**ERROR HANDLING:**

**TYPES OF ERRORS:**

**1. Expected Errors:**

Expected errors are common issues or problems that developers anticipate and handle gracefully within the application. These errors do not disrupt the entire application and can be recovered from without causing severe consequences. They are part of the normal operation and might occur due to user input, server-side processing, or external dependencies.

**Example:**

A user submits a form with incomplete information. The application detects the missing fields and displays an error message asking the user to fill in the required information before proceeding.

**2. Warnings:**

Warnings are non-critical issues that do not stop the application from functioning but alert developers or users about potential problems or misuse. They serve as alerts and might require further investigation to prevent more severe errors.

**Example:**

A web application detects that a user's password will expire soon. It displays a warning message to the user, advising them to change their password for security reasons.

**3. Fatal Errors:**

Fatal errors are severe issues that prevent the web application from continuing its normal operation. They can crash the application or lead to data loss if not handled properly. These errors often require immediate attention from developers to resolve the underlying cause.

**Example:**

The web application's database server encounters a critical error and becomes unresponsive. As a result, all database operations fail, and users cannot access the application.

In web engineering, it is essential to handle errors properly by implementing error handling mechanisms such as try-catch blocks, logging, and appropriate error messages. This helps in maintaining the application's stability, improving user experience, and aiding developers in identifying and resolving issues promptly.

**ERROR RECOGNITION & HANDLING in PHP:**

In PHP, error reporting is essential for debugging and maintaining the stability of your web application. There are various ways to handle and report errors in PHP. The primary techniques include:

***1. Displaying Errors on the Webpage:***

During development and debugging, it's common to display errors directly on the webpage to quickly identify and fix issues. You can enable error reporting by adding the following lines at the top of your PHP script:

<?php

ini\_set('display\_errors', 1);

ini\_set('display\_startup\_errors', 1);

error\_reporting(E\_ALL);

The `display\_errors` directive controls whether errors should be shown in the output, `display\_startup\_errors` determines if PHP should display errors that occur during PHP's startup sequence, and `error\_reporting` sets the level of errors to be reported (in this case, all types of errors).

***2. Logging Errors:***

In a production environment, it's not advisable to display errors on the webpage as they may reveal sensitive information to users. Instead, you can log errors to a file or a centralized logging system. To enable error logging, add the following lines to your PHP script:

<?php

ini\_set('log\_errors', 1);

ini\_set('error\_log', '/path/to/error\_log.log');

Replace `/path/to/error\_log.log` with the actual path where you want to save the error log file. This file will capture all the errors, warnings, and notices generated by your PHP application.

***3. Custom Error Handling:***

PHP allows you to define custom error handlers that will be triggered when certain types of errors occur. This gives you more control over how errors are handled and reported. To create a custom error handler, use the `set\_error\_handler` function:

<?php

function customErrorHandler($errno, $errstr, $errfile, $errline) {

    // Your custom error handling code here

    // For example, log the error or display a user-friendly message

}

set\_error\_handler('customErrorHandler');

***4. Exception Handling (for Fatal Errors):***

For fatal errors, you can use exception handling in PHP to gracefully handle the errors and prevent the application from crashing. By wrapping your code in a try-catch block, you can catch exceptions and handle them appropriately:

<?php

try {

    // Your code that might throw an exception

} catch (Exception $e) {

    // Handle the exception here

}

**Error Handling in JS:**

1. **Try-Catch Blocks**: JavaScript supports try-catch blocks, similar to PHP and other programming languages. You can use try-catch to catch and handle exceptions (errors) that occur during the execution of your code.

**Example**:

try {

// Code that may throw an exception

if (someCondition) {

throw new Error("This is an exceptional condition.");

}} catch (error) {

// Handle the exception

console.error("Caught exception: " + error.message);

}

1. **window.onerror Event Handler:** The **window.onerror** event handler is a global event handler that allows you to capture unhandled JavaScript errors on the window object. It's useful for logging unexpected errors and diagnosing issues in production environments.

**Example**:

window.onerror = function (message, source, lineno, colno, error)

{

console.error("Unhandled error:", message, "at", source, "line", lineno);

return true;

};

1. **Promise Rejection Handling**: When working with JavaScript promises (asynchronous operations), it's crucial to handle promise rejections to avoid unhandled promise rejections. Unhandled rejections can lead to unexpected application behavior and errors.

**Example:**

function fetchData() {

return new Promise((resolve, reject) => {

// Simulate an error condition

const errorCondition = true;

if (errorCondition) {

// Reject the promise with an error message

reject(new Error("Failed to fetch data: An error occurred."));

} else {

// Simulate successful data retrieval

const data = { name: "John", age: 30 };

resolve(data);

}

});

}

1. **Handling Asynchronous Errors**: For asynchronous code that uses callbacks or event listeners, you should handle errors appropriately within the callbacks or event handlers to prevent unhandled exceptions.

**Example:**

fetchData()

.then((data) => {

console.log("Data:", data);

// Process the data here when the promise is resolved

})

.catch((error) => {

console.error("Error:", error.message);

// Handle the error here if the promise is rejected

});

**Authentication**

Authentication is the process of verifying the identity of a user, device, or system to ensure that they are who they claim to be. It is a critical aspect of cybersecurity and information security, as it helps prevent unauthorized access and ensures the confidentiality, integrity, and availability of resources and data. Authentication mechanisms come in various types, depending on the factors used to verify the identity. Here are some common authentication types:

1. **Single-Factor Authentication (SFA):** Single-factor authentication requires the user to provide only one type of credentials to verify their identity. The most common form of SFA is the use of a password or a PIN. However, other types of credentials, such as a biometric (fingerprint or face recognition), can also be used for SFA.

Example: Entering a password to access an email account.

1. **Two-Factor Authentication (2FA):** Two-factor authentication adds an extra layer of security by requiring the user to provide two different types of credentials to prove their identity. Typically, it combines something the user knows (e.g., a password) with something the user possesses (e.g., a smartphone or hardware token) or something inherent to the user (e.g., a fingerprint).

Example: After entering a password, the user receives a one-time code on their smartphone, which they must enter to complete the login process.

1. **Multi-Factor Authentication (MFA):** Multi-factor authentication goes beyond two factors and involves the use of three or more types of credentials to authenticate a user. The additional factors may include something the user is, knows, and possesses, making it more secure than 2FA or SFA.

Example: To access a sensitive database, the user might need to provide a password, a fingerprint scan, and a smart card.

**HTTP (Hypertext Transfer Protocol)**

HTTP (Hypertext Transfer Protocol) supports various forms of authentication mechanisms that help secure access to web resources and protect sensitive data. These authentication methods are communicated between the client (usually a web browser) and the server through the "www-authenticate" response header. Here are three common forms of authentication supported by HTTP:

1. **HTTP Basic Authentication**: HTTP Basic Authentication is the simplest and most straightforward authentication method. When a client makes a request to a server that requires authentication, the server responds with a "401 Unauthorized" status code along with the "www-authenticate" header containing the authentication method required, which, in this case, is "Basic."

The client then includes the authentication information in the "Authorization" header of subsequent requests. The information is a base64-encoded string containing the username and password separated by a colon.

1. **HTTP Digest Authentication**: HTTP Digest Authentication is more secure than Basic Authentication because it sends the password as a hash value instead of plaintext. The server challenges the client with a "401 Unauthorized" status code and includes the "www-authenticate" header with the "Digest" method. The client computes a hash value based on the username, password, and other information, and sends it in the "Authorization" header of subsequent requests.
2. **Form-Based Authentication**: Form-Based Authentication is commonly used in web applications. It works differently from Basic and Digest Authentication, where the server responds with a web page containing a login form. The client submits the credentials using the form, and the server verifies them. Once authenticated, the server sets a session cookie or token, and subsequent requests from the client include this token for authentication.

Form-Based Authentication is flexible and commonly used for web applications, but it requires the application to handle session management securely.

Common **threat vectors** are techniques or methods used by attackers to exploit vulnerabilities in systems, applications, or networks. Let's explore the three threat vectors you mentioned and how they help mitigate certain security risks:

1. **Throttle Login Attempts:** Throttling login attempts is a security measure used to protect against brute-force attacks. In a brute-force attack, attackers try a large number of possible username and password combinations to gain unauthorized access to an account.
2. **Limit Number of Guesses:** Limiting the number of guesses is another technique to prevent brute-force attacks and password guessing attacks. It is similar to throttling but restricts the number of total guesses that can be made rather than the rate of attempts. This means that after a certain number of incorrect login attempts, the system locks out the user account or IP address for a specified period, making it more difficult for attackers to guess the correct password.

For instance, a system might allow a maximum of five incorrect login attempts. If the user enters an incorrect password five times, the system locks the account for a specific time (e.g., 30 minutes), protecting against automated or manual password guessing attacks.

1. **CAPTCHA**: CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) is a security mechanism used to verify that the user accessing a system or service is a human and not an automated script or bot. CAPTCHA presents users with challenges that are typically easy for humans to solve but difficult for automated scripts to decipher.

Examples of CAPTCHA challenges include distorted text, image recognition tasks, or solving simple puzzles. By requiring users to solve a CAPTCHA before performing certain actions, such as submitting a form or creating an account, websites can prevent automated bots from abusing their services.