**Module - 3**

1. What is RDBMS ?

* **RDBMS** (Relational Database Management System) is a type of database management system (DBMS) that stores data in a structured format using rows and columns, typically organized into tables.

### **Key Characteristics of RDBMS:**

1. **Tables (Relations)**: Data is stored in tables (also called relations), where each table consists of rows (records) and columns (fields).
2. **Primary Key**: Each table has a primary key that uniquely identifies each record in that table. No two rows in a table can have the same primary key value.
3. **Foreign Key**: A foreign key is a field (or a combination of fields) in one table that uniquely identifies a row of another table. It creates relationships between tables.
4. **Normalization**: The process of organizing data in the database to reduce redundancy and dependency by dividing large tables into smaller, related tables.
5. **ACID Properties**: RDBMS ensures the integrity of transactions through the ACID properties—**Atomicity, Consistency, Isolation, and Durability**.
6. **SQL (Structured Query Language)**: RDBMS uses SQL as its primary language for querying and managing data. SQL allows for operations like SELECT, INSERT, UPDATE, DELETE, and more complex queries involving joins, subqueries, and aggregates.

### **Examples of RDBMS:**

1. MySQL
2. PostgreSQL
3. Oracle Database
4. Microsoft SQL Server
5. SQLite

* **Advantages of RDBMS :**

1. Data Integrity: Ensures the correctness and reliability of data through constraints like primary keys, foreign keys, and unique constraints.
2. Flexibility: Supports complex queries and transactions.
3. Scalability: Can handle large volumes of data and transactions efficiently.
4. Consistency: Maintains consistent data even in multi-user environments through transaction management.

* **Use Cases:** RDBMS systems are widely used for applications like financial systems, enterprise resource planning (ERP), customer relationship management (CRM), and many others where data integrity, consistency, and complex querying are essential.

2. What is SQL ?

* **SQL** (Structured Query Language) is a standardized programming language used to manage and manipulate relational databases. SQL is essential for interacting with databases to perform tasks such as querying, updating, inserting, and deleting data, as well as managing the structure of the database itself (e.g., creating tables, defining relationships).

### **Types of SQL:**

SQL is a broad language, and its functionality is divided into several types of commands:

1. **DDL (Data Definition Language)**: Deals with the structure of the database.
   * CREATE, ALTER, DROP, TRUNCATE
2. **DML (Data Manipulation Language)**: Deals with the data within the tables.
   * SELECT, INSERT, UPDATE, DELETE
3. **DCL (Data Control Language)**: Deals with user permissions and access control.
   * GRANT, REVOKE
4. **TCL (Transaction Control Language)**: Deals with transaction management.
   * COMMIT, ROLLBACK, SAVEPOINT

**Advantages of SQL:**

* **Ease of Use**: SQL is widely regarded as a user-friendly and declarative language. Users describe *what* data they need without worrying about how it will be retrieved.
* **Standardized**: SQL is a standardized language, making it compatible across many database systems, although some variations (e.g., MySQL, PostgreSQL, SQL Server) exist.
* **Powerful and Flexible**: SQL can handle complex queries, joins, aggregations, and transactions

3. Write SQL Commands ?

**Common SQL Commands:**

1. **SELECT: Retrieves data from a table.  
   sql  
      
   SELECT \* FROM employees;**

**This query fetches all columns and rows from the "employees" table.**

1. **INSERT: Adds new data into a table.  
   sql  
      
   INSERT INTO employees (name, position, salary)**

**VALUES ('John Doe', 'Manager', 60000);**

1. **UPDATE: Modifies existing data in a table.  
   sql  
      
   UPDATE employees**

**SET salary = 65000**

**WHERE name = 'John Doe';**

1. **DELETE: Removes data from a table.  
   sql  
      
   DELETE FROM employees**

**WHERE name = 'John Doe';**

1. **CREATE TABLE: Defines a new table in the database.  
   sql  
      
   CREATE TABLE employees (**

**id INT PRIMARY KEY,**

**name VARCHAR(100),**

**position VARCHAR(50),**

**salary DECIMAL(10, 2)**

**);**

1. **ALTER TABLE: Modifies an existing table (e.g., adding or removing columns).  
   sql  
    ALTER TABLE employees**

**ADD COLUMN hire\_date DATE;**

1. **DROP TABLE: Deletes an entire table and its data.  
   sql  
      
   DROP TABLE employees;**

4. What is join ?

* In SQL, a **JOIN** is a powerful operation used to combine data from two or more tables based on a related column between them. This allows you to retrieve data from multiple tables in a single query, which is essential for working with relational databases where data is often split across several related tables.

5. Write types of joins.

### **Types of SQL JOINs:**

There are several types of joins, each serving different purposes depending on how you want to match the data between the tables:

#### **1. INNER JOIN**

An **INNER JOIN** returns only the rows where there is a match in both tables. If there is no match between the tables, those rows are not included in the result set.

**Example:**

sql

* SELECT employees.name, departments.department\_name
* FROM employees
* INNER JOIN departments
* ON employees.department\_id = departments.id;

This query returns the employee names and their respective department names, but only for employees who are assigned to a department (i.e., where there is a match on department\_id).

#### **2. LEFT JOIN (or LEFT OUTER JOIN)**

A **LEFT JOIN** returns all rows from the **left table** (the first table in the query), along with matching rows from the **right table** (the second table). If there is no match in the right table, the result will contain NULL values for columns from the right table.

**Example:**

sql

* SELECT employees.name, departments.department\_name
* FROM employees
* LEFT JOIN departments
* ON employees.department\_id = departments.id;

This query will return all employees, along with their department names. If an employee is not assigned to any department (i.e., no matching department), the result will show NULL in the department\_name column for that employee.

#### **3. RIGHT JOIN (or RIGHT OUTER JOIN)**

A **RIGHT JOIN** is the opposite of a **LEFT JOIN**. It returns all rows from the **right table**, and the matching rows from the **left table**. If there is no match in the left table, the result will contain NULL values for columns from the left table.

**Example:**

sql

* SELECT employees.name, departments.department\_name
* FROM employees
* RIGHT JOIN departments
* ON employees.department\_id = departments.id;

This query will return all departments and their employees. If a department does not have any employees, the result will show NULL for the employee.name field for that department.

#### **4. FULL JOIN (or FULL OUTER JOIN)**

A **FULL JOIN** (or **FULL OUTER JOIN**) returns all rows when there is a match in either the left or right table. If there is no match, the result will contain NULL values for the columns from the table that does not have a match.

**Example:**

sql

* SELECT employees.name, departments.department\_name
* FROM employees
* FULL JOIN departments
* ON employees.department\_id = departments.id;

This query will return all employees and all departments. If an employee does not belong to a department, the department column will be NULL. If a department has no employees, the employee column will be NULL.

#### **5. CROSS JOIN**

A **CROSS JOIN** returns the Cartesian product of both tables, meaning it will return all possible combinations of rows between the two tables. This can result in a very large result set, especially if the tables are large.

**Example:**

sql

* SELECT employees.name, departments.department\_name
* FROM employees
* CROSS JOIN departments;

This query will return every possible combination of employees and departments, regardless of whether they are related.

#### **6. SELF JOIN**

A **SELF JOIN** is a join where a table is joined with itself. This can be useful when you need to compare rows within the same table.

**Example:**

sql

* SELECT e1.name AS Employee, e2.name AS Manager
* FROM employees e1
* LEFT JOIN employees e2
* ON e1.manager\_id = e2.id;

In this case, the employees table is joined with itself, where e1 represents employees and e2 represents their managers. If an employee does not have a manager, NULL will appear in the Manager column.

### **How Joins Work:**

Joins operate based on a **matching condition**, usually a key that exists in both tables. The most common join condition is the equality of a column in one table with a column in another (i.e., using = in the ON clause).

* **Example Matching Condition:**
  + In the INNER JOIN, you typically match the foreign key in one table (e.g., department\_id in employees) with the primary key of the other table (e.g., id in departments).
* sql  
  Copy code  
  SELECT employees.name, departments.department\_name
* FROM employees
* INNER JOIN departments
* ON employees.department\_id = departments.id;

### **Summary:**

* **JOINs** combine data from multiple tables based on related columns.
* **INNER JOIN**: Only returns rows with matching data in both tables.
* **LEFT JOIN**: Returns all rows from the left table, matching rows from the right table (or NULL if no match).
* **RIGHT JOIN**: Returns all rows from the right table, matching rows from the left table (or NULL if no match).
* **FULL JOIN**: Returns all rows when there is a match in either the left or right table.
* **CROSS JOIN**: Returns all combinations of rows between the two tables.
* **SELF JOIN**: Joins a table with itself.

Joins are essential for efficiently retrieving and combining related data stored in different tables within a relational database.

6. How many constraint and describes itself.

In SQL, **constraints** are used to define rules for the data in a table. They help maintain data integrity and enforce the relationships between different data points. There are several types of constraints, each serving a specific purpose to ensure data accuracy, consistency, and validity.

### **1. PRIMARY KEY**

* **Description**: A primary key uniquely identifies each record in a table. It cannot have NULL values and must contain unique values for each row. Each table can have only one primary key.
* **Purpose**: To ensure that each row in a table can be uniquely identified.
* **Example**:  
  sql  
    
  CREATE TABLE employees (
* id INT PRIMARY KEY,
* name VARCHAR(100),
* position VARCHAR(50)
* );

### **2. FOREIGN KEY**

* **Description**: A foreign key is a column (or combination of columns) that links one table to another. It establishes a relationship between the data in two tables. The values in the foreign key column(s) must match the primary key in the referenced table (or be NULL).
* **Purpose**: To ensure referential integrity between two related tables.
* **Example**:  
  sql  
    
  CREATE TABLE departments (
* id INT PRIMARY KEY,
* department\_name VARCHAR(100)
* );
* CREATE TABLE employees (
* id INT PRIMARY KEY,
* name VARCHAR(100),
* department\_id INT,
* FOREIGN KEY (department\_id) REFERENCES departments(id)
* );

### **3. UNIQUE**

* **Description**: The UNIQUE constraint ensures that all values in a column (or a set of columns) are distinct across the table. It allows NULL values unless explicitly specified otherwise, but the NULL values must still be unique.
* **Purpose**: To enforce uniqueness for values in a column, ensuring no duplicates.
* **Example**:  
  sql  
    
  CREATE TABLE users (
* email VARCHAR(255) UNIQUE,
* username VARCHAR(100)
* );

### **4. NOT NULL**

* **Description**: The NOT NULL constraint ensures that a column cannot contain NULL values. This constraint is often used when it is essential that a column always has a value.
* **Purpose**: To ensure that a column always contains valid data.
* **Example**:  
  sql  
  Copy code  
  CREATE TABLE orders (
* id INT PRIMARY KEY,
* order\_date DATE NOT NULL,
* total\_amount DECIMAL(10, 2) NOT NULL
* );

### **5. CHECK**

* **Description**: The CHECK constraint allows you to define a condition (a boolean expression) that must be true for each row in the table. If a row does not satisfy the condition, it cannot be inserted or updated in the table.
* **Purpose**: To ensure that data entered into a column meets specific conditions (e.g., age must be above 18).
* **Example**:  
  sql  
    
  CREATE TABLE employees (
* id INT PRIMARY KEY,
* name VARCHAR(100),
* age INT,
* CHECK (age >= 18)
* );

### **6. DEFAULT**

* **Description**: The DEFAULT constraint provides a default value for a column when no value is specified during an insert. If a value is not explicitly provided, the default value is used.
* **Purpose**: To ensure that a column has a default value when no data is provided.
* **Example**:  
  sql  
    
  CREATE TABLE orders (
* id INT PRIMARY KEY,
* order\_date DATE DEFAULT CURRENT\_DATE,
* status VARCHAR(50) DEFAULT 'Pending'
* );

### **7. INDEX**

* **Description**: While technically not a constraint, an INDEX is a database object that can be created on one or more columns of a table to speed up the retrieval of rows. An index does not enforce data integrity, but it improves query performance.
* **Purpose**: To improve the performance of SELECT queries that search or sort by specific columns.
* **Example**:  
  sql  
  Copy code  
  CREATE INDEX idx\_employee\_name ON employees(name);

### **8. AUTO\_INCREMENT (in MySQL) / SERIAL (in PostgreSQL)**

* **Description**: This is a special constraint that automatically generates a unique numeric value for a column, typically used for primary key fields. In MySQL, this is defined using AUTO\_INCREMENT, while in PostgreSQL, the equivalent feature is implemented with SERIAL.
* **Purpose**: To automatically generate unique values for a column, often used for primary key fields.
* **Example** (MySQL):  
  sql  
    
  CREATE TABLE employees (
* id INT AUTO\_INCREMENT PRIMARY KEY,
* name VARCHAR(100),
* position VARCHAR(50)
* );

### **Summary of SQL Constraints:**

| **Constraint** | **Description** |
| --- | --- |
| **PRIMARY KEY** | Uniquely identifies each record in a table. Cannot contain NULL values. |
| **FOREIGN KEY** | Establishes a relationship between two tables and ensures referential integrity. |
| **UNIQUE** | Ensures all values in a column or set of columns are unique (can contain NULL). |
| **NOT NULL** | Ensures that a column does not accept NULL values. |
| **CHECK** | Defines a condition that must be true for the rows in the table. |
| **DEFAULT** | Specifies a default value for a column when no value is provided. |
| **INDEX** | Improves query performance by creating an index on one or more columns. |
| **AUTO\_INCREMENT** | Automatically generates unique numeric values for a column (MySQL) or SERIAL (PostgreSQL). |

### **Example of Table with Multiple Constraints:**

sql

* CREATE TABLE employees (
* id INT PRIMARY KEY AUTO\_INCREMENT, -- PRIMARY KEY and AUTO\_INCREMENT
* name VARCHAR(100) NOT NULL, -- NOT NULL constraint
* email VARCHAR(255) UNIQUE, -- UNIQUE constraint
* age INT CHECK (age >= 18), -- CHECK constraint
* department\_id INT, -- Foreign key (to be added below)
* hire\_date DATE DEFAULT CURRENT\_DATE, -- DEFAULT constraint
* FOREIGN KEY (department\_id) REFERENCES departments(id) -- FOREIGN KEY constraint
* );

In this example, the employees table has several constraints:

* **PRIMARY KEY** on id.
* **AUTO\_INCREMENT** on id (for auto-incrementing values).
* **NOT NULL** on name.
* **UNIQUE** on email.
* **CHECK** on age to ensure it's greater than or equal to 18.
* **DEFAULT** on hire\_date to use the current date if no date is provided.
* **FOREIGN KEY** linking department\_id to the departments table.

### **Conclusion:**

Constraints in SQL are vital for ensuring data integrity, consistency, and validity in relational databases. Each constraint type serves a specific purpose, from ensuring uniqueness to enforcing referential integrity, helping to maintain a well-structured and reliable database.

7. Difference between RDBMS vs DBMS.

| **Points**  \*Data Structure  \*Data Integrity  \*Normalization  \*ACID Properties  \*Relationship support  \*Query language   |  | | --- |   \* Concurrency control  \*Scalability  \*Examples   |  | | --- | | **DBMS**  Can use hierarchical, network, or flat files.  No enforcement of data integrity rules.  Not necessarily supported.   | May not support ACID transactions.  No mechanism for defining relationships. | | --- |   May not have a standard query language.   * Limited or none.  |  | | --- |  * Typically suited for small scale applications. * Microsoft access, IMS, IDS. | **RDBMS**  Data is stored in tables (rows and columns).  Ensures data integrity through constraints (primary keys, foreign keys).  Support data normalization.  Supports ACID properties for transactions.  Supports relationships between tables using foreign keys.  Uses SQL.   |  | | --- |  * Advanced concurrency control mechanism. * Scalable for large applications and high traffic. * Mysql, Oracle, PostgreSQL, SQL server, SQLite. |
| --- | --- | --- | --- | --- | --- | --- | --- |

8. What is API testing ?

* **API testing** involves sending requests to an API (Application Programming Interface) and verifying the response to ensure that the API functions as expected. It checks the API's functionality, reliability, performance, and security by validating responses against various test cases.

### **Common Tools for API Testing:**

* **Postman**: A popular tool for manually testing APIs by sending requests and viewing responses.
* **SoapUI**: Primarily used for testing SOAP and REST APIs with a focus on functional and security testing.
* **JMeter**: Mainly used for performance testing, simulating loads on APIs.
* **Rest Assured**: A Java library used for writing automated API tests for REST services.

### **Why is API Testing Important?**

* **Reliability**: Ensures APIs work as expected, especially when they are critical for system functionality.
* **Efficiency**: Helps detect issues early, reducing potential bugs in the final product.
* **Security**: Identifies vulnerabilities that could lead to data breaches or unauthorized access.
* **Performance**: Ensures APIs can handle heavy traffic and scale appropriately without affecting user experience.

In short, API testing is essential to ensure that APIs are functional, secure, and reliable, helping to maintain the overall quality of an application or service.

9. Types of API testing ?

### **1. Functional Testing**

Functional testing is the most common type of API testing and ensures that the API works as expected according to the defined specifications. This type of testing verifies that each endpoint and its parameters produce the correct output.

#### **Key Focus Areas:**

* **Input/Output Validation**: Ensuring that API requests return the expected results.
* **Response Codes**: Validating that the correct HTTP status codes (e.g., 200, 201, 400) are returned.
* **Request Validation**: Checking that the API handles and processes requests accurately, including handling required parameters and body data.

#### **Example:**

* Test if a GET /users/{id} endpoint correctly returns the user data when the ID exists.

**2. Performance Testing**

Performance testing evaluates how well an API performs under different load conditions. This includes testing for speed, scalability, and response time under various stress levels.

#### **Key Focus Areas:**

* **Load Testing**: Testing how the API behaves under expected traffic or load, measuring response times, and ensuring it can handle the load without performance degradation.
* **Stress Testing**: Pushing the API beyond its limits to see how it responds under extreme conditions (e.g., high volume of requests or data).
* **Scalability Testing**: Evaluating how well the API can scale with increasing requests or user load.
* **Latency Testing**: Measuring the response times of the API under normal and peak conditions.

#### **Example:**

* Simulating 1,000 concurrent users calling the POST /orders endpoint to check if the server handles the load efficiently without crashing.

**3. Security Testing**

Security testing is critical for identifying vulnerabilities and ensuring that the API is protected from malicious attacks, unauthorized access, and data breaches.

#### **Key Focus Areas:**

* **Authentication**: Verifying that only authorized users or applications can access the API (e.g., OAuth tokens, API keys, JWT).
* **Authorization**: Ensuring users can only access resources they are authorized to.
* **Data Encryption**: Verifying that sensitive data is encrypted in transit (e.g., HTTPS, TLS) and, when necessary, in storage.
* **Injection Attacks**: Ensuring the API is protected against SQL injection, Cross-Site Scripting (XSS), and other common attack vectors.
* **Rate Limiting**: Ensuring the API has proper rate limiting and protection against DoS (Denial of Service) attacks.

#### **Example:**

* Testing an API endpoint to check whether a user without proper authorization can access resources that should be restricted.

**4. Reliability/Failure Testing**

Reliability testing ensures that the API performs correctly and consistently over time. It checks whether the API can recover gracefully from errors or failures.

#### **Key Focus Areas:**

* **Error Handling**: Testing how the API handles unexpected inputs or conditions, ensuring it returns proper error codes (e.g., 400 Bad Request, 500 Internal Server Error).
* **Failover Testing**: Ensuring the API can handle failures (e.g., database connection failures) without crashing and can gracefully recover.
* **Timeouts and Retries**: Testing how the API behaves when it faces network delays, server timeouts, or when a request needs to be retried.

#### **Example:**

* Sending an invalid request to an endpoint and verifying that the API returns a specific error code (e.g., 404 Not Found).

**5. Compatibility Testing**

Compatibility testing checks how well the API interacts with different devices, platforms, and browsers, ensuring the API functions across different environments.

#### **Key Focus Areas:**

* **Cross-Platform Compatibility**: Verifying that the API works across different operating systems (Windows, Linux, macOS) or mobile platforms (iOS, Android).
* **Browser Compatibility**: Ensuring that the API behaves correctly when consumed by different web browsers (e.g., Chrome, Firefox, Safari).
* **Version Compatibility**: Ensuring that new versions of the API are backward compatible with older versions.

#### **Example:**

* Testing a REST API on multiple devices (desktop, tablet, smartphone) to ensure consistent behavior.

**6. Boundary Testing (Edge Case Testing)**

Boundary testing focuses on verifying the behavior of the API under extreme conditions or edge cases. This helps identify unexpected issues in scenarios where the system is pushed to its limits.

#### **Key Focus Areas:**

* **Boundary Value Testing**: Testing the API with boundary values like maximum and minimum input sizes, large payloads, or long strings.
* **Input Range Testing**: Verifying that the API handles values at the edges of valid input ranges (e.g., an integer between 1 and 100, testing 0 and 100).

#### **Example:**

* Testing an API endpoint that accepts an integer within the range of 1 to 100 to ensure it handles values of 0, 1, 100, and 101 correctly.

**7. Validation Testing**

Validation testing ensures that the API returns the expected output according to the business logic. It validates that the system meets the functional requirements and that the API responses match the expected outputs.

#### **Key Focus Areas:**

* **Data Accuracy**: Ensuring that the returned data is accurate and matches the business rules.
* **Response Structure**: Verifying that the response follows the correct structure (e.g., field names, data types, arrays) as outlined in the API documentation.

#### **Example:**

* After adding a new task via POST /tasks, the GET /tasks request should return the newly created task with accurate data.

**8. Usability Testing**

While not as common as the other types, usability testing evaluates the developer experience when interacting with the API. This is particularly important for public APIs, where a good developer experience can lead to adoption.

#### **Key Focus Areas:**

* **Documentation Quality**: Ensuring that the API is well-documented and easy to understand.
* **Ease of Use**: Verifying that API endpoints, methods, and parameters are logical and easy to work with.
* **Error Messages**: Ensuring that error messages are helpful, informative, and provide the user with actionable feedback.

#### **Example:**

* Verifying that the API documentation correctly describes the endpoint POST /tasks, the required parameters, and the expected response structure.

**9. Mutation Testing**

Mutation testing is used to evaluate the effectiveness of your tests by intentionally introducing small changes (mutations) into the API or codebase to check if the test cases can catch these defects.

#### **Key Focus Areas:**

* **Defect Detection**: Checking whether the API tests can detect unexpected changes or errors in the code.
* **Test Coverage**: Ensuring that test cases cover a wide range of possible API behaviors and conditions.

#### **Example:**

* Making a small change to a response code, such as changing the status code from 200 OK to 404 Not Found, and checking if the test suite catches the discrepancy.

10. What is Responsive Testing ?

* **Responsive testing** is the process of checking how a website or web application looks and works on different devices, screen sizes, and browsers.
* For example, a website might look great on a large desktop screen, but you want to ensure that it also looks good and works well on smaller screens, like phones or tablets. Responsive testing makes sure that images, text, buttons, and menus resize, rearrange, or change to fit the screen without any issues.

### **Simple Example:**

* If you visit a website on your phone, responsive testing ensures that the content isn't too large to view or too small to interact with. The layout should change so that it fits your phone's screen size, and touch buttons should be easy to tap.

In short, responsive testing ensures the best user experience no matter what device or screen size someone uses to access a website.

### **Types of Responsive Testing**

1. **Manual Testing**:
   * This involves testing the responsiveness of the web application by manually resizing the browser window or using developer tools in browsers like Chrome's Developer Tools to simulate different screen sizes (e.g., mobile, tablet).
   * The tester checks if the layout, content, and user interface elements adapt to the screen size as expected.
2. **Automated Testing**:
   * Automation tools like **Selenium**, **Cypress**, and **TestCafe** can be used to automate responsive tests by simulating different screen sizes and orientations.
   * These tools allow you to write tests that verify if elements are displayed correctly across a range of devices and if interactions work as expected on each.
3. **Device Testing**:
   * This involves testing on real physical devices (e.g., smartphones, tablets) or using device simulators/emulators (e.g., BrowserStack, Sauce Labs).
   * This is crucial for ensuring the web application works on a variety of mobile devices and operating systems (iOS, Android).
4. **Viewport Testing**:
   * Viewport testing involves checking how the application behaves in various viewport sizes, ensuring that no content is cut off or inaccessible at smaller screen sizes.

11. Which types of tools are available for responsive testing ?

### **Tools for Responsive Testing:**

1. **Browser Developer Tools**:
   * Most modern browsers like Chrome, Firefox, and Safari have built-in developer tools that allow testers to simulate different screen sizes and device resolutions. For example, Chrome’s **Device Mode** lets you test different mobile devices and simulate touch events.
2. **BrowserStack**:
   * BrowserStack is a cloud-based service that lets you test your website on real devices and browsers in various combinations. It supports testing on both desktop and mobile devices, including their responsiveness across various screen sizes.
3. **Sauce Labs**:
   * Similar to BrowserStack, Sauce Labs is a cloud testing platform that allows you to run automated tests across multiple browsers, operating systems, and devices. It provides responsive testing across real devices and emulators.
4. **Responsinator**:
   * Responsinator is an online tool that allows you to test your website’s responsiveness on different devices by simply entering the URL. It’s a quick way to verify the layout across common screen sizes.
5. **LambdaTest**:
   * LambdaTest is another cloud platform that offers cross-browser and responsive testing across multiple devices and operating systems.
6. **Screenfly**:
   * Screenfly is a simple tool that lets you test your website’s responsiveness across a wide range of screen sizes and devices.
7. **TestComplete**:
   * TestComplete is an automated testing tool that supports responsive testing, where you can create scripts to simulate different screen sizes and viewports.

12. What is the full form of .ipa, .apk

* The full form of **.ipk** is **"It’s Package"**.
* IPK files contain software and related metadata, and they are used to install or manage software on devices running embedded Linux distributions, such as routers, set-top boxes, or IoT (Internet of Things) devices.

### **Key Points:**

* **IPK (It’s Package)**: A package format used by embedded Linux systems.
* **Purpose**: To install, update, or manage software on embedded Linux devices.
* **Similar to**: .deb files (for Debian-based systems) or .rpm files (for Red Hat-based systems).

- .apk = The full form of **.apk** is **Android Package Kit**.

An **APK** file is the file format used by Android operating systems to distribute and install applications. It is the package file format used for installing software (apps) on Android devices, similar to how **.exe** files are used for Windows applications or **.app** files for macOS applications.

13. How to create a step for to open the developer option mode ON ?

To enable Developer Options on an Android device, you can follow these steps:

### 1. Open the Settings App

* Go to your device's Settings app. You can find this app in your app drawer or swipe down the notification bar and tap the gear icon.

### 2. Scroll Down and Find "About Phone"

* Scroll to the bottom of the settings menu and tap on About Phone (or About Device, depending on the manufacturer).

### 3. Find "Build Number"

* In the About Phone section, look for the Build Number entry. This might be under a sub-menu like "Software Information."

### 4. Tap Build Number Multiple Times

* Tap on the Build Number entry 7 times in quick succession. You may be prompted to enter your device's PIN or password to confirm.
* After tapping 7 times, you should see a message saying that Developer Options have been enabled.

### 5. Access Developer Options

* Now, go back to the main Settings menu. You should see a new section called Developer Options (often under "System" or "Additional settings" depending on the manufacturer).

### 6. Turn On Developer Options

* Tap on Developer Options and toggle the switch to ON at the top of the screen. You can now access various features such as USB debugging, logging, and other advanced settings for app development.

This process may slightly vary depending on your Android device's manufacturer (Samsung, Google, Xiaomi, etc.), but the core steps are generally the same.