**What is a Program**

* **Definition:** A program is a set of instructions that tells a computer what to do. It's like a recipe for the computer, outlining the steps to achieve a specific goal.

**LAB EXERCISE : "Hello World"**

* **Python:**  
  Python  
  print("Hello, World!")
* **JavaScript:**  
  JavaScript  
  console.log("Hello, World!");
* **Comparison:**
  + Both achieve the same output: displaying "Hello, World!" on the console.
  + **Structure:** Both use a function call (print() in Python, console.log() in JavaScript) to output the message.
  + **Syntax:** Minor differences in how the function is called and how the string is represented.

**THEORY EXERCISE: What is a Program and How it Functions**

* A program is a sequence of instructions written in a programming language that a computer can understand and execute.
* **How it Functions:**
  1. **Writing:** The programmer writes the code using a text editor or an Integrated Development Environment (IDE).
  2. **Compilation/Interpretation:**
     + **Compiled Languages:** The code is translated into machine code (binary instructions) by a compiler. This machine code can then be directly executed by the computer's processor.
     + **Interpreted Languages:** The code is executed line by line by an interpreter. The interpreter translates each line into machine code and executes it immediately.
  3. **Execution:** The computer's processor executes the machine code instructions, performing the actions specified in the program.

**What is Programming?**

**THEORY EXERCISE: Key Steps in the Programming Process**

1. **Problem Definition:** Clearly understand the problem to be solved.
2. **Algorithm Design:** Create a step-by-step plan (algorithm) to solve the problem.
3. **Code Implementation:** Write the code in a chosen programming language.
4. **Testing:** Thoroughly test the program to find and fix errors (debugging).
5. **Maintenance:** Update and improve the program as needed.

**Types of Programming Languages**

* **High-Level Languages:**
  + **Human-readable:** Easier for humans to understand and write.
  + **Abstraction:** Provide a higher level of abstraction, hiding the complexities of the underlying hardware.
  + **Examples:** Python, JavaScript, Java, C++, C#
* **Low-Level Languages:**
  + **Machine-oriented:** Closer to the computer's hardware.
  + **Difficult for humans to read and write directly.**
  + **Examples:** Assembly language, Machine code

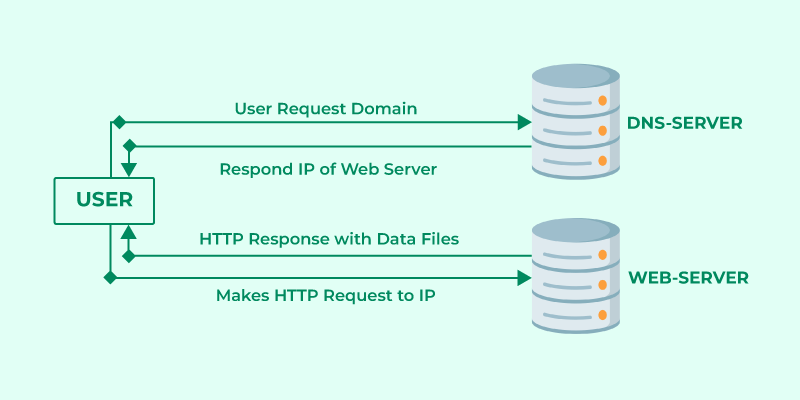
**THEORY EXERCISE: Differences Between High-Level and Low-Level Languages**

| Feature | High-Level Languages | Low-Level Languages |
| --- | --- | --- |
| Readability | Easier to read and write for humans | Difficult for humans to read and write directly |
| Abstraction | Higher level of abstraction, hides hardware details | Lower level of abstraction, closer to hardware |
| Portability | Generally more portable across different systems | Less portable, often specific to a particular hardware |
| Execution Speed | Usually slower than low-level languages | Generally faster, but harder to develop and maintain |

**World Wide Web & How Internet Works**

**LAB EXERCISE: Data Transmission Diagram (Client-Server)**

* **Diagram:**



**THEORY EXERCISE: Roles of Client and Server**

* **Client:**
  + **Initiates:** Starts the communication by sending a request to the server.
  + **Requests:** Sends requests for data or services (e.g., web pages, files).
  + **Receives:** Receives data or services from the server.
* **Server:**
  + **Waits:** Listens for requests from clients.
  + **Receives:** Receives requests from clients.
  + **Processes:** Processes the requests and determines the appropriate response.
  + **Sends:** Sends responses (data or services) back to the clients.

**Network Layers on Client and Server**

* **TCP/IP Model:**
  + **Application Layer:** Handles application-specific protocols (HTTP, FTP, SMTP).
  + **Transport Layer:** Ensures reliable data delivery (TCP) or faster, less reliable delivery (UDP).
  + **Network Layer:** Handles the logical addressing and routing of data packets.
  + **Link Layer:** Deals with the physical transmission of data over the network.

**LAB EXERCISE 3: Simple HTTP Client-Server (Python)**

**Server (http\_server.py)**

import socket

HOST = '127.0.0.1' # Localhost

PORT = 8080 # Port number

# Create a socket object

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.bind((HOST, PORT))

server\_socket.listen(5)

print(f"Server running on http://{HOST}:{PORT}")

while True:

client\_socket, client\_address = server\_socket.accept()

request = client\_socket.recv(1024).decode()

print(f"Received request:\n{request}")

# Simple HTTP response

response = """\

HTTP/1.1 200 OK

Content-Type: text/html

<html>

<body>

<h1>Hello, World!</h1>

</body>

</html>

"""

client\_socket.sendall(response.encode())

client\_socket.close()

**Client (http\_client.py)**

import socket

HOST = '127.0.0.1' # Server address

PORT = 8080 # Server port

# Create a socket object

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client\_socket.connect((HOST, PORT))

# Send an HTTP GET request

request = "GET / HTTP/1.1\r\nHost: 127.0.0.1\r\n\r\n"

client\_socket.sendall(request.encode())

# Receive response

response = client\_socket.recv(4096).decode()

print("Received response:\n", response)

client\_socket.close()

**How It Works**

1. Run http\_server.py to start the server.
2. Run http\_client.py to send an HTTP request.
3. The server responds with a simple "Hello, World!" HTML page.
4. The client prints the server's response.

**THEORY EXERCISE: TCP/IP Model and its Layers**

* **TCP/IP Model:** A conceptual framework for how data is transmitted over the internet.
* **Layers:**
  + **Application Layer:** Handles application-specific protocols (HTTP, FTP, SMTP).
  + **Transport Layer:** Ensures reliable data delivery (TCP) or faster, less reliable delivery (UDP).
  + **Network Layer:** Handles the logical addressing and routing of data packets (IP addresses).
  + **Link Layer:** Deals with the physical transmission of data over the network (e.g., Ethernet).

**Client and Servers**

**THEORY EXERCISE: Client-Server Communication**

* **Client-Server Architecture:** A model where clients request services from servers.
* **How it Works:**
  1. Client initiates a connection to the server.
  2. Client sends a request to the server (e.g., HTTP GET request for a web page).
  3. Server receives the request and processes it.
  4. Server sends a response back to the client (e.g., the requested web page).
  5. Client receives and displays the response.

**Types of Internet Connections**

**LAB EXERCISE: Research different types of internet connections**

| Connection Type | Pros | Cons |
| --- | --- | --- |
| **Broadband** | Widely available, relatively affordable | Speed can vary, potential for congestion |
| **Fiber Optic** | Fastest speeds, high bandwidth, low latency | More expensive, less widespread availability |
| **Satellite** | High availability (rural areas), large coverage | High latency, potential for signal disruption (weather) |
| **Mobile (4G/5G)** | Portable, increasing speeds | Can be expensive, data caps, signal strength can vary |
| **Dial-up** | Inexpensive (older technology) | Very slow speeds, limited availability |

**THEORY EXERCISE: How does broadband differ from fiber-optic internet?**

* **Broadband:** Generally refers to high-speed internet connections using various technologies like DSL (Digital Subscriber Line) over copper wires or cable internet over coaxial cables.
* **Fiber Optic:** Uses thin glass strands to transmit data as light signals.

**Key Differences:**

| Feature | Broadband | Fiber Optic |
| --- | --- | --- |
| **Technology** | Uses existing copper or coaxial lines | Uses glass fibers to transmit light signals |
| **Speed** | Generally slower than fiber optic | Significantly faster speeds |
| **Latency** | Higher latency (more delay) | Lower latency (less delay) |
| **Availability** | Widely available, but speeds can vary | Less widespread availability |
| **Cost** | Generally more affordable | Typically more expensive |

**Protocols**

**LAB EXERCISE: Simulate HTTP and FTP requests using command line tools (e.g., curl)**

* **HTTP GET:**  
  Bash  
  curl -v http://www.example.com   
  + -v: Verbose output (shows request and response headers)
* **HTTP POST:**  
  Bash  
  curl -d "name=John&city=NewYork" http://www.example.com/submit   
  + -d: Send data as POST parameters
* **FTP Download:**  
  Bash  
  curl -O ftp://username:password@ftp.example.com/file.txt   
  + -O: Download and save to current directory

**THEORY EXERCISE: What are the differences between HTTP and HTTPS protocols?**

* **HTTP (Hypertext Transfer Protocol):**
  + **Plaintext:** Transmits data in plain text, making it vulnerable to eavesdropping and data interception.
  + **Not Secure:** Does not provide encryption for data in transit.
* **HTTPS (Hypertext Transfer Protocol Secure):**
  + **Encrypted:** Uses TLS/SSL encryption to secure the communication channel.
  + **Secure:** Protects data from being intercepted or tampered with during transmission.
  + **Authentication:** Verifies the identity of the website server.

**In Summary:**

* **HTTPS** is a more secure version of HTTP that encrypts data and provides authentication, making it essential for sensitive transactions like online banking and e-commerce.

**Application Security**

**LAB EXERCISE: Three Common Application Security Vulnerabilities**

1. **SQL Injection:**
   * **Explanation:** Attackers exploit vulnerabilities in web applications that allow them to inject malicious SQL code into database queries. This can lead to data theft, unauthorized access, or system crashes.
   * **Example:** If an application directly uses user input in SQL queries without proper validation, an attacker could inject commands like ' OR 1=1-- to bypass authentication.
   * **Solution:**
     + **Prepared Statements:** Use parameterized queries to prevent direct injection of user input into SQL statements.
     + **Input Validation:** Validate and sanitize all user input to remove any potentially harmful characters.
     + **Least Privilege:** Grant database users only the necessary permissions to perform their tasks.
2. **Cross-Site Scripting (XSS):**
   * **Explanation:** Attackers inject malicious scripts into web pages viewed by other users. This allows them to steal cookies, hijack sessions, or redirect users to malicious websites.
   * **Example:** If an application displays user-provided data without proper encoding, an attacker could inject JavaScript code into a comment field, which would then be executed by other users visiting the page.
   * **Solution:**
     + **Input Validation:** Validate and sanitize all user input to remove or escape any potentially harmful characters.
     + **Output Encoding:** Encode all user-provided data before displaying it on the page.
     + **Use a Content Security Policy (CSP):** A CSP can help mitigate XSS attacks by restricting the sources from which the browser can load scripts.
3. **Broken Authentication and Session Management:**
   * **Explanation:** Weaknesses in authentication and session management can allow attackers to compromise user accounts, steal sensitive information, or hijack user sessions.
   * **Example:** Storing passwords in plain text, using weak session management techniques, or not implementing proper password policies can leave applications vulnerable to attacks.
   * **Solution:**
     + **Strong Password Policies:** Enforce strong password requirements (length, complexity).
     + **Secure Password Storage:** Hash and salt passwords before storing them.
     + **Secure Session Management:** Use secure session IDs, implement session timeouts, and protect session cookies.
     + **Multi-Factor Authentication (MFA):** Implement MFA to add an extra layer of security.

**THEORY EXERCISE: Role of Encryption in Securing Applications**

Encryption plays a crucial role in securing applications by protecting sensitive data both at rest and in transit:

* **Data at Rest:** Encryption protects data stored on servers, databases, and other storage devices from unauthorized access.
* **Data in Transit:** Encryption protects data transmitted over networks, such as between a web browser and a server, from being intercepted and read by third parties.

**Software Applications and Its Types**

**LAB EXERCISE: Classifying 5 Daily Applications**

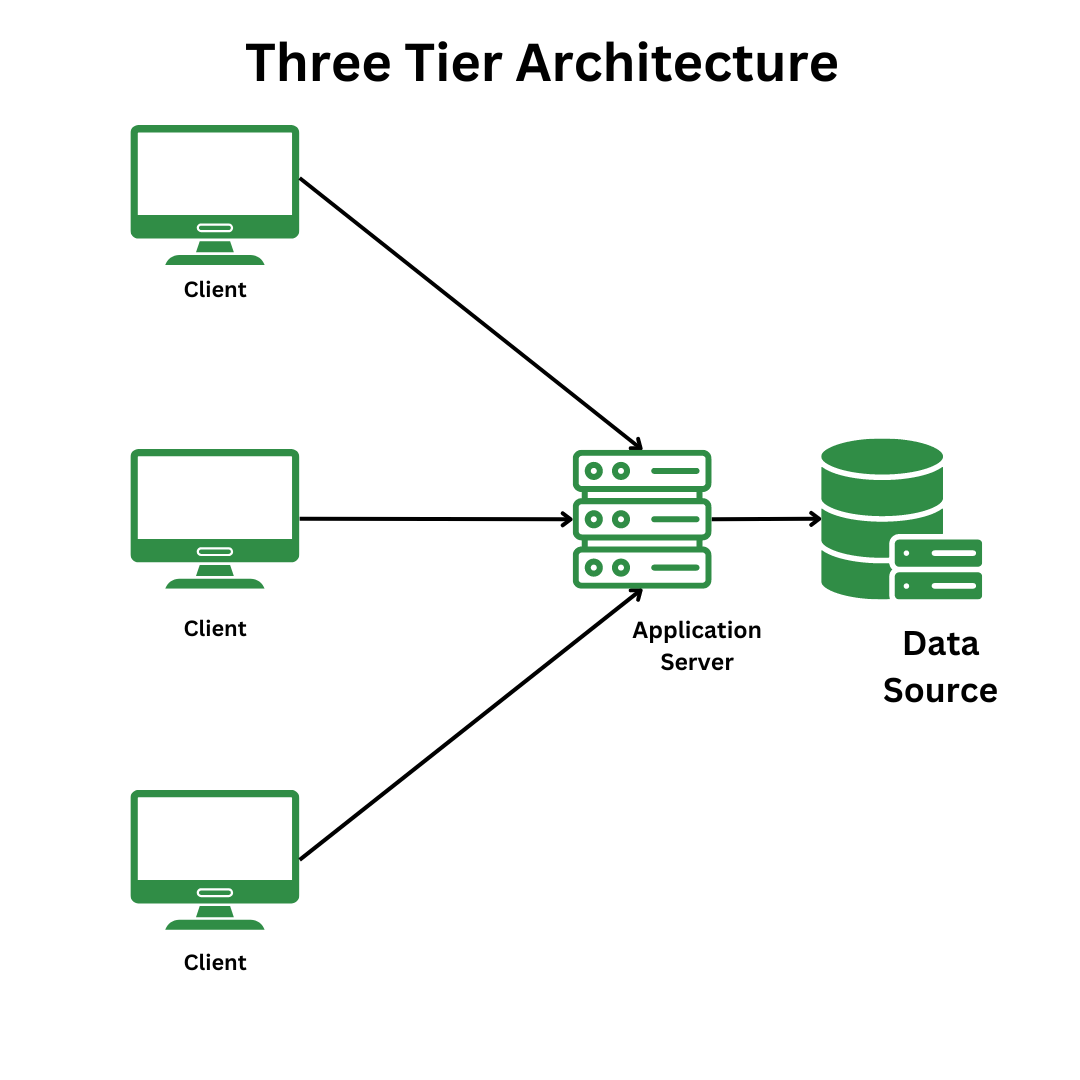
1. **Web Browser (Application Software):** Used for accessing and interacting with websites.
2. **Operating System (System Software):** Manages computer hardware and software resources.
3. **Word Processor (Application Software):** Used for creating and editing text documents.
4. **Antivirus Software (System Software):** Protects the computer from malware and viruses.
5. **Video Player (Application Software):** Used for playing video files.

**THEORY EXERCISE: System Software vs. Application Software**

* **System Software:**
  + **Purpose:** Controls and manages the computer hardware and provides a platform for other software to run.
  + **Examples:** Operating systems (Windows, macOS, Linux), device drivers, firmware.
* **Application Software:**
  + **Purpose:** Performs specific tasks for users.
  + **Examples:** Web browsers, word processors, spreadsheets, games, media players.

**Software Architecture**

**LAB EXERCISE: Three-Tier Architecture Diagram**

****

**THEORY EXERCISE: Significance of Modularity in Software Architecture**

Modularity is crucial in software architecture because it:

* **Improves Maintainability:** Makes it easier to modify and update specific parts of the application without affecting other parts.
* **Enhances Reusability:** Allows components to be reused in other applications.
* **Reduces Complexity:** Breaks down complex systems into smaller, more manageable units.
* **Facilitates Team Collaboration:** Enables different teams to work on different modules independently.

**Layers in Software Architecture**

**LAB EXERCISE: Case Study on Layers (Example: E-commerce Website)**

* **Presentation Layer:**
  + User interface elements: Product catalog, shopping cart, checkout page, user profiles.
  + Handles user interactions and displays information.
* **Business Logic Layer:**
  + Calculates prices (including discounts and taxes).
  + Processes orders (payment processing, inventory management).
  + Manages user accounts and authentication.
  + Implements business rules (e.g., eligibility for discounts).
* **Data Access Layer:**
  + Interacts with the database to:
    - Retrieve product information.
    - Store order details.
    - Update inventory levels.
    - Manage user accounts.

**THEORY EXERCISE: Why are Layers Important in Software Architecture?**

Layers are essential for:

* **Improved Maintainability:** Changes in one layer have minimal impact on other layers.
* **Increased Reusability:** Layers can be reused in different applications.
* **Enhanced Testability:** Allows for easier testing of individual components.
* **Better Scalability:** Enables easier scaling of specific parts of the application.

**Software Environments**

**LAB EXERCISE: Exploring Different Software Environments**

* **Development Environment:**
  + **Purpose:** Used by developers to write, test, and debug code.
  + **Components:** Code editor, compiler/interpreter, debugger, version control system (e.g., Git).
* **Testing Environment:**
  + **Purpose:** Used to test the application before deployment to ensure it meets requirements and is free of bugs.
* **Production Environment:**
  + **Purpose:** The live environment where the application is deployed and used by end-users.
* **Setting up a Basic Environment in a Virtual Machine:**
  1. **Install a Virtual Machine Software:** VirtualBox, VMware Workstation.
  2. **Create a Virtual Machine:** Choose an operating system (e.g., Linux, Windows).
  3. **Install Necessary Software:** Install the development environment (e.g., IDE, compiler, version control system).

**THEORY EXERCISE: Importance of a Development Environment**

* **Provides a Controlled Environment:** Isolates development activities from the production environment, preventing accidental changes to live systems.
* **Facilitates Collaboration:** Enables developers to work independently and share code effectively.
* **Improves Productivity:** Provides tools and resources to increase development speed and efficiency.
* **Reduces Errors:** Helps identify and fix bugs early in the development process.

**Source Code**

**LAB EXERCISE: Writing and Uploading Source Code to GitHub**

1. **Write a Simple Program:** (e.g., a "Hello, World!" program in Python)
2. **Create a GitHub Account:** If you don't have one already.
3. **Create a New Repository:** On GitHub, create a new repository for your project.
4. **Initialize Git:** In your local project directory, initialize a Git repository: git init
5. **Stage and Commit Changes:** Add your code to the staging area: git add . Commit your changes with a descriptive message: git commit -m "Initial commit"
6. **Push to GitHub:** Connect your local repository to the remote GitHub repository:  
   Bash  
   git remote add origin <your-repository-url>  
   git push -u origin master

**THEORY EXERCISE: Source Code vs. Machine Code**

* **Source Code:** The human-readable code written by programmers in a programming language (e.g., Python, Java, C++).
* **Machine Code:** The low-level instructions that the computer's processor directly understands and executes. It is represented in binary format (0s and 1s).

**GitHub and Introductions**

**LAB EXERCISE: Create a GitHub Repository and Document Commit and Push**

1. **Create a GitHub Account:** If you don't have one, sign up at github.com.
2. **Create a New Repository:**
   * Go to your GitHub profile and click "New repository."
   * Give your repository a name (e.g., "my-first-repo").
   * Add a brief description (optional).
   * Initialize with a README file (recommended).
   * Click "Create repository."
3. **Create a Local Repository:**
   * On your computer, create a new folder for your project.
   * Open your terminal or command prompt and navigate to that folder.
   * Initialize a Git repository: git init
4. **Create a File:**
   * Create a new file (e.g., hello.txt) and add some content to it.
5. **Stage Changes:**
   * Add the file to the staging area: git add hello.txt
   * Alternatively, add all changes: git add .
6. **Commit Changes:**
   * Commit the changes with a descriptive message: git commit -m "Initial commit"
7. **Push to GitHub:**
   * Add the remote repository: git remote add origin <your-repository-url>
   * Push your local branch to the remote: git push -u origin main (or git push -u origin master)

**THEORY EXERCISE: Why is Version Control Important in Software Development?**

* **Tracking Changes:** Version control systems like Git allow you to track every change made to the code over time.
* **Collaboration:** Enables multiple developers to work on the same project simultaneously without conflicts.
* **Experimentation:** Allows for safe experimentation with code changes, as you can easily revert to previous versions if needed.
* **Backup and Recovery:** Provides a reliable backup of your code in case of accidental deletion or system failures.
* **Code Review:** Facilitates code reviews by making it easy to compare different versions of the code.

**Student Account in GitHub**

**LAB EXERCISE: Create a Student Account on GitHub and Collaborate on a Project**

1. **Create a GitHub Student Account:**
   * Follow the instructions in the previous section to create a GitHub account.
   * Verify your student status to get access to the GitHub Student Developer Pack.
2. **Collaborate on a Project:**
   * **Find a Partner:** Find a classmate to collaborate with.
   * **Create a Repository:** Create a new repository for your joint project.
   * **Add Collaborator:** Go to the repository's settings and add your classmate as a collaborator.
   * **Work Together:** Both of you can now clone the repository to your local machines, make changes, commit, and push your contributions.
   * **Resolve Conflicts:** If you both make changes to the same file, you'll need to resolve any conflicts using Git's merge tools.

**THEORY EXERCISE: Benefits of Using GitHub for Students**

* **Portfolio Building:** Showcase your projects and coding skills to potential employers.
* **Collaboration:** Learn to work effectively in teams and contribute to open-source projects.
* **Skill Development:** Develop essential skills like version control, code review, and project management.
* **Learning Resources:** Access a vast repository of open-source code and learn from others' projects.
* **Community:** Connect with other developers, find mentors, and participate in the open-source community.

**Types of Software**

**LAB EXERCISE: Classifying Software**

* **System Software:**
  + Operating System (e.g., Windows, macOS, Linux)
  + Antivirus Software
  + Device Drivers
* **Application Software:**
  + Web Browser (e.g., Chrome, Firefox)
  + Word Processor (e.g., Microsoft Word, Google Docs)
  + Media Player (e.g., VLC, Spotify)
  + Social Media Apps (e.g., Facebook, Instagram)
  + Games
* **Utility Software:**
  + File Compression Tools (e.g., WinZip, 7-Zip)
  + Disk Cleanup Tools
  + Backup Software
  + System Monitoring Tools

**THEORY EXERCISE: Open-Source vs. Proprietary Software**

* **Open-Source Software:**
  + Source code is publicly available and can be modified and distributed freely.
  + Often free to use, but may have licensing restrictions.
  + Examples: Linux, Android, Firefox, Python
* **Proprietary Software:**
  + Source code is kept secret and not publicly available.
  + Typically licensed for a fee.
  + Examples: Microsoft Windows, Adobe Photoshop, macOS

**GIT and GITHUB Training**

**LAB EXERCISE: GIT Tutorial: Cloning, Branching, and Merging**

1. **Clone a Repository:**
   * Choose an existing public repository on GitHub (e.g., a simple "Hello World" project).
   * Use the following command in your terminal to clone the repository:
   * Bash

git clone <repository\_url>

1. **Create a Branch:**
   * Navigate to the cloned repository directory.
   * Create a new branch:
   * Bash

git checkout -b <branch\_name>

* + (e.g., git checkout -b my-feature)

1. **Make Changes and Commit:**
   * Modify a file within the branch.
   * Stage and commit your changes:
   * Bash

git add <file\_name>

git commit -m "Your commit message"

1. **Switch Back to Main Branch:**
   * Checkout the main branch:
   * Bash

git checkout main

1. **Merge Branches:**
   * Merge your feature branch into the main branch:
   * Bash

git merge <branch\_name>

* + (e.g., git merge my-feature)

1. **Push Changes to Remote:**
   * Push the merged changes to the remote repository:
   * Bash

git push origin main

**THEORY EXERCISE: How does GIT improve collaboration in a software development team?**

* **Parallel Development:** Git allows multiple developers to work on different features or bug fixes simultaneously on separate branches without interfering with each other's work.
* **Code Review:** Git facilitates code reviews by allowing developers to create pull requests, which enable team members to review code changes before they are merged into the main codebase.
* **Version Control:** Git tracks every change made to the code, allowing developers to easily revert to previous versions if needed and understand the history of the project.
* **Conflict Resolution:** Git provides tools for resolving conflicts that may arise when multiple developers modify the same part of the code.
* **Remote Collaboration:** Git enables teams to collaborate effectively, even when they are geographically distributed.

**Application Software**

**LAB EXERCISE: Report on Types of Application Software and Productivity**

**Types of Application Software:**

* **Office Productivity Suites:** Word processors, spreadsheets, presentation software (e.g., Microsoft Office, Google Workspace) - **Improve productivity** by streamlining document creation, data analysis, and presentations.
* **Business Applications:** Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), Project Management software - **Improve productivity** by automating business processes, improving communication, and increasing efficiency.
* **Graphics and Multimedia:** Image editing, video editing, audio editing software (e.g., Adobe Photoshop, Premiere Pro, Audacity) - **Improve productivity** by enabling the creation of high-quality visual and audio content.
* **Education and Training:** Learning management systems (LMS), e-learning platforms, educational software - **Improve productivity** by facilitating learning and knowledge acquisition.
* **Specialized Applications:** Industry-specific software (e.g., medical software, engineering software) - **Improve productivity** by providing specialized tools and functionalities for specific domains.

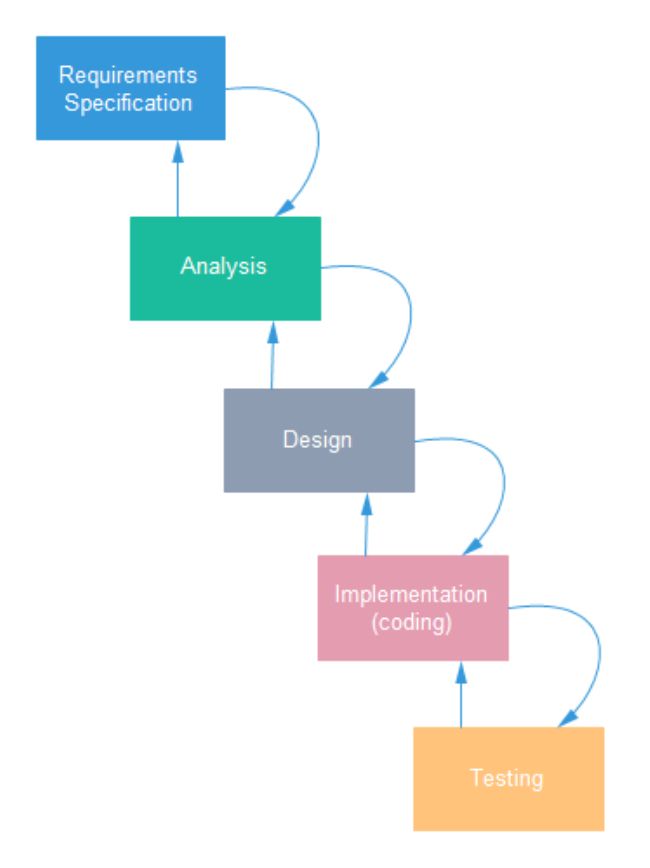
**THEORY EXERCISE: Role of Application Software in Businesses**

* **Automation:** Automating repetitive tasks, reducing manual effort, and increasing efficiency.
* **Communication and Collaboration:** Facilitating communication and collaboration among employees, clients, and partners.
* **Data Management and Analysis:** Helping businesses collect, store, analyze, and utilize data to make informed decisions.
* **Customer Service:** Improving customer service through tools like CRM and help desk software.
* **Innovation:** Enabling businesses to develop and implement new products and services.

**Software Development Process**

**LAB EXERCISE: Flowchart Representing SDLC**

* **Requirements Gathering:** Define the problem and gather user requirements.
* **Design:** Create a blueprint of the software system, including architecture, data structures, and algorithms.
* **Implementation:** Write the code based on the design.
* **Testing:** Test the software for bugs and errors.
* **Deployment:** Release the software to users.
* **Maintenance:** Maintain and update the software after deployment.



**THEORY EXERCISE: Main Stages of the Software Development Process**

1. **Requirements Gathering:** Understanding the needs and expectations of the users or stakeholders.
2. **Design:** Planning the architecture, components, and functionalities of the software.
3. **Implementation:** Writing the actual code for the software.
4. **Testing:** Identifying and fixing bugs and ensuring the software meets the requirements.
5. **Deployment:** Releasing the software to the users.
6. **Maintenance:** Ongoing support, updates, and bug fixes after deployment.

**Software Requirement**

**LAB EXERCISE: Requirement Specification for a Simple Library Management System**

**Functional Requirements:**

* **User Management:**
  + Register new users (librarians and members).
  + Login and logout functionality.
* **Book Management:**
  + Add new books to the library catalog.
  + Edit book details (title, author, ISBN, etc.).
  + Remove books from the catalog.
  + Search for books by title, author, or ISBN.
* **Member Management:**
  + Issue books to members.
  + Renew book loans.
  + Return books.
  + Track member borrowing history.
* **Reporting:**
  + Generate reports on book availability, overdue books, and member activity.

**Non-Functional Requirements:**

* **Performance:** The system should be able to handle a large number of users and transactions efficiently.
* **Security:** User data and library data should be protected from unauthorized access.
* **Usability:** The system should be easy to use and navigate for both librarians and members.
* **Reliability:** The system should be available and operational most of the time.
* **Maintainability:** The system should be easy to maintain and update.

**THEORY EXERCISE: Why is the Requirement Analysis Phase Critical in Software Development?**

* **Ensures User Satisfaction:** By clearly understanding user needs, the development team can build software that meets those needs.
* **Reduces Costs and Time:** Identifying and addressing requirements early in the process can prevent costly rework later.
* **Improves Quality:** Clear and well-defined requirements lead to a better-designed and more reliable software product.
* **Minimizes Risks:** Identifying potential risks and challenges early on allows for proactive mitigation strategies.
* **Facilitates Communication:** A well-documented set of requirements serves as a common understanding between stakeholders, developers, and testers.

**Software Analysis**

**LAB EXERCISE: Functional Analysis for an Online Shopping System**

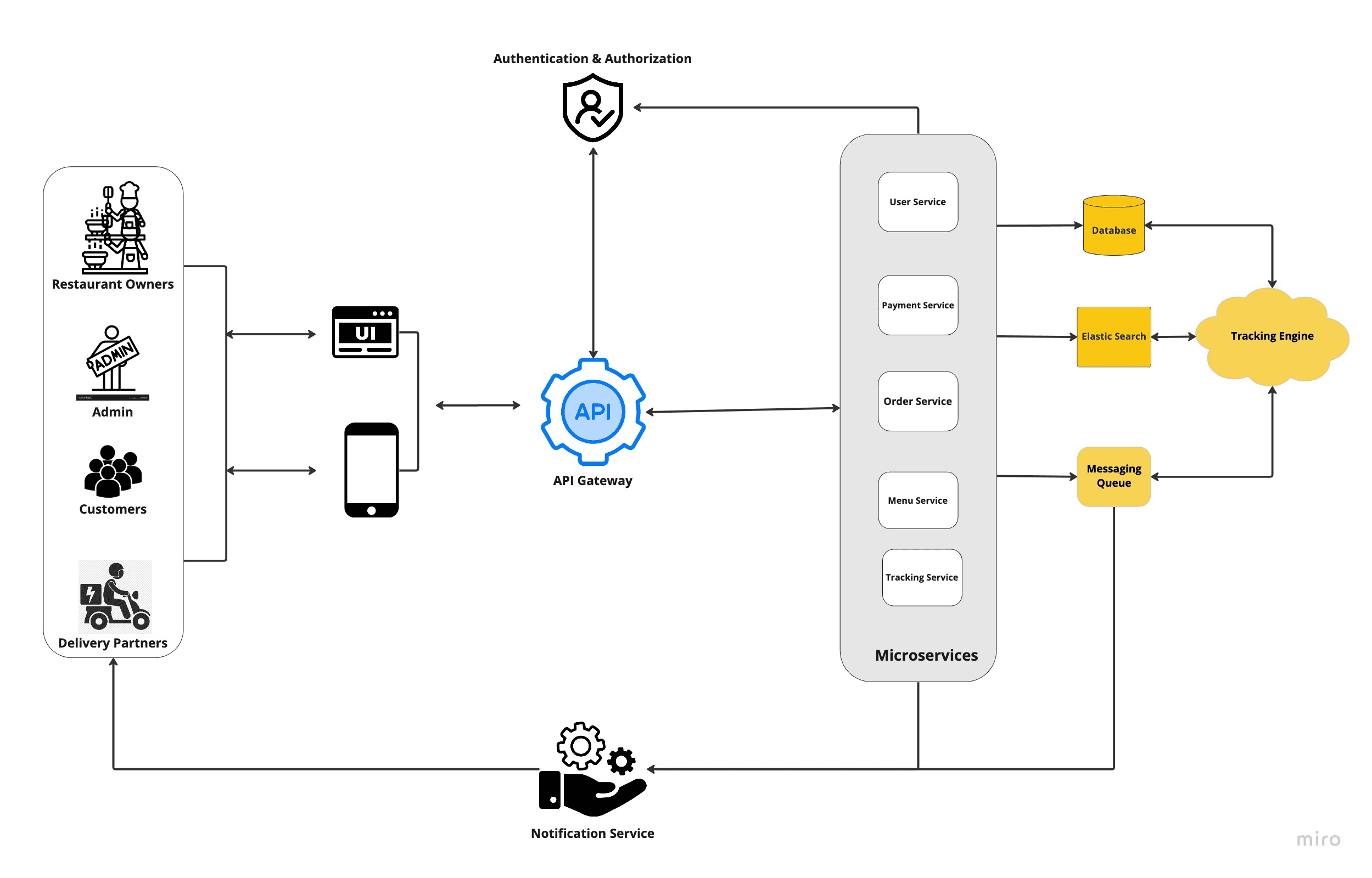
* **Use Case Diagram:** Visual representation of the interactions between users (customers, administrators) and the system.
* **Activity Diagram:** Flow of activities involved in a specific process, such as placing an order or managing products.
* **Sequence Diagram:** Interaction between different objects or components in the system, showing the sequence of messages exchanged.

**THEORY EXERCISE: Role of Software Analysis in the Development Process**

* **Understanding Requirements:** Analyzing user needs and translating them into functional and non-functional requirements.
* **System Design:** Identifying the components of the system, their interactions, and how they will work together.
* **Risk Identification:** Identifying potential risks and challenges that may arise during development.
* **Decision Making:** Providing the information needed to make informed decisions about the design and implementation of the software.
* **Quality Assurance:** Ensuring that the software meets the specified requirements and is of high quality.

**System Design**

**LAB EXERCISE: Design a basic system architecture for a food delivery app.**

****

**Key Components:**

1. **User App:**
   * **Functionality:** User registration/login, restaurant browsing, menu viewing, order placement, order tracking, payment integration, reviews and ratings.
   * **Technologies:** Mobile-first design, responsive UI, native or cross-platform development (e.g., React Native, Flutter).2
2. **Restaurant App:**
   * **Functionality:** Order management (accept/reject, track status), menu updates, view order history, manage profile.3
   * **Technologies:** Similar to User App, potentially with a simplified UI.
3. **Delivery Partner App:**
   * **Functionality:** Order assignment, real-time navigation, order pickup/delivery updates, earnings tracking.
   * **Technologies:** GPS integration, real-time location tracking, efficient routing algorithms.
4. **Central Server:**
   * **Functionality:** Handle user authentication, order processing, payment gateway integration, notifications, data storage, and API endpoints for all apps.
   * **Technologies:** Cloud-based infrastructure (AWS, Azure, GCP), microservices architecture, robust database (e.g., PostgreSQL, MongoDB).
5. **Database:**
   * **Functionality:** Store user data, restaurant information, menus, orders, delivery partner details, payment history, and more.
   * **Technologies:** Relational database (PostgreSQL) or NoSQL database (MongoDB) depending on data structure and query needs.
6. **Payment Gateway:**
   * **Functionality:** Integrate with third-party payment providers (e.g., Stripe, PayPal) for secure and seamless transactions.
7. **Cloud Services:**
   * **Functionality:** Utilize cloud services for storage, compute power, and scalability.
   * **Technologies:** Cloud storage (AWS S3, Azure Blob Storage), cloud functions (AWS Lambda, Azure Functions).

**Diagram:**

**THEORY EXERCISE: What are the key elements of system design?**

* **Requirements Gathering:** Understanding user needs and system goals.
* **Functional Decomposition:** Breaking down the system into smaller, manageable components.
* **Component Design:** Designing the individual components and their interactions.
* **Data Modeling:** Designing the database schema to store and retrieve data efficiently.
* **Scalability and Performance:** Ensuring the system can handle increasing loads and provide a good user experience.
* **Security:** Implementing security measures to protect user data and prevent unauthorized access.
* **Deployment and Maintenance:** Planning for deployment, monitoring, and maintenance of the system.

**Software Testing**

**LAB EXERCISE: Develop test cases for a simple calculator program.**

| Test Case ID | Description | Input | Expected Output | Actual Output | Pass/Fail |
| --- | --- | --- | --- | --- | --- |
| TC\_01 | Addition | 2 + 3 | 5 | 5 | Pass |
| TC\_02 | Subtraction | 5 - 2 | 3 | 3 | Pass |
| TC\_03 | Multiplication | 2 \* 3 | 6 | 6 | Pass |
| TC\_04 | Division | 6 / 2 | 3 | 3 | Pass |
| TC\_05 | Division by Zero | 5 / 0 | Error | Error | Pass |
| TC\_06 | Large Numbers | 999999 + 1 | 1000000 | 1000000 | Pass |
| TC\_07 | Negative Numbers | -5 + 2 | -3 | -3 | Pass |
| TC\_08 | Decimal Numbers | 2.5 + 3.5 | 6.0 | 6.0 | Pass |

**THEORY EXERCISE: Why is software testing important?**

* **Quality Assurance:** Ensures the software meets the specified requirements and functions as expected.
* **Bug Prevention:** Identifies and fixes defects early in the development process, reducing the cost of fixing them later.
* **User Satisfaction:** Delivers a high-quality product that meets user expectations and provides a positive experience.
* **Risk Mitigation:** Reduces the risk of system failures and data loss.
* **Improved Reliability:** Increases the reliability and stability of the software.

**Maintenance**

**LAB EXERCISE: Document a real-world case where a software application required critical maintenance.**

* **Example: Social Media Platform**
  + **Scenario:** A major social media platform experiences a widespread outage due to a critical bug in the core infrastructure.
  + **Maintenance Activities:**
    - **Emergency Bug Fixes:** Quickly identify and fix the bug to restore service.12
    - **Root Cause Analysis:** Investigate the root cause of the bug to prevent future occurrences.13
    - **System Upgrades:** Upgrade infrastructure components to improve stability and prevent similar issues.
    - **Communication:** Communicate with users about the outage and the steps being taken to resolve it.

**THEORY EXERCISE: What types of software maintenance are there?**

1. **Corrective Maintenance:** Fixing bugs and errors in the software.
2. **Adaptive Maintenance:** Modifying the software to adapt to changes in the environment or user requirements.
3. **Perfective Maintenance:** Improving the performance, usability, or maintainability of the software.
4. **Preventive Maintenance:** Taking proactive steps to prevent future problems, such as code refactoring and performance optimization.

**Development**

**THEORY EXERCISE: What are the key differences between web and desktop applications?**

| Feature | Web Application | Desktop Application |
| --- | --- | --- |
| **Platform Independence** | Accessible from any device with a web browser | Requires installation on a specific operating system |
| **Deployment** | Easier to deploy and update | Requires installation on each user's device |
| **Offline Access** | Limited or no offline functionality | Can work offline |
| **Performance** | Can be affected by network speed and latency | Generally faster performance |
| **User Interface** | Typically uses web technologies (HTML, CSS, JavaScript) | Uses native operating system UI elements |
| **Security** | More vulnerable to security threats | Can have better security controls |

**THEORY EXERCISE: What are the advantages of using web applications over desktop applications?**

* **Platform Independence:** Accessible from any device with a web browser, regardless of operating system.
* **Easier Deployment and Updates:** Updates can be deployed centrally and automatically to all users.
* **Cost-Effectiveness:** Lower development and maintenance costs compared to desktop applications.
* **Accessibility:** Easier to access and use from anywhere with an internet connection.
* **Collaboration:** Facilitates collaboration and sharing of information among users.

**Designing**

**THEORY EXERCISE: What role does UI/UX design play in application development?**

UI/UX design plays a crucial role in application development by focusing on the user's experience and interaction with the application.

* **User Experience (UX) Design:**
  + **Focus:** The overall experience a user has while interacting with the application.
  + **Key Aspects:** Usability, accessibility, findability, credibility, value, and desirability.
  + **Activities:** User research, information architecture, interaction design, usability testing.
* **User Interface (UI) Design:**
  + **Focus:** The visual appearance and style of the application's interface.
  + **Key Aspects:** Visual design, typography, color schemes, layout, and interaction elements.
  + **Activities:** Wireframing, prototyping, visual design, and testing.

**Benefits of Strong UI/UX Design:**

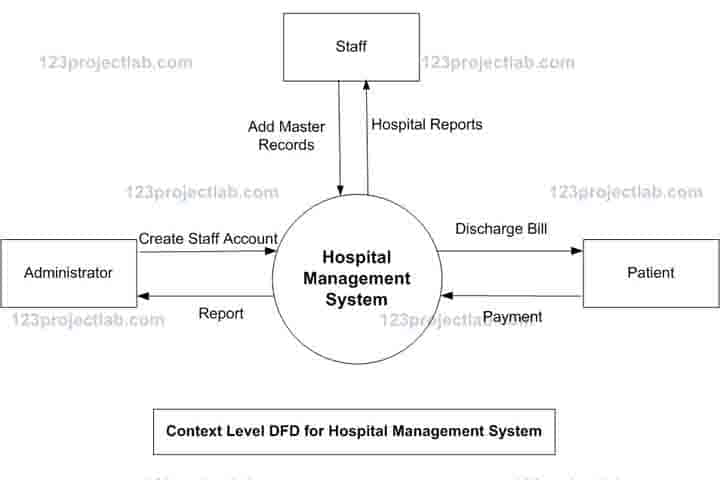
* **Increased User Satisfaction:** A well-designed application is more enjoyable and easier to use, leading to higher user satisfaction.
* **Improved User Engagement:** Engaging and intuitive interfaces encourage users to spend more time with the application.
* **Enhanced Brand Perception:** A visually appealing and user-friendly application reflects positively on the brand.
* **Higher Conversion Rates:** Clear calls to action and a smooth user experience can lead to increased conversions (e.g., purchases, sign-ups).
* **Reduced Support Costs:** A well-designed application is easier to use, reducing the need for user support.

**THEORY EXERCISE: What are the differences between native and hybrid mobile apps?**

| Feature | Native Apps | Hybrid Apps |
| --- | --- | --- |
| **Development** | Built using platform-specific languages (e.g., Swift for iOS, Java/Kotlin for Android) | Built using web technologies (HTML, CSS, JavaScript) and wrapped in a native container |
| **Performance** | Generally better performance, faster loading times, smoother animations | Can have performance limitations, especially for demanding tasks |
| **User Experience** | Seamlessly integrate with device hardware and operating system features | May have some limitations in accessing device features |
| **Development Time** | Often require separate codebases for each platform | Can be developed faster as they use a single codebase for multiple platforms |
| **User Interface** | Typically provide a more native-like appearance and feel | Can sometimes have a less polished look and feel compared to native apps |
| **Examples** | Instagram, Facebook, Uber | Many popular apps like Twitter, Instagram, and Facebook also have hybrid components |

**LAB EXERCISE: Create a DFD for a Hospital Management System.**

* **Context:** A hospital management system handles patient records, appointments, and billing.



**Data Flow Diagram (DFD) Levels:**

1. **Level 0 (Context Diagram):**
   * **External Entities:** Patients, Doctors, Staff, Insurance Companies.
   * **Process:** Hospital Management System.
   * **Data Flows:** Patient information, Appointments, Billing, Insurance claims.
2. **Level 1 (First-Level DFD):**
   * **Processes:** Patient Registration, Appointment Scheduling, Doctor Management, Billing & Payments, Inventory Management, Reports.
   * **Data Stores:** Patient Records, Doctor Records, Appointments, Bills, Inventory.
3. **Subsequent Levels (Detailed DFDs):** Each of the Level 1 processes can be further broken down into more detailed DFDs as needed.

**THEORY EXERCISE: What is the significance of DFDs in system analysis?**

* **Visual Representation:** DFDs provide a clear and visual representation of the system's data flows and processes.
* **System Understanding:** Help analysts understand the system's functionality and identify key components.
* **Requirements Gathering:** Assist in gathering and documenting system requirements.
* **Communication:** Facilitate communication between stakeholders by providing a common understanding of the system.
* **System Design:** Guide the design of the system's data structures and processes.
* **Testing:** Help identify potential problems and areas for testing.

**LAB EXERCISE: Build a simple desktop calculator application using a GUI library.**

* **Choose a GUI Library:**
  + **Python:** Tkinter (built-in), PyQt, Kivy
  + **Java:** JavaFX, Swing
  + **C#:** Windows Forms, WPF
* **Key Components:**
  + **GUI Elements:** Create buttons for numbers (0-9), operators (+, -, \*, /, =), and a display for the input and result.
  + **Event Handling:** Implement event handlers for button clicks (e.g., when a number or operator button is clicked).
  + **Logic:** Implement the logic for calculations (e.g., handling user input, performing calculations, displaying the result).

**THEORY EXERCISE: What are the pros and cons of desktop applications compared to web applications?**

| Feature | Desktop Applications | Web Applications |
| --- | --- | --- |
| **Performance** | Generally better performance, faster loading times | Can be affected by network speed and latency |
| **Offline Access** | Can work offline | Limited or no offline functionality |
| **User Interface** | Often have a more native look and feel | Can be accessed from any device with a web browser |
| **Deployment** | Requires installation on each user's device | Easier to deploy and update |
| **Security** | Can have better security controls | More vulnerable to security threats |
| **Resources** | May require more resources (storage, processing power) | Typically have lower resource requirements |

**LAB EXERCISE: Draw a flowchart representing the logic of a basic online registration system.**

1. **Start**
2. **Display registration form:**
   * Fields: Username, password, email, etc.
3. **User enters information**
4. **Validate input:**
   * Check for empty fields, valid email address, strong password, etc.
5. **If input is valid:**
   * Proceed to next step
6. **If input is invalid:**
   * Display error messages
   * Go back to step 2
7. **Create user account:**
   * Store user information in the database
8. **Send verification email:**
   * Send an email with a verification link to the user
9. **User verifies email:**
   * Click on the verification link in the email
10. **Account activated:**
    * User account is successfully activated
11. **Login:**
    * Redirect the user to the login page
12. **End**

**THEORY EXERCISE: How do flowcharts help in programming and system design?**

* **Visual Representation:** Provide a clear and visual representation of the logic and flow of a program or system.
* **Problem-solving:** Help break down complex problems into smaller, more manageable steps.
* **Design and Development:** Guide the design and development process by providing a blueprint for the implementation.
* **Communication:** Facilitate communication among team members by providing a common understanding of the system's logic.
* **Testing:** Help identify potential issues and test cases.
* **Debugging:** Assist in debugging by providing a visual representation of the program's execution flow.