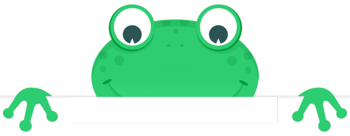
leapfrog

Developers Handbook

Guide and best practices for developers



Credits

Authors Anish Manandhar

Bala Maharjan

Nischal Khadka

Robus Gauli

Sawan Vaidya

Saugat Gautam

Editor Suja Manandhar

Designer Ishan Manandhar

Special Thanks

Leapfrog Lead Engineers Team

All Leapfroggers

Table Of Contents

Introduction

Software development is as much of an art as it is science. It is not only the application of one's logic but also the expression of their imagination and craftsmanship. Though it might not seem like a big deal for a new developer to create working applications, gaining mastery of this craft requires passion, dedication, pragmatism and continuous learning. This handbook has been designed to provide you with a concise introduction to the best practices that one should follow in their road to mastering the art of software development. Though it may not be possible to precisely quantify the value of these best practices, they are observed to be commonly used by the best of the developers and successful organizations.

While it is not feasible to cover every topic in depth, the handbook intends to provide you with a starting point. You should dive deeper on your own to gain an in-depth knowledge of the best practices and to implement them in real time scenarios.

Also, this excerpt from Josh Bloch's seminal book ***Effective Java*** is equally applicable to this handbook:

> “While the rules in this handbook do not apply 100 percent of the time, they do characterize best programming practices in the great majority of cases. You should not slavishly follow these rules, but violate them only occasionally and with good reason. Learning the art of programming, like most other disciplines, consists of first learning the rules and then learning when to break them.
>
> **Josh Bloch**
> Author of Effective Java

Finally, you're bound to make mistakes or errors in judgement no matter how much you try to adhere to the best practices. Learn from your mistakes and take the responsibility to rectify them. Be open to admit ignorance or error or to even admit that you need help.

Who Is This Handbook For?

This handbook is primarily written for software developers. These terms mean different things but this handbook uses these terms interchangeably. This book can also be used by people other than developers. For example a Quality Assurance Engineer can use this to know about core weaknesses of a project which leads to bugs and other problems. Similarly, a Project Manager can read this to know what developers should do to improve the health of the project.

Readability or Understandability

Readable code is one in which the components of the code are easy to grasp and the goal of the overall code is well defined. Building a codebase that is readable is not just a single goal for a team but a commitment that lasts the entire duration of the project. Developers should think of themselves as authors of their code instead of configuration of a complex piece of software.

Here are some cultural habits that leads to a readable code:

- A good code review practice is in place and rules are set from the onset of the project.
- Developers are mindful of how someone new will perceive their code.
- The Boy Scout Rule - "Leave the campground cleaner than you found it" is followed which iteratively improves the readability of code.

Well Formatted and Follows Standards

Every popular language/framework has guidelines over how code should be formatted and a list of dos and don'ts. It is best to start with these standards at the onset of the project. Sometimes the team may have custom stylistic preferences. These should be well documented and people from outside the team may not follow these standards. Also, it is important to not overstretch the standards as this makes documenting and enforcing standards a burden in itself. As it could be a tedious process to monitor, evaluate and enforce the standards, most teams use linters and prettifiers. Additionally, standard code quality analyzer tools like Sonarqube and Codeclimate can dramatically improve the code review process through automation.

Right Documentation

It is popularly said that good code is self documented. While code should definitely have inherent readable qualities, it is wishful thinking to rely on this alone. Just like coding skills, documentation skills can be developed through learning and practice. Developers are recommended to learn about documentation syntaxes and standards for their respective frameworks. Expertise in Docblocks, knowledge of Markdown, RST or alike, and of diagram tools such as draw.io, lucidcharts etc help in producing crisp documentation. A project can also start using automatic documentation generation tools of their liking.

Convention Over Configuration

Conventions such as naming and placement of artifacts increase the readability of code in two ways. First, they enforce a high degree of consistency, and second they eliminate the burden of thinking about what parameters needs to be set in a configuration file.

To take this a step further is to create structures such as Abstract classes, Enumerations and choose a logic organization strategy such as MVC (Model View Controller).

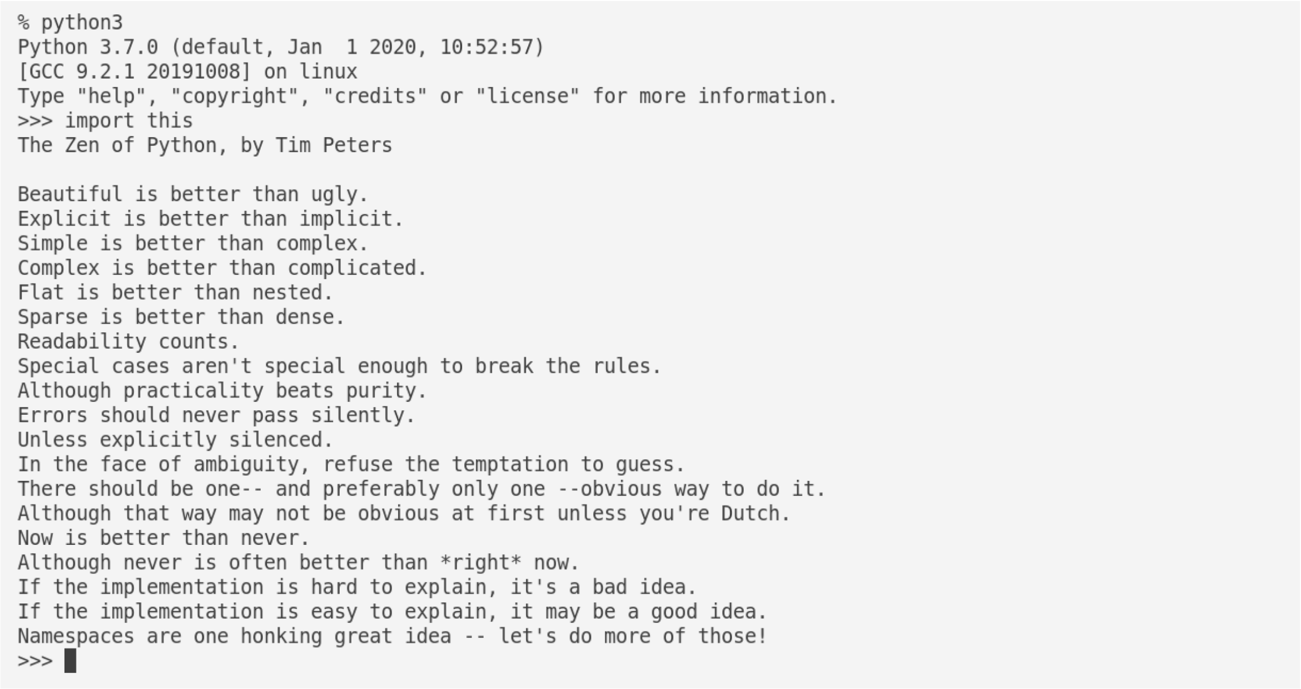
Libraries

It is advised to practice restraint on picking libraries. We can run into libraries that fully solve our problems but are obscure, and there are ones that partially solve our problems that are very popular.

Writing one's own library can sometimes seem tempting and can be beneficial as the end product is tailored to the project's needs and is fully under the developer's control. However, with the power also comes the necessity to maintain and document the workings of the library. Libraries which are publicly available do not necessitate this as the documentation and maintenance is performed by the owner or the community.

Zen of Python

Although this topic might look like it's specific to Python, you will find it useful across all languages and platforms. Like "Zen Religion", you will find that many things in this guideline is not absolute and is open to interpretation.



```
% python3
Python 3.7.0 (default, Jan  1 2020, 10:52:57)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>> █
```

Useful Patterns

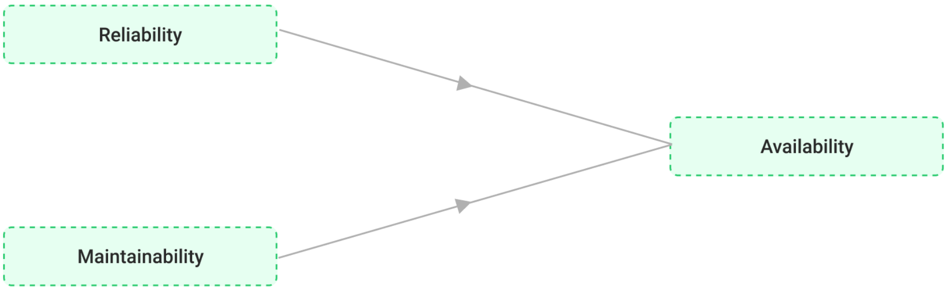
Some other ways to increase readability of code are as follows:

- Use Objects, Maps or Dictionaries instead of too many conditionals.
- Use Programmable objects in representing APIs such as the ones provided by ORMs.
- Naming should be consistent across variables, functions, files, database artifacts etc. For example, having a consistent style of prefixing such as "is_" prefix for booleans, "on_" prefix for callbacks, "view_" prefix for views etc can enhance readability.
- Write functions that have a single responsibility. Avoid writing functions with side-effects.
- Focus on unit tests to improve code quality as well as readability

Software Dependability

Software dependability is a measure of a software's availability, reliability, and its maintainability. Availability refers to readiness for correct service, reliability to continuity of correct service, and maintainability to ability for easy maintenance and repair. These three areas are interdependent and will be talked about together in the following sections. The following sections will also talk about how a developer can contribute to improving these areas.

RAM (Reliability - Availability - Maintainability)



Reliability

Reliable code is one that is failure tolerant. When things go wrong in reliable code, the user experience is shielded from the impact as much as possible. Reliable code is written on the assumption that things will fail. Code that assumes nothing will go wrong generally ends up failing catastrophically.

Availability

First and foremost, if your code is maintainable, it makes issues easier to fix and if its reliable less issues will occur. As a whole, this contributes to the availability of the system. Additionally, developers can use automatic corrective and preventive measures that increase a system's availability. High availability not only enables a developer to focus on non maintenance tasks but also impacts the user's perspective of the product.

Maintainability

Maintainability is the ability of any software system to support changes and be adaptable. Good comments, descriptive variables, methods and class names, encapsulation and abstraction, formatting guidelines, and documentation/diagrams are things to name only a few, that lead to maintainable code.

There are three key aspects of maintainable software:

Adaptive maintenance

The system needs to adapt to the dynamic nature of change in regards to framework, runtime, operating system or infrastructure.

Corrective maintenance

Corrective actions should be easy to initiate if the failure occurs. Similarly, any corrections made should not lead to failure of any other part of software. Design patterns, loose coupling and quality of code are the key areas for correctness.

Preventive maintenance

It deals with the activities that are undertaken to prevent the occurrence of errors or faults.

Road to Dependability

It is expected of you to make your code reliable and maintainable so that the application you make or contribute in making is dependable. The following sections suggest best practices that lead to dependability.

Improve Coding Pattern

There are lots of paradigms, patterns etc which helps in maintaining the code consistency. Adhering to a paradigm and architecture will help avoid repeated logic, re-invention of wheel and thus, makes the code more maintainable and reliable.

Follow SOLID principles

SOLID is a widely recognized design principle in Object Oriented Programming (OOP). If you adhere to this principle, your code will be highly maintainable. For a concise description of SOLID refer to appendix.

Keep things simple

Follow the KISS (Keep it simple, stupid) principle to heart. As soon as we spot that some parts of your code are starting to grow, it is time to step back and reflect on the overall design and responsibility of that part of code. Refactor to keep things simple in the light of new requirements.

Don’t re-architect your code for features you don’t know

Design only for things you know. Be open to refactoring the design as you go to suit your needs. To make safe changes, build up a good test automation suite, which can give you the confidence that your expectations are held as you keep changing the code.

Clean up unnecessary code

Less is more. Don’t shy away from removing code. Code which is not needed and just lying around means more code to maintain. Get into habits of removing dead code.

Isolate cross cutting concerns

Cross-cutting concerns are parts of a program that rely on or must affect many other parts of the system. Eg: caching, data validation, error detection and correction, internationalization and localization, logging, memory management, monitoring, persistence etc. By separating these cross-cutting concerns in a central place outside of your core program logic, you will gain modularity in your code and thus simplify maintenance.

Design for non-deterministic behavior

Every system will have non-deterministic aspects which are not predictable. Some factors that introduce non-deterministic behavior are execution completeness, message ordering and communication timing. To deal with non-deterministic nature of software, the following measures should be taken into consideration:

- Design program assuming the system will fail.
- Plan initially for recovery in the case of sudden disasters. (More details in Disaster Recovery section).
- Have performance monitoring tools in place for tracing and loggings of metrics which would span across multiple services.
- Deploy chaos testing within your environments and plan out the strategies accordingly.

Watch out for memory leaks

Memory Leak is the gradual loss of system available memory when the application fails to return the memory that it has consumed for temporary needs. Though, nowadays programming languages have very good memory leak detection and mitigation mechanisms like Garbage Collection, ARC etc, you still need to be aware of object lifecycle. Especially if you are using low level unsafe and unmanaged data types like `UnsafePointer` for example.

Prevent concurrency hazard

Whenever you are doing multithreaded programming, you will encounter many new and exciting issues like Race Condition, Deadlock, LiveLock, Starvation etc. during multithreaded concurrent programming.There are many ways of correcting such concurrency hazards. But the real solution here is to actually be aware that such conditions will exist in concurrent programming.

Maintain data consistency

An operation is transactionally consistent if it is true that either all steps of the operation completed successfully, or none of them do. The data being operated on reflects either the complete transaction, or remains unchanged.

Thus, your system should be designed so that a transaction will affect the data holistically in allowed ways only. Should a transaction fail, your system should be able to rollback to the last stable state.

Choose third party libraries carefully

You as a developer is always responsible for the libraries that you use in your project and any issues that the library might have is going to be your issue. So always choose those libraries or frameworks carefully. While choosing a publicly available library, also look at the quality of maintenance of the code. Indicators such as good documentation, recency of commits, number of stars and number of downloads give clues towards the reliability of a library. As a rule of thumb don’t use libraries that have less than 1000 stars.

Handle errors properly

Failure and errors should be the first class citizens in your program. Always embrace errors and put priority to handling failures and errors. Anticipation, detection, and resolution of programming, application, and communication errors makes your application very dependable.

Following measures should be taken into consideration for proper error handling :

Prefer built-in error object

Using built-in Error object helps to keep uniformity within your code and with third party libraries. It also preserves significant information like the StackTrace.

Custom exceptions can also lead to confusing stack traces when enough planning and thought is not put into designing them. As such, it is generally better to learn about standard exceptions provided by your programming language and use them instead of creating new ones.

Handle errors centrally

Error handling logic should be encapsulated in a dedicated and centralized object that all end-points (e.g. Express middleware, cron jobs, unit-testing) call when an error comes in. Not doing so will lead to code duplication and probably to errors that are handled improperly or not handled at all. So treating error handling as a cross cutting concern and thus isolating it from the core logic is the right way to go.

Perform validation first

It is generally a good practice to perform validations first at the top level of a function. If the validation fails we can exit from the function fast. The actual logic will come after the validation logic. This way it will be evident that the rest of the code after this early error handling is just business logic. The logic will not be cluttered with confusing error checks.

Throw early, catch late

It is a principle that states that errors should be raised where they occurred but they should be handled as late as possible. By throwing early as soon as a fault has occurred, we are letting the upstream developers know about it. What catching late really means is that we don’t want to rectify the error at the level it is thrown but catch as early as we know how to handle the issue properly. Also practice catching specific error. Example: Arithmetic Exception as opposed to General exception.

Use proper error codes

Error handling is usually done with returning special error codes. Error codes have significant advantages over writing everything in the return error object. You can always look for the error codes in a proper document and get the verbose explanation which helps in root cause identification of error.

As a developer, you must be aware of the HTTP Response Status Code. It is very important to know which range of error codes is reserved for which scenario. Eg: 4xx is for success, 5xx is for server errors etc. You can find the HTTP Status Codes and their usage in the appendix.

Employ Proper Logging

Logging refers to recording events that occur in your application which gives valuable information for current debugging purpose or future reference. Logging can be as basic as printing in a console or as sophisticated as collecting logs to a centralized location.

In order to overcome common troubleshooting complexities and achieve a more holistic view of your application and systems, you should monitor and log across all system components.This includes logging all relevant metrics and events from the underlying infrastructure, application layers and end user clients are equally as important.

Let's look at the things to consider in order to achieve effective logging during development.

Set a strategy

Carefully consider what you are logging and why. Logging needs to have a plan and strategy. A logging plan will generally include the following.

- List of the things that must be logged. You shouldn’t just log everything and anything.
- Log storage and transport plans, services and methodologies.
- Log archiving strategy - Consider automated archiving and moving to the other cheaper storage location
- Log rotation strategy.
- Notifications and alert plans.
- Identification of key members who need to be notified.
- Visualization of the logs.

Use logging framework

Logging frameworks makes your life easier and usually have following benefits:

- Don’t have to worry about formatting
- Facility to manage log levels
- Logging asynchronously
- Manages Information and Error automatically

Separate and centralize your log data

Logs should always be automatically collected and shipped to a centralized location, separate from your production environment.

Effective management of logs is imperative while building a software because logs are your only guiding lights at dark times when your application is live. We generally use in-built logging features of a language with other third party logging service which helps us manage the logs. Typically log management is achieved using softwares like Sentry, ELK, Logly etc. or a combination of those.

Categorize and segregate logs to suitable log levels

Often you'll only want to know if there are any problems or warning conditions. But sometimes you'll really want to dial up the information for troubleshooting purposes. Thus, segregating and communicating such information through the log itself is very crucial. Here are some of the common Log Levels.

Log Levels	Description
Fatal	Fatal represents truly catastrophic situations
Error	An error is a serious issue and represents the failure of something important going on in your application.
Warm	You use the WARN log level to indicate that you *might* have a problem and that you've detected an unusual situation.
Info	INFO messages correspond to normal application behavior and milestones.
Debug	With DEBUG, you start to include more granular, diagnostic information.
Trace	This is really fine-grained information. Finer even than DEBUG

Add context

When using logs as data, it’s important to consider the context of each data point. Knowing a user clicked a button may not be as useful as knowing a user specifically clicked the “purchase” button. Adding further context can reveal the type of purchase made. If the user’s purchase resulted in an error, the available context will facilitate swifter resolution. Here is the list of what a log should contain. The list is not comprehensive but it is a good indicator.

- Date Time
- Stack Errors
- Service Names
- Function/Class/File Name
- Server/Client IP address
- User Agent
- HTTP Codes

Perform Real-Time Monitoring

As the name suggests, Real-Time Monitoring means observing activities in your application as it's happening. We can only stream and monitor information with low latency. Information with high latency are generally monitored using periodic reports.

Real-time monitoring could refer to things such as:

- Observing user activity as it's happening
- Locating relevant and current data without having to shift through lengthy reports
- Catch shady activity as it happens
- Having "live tail" visibility into your log data
- Knowing the status and health of application or services

With the help of real-time monitoring, you can continuously monitor the state of your application and take proper action during or even before an issue occurs. This can also enable us to integrate notification mechanisms to the software

Everything mentioned above and more can be achieved by using Application Performance Management (APM) Tools. Some of the good products are NodePing, Pingdom, Uptime Robot, New Relic, Data Dog etc.

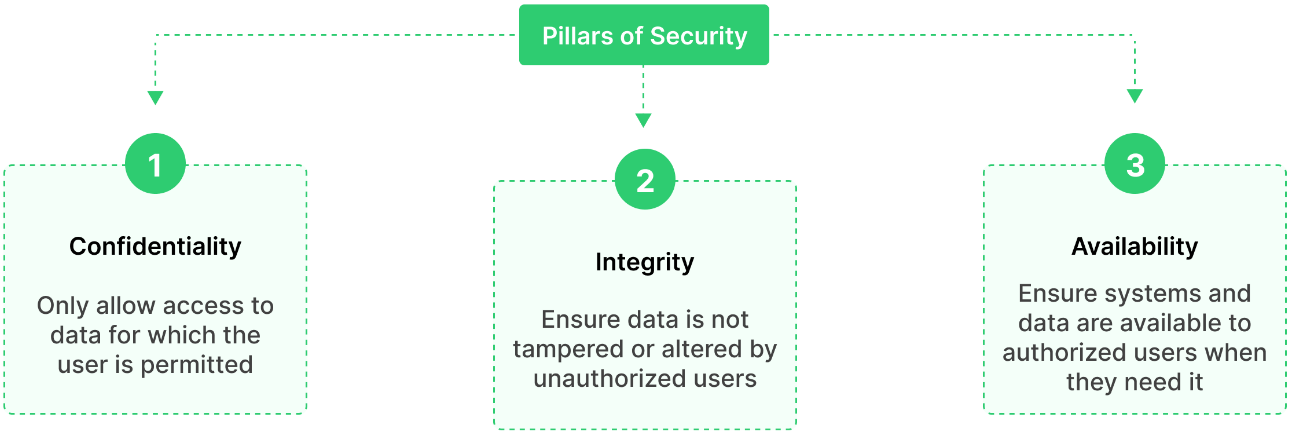
Test and Benchmark Your Product

In order to make your system more reliable you have to test your application, assess its performance, and verify that it works as expected in every situation. You can achieve this by performing different types of tests mentioned in the Testing Section. Unit Tests and integrations tests for example ensures that any new changes don't break your application. Similarly, load and performance tests help to ensure that your application doesn't fail in harsh conditions like low end machines or high traffic conditions.

Security

Software security is an idea implemented to protect software against malicious attacks and other hacker risks so that the software continues to function correctly under such potential risks.

It is not a feature, it is a necessity in today’s world. Security principles denote the basic guidelines that should be used when designing a secure system. Experience shows that a crucial success factor in the design of a secure system is the correct consideration of security principles. Vulnerabilities and attacks in most cases can be ascribed to the inadequate application of some principle. Information security has relied upon the following pillars:



Pillars of Security

1 **Confidentiality**

Only allow access to data for which the user is permitted

2 **Integrity**

Ensure data is not tampered or altered by unauthorized users

3 **Availability**

Ensure systems and data are available to authorized users when they need it

Security Principles

Minimize Attack Surface Area

The attack surface of a software environment is the sum of the different points (user input fields, hidden fields, public APIs etc) where an unauthorized user (the "attacker") can try to enter data to or extract data from an environment. Keeping the attack surface as small as possible is a basic security measure.

- Reduce amount of code available to unauthorized users
- Reduce entry points available to untrusted users
- Eliminate services requested by relatively few users

Principle Of Least Privilege

Your customer can be your first attacker since they are the one who have legal privilege to the application.The principle of least privilege recommends that accounts have the least amount of privilege required to perform their business processes. This

encompasses user rights, feature access, resource permissions, file system permissions etc.

For example, if a view is good enough to perform certain actions for a specific user role, then restrict the rest of the pages to that role.

Another example, if a middleware server only requires access to the network, read access to a database table, and the ability to write to a log, this describes all the permissions that should be granted. Under no circumstances should the middleware be granted administrative privileges or any more actions than these.

Defense In Depth

Defense in Depth is commonly referred to as the "**castle approach**" because it mirrors the layered defenses of a medieval castle. Before you can penetrate a castle you are faced with the moat, ramparts, draw-bridge, towers, battlements and so on. The lesson here is not to trust security measurements from preceding functions. Prepare for the worst possible scenario. Implement multiple defense mechanisms. Create a security architecture or design and document the different layers of protection. If one security service fails the security system should still be resistant against threats.

For example, in addition to making software available only in private network, all of the following could be equally important even though they are redundant:

- Authentication and authorization for APIs.
- Secured connection between the services.
- Encrypting Secrets.
- Database encryptions.

Fail Securely

Applications regularly fail to process transactions for many reasons. How they fail, can determine if an application is secure or not? If either codeWhichMayFail() or hasAdminRights fails or throws an exception, the user stays an admin, as isAdmin variable is not changed. This could have been prevented by either modifying the exception handling mechanism or by initializing the start value (isAdmin) with a lower privilege.

```
isAdmin = true;
try {
  codeWhichMayFail();
  isAdmin = hasAdminRights(params);
}
catch (Exception ex) {
  log.write(ex.toString());
}
```

Don’t Trust Services

Services refer to external software, data integration points which are used to perform certain business actions. These integrated services may contain vulnerabilities that are out of project control. Always validate the response against security checks and monitor unusual responses from these integrated services. Integrate these services only through secure information exchange interfaces like HTTPS, FTPS etc. Make agreements with the parties involved if possible.

Separation Of Duties

A key fraud control strategy is separation of duties. Separation of duties is the concept of having more than one person required to complete a task. The separation of a single task to multiple individuals, acts as an internal control mechanism. This works by enforcing a system of check and balance between individuals

For example, database specific configuration can be done by a DBA experts, CI/CD pipeline can be managed by a Devops expert, application specific configuration can be managed by a core developer. This reduces the chance of fraud and error on different layers

Avoid Security By Obscurity

Security through obscurity(the state of being unknown) is a weak security control. This is not to say that keeping secrets is a bad idea, it simply means that the security of key systems should not only be reliant upon keeping details hidden.

The security of an application should not rely upon knowledge of the source code or other artifacts being kept secret. The security should rely upon many other factors, including reasonable password policies, configurations, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls.

Keep Error Generic While Dealing With Confidentiality

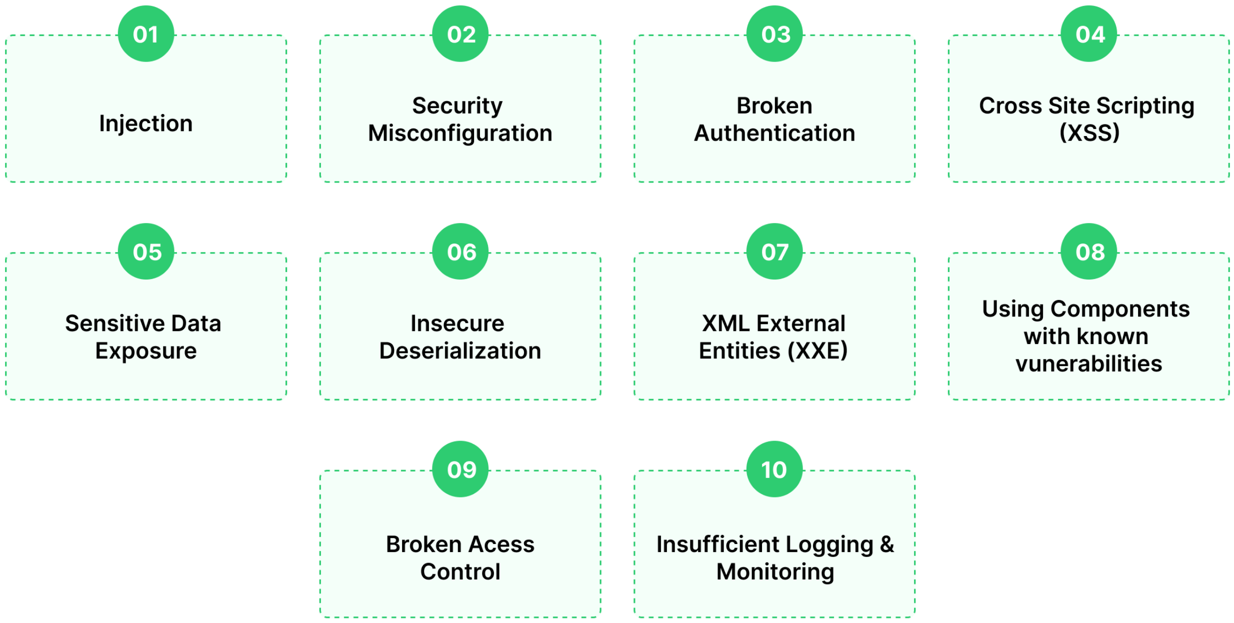
While it may seem helpful to let users know whether a piece of data exists, it’s also very helpful to attackers. When dealing with accounts, emails, and personally identifiable information, it's the most secure to error on the side of less. Instead of returning “Your password for this account is incorrect,” try the more ambiguous feedback “Incorrect login information,” and avoid revealing whether the username or email is in the system.

In order to be more helpful, provide a prominent way to contact a human in case an error should arise. Avoid revealing all the information that isn’t necessary. If nothing else, for heaven’s sake, don’t suggest data that’s a close match to the user input.

Learn About OWASP

The Open Web Application Security Project (OWASP) is a worldwide not-for-profit charitable organization focused on improving the security of software. OWASP's mission is to make software security visible, so that individuals and organizations are

TOP 10 APPLICATION SECURITY RISK 2019



01 Injection

02 Security Misconfiguration

03 Broken Authentication

04 Cross Site Scripting (XSS)

05 Sensitive Data Exposure

06 Insecure Deserialization

07 XML External Entities (XXE)

08 Using Components with known vunerabilities

09 Broken Acess Control

10 Insufficient Logging & Monitoring

able to make informed decisions. OWASP is in a unique position to provide impartial, practical information about application security to individuals, corporations, universities, government agencies, and other organizations worldwide. Operating as a community of like-minded professionals, OWASP issues software tools and knowledge-based documentation on application security.

The OWASP Top 10 is a powerful awareness document for web application security. It represents a broad consensus about the most critical security risks to web applications. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.

The top 10 vulnerabilities may change timely according to the security trends, innovations and practices. Please refer to the OWASP official site to stay updated.

Optimization

Optimization is the process of modifying functional code so as to improve speed, efficiency, memory consumption, data structures, etc. These goals often contradict one another. For instance, spawning a thread pool to perform large computation often hamper readability as well as exposes program to deadlocks and data races. Often there is no "one size fits all" design, so we need to make trade-offs to optimize the attributes of highest impact.

Levels of Optimization

Design

Architectural design principles of system greatly impacts overall performance of the system. Few examples are as follows:

- I/O bound system can be designed for minimizing network round trips as opposed to optimizing for CPU cycles.
- Breaking down components into an independent unit of computation for parallel execution. Different languages offers abstraction over parallelism and concurrency such as thread pool executor in python, goroutines in golang, supervision tree in erlang, process pool executor, etc

Data Structure

After design, both the choice of algorithms and data structures affect efficiency more than any other aspect of the program. Generally data structures are more difficult to change than algorithms, as a data structure assumption and its performance assumptions are used throughout the program. This can be minimized by the use of abstract data types in function definitions, and keeping the concrete data structure definitions restricted to a few places.

For algorithms, this primarily consists of ensuring that algorithms are constant O(1), logarithmic O(log n), linear O(n), or in some cases log-linear O(n log n) in the input (both in space and time). Algorithms with quadratic complexity $O(n^2)$ fail to scale, and even linear algorithms cause problems if repeatedly called, and are typically replaced with constant or logarithmic if possible.

Few of the scenarios around choice of data structure can be:

- Using Set data structure if inclusion test and is a frequent operation.
- Using a Dictionary/Map might be suitable for frequent member access and manipulation operation. This implementation requires us to create key value pair to represent data.
- Using Heap Queue when accessing minimum weight among the collection of objects is preferred.

Finally, when it comes to applying suitable data structure to better optimize, it is recommended to utilize abstractions provided by the programming languages itself before implementing data structures and algorithms from scratch.

Source Code

After analysis of complexity in terms of space and time, there are areas in your source code that we could potentially optimize for, such as looping constructs, branches, chain of inheritance and composition, runtime reflections, cyclomatic complexities. Some examples would be inlining function calls instead of chain of function calls, limiting inheritance tree and relying on composition to reduce runtime bottlenecks, using caching to reduce re-computation (lru-cache), using persistence data structures for immutability and so forth and so on.

Before Optimization

Optimization can reduce readability and add code that is used only to improve the performance. This may complicate programs or systems, making them harder to maintain and debug. As a result, optimization or performance tuning is often performed at the end of the development stage.

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%
>
> **Donald Knuth**
> Computer Scientist

"Premature optimization" describes a situation where a programmer lets performance considerations affect the design of a piece of code. This can result in a design that is not as clean as it could have been or a code that is incorrect, because the code is complicated by the optimization and the programmer is distracted by optimizing.

When deciding whether to optimize a specific part of the program, Amdahl's Law should always be considered: the impact on the overall program depends very much on how much time is actually spent in that specific part, which is not always clear from looking at the code without a performance analysis.

A better approach is therefore to design first, code from the design and then profile/ benchmark the resulting code to see which parts should be optimized. A simple and elegant design is often easier to optimize at this stage, and profiling may reveal unexpected performance problems that would not have been addressed by premature optimization.

In practice, it is often necessary to keep performance goals in mind when first designing software, but the programmer balances the goals of design and optimization.

Documentation

In software engineering, documentation encompasses all artifacts and materials that explains with the software product’s development and use. All the software development products, whether created by a small team or a large corporation requires a basic documentation. Documentation exists to explain product functionality, unify project-related information, and provide references for discussing all the significant questions arising between stakeholders and developers.

While we generally refer to documentation in written format, it can also be visual (diagrams, slides) and can incorporate multimedia (recorded demo). This section will describe different types of documents and explain their use cases.

Documentation can be boring but can prove to be helpful when the existing developer exits from the project or the new developer joins the team. Documentation can be the key in understanding when it comes to what are you going to do, why do you need it and how to do it. Any kind of documentation can be helpful when we need to refer to them for future purpose.

Guidelines For Documenting Project Practices

The purpose of documenting standards is to make sure that all practices and artifacts are designed, written and laid out in the same manner.

This makes it easier for the team members to understand what you switch from one person to another without the needed change of mentality to read someone else's style. It's all about uniformity, and nothing about right and wrong. Here are some recommendations for documenting standards within projects:

- Provide instructions for project setup in both production and development environments
- Document coding convention and styles used in the project
- Mention tools and third party services
- Specify what Git branching strategy is used
- Give high importance to security and access management
- Ensure a standard error handling and logging practice is in place
- Enforce a good code review process with mandate baseline for testing
- Lay out the CI/CD & release process

Guidelines for documenting architecture

Architecture is a well-structured set of concepts for a structure that represents certain principles, elements and components.

Types of Architectural Documents

Documentation of deployment

1. Maintain a deployment architecture diagram and documentation
    - a. Should have information about all the installation steps of packages or dependencies
    - b. Should have instructions to make necessary changes to Eenvironment files
    - c. Should delineate the instructions on the basis of tech stack
        - i. Frontend Stack
        - ii. Backend Stack
        - iii. Database Stack
2. Prepare and follow a deployment checklist
    - a. Should have infrastructure (Server, Backup Plan) setup steps
    - b. Should have instructions on security measures
        - i. Use of SSL
        - ii. Server exposure to public APIs
        - iii. Check for Compliance
            - Data at rest (eg: in database or in memory)
            - Data at movement (eg: data transferred over internet using protocols like HTTPS, SSH, etc)
    - c. Should have instructions to bump version
    - d. Should have instructions to publish release notes
    - e. Should have instructions for Smoke or Sanity Testing
    - f. Should have instructions of Rollback strategy, if deployment fails, and others depending on your project’s requirements

Documentation of Dataflow

Data flow diagrams are used to graphically represent the flow of data in a business information system. DFD describes the processes that are involved in a system to transfer data from the input to the file storage and report generation.

Data flow diagrams can be divided into **logical** and **physical**. The logical data flow diagram describes the flow of data through a system to perform certain functionality of a business. The physical data flow diagram describes the implementation of the logical data flow.Data flow diagrams are used to graphically represent the flow of data in a business information system. DFD describes the processes that are involved in a system to transfer data from the input to the file storage and report generation.

Documentation of Database

The primary method of documenting a database is to use an architecture diagram itself. In addition to this, there are, a few diagram types specific to databases which are described in the section below:

An **entity relationship diagram** (ERD) shows the relationships of entity sets stored in a database. An entity in this context is an object, a component of data. An entity set is a collection of similar entities. These entities can have attributes that define its properties. ER diagrams are used to sketch out the design of a database.

Documentation of API

APIs are only as good as their documentation. A great API can be rendered useless if people don’t know how to use it, which is why documentation can be crucial for

success in the API economy. A well documented API can prove useful to users, developers and decision makers alike.

- Add detailed information on the authentication schemas
- Communicate error standards (error codes and messages) properly
- Identify and list all the resources that are exposed by the API
- Maintain and document the changes that the API goes through with a change log
- Learn about the target audience and try to avoid jargon accordingly
- Document Request and Response schemes and the flow between these events

- Use automated API documentation tools such as Swagger, Apiary, WSDL, etc to help you keep the documentation up to date

Guidelines for documenting with diagrams

We will talk separately about diagrams in documentation here. This is because we believe that technical documents at any level can be enhanced with figures, examples and diagrams. Like there are standards for coding, could there be something similar to building diagrams?

Unified Modeling Language (UML)

Unified Modeling Language (UML) is one of the most popular business process modeling techniques. It is based on diagrammatic representations of software components but can be hard at times to achieve this.

Types of UML Diagrams

UML diagrams can be broadly classified into two types. The first type deals with the structure of the system and can be useful for depicting architecture.

Diagrams	Description
Class diagram	Describe the static structure of a system
Package diagrams	Organize elements of a system into related groups to minimize dependencies between packages.
Object diagrams	Describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.
Composite structure diagrams	Show the internal part of a class.
Deployment diagram	Shows the execution architecture of the system

The second type of UML deals with interaction with the system or shows what goes on in the system when it is in use. This can be useful for depicting behavior.

Diagrams	Description
Activity diagrams	Illustrate the dynamic nature of a system by modeling the flow of control from activity to activity.
Sequence diagrams	Describe interactions among classes in terms of an exchange of messages over time.
Use case diagrams	Model the functionality of a system using actors and use cases.
State diagrams	Now known as state machine diagrams and state diagrams describe the dynamic behavior of a system in response to external stimuli. State diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.
Interaction overview diagrams	Model a sequence of actions and let you de-construct more complex interactions into manageable occurrences.
Timing diagrams	Focuses on processes that take place during a specific period of time.

Best Practices For Diagrams

Diagrams should be self-explanatory. If they're not, then they're failing. To make sure yours is easy to understand, make sure you keep variable elements consistent and explain everything in the legend, key, or glossary. Here are some key rules to help you.

- Choose the optimal number of diagrams
- Diagrams should be consistent with the others in terms of boxes, shapes, borders, lines etc.
- Track changes across diagram revisions and Include legends/keys/glossaries

Disaster Recovery

An IT disaster is characterized by some event that impedes an organization or a project from providing its regular service to the customers, causing significant loss that is irreversible in nature. **Disaster recovery (DR**) is a risk mitigation strategy that aims to protect an organization from the effects of significant negative events. DR allows an organization to maintain or quickly resume mission-critical functions following a disaster.

Software, hardware and network failures can be reversed in most cases, but it is hard to recover the loss of some intangible aspects of the project. These can be the reputation of the organization, the data privacy assurance of the customers, the time wasted because of the disaster etc. All of these can be translated to financial loss for the project, which makes disaster recovery a widely studied subject in the business domain as well. In the business domain, disaster recovery is addressed by formulating a Business Continuity Plan and by carrying out a Business Impact Analysis. These topics are out of the scope of this handbook but you may look for them if you are interested in knowing further.

Like most things, there is not a one-size-fits-all rule to disaster recovery. However, some generally applicable methods to reduce the risk of disasters are talked about in the following sections:

Disaster Risk Mitigation

The decision on how much to invest in disaster recovery generally depends on the size of the project. For example in smaller projects to mitigate server crash, developers can simply download the data and transfer them manually to another machine without significant impact on the customers. However in large projects, the same would generally be forbidden because of data privacy issues. Additionally, loss of availability can persuade a customer to switch to another company. In order to get an indication of how much you should invest in disaster recovery, start by doing a simple cost analysis of loss per day.

Create A Written Disaster Recovery Plan

A disaster recovery plan serves as the first place to look for help towards the process when a disaster occurs. Plan could include several things which could be technical or non-technical in nature. As such, creating the recovery plan is a collaborative effort of all the personnels of the project. The following are a few things a disaster recovery plan could include:

- A disaster recovery policy statement, plan overview and main goals of the plan.
- Key personnel and DR team contact information.
- Description of emergency response actions immediately following an incident.
- A diagram of the entire network and recovery site.
- Directions for how to reach the recovery site.
- Directions on how to recreate infrastructure, restore backups and/or redeploy the application
- Tips for dealing with the media.
- Summary of insurance coverage.
- Proposed actions for dealing with financial and legal issues.
- A list of software and systems that will be used in the recovery.
- Sample templates for a variety of technology recoveries, including technical documentation from vendors.
- Ready-to-use forms to help complete the plan.

As a developer, you may not be responsible for every item in the plan. The plan itself could contain the roles and responsibilities of each member. As technologies change, the disaster plans can have an expiration date. As such, the disaster recovery plan should be revised periodically.

Backup Based On Recovery Objectives

In disaster recovery, two important measurements made are the Recovery point objective (RPO) and the recovery time objective (RTO). RPO is the maximum time before the disaster that an organization must be able to recover from backup storage, for normal operations to resume after a disaster. The recovery point objective determines the minimum frequency of backups. For example, if an organization has an RPO of four hours, the system must back up at least every four hours. RTO is the maximum amount of time, following a disaster, for an organization to recover files from backup storage and resume normal operations. In other words, the recovery time objective is the maximum amount of downtime an organization can handle. If an organization has an RTO of two hours, it cannot be down for longer than that.

Conduct Disaster Drills

To predict what can happen during a disaster, and to plan is an undertaking that could be filled with flaws. To detect these flaws and to improve the readiness of the team, conduct role plays that test the disaster mitigation steps. Ask the participants for feedback during the drill and address them by improving the recovery plan.

Use Non-overlapping Recovery Mechanism

The backups and the recovery servers should be independent of the original infrastructure. For example, the backups placed in the same network could be compromised in case of security breach. If backup data is in the same AWS region as the original data, it could become unavailable too if the regional servers fail.

Set Importance To Human Resources

During a disaster, the team members of the project are the ones most capable of fixing the problems. As such, keep the team members, especially new members trained on the procedures. Follow standard procedures to off board team members by revoking access to the project artifacts which can include credentials, code and data.

Set Importance To Development Machines

The development machines can take up more time to set-up than a server. Backing up development machines and creating a productive development environment can be beneficial during disasters. The backups themselves can not only prevent developers from stalled productivity but also allow creation of new development machines in a short period of time when a disaster takes place. Tools such as Docker for containerization, scripts for automated setup can also be quite beneficial as they help to setup development machines quickly. Lastly, development machines and the backups should also be secured as they can lead to security vulnerabilities.

Keep Up To Date With Disaster Recovery Techniques

Just like any other aspect of technology, the techniques to mitigate the risks of disaster changes over time. 10 years ago it was not as easy to backup your data to the cloud or to create replica servers in regions around the world. Also, machines can be created or destroyed at a rapid speed using automatic deployment tools such as Ansible and Terraform. Knowing about these and employing the latest techniques helps your organization reduce the cost of disaster recovery itself.

Post Disaster

Despite following the best industry practices and having an impressive array of software and utilities, projects can have bad days. The following steps help mitigate the damage caused by a disaster and reduce the recovery time:

Refer To The Disaster Plan

There are multiple ways to mitigate the effects of the disaster after it occurs. Paradoxically, this can cause a state of chaos if the process of recovery is not clearly defined. As such the first thing you should do is to refer to the disaster plan if you have one and communicate what the plan is to all the stakeholders

Establish A Disaster Recovery Team With A Lead

In order to execute the disaster plan, establish a team with a lead. The team leader should clarify the responsibility of each individual in the team and take ownership of the disaster plan. The team should not only look into ways to recover from the disaster but also be the point of communication for the concerned stakeholders.

Provide Temporary Relief

Temporary relief post-disaster can be provided to the users by suggesting alternatives. For example, if an application that takes orders online is down due to a disaster, you could establish an email support group for ordering critical items. Another means is to look for partners who can provide temporary infrastructure and service.

Counsel Affected Stakeholders

Communication is key when it comes to reducing the loss caused by damaged public relations. In communicating with the affected stakeholders, do not impart conflicting or misleading information. Having a single designated point of communication can be beneficial in achieving this. Take ownership and state that as a disaster has occurred the team is trying its best. This provides confidence to both the clients and the team. Social networks can be useful in disseminating information about the disaster.

Prioritize Recovery And Avoid Blame

Do not be side tracked by determining who is to blame for the disaster, unless something illegal (out of scope) has occurred. Prioritize customers and the resumption of the service by concentrating on fixing the issue.

Document The Disaster

Documenting the disaster does two things. First, it provides confidence to the customers that the team is serious about the issue. Second, it provides warnings and guidelines on how to not repeat the mistakes in the future.

Take Steps To Avoid Disaster From Recurring

Lastly, act on the vulnerabilities of the system that led to the disaster by fixing them immediately. Also, act on revising the disaster plan based on the learnings of the disaster.

How Can You Help?

- Ensure that your team is backing up essential configuration and data
- Ensure that rollback and recovery mechanisms are tested properly
- Ensure good security practices are followed and apply automation
- Ask what documentation could be missing that could help you.

Scalability

Scalability is an attribute that describes the ability of a software, network, process or organization to grow and manage increased demand. A system, business or software that is described as scalable has an advantage because it is more adaptable to the changing needs or demands of its users or clients.

Even if a system is working reliably today, that doesn't mean it will necessarily work reliably in the future. One common reason for degradation is increased load: perhaps the system has grown from 10,000 concurrent users to 100,000 concurrent users, or from 1 million to 10 million. Perhaps it is processing much larger volumes of data than it did before. By accommodating various use cases that come when dealing with scalability, we can categorize scalability in terms of 4 key metrics : performance, availability, maintenance, expenditures.

Scalability of Performance

High performance is paramount as the data and user base grows. Intuitively, adding more capacity to an application component should increase the component's performance. Application from its initial phase to long term maintenance phase should adhere to the optimized performance level, whether its linear scaling (e.g. adding more CPUs/memory/storage) or horizontal scaling (e.g. adding multiple nodes of database or application servers)

Scalability of Availability

Availability of a system means the guarantee that every request receives a response about whether it was successful or failed. In other words, every request received by a single non-failing node in the system must result in a response.

Eventually, software will grow in terms of codebase, data and users and we tend to increase the number of nodes working together to fulfill increasing demands. This introduces network partitions within a complete software ecosystem where more than one node collaborate with each other to perform certain actions. And when we have multiple nodes talking to each other, there comes a question of Consistency which states that reads are always up to date, which means any client making a request to the database/cache will get the same view of data.

The **CAP**(**C**onsistency**, A**vailability and network **P**artition tolerance**)** theorem states that a distributed system cannot simultaneously provide all the guarantees of consistency, availability, and partition tolerance. In many web-scale applications, strong consistency guarantees are often being dropped in favor of the other two.

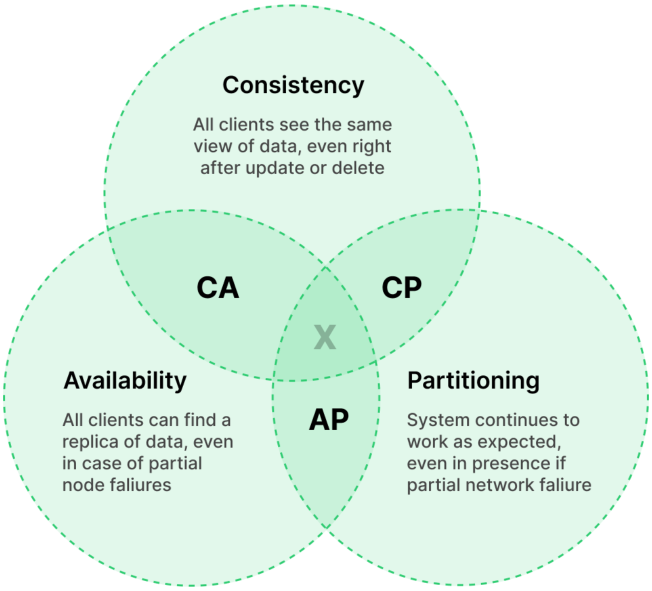
Instead, they rely on eventual consistency: at some point the system will let all users read the most recently made updates, but all will not do so immediately.

This greatly reduces synchronism in our application and allows the system to remain available to all users, even in the face of network partitions — during the partition, users may see inconsistent data, but as the partition heals the consistency of the system will eventually be restored.

Scalability of Maintenance

Software is not just written and forgotten, it must be maintained. Likewise, servers are not just started, they must be maintained.

Software maintenance is much more complicated and expensive in terms of resource, design, effort and cost. It is easy to add one or two features in the early stage of a software cycle but as software grows, new features or upgrading the feature have to be executed in regards to keeping the older feature intact. So developers should be aware about the cost of fixing bugs, addressing new use cases, adapting to new platforms, adding new features, investigating failures and maintaining the servers.


Consistency
All clients see the same view of data, even right after update or delete
CA
CP
X
Availability
All clients can find a replica of data, even in case of partial node faliures
AP
Partitioning
System continues to work as expected, even in presence if partial network faliure

Scalability of Expenditures

Software expenditures include development, maintenance, and operational expenditures. When we design a system, we need to balance making use of existing components and developing everything ourselves. Existing components rarely fully match our needs, but whatever training and additional development required to make them fit might cost less than developing an entirely unique solution.

Using industry standard technology might also make it easier to hire an expert in the future. Conversely, publishing a unique solution as open source might help us identify community members that would make great additions to our team.

Scalability Design Principles

Computer systems fail. Software fails. Hardware fails. Designs fail. Failure handling fails! Be prepared for failure, but spare end users from witnessing it too obviously. It reflects poorly on you, even if failure is inevitable. The design principles that will allow you to be prepared to scale and reduce the fate of failure are as follows:

Avoid The Single Point Of Failure

We can never just have one of anything, we should always assume and design for having at least two of everything. This adds cost in terms of additional operational effort and complexity, but we gain tremendously in terms of availability and performance under load. Also, it forces us into a distributed-first mindset. If you can't split it, you can't scale it has been said by various people, and it's very true.

Identify the point of failures. What if the API Gateway is down, will your system no longer accept the user requests or you have multiple instances of API gateway or a load balancer to entertain the incoming user request even if any one API gateway is down?

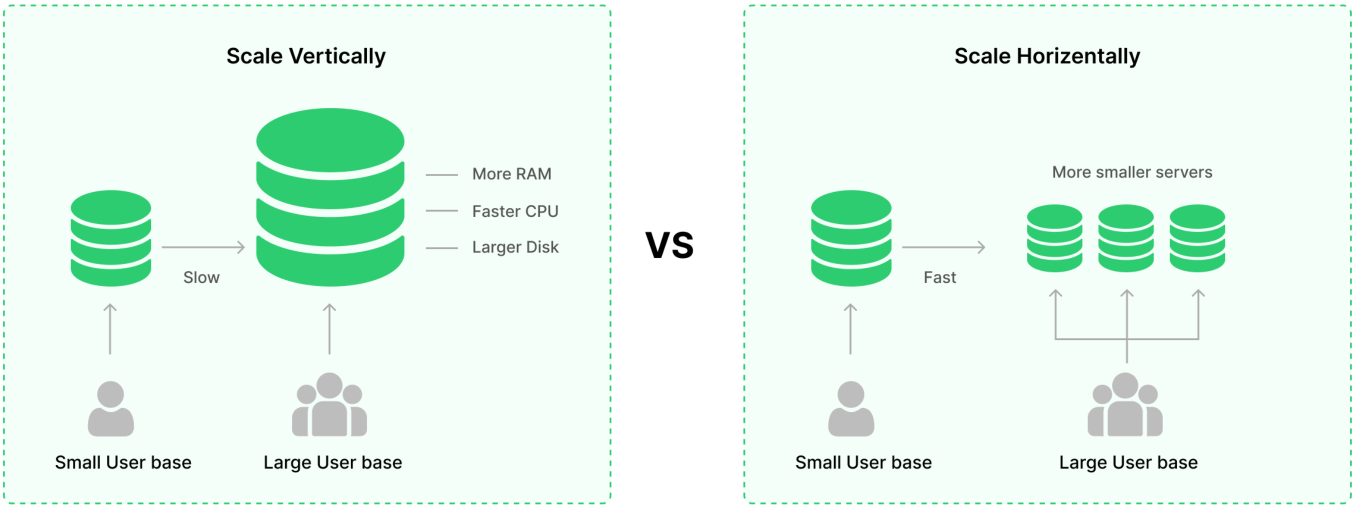
What if any of the feature/service stops working? Will your whole system stop working or rest of the feature continue working?

These all examples comprehend the scalability of availability by basically avoiding the single point of failure and splitting the system, adding multiple points such that even if one stops working, the rest of the points will function usually.

Scale Horizontally, Not Vertically

From adding code with composition and inheritance, partitioning DB into multiple smaller , as well as adding multiple application nodes in the same data center or in another geographical location, all comprehend to the principle of scaling horizontally.Vertical scaling means that we scale by adding more power (CPU, RAM, Storage) to an existing machine.

There is a limit to how large a single server can be, both for physical and virtual machines. There are limits to how well a system can scale horizontally, too. That limit, though, is increasingly being pushed further ahead. Even databases are moving in that direction. Furthermore, the cost of (vertically) upgrading a server increases exponentially whereas the cost of (horizontally) adding yet another server increases linearly.



Separation Of Concerns

Most important concepts that a software development must internalize; It is also one of the most difficult to describe, because it is an abstraction of an abstraction. Separate your view layers, presentation and logic, your business logic, your persistence layer, your cross cutting concerns (logging, securities), and your deployment configuration and scale them individually. Examples of such abstractions could be achieved architectural principles such as microservices, polylith, etc

API First

The API(Application Programming Interface) is the set of definitions and protocols for building and integrating application software. It defines what functionality an application does rather than how application does.

Design your application around the APIs. APIs cannot address all the needs of various types of clients say smartphone apps, desktop or web app. If the API does not make assumptions about which clients will connect to it, it will be able to serve all of them. Otherwise, you will end up maintaining the application as the client needs changes.

Cache As Much As Possible.

Caches are essentially storages of precomputed results that we use to avoid computing the results over and over again. This is a godsend for scalability of performance, so we must use it. For example, for an organization having 200 employees, how often the employee details like (name address, contacts) changes?

Does it make sense to hit the database every time to fetch the employee info? Usually an enterprise application will have 80% read operation and 20% write operation. When you open your facebook feed, how many api calls are read vs write?

This is where cache comes in action. That is why, for scalability of performance, we need to cache data as much as possible while keeping in mind to evict the cache according to the nature of data.Provide data as fresh as needed. Depending on your application, users might not need the most updated data right away.

Design For Maintenance And Automation

Do not under-estimate the time and effort spent in maintaining your application. Your initial public software release is a laudable milestone, it also marks when the real work begins. As we move out of the “servers are pets” era and into the “servers are cattle” era, our mindset has to change.

Do we even need to reconfigure servers anymore instead of just taking the old ones down and replacing them with new ones? What monitoring data is important to us, and what information does the data provide us with? These are the questions you should be answering for growing your product.

Asynchronous Rather Than Synchronous

Synchronous basically means that you can only execute one thing at a time. Asynchronous means that you can execute multiple things at a time and you don't have to finish executing the current thing in order to move on to the next one.

We already understand asynchronous communication perfectly in the physical world. We drop off a letter in the mail, and some time later, it arrives. Until it does, we convince ourselves that it is underway, oblivious to the complexity of the postal system. We only use personal couriers for very important messages. A similar approach should be taken for our applications. Did a user just hit submit? Tell the user that the submission went well, and then process it in the background. Perhaps show the update as if it is already completely done in the meantime. This contributes to scalability of availability.

Strive For Statelessness

Stateful means the application keeps track of the state of interaction, usually by setting values in a storage field designated for that purpose. On the other side, Stateless means there is no record of previous interactions and each interaction request has to be handled based entirely on information that comes with it.

While it may seem tempting to avoid inter-component communication by keeping track of certain state information in e.g. your application servers, don’t.! Unless we host purely static pages, we can never get away from state information. We must make sure that state information is kept in as few places as possible, and within components made for it. Web and application servers are not, but distributed key-value stores are.

Keeping it there lets you treat your web and application servers as completely replaceable instances. Moreover, it is ideal from a scalability point of view since your server fleet can much more easily be modified when any server is able to handle any client request (despite the client being in the midst of a “session”).

Reusability

Reusability of software components is a factor that largely contributes to the expansion of the IT industry itself. An alternate way to look at how reusable a piece of code is, think about how easy it would be to replace it. Reusability can be best achieved by planning ahead and is difficult to achieve when large portions of software has already been built. Reusability not just improves the team's productivity in the long run but can also help the project be flexible enough to transition from monolithic to microservices. At the least, it leads to code that is better organized and maintainable. Several best programming practices come into play to make a code reusable:

Make Code Modular

A modular code is reusable because of the separation of functionality. On planning a module, make the interfaces of the module clear. Think about how other modules will interact with it or how a user will use it. The API-first and/or Design-first principles represent this philosophy as they first lay out how a module will be used before it is even built.

Write Testable Code

The reusability of code can also be improved by working on testability. Have you noticed that when you try to write unit tests to one piece of working code, you end up refactoring 10 things? This is because some code is more testable than others. A code that does not mutate the states is easier to predict and also easier to isolate. This property of code is also known as **idempotency**. It can also be called code with less side effects. Other factors that improve testability of code are: having loosely coupled modules and having DRY code.

Use Layered Approach

Code can be organized in an infinite different ways. One way that improves reusability, readability and maintainability of code greatly is when code is separated into layers. In Web frameworks a popular layered approach is the Model-View-Controller(MVC). Several variations of this approach have come about. Another popular example is the Open Systems Interconnection (OSI) model where

communication functions and the underlying technology is broken down into 7 layers. Here are some layers that could come in handy for developers. The nomenclature of these may differ in other sources.

Name	Description
Persistence	This represents the part of the code that interacts with databases or files. The storage form could be relational or non-relational, structured or unstructured. Developers usually abstract out direct interaction with the persistence layer using ORMs
Models	Usually models are programmable objects that are direct representation of the data layer. In basic MVC, business logic goes into Models but developers usually prefer to keep Models lightweight and place business logic in a separate layer (eg: Business layer)
Views	This is the aspect seen by the user. Eg: HTML/CSS, CLI.
External Services	This represents the interaction of code with external services
Jobs/Events	This contains watchers and handlers of software-wide events. The events can run asynchronously with the rest of the software and can do background jobs like large data processing, notifications, cleaning up etc.
Controllers/API	The best practice is to keep controllers light and place only validation and redirection logic here. This generally ends up being the direct representation of API
Utilities	This should contain the business logic independent code. Eg date utils
Business Layer/Service Layer	The business logic layer, eg. eEmployee account change in an HRM, can go into operations

View-Model	The models (usually objects) that closely resemble the data that is going to be displayed can be segregated into View-Models layer

The above list may differ from project to project. The layers above are generally what ends up in a large project but developers should **NOT** be obsessed about setting them up from the start.

Refer to Standard Patterns

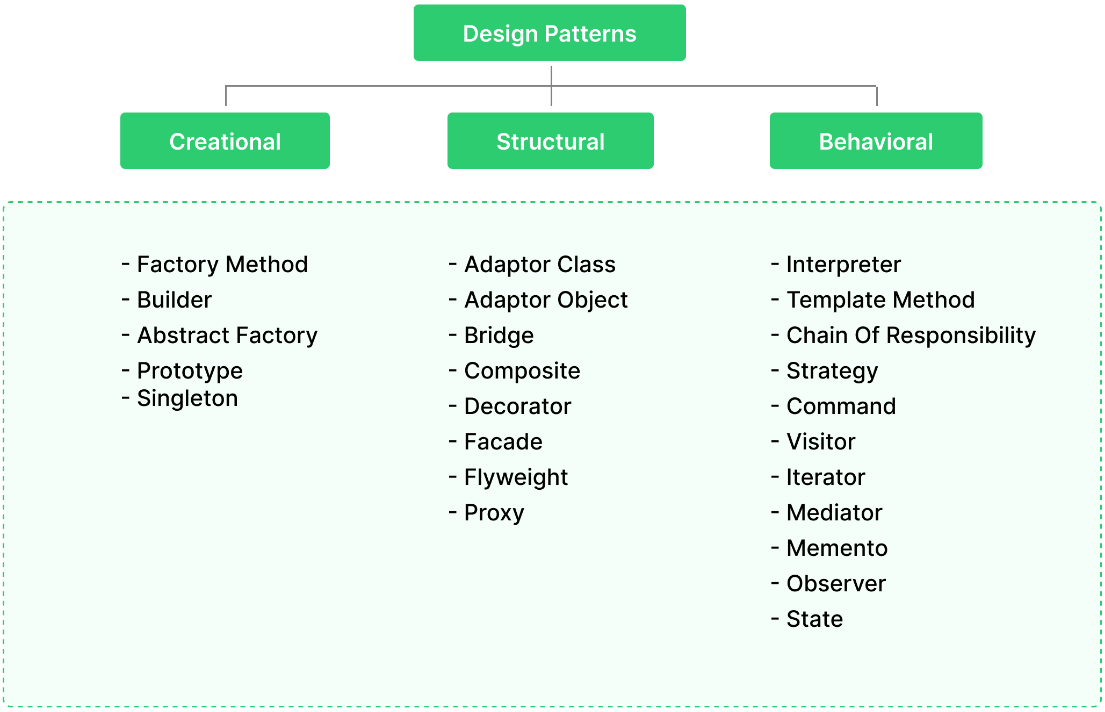


Fig: Classification Of Design Pattern

One of the most popular sources for learning about design patterns in coding is the **Design Patterns: Elements of Reusable Object Oriented Software** which is also known as the Gang of Four book. Turns out that most code organization decisions that developers make can be categorized into these 23 design patterns. These coding patterns are born out of the need for reusability and maintainability of code, and as such is beneficial to know about.

In addition to coding patterns there are some architectural patterns. These patterns help with both logical grouping and physical placement of the code:

Web development architectural patterns

Name	Description
MVC/MVP	**M**odel-**V**iew-**C**ontroller is used to separate business logic (models), display logic (view), routing and validation logic (controller). If the controller does not directly interact with the view, it is also called Presenter.
MVVM	The views use a different model called **V**iew-**M**odel which is subset of the main Model.
MOVE	Controllers are replaced by **O**perations that listen to **E**vents. Also views listen to events and update themselves
MVCs	In this pattern the model is kept light and business logic / data manipulation logic is handled within **S**ervices

Web development frameworks have come a long way. The code used to render web pages used to have the display and other logic melded together. Nowadays, there are various architectural patterns for organization of logic that you could study to segregate the way your code is placed and behaves.

Other pattern

There are several other programming and architectural patterns that you can refer to. Also note that some of these patterns can overlap within a project. For example, you may use MVC to organize your web-framework while using Domain Driven Design to develop your models. These design practices can help in code organization, and at the same time help you conceptualize a piece of software that may have become large over time.

Name	**Description**
ETL	Separates functionality in data processing into Extraction (reading of data from disparate sources to bring them into one place), Transform (mapping and fitting of data into a common easier to use format), and Load (loading of data to warehouses for creation of reports or for use by business intelligence tools)
Event Driven	Two types of actions take place. 1. Publishing of events to an event interface or Queue 2. Reading off the events and taking appropriate actions
Data Pipelining	Functions are organized into filters and pipes. Pipes are used to read data from a source or write to another pipe or data sink whereas, filters are used to transform the data within pipes
Map Reduce	Data is first broken down or replicated into multiple pieces (**M**apping). The mapped data is individually processed in multiple machines or threads then brought together (**R**educe)
SOA and Microservices	Both Service Oriented Architecture (SOA) and Microservices pattern break down functionality and components of the software into smaller components which can generally be deployed separately
Domain driven design	Involves first breaking down a complex piece of software into components separated by business domain. The components can interact with one another via services

Abstraction

Abstraction can benefit reusability of code by creating a framework that does the grunt work, so that, developers can focus on implementing the business logic. A good abstraction also allows extending, overriding or changing just the needed aspects of code keeping the overall logic constant. Examples of good abstractions:

- A Tree Map works just like a Map in Java and automagically helps developers keep their keys sorted.
- In Cascading, a map reduce abstraction framework written in Java, the transformation and conversion of data is represented by Taps, Pipes and flow such that developers don't need to do map reducte at all.
- A file system abstraction can make the underlying filesystem indistinguishable and allows code to interact with the filesystem with basic create, read, update and delete operations

Abstractions can also be dangerous if created prematurely or without an overall understanding of the system. Here are a few ways to make sure abstraction is not horrible.

Avoid the YoYo Problem

When you try to understand or modify a piece of code and you end up dealing with a chain of classes linked by hierarchy, you are dealing with a Yoyo problem. Hierarchy should be shallow. Any noteworthy saying is “Composition over Inheritance”, which states that instead of inheriting classes to extend their behavior, include a reference or instance of the class in the new class. To put it simply, choose wrapper classes instead of extending when you can.

Avoid the "buy a banana, get a gorilla" problem

Fat classes can instantiate too many other classes can lead to too many dependencies being loaded when a single class is sought. Try to keep the classes light.

Don't write it if YAGNI

If "you ain't gonna need it" anytime soon, don't write it. This helps eliminate unnecessary abstractions. If it can be done easily in the future don't do it now unless necessary.

Avoid being obsessive about DRY

DRY (Don't Repeat Yourself) code is good but do not be obsessed about keeping code DRY. Sometimes, making the code DRY involves creating complex abstractions. In an alternate principle called WET (Write Everything Twice), the rule of thumb is to not worry about DRY code unless you have repeated code in more than two places.

Avoid unnecessary classes

Classes should be mind sized i.e its purpose should be clear and singular, which is easy to comprehend. If your classes have too many unrelated public methods, they may not have a well defined interface and can be difficult to make sense of. If you have to do this, then explicitly mark them as factory classes. Otherwise, decompose large classes into smaller related ones. On the other hand, if your classes just have a method or two and do nothing else, the classes may be replaceable with functions.

Testing

Every topic in this handbook is aimed at improving quality. Testing refers to the process of checking if that quality is ensured.

The goal of testing is as follows:

Verification means to ensure that We are developing the product/system/software correctly, where we need to verify all specifications are implemented correctly

Validation means to ensure that we are developing the right product/system/software where need to examine the specifications from the user's point of view Ensure that the product meets the need of end user

This topic will primarily talk about what tests can developers conduct to improve the level of quality in their product. That being said, who is actually responsible for quality?

Is 'Quality' A Shared Responsibility?

Tests performed by the Quality Assurance team cannot solely ensure the quality of a product. Seemingly insignificant aspects such as defining clear requirements (business analysts), setting practical deadlines (project managers), and performing unit / basic testing (developers) equally contribute to the overall end product.

As such 'Quality' is a shared responsibility where each and every member has their own roles and responsibilities:

- Project managers are responsible for processes used to gather requirements, ensuring that the team has sufficient time to develop them, fix issues and getting any problems out of the developers and testers.
- Developers work against the requirements as do the testers. They write the code to develop the software according to the requirements and fix any issues. However, what if the requirements are wrong?
- Software testers or Quality assurance identify bugs and prevent their recurrence.They verify that requirements are implemented correctly. They perform various tests to verify and validate the requirements and report any issues as 'bug reports' which developers use to fix them.

When to Start Testing?

It may seem that testing is a phase that is conducted only after a usable product has been developed. In actuality, testing activities can be started as soon as the Software Requirement Specifications has been prepared.

As development commences, testing can take place at random times during the development process. One way to add method to the haphazardness caused by this is to create a testing strategy containing when, what and how of testing. It gives structure to your tests. It also allows testing phases to be estimated during sprint planning eliminating scenarios such as the "you did not say i need to test this" problem.

How to Conduct the Test?

Conduct Manual Tests

During and after development, a developer carries out several tests. The following types of tests should be considered to ensure that the task was successful and did not cause any adverse effect on the overall product.

Functional testing

Test the developed functionality according to the specified requirements, Follow the test cases prepared by QA member and validate the implementation And Perform sanity or smoke tests after modification in code

Non-Functional testing

Test the performance of the application in developer or local machine, Verify the design, Test that the developed functionality works as expected in various specified platforms, browsers, devices etc.

Regression testing

It is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software.Things to consider during Regression testing are:

Is the program still excitable?

Making a full build involving, deploying to the target platform(s) and performing a smoke test.

Does the change have the desired effects?

Is the bug resolved? Does the feature you implemented behave as specified? Are edge cases covered?

Does the change have no undesired side effects?

Make sure that what you just did only does what you intended it to do. It should not change or break other features.

Conduct Automated Tests

Significant time can be saved if the tests are automated. Two types of automated tests are discussed in the following sections

Unit test

It is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc.

Unit testing allows you to refactor code or upgrade system libraries at a later date and make sure the module still works correctly. Unit tests detect changes that may break a design contract. They help with maintaining and changing the code. So always write unit tests.

When - Unit Testing is the first level of software testing and is performed prior to Integration Testing.

Who - It is normally performed by software developers themselves or their peers. In rare cases, it may also be performed by independent software testers.

Integration tests

It is a level of software testing where program units are combined and tested as groups in multiple ways. It can expose problems with the interfaces among program components before trouble occurs in real-world program execution.

Perform load and performance tests

By simulating the HTTP requests generated by hundreds or even thousands of simultaneous users, you can test your web server performance under normal and excessive loads to ensure that critical information and services are available at speeds your end-users expect. This is the advantage of load/stress test.

Conduct Reviews

Review is a static testing process conducted to find the potential defects in the design of any program. A code review is a process where someone other than the developer of a piece of code examines that code. Code reviewer should look at:

Topic	Description
Design	Is the code well-designed and appropriate for your system?
Functionality	Does the code behave as the author likely intended? Is the way the code behaves good for its users?
Complexity	Could the code be made simpler? Would another developer be able to easily understand and use this code when they come across it in the future? Tests: Does the code have correct and well-designed automated tests?
Naming	Did the developer choose clear names for variables, classes, methods, etc.?
Comments	Are the comments clear and useful?
Style	Does the code follow our style guides?
Documentation	Did the developer also update relevant documentation?

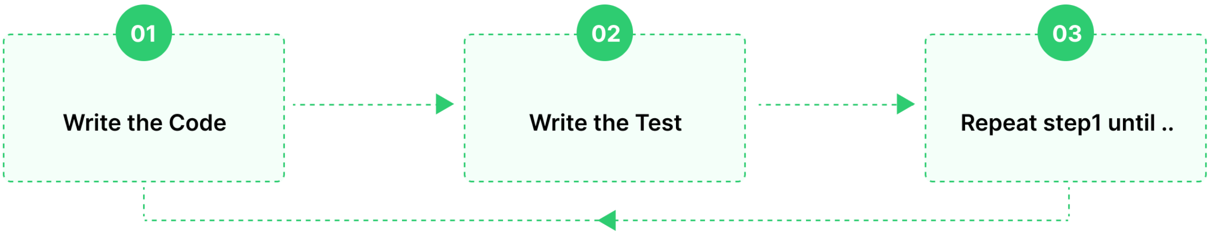
Walkthrough is an evaluation process where the author presents their developed artifact to an audience of peers. Peers question and comment on the artifact to identify as many defects as possible.

Inspection is used to verify the compliance of the product with specified standards and requirements. It is done by examining, comparing the product with the designs, code, artifacts and any other documentation available.

Approaches For Writing Automated Tests

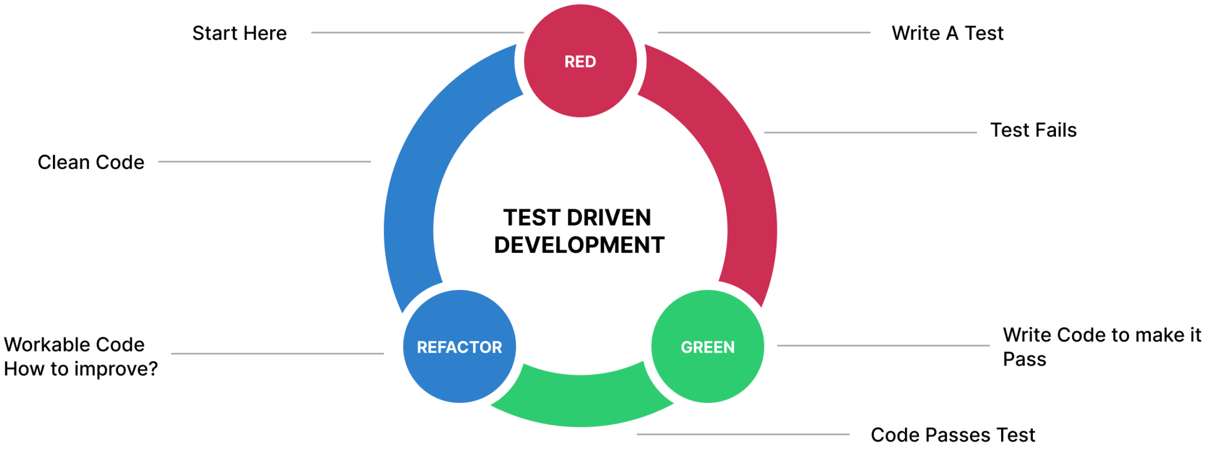
Test-After Development or Plain Old Unit Testing (POUT)

This approach suggests the following steps:



Test Driven Development (TDD)

TDD is a software development process in which tests are written before any implementation of the code. TDD has a test-first approach based on repetition of a very short development cycle. According to it, each new feature begins with writing a test. The developer writes an automated test case before he/she writes code to fulfill that test. This test case will initially fail. The next step will be to write the code focusing on functionality to get that test passed. After these steps are completed, a developer refactors the code to pass all the tests.



Behavior Driven Development (BDD)

BDD is an extension of TDD that emphasizes developing features based on a user story and writing code that provides a solution to real problems. The simple language used in the scenarios helps even non-technical team members to understand what is going on in the software project. This helps and improves communication among technical and non-technical teams, managers, and stakeholders.

Tests in BDD can be written in the following format:

Feature: LMS Login

User Story:
As a Leapfrog Employee,
I should be able to login to LMS
So that I can login to dashboard

Test Scenario:
Given that I am a registered Leapfrog Employee
When I enter valid login credentials
And click login button
Then I should be able to view dashboard
And see the lists of Employees

Best Practices For Automated Tests

- Follow (Arrange/Act/Assert) or (Given/When/Then) pattern.
- Arrange all necessary pre-conditions and inputs using various test data preparation techniques.
- Act on the object or method under test.
- Assert that the expected results have occurred.
- Check one feature/use case at a time.
- Avoid adding business logic to the test to test.
- Write a test such that it provides baseline /guidelines for refactoring. If the test failed, then it states refactoring also failed.
- Avoid erratic tests : Sometime passes, sometime fails.

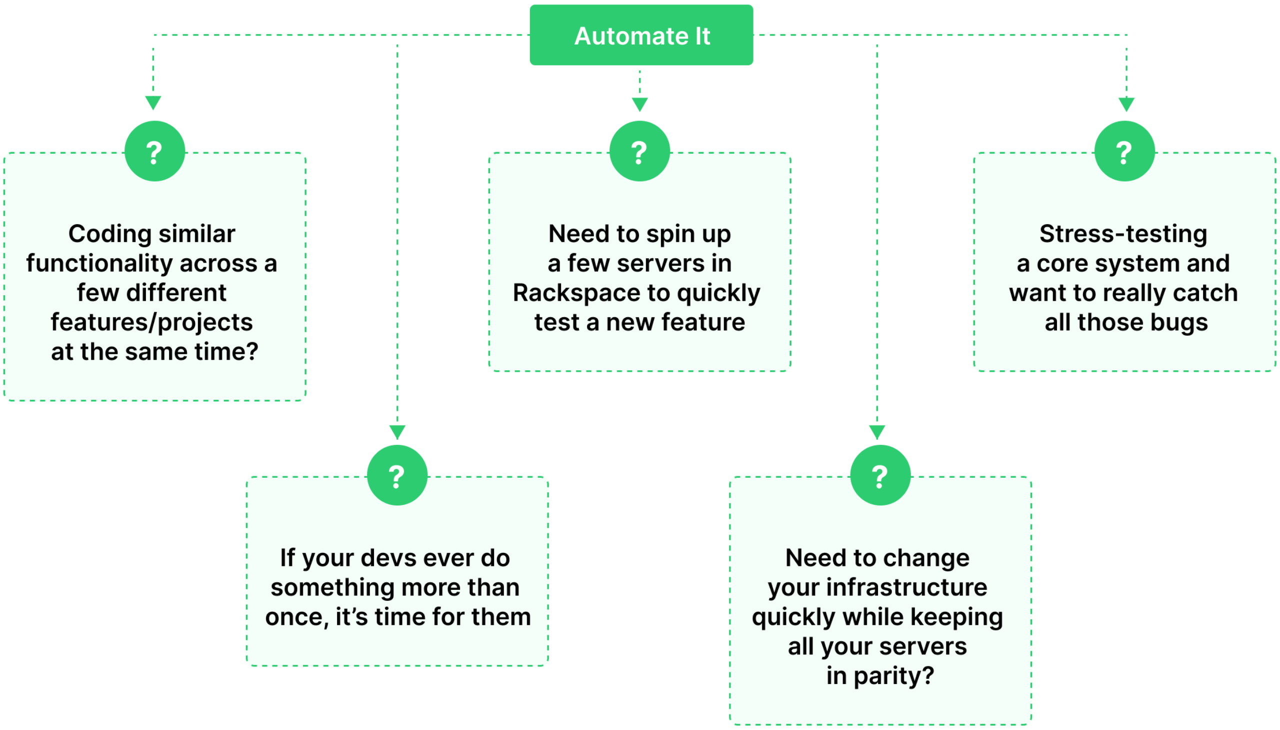
- Avoid tests that only runs on one's developer machine.
- Test state over behaviour.
- Use behaviour verification only if there is no state to verify.
- Refactoring is easier due to less coupling to implementation.
- Tests should be fast executable, isolated, repeatable.
- Display clear fail reason.
- Is Test too complicated? Either refactor the code or divide into multiple tests.
- Write a test such that it provides documentation to other developers. (What values does the function can take, and what output it gives under certain inputs?)

Periodically Benchmark Your Application Code

It is a good practice to set aside a couple of hours in every sprint to perform benchmark testing of your codes and application as a whole. There are lots of tools out there which helps you in this. In fact it’s likely that the IDE that you are using has those tools embedded within.

Automation

Software automation is the practice of reducing repetitive steps and executing them automatically that come on any stage of development life cycle. For instance, automation can be introduced in project setup steps, day to day coding, easy deployment and delivery process. Automation mainly aims to minimize human intervention as much as possible to lower human errors, speed up development and delivery cycles.



Automate It

Coding similar functionality across a few different features/projects at the same time?

If your devs ever do something more than once, it's time for them

Need to spin up a few servers in Rackspace to quickly test a new feature

Need to change your infrastructure quickly while keeping all your servers in parity?

Stress-testing a core system and want to really catch all those bugs

Automation Best Practices

Minimize steps for project set up

It is crucial to have the project up and running in a matter of minutes even in a new development machine. As such, we need to take care of a few things that will eliminate unnecessary rummaging through documents, waiting up on existing developers(teammates), hits and trials etc, when we want to setup the project anew. Here are some of the ways to minimize the project setup effort:

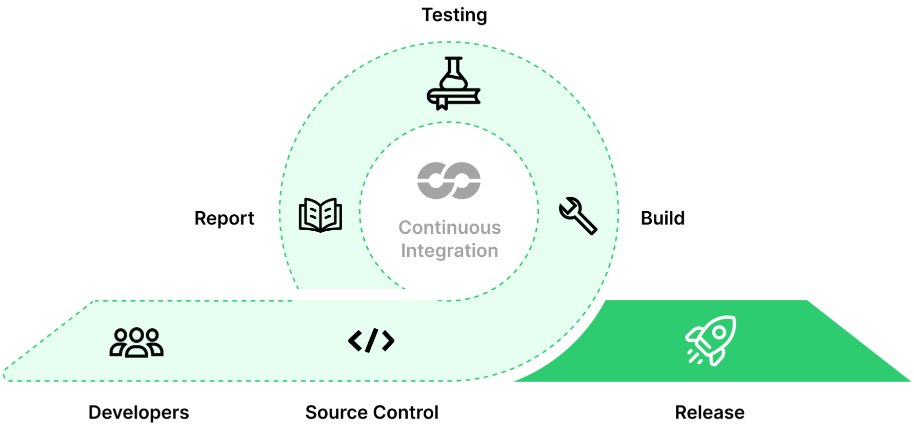
- Containerize the project. e.g Docker
- Check if parameters, configuration and steps to make project development ready is minimal. Projects that usually require hours and days to configure are an example of a bad architecture.

- Setup standard source code version controls. e.g Github
- Make the environment, libraries and components required for development easily available.
- Minimize the number of files to execute project startup scripts.
- Execute long running process asynchronously, if needed. eg: ETL etc

Continuous Integration

Continuous Integration is the automated practice of merging all developer working copies to a shared mainline or branch several times a day, ensuring that all the works are on the same page and the changes are dev releasable. It embodies:

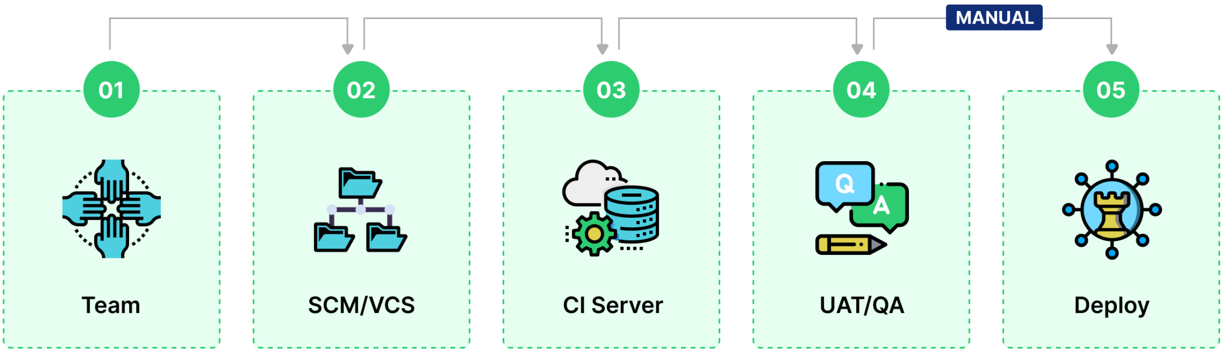
- Code compiles successfully,
- Has minimum warnings and errors,
- Validate the code quality,
- Run unit and integration test
- Generate report in terms of execution time, test run, number of merges, quality codes



Continuous Delivery

Continuous Delivery is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time and, when releasing the software, doing so manually.This does not mean the code or project is 100% complete, but the feature sets that are available are vetted, tested, debugged and ready to deploy, although you may not deploy at that moment. It aims at building, testing, and releasing software with greater speed and frequency. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery. It embodies

- Create multiple environment checkpoints and deploy to every environment (dev/qa/uat/staging/production) the same ways but with different configuration parameters as per the environment.
- Version controlling the delivery configuration and steps.
- Build binaries/bundles at once.
- Deploy in production like environment.
- Instant reflection and propagation.
- Smoke test the deployment.

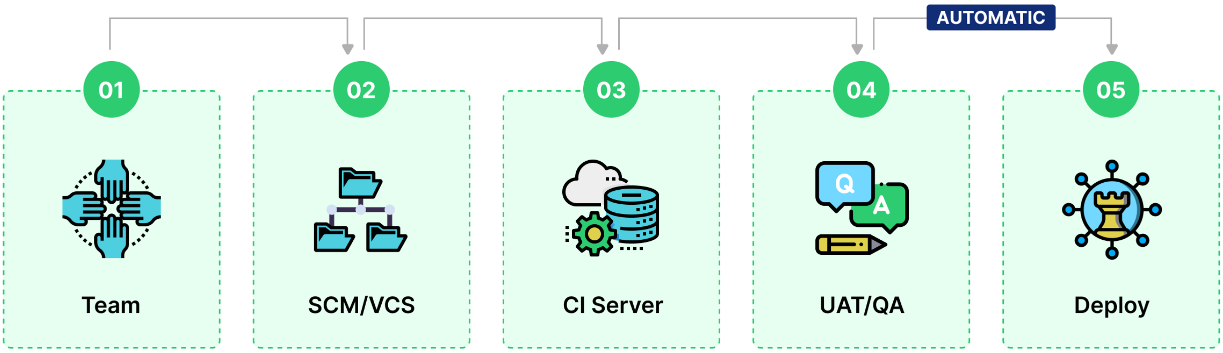


Continuous Deployment

It is a strategy for software releases wherein any code commit that passes the CI/CD phase is automatically released into the production environment, making changes that are visible to the software's users. It is a process that automatically deploys the results of Continuous Delivery into the final production environment, usually every time a developer changes code.

- So, with this approach, the quality of the software release completely depends on the quality of the test suite as everything is automated. It embodies:
- Every previous steps from CI/CD is run successfully.

- Having a rollback strategy.
- Having at least two identical production environments (blue-green deployment).
- Automating backups of data and crucial information.



Automate Monitoring

Various aspects of projects should be monitored like error, system and resource usage, overall health status etc. We can and should automate these for efficient issue tracking before the incident occurs.

By implementing proper logs and metrics, we can monitor incoming requests, outgoing response, errors, resource usage, and health status. For example, we can implement mechanisms to automatically identify that the system is about to reach upper bound or exceeded in terms of resource usage.Say, we set the upper limit threshold 85% CPU usage/storage/memory or N number of requests at a given time frame. As soon as the threshold exceeds, alert the designated people by various means like email, SMS, so that preventive actions can be taken on time.

Similar scenarios can be applied when application is generating a lot of error logs, if some service is unhealthy or down or if some event happened that is undesirable.

Automated testing

Automated Testing means using an automation tool to execute your test case suite. On the contrary, manual testing is performed by a human sitting in front of a computer carefully executing the test steps.

The automation software can also enter test data into the system under test, compare expected and actual results and generate detailed test reports. You can refer to the testing section for more details.

References

- Darius S. (2018) Security by Design Principles https://blog.threatpress.com/security-design-principles-owasp/

- Elastisys AB (2015) Scalability-design-principles https://elastisys.com/2015/09/10/scalability-design-principles/

- Noel Ceta (2018) All You Need to Know About UML Diagrams: Types and 5+ Examples https://tallyfy.com/uml-diagram/

- Ham Vocke (2018) The Practical Test Pyramid https://martinfowler.com/articles/practical-test-pyramid.html

- Google (NA) Code Review Developer Guide https://google.github.io/eng-practices/review/

- Vskills (NA) Learning TDD https://www.vskills.in/certification/tutorial/tag/tdd/

- Github Contributors (2020) Node Best Practices https://github.com/goldbergyoni/nodebestpractices/blob/master/README.md

- Smart Bear (NA) Automated Testing Best Practices and Tips https://smartbear.com/learn/automated-testing/best-practices-for-automation/

- Urs Enzler (2014) Clean Code Cheat Sheet https://www.planetgeek.ch/2014/11/18/clean-code-cheat-sheet-v-2-4/

- Contributors (2004) The Zen of Python https://www.python.org/dev/peps/pep-0020/

- Quora Answers (2010) What is the YoYo problem in Object Oriented Programming? https://www.quora.com/What-is-the-YoYo-problem-in-Object-Oriented-Programming

- John D. Cook (2011) You wanted a banana but you got a gorilla holding the banana https://www.johndcook.com/blog/2011/07/19/you-wanted-banana/

Leapfrog Technology, Inc is a global full-service technology services company. Founded in 2010, we have offices in Seattle, Boston and Kathmandu. With a high performing team of over 200 engineers, designers, and product analysts, we help companies solve technology innovation and digital transformation challenges.

We have instilled a culture of continuous learning in our people. Our mission is to be the best and most trusted technology services company to execute clients digital vision.

Developers Guide

Handbook

leapfrog

Leapfrog Technology Nepal Pvt. Ltd.

Siddhi Mani Bhawan, Charkhal Rd, Dillibazar,
Kathmandu, Nepal

www.lftechnology.com