web technology assignment

1. Explain the architecture of tier technology. Compare single-tier, two-tier, and three-tier architectures with suitable examples.

Tiered architecture (or *n-tier architecture*) splits an application into logical layers (tiers). Each tier is responsible for specific concerns — separation of presentation, logic, and data improves maintainability, scalability, and reuse.

Common tiers:

- **Presentation tier** (UI, client) — what the user sees (browser, mobile app).
- **Application/business logic tier** — processes data, enforces rules.
- **Data tier** — database/storage.

Single-tier (monolithic / fat client)

- **Definition:** All components (UI, business logic, data) run in the same process or machine.
- **Example:** A single desktop application (e.g., MS Word storing and processing documents locally). Or an early PHP script and MySQL running on the same server, all tightly coupled.
- **Pros:** Simple to develop and deploy for small apps; fewer network calls.
- **Cons:** Hard to scale, maintain, or reuse; upgrading one part may require redeploying the entire app.

Two-tier (client–server)

- **Definition:** Client directly interacts with server; typically UI runs on client and the database/server hosts data and logic.
- **Example:** A desktop app that connects directly to a remote database (client = GUI app, server = DBMS). Or a thin web client that makes SQL queries to a DB server (not recommended in practice).
- **Pros:** Simpler than n-tier; separation between UI and data.
- **Cons:** Business logic often resides on the client or server in an ad-hoc way; scaling clients or supporting many clients is harder; direct DB access from clients is insecure.

Three-tier (presentation, application, data)

- **Definition:** Clear separation: client (presentation) ↔ application server (business logic) ↔ database server (data).
- **Example:** Classic web app: Browser (HTML/CSS/JS) → Web server / app server (Node.js, Java EE, Django) → Database (MySQL, PostgreSQL).
- **Pros:** Scalable (you can scale app servers separately), more secure (DB not exposed), easier to maintain and test, reusability of business logic across different clients.
- **Cons:** More complex to design and deploy; network latency between tiers.

Comparison summary table (quick):

- **Complexity:** single-tier < two-tier < three-tier
- **Scalability:** single-tier low, two-tier medium, three-tier high
- **Security:** single-tier low, two-tier medium, three-tier high
- **Maintainability:** single-tier poor, three-tier best

2. What is the World Wide Web (WWW)? Describe its components and how they work together.

World Wide Web (WWW) is a system of interlinked hypertext documents and resources accessed via the Internet using web protocols (mainly HTTP/HTTPS). It's the layer that allows humans to access information pages, media, and applications through browsers.

Major components & how they interact:

1. **Client (Web Browser)** — user agent that requests resources and renders them (e.g., Chrome, Firefox).
    - Sends HTTP requests, renders HTML/CSS/JS, runs client-side scripts.
2. **Web Server** — software that receives HTTP requests and returns responses (HTML pages, JSON, images). Examples: Apache, Nginx.
    - May serve static files or pass requests to application servers for dynamic content.
3. **Application Server / Backend** — executes business logic and generates dynamic content (Node.js, Django, Spring).
    - Interacts with databases, authentication systems, external APIs.
4. **Database / Storage** — persistent storage for data (RDBMS, NoSQL, file systems, object storage).
    - Stores users, posts, transactions, session data.
5. **Protocols** — rules for communication (HTTP/HTTPS, TCP/IP, DNS).
    - DNS maps domain names to IP addresses; TCP/IP handles underlying transport; HTTPS secures traffic with TLS.
6. **Client-side resources** — HTML, CSS, JavaScript, images, fonts.
    - Browser downloads these and renders the page.
7. **APIs & Services** — external services (payment gateways, maps, analytics) that the app consumes.
8. **Middleware / Caching / CDN** — performance and scaling: CDNs deliver static content near users, caches reduce backend load (Redis, Varnish).
9. **Security layers** — TLS/SSL, authentication, firewalls.

End-to-end flow (simple):
User enters URL → DNS resolves domain → browser connects to server (via HTTPS/TCP) → sends HTTP request → server/app processes and queries DB if needed → server sends back HTTP response with HTML/JSON → browser renders page, fetches assets, executes JS → page displayed.

3. Define a web browser. Explain its functions and list at least four examples along with their JavaScript engines.

Web browser: A software application that retrieves, interprets, and displays content from the Web (HTML, CSS, JavaScript, images, video). It acts as the client in the client-server model and provides the interface for users to interact with web apps.

Main functions:

- **URL navigation & address bar** — let user enter web addresses.
- **HTTP/HTTPS requests** — send requests to servers and handle responses.
- **Rendering engine (HTML/CSS layout)** — parse HTML/CSS and compute layout and paint pixels.
- **JavaScript engine** — parse and execute JavaScript to enable interactivity and dynamic behavior.
- **Networking** — manage TCP/TLS, caching, and resource loading.
- **DOM (Document Object Model) management** — expose a programmatic structure for scripts to interact with the page.

- **Security sandboxing** — isolate website content from the system and other tabs.
- **Extensions & developer tools** — allow plugins and debugging tools.

Examples + JavaScript engines:

1. **Google Chrome** — *V8* JavaScript engine.
2. **Mozilla Firefox** — *SpiderMonkey* (and IonMonkey for JIT).
3. **Microsoft Edge** (Chromium-based) — *V8* (since it uses Chromium engine).
4. **Apple Safari** — *JavaScriptCore* (also known as *Nitro*).
   (Other examples: Opera uses V8; Brave uses V8.)

4. What is a web server? Explain its role and list its major functions with examples.

Web server: Software (and often hardware) that listens for HTTP(S) requests from clients (browsers) and returns HTTP responses (files, generated HTML, JSON, etc.). It's the endpoint that hosts and serves web resources.

Role: Accept requests, process them (or forward to application logic), and respond with appropriate resources, while managing connections, security, and logging.

Major functions:

- **Request handling** — receive and parse HTTP requests.
- **Static content serving** — deliver files like HTML/CSS/JS/images directly.
- **Dynamic content forwarding** — hand off requests to application servers (CGI, FastCGI, reverse proxies).
- **Security & TLS termination** — manage HTTPS certificates and encryption.
- **Load balancing & reverse proxying** — distribute traffic among multiple backend servers (Nginx, HAProxy).
- **Logging & monitoring** — request logs, access/error logs for debugging and analytics.
- **Compression & caching** — gzip/ Brotli and caching headers to speed delivery.
- **URL rewriting & redirects** — map friendly URLs to real resources or redirect traffic.
- **Authentication / access control** — restrict or permit access to resources.

Examples of web server software:

- **Apache HTTP Server** — widely used, modular.
- **Nginx** — high-performance, reverse proxy and static server.
- **Microsoft IIS** — Windows-based web server.
- **LiteSpeed, Caddy** — other modern servers with specific features.

5. Differentiate between static and dynamic web pages with examples.

Static web pages:

- **Definition:** Content fixed on the server as files; the same HTML is returned to every user (unless manually changed).
- **Example:** A plain HTML portfolio page, or a static blog page generated by a static site generator (Hugo, Jekyll).
- **Characteristics:** Quick to serve, easier to cache/CDN, no server-side processing per request, lower server load.
- **When to use:** Brochure sites, documentation, blogs with infrequent updates.

Dynamic web pages:

- **Definition:** Content is generated on-the-fly based on user input, database queries, or runtime logic. Different users can see different content.
- **Example:** Social media feeds, e-commerce product pages that show personalized recommendations, or a dashboard showing a user's data (Facebook, Amazon product pages).
- **Characteristics:** Requires server-side processing or client-side rendering with API calls; can be personalized; involves databases and business logic.
- **When to use:** Personalized sites, apps requiring frequent updates, user-specific content.

Concrete examples:

- Static: `about.html` with company info.
- Dynamic: `profile.php?user=123` which pulls user data from a DB and builds the HTML.

6. Explain how a dynamic web page works with step-by-step processing workflow.

Let's follow a typical **server-rendered dynamic page** (e.g., a user visits their profile page):

1. **User action**: User types URL or clicks a link (e.g., `https://site.com/profile`) and presses Enter.
2. **DNS resolution**: Browser asks DNS to resolve `site.com` → receives IP of the web server.
3. **TCP/TLS connection**: Browser opens TCP connection to server and negotiates TLS (if HTTPS).
4. **HTTP request**: Browser sends an HTTP GET request for `/profile` with headers (cookies, user-agent).
5. **Web server receives request**: Server (Nginx/Apache) accepts request. If static resource — serve file; if dynamic — forward to application server or invoke script.
6. **Application server processing**: Backend (e.g., Node, Django, PHP) runs route/controller corresponding to `/profile`.
    - **Authentication check**: Validate session cookie or token to identify the user.
    - **Business logic**: Determine what data is needed (user info, friends list, notifications).
    - **Database queries**: Query DB for user details, posts, etc.
    - **Data processing**: Format results, apply business rules (filtering, sorting).
7. **Template rendering**: Populate an HTML template with the retrieved data to produce final HTML.
8. **Server response**: Application returns the generated HTML to the web server, which sends it back as an HTTP response with headers (cache-control, cookies).
9. **Browser receives response**: Parses the HTML, constructs the DOM, requests additional resources (CSS, JS, images).
10. **Client-side scripts**: JavaScript may run to fetch more data via AJAX/Fetch APIs or make UI interactive.
11. **Rendering**: Browser renders page layout and paints it to screen.
12. **Asynchronous updates**: Later, the client may call APIs (JSON endpoints) to update parts of the page without a full reload (SPA behavior).

If using **client-side rendering / SPA** (React/Angular/Vue): backend may only provide JSON API; the browser loads a shell HTML and JS bundle, which then fetches data and renders the UI entirely in the client.

7. What is client-side scripting? Explain its features, advantages, and common languages used.

Client-side scripting: Code that runs in the client (browser) to add interactivity, validate input, manipulate DOM, and

sometimes render UI. It executes after resources are downloaded, inside the user's browser.

Features:

- Runs in the browser sandbox.
- Manipulates the DOM dynamically (change content without reload).
- Responds to user events (clicks, form submissions).
- Can make asynchronous HTTP requests (AJAX/fetch) to backend APIs.
- Works offline or partially offline with service workers (progressive web apps).
- Immediate feedback to user (fast UI interactions).

Advantages:

- **Faster UI responsiveness** — reduces round trips for UI updates.
- **Reduced server load** — some processing moves to client.
- **Rich interactivity** — dynamic effects, single-page apps (SPAs).
- **Better UX** — immediate validation and feedback.

Limitations / Caveats:

- Security — client code can be viewed and manipulated by the user; never trust client-side validation for security-critical checks.
- Browser compatibility issues (handled by polyfills/transpilation).
- Performance varies with device capability.

Common client-side languages/tools:

- **JavaScript (ECMAScript)** — the lingua franca of the web.
- **TypeScript** — typed superset of JavaScript (compiles to JS).
- **HTML/CSS** — markup and styling (not scripting but critical client-side technologies).
- **WASM (WebAssembly)** — for high-performance tasks, run compiled code in browser (with JS glue).

8. What is server-side scripting? Describe its need and commonly used languages.

Server-side scripting: Code that runs on the web server to handle requests, interact with databases, and generate content. It's responsible for business rules, data persistence, authentication, and security-critical operations.

Need for server-side scripting:

- **Secure data handling** — secret keys, database credentials must remain on server.
- **Database operations** — queries, transactions, joins handled server-side.
- **Business logic** — calculations, validation, authorization.
- **Dynamic content generation** — produce HTML or API responses based on data and user state.
- **Integration** — talk to payment gateways, email services, third-party APIs.

Common server-side languages & stacks:

- **JavaScript (Node.js / Express)** — event-driven, good for I/O heavy apps.
- **Python (Django, Flask)** — concise syntax, strong ecosystem.
- **PHP (Laravel, core PHP)** — historically common for web apps and CMS (WordPress).

- **Java (Spring, Java EE)** — enterprise-grade apps.
- **C# (.NET / ASP.NET Core)** — Windows and cross-platform web apps.
- **Ruby (Rails)** — convention-over-configuration web development.
- **Go** — performant, simple concurrency for web services.
- **Rust, Elixir** — newer languages used for niche high-performance backends.

9. Compare client-side and server-side scripting. Provide at least five differences.

Here are clear differences:

1. **Where code runs:**
   - *Client-side:* Runs in the browser on the user's device.
   - *Server-side:* Runs on the web server.
2. **Security & trust:**
   - *Client-side:* Visible to users and editable — not trustworthy for validation/security.
   - *Server-side:* Hidden from users — used for secure operations (auth, DB access).
3. **Resource access:**
   - *Client-side:* Access limited (no direct DB or server resources).
   - *Server-side:* Full access to databases, file systems, and secret keys.
4. **Performance & scalability concerns:**
   - *Client-side:* Shifts computation to user device — relieves server but depends on client hardware.
   - *Server-side:* Centralized processing; can become a bottleneck requiring scaling.
5. **Network dependency:**
   - *Client-side:* Can provide offline interactions with service workers; reduces round trips for UI updates.
   - *Server-side:* Requires network calls for dynamic content or API data.
6. **Primary languages:**
   - *Client-side:* JavaScript/TypeScript, HTML/CSS.
   - *Server-side:* Node.js, Python, PHP, Java, C#, Ruby, Go, etc.
7. **Use cases:**
   - *Client-side:* UI animation, form validation, SPA rendering.
   - *Server-side:* Authentication, database queries, business rules, generating content.

(That's more than five — nice & solid for exam points.)

10. Explain HTTP. Describe how HTTP works step-by-step from request to response.

HTTP (Hypertext Transfer Protocol) is the application-layer protocol used for fetching resources (HTML, images, JSON) on the Web. It's stateless and follows a request-response model over TCP (often with TLS → HTTPS).

Key concepts:

- **Methods (verbs):** GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS (specify intent).

- **Status codes:** 200 OK, 301/302 redirects, 404 Not Found, 500 Server Error, etc.
- **Headers:** Metadata about request/response (Content-Type, Authorization, Cache-Control).
- **Body:** Optional payload (POST data, JSON).
- **Stateless:** Each request is independent (sessions/cookies preserve state).

Step-by-step flow (request → response):

1. **User initiates**: Browser enters URL or clicks a link. Example: `https://example.com/page`.
2. **URL parsing**: Browser extracts protocol (https), host (example.com), path (/page), port (default 443 for HTTPS).
3. **DNS lookup**: Resolve domain to an IP address via DNS.
4. **TCP connection**: Browser opens a TCP connection to the server IP at port 443.
5. **TLS handshake (if HTTPS)**: Client and server negotiate encryption keys for a secure channel.
6. **Forming HTTP request**: Browser constructs HTTP request line and headers, e.g. `GET /page HTTP/1.1`, headers like `Host`, `User-Agent`, `Accept`, `Cookie`. Include body for methods like POST.
7. **Send request**: Browser sends request bytes over the (encrypted) TCP connection.
8. **Server receives & processes**: Web server accepts connection, parses request. If static, it reads file; if dynamic, it forwards to app server which runs code and/or queries DB.
9. **Server builds response**: Compose response status line (`HTTP/1.1 200 OK`), headers (`Content-Type: text/html; charset=utf-8`, `Content-Length`, caching headers) and body (HTML/JSON).
10. **Server sends response**: Bytes sent back across the connection to the client.
11. **Client receives & processes**: Browser parses response; if `Content-Type: text/html`, it constructs DOM, requests additional resources (CSS/JS/images) via new HTTP requests.
12. **Rendering & JS execution**: Browser lays out page and executes client-side scripts; scripts may make further API calls (XHR/fetch).
13. **Connection management**: HTTP/1.1 used persistent connections by default; HTTP/2 multiplexes many streams on one connection; connections close based on headers or timeouts.
14. **Caching & conditional requests**: Browser or proxies might serve cached responses or send conditional requests (`If-Modified-Since`) to check freshness.
15. **Follow-up actions**: Redirects (3xx) cause the browser to repeat the flow for a new URL; forms and AJAX trigger additional requests.

Example: GET `/index.html`

- Request: `GET /index.html HTTP/1.1` + headers
- Response: `HTTP/1.1 200 OK` + `Content-Type: text/html` + HTML body

Security & modern features:

- Use HTTPS for encryption; HTTP/2 for multiplexing and better performance; headers like `HSTS`, `CSP` improve security.
