way to send user data to server
- <form> tag
- Input elements

```
<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
- When user presses submit, a request can be sent

↑  
boolean attribute

# HTML Form

- **action** attribute defines the URL of the HTTP request
- **method** attribute defines which HTTP method will be used
  - GET: usually used for queries and filters
    - E.g., Google search
    - Data are appended to the end of the URL
  - POST: data sent in body of HTTP request
    - Usually used for secure transfer, e.g., if password is in the form
- Form data consists of key-value pairs of input name and their values
- Example:

```
<form action="/submit" method="POST">  
    ...  
</form>
```

# HTML Validation

- Helps you identify issues with your HTML code
- Go to <https://validator.w3.org/>
- Detects many types of issues
  - Syntax error
    - E.g., missing a closing tag
  - Semantic error
    - E.g., missing required elements, such as <title>
  - Warnings
    - Not critical but would be nice to fix
    - <img> tag should always have the alt attribute for accessibility reasons

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 3: Cascading Style Sheet

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

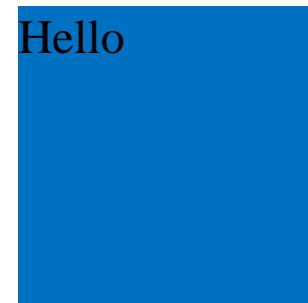
# Web Standards Model

- Separation of content (HTML) and appearance (CSS)
  - Modern websites are dynamic – content is frequently updated
  - Most websites strive for consistency in appearance
    - Improves look-and-feel and brand recognition
  - This allows content and appearance to be updated independent of each other
- Other benefits
  - Accessibility
    - Simplifies the work of screen reader
  - Device compatibility
    - Web page can be rendered nicely on difference devices
  - Search engine
    - Helps search engine with parsing your web pages; avoids misclassification

# Cascading Style Sheet

- Describes the presentation of a document written in markup languages
- CSS **property**
  - Defines the style or behaviour of an element
  - Full list of properties: <https://www.w3schools.com/cssref/index.php>
  - Syntax
    - *property name : property value(s);*
- Where to specify properties?
  1. Inline style
    - An attribute named “style”
    - Style only applies to this element

```
<div style="width:200px;height:200px;background-color:blue;">Hello</div>
```



# Where to specify properties?

## 2. CSS rule

- One or more properties that apply to one or more elements
- CSS selector
  - Determines which elements are targeted
- Syntax: *selector { properties ... }*



- Can write CSS rules inside `<style>` element, or in a `.css` file

# Where to write CSS rules?

- <style> tag
  - Needs to be inside the <head> element
  - Not recommended except during development or debugging
- css file
  - Needs to be “imported” in the html file
  - Can import multiple css files; provides basic modularity
  - <link> tag should go in the <head> element
    - `<link rel="stylesheet" href="style.css">`
- What does “cascading” mean?
  - Multiple rules can affect the same element.

# CSS Cascade

- Given the following paragraph element

```
<p class="danger">This is an important message.</p>
```

- and the following CSS rules

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

element selector  
↓  
p {  
 font-family: san-serif;  
 font-size: 1rem;  
 background-color: aliceblue;  
}  
↑  
some colors have names

- Output:

This is an important message.

- What happens if two rules override the same property?

# CSS Specificity

- A less specific rule is overridden by a more specific rule
  1. Order of appearance (later is better)
  2. Elements and pseudo-elements
    - Example: p, h1, ::before
  3. Classes, pseudo-classes, and attribute selectors
    - Example: .danger, :hover, [name]
  4. IDs
    - Example: #footer
  5. Inline style
  6. !important rule
    - Example: font-size: 1rem !important;
- See <https://wattenberger.com/blog/css-cascade> for more detail

# CSS Selector

- Element selector

- Syntax: tagname

- Example:

```
p {  
    color: blue;  
}
```

- Class selector

- Syntax: .classname

- Example:

```
.danger {  
    color: red;  
}
```

- ID selector

- Syntax: #idname

- Example:

```
#title {  
    color: black;  
}
```

- Pseudo-class selector

- Syntax: :pseudoclass

- Selects element in a special state

- Example

```
:hover {  
    color: green;  
}
```

# Advanced CSS Selector

- See w3schools for a list of all CSS selectors
  - [https://www.w3schools.com/cssref/css\\_selectors.php](https://www.w3schools.com/cssref/css_selectors.php)
- Attribute selector
  - Syntax: [attrname]
  - Select element with attribute name
- Example:
  - `[href] { color: rebeccapurple; }`
- Pseudo-element selector
  - Syntax: ::pseudoelement
  - Example:
    - `::first-letter { font-size: 200%; }`

# Combining CSS Selectors

- AND condition

- Syntax: join multiple selectors without space in between

`p.danger`

`a:hover`

`p.danger.big`

- Descendant condition

- Subsequent selector must be child or descendant or current selector

- Syntax: join multiple selectors with space in between

`.center p`  
`header nav a`

- OR condition

- Syntax: join multiple selectors with comma in between

`h1, h2, p { ... }`

`h1, .center { ... }`

- Other

- Immediate child condition

`form > input`

- Adjacent sibling condition

`div + p` (selects p after a div)

# Exercise 2

Do question 2 to 4.

# Font Properties

- color
  - Selects text color
- font-family
  - Select one or more fonts, in that order (in case former one is unavailable)
  - Web safe fonts
    - [https://www.w3schools.com/cssref/css\\_websafe\\_fonts.php](https://www.w3schools.com/cssref/css_websafe_fonts.php)
- font-style:
  - Either normal or italic
- font-weight:
  - `normal` or `bold` are most used options, others are possible

# More Font Properties

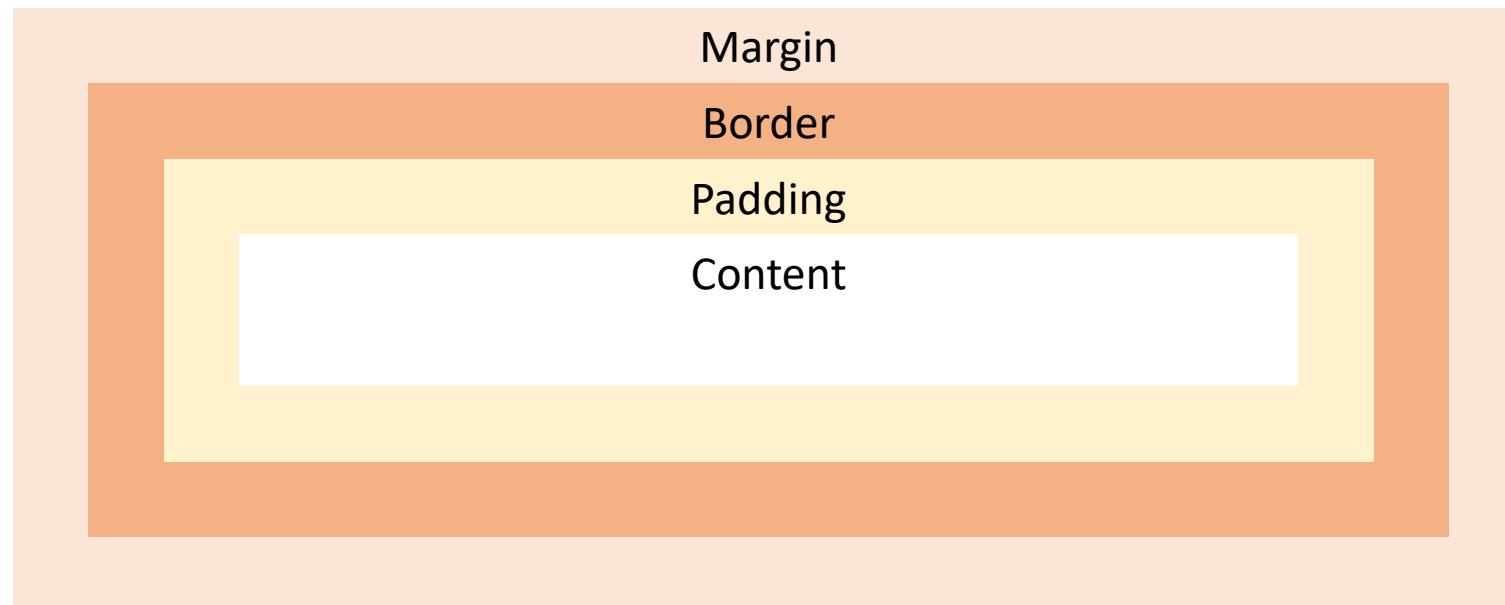
- **text-decoration**
  - none, underline, overline, or line-through
- **text-align**
  - left, right, center, justify (all lines of text are same width)
  - You likely want to use the hyphens property with justify
- **text-size**
  - Set size of text
- **Further reading**
  - font, line-height, text-shadow, text-transform, letter-spacing, word-spacing, etc.
  - <https://developer.mozilla.org/en-US/docs/Web/CSS/font>

# Units

- Used by any property that specifies size or length
- Absolute units
  - cm (centimeter), in (inch), px (pixels)
- Relative units
  - rem: root element's font-size (default is 16px)
  - em: parent element's font-size
  - vh, vw: current screen's (viewport) height or width
  - %: a percent relative to the size of the parent element
  - fr: fraction (of the available space)
- calc function: allows you to do math on different units  
`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



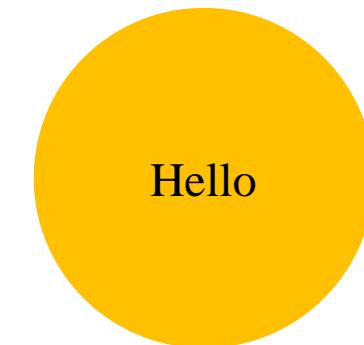
# Spacing Properties

- Many ways to specify margin/padding/border
  - Using margin, padding, or border-width
- Specify all edges
  - `border-width: 1px 2px 3px 5px;` (top-right-bottom-left)
  - `margin: 0;` (all edges)
  - `padding: 1rem 2px;` (top & bottom, left & right)
- Specify specific edges (top, right, bottom, left)
  - Append which edge you want to change (except border)
    - `border-top-width: 1rem;`
    - `margin-bottom: 12px;`
- Trick: 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
  - Trick: can create a circle when radius is exactly half the width and height

```
p {  
    width: 400px;  
    height: 400px;  
    line-height: 400px;  
    text-align: center;  
    border-radius: 200px;  
    background-color: gold;  
}
```



# Position Property

- Specifies how to position an element
- [https://www.w3schools.com/css/css\\_positioning.asp](https://www.w3schools.com/css/css_positioning.asp)
- static
  - Default behaviour. top, bottom, left, and right properties are ignored.
- relative
  - Relative to its static position (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 this element, as if it were removed
- **inline-block:**
  - Same as inline, but width and height properties are allowed.

# Responsive Design

- A web design approach
  - Pages adjust themselves to “look good” on all screen sizes
- Viewport setup
  - Required to ensure viewport adjusts to 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 of viewport width, e.g., 10vw (10% of width)
  - Use clamp function, e.g., `font-size: clamp(1rem, 10vw, 2rem);`

# 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 other elements to flow around it
  - Should *not* be used to design page layout in modern days

  Lorem ipsum dolor sit amet consectetur adipisicing elit.



  Inventore, voluptatem incident voluptas  
nobis placeat facilis commodi laboriosam  
similique id veritatis molestias,  
dignissimos praesentium, autem tenetur  
consequatur beatae itaque. Ipsa, iure.

  Morbi ornare tortor nisi, sed fermentum  
ipsum faucibus vitae. Aenean viverra mauris sit

# Flexbox

- Flexibly places items inside the parent element, a.k.a, a **container**
  - `display: flex;`
- Container size automatically adjusts to size of child elements
- **justify-content**
  - right, left (default), space-evenly, space-between, space-around
- **flex-wrap**
  - wrap (to the next line), nowrap (default)
- **flex-direction**
  - column (top down), column-reverse (bottom up), row, row-reverse

# Flexbox Properties

- align-items
  - Where to place child elements in a large container
  - center, flex-start, flex-end, etc
- align-content
  - Determines where to place each flex line (row or column) in a large container
  - center, flex-start, flex-end, stretch (default), space-between, etc.
- Trick: perfect centering

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

Better than using vertical-align!

# Flex item properties

- **Flex item**
  - The direct child elements of a flex container
- **flex-grow, flex-shrink**
  - How much a flex item will grow or shrink. Default is 0 (no growth/shrinkage)
  - Relative to other items
- **flex-basic**
  - Initial length of the flex item (width if row, height if column)
- **align-self**
  - Overrides the container's align-items property
- Exercise 2: <https://flexboxfroggy.com/>

# Grid Layout

- Grid supports two-dimensional layout, similar to table
  - However, use grid for layout; use table for tabular data
  - `.container { display: grid; }`
- `grid-template-columns`
  - For each column, specify a size value, e.g., 3 columns:  
`grid-template-columns: auto auto auto;`
- `grid-template-areas`
  - Uses named grid items to specify rows and columns  
`grid-template-areas: 'menu top top'  
'menu bot bot';`
- `gap, row-gap, column-gap`
  - Space between rows and/or columns

# Grid Item Properties

- Grid item
  - Direct child of a grid container
- grid-column-start, grid-column-end
  - Similar to colspan for <td>; allows you to span multiple grid columns

```
.item1 { grid-row-start: 1; grid-row-end: 3; }
```
- grid-row-start, grid-row-end
  - Similar to rowspan for <td>; allows you to span multiple grid rows
- grid-area
  - Specify which named area this grid item belongs to

```
.item1 { grid-area: menu; }
```
  - Can also be used to specify unnamed area to span both rows and columns

# Exercise 2

Do question 6.

# Media Query

- Checks the capability of the device 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
- Example:

```
@media screen and (min-width: 480px) {  
    #leftsidebar {width: 200px; float: left;}  
    #main {margin-left: 216px;}  
}
```

- Mostly likely used expressions:
  - min-width, max-width

# Browser Support

- Not all browsers support the same CSS properties
- <https://caniuse.com/>
  - Tells you if a feature is supported
- Example
  - Media Queries: Range Syntax
  - Allows you to write media query like this:

```
@media (100px <= width <= 1900px)
```
  - Easier to read compared to this:

```
@media (min-width: 100px) and (max-width: 1900px)
```
- Supported on Chrome 104 or higher, Firefox 63 or higher
- Currently not supported on Safari. (as of January 2023)

# CSS Framework

- Ready-to-use CSS libraries (may include JavaScript)
-  Bootstrap,  Tailwind CSS,  Bulma
- Provides basic and advanced interface components
  - E.g., Bootstrap modal
    - <https://getbootstrap.com/docs/4.1/components/modal/>
- 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 done for you

# CSS Tools

- Minifier

- CSS “Compresses” CSS files 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

- CSS is declarative, but it has some useful functions
- var()

- Use a custom defined variable in place of a property value

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

- url()
- Use for properties that references a file (usually image)

```
background-image: url("star.gif");
```

- max(), min()
- Selects the maximum or minimum of a set of values

```
width: max(20vw, 400px);
```

# CSS Animations

- [https://www.w3schools.com/css/css3\\_animations.asp](https://www.w3schools.com/css/css3_animations.asp)
- Allows animating HTML elements natively (without JS or Flash)

```
.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
- *Sass*, **{less}**, *stylus* Stylus
- Can declare variables, create loops, support inheritance
  - Helps with writing concise and maintainable CSS
  - Especially useful for making powerful animations
- Interpreter (or compiler) translates preprocessor script into CSS
  - Usually done automatically as soon as you update the script

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



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

# Before you go

- Next week
  - Install Django on your local computer
  - <https://docs.djangoproject.com/en/4.1/topics/install/>
- Assignment 1
  - Due Sunday January 29<sup>th</sup>
- Project
  - We will assign you random partners if you have not chosen yet
    - Deadline is Friday January 20<sup>th</sup>
  - Sign up on Calendly to book an interview session for project grading
    - <https://calendly.com/csc309-2023s>

# Miscellaneous Properties

- **list-style-type**
  - The marker for list (unordered or ordered)
  - <https://developer.mozilla.org/en-US/docs/Web/CSS/list-style-type>
- **z-index**
  - Determines which element is on top when multiple elements overlap
  - Larger the better
- **opacity**
  - Transparency of an element; from 0.0 (invisible) to 1.0 (fully visible)
- **transform**
  - Allows for rotation, skewing, scaling, etc, of an element

# CSC309H1S

## Programming on the Web

Winter 2023

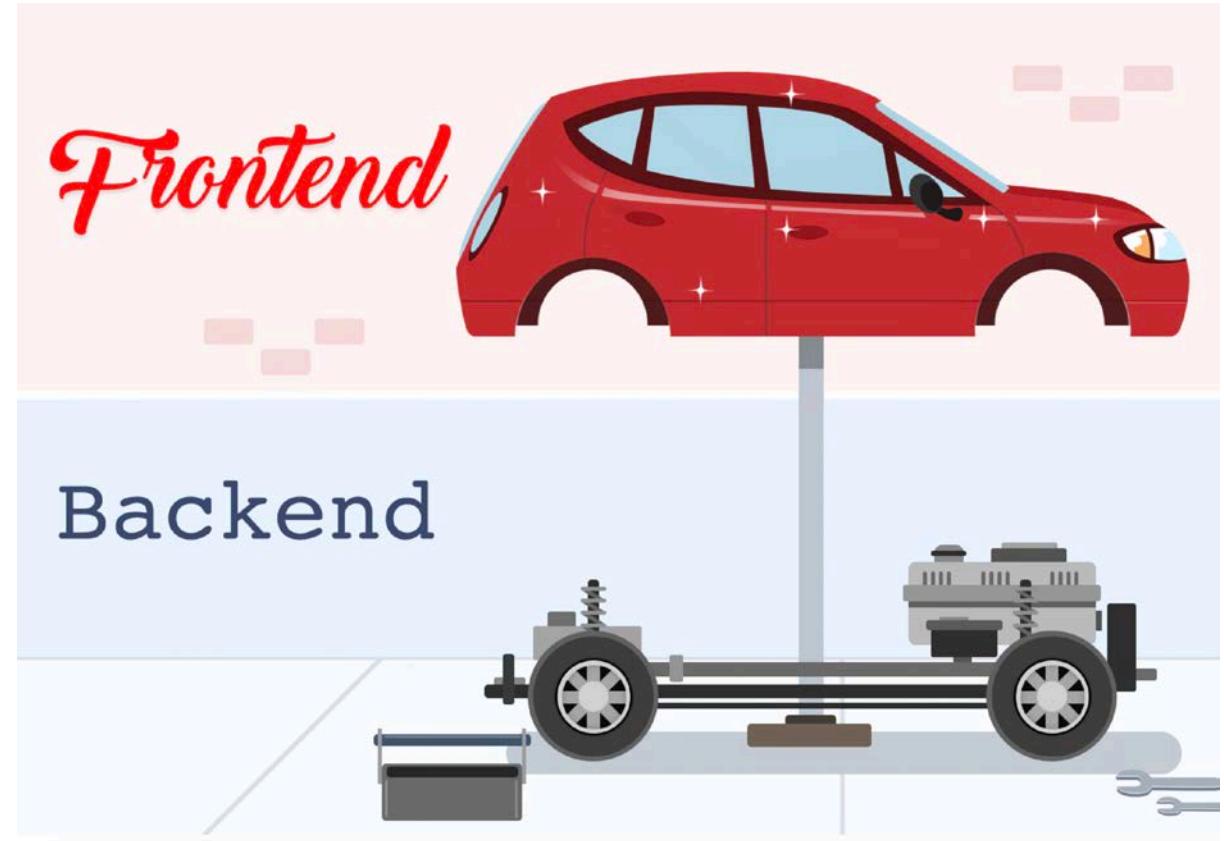
### Lecture 4: Introduction to Backend Development

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# Web Development

- Separation of concern
- 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\**
    - Server can do some frontend work
  - Data storage and business logic



Source: blog.back4app.com

# Abstraction in Web Architecture

## BACKEND vs FRONTEND

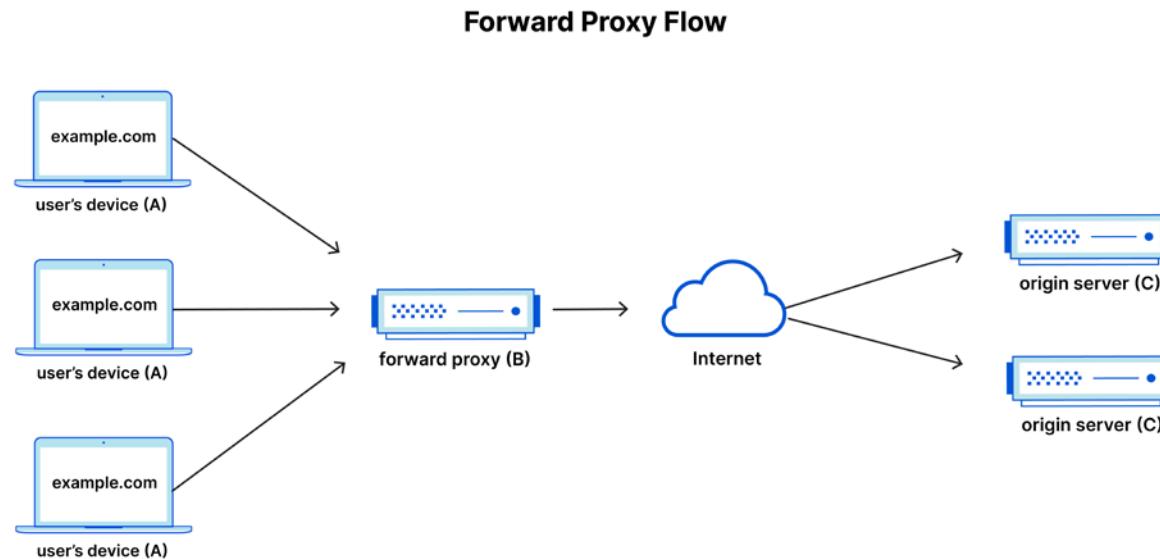


Source: [https://www.reddit.com/r/ProgrammerHumor/comments/m187c4/backend\\_vs\\_frontend/](https://www.reddit.com/r/ProgrammerHumor/comments/m187c4/backend_vs_frontend/)

# Web Server

- Listens on specific port(s) for HTTP/HTTPS requests
- Examples:  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
    - Note: this only works for HTTP requests (unlike VPN)

# Forward Proxy

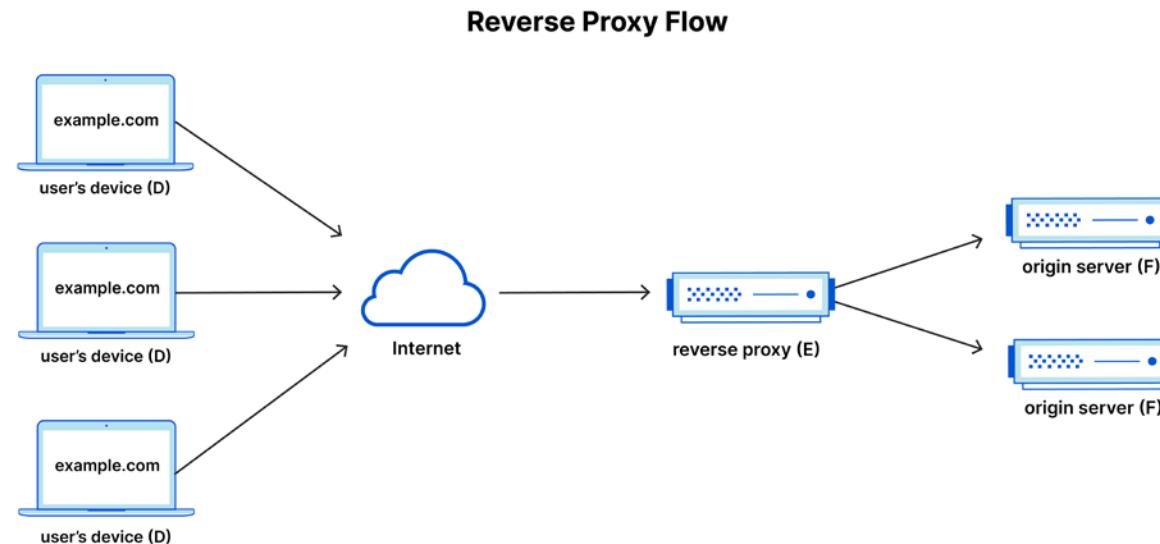


- **Usages**

- 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

<https://www.cloudflare.com/en-gb/learning/cdn/glossary/reverse-proxy/>

# Reverse Proxy

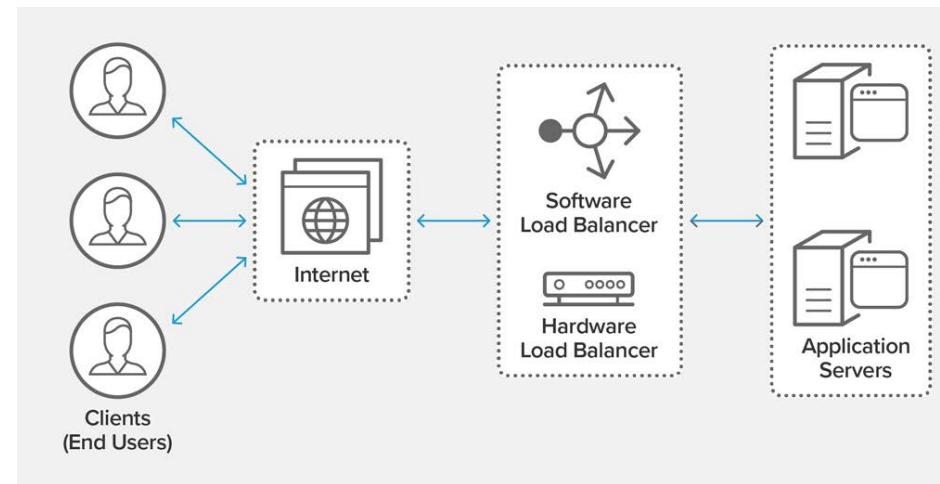


- Usages
  - Caches content for geographically distant web server
  - Acts as a front for security purposes, e.g., encryption, prevent DDoS attack
  - Provides **load balancing**

<https://www.cloudflare.com/en-gb/learning/cdn/glossary/reverse-proxy/>

# Load Balancer

- Popular websites can serve *millions* of concurrent requests!
- Load balancer distributes incoming requests among backend servers
- Ensures all servers have similar utilization
- Allows adding/removing servers based on current demand
  - Reduces energy consumption during times of low traffic

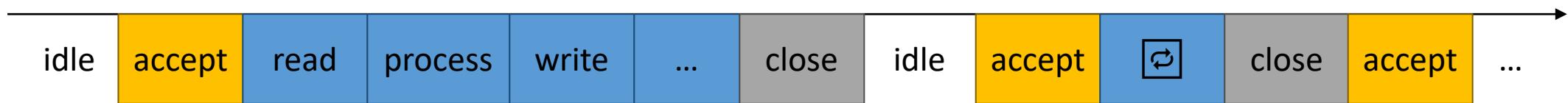


Source: <https://www.nginx.com/resources/glossary/load-balancing/>

# Web Server Architecture

- Single-threaded server

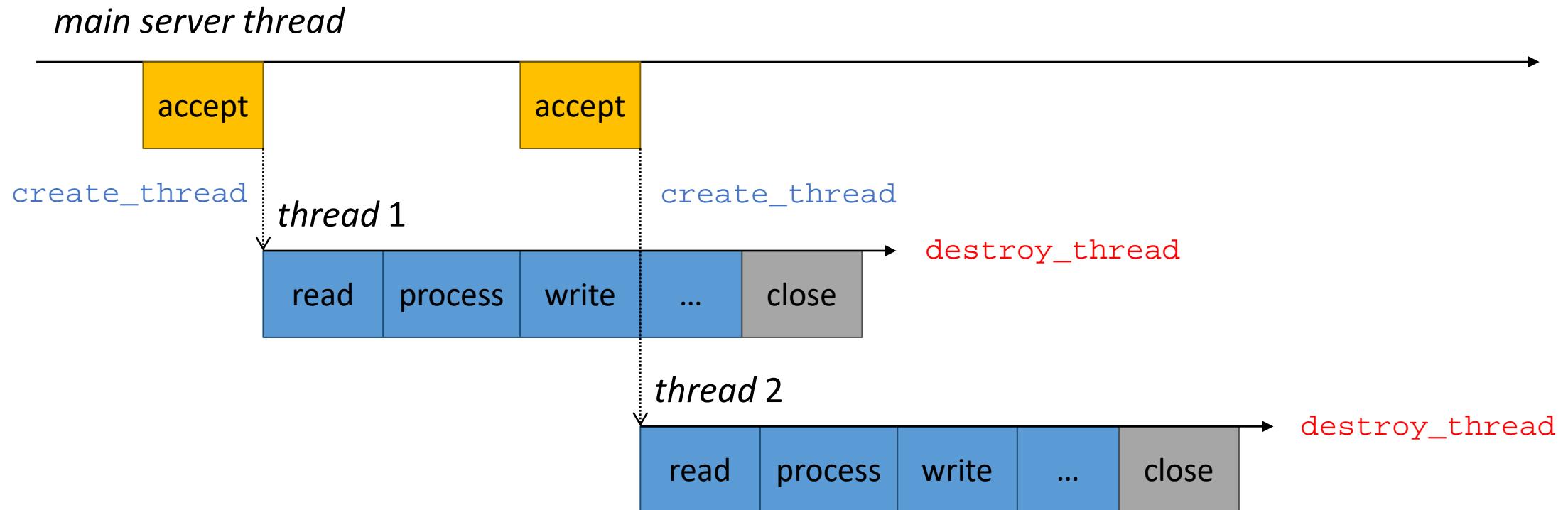
*main server thread*



- Cannot only handle one connection at a time!

# Web Server Architecture

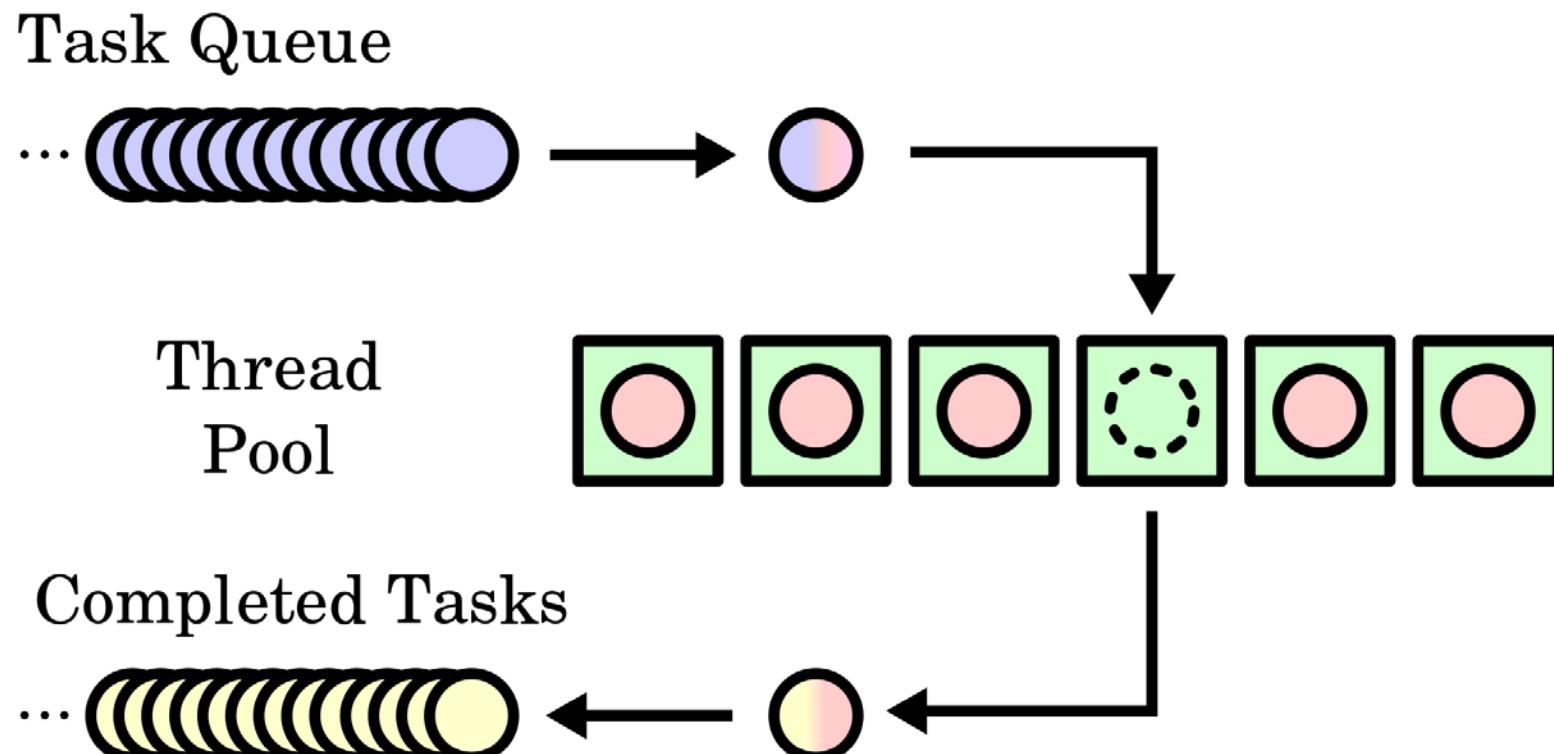
- Multi-threaded server



- Problem: creating threads is expensive and http requests are short-lived.

# Web Server Architecture

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

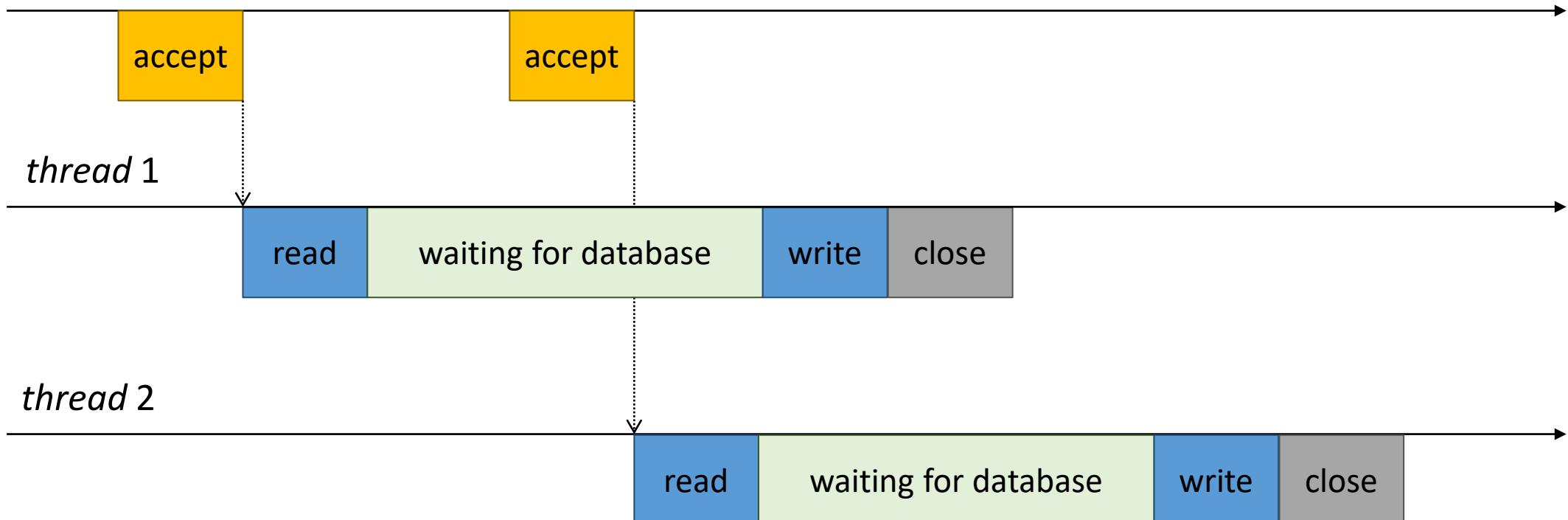


[https://en.wikipedia.org/wiki/Thread\\_pool](https://en.wikipedia.org/wiki/Thread_pool)

# Web Server Architecture

- Problem: threads are frequently blocked waiting for IO

*main server thread*



# Web Server Architecture

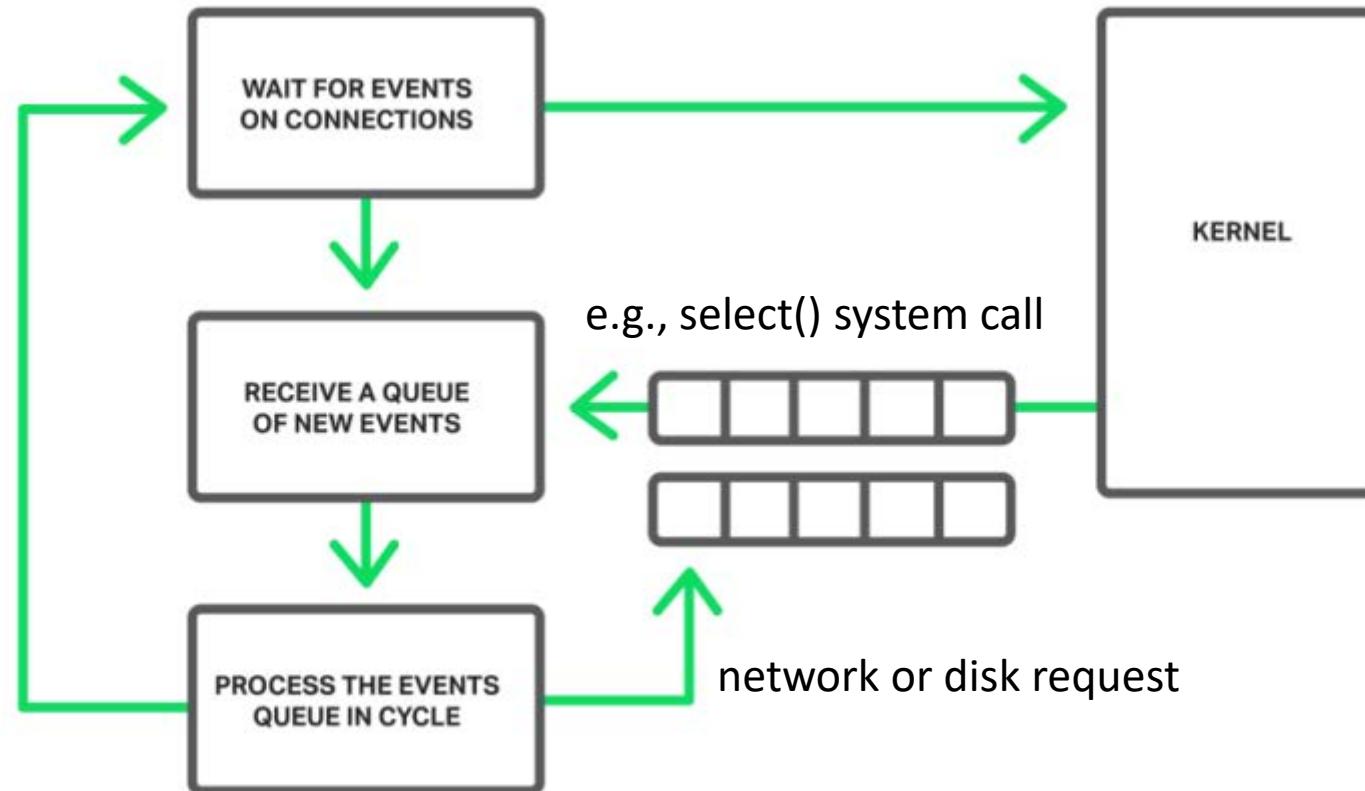
- Event-driven web server

*main server thread*



- 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

# Nginx Event Loop



<https://www.nginx.com/blog/thread-pools-boost-performance-9x/>

# Common Gateway Interface

- Allows web server to run an external program to process requests
  - In early days, to process forms, e.g., POST request
- 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
- Nowadays, there are many similar standards
  - WSGI (web server Gateway Interface): used by Python programs
  - Rack: used by Ruby programs
  - JSGI (JavaScript Gateway Interface): used by JavaScript programs

# Programming Languages

- Technically, any language can be used in the backend
  - It just needs a library that understands HTTP protocol
-  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
    - RAM access is  $10^6$  times faster than network access (ns vs ms)
    - Most performance optimization focuses on reducing network latency
      - E.g., using CDN and asynchronous requests

# Runtime Environment

- Hardware and software infrastructure for running code
- It is *not* the programming language itself
- Examples:
  - CPython
    - Interpreter that runs the Python programming language
  - Node.js
    - Runs on V8 JavaScript Engine
    - Favorite among web developers
      - Can write both frontend and backend with just one programming language
  - PHP interpreter
    - Runs the PHP programming language

# Backend Frameworks

- Libraries on the server-side that helps build a web application
- Avoids doing everything from scratch!
  - Listen on a port, process HTTP request, retrieve data from storage, process data, create HTTP responses, etc.
  - Don't reinvent the wheel
- PHP:  Laravel,  CodeIgniter
- Python:  Django,  Flask,  FastAPI
- JavaScript:  ExpressJS,  Spring
- Ruby:  Ruby on Rails



<https://www.geeksforgeeks.org/top-10-django-apps-and-why-companies-are-using-it/>

# Python Project

- Requires use of external packages
- Python's package manager: [pip](#)
  - Helps install and manage software packages
  - Automatically handles *dependencies*
    - Other packages (and their versions) that are required to use this package
- [pip3 install Django](#)
  - Command to install latest version of Django (4.1 as of Jan 2023)
- Multiple projects
  - How do I deal with different versions of Django? (or even Python itself)

# Virtual Environment

- Manages separate package installations for different projects
- An *isolated* environment with its own version of everything
  - Python interpreter, pip, and packages
  - Avoids dependency conflicts and version differences
    - Code for one version of Django may not run for a different version
- virtualenv
  - Provides lightweight encapsulation of Python dependencies
  - Note: does not encapsulate the operating system (heavyweight)
- Create a new virtual environment (with a specific Python version)
  - `virtualenv -p /usr/bin/python3.9 venv`

name of virtual environment

# Virtual Environments

- `source venv/bin/activate`
  - Activates the virtual environment
  - Note that `venv` is the folder where you created your virtual environment
- `deactivate`
  - Deactivates the virtual environment (if you're in one)
- Packages will not be installed globally via `pip`
- To remove a virtual environment, simply delete the folder (`venv`)
- Keep a text file that includes all the required packages
  - To recreate the virtual environment, simply run:

```
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 following command:
  - `django-admin startproject <name> .` ← Create the project in the current directory
  - This creates the skeleton code for your project
  - You should see the following files created:
    - `manage.py`: a command-line utility that can do various things
    - A folder that's the same as your project name: contains project-wide settings
  - For an alternative tutorial, visit:
    - <https://docs.djangoproject.com/en/4.1/intro/tutorial01/> (highly suggested)

# Development Server

- Used for testing and development *only*
  - Not suitable for deployment
- `python3 manage.py runserver`
  - Starts the development server
- Your website is accessible at:
  - <http://localhost:8000>
  - localhost: domain name for this machine
  - 8000: the port number
    - To avoid conflict with actual web server
- Ignore the migration warning for now



# Django apps

- Django is intended for big projects
  - Can have 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 manage products
- `./manage.py startapp <name>`
  - Creates a new app and its folder
    - Contains `views.py` (today), `migration` folder, `models.py`, and `admins.py` (next week)
- Remember to add the app name to `INSTALLED_APPS` in `settings.py`
  - Otherwise, it won't be loaded

# Django View

- Code that runs when a specific endpoint, i.e., URL, is requested
  - Can be any callable, e.g., function, class that implements `__call__`
- Example: simple view

```
from django.http import HttpResponse

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

- Function 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

- 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 this 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
- Example:

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

- The corresponding view function now takes two extra arguments:

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

- Exercise
  - Do Question 6 on Quercus

# HTTP Request Data

- `request.method`
  - Tells you 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
- Exercise
  - Do Question 7 on Quercus

# 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
  - User can bypass front-end restrictions
    - Inspect element and submit form with erroneous input data
    - Handcrafted HTTP request

# Processing POST request

- Validation error
  - If data is invalid, should return a 400-level error code
    - 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 or index page after logging in
- Use HttpResponseRedirect or redirect (from django.shortcuts)
  - E.g., `redirect('/some/url')`
  - Is there a problem with the above example?

# 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 your redirects to break
- Add **name** or **namespace** argument to the path object
- project's urls.py:

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

- accounts' 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 in spirit to PHP, running PHP code inside <?php ... code ... ?>
- <https://docs.djangoproject.com/en/4.1/topics/templates/>
- Variables
  - Surrounded by {{ and }}, like this:  
`<p>Hello, {{ username }}.</p>`
- Tags
  - Provides arbitrary logic in the rendering process
  - Surrounded by { % and % }, like this:  
`{% if has_error %} <p class="error">Bad! </p>{% endif %}`

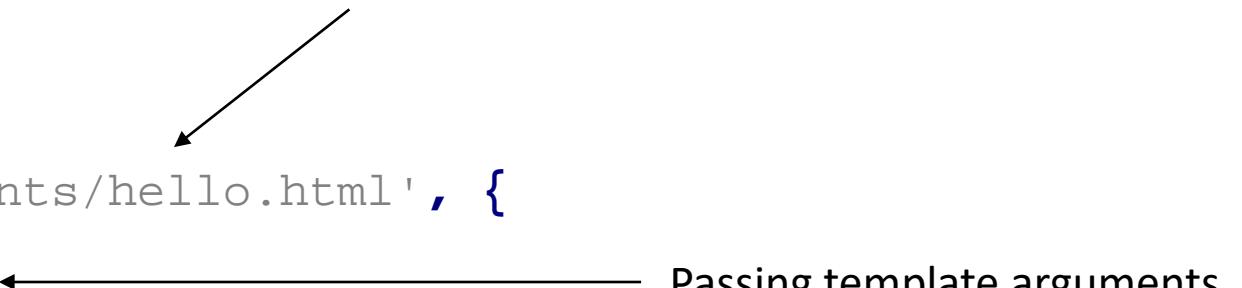
# Template Response

- 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 easily use Django templates
- E.g., in `accounts/views.py`:

```
from django.shortcuts import render
def signup(request):
    error = None
    code = 200      # success
    ...
    return render(request, 'accounts/hello.html', {
        "error" : error,
        "username" : username,
    }, status=code)
```

template path

Passing template arguments



# 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 secretly deplete his/her bank fund
  - Prevention
    - Using CSRF token
      - Add this to your form `{% csrf_token %}`
      - It will generate the following:
- ```
<input type="hidden" name="csrfmiddlewaretoken" value="KbyUmhTLMpYj7CD2di7JKP1P3qmLlkPt">
```
- 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.
  - It simplifies the task of locating the files and serving them
- To use a static file, create a folder named `static` (recommended)
  - Put your static files in here, or its subfolders
- Add this to `settings.py`:

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

- In the HTML file, you can specify a static file like this:

```
{% load static %} ← Load this template tag
<!DOCTYPE html>
...

```

# Static Files

- Django development can serve static files
  - For testing only. 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 = [
    path('accounts/', include('testapp.urls', namespace='accounts')),
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- Exercise
  - Do Question 8 on Quercus

# Before you go

- Assignment 1
  - Extension added
  - Due Monday January 30<sup>th</sup>
  - MarkUs Autotester is now available
    - Your mark will be the same as the autotester output
- Project
  - Students without groups will be assigned one on January 24th
  - Sign up on Calendly to book an interview session for project grading
    - <https://calendly.com/csc309-2023s>
  - Phase 1 due date is Feb 5<sup>th</sup>

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 5: Django Templates and Models

Instructor: Kuei (Jack) Sun

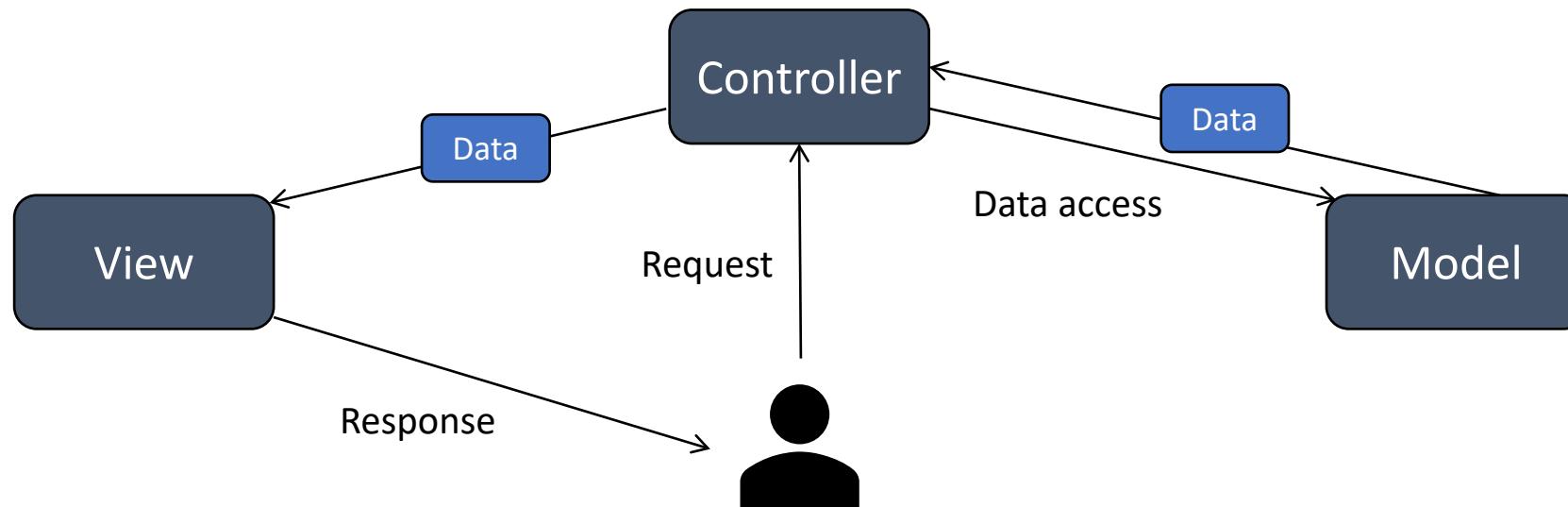
Department of Computer Science  
University of Toronto

# Architectural Design Patterns

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

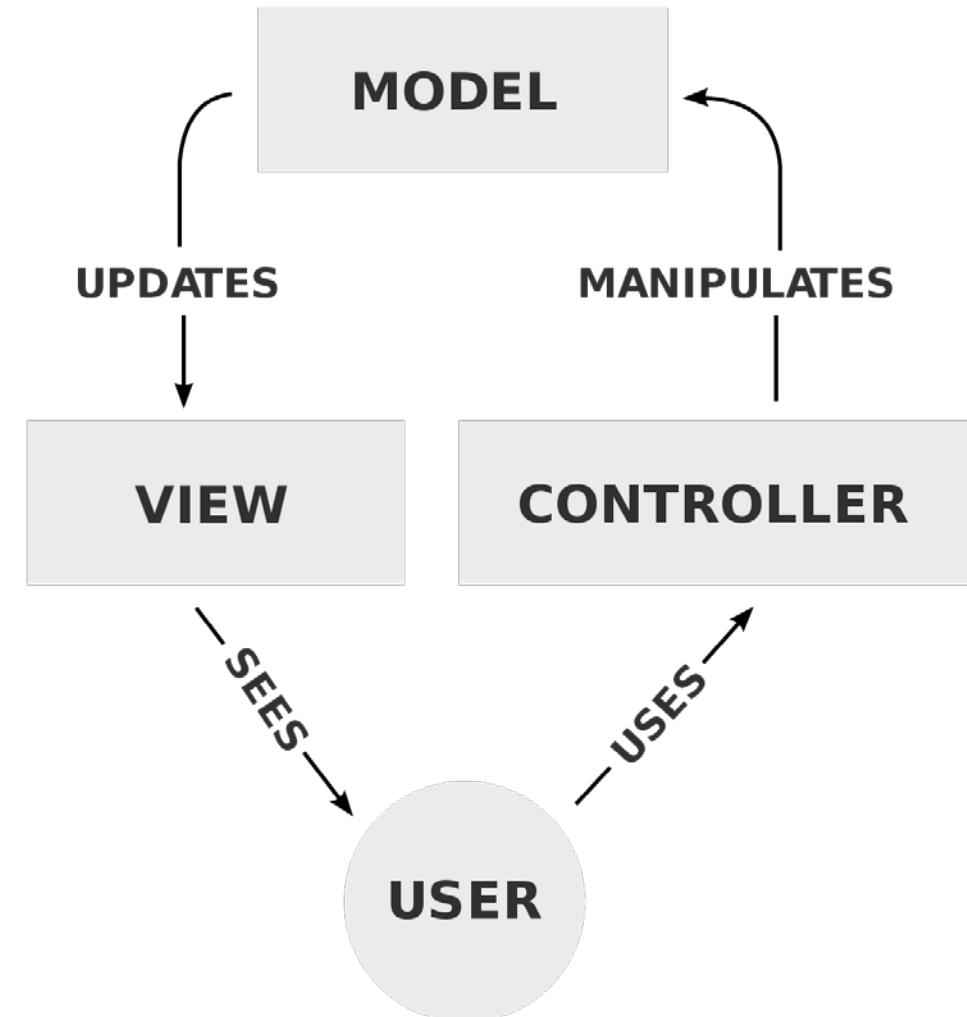
# Model-View-Controller (MVC)

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



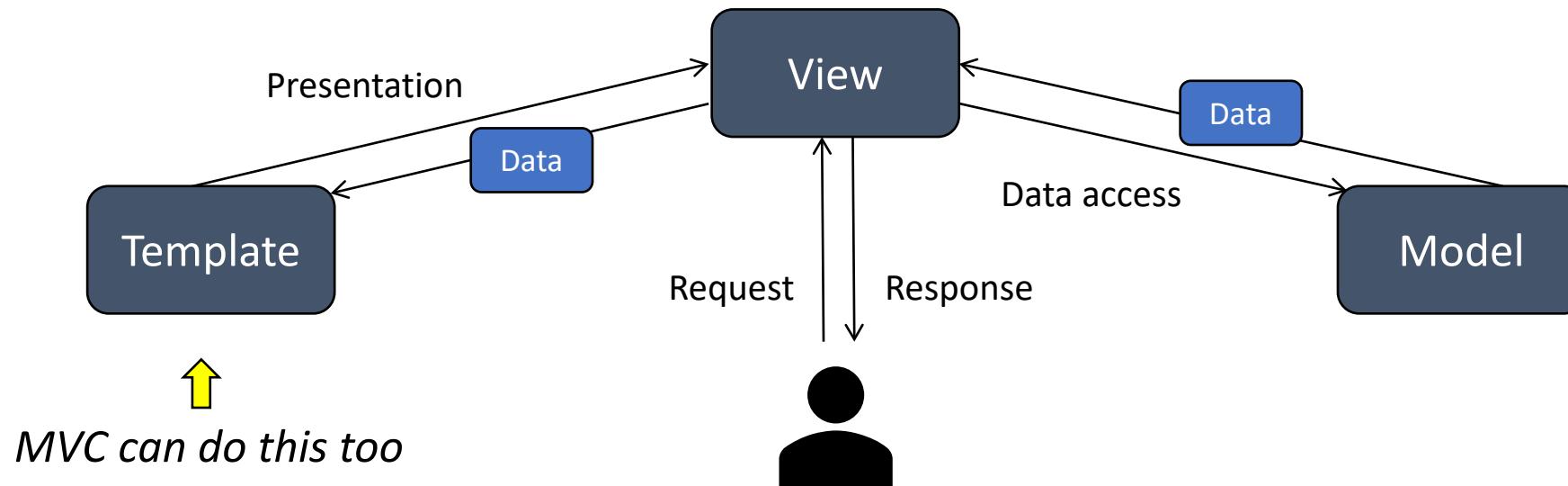
# MVC Pattern

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



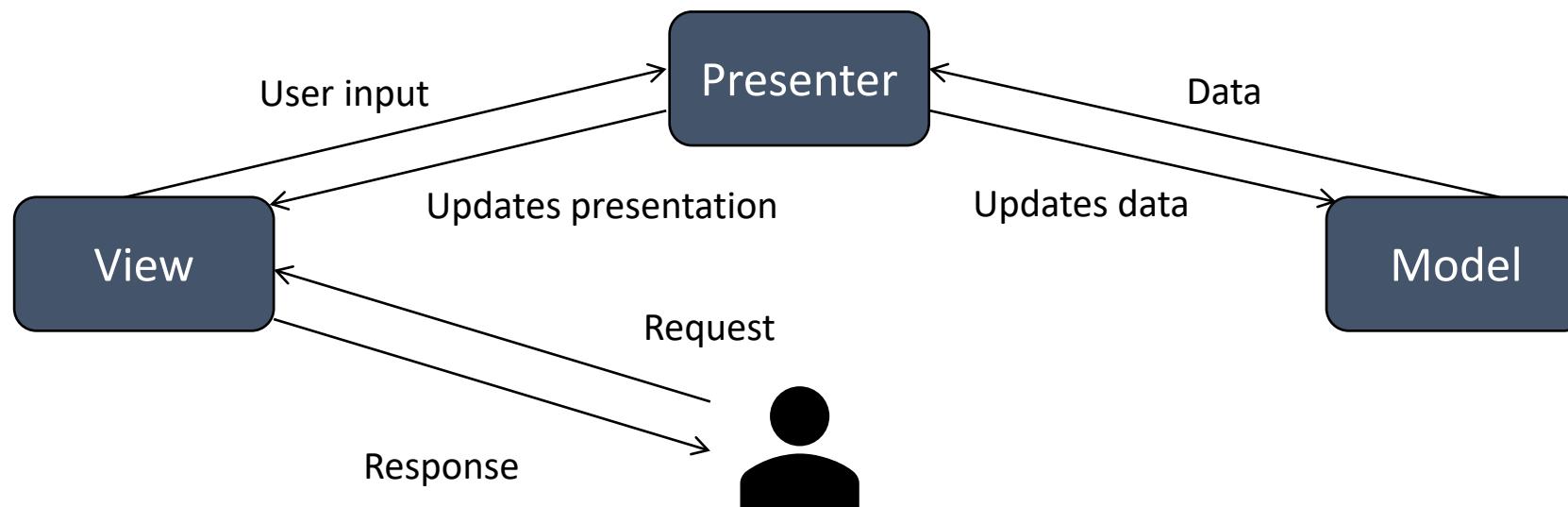
# Model-View-Template (MVT)

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

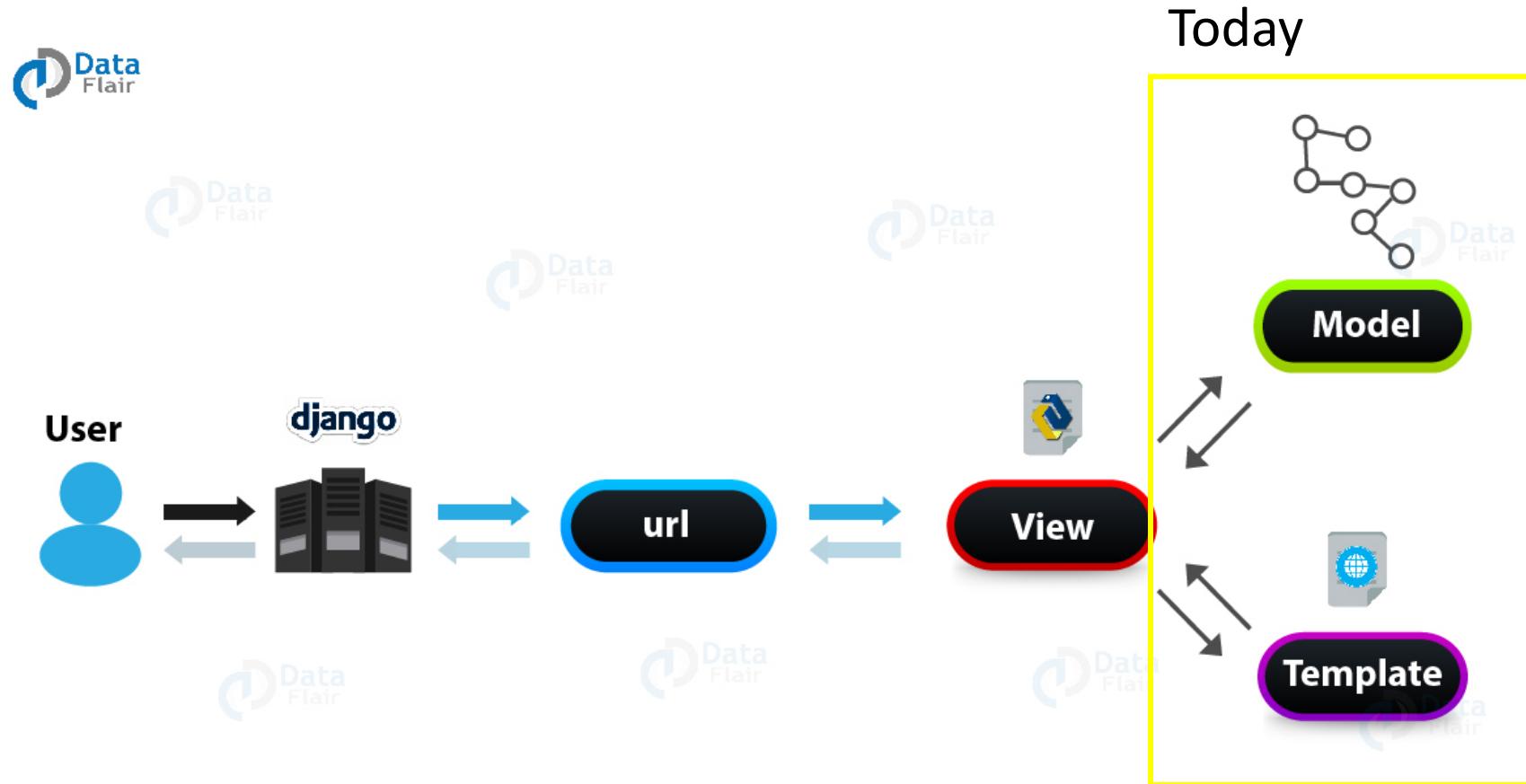


# Model-View-Presenter (MVP)

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



# Django's Architecture



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

# Django Template Language

- Review

- DTL supports imperative programming features
- Data passed from view to template via **context**.
- Variables are replaced by data from context
- Tags control the logic of the template

- Control flow tags

- for loop

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

- if, elif, else

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

# Django Template Tips

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

# Django Template Filters

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

# Django Template Inheritance

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

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

# Django Template Inheritance

- Example child template

```
{% extends 'parent.html' %}           ← extends must be on the first line  
{%- load static %}  
  
{% block staticfiles %}               ← child template does not  
    {{ block.super }}  
    <link rel="stylesheet" href="{% static '/css/child.css' %}">  
{% endblock %}  
  
{% block title %}My Child{% endblock %}  
  
{% block sidebar %}                  ← adds parent's code in block  
    {{ block.super }}  
    <li><a href="{% url 'child' %}">Child Page</a></li>  
{% endblock %}  
  
{% block content %} <h1>Child Page</h1> <p>{{ lorem }}</p> {% endblock %}
```

Note: code outside of block tags are ignored

# Root Template Folder

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

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

adds this line

# Django include tag

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

# Database

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

# Object Relational Mapper

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

# Object Relational Mapper

- Advantages

- Simplicity: no need to learn SQL or other database languages
- Consistency: everything is in the same language (e.g., Python)
  - Enables object-oriented programming
- Flexibility: can switch database backend easily
- Security: runs secure queries that can prevent attacks like SQL injection
- Seamless conversion from in-memory types to storage types, and vice versa
  - E.g., storing datetime as an integer in database

- Disadvantage

- Additional layer of abstraction reduces overall performance
- Hiding implementation detail may result in poorly designed database schema

# SQLite

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



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

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

# Django Models

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

# Django Security Model

- Authentication

- Verifies identity of a user or service
- Typically done through verification of correct username and password
  - Other methods include API key, session token, certificate, etc.
- Two-factor authentication
  - Provides additional layer of protection by asking additional information
    - E.g., one-time passcode sent to email or phone

- Authorization

- Determines a user's access right
- Checks user's privilege level (group) and permissions

# User Authentication in Django

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

# Setting Up Database Tables

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

# Working with ORM Objects

- Create an object

- `User.objects.create_user(username='dan1995', password='123', \ first_name='Daniel', last_name='Zingaro')`
- Some fields are optional, e.g., `first_name`, `last_name`
  - Preview: the field is defined with `blank=True`

- Get all objects of the same type

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

- Get just one object based on exact match

- `dan = User.objects.get(username='dan1995')` ← can return not found or not unique.

- Delete object(s)

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

# Working with ORM

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

# QuerySets

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

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

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

# Update Queries

- Update a single instance

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

- Update everything in a QuerySet

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

- Attributes are locally cached values

- To refresh
  - dan.refresh\_from\_db()

- Exercise

- Do Question 5 on Quercus

# Authentication

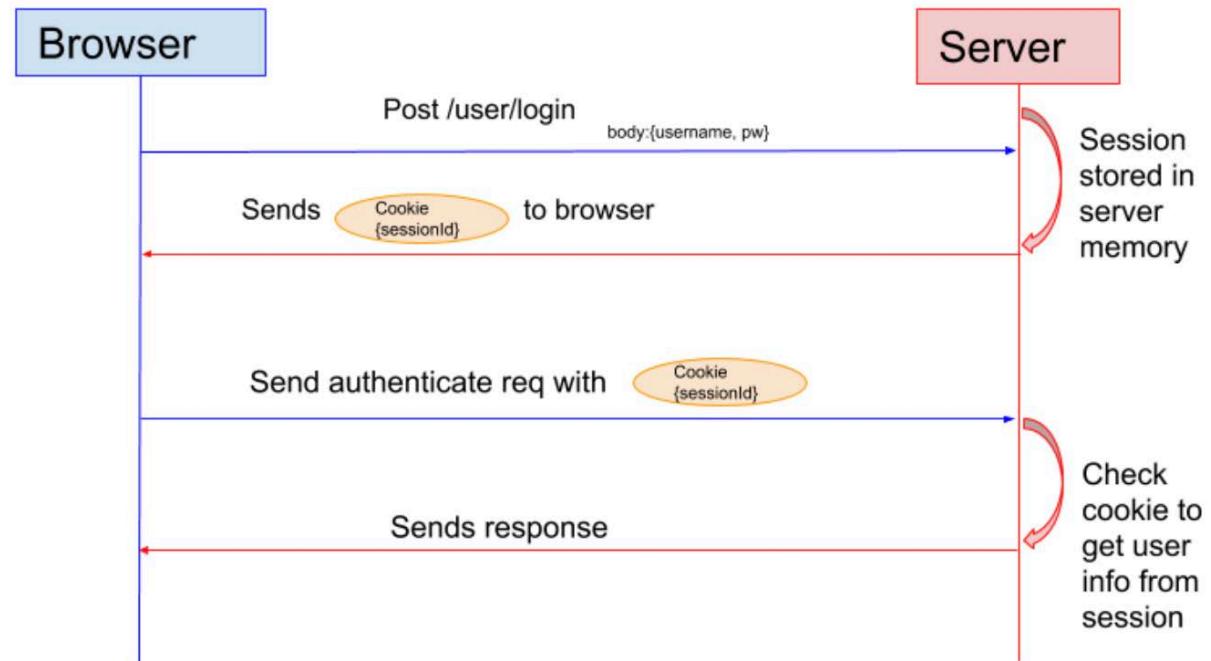
- Clients should tell the server who they are
  - Can use Authorization header in HTTP
  - Several authentication methods available
1. (Basic) password authentication
    - Sends username and password for every request
      - No concept of login and logout
    - User information is unencrypted and encoded in base64
      - Insecure without HTTPS

# Session Authentication

## 2. Session authentication

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

# Session Authentication



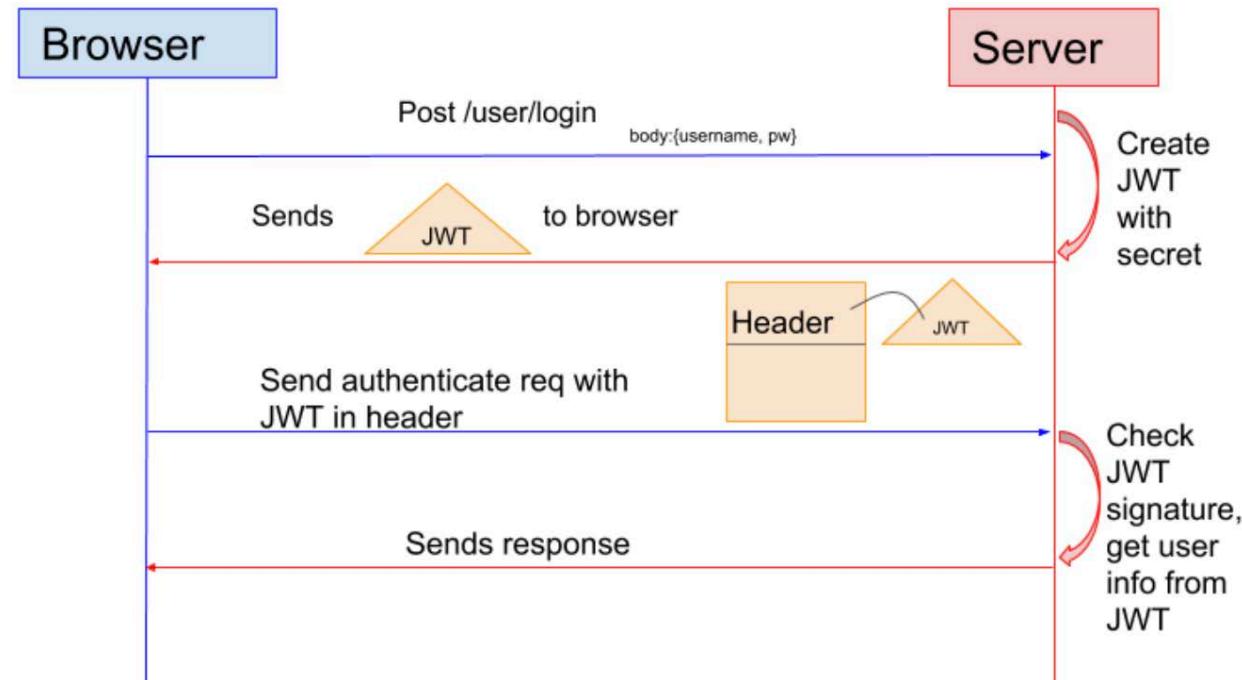
Source: <https://sherryhsu.medium.com/session-vs-token-based-authentication-11a6c5ac45e4>

# Token Authentication

## 3. Token authentication

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

# Token Authentication



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

# Django Session Authentication

- Checks that username/password combination is correct
  - `user = authenticate(username='john', password='secret')`
- Django's login function
  - Attaches user to the current session
  - `login(request, user)`
- Django does the session id lookup internally
  - User object is attached to the request object
    - `request.user`
  - User type is `AnonymousUser` if current visitor is not authenticated
- `logout()` function
  - Removes session data

# Admin Panel

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

# Exercise

---

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

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 6: Custom Models and CRUD

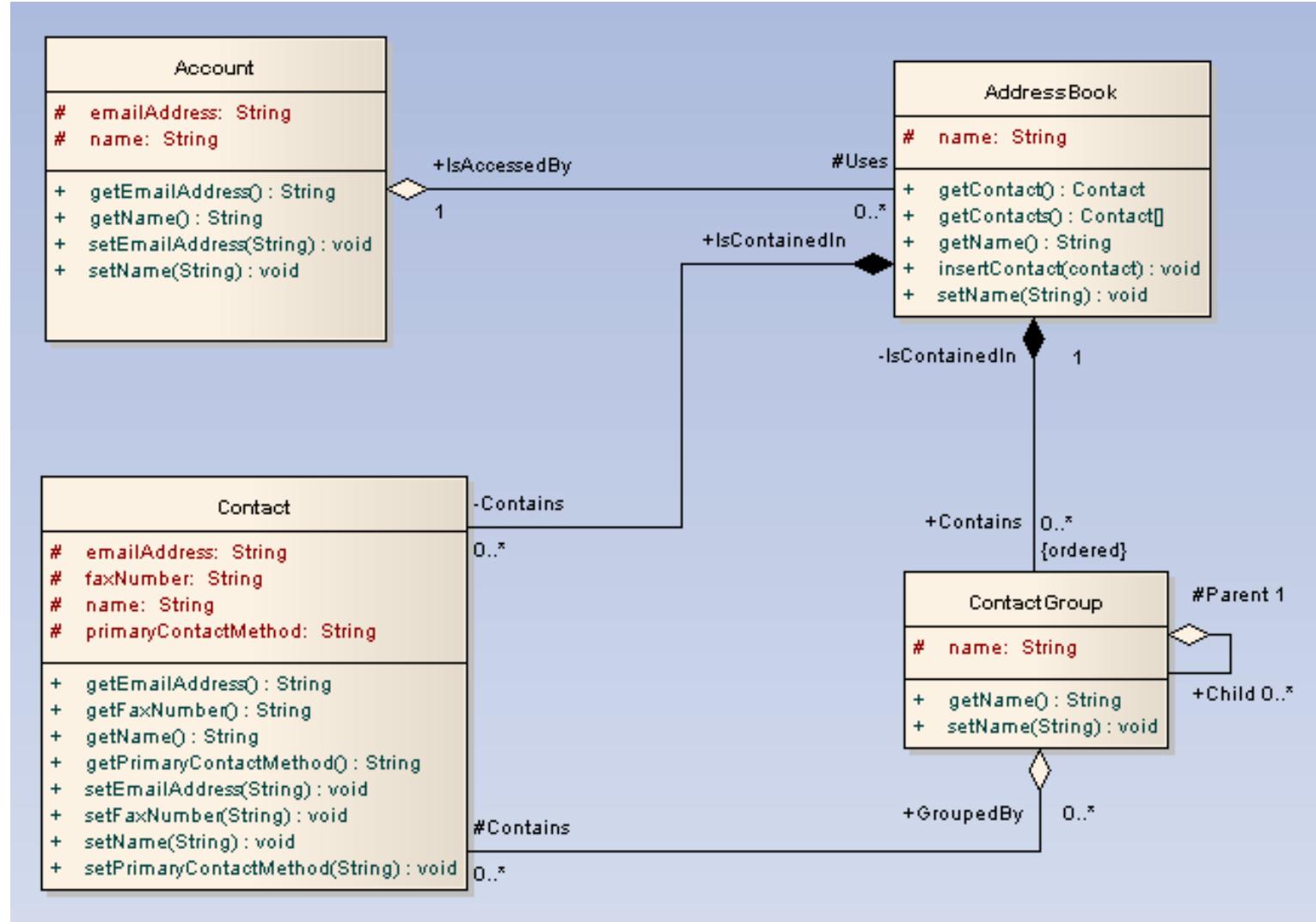
Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# Designing Models

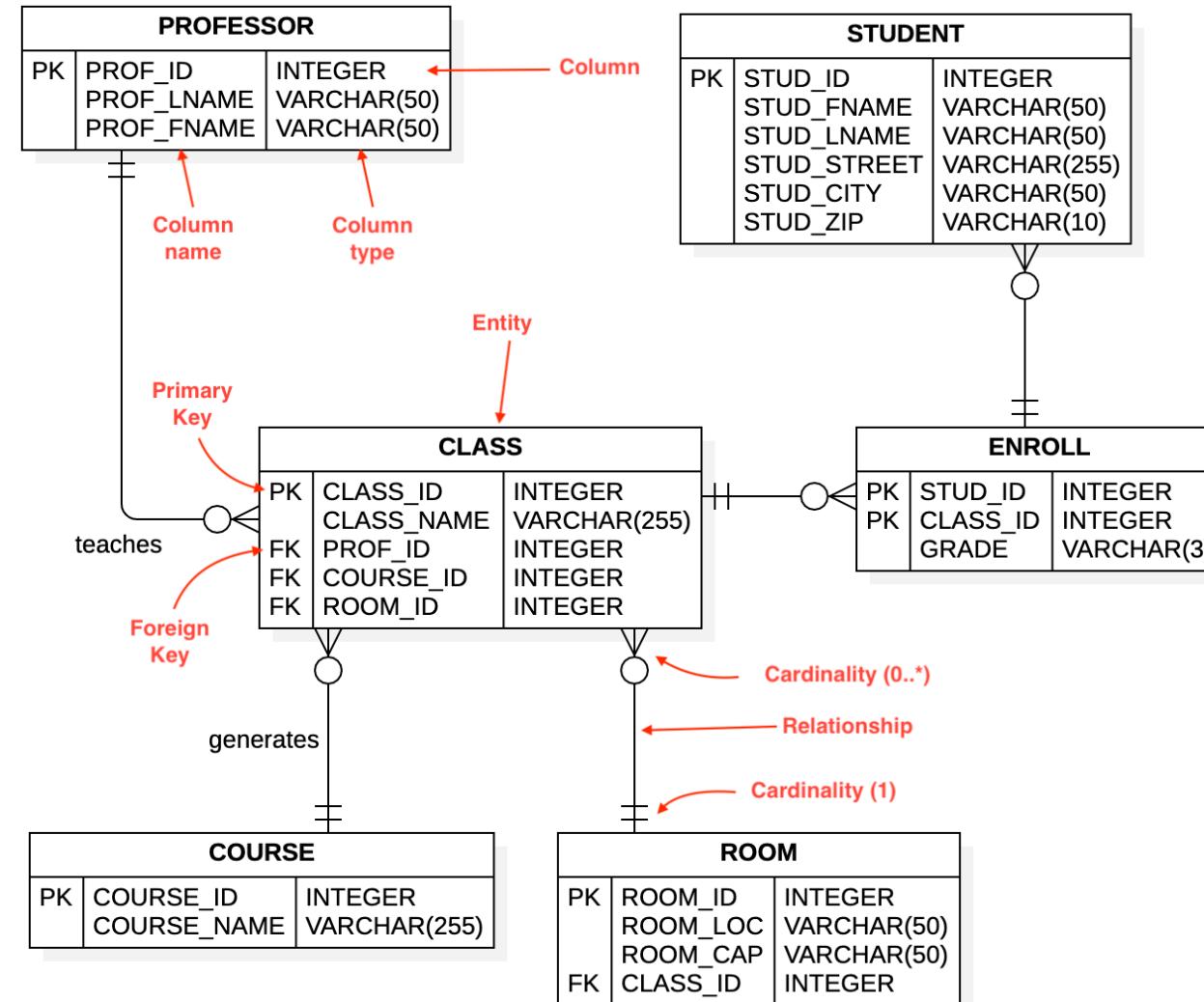
- Models involve representing and storing user data
  - A secure and efficient design is paramount to the success of an application
- General practice
  - Modeling should be done *before* coding starts
  - Changing models becomes more difficult when further into development cycle
    - Especially during the production phase
- Can be done independent of programming language or framework
  - E.g., Universal Modeling Language (UML)
    - Provides graphical notations to visualize the design of an application
    - Not learned in this course, take CSC301
  - E.g., ER (entity relationship) diagrams

# UML Diagram



Source: [https://sparxsystems.com/images/screenshots/uml2\\_tutorial/cl01.png](https://sparxsystems.com/images/screenshots/uml2_tutorial/cl01.png)

# Entity Relationship Diagram (ERD)



Source: <https://docs.staruml.io/working-with-additional-diagrams/entity-relationship-diagram>

# Django Models

# Creating Models

- In `models.py`, add subclasses of `django.db.models.Model`
- Add **fields** from your diagram to each model
  - <https://docs.djangoproject.com/en/4.1/ref/models/fields/>
  - Each field is mapped to a database column by the ORM layer

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

- Convention for large projects:
  - Create a `models` directory and put each 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 them
  - FileField, ImageField
- Numbers
  - IntegerField
  - BigIntegerField
    - For large numbers (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`
  - Used to allow the lack of value
- `blank: bool = False`
  - Allows field to be unspecified
  - Automatically given *empty* value
    - E.g., zero, empty string, etc.
- `unique: bool = False`
  - Requires value to be unique
    - Throughout the database table
  - Otherwise throws `IntegrityError`
- `choices: [ ( Any , Any ) , ... ]`
  - A list of key-value pairs
  - Key should be an abbreviation
  - Value is what is displayed to user
- `max_length: int`
  - Only for CharField or its subclasses
  - Limits the number of characters
- `default: Any`
  - 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 behavior when referenced object is deleted
    - **CASCADE**: delete everything that references the deleted object
    - **SET\_NULL**: set reference to NULL. (Used to support 0..1 relationship)
- `related_name`
  - Provides alternative name for reverse traversal by a field
    - Default is `<model_name>.set`, e.g., `prof.class_set.all()`

# Foreign Key

```
class Professor(models.Model):  
    name = models.CharField(max_length=255)
```

Does not delete the class  
when the professor is deleted.  
Requires null=True.

```
class Class(models.Model):  
    room = models.CharField(max_length=255)  
    capacity = models.IntegerField()  
    instructor = models.ForeignKey(Professor, on_delete=models.SET_NULL,  
                                   null=True, related_name="classes")  
    course = models.ForeignKey('Course', on_delete=models.CASCADE)
```

Forward reference must  
be specified in a string

```
class Course(models.Model):  
    code = models.CharField(max_length=16)
```

deletes the class when  
the course is deleted

# Models to Tables

- Every time the model changes, you must create and run **migrations**
  - `./manage.py makemigrations`
  - `./manage.py migrate`
  - More on this later
- By default, Django creates an AutoField named `id`
  - Configurable in `<app_name>/apps.py`
  - It is used as the primary key for the table
  - Can be overridden with `primary_key=True` for another column
    - Not a good idea in general
- Exercise 5
  - Do question 2 on Quercus

# Admin Panel

- <https://docs.djangoproject.com/en/4.1/ref/contrib/admin/>
- Register your model to the admin panel
  - In `admin.py` add:
    - `admin.site.register(Store)`
- Field options for admin panel (and also form)
  - `help_text`: adds help text in tooltip
  - `verbose_name`: alternative name for the field
- `__str__()`
  - Called when typecasting Python object to str, i.e., `str(obj)`
  - 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
  - Too many levels of inheritance can reduce performance

```
class Product(models.Model):  
    name = models.CharField(max_length=255)  
    price = models.FloatField(default=0.,)  
    store = models.ForeignKey(Store, on_delete=models.CASCADE)  
  
class Produce(Product):  
    expiry_date = models.DateTimeField()
```

# Many-to-Many Relationship

- **ManyToManyField**

- Defines a many-to-many relationship, e.g., classes and students
- By default, an intermediary join table is used to represent the relationship
- Also supports recursive relationship (ForeignKey can do the same)

```
class Student(models.Model):  
    friends = models.ManyToManyField("self")
```

← Symmetrical relationship

- Can specify the intermediary table manually

```
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
  - Associate two objects in a one-to-many or many-to-many relationship

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

- Can access foreign object(s) through current object
  - May require multiple database queries, so be mindful of performance

```
apple.refresh_from_db()
apple.owner.first_name = 'Tim'
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
  - 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)
```

# 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
    - i.e., database tables are independent of the model classes
- ORM must apply *current* application schema to the database
  - This requires running DDL (data definition language) queries
    - E.g., CREATE, ALTER, TRUNCATE, or DROP commands
    - E.g., create/remove a table
    - E.g., create/remove a column due to updated model
    - E.g., change column options due to updated field attributes

# Migration

- Whenever model changes, database should **migrate** to the new state
- Django does not perform migration automatically.
  - Why?
    - Quercus Exercise 5 Question 7
- Django does not monitor potential model changes
  - If there's a mismatch, database exception will occur when executing queries
    - E.g., selecting a table/column that does not exist
- There are two steps to perform migration
  1. Make migration
  2. Migrate

# Make Migration

- Generates a list of operations needed to migrate to new state
  - Similar to a git commit
    - It tracks what has changed since the last migration

```
class Migration(migrations.Migration):  
    dependencies = [ ('products', '0001_initial'), ]  
    operations = [ migrations.AlterField(  
        model_name='product', name='name',  
        field=models.CharField(max_length=128),  
    ), ]
```

- The 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`

# Make Migration

- 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
  - 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
  - A new migration file is created with a Migration class defined
    - E.g., 0003\_delete\_produce.py
- Note:
  - `__init__.py` must be present in the migrations folder for the command to work

# Applying Migrations

- `./manage.py migrate`
  - DDL queries are generated from each migration file
  - Then, they are executed to apply the migration
- How does Django know which migration has not been applied?
- Migration information is stored in the database
  - In a table named `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

- You should *never* need to manipulate migration files/table
  - Let Django ORM do its things
- Lots of assumptions go into its implementation
  - E.g., deleting a migration file may permanently break future migrations
- Migrate errors can take many hours to resolve
  - Can occur when same model is modified by multiple developers
    - E.g., similar to resolving conflicts during git merge
  - Typically avoided because migrations have dependencies
- Possible solutions
  - You can *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. Back up is recommended
      - E.g., created tables may be deleted, new columns may be deleted
    - Create a full backup of the database in a JSON file
      - `./manage.py dumpdata > db.json`
    - Load the database with data from backup file
      - `./manage.py loaddata db.json`
    - More information: <https://docs.djangoproject.com/en/4.1/ref/django-admin/>
  - The corresponding row in `django_migrations` is deleted
    - You may then delete the migration file that was unapplied
      - **Do not delete** a migration file before it is unapplied

# Migration Tips

- `./manage.py migrate --fake`
  - Only creates a row in `django_migrations`
    - Without executing any database queries
  - Use case
    - When the database state is already consistent with the models
- Full reset
  - Last resort, if the migrations are becoming too messy
    - Remember to make a backup if there are needed user data in the database
  - 1. Delete the entire database
    - E.g., delete `db.sqlite3`
  - 2. Delete all migrations files to start over from fresh

# Advanced Views

# Class-Based Views

- Function-based views can become too big
  - 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 for large projects
  - 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

- Function-based 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()
```

- Class-based views

```
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 and ListView
- Generic editing views
  - <https://docs.djangoproject.com/en/4.1/ref/class-based-views/generic-editing/>
  - Designed to create, update, or delete data
  - CreateView, UpdateView, DeleteView, FormView (next week)

# List View

- A page that displays a list of objects
- <https://docs.djangoproject.com/en/4.1/ref/class-based-views/generic-display/>

- 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
- queryset
  - Same as model, but allows you to specify a subset or ordering
- `Store.objects.filter(is_active=True)`
- `get_queryset(self)`
  - Override to customize query set
- URL arguments stored under `self.kwargs` e.g., `self.kwargs['pk']`
- `context_object_name`
  - By default, this is `object_list` or `object` (`DetailView`)
  - Allows alternative context name
- `get_context_data(self, **kwargs)`
  - Override to add extra context
- `get_object(self)`
  - `DetailView` only
  - Override to retrieve object
- 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
  - Template

```
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')
```

```
{% 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 when success
  - Must use `reverse_lazy` here
    - URL dispatcher loaded later
- `get_success_url(self)`
  - Needed if `reverse` needs argument

```
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>
```

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 7: Django Forms and REST API

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# Django Form

- <https://docs.djangoproject.com/en/4.1/topics/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
  - Extremely similar to a 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` directory 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)
  - Override to add custom logic

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

# Model Form

- <https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/>
- Form that maps closely to Django model

```
class ArticleForm(forms.ModelForm):  
    class Meta:  
        model = Article  
        fields = ['pub_date', 'headline', 'content', 'reporter']
```

- 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:

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

- With a class-based view:

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

# Form Widgets

- Forms can be passed into template and rendered
  - E.g., {{ form }}
  - Result would be based on the form renderer (can be customized)
- 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 customize the rendering

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

- Not recommended for large projects
  - In MVC pattern, **view** should be separate from **controller**

# Form View

- One of Django's generic editing view
- Similar to other CRUD views, except more customizable
- `form_valid` method
  - Called when form is valid, i.e., the POST request contains valid data
  - Where business logic should be placed

```
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: override 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 implements the `get_object` method
- A default `form_valid` method is implemented that saves the object

```
class SignupView(CreateView):
    form_class = SignupForm
    template_name = 'accounts/signup.html'
    success_url = reverse_lazy('accounts:welcome')

    def form_valid(self, form):
        self.request.session['from'] = 'signup'
        return super().form_valid(form)
```

Automatically saves the model object, i.e., User

# Authenticated Views

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

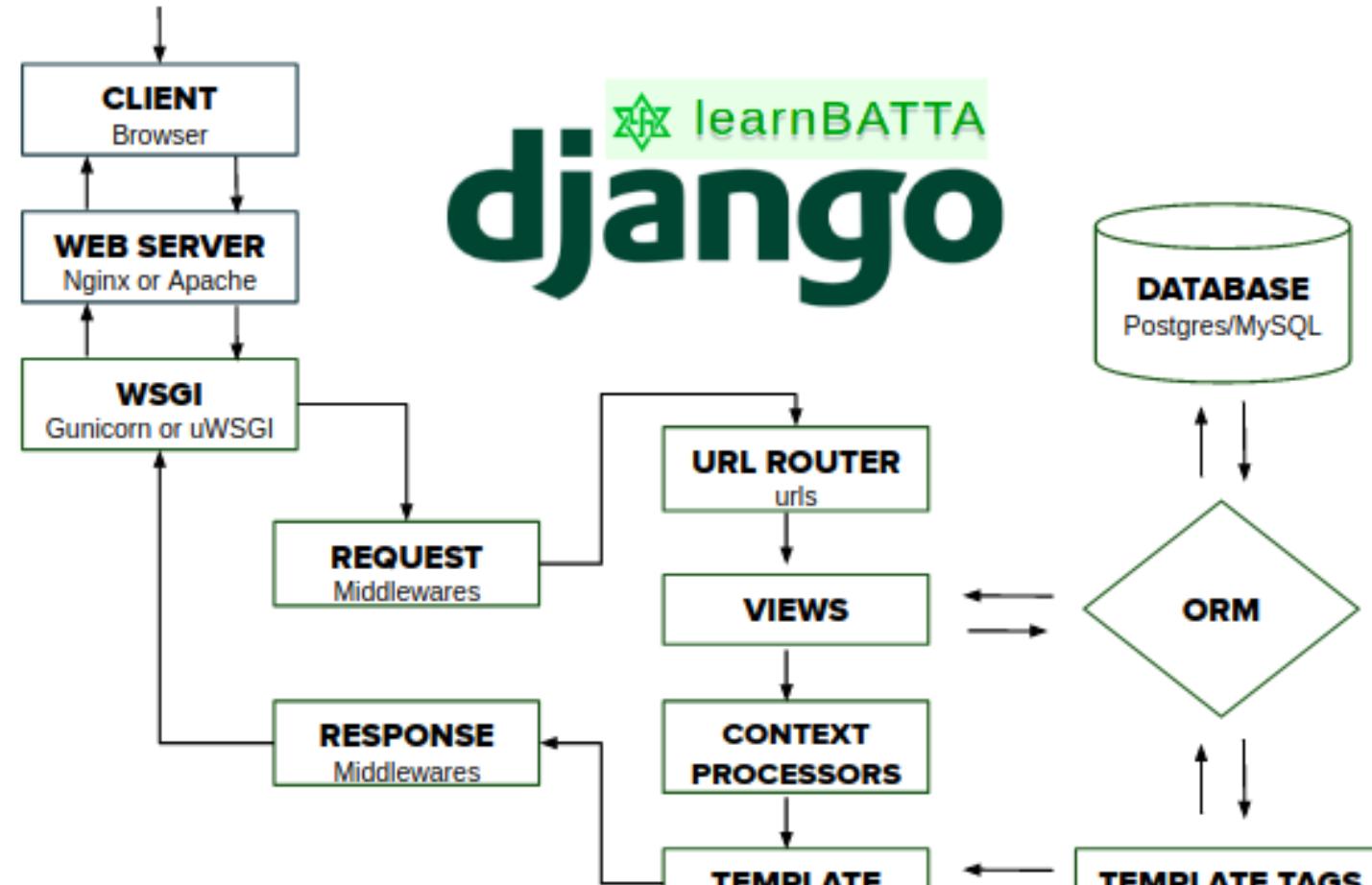
```
from django.contrib.auth.decorators import login_required
@login_required(login_url=reverse_lazy('accounts:login'))
def admin(request):
    return render(request, "accounts/admin.html", {})
```

- Class-based views:
  - Requires login\_url to be specified for redirect

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

# REST APIs

# Current way of building Django website



request-response lifecycle in Django

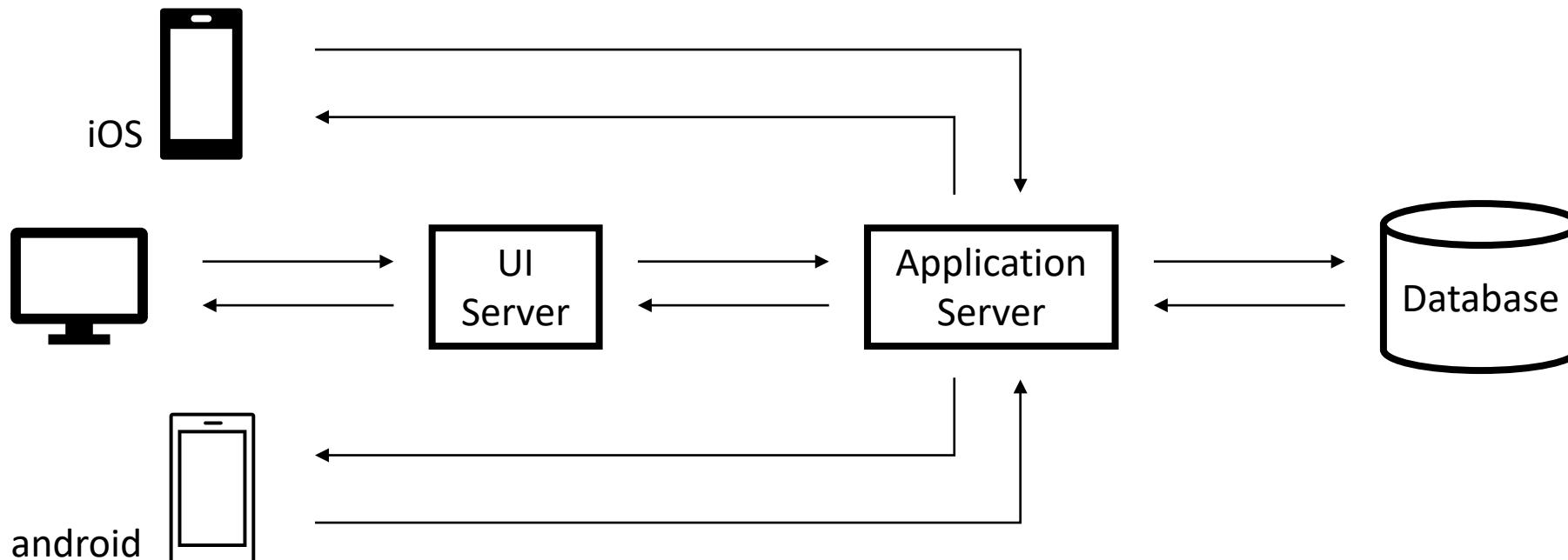
Source: <https://learnbatta.com/blog/understanding-request-response-lifecycle-in-django-29/>

# Full Stack Framework

- Django is a **full-stack framework**
  - Libraries 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*
  - e.g., web, android, iOS
- Improves **modularity**
  - 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 communicates via HTTP requests
- Backend views are responsible for data retrieval and manipulation
  - Should not care about how data is presented
    - e.g., should not handle templates or static files
      - i.e., does not need to work with HTML or CSS
- How should frontend and backend communicate then?

# JavaScript Object Notation

- Popular standard for backend responses
- Derived from JavaScript syntax for defining objects
  - Simplifies use in a browser, which supports JavaScript natively
- Advantages
  - Easy to read, easy to use, and fast
  - Many programming languages have built-in parser and support

- Example:

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

# JSON Format

- Primitives types
  - number, string, boolean and 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
  - Primitive or aggregate

- Example:

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

# JSON Exercise

- Given the following tables where each store has an owner, *serialize* the User with username jack. *Nest* all related data.
- Store

<b>id</b>	<b>name</b>	<b>url</b>	<b>email</b>	<b>is_active</b>	<b>owner_id</b>
1	Apple	<a href="https://www.apple.com">https://www.apple.com</a>	apple@test.com	1	1
2	Adidas	<a href="https://www.adidas.com">https://www.adidas.com</a>	adidas@test.com	1	2
3	Nike	<a href="https://www.nike.com">https://www.nike.com</a>	nike@test.com	1	1
4	Sobeys	<a href="https://www.sobeys.com">https://www.sobeys.com</a>	sobeys@test.com	1	null

- User

<b>id</b>	<b>username</b>	<b>first_name</b>	<b>last_name</b>	<b>email</b>	<b>last_login</b>
1	jack	Jack	Smith	jack@test.com	2023-02-10 07:23:53.568000

# 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, PUT, POST, etc.
    - Resource should be identified through URLs
  - 2. Requires stateless client-server communication
  - 3. Responses should be clearly labeled 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 now, due to advent of REST

# 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 this:

```
REST_FRAMEWORK = {  
    'DEFAULT_PERMISSION_CLASSES': [  
        'rest_framework.permissions.AllowAny'  
    ]  
}
```

No authentication required. Do not use.

# REST Views

- Same idea, but 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 format that can be **transferred** and **reconstructed**
- DRF provides JSON serializer
  - Very similar in flavor as Django Model Form
  - Plain serializer (not mapped to a model) also available
- Create a **serializer.py** or a **serializers** directory in the app

```
from rest_framework.serializers import ModelSerializer

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

# REST CRUD views

- Same idea, but requires a model serializer instead
- CreateAPIView
  - Overrides create method (returns 201 Created on success, accepts HTTP POST)
- RetrieveAPIView and 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

# More about CRUD views

- `ListAPIView`
  - Requires `queryset` attribute or `get_queryset` method
- `RetrieveAPIView`, `UpdateAPIView`, `DeleteAPIView`
  - Requires `get_object` method
- `CreateAPIView`
  - Does not require any addition method or attribute
- You can mix multiple views in one class, i.e., multiple inheritance
  - Works as long as each view uses a different HTTP method
- Can use same serializer across different views
  - In some cases, you may want to create separate serializers

# Example

- Retrieve Store View

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

class StoresRetrieve(RetrieveAPIView):
    serializer_class = StoreSerializer
    def get_object(self):
        return get_object_or_404(Store, id=self.kwargs['pk'])
```



Django shortcut that returns 404 NOT FOUND if the object is not found

- Testing

- Postman or DRF's built-in browsable APIs in development mode

# Serialization Fields

- Fields have similar options to Django's model field
  - Exceptions: null → allow\_null, blank → allow\_blank
  - read\_only: makes a field non-writeable
- Field validations are done automatically
- Foreign Key
  - By default, serializes to id of referenced object
- 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)  
    ...
```

# Token-Based Authentication

# REST Authentication

- DRF's browsable API works with session auth
  - However, REST APIs must be stateless!
- REST APIs uses **token-based** authentication
- JWT (JSON Web Token) packages
  - The other package is deprecated; therefore, we will use `simplejwt`
    - [https://django-rest-framework-simplejwt.readthedocs.io/en/latest/getting\\_started.html](https://django-rest-framework-simplejwt.readthedocs.io/en/latest/getting_started.html)

## • Installation

- `pip3 install djangorestframework-simplejwt`

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

# Setting up simplejwt

- Create login view (provided by simplejwt)

```
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

- Five minutes by default
- Can be changed to other durations
  - <https://django-rest-framework-simplejwt.readthedocs.io/en/latest/settings.html>
- A **refresh** token can be used to extend its duration

# Obtaining a Token

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'API Network', and 'Explore' buttons. A search bar says 'Search Postman'. On the right of the bar are 'Invite', 'Settings', 'Bell', and 'Upgrade' buttons.

The main area shows a collection named 'localhost:8000/api/token/'. Below it, a specific POST request is selected, targeting 'localhost:8000/api/token/'. The request method is set to 'POST' and the URL is 'localhost:8000/api/token/'. There are 'Save' and 'Send' buttons.

The 'Body' tab is active, showing a 'form-data' structure with two fields: 'username' (value: 'jack') and 'password' (value: '123'). Other options like 'raw', 'binary', and 'GraphQL' are available but not selected.

Below the body, the response status is shown as 'Status: 200 OK' with a duration of 'Time: 123 ms' and a size of 'Size: 794 B'. There are tabs for 'Body', 'Cookies', 'Headers (10)', and 'Test Results'. The 'Body' tab is currently selected, displaying a JSON response with two keys: 'refresh' and 'access'. The 'refresh' value is a long string of characters, and the 'access' value is another long string.

At the bottom, there are various status indicators like 'Online', 'Find and Replace', 'Console', and 'Continue learning'.

# REST Permissions

- A set of permissions can be applied to APIViews
  - E.g.: IsAuthenticated
    - This 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)
```

- Custom permissions can be created as well
  - Subclass BasePermission and implement has\_permission method

# Using a Token

The screenshot shows the Postman application interface. At the top, there are navigation tabs: Home, Workspaces, API Network, and Explore. A search bar says "Search Postman". On the right side of the header are buttons for "Invite", "Settings", "Bell", and "Upgrade". Below the header, there's a list of recent requests: "POST localhost:8000/api/to", "POST http://localhost:8000/", and "GET http://localhost:8000/s". A new request button "+" and an ellipsis "...>" are also present.

The main area shows a request for "http://localhost:8000/stores/owned/" using a "GET" method. The "Headers" tab is selected, showing a table with one row. The row has a checkbox next to "Authorization", which is checked. The "Value" column contains the value "Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJ0b2tlbl9...". A yellow box highlights this value, and an arrow points from the text "Access Token" to it.

Below the headers, the "Body" tab is selected, showing a JSON response:

```
1 {  
2   "name": "Apple",  
3   "url": "http://www.apple.com",  
4   "email": "apple@test.com",  
5   "is_active": true,  
6   "owner": 1,  
7   "owner_username": "jack"  
8 }
```

The status bar at the bottom shows "Status: 200 OK", "Time: 28 ms", and "Size: 793 B". There are also buttons for "Save Response", "Pretty", "Raw", "Preview", "Visualize", "JSON", "Cookies", "Capture requests", "Runner", "Trash", "Continue learning", and a help icon.

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 8: Introduction to JavaScript

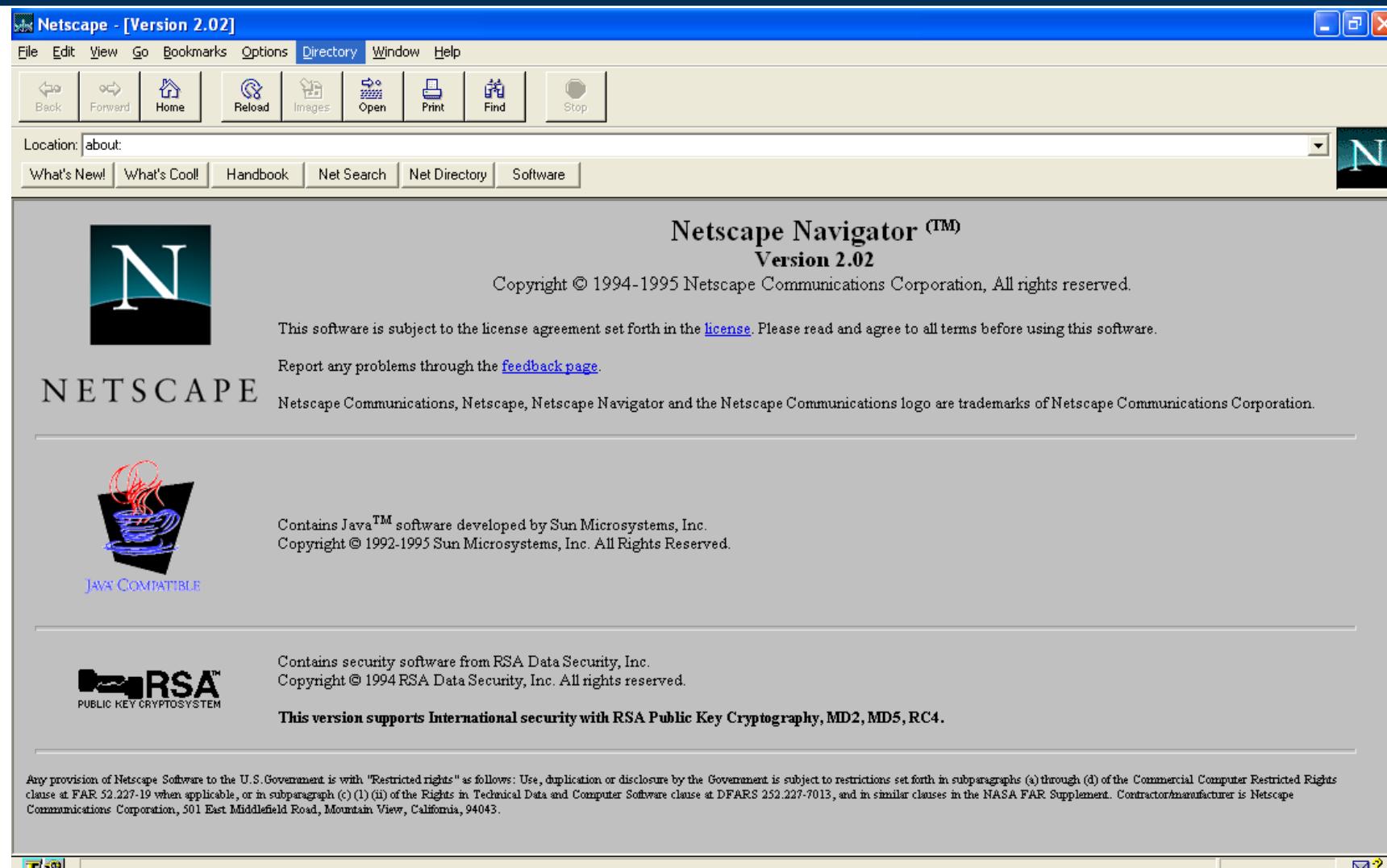
Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# JavaScript

- Often abbreviated as JS
- Where JSON (JavaScript Object Notation) is derived from
- A programming language that the browser understands
- Now, used in both frontend and backend development
  - Node.js being the most popular JavaScript engine
- A high-level, runtime interpreted scripting language
  - Dynamically typed and multi-paradigm
- One of the most popular programming languages
  - <https://madnight.github.io/githut/>
  - Recent trend suggests that TypeScript is rising in popularity

# Background



<https://www.springboard.com/blog/data-science/history-of-javascript/>

# History

- Netscape supported Java applets within its browser
  - A deal between Netscape and Sun microsystems
- Netscape programmer Brendan Eich developed [JavaScript](#) in 1995
  - Took him just 10 days
  - Originally intended as “glue code”
- JavaScript is *inspired* by Java, not otherwise similar
- TypeScript
  - Strongly typed language
  - A strict superset of JavaScript
  - Gaining traction for backend development



Brendan Eich

# JavaScript

# Variables

- 3 different ways to declare a variable

```
var x = 5;      // 1
let y = 4;      // 2
z = 3;          // 3
```

Use `const` to create constants, e.g.,  
`const PI = 3.14;`

## 1. var

- Creates a variable in global or `function` scope

## 2. let

- Creates a variable in global or `block` scope

## 3. Third way is not recommended

- Hard to know if declaring or modifying the variable
- Variables can be reassigned different types (like Python)

# Scope

- [https://www.w3schools.com/js/js\\_scope.asp](https://www.w3schools.com/js/js_scope.asp)
- JavaScript has 3 types of scopes
- Global scope
  - Variables outside of any function
  - Variables can be accessed from anywhere in the program
- Function scope
  - Variables defined anywhere inside the function are local to that function
  - Variable cannot be used outside the function
- Block scope
  - Variable is only accessible inside the block it is declared in

# Local Scopes

- Function scope

```
function foo(n) {  
    if (n > 10) {  
        var tmp = 2;  
    }  
    // tmp CAN be accessed here  
}  
  
// tmp CANNOT be accessed here
```

- Block scope

```
function foo(n) {  
    if (n > 10) {  
        let tmp = 2;  
    }  
    // tmp CANNOT be accessed here  
}  
  
// tmp CANNOT be accessed here
```

- `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
  - Integers or floating point
  - JavaScript does not differentiate between the two
- String
  - Same as Python.
  - Can be enclosed in single quotes or double quotes
- Boolean
  - true or false (same as Java)
- Function
  - A first-class citizen, can be created anywhere (locally or globally)

Use `typeof` function to see what the data type of a value is

# Function

- Syntax

```
function foobar(parameter1, parameter2, parameter3) {  
    // ...  
    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 and Python dictionary, 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 JavaScript) can be modified in two ways:

```
person.firstName = "Joe";  
person['lastName'] = "Jordan";
```

```
var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue",  
    height: 6.5,  
    company: null,  
};
```

# Array

- Syntax
  - Exactly the same as Python list
- Arrays are objects with special syntax
  - [https://www.w3schools.com/js/js\\_array\\_methods.asp](https://www.w3schools.com/js/js_array_methods.asp)

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Grape");           // same as append in Python
console.log(fruits[0]);        // prints Banana
console.log(fruits.length);    // prints 5
```

- Mutability
  - Objects and arrays are mutable (can be changed)
  - Other data types are immutable

# Method

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

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

```
var foo = { x : 0, inc : function(x) { this.x += x; } };
foo.inc(5);
console.log(foo.x);      // prints 5
```

# Class

- [https://www.w3schools.com/js/js\\_classes.asp](https://www.w3schools.com/js/js_classes.asp)
- Class is a special function that creates an object
- Requires special **constructor** method
- Classes supports **inheritance**

Extra reading:  
Getters and setters

```
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

```
if (typeof(x) === "number" && x < 0) {  
    x = -x;  
}  
else {  
    console.log("bad element");  
}
```

- == VS ===
  - == performs implicit typecasting to satisfy the comparison
  - === does not perform typecasting
- You should almost never use ==
  - <https://www.scaler.com/topics/javascript/difference-between-double-equals-and-triple>equals-in-javascript/>

# JavaScript can be Weird

```
: Console What's New  
top  
> 2 + 2  
< 4  
> "2" + "2"  
< "22"  
> 2 + 2 - 2  
< 2  
> "2" + "2" - "2"  
< 20
```



# Loops

- Classic C-like loop

```
for (var i=0; i<10; i++)
  console.log(i * i);
```

- While loop

```
var cars = [];
while (cars.length > 0)
  cars.pop();
```

- Array.forEach method

- Takes a function as argument

- for ... of loop

- Loops through elements
  - Typically the one you want to use

```
var cars = [];
for (var car of cars)
  console.log("Here is " + car);
```

- for ... in loop

- Loops through *properties*
  - Similar to looping through keys of a dictionary

# Switch

- Same as Java or C switch statements

```
switch (new Date().getDay()) {  
    case 4:  
    case 5:  
        text = "Soon it is Weekend";  
        break;  
    case 0:  
    case 6:  
        text = "It is Weekend";  
        break;  
    default:  
        text = "Looking forward to the Weekend";  
}
```

- Also accepts any mixture of data types for cases

# Document Object Model (DOM)

# JavaScript in HTML

- Can be placed into HTML in three ways

## 1. Inline JavaScript

```
<script>  
  console.log(1 + 2 + 3)  
</script>
```

## 2. JavaScript file

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

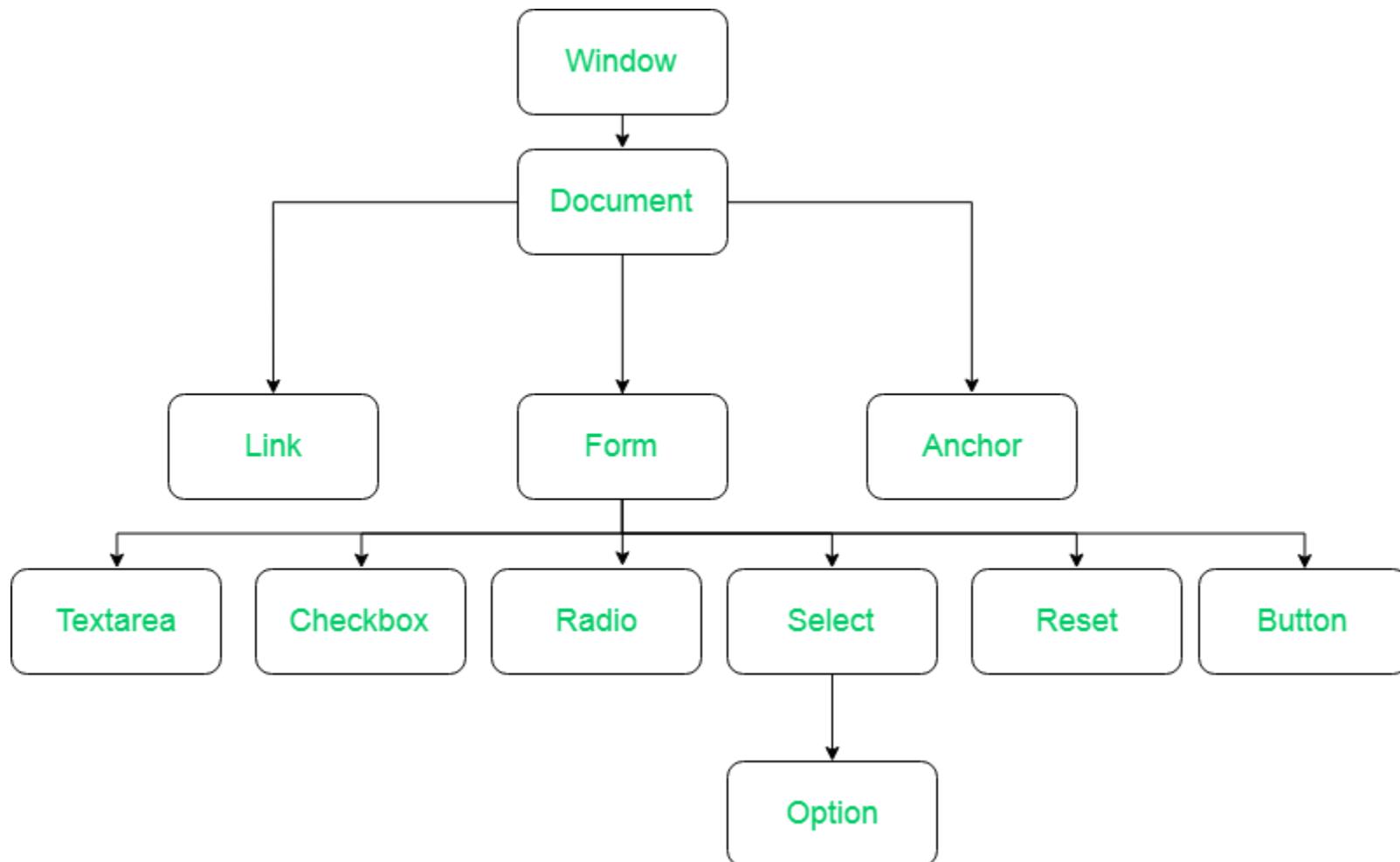
## 3. As an event attribute

```
<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
  - The ancestor of all nodes
- Each element can have zero or more child nodes
- Scripts accesses DOM elements through **document**
- **document**
  - A global variable
  - Contains various methods

# DOM Tree



<https://www.geeksforgeeks.org/dom-document-object-model/>

# Access Elements

- Various methods to retrieve element(s)
- Basic elements getters

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

- 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
- Example:

```
let img = document.querySelector("body section:first-child > img");
let par = img.parentNode;
console.log(par.childNodes.length);
```

- Quercus Exercise
  - Do Question 3

# Manipulating Elements

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

```
let body = document.body;
body.innerHTML = "<h3>hello!</h3>";
h3 = document.getElementsByTagName("h3");
h3.style.color = "green";
h3.setAttribute("class", "title");
console.log(h3.getAttribute("style"));
```

# Event

---

- JavaScript supports event-driven paradigm
- Various events are monitored by the browser
  - [https://www.w3schools.com/tags/ref\\_eventattributes.asp](https://www.w3schools.com/tags/ref_eventattributes.asp)
- Document events
  - Occurs to the entire page
  - E.g., onload, onkeydown, onkeyup
  - Convention: script should only be run after onload event
    - Ensures all contents have been loaded prior to running the script
- Element events
  - Occurs to a specific element, typically of a specific type of element
  - E.g., onclick, onmouseover, ondrag, oncopy, onfocus, onselect, onsubmit

# Event Listener

- Two ways to add an event listener function
  1. Set the event property of a DOM element to a function

```
h1 = document.getElementById("page-title");
h1.onclick = function() {
    this.innerHTML = "you just clicked on me!";
};
```

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)" onmouseover="console.log(new Date())"></h3>
```

# Quercus Exercise 4

- Create a form with the following fields:
  - Username
  - Email
  - Password
  - Repeat password
  - Security question: "What's  $8 + 16/4$ ?"
- Implement client-side validation, with the following checks:
  - Checks if the security question is answered correctly.
  - Checks password and repeat password are the same.
  - Checks if domain name of email address ends with "utoronto.ca".
- Errors should appear dynamically (only when there is error)

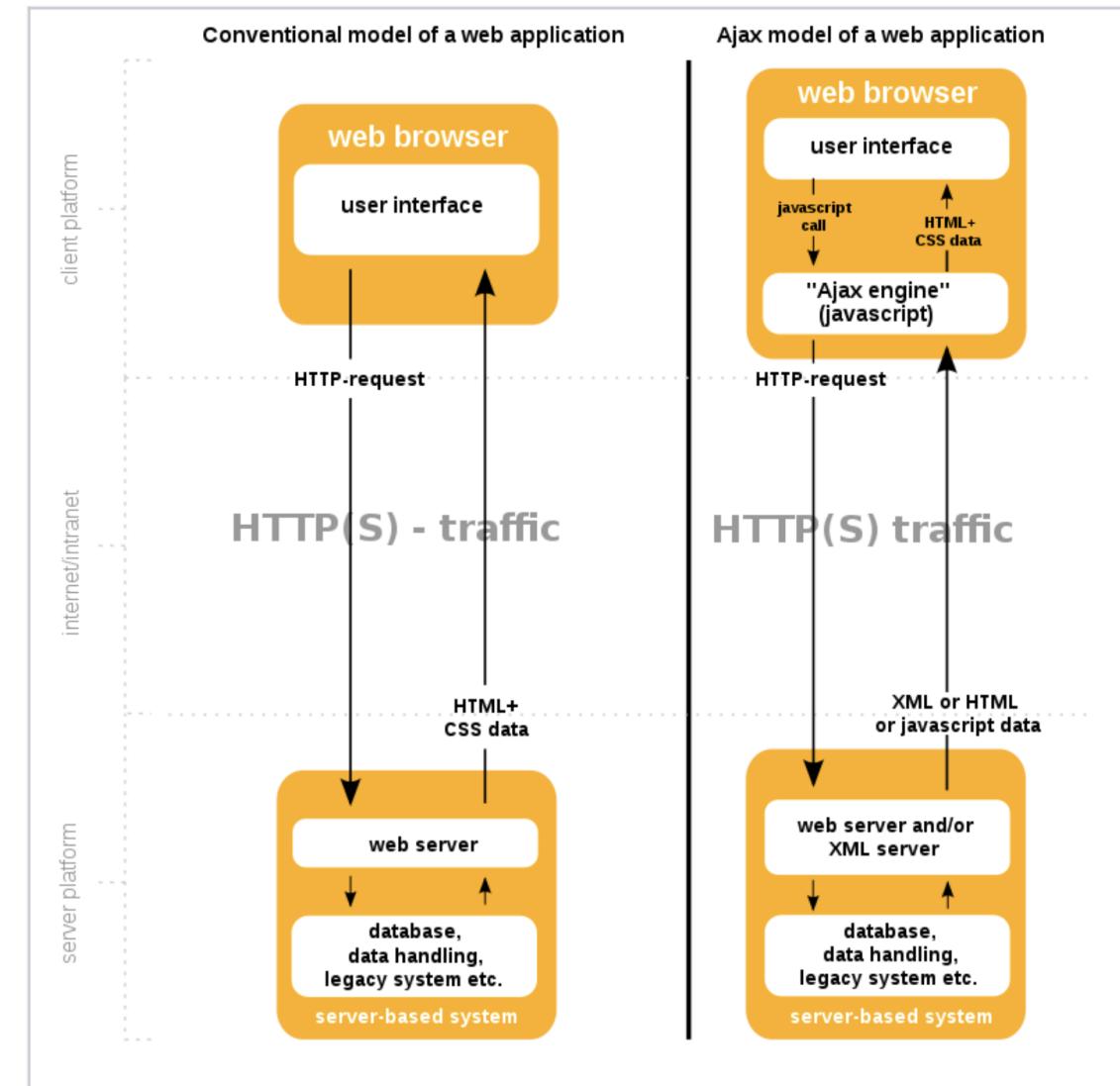
# Asynchronous Requests

# Requests

- One main request is made to the server
  - Upon entering URL or submitting a form
- 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
  - Can result in poor user experience due to high load time
  - Django's full stack framework does this.
- Solution?

# Asynchronous Request

- Ajax
  - Asynchronous JavaScript and XML
  - Browser sends background request
    - Main thread is not blocked
    - Web page 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

- Instantiate a new Ajax request object

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

- Define a handler for onreadystatechange

```
req.onreadystatechange = function() {  
    // Process the server response here.  
};
```

- Set method and endpoint and send request

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

- The event handler will trigger when response is received

# Example

- Load a text file and place response into element with id="demo"

```
function loadDoc() {  
    var xhttp = new XMLHttpRequest();  
    xhttp.onreadystatechange = function() {  
        if (this.readyState == 4 && this.status == 200) {  
            document.getElementById("demo").innerHTML = this.responseText;  
        }  
    };  
    xhttp.open("GET", "ajax_info.txt", true);  
    xhttp.send();  
}
```

- Must check readyState == DONE (4) before accessing responseText
- Too verbose, we won't actually use it as-is.
  - Next week, JQuery JavaScript library.

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 9: jQuery and Advanced JavaScript

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# jQuery

- One of the most popular JavaScript libraries
- Simplifies HTML DOM tree traversal and manipulation
- Helps with event handling and AJAX requests
- Installation
  - <https://jquery.com/download/>
  - `<script src="jquery-3.6.3.min.js"></script>`
  - Choose between compressed (smaller, faster) vs. uncompressed (readable)
  - Alternatively, can use directly from a CDN
    - E.g., <https://developers.google.com/speed/libraries#jquery>
- Slowly being replaced by React



# jQuery Basics

- Syntax

- Everything is done through the `$` function, based on `query` selectors
- Example
  - `$(“p”).hide()`
  - `$(document).ready(function() { /* initialization code */});`

- Effectively a wrapper around plain JavaScript
- jQuery objects have different methods/properties

```
// Plain JavaScript
document.querySelector("#title").innerHTML = "<h1>Hello</h1>";

// jQuery
$("#title").html("<h1>Hello</h1>");
```

- Designed to support *chaining*

# Common jQuery Methods

- `val( [value] )`

- Get or set input value

```
username.val()
```

- `attr( k [ , value ] )`

- Get or set attribute with name *k*

```
$( "a" ).attr( "href" );
```

- `css( p [ , value ] )`

- Get or set CSS property with name *p*

```
input.css( "color" , "blue" );
```

- `html( [value] )`

- Get or set arbitrary HTML

- `click( function )`

- Register onclick event

```
$( "input" ).click( function() {  
    $( this ).css( "background-color" ,  
        "lightcyan" );  
});
```

- `parent( ) , children( )`

- Get parent or children

- `next( ) , prev( )`

- Get next or previous sibling

- `addClass( ) , removeClass( )`

- Add or remove class(es)

# Quercus Exercise Q1

- Create a form with the following fields:
  - Username
  - Email
  - Password
  - Repeat password
  - Security question: "What's  $8 + 16/4$ ?"
- Implement client-side validation *using jQuery*, with these checks:
  - Checks if the security question is answered correctly.
  - Checks password and repeat password are the same.
  - Checks if domain name of email address ends with "utoronto.ca".
- Errors should appear when “Sign me up!” is clicked

# Ajax with jQuery

- `$.ajax(url [, settings])`
- Can specify URL, method, etc.
  - All optional
  - <https://api.jquery.com/jquery.ajax/>
- Accepts handler for success or error
- On success
  - data parameter contains JSON result

```
var jqxhr = $.ajax("example.php")
  .done(function(data) {
    alert("success: " + data);
  })
  .fail(function() {
    alert("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();
      }
    }
  });
});
```

# More JavaScript

# Built-in Functions/Methods

- `parseInt(x [, base])`
  - Attempts to convert string to integer
  - Returns NaN on failure
- `isNaN(x)`
  - Checks if value is NaN
  - Note: 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* millisec
- `setInterval(code, time)`
  - Execute *code* every *time* millisec
- `String.trim()`
  - Remove leading/trailing spaces
- `escape(x)`
  - Convert to URL-encoded string
- `unescape(x)`
- Quercus Exercise Q2

# Sessions

- Session-based authentication
  - Browser already stores/sends `cookies` header
- Token-based authentication
  - You are responsible for storing/using the token
  - Use `localStorage` global variable

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

- Set `Authorization` header with the token value
  - In jQuery ajax request settings:

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

# Closures

- Recall functions are *first class citizens* in JavaScript
  - Functions can be defined inside a function and be returned

- Closure

- Nesting of functions where inner function has access to local variables in the outer function(s)

- Questions

- What's stored in **X** and **Y**?
- What's the scope of **a**, **b**, **c**?
- What should happen to **b** at the end of function call?

```
function outer() {  
    var b = 10;  
    var c = 100;  
  
    function inner() {  
        var a = 20;  
        console.log("a = " + a + ", b = " + b);  
        a++;  
        b++;  
    }  
  
    return inner;  
}  
  
var X = outer(); var Y = outer();
```

# Capture

- Inner function *captures* local variable(s) from outer function
- Captured variables can be referenced by inner function
  - Each invocation of outer function creates new copies of outer variables
- Can capture function arguments as well

```
function foo(i) {  
    var x = { count : 0 };  
    return function() {  
        x.count += i;  
        console.log("x.count = " + x.count);  
    };  
}  
  
m = foo(5);  
n = foo(7);  
m(); m(); n();
```

Output:

```
x.count = 5  
x.count = 10  
x.count = 7  
x.count = 14
```

# For Loop and Closures

- Recall that:
  - `var` declares function scope variable
  - `let` declares block scope variable
- In a for loop, `var` and `let` also behaves differently
  - `var` declares a variable once and updates its value
  - `let` redeclares the variable multiple times with different values

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

What's the output of the following code execution?

```
for (fun of outer()) {  
    console.log("i = " + fun());  
}
```

# For Loop and Closures

- Problem
  - var only creates one variable
  - All closures created in the invocation of function references same variable
    - This causes aliasing among different closures
- Solution 1:
  - Force a copy by using immediately invoked function expression

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

Output:

```
i = 1  
i = 2  
i = 3  
i = 4  
i = 5
```

# For Loop and Closures

- Problem
  - var only creates one variable
  - All closures created in the invocation of function references same variable
    - This causes aliasing among different closures
- Solution 2:
  - Use let to declare loop variable, which creates one variable per iteration

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

# Arrow Function

- Similar to lambda function in Python
  - More powerful because it allows for a code block on left side of arrow
- Syntax:
  - `(param1, param2, ...) => expression / body`

- Before:

```
function regular(a, b) { return a + b; }
```

- After:

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

For 1 parameter, can simplify to:

```
const f = x => x + 1;
```

- Simplified:

```
const concise = (a, b) => a + b;
```

# Functional Programming

- Arrow function used often in functional programming paradigm
- JavaScript arrays have higher order functions
- `forEach`
  - Given each element and its index, do something

```
var names = ["ali", "hassan", "mohammad"];
names.forEach(item, index) => console.log(index + ": " + item);
```

- `map`
  - For each element, modify it in some way and return a new array

```
upper = names.map(item => item.toUpperCase());
// ['ALI', 'HASSAN', 'MOHAMMAD']
```

# Higher Order Functions

- filter
  - Returns a new array, keeping elements that satisfies the condition

```
var students = [{name: "Jay", id: 1}, {name: "Ali", id: 2}, {name: "Jay", id: 3}]
let jays = students.filter(item => item.name === "Jay");
// [{name: "Jay", id: 1}, {name: "Jay", id: 3}]
```

- reduce
  - Returns an aggregate value after processing the array
  - Accumulator takes an initial value

```
var names = ["ali", "hassan", "mohammad"];
let longest = names.reduce((acc, cur) => Math.max(cur.length, acc),
                           Number.NEGATIVE_INFINITY);
// 8
```

# Arrow Function and this

- Regular functions can have their own `this` value
  - Refers to the object that called the function (method)
  - Event listeners
    - `this` refers to the element that triggered the event
- Arrow function *does not* have their own `this` value
  - Do **not** use as event listeners or object methods
  - But, arrow function can capture this in a closure (unlike regular functions)

```
const person = {  
    name : 'King Bob',  
    greet() {  
        setTimeout(() => console.log(this.name + " says hi!"), 500);  
    }  
};
```

# Destructuring

- Unpack values from arrays or objects into local variable
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

- Destructuring an object

- Remember the variable names has to be same as property names

```
const hero = { alias : "Batman", name : "Bruce Wayne" };
const {alias, name} = hero;
console.log(name + " is " + alias);
```

- Can place the rest into a sub-object

```
const {alias, ...rest} = hero;
// rest is { name : 'Bruce Wayne' }
```

- Destructuring an array

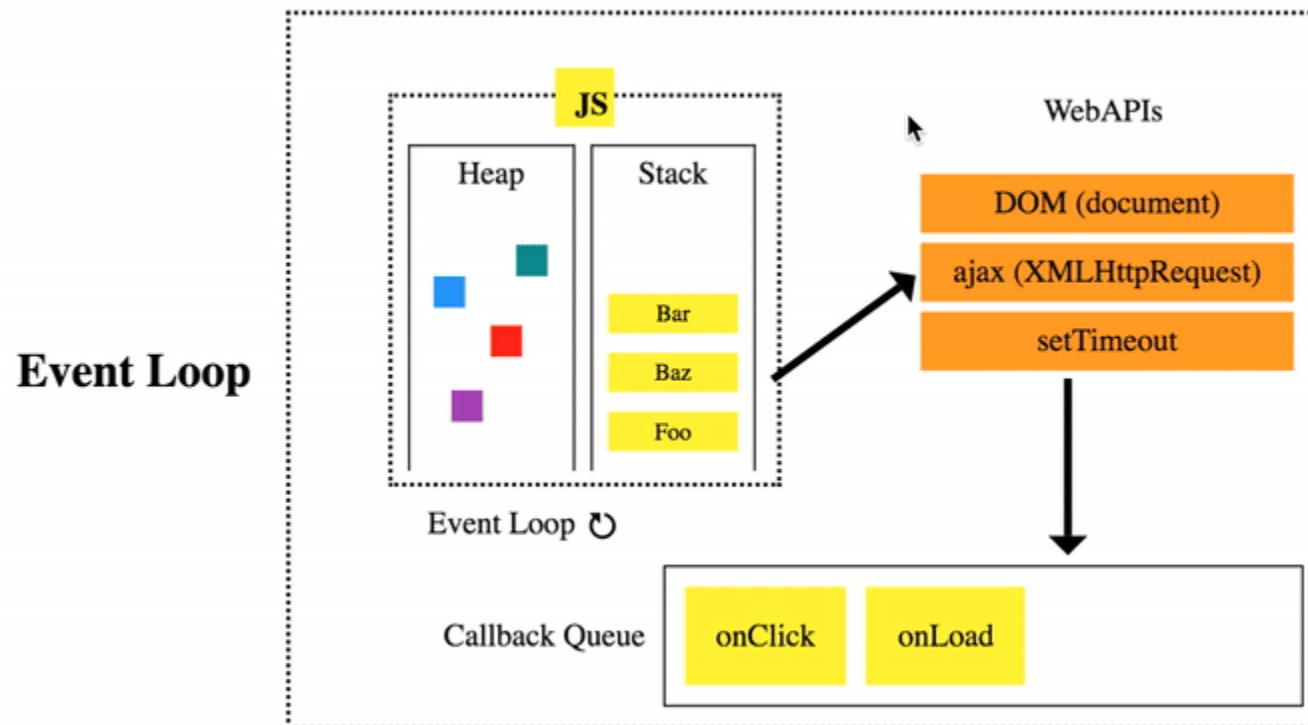
```
let a, b;
[a, b] = [3, 7]; // a = 3 and b = 7
```

# Event Loop and Promises

# Event Loop

- JavaScript supports event-driven paradigm
- JavaScript code is run in a single thread
  - Scripts are executed at load time; the rest are all *events*.
  - Reduces overhead of managing threads and shared variables
- Event loop provides the illusion of multiple threads
- Events are pushed to the event queue
  - Example: ready, click, ajax, setTimeout
- Event loop constantly checks for new events and execute their callback
  - This happens synchronously

# Visualization of Event Loop



<https://medium.com/@Rahulx1/understanding-event-loop-call-stack-event-job-queue-in-javascript-63dcd2c71ecd>

# Callback Hell

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function (err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Code written in a way where execution happens from top to bottom.

Don't write JavaScript like this

# Promises

- Problem

- Callbacks can make code hard to understand due to **nesting**
  - Example: jQuery ajax has at least two callbacks for success and error

- Promise

- An alternative to using callbacks

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

- Code inside of promise is executed immediately
- Calling **resolve** or **reject** pushes events to event queue (asynchronous)
- Can later handled by the methods **then** or **catch**

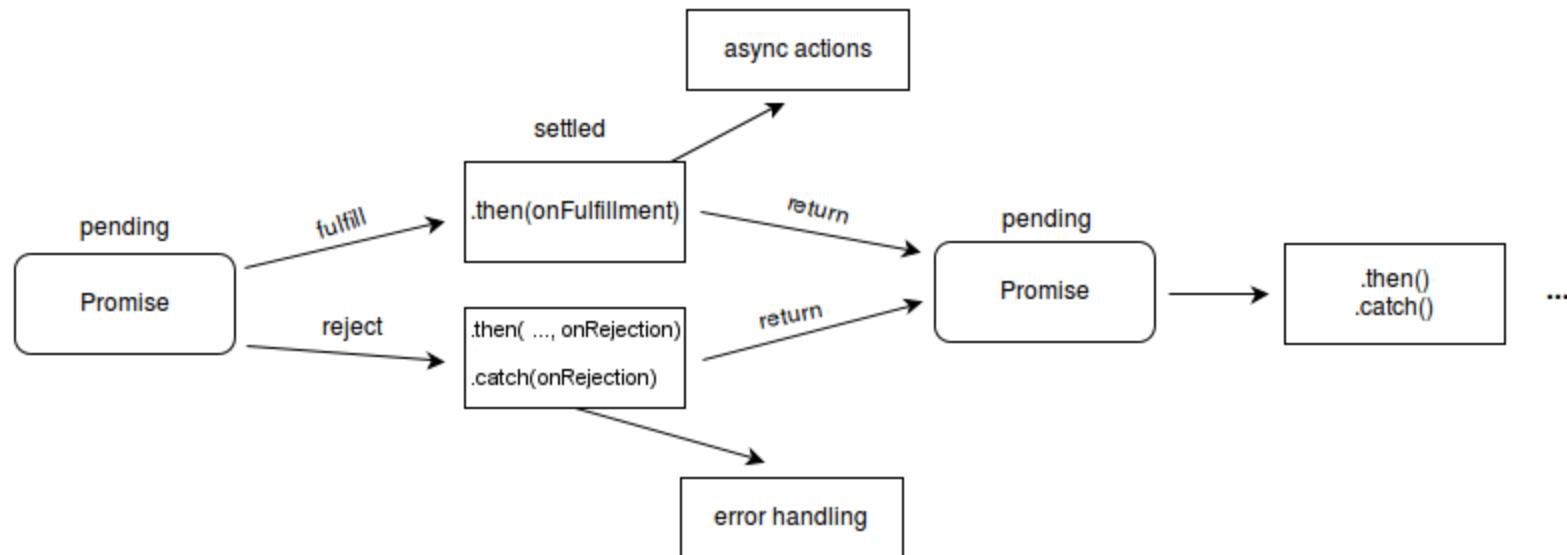
# Fetch API

- Returns a Promise object

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

- Callback is specified in the then method(s) 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

# Promise State Transition



[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

# Example Promise

- Delayed division

```
const slow_divide = (a, b) => new Promise((resolve, reject) => {
    if (b != 0) {
        setTimeout(() => resolve(a / b), 1000);
    }
    else {
        reject("Error: attempting division by zero");
    }
});

slow_divide(6, 2)
    .then((res) => {
        console.log("result is " + res);
    })
    .catch((msg) => {
        console.log(msg);
    });
}
```

Both resolve and reject takes only *one* argument. If you need to pass more than one values, use an object.

# Chaining Promises

- then/catch will get called even if promise is already settled
- Multiple callbacks can be added by calling then several times
- What's the output?

```
const add = (num1, num2) => new Promise((resolve) => resolve(num1 + num2));  
  
add(2, 4)  
  .then(result) => {  
    console.log(result); return result + 10; ← Return value wrapped in Promise  
  })  
  .then(result) => {  
    console.log(result); return add(result, 2); ← Explicitly returning a Promise  
  })  
  .then(result) => {  
    console.log(result);  
  })
```

# Promise vs. Callback

- Promise

- Still a bit of callback hell

```
slow_divide(36, 2)
  .then((res) => {
    console.log("result is " + res);
    return slow_divide(res, 3);
  })
  .then((res) => {
    console.log("result is " + res);
    return slow_divide(res, 6);
  })
  .then((res) => {
    console.log("result is " + res);
  })
  .catch((msg) => console.log(msg));
```

- Callback

- Code difficult to read and maintain

```
function slow_divide(a, b, c, d) {
  const TIMEOUT = 1000;
  setTimeout(() => {
    let res = a / b;
    console.log("result is " + res);
    setTimeout(() => {
      res = res / c;
      console.log("result is " + res);
      setTimeout(() => {
        res = res / d;
        console.log("result is " + res);
      }, TIMEOUT);
    }, TIMEOUT);
  }, TIMEOUT);
}

slow_divide(36, 2, 3, 6);
```

# async and await

- Promises still breaks program logic
- `async` function
  - `await` operator: waits for a Promise to be fulfilled before continuing code
  - Error handling naturally done through try/catch

```
async function divide_thrice(a, b, c, d) {  
    try {  
        let res = await slow_divide(a, b);  
        console.log("result is " + res);  
        res = await slow_divide(res, c);  
        console.log("result is " + res);  
        res = await slow_divide(res, d);  
        console.log("result is " + res);  
    } catch(err) {  
        console.log(err);  
    }  
}
```

Simplifies code that consumes result from Promise objects.

No more callback hell.

Note: `await` can only be used inside `async` functions.

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 10: Introduction to React

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# Review

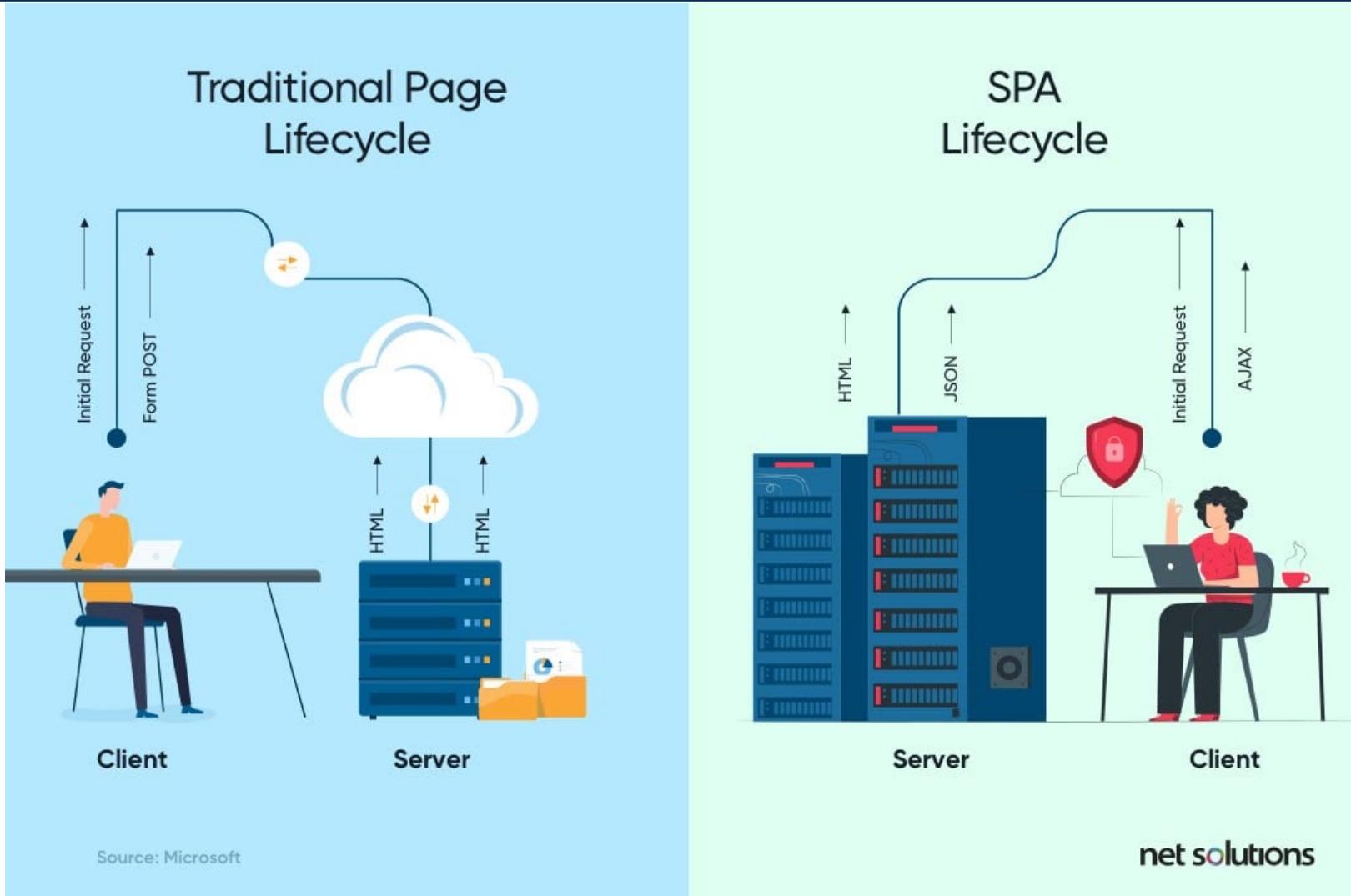
---

- Server-side rendering
  1. Backend server listens for request
  2. Upon entering a URL, browser sends request to server
  3. Server returns an HTML page in response
    - Many contain links to other static files
      - E.g., js, css, image, etc
  4. Separate requests are sent for static files
  5. Browser renders HTML and CSS, runs scripts
- Each link or form submission yield a new web page
  - Requires a full reload. May degrade user experience (UX)
- Solution?

# Single Page Application

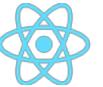
- Single page application (SPA)
  - Executed in the browser's built-in JavaScript engine
  - Only requires one hard URL reload
  - Subsequent request/rendering can be done through [Ajax](#) in background
- Benefits
  - Seamless user experience
    - Performing an action does not reset the page
  - Efficiency
    - Only relevant parts of page are updated, not entire page
  - Improves load time
    - Initial load (when nothing is there) takes less time

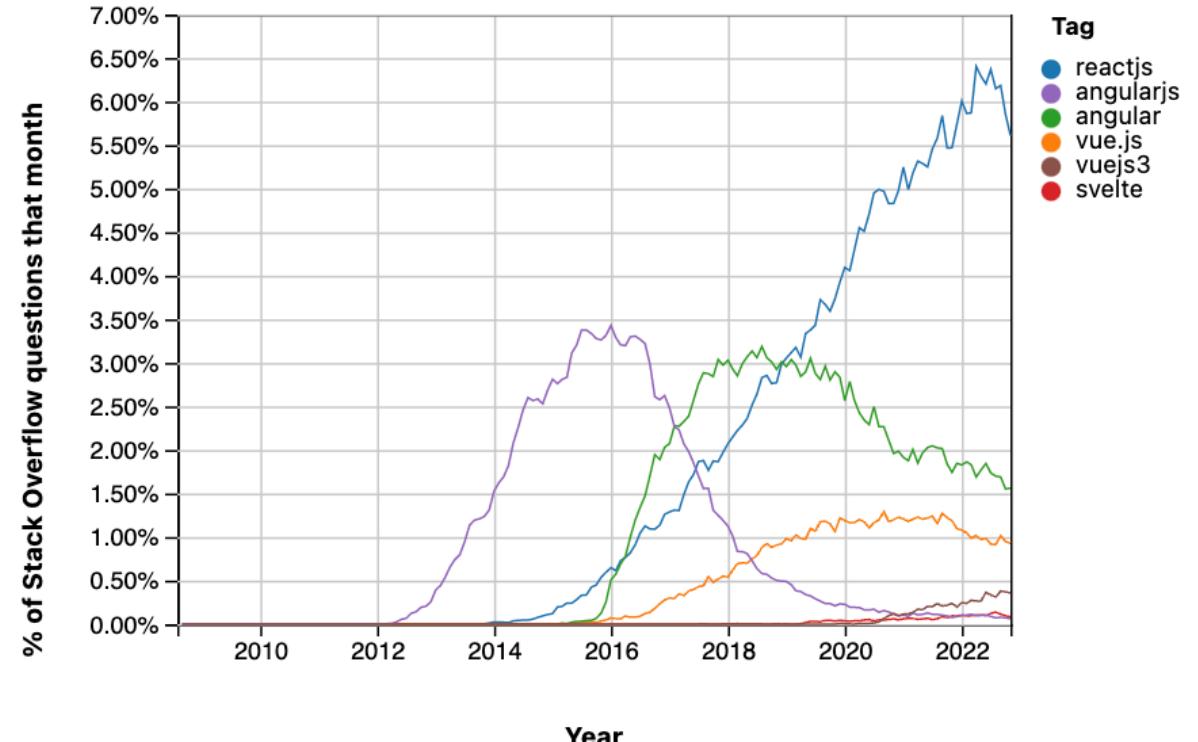
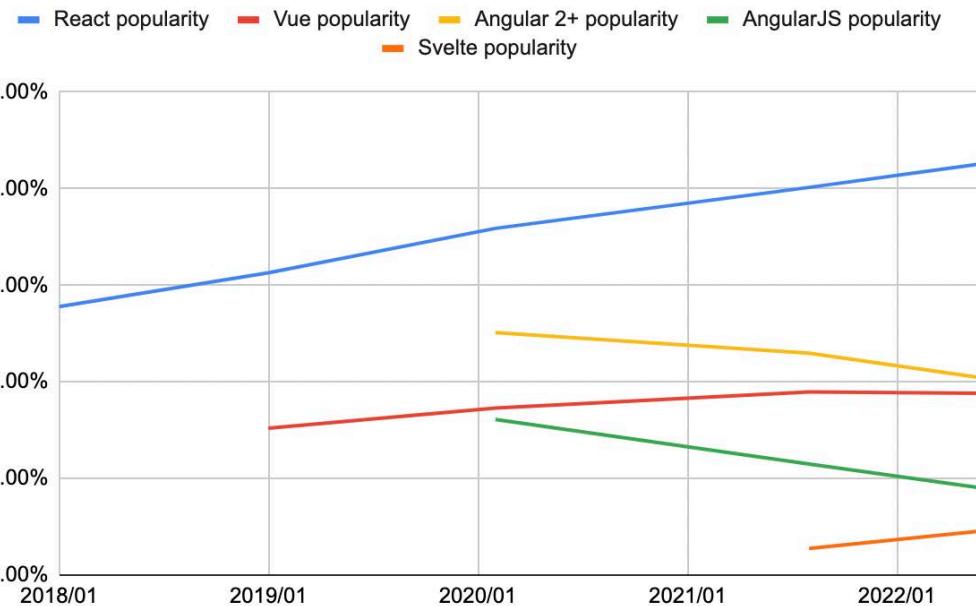
# Page Lifecycle



<https://www.netsolutions.com/insights/single-page-application/>

# Building SPA

- Nobody build SPA with Ajax alone
- Frontend frameworks
  - React  , Angular  , Vue 



<https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190>

# React JS

- Released by Facebook in 2013
- A JS library for building interactive user interfaces
- React takes charge of re-rendering when *something* changes
  - You no longer need to manipulate elements manually
- Virtual DOM
  - A representation of UI kept in memory and synced with real DOM
    - Handled by a library named ReactDOM
  - When something changes, it compares new and old DOMs
    - Finds what has been updated
    - Updates only those elements in the browser's DOM
      - Because updating re-rendering real DOM is expensive!

# JSX

- React uses a special variation of JavaScript
- JSX
  - Short for JavaScript XML
  - Merges HTML and JavaScript into one language
  - Example:

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

- Browser *does not* understand JSX natively
  - Requires translation before execution
- Babel JS
  - A JavaScript compiler. Can translate JSX code into pure JS code

# Translation

- JSX

```
const element = <span className="red">  
    Hello World!  
</span>;  
  
const name = "Joe";  
const id = "div-1";  
  
const element2 = (  
    <p>  
        <div id={id}>  
            Hi, {name}!  
        </div>  
    </p>  
);
```

These are React  
elements, not  
real JS elements

- JavaScript

```
const element = /*#__PURE__*/  
    React.createElement("span", {  
        className: "red"  
    }, "Hello World!");  
  
const name = "Joe";  
const id = "div-1";  
  
const element2 = /*#__PURE__*/  
    React.createElement("p", null,  
        /*#__PURE__*/  
        React.createElement("div", {  
            id: id  
        }, "Hi, ", name, "!"));
```

# Make it Real

- Import React and Babel (JSX) libraries into your HTML

```
<script src="https://unpkg.com/react@18.2.0/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18.2.0/umd/react-dom.production.min.js"></script>
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

- Render your element inside an actual JS element

```
<script type="text/babel">
  const element = <h1>Hello World!</h1>;
  const root = ReactDOM.createRoot(document.body);
  root.render(element);
</script>
```

- React roots are used to render React elements into the real DOM

# React Components

- React components
  - Functions that return a JSX element, or
  - Classes that extend `React.Component` and implement the `render` method
- Key concept in React
- Allows you to make your elements reusable
  - Components can be reused like an HTML tag

```
<main id="root"></main>
<script type="text/babel">
function Hello() { return <h1>Hello World</h1>; }
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Hello />);
</script>
```

Rendering  
React elements  
inside `<main>`

Reusing Hello  
component

# Components Basics

- Void tags must always end with />, e.g., <Hello />
- Component name must be capitalized
  - To distinguish from built-in HTML elements (always lowercase)
- A JSX element must be wrapped in one enclosing tag
- React fragment
  - A workaround for returning multiple elements

```
function Items() {  
  return <>  
    <li>Hello</li>  
    <li>World</li>  
  </>;  
}
```

# Components and Props

- You can put any JS expression inside curly braces in JSX

```
const id = "content";
<div id={id}>...</div>
```

## • Props

- Read-only arguments passed into React components via a dictionary

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

- You 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
  - Important: styles must be placed inside a dictionary

```
function Text(props) {  
    const {value, size} = props;  
    return <p className="text" style={{fontSize: size}}>{value}</p>;  
}
```

- Tips
  - Use destructuring to simplify components with many props
  - you *do not* need to add quotations marks around attribute values
    - Compiler does that for you automatically

```
root.render(<Text size={30} value="Hello world" />);
```

# Loop-Generated Elements

- Elements created in a loop must have a unique `key` prop.
- `key` prop
  - 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 real DOM

```
function List({title, values}) {  
  return <>  
    <Text value={title} size="2em" />  
    <ul>  
      {values.map((item, index) => (  
        <li key={index}>  
          {item}  
        </li>  
      ))}  
    </ul>  
  </>;  
}
```

Destructuring in  
function parameter

`key` can just be  
the loop index

# Paired Tag

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

```
function Box({children}) {  
  return <div className="box">{ children }</div>;  
}
```

- Example

```
const mybox = (  
  <Box>  
    <List title="Cats" values={[ "Felix", "Oscar", "Fluffy", "Whiskers" ]} />  
  </Box>  
);  
  
root.render(mybox);
```

# Class Components

- Another way to define a component
  - Extends `React.Component` base class
  - Implements the `render` method
  - Can have *states*
    - In contrast, functional components are “stateless” components
- Props are passed to constructor. Can access through `this.props`
  - The super class constructor handles the above already
- Example

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

# Component State

- Class components have a built-in state
  - 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

```
class Counter extends React.Component {  
  constructor(props){  
    super(props);  
    this.state = { counter: 0, };  
  }  
  
  render(){  
    return <h3>{this.state.counter}</h3>;  
  }  
}
```

# Updating State

- React states should never be mutated directly
  - Except inside the constructor
  - The two approaches below will *not* trigger re-rendering

```
// wrong way 1
this.state.counter += 1;
```

```
// wrong way 2
this.state = { counter: this.state.counter + 1 };
```

- `setState` method
  - Updates the state AND triggers re-rendering

```
// correct way to update state
this.setState({ counter: this.state.counter + 1 });
```

# Events

- React has the same set of events as vanilla JavaScript
- Syntax differences
  1. React events are written in **camelCase**
    - E.g., `onClick` instead of `onclick`
  2. The actions must be a function, not just an expression
    - E.g. `onClick={() => alert()}` instead of `onclick="alert()"`
- Can define event handler with component method

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

// in render method
<button onClick={this.increment}> Click me </button>
```
- But..., this doesn't work. Why?

# 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 special bind method. Enables early binding.
  - Very ugly and unrelated to application logic. Do not use.

```
constructor() {  
    this.onClick = this.onClick.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`

- The element that triggered the event

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

- Tip for building complex components

- Lifting the state up: <https://reactjs.org/docs/lifting-state-up.html>

1. Pass shared states between subcomponents through their common ancestor
2. Initial value can be passed as `props` to subcomponents
3. Pass a `setter` function to subcomponents as change handler

- Quercus Exercise Q1

- Build a two-way Celsius to Fahrenheit converter

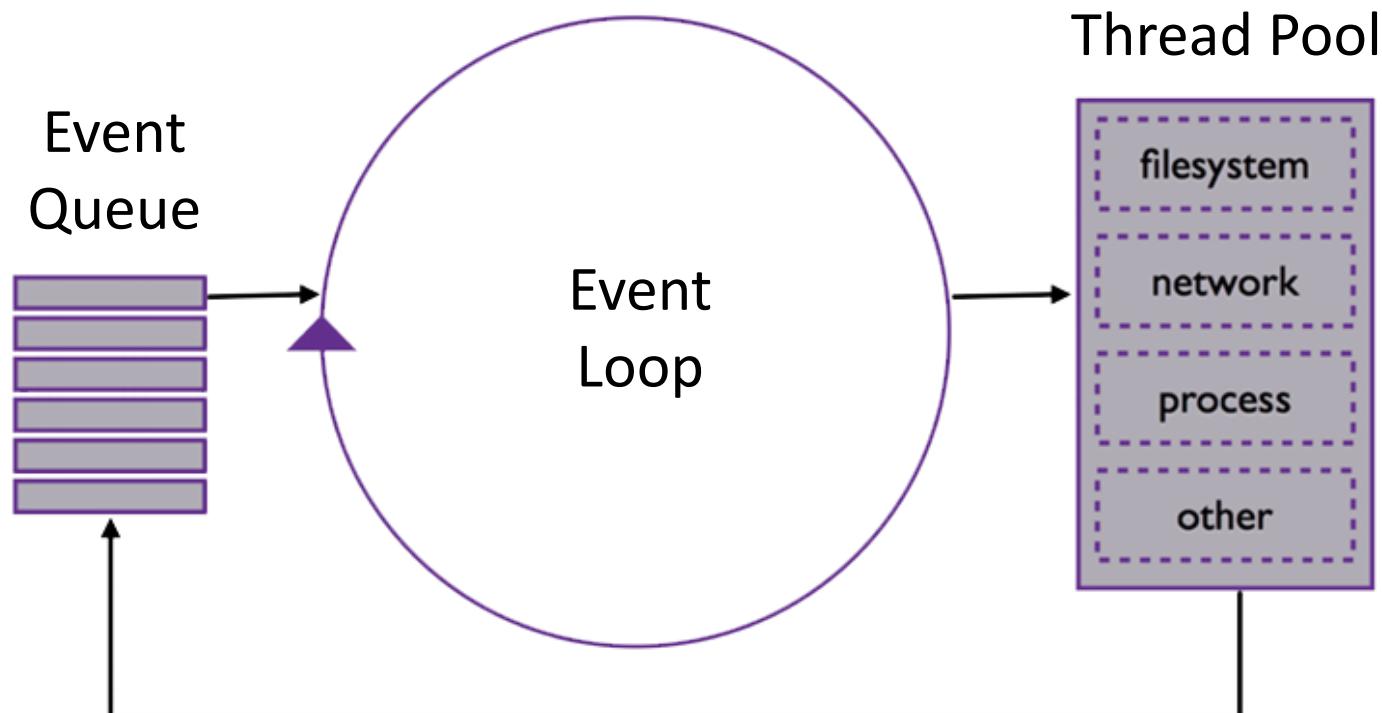
# React Project

# React

- Enabled by importing scripts to our HTML file
- JSX code are translated to JavaScript *every time* page is loaded
  - Very slow!
- Alternative: [React project](#)
  - Neither a backend nor HTML project
  - [Frontend](#) server that returns appropriate files per request
  - A precompiled and bundled build for production
- [Node.js](#): a runtime environment for running JavaScript on *server-side*
  - Installation: <https://nodejs.org/en/download/>
  - Includes a package manager, interactive console, build tools, etc.

# Node.js Processing Model

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



[https://www.youtube.com/watch?v=zphcsoSJMvM&ab\\_channel=node.js](https://www.youtube.com/watch?v=zphcsoSJMvM&ab_channel=node.js)

# Node Console

- Can be opened with the `node` command
- Allows you to execute inline JavaScript code
- There is no `window` or `document` global object
  - We are no longer inside a browser
- Can execute scripts as well
  - `node <filename>`
- Console start up message:

```
~$ node
Welcome to Node.js v18.14.2.
Type ".help" for more information.
>
```

# Installing Modules

- Node Package Manager (npm)
  - Extremely similar to Python pip
- Install packages via `npm install <package_name>`
  - Packages are stored in the `node_modules` directory
    - Similar to `venv` directory in a virtual environment
- Automatically generates and maintains a file named `package.json`
  - Similar to the `requirement.txt` file for tracking dependencies
- Node Package eXecute (npx)
  - Allows executing JS packages without having installed them
  - Will download all necessary packages to execute the command

# 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
```

- Project contains same code but more organized

- Important files:

- public/index.html

- Contains base HTML code
- Note a div with id="root"
  - DOM rendered inside it

- src/index.js

- Invokes ReactDOM.createRoot
- By default, renders <App />

- src/App.js

- Placeholder App component

# Exports

- In JavaScript, 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
  - Syntax 1: `const var1 = 3, var2 = (x) => x + 1;`  
`export { var1, var2 };`
  - Syntax 2: `export const var1 = 3, var2 = (x) => x + 1;`
- **import** statement:

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

# Default Export

- Each module can have one default export
  - Usually, it is the component defined within the module

```
export default App;
```

- Importing the default export

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

- Importing default export *does not* require matching name
  - Can be imported under any arbitrary name

```
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
- Images, 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 into the HTML
  - All static file imports, including js and css, are handled automatically by server

# Organizing Components

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

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

- Import other components like this:

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

Rule of thumb:

Components should be  
*small*, e.g., < 100 loc.

Large components should  
be split into small, nested  
child components

# Final notes

---

- Quercus Exercise Q2
  - Redo Q1
  - Refactor the temperature converter into a React project
- Read React tutorials
  - <https://reactjs.org/docs/hello-world.html>
  - <https://reactjs.org/tutorial/tutorial.html>
- Important announcement
  - Class cancellation notice
  - Classes on March 22, March 24, and March 27 are *cancelled*
  - Please spend the extra time on timely completion of A3 and P3

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 11: React Hooks, Context, and Router

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# Hooks

# History

- Functional components used to be “dumb”
  - Used for *presentation* only; cannot track internal state
  - Does not have access to lifecycle methods (more on this later)
- Class components are difficult to work with
  - Verbose syntax
  - Hard to reuse and share component logic
- Hooks
  - Introduced in React 16.8 (2019)
  - Make functional components much more versatile
  - Now the de-facto way for write clear and concise components

# Hooks

- A set of functions that you 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`
  - Component re-rendered when `update` is called to change the variable

```
import React, {useState} from 'react';
const Status = (props) => {
  const [status, setStatus] = useState("good");
  const toggleStatus = () => setStatus(status === "good" ? "bad" : "good");
  return <>
    <h3>Situation is {status}</h3>
    <button onClick={toggleStatus}>Toggle</button>
  </>;
};
```

# Using Hooks

- Rules of hooks
  1. Only call hooks at the top level
    - Required to ensure deterministic call ordering
  2. Only call hooks from React functions
- Further reading
  - <https://medium.com/@ryardley/react-hooks-not-magic-just-arrays-cd4f1857236e>
- Benefits
  - Supports multiple state variables
  - Easy to share state(s) with child elements
  - Easier to use compared to class components
- Quercus Exercise Q1

# Lifecycle

- So far, we only run code when `render` is called
  - For both class and functional components
- However, we don't want to run expensive operation on every re-render
  - E.g., sending an ajax request only when component is first loaded
- Lifecycle methods
  - Executes when something happens to a component
  - Class components
    - `componentWillMount()`: before loading a component
    - `componentDidMount()`: after loading a component
    - `componentDidUpdate()`: after updating a component (except initial load)
    - `componentWillUnmount()`: before unloading a component

# useEffect

- Replaces lifecycle methods

```
import React, {useState, useEffect} from 'react';
```

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

```
useEffect(() => {
  console.log("This is called when component mounts");
}, []);
```

- Subscription

```
useEffect(() => {
  console.log("props size or status has changed");
}, [status, props.length]);
```

Tip: Should have one useEffect per concern

# Function vs. Class Component

```
function ShowCount(props) {  
  const [count, setCount] = useState();  
  
  useEffect(() => {  
    setCount(props.count);  
  }, [props.count]);  
  
  return <div>  
    <h1>Count : {count}</h1>  
  </div>;  
}
```

Function components  
is much more concise  
and readable.

```
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>;  
  }  
}
```

# useEffect Notes

- If dependency is missing, effect would run at every re-render
  - Typically, this is not what you want, except...

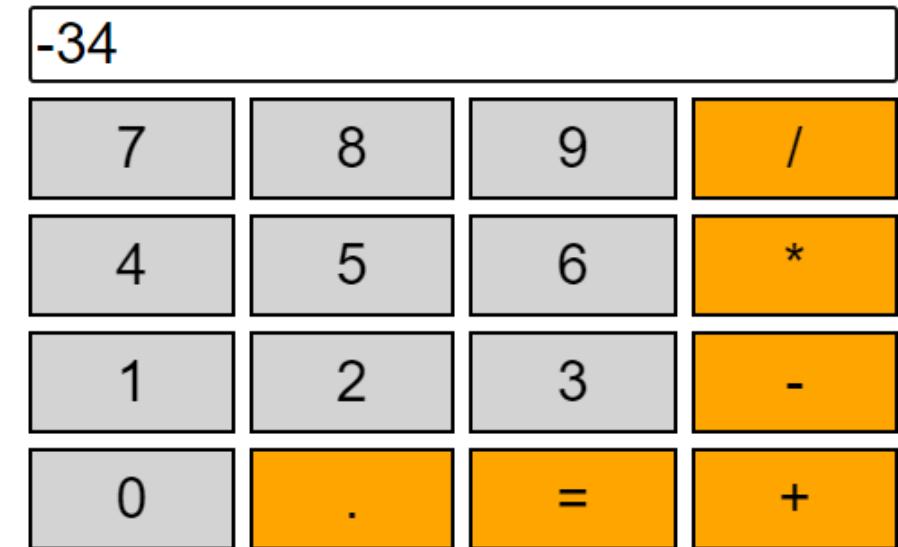
```
function Counter() {  
  const [count, setCount] = useState(0);  
  
  useEffect(() => {  
    document.title = `You clicked ${count} times`;  
  });  
  
  return <button onClick={() => setCount(count + 1)}>+1</button>;  
}
```

- Dependency array should include all variables used in the effect
  - Otherwise it might use **stale** values at re-render
    - React sometimes caches values for optimization

# Quercus Exercises

- Question 2

- Build a simple calculator
- When a button is clicked, update the display, until = is clicked.
- Tip: use the eval() built-in function to evaluate an arbitrary JavaScript expression



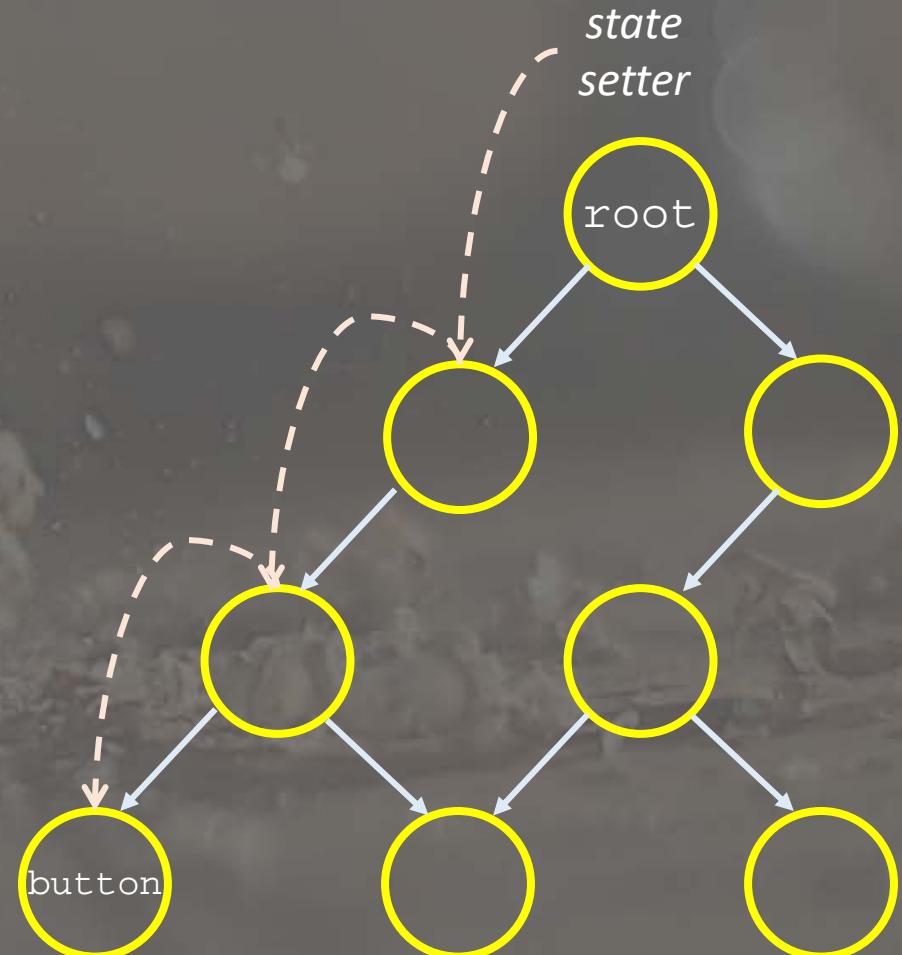
- Question 3

- Generate a table of baseball players
  - Using Fetch API
  - <https://www.balldontlie.io/api/v1/players>
- Hint: do this on load and not on re-render!
- Add autocomplete search feature and pagination

# Global State

# Prop Drilling

- Passing state(s) down to descendants components can be cumbersome
- Example:
  - The subcomponent that fires the request is a deeply nested button
  - You need to pass both the state and its setter function *all the way* down to the button
- Solution?



# Global State

---

- A global state can be a great alternative
  - Accessible everywhere
  - No need to pass states all the way down
- Like global variables, don't use them for everything!
  - Makes your code dirty and harder to understand
  - Makes component harder 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

- Convention
  - Create a `contexts` folder under `src` and put all context files inside
- `createContext`
  - Creates a context that can be later used

```
import { createContext } from "react";

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

- Advice
  - Put default initial values for every variable that you 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);
```

# Benefits

---

- Context enables you to handle API data easily
- Many components need to access them
  - E.g., username, profile data, etc.
- Various components can call APIs to fetch data
- Advice
  - For each Django app, create a `context` in React
  - Then, write a function that sets up relevant values and their setters
    - Name of this function should start with “use”
- Further reading
  - <https://dmitripavlutin.com/react-context-and-usecontext/>

# Context Example

- “use” function

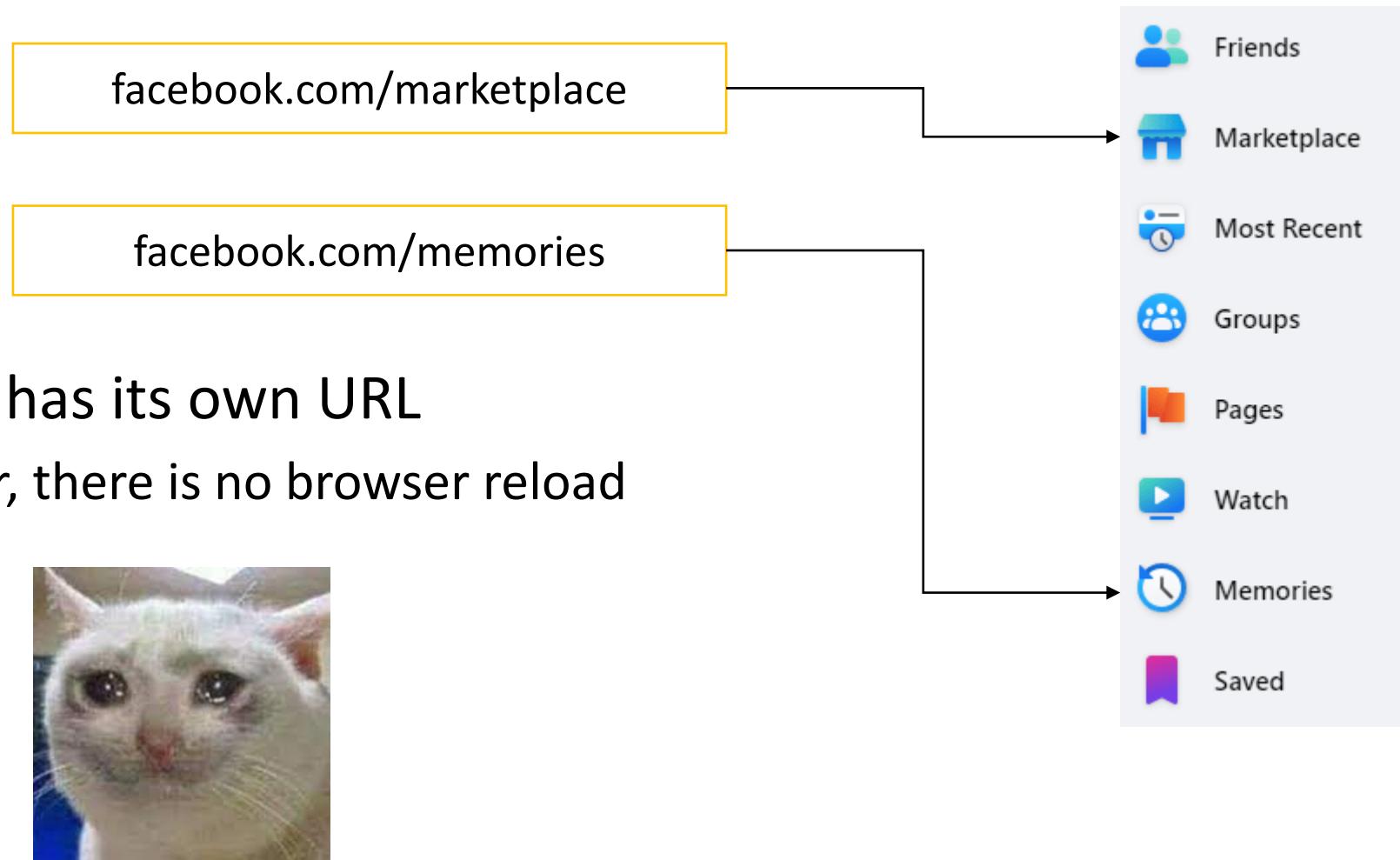
```
export function useAPIContext() {  
  const [deployment, setDeployment] = useState([]);  
  const [servers, setServers] = useState([]);  
  const [applications, setApplications] = useState([]);  
  const [applicationStatus, setApplicationStatus] = useState([]);  
  const [availableLogDates, setAvailableLogDates] = useState([]);  
  
  return {  
    deployment, setDeployment,  
    servers, setServers,  
    applications, setApplications,  
    applicationStatus, setApplicationStatus,  
    availableLogDates, setAvailableLogDates,  
  };  
}
```

Inside the Provider

```
<APIContext.Provider  
  value={useAPIContext()}>  
  <ControlPanel />  
</APIContext.Provider>
```

# Multi-Page React App

# Pages



# Naïve Approach

- Using what we know so far...

Even with this solution, we still do not have individual URLs for each component, making it difficult to return to a specific page quickly.

## Wants

- Access to specific component via an URL
- Changing URL does not result in browser reload

```
function Facebook() {  
    const [page, setPage] = useState("");  
    const Navbar = () => <nav>  
        <a onClick={() => setPage("watch")}>Watch</a>  
        <a onClick={() => setPage("groups")}>Groups</a>  
        <a onClick={() => setPage("marketplace")}>Marketplace</a>  
    </nav>;  
  
    const Page = () => {  
        switch(page) {  
            case "watch":  
                return <Watch />;  
            case "groups":  
                return <Groups />;  
            case "marketplace":  
                return <Marketplace />;  
            default:  
                return <Feed />;  
        }  
    }  
  
    return <Navbar><Page /></Navbar>;  
}
```

# Router

- Installation

- Run this in your project directory

```
npm install react-router-dom
```

- Convention

- Create a pages folder inside src
- Put each page's component in a separate file or directory (preferred)

- Further reading

- <https://reactrouter.com/en/6.9.0/start/overview>
- [https://www.w3schools.com/react/react\\_router.asp](https://www.w3schools.com/react/react_router.asp)

*Example organization*

- src
  - pages
    - Groups
      - index.jsx
    - Marketplace
      - index.jsx
    - Watch
      - index.jsx

# Routes and Links

- Set up the **routes** in App.js
  - Same idea as 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>;
}
```



A yellow rectangular box with a thin black border is positioned to the right of the code. Inside the box, the text "Root path" is written in a black sans-serif font. A thin black arrow points from the top-left corner of the box towards the first `<Route path="/">` line in the code.

# Link

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

```
import { Link, useParams } from "react-router-dom";  
  
<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

```
import { useSearchParams } from "react-router-dom";  
  
// By convention, underscore in front of a name means "don't care".  
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, you need a URL change via code
- Example
  - When Response is 401, redirect to the login page
- Vanilla JavaScript
  - This causes the browser to reload!

```
window.location.replace("/marketplace");
```

- React Router

```
import { useNavigate } from "react-router-dom";  
  
let navigate = useNavigate();  
navigate("/marketplace");
```

# Outlet

- We need a **navbar** to navigate through pages
  - Bad idea to copy it to all the pages
- What happens 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 component that *partially matches* the URL and has an element
- However, it continues matching the remaining URL and returns the **matching child** components as <Outlet />
- Convention
  - Root element is used to specify layout; child components are rendered within.

# Using Outlet

- In App.js

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

```

- In Layout.jsx

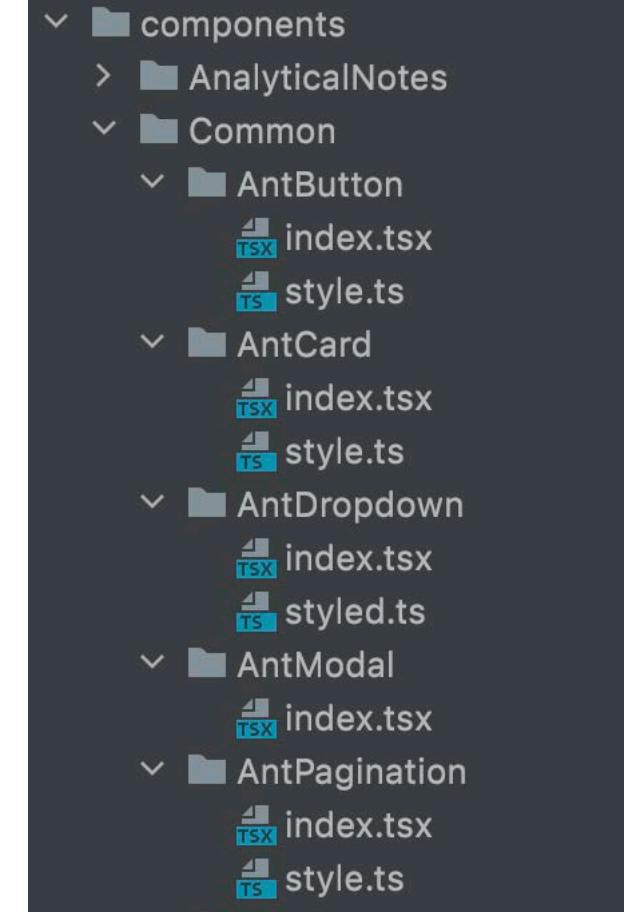
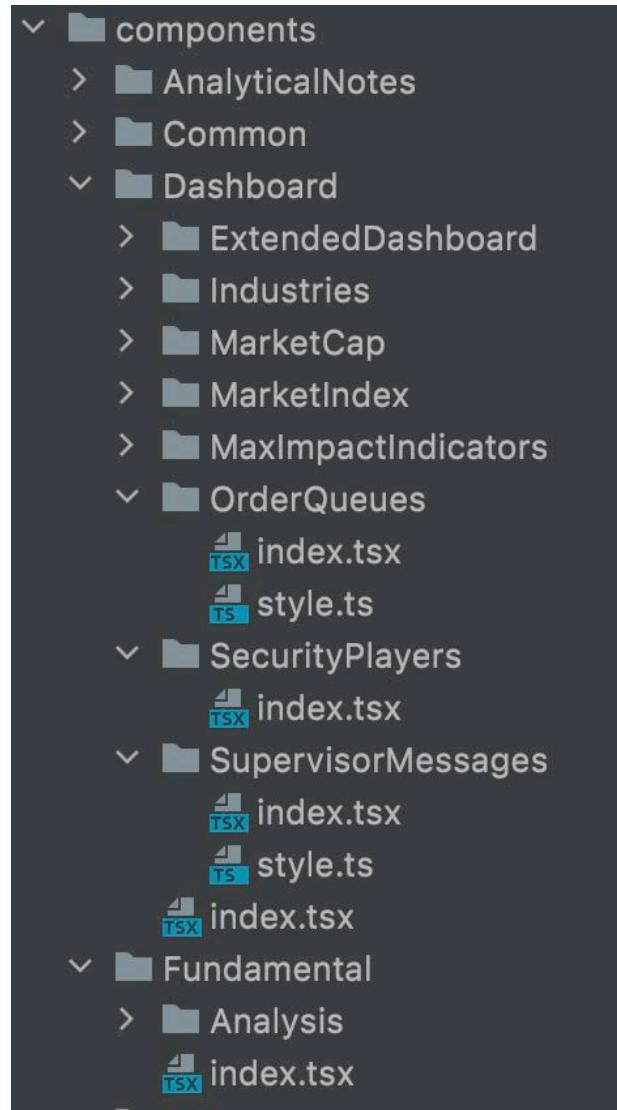
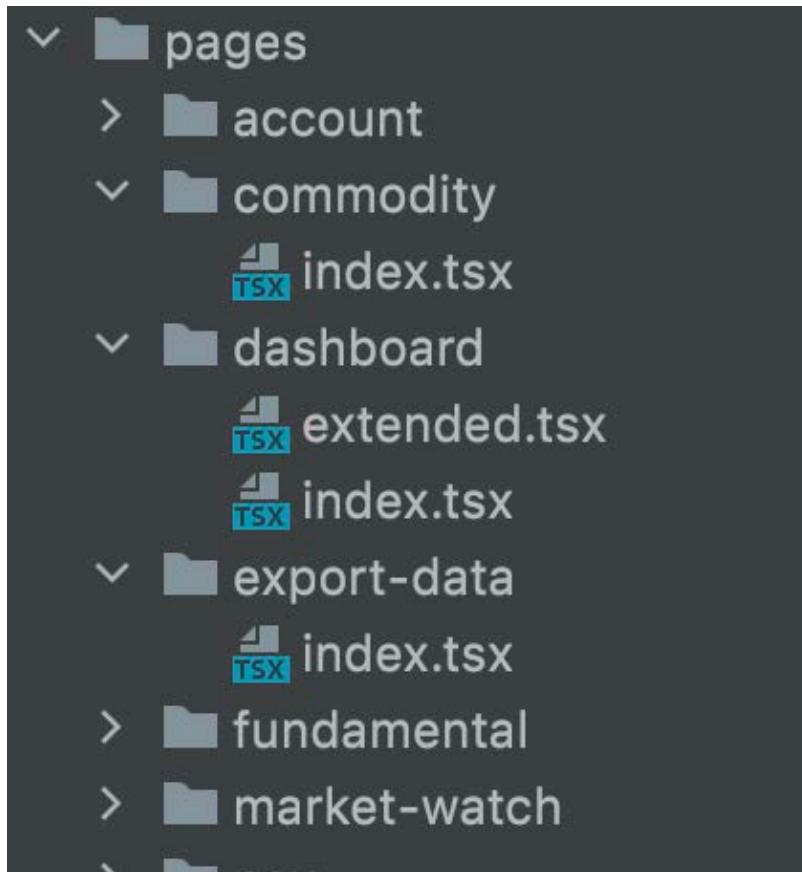
```
const Layout = () => {
  return <>
    <header>
      <Link to="/watch">Watch</Link>
      <Link to="/groups/88/?name=joe">Groups</Link>
      <Link to="/marketplace">Marketplace</Link>
    </header>
    <Outlet />
  </>;
}
```

Child components will be rendered where `<Outlet />` is.

# Term Project

- File structures for React project varies
- Good practice to separate pages from reusable components
  - E.g., inputs, tables, forms, buttons, etc.
- Do not let a component become too big (in LOC)
  - Refactor by extracting child components
- Expect most components to have multiple children
  - Thus, each component/page should have its own directory, not just file
  - You can put child components in a subfolder of the parent component
- Dedicate a page to login, signup, forms, and navbar items
- Use function components and hooks instead of class components

# Example File Structure



# Final notes

---

- Important announcement
  - Class cancellation notice
  - Classes on March 22, March 24, and March 27 are *cancelled*
  - Please spend the extra time on timely completion of A3 and P3
- Exercise 9 + 10
  - Due next Sunday, Apr 2nd
- Midterm Results
  - Should be released on Wednesday
    - Some TAs are not done yet

# CSC309H1S

## Programming on the Web

Winter 2023

### Lecture 12: Web Deployment

Instructor: Kuei (Jack) Sun

Department of Computer Science  
University of Toronto

# Introduction

---

- So far
  - Frontend development
    - HTML/CSS
    - Client-side scripting with JavaScript
    - React
  - Backend development
    - Django
  - Everything currently running on the *local computer!*
- This lecture
  - Deployment of a web application
  - DevOps

# Development vs. Production

Development	Production
Uses lightweight database, e.g. SQLite	Uses real database, e.g. MySQL, MongoDB
Runs a development server	Runs a real webserver, e.g., Nginx
Hosts on local machine	Hosts on public machine or cloud platform
Hosts on a local IP address	Hosts on a static IP address
Does not have a domain name	Has a 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 not handle high traffic	Can handle high traffic

# IP Address

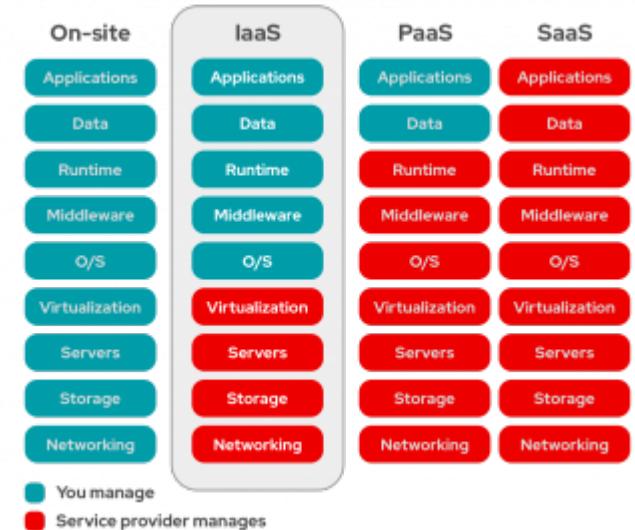
- Most IPv4 addresses have almost been used up
- Transition to IPv6 has been slow (~34% globally in 2022)
- Static IP address
  - Fixed IP address for the machine
  - Does not change over time, even if you power off your machine
- Dynamic IP address
  - IP address assigned by DHCP server
  - Can change the next time you connect to the Internet
- Production server typically uses static IP address(es)

# Domain Name

- Your website likely needs 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
  - Example
    - GoDaddy
    - NameCheap
- You will need to buy a domain name from a registrar
  - Price varies depending on popularity and top-level domain
    - E.g., .com is the most popular top-level domain

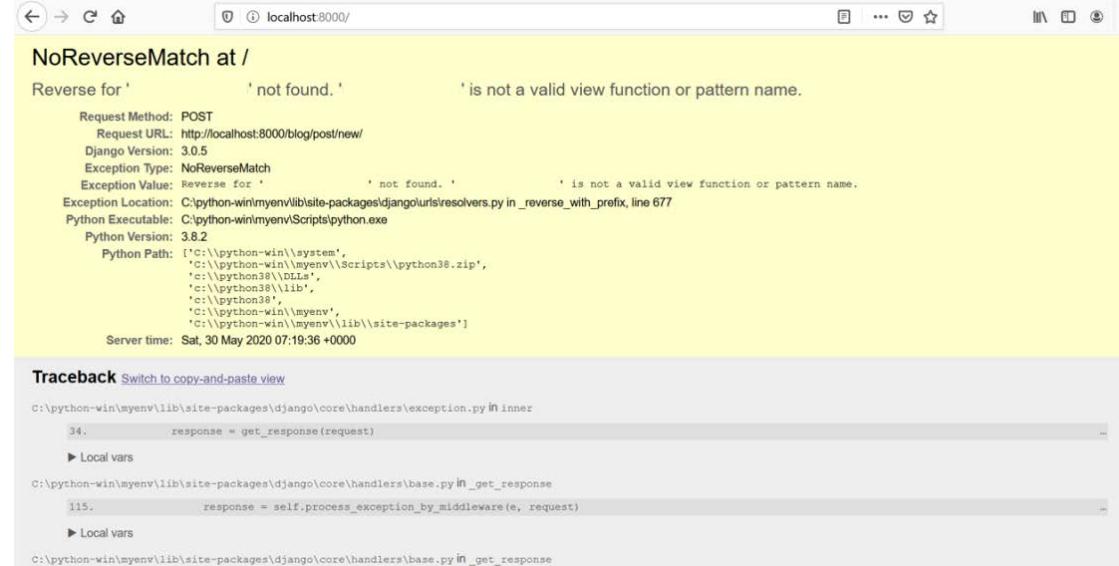
# Web Hosting

- Various options available
- Dedicated Hosting
  - Entire physical server to your 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)



# Error Handling

- During development, Django provides nice diagnostic messages



The screenshot shows a browser window with the URL `localhost:8000/`. The page displays an error message for a `NoReverseMatch` exception. The error details include:

- Request Method: POST
- Request URL: `http://localhost:8000/blog/post/new/`
- Django Version: 3.0.5
- Exception Type: `NoReverseMatch`
- Exception Value: Reverse for '...' not found. '...' is not a valid view function or pattern name.
- Exception Location: `C:\python-win\myenv\lib\site-packages\django\urls\resolvers.py` in `_reverse_with_prefix`, line 677
- Python Executable: `C:\python-win\myenv\Scripts\python.exe`
- Python Version: 3.8.2
- Python Path: `'C:\python-win\system', 'C:\python-win\myenv\Scripts\python38.zip', 'C:\python38\DLLs', 'C:\python38\lib', 'C:\python38', 'C:\python-win\myenv', 'C:\python-win\myenv\lib\site-packages'`
- Server time: Sat, 30 May 2020 07:19:36 +0000

Below the error message, there is a **Traceback** section with a "Switch to copy-and-paste view" link. The traceback shows the call stack from `exception.py` to `base.py`:

```
C:\python-win\myenv\lib\site-packages\django\core\handlers\exception.py in inner
    34.         response = get_response(request)
    ▶ Local vars
C:\python-win\myenv\lib\site-packages\django\core\handlers\base.py in _get_response
    115.             response = self.process_exception_by_middleware(e, request)
    ▶ Local vars
C:\python-win\myenv\lib\site-packages\django\core\handlers\base.py in _get_response
```

- During production, you will only see 500 or 404 errors

## Not Found

The requested URL was not found on this server.

---

Apache/2.4.29 (Ubuntu) Server at cs.toronto.edu Port 80

# Deploying Django Project

# Deployment Options

- You 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 required root privilege
  - Highly not recommended

- **Gunicorn (Green Unicorn)**

- A fast and lightweight WSGI HTTP server

- Installation

- `sudo pip3 install gunicorn`

- Start in terminal

- In the Django project root folder, run:

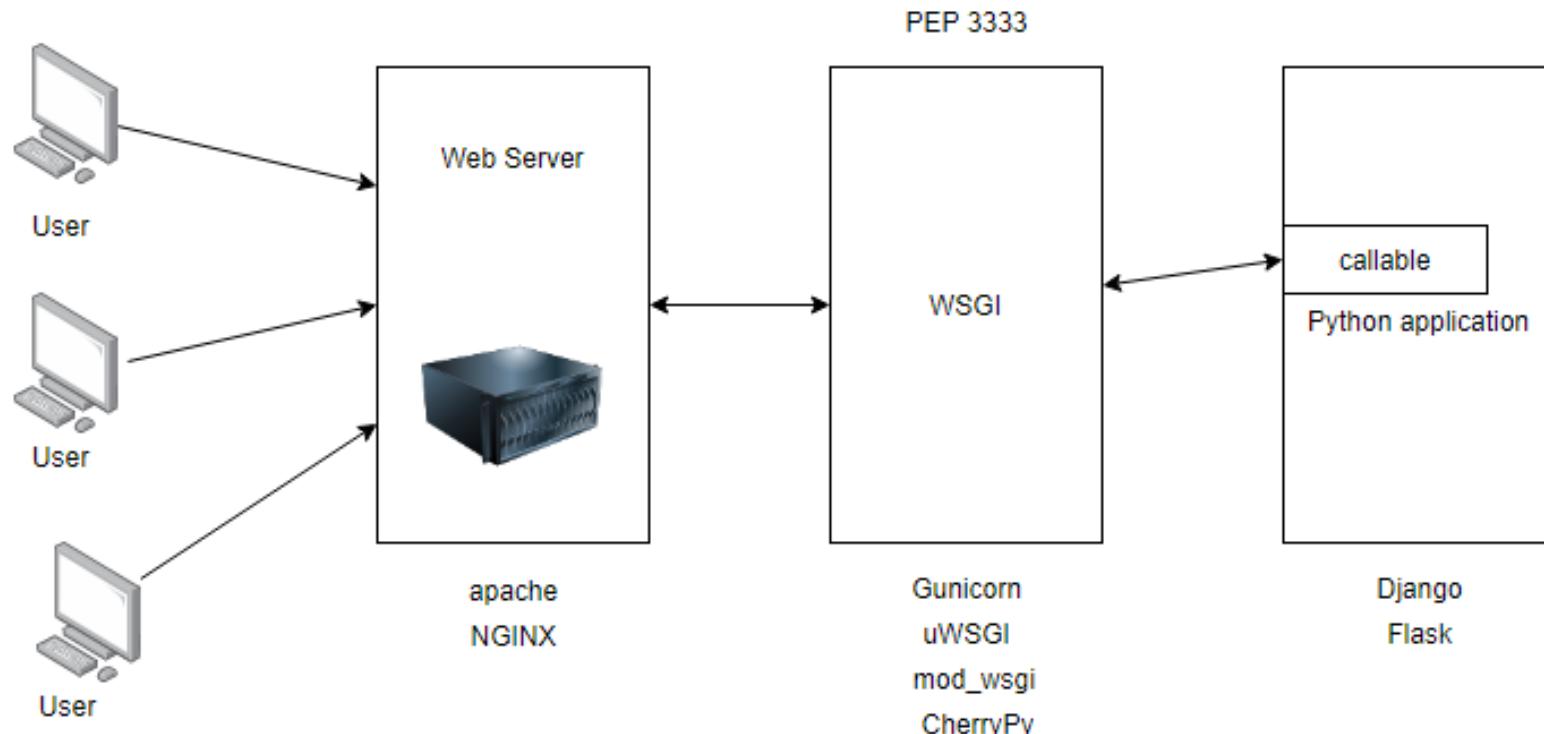
- `gunicorn --bind 0.0.0.0:80 <project>.wsgi`

- Does not serve static file!

Note: `<project>.wsgi`  
is name of a *new* file

# WSGI

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



Source: <https://medium.com/analytics-vidhya/what-is-wsgi-web-server-gateway-interface-ed2d290449e>

# Apache Webserver

- Django's development server
  - Not meant to handle high loads, withstand attacks, etc
- Installation on Ubuntu
  - 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 or entire project folder into `/var/www/`
    - You 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

- Production settings (`settings.py`)
  - DEBUG
    - Should set to False
  - SECRET\_KEY
    - Use a secure secret key instead
  - ALLOWED\_HOSTS
    - Should add your domain name
  - DATABASE
    - You should probably not use SQLite for production purposes
    - <https://docs.djangoproject.com/en/4.1/ref/databases/>
- Do not push `settings.py` into repository!
  - Can be a security leak if you do because database password is stored

# Production Settings

- DJANGO\_SETTINGS\_MODULE
  - Django loads settings from this environment variable
  - Can create another file that imports from `settings.py` and override some options
    - E.g., `export DJANGO_SETTINGS_MODULE=project.production_settings`
- if DEBUG:
  - Can separate debug versus production settings
- .env file
  - Load settings from an environment file
  - One for local, one for production
  - Load it on startup
    - [python-dotenv](#) package

```
DATABASES = {
    'default': {
        'ENGINE': os.environ['DB_ENGINE'],
        'NAME': os.environ['DB_NAME'],
        'USER': os.environ['DB_USER'],
        'PASSWORD': os.environ['DB_PASSWORD'],
        'HOST': os.environ['DB_HOST'],
        'PORT': os.environ['DB_PORT'],
    }
}
```

# Static Files

- Static files are file/directory access granted to the webserver
- In Django project, they are **scattered** in many places
  - app/static folders
  - Global static folder
  - From Django contrib package, e.g., admin panel
- Django can *collect* all of them into STATIC\_ROOT folder
  - Required for security and performance reasons
  - Typically served by the webserver
    - 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 you close the terminal
    - You can start it using `nohup`, which will not shutdown when terminal closes
    - However, it does not restart if machine reboots
  - Solution: make it a [service](#)
    - Runs forever
    - Restarts upon error
    - Runs on startup

# Service

- Linux has a tool to register and monitor your services
- Create new file under the following directory
  - /etc/systemd/system/<service\_name>.service

```
[Unit]
Description=first_projet gunicorn daemon
After=network.target

[Service]
User=ubuntu
WorkingDirectory=/home/ubuntu/first_project
ExecStart=/home/ubuntu/first_project/venv/bin/gunicorn \
           --access-logfile - \
           --workers 3 \
           --bind unix:/home/ubuntu/first_project/first_project.sock \
           first_project.wsgi:application

[Install]
WantedBy=multi-user.target
```

# Service Cont.

- Manage your service via these commands:
  - sudo service <name> restart
  - sudo service <name> status
  - sudo service <name> stop
- If you change the service file, you might need to reload it
  - sudo systemctl daemon-reload
- Note that it binds to a socket, not 0.0.0.0:8000
  - We want a real webserver to serve our application on port 80
  - We cannot let gunicorn take port 80. Why?
- A real webserver forwards dynamic requests to gunicorn
  - Gunicorn services the backend project through a local socket

# Deploy with Nginx

- /etc/nginx/sites-available/<project\_name>
  - Create your config file here
- Make a symbolic link to that file
  - At /etc/nginx/sites-enabled/
- Restart Nginx!

```
/etc/nginx/sites-available/myproject

server {
    listen 80;
    server_name server_domain_or_IP;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root /home/sammy/myproject;
    }

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/sammy/myproject/myproject.sock;
    }
}
```

<https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-22-04>

# Deploying React Project

# Deploy React Project

- Much easier than backend servers

1. Build your React project

- npm run build

2. Configure webserver to serve the appropriate files

- Nginx example

- Just route all requests to the build folder
- Then, restart Nginx

```
server {
    listen 80;
    server_name mysite.com www.mysite.com;
    root /home/ubuntu/app-deploy/build;
    location / {
        try_files $uri /index.html;
    }
}
```

- Further reading

- <https://enrico-portolan.medium.com/how-to-host-react-app-with-nginx-634ad0c8425a>