Nick Marshall-Eminger

Error Handling

# Introduction

For this project we want to prioritize the back end development, then the front end. The back end will encompass our program logic, database, and the entry point. While the front end will utilize that entry point to access all functions behind it.

With each implementation of error handling developers must first identify the potential problems, catch specific errors, ensure that the information provided about the error is not revealing (for security), log each error appropriately, and prevent the system from crashing. One noted exception to the above rules is we will not be logging failed logging attempts, so as to prevent an infinite loop of log attempts.

*Error handling should be handled as components are developed, by the developer working on that portion of code, and not put in as an afterthought.*

Web applications frequently generate error conditions during normal operation. Out of memory, null pointer exceptions, system call failure, database unavailable, network timeout, and hundreds of other common conditions can cause errors to be generated. These errors must be handled according to a well thought out scheme that will provide a meaningful error message to the user, diagnostic information to the site maintainers, and no useful information to an attacker.

Simple error messages should be produced and logged so that their cause, whether an error in the site or a hacking attempt, can be reviewed.

With that mentioned, error handling must be done on each level of the system. We will separate this into general rules and general layers for now, that start with the back end moving to the front end.

# 

# General Rules

Split into DOs, DO NOTs, and CONSIDERations :

* **DO** report execution failures by throwing exceptions.
* **DO** document all exceptions thrown by publicly callable members because of a violation of the member contract (rather than a system failure) and treat them as part of your contract.
* **DO NOT** return error codes.
* **DO NOT** use exceptions for the normal flow of control, if possible.
* **DO NOT** have public members that can either throw or not based on some option.
* **DO NOT** have public members that return exceptions as the return value or an “out” parameter.
* **DO NOT** throw exceptions from exception filter blocks.
* **DO NOT** allow publicly callable APIs to explicitly or implicitly throw [NullReferenceException](https://docs.microsoft.com/en-us/dotnet/api/system.nullreferenceexception), [AccessViolationException](https://docs.microsoft.com/en-us/dotnet/api/system.accessviolationexception), or [IndexOutOfRangeException](https://docs.microsoft.com/en-us/dotnet/api/system.indexoutofrangeexception). These exceptions are reserved and thrown by the execution engine and in most cases indicate a bug. Do argument checking to avoid throwing these exceptions. Throwing these exceptions exposes implementation details of your method that might change over time.
* **CONSIDER** the performance implications of throwing exceptions. (Throw rates above 100 per second are likely to noticeably impact the performance of most applications.)
* **CONSIDER** using exceptions builder methods. (It is common to throw the same exception from different places. To avoid code bloat, use helper methods that create exceptions and initialize their properties.)
* **AVOID** explicitly throwing exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

From:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/exception-throwing>

# Back End Error Handling

## Managerial Layer

### *Entry Point*

The entry point must handle all user initiated errors, reporting to the user what general error has occurred, and ,if necessary, how they should proceed to fix this error on their end. If no fix is available (such as “Invalid password entered”) we will advise them as such and provide alternative options.

We do not want to provide specifics that can be used by malicious users to identify weaknesses or what effects their actions are having on the system. This should help prevent some threats.

Specific errors we want to catch at the entry point (or lower) are:

* Invalid credentials
* Invalid requests for information
* Failed uploads
* Failed saves
* Duplication failures
* Deletion failures
* Save state failures
* Log failures
* Login or logout failures
* Query failures
* Page load failures
* Invalid input
* Query timeout

### *Business Logic*

Business logic errors should be handled within their respective classes as appropriate. If the need for custom error handling classes is needed, they should be created. Custom exceptions should not be made out of business rules, exceptions are unexpected events that we want to make sure are not ignored.

Errors likely to propagate in the business logic are:

* NullPointerException

## Service Layer

All reusable code must either throw the appropriate exceptions to be caught, or must be handled within the code. Documentation for each class must be provided with the appropriate throw noted, and what that throw is intended to catch.

## Data Access

Data access errors must be handled with each call, logging and storing errors, but never returning specifics to the end user. The user should get a very generic error response such as “Unavailable, try again later” so as to prevent malicious users from identifying why they do not have access (such as access denial from lower permissions).

# Front End Error Handling

# Resources Needed

# Risk Assessment

Good error handling mechanisms should be able to handle any feasible set of inputs, while enforcing proper security. Simple error messages should be produced and logged so that their cause, whether an error in the site or a hacking attempt, can be reviewed. Error handling should not focus solely on input provided by the user, but should also include any errors that can be generated by internal components such as system calls, database queries, or any other internal functions.

One common security problem caused by improper error handling is the fail-open security check. All security mechanisms should deny access until specifically granted, not grant access until denied, which is a common reason why fail-open errors occur. Other errors can cause the system to crash or consume significant resources, effectively denying or reducing service to legitimate users.

A specific policy for how to handle errors should be documented, including the types of errors to be handled and for each, what information is going to be reported back to the user, and what information is going to be logged. All developers need to understand the policy and ensure that their code follows it.

In the implementation, ensure that the site is built to gracefully handle all possible errors. When errors occur, the site should respond with a specifically designed result that is helpful to the user without revealing unnecessary internal details. Certain classes of errors should be logged to help detect implementation flaws in the site and/or hacking attempts. Very few sites have any intrusion detection capabilities in their web application, but it is certainly conceivable that a web application could track repeated failed attempts and generate alerts. Note that the vast majority of web application attacks are never detected because so few sites have the capability to detect them. Therefore, the prevalence of web application security attacks is likely to be seriously underestimated.

The OWASP Filters project is producing reusable components in several languages to help prevent error codes leaking into user’s web pages by filtering pages when they are constructed dynamically by the application.

**Resources necessary**

**Amount of time it will take the feature to be completed from start to finish.**

## Step 4: Create the Project Schedule

## Step 5: Identify Issues and Complete a Risk Assessment

## 

## 

## 