WEB PROGRAMMING QUIZ NOTES

MCQ Practice Quiz Answers:

HTTP [methods] such as GET and POST are also known as HTTP [verbs]. HTTP [requests and responses] contain a list of headers and some request may contain a [body]

Question: HTTP is a communication protocol that allows clients and servers to communicate about resources over the Web. Select all the true statements about HTTP:

- **An HTTP request must contains all the information about the location (URL) of the resource**
- **HTTP requests and responses will normally contain "representations" of resources or resource changes.**

Question: Select all the true statements about cookies and sessions:

- **Web apps can set cookies on the client's browser using the Set-Cookie header.**
- **Each cookie in the browser is associated with a specific host, and any requests to the host will contain the cookies in the Cookie request header, Sessions can be used to keep track of which users are currently logged in or not.**

Question: Django migrations allow the developer to easily maintain the app's models in "sync" with the database tables. When working with models and migrations, order the following actions in the standard chronological order -- when developer need to update their models:

- **Make changes to models in models.py**
- **Make the migrations using "makemigrations" command**
- **Inspect the migrations using "sqlmigrate" command**
- **Apply the migrations using the "migrate" command**

Question: Web frameworks are powerful tools that help developers build complex and secure web applications. Select all the true statements about web frameworks:

- **In frameworks adopting the model-view-presenter (MVP) architecture, the business logic is implemented in the "presenter",**

Question: Django's ORM and QuerySets allows the developer to access the database without having to write SQL queries

- **In frameworks adopting the model-view-presenter (MVP) architecture, the business logic is implemented in the "presenter",**
- **Django's ORM and QuerySets allows the developer to access the database without having to write SQL queries**

The Django "render" function is normally used to create HttpResponse objects. This function takes three inputs:

- **request object, template, Python dictionary**

Match each regular expression with its correct description

- **/[ae]+\./ → Strings containing one or more a's and e's, and ending with a full stop,**

1.1.4 HTTP Message Structure

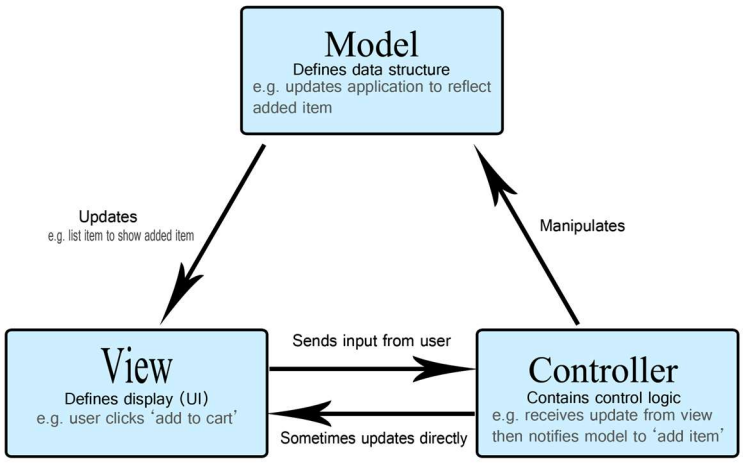
- **Request Message**
    - **Request Line**: METHOD URI HTTP/VERSION (e.g., GET /index.html HTTP/1.1)
    - **Headers**: Key-value pairs (e.g., Host: example.com)
    - **Blank Line**: Indicates the end of headers.
    - **Body**: Optional, used with methods like POST or PUT.
- **Response Message**
    - **Status Line**: HTTP/VERSION STATUS_CODE REASON_PHRASE (e.g., HTTP/1.1 200 OK)
    - **Headers**: Key-value pairs (e.g., Content-Type: text/html)
    - **Blank Line**: Indicates the end of headers.
    - **Body**: Optional, contains the response data.

1.2.1 Overview

- **Definition**: REST is an architectural style for designing networked applications. It leverages HTTP protocols and is based on resources identified by URIs.
- **Principles**:
    1. **Client-Server**: Separation of concerns between client and server.
    2. **Stateless**: Each request contains all necessary information.
    3. **Cacheable**: Responses can be cached to improve performance.
    4. **Uniform Interface**: Standardized methods for communication.
    5. **Layered System**: Intermediate layers can exist without client/server awareness.
    6. **Code on Demand (Optional)**: Servers can extend client functionality by sending executable code.

1.2.2 RESTful API Design

- **Resource Identification**
    - **URI Structure**: Use nouns to represent resources (e.g., /users, /orders/123).
    - **Hierarchy**: Reflect resource relationships (e.g., /users/123/orders).
- **HTTP Methods Usage**
    - **GET**: Retrieve resources.
    - **POST**: Create new resources.
    - **PUT/PATCH**: Update existing resources.
    - **DELETE**: Remove resources.



Model

- **Purpose**: The model is the data layer of the application and is responsible for managing the logic of data storage and retrieval.
- **Responsibilities**:
    - Defines the structure of the data (e.g., fields, data types, and relationships).
    - Interacts with the database using Django's ORM (Object-Relational Mapper).
    - Manages business logic related to the data (e.g., validation rules, constraints).
- **Significance**: Decouples database operations from other components, ensuring a clean and maintainable architecture.

View

- **Purpose**: The view handles the presentation layer, determining how data is displayed to the user.
- **Responsibilities**:
    - Formats data retrieved from the model into a user-friendly format (e.g., HTML, JSON).
    - Ensures user interaction is communicated back to the controller for processing.
- **Significance**: Separates the user interface from the underlying data and logic.

Controller

- **Purpose**: The controller manages user input, processes requests, and updates the model and view accordingly.
- **Responsibilities**:
    - Routes incoming HTTP requests to the appropriate handlers.
    - Executes business logic to process user input.

    - Each function receives a request and returns a response.
    - Suitable for smaller, single-purpose views.
- **Advantages**:
    - Simple to understand and implement.
    - More explicit and easier to debug for beginners.
- **Disadvantages**:
    - Becomes less maintainable and scalable for complex or repetitive logic.

Class-Based Views (CBVs)

- **Definition**: A more modular and reusable approach to define views using Python classes.
- **Key Features**:
    - Views are represented as classes, with specific methods (e.g., get, post) handling different HTTP methods.
    - Allow inheritance and mixins to extend functionality.
- **Advantages**:
    - Cleaner and more organized for complex logic.
    - Enables code reuse through inheritance.
    - Easier to modify behavior by overriding specific methods.
- **Disadvantages**:
    - Slightly steeper learning curve compared to FBVs.

URL Dispatcher

- **Definition**: Django's URL dispatcher maps URLs to their corresponding views, ensuring that requests are routed correctly.
- **Responsibilities**:
    - Define the structure and hierarchy of URLs for the application.
    - Connect specific URLs to the appropriate views.
    - Handle dynamic parameters in URLs (e.g., IDs, slugs).
- **Key Features**:
    - Flexible URL patterns using regular expressions or path converters.
    - Supports reusable app-level URL configurations.

- **Definition**: Decorators are functions that modify the behavior of another function or method without changing its code.
- **Purpose in Django**:
    - Add functionality to views, such as enforcing authentication or restricting access.
- **Common Decorators**:
    - **Authentication**: Restrict views to logged-in users (e.g., @login_required).
    - **CSRF Protection**: Enable/disable Cross-Site Request Forgery protection (e.g., @csrf_exempt).
    - **Caching**: Cache views for improved performance (e.g., @cache_page).
- **Significance**: Decorators enhance reusability and maintainability by separating concerns and avoiding code duplication.

- **Use Cases in Django**
- **1. Authentication**
- When you want to restrict a view to authenticated users:
- Without a decorator, you'd have to write repetitive code to check authentication for every view.
- With the @login_required decorator, you simply apply it to the view, and Django handles the check.
- **2. Access Control**
- For example, using @permission_required, you can enforce that a user has specific permissions to access a view.
- **3. Input Validation**
- Decorators can preprocess the request data, ensuring it meets specific criteria before the main function executes.
- **4. Request Methods**
- Decorators like @require_POST enforce that only specific HTTP methods (e.g., POST) are allowed for a view, reducing potential errors.

How Decorators Work

Decorators leverage Python's feature of treating functions as **first-class objects**, meaning functions can:

- Be assigned to variables.
- Be passed as arguments to other functions.

WEB PROGRAMMING QUIZ NOTES

Quantifiers:

Tell how many times something should occur:

* → Match 0 or more times.

Example: ba* matches b, ba, baaa.

+ → Match 1 or more times.

Example: ba+ matches ba, baa, but not b.

? → Match 0 or 1 time (optional).

Example: colou?r matches color and colour.

{n} → Match exactly n times.

Example: a{3} matches aaa.

Anchors:

Pinpoint positions in text:

^ → Start of a string.

Example: ^Hello matches Hello world, but not Hi Hello.

$ → End of a string.

Example: world$ matches Hello world, but not worldwide.

Special Characters:

Brackets and symbols help refine matches:

[abc] → Match any character inside (e.g., a, b, or c).

[^abc] → Match anything except these characters.

(abc) → Groups parts together. Useful for patterns.

| → Logical OR (e.g., cat|dog matches either "cat" or "dog").

Cookies and Sessions

Cookies

- **Definition**: Small data pieces stored in the user's browser.
- **Uses**:
    - Store user preferences.
    - Track sessions.
- **Attributes**:

2. **Readable Syntax**:
    - Cleaner, more intuitive code compared to XMLHttpRequest.
3. **Supports Advanced Options**:
    - Easily customize requests with headers, query parameters, or request bodies.

Comparison: Ajax vs. Fetch API

Feature	Ajax	Fetch API
Ease of Use	Complex, uses callbacks	Simpler, uses promises
Modernity	Older, relies on `XMLHttpRequest`	Newer, standard JavaScript API
Readability	Less intuitive	Cleaner and more readable syntax

Web App Security

Common Threats

- **SQL Injection**: Exploiting input fields to manipulate databases.
- **Cross-Site Scripting (XSS)**: Injecting malicious scripts.
- **Cross-Site Request Forgery (CSRF)**: Unauthorized actions on authenticated sessions.

Best Practices

- Sanitize inputs.
- Use HTTPS for secure transmission.
- Implement CSRF tokens.

Web App Testing & Deployment

Testing

- **Unit Tests**: Validate individual components.
- **Integration Tests**: Check interactions between components.
- **End-to-End Tests**: Simulate real-world user scenarios.

Deployment

- **Key Steps**:
    - Set up a production server.
    - Use tools like Docker for containerization.
    - Monitor app performance post-deployment.

- **Why**: Prevents attackers from injecting harmful data that could exploit vulnerabilities like XSS or SQL Injection.
- **How**:
  - Validate inputs on both client and server sides.
  - Use **whitelisting** for inputs (e.g., allow only specific characters or patterns).
  - Escape potentially dangerous characters (e.g., <, >, ', ") when rendering user data.

2. **Use HTTPS for Secure Transmission**
   - **Why**: HTTPS encrypts data between the client and server, protecting against eavesdropping and man-in-the-middle attacks.
   - **How**:
     - Always redirect HTTP traffic to HTTPS.
     - Ensure valid SSL/TLS certificates are in place.
     - Consider **HTTP Strict Transport Security (HSTS)** to enforce HTTPS usage.
3. **Implement CSRF Tokens**
   - **Why**: Prevents unauthorized actions on behalf of authenticated users by verifying that requests originate from the user's session.
   - **How**:
     - Include a unique CSRF token in every form or state-changing request.
     - Validate the CSRF token on the server side before processing the request.

Additional Security Measures

1. **Authentication & Authorization**
   - **Use strong password policies** (e.g., minimum length, complexity).
   - Implement **multi-factor authentication (MFA)** for sensitive actions.
   - Use **role-based access control (RBAC)** to ensure users have the appropriate permissions.
2. **Session Management**
   - **Secure cookie attributes**: Set HttpOnly, Secure, and SameSite flags for cookies.

    - **HttpOnly**: Prevents access to cookie data via JavaScript (reduces risk of XSS attacks).
    - **Secure**: Ensures the cookie is only sent over HTTPS, protecting data from being intercepted.
    - **SameSite**: Prevents cross-site request forgery (CSRF) by controlling when cookies are sent with cross-origin requests.
    - **Expires/Max-Age**: Defines the lifespan of the cookie.
    - **Path**: Restricts the cookie to a specific URL path.
- **Common Uses**:
    - **Authentication**: Cookies store session IDs to track logged-in users.
    - **Tracking**: Used in analytics to track user behavior on websites.
    - **Preferences**: Remember settings like language preferences or shopping cart contents.
- **Security Considerations**:
    - **HttpOnly**: Prevents client-side scripts from accessing cookies.
    - **Secure**: Cookies are only transmitted over secure HTTPS connections.
    - **SameSite**: Controls cross-site requests to mitigate CSRF risks.
    - **Encryption**: Sensitive cookie data should be encrypted to prevent exposure if cookies are intercepted.

Sessions

- **Definition**: Server-side storage that keeps track of user interactions and data over multiple requests.
- **Mechanism**:
    - A **session ID** is typically stored in a **cookie** on the client-side.
    - The server maintains a **session store** where user-specific data (such as authentication status, user preferences) is associated with the session ID.
- **How It Works**:
    - **Session ID** is generated upon login or first user interaction and stored in a cookie on the client-side.
    - The server maintains a session store (could be memory, database, or file-based) mapping session IDs to user data.
    - On subsequent requests, the browser sends the session ID to the server via cookies, allowing the server to load the user's session data.

1. **Set Up a Production Server**:
    - **Steps**:
        - Choose an appropriate hosting provider (e.g., AWS, DigitalOcean, Heroku).
        - Install necessary dependencies (e.g., web server software like Nginx or Apache).
        - Set up environment variables for production (e.g., database credentials, secret keys).
    - **Considerations**: Ensure that production server configurations (e.g., database, web server) are optimized for security and performance.
2. **Use Docker for Containerization**:
    - **Definition**: Containerize the application to ensure consistent environments across development, testing, and production.
    - **Steps**:
        - Write a Dockerfile to define the app's dependencies and setup.
        - Use **Docker Compose** to orchestrate multi-container apps (e.g., app, database, cache).
        - Build and deploy the Docker container to cloud platforms or a local server.
    - **Advantages**: Ensures the app runs consistently across different environments, reduces dependency issues.
3. **Monitor App Performance Post-Deployment**:
    - **Monitoring Tools**:
        - **Logging**: Use logging frameworks (e.g., Logstash, Winston) to track app errors and performance metrics.
        - **Performance Monitoring**: Use tools like **New Relic**, **Datadog**, or **Prometheus** for real-time monitoring of server performance and application metrics.
        - **Error Reporting**: Integrate error tracking services like **Sentry** to capture runtime errors in production.
    - **Considerations**:
        - Set up **alerts** for critical performance issues (e.g., high CPU usage, slow response times).
        - Regularly check **server logs** for errors or unexpected behavior.
        - Ensure backups of critical data (e.g., database).
