# SOFTWARE FRAMEWORKS – AN OVERVIEW

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

# CONTENTS

- Preliminaries
- Programming vs Software Engineering
- Generic functionalities in Software Development
- Frameworks: what are they?
- How does Frameworks help with the Software Engineering Approach?
- Framework vs. Libraries
- Why use frameworks?
- Advantages of using frameworks
- Limitations of using frameworks

# PRELIMINARIES

- What is this course?
  - **PAF (IT3030)** introduces the fundamentals of full stack web development with an emphasis on application of good software engineering practices and Industry standards.
- Learning Outcomes of this course?
  - After completing this module, you will,
    - Know the basics of  Software Frameworks
    - Apply Industry standard software development best practices
    - Understand the REST architectural style
    - Develop Java based REST web services (Back-end)
    - Develop JavaScript based Interactive web applications (Front-end)
- Module Outline
- Assessments

# PROGRAMMING VS SOFTWARE ENGINEERING

- What is Programming?
  - Programming simply means writing Code to create an application.
- What is Software Engineering?
  - Software engineering is the end-to-end process of applying good engineering principles and software development practices to produce quality software.
- Programming vs Software Engineering
  - Software Engineering is **NOT** just coding. Coding is just one part of a large process.
  - **Q: What other practices come under software engineering?**

# PROGRAMMING VS SOFTWARE ENGINEERING

- What other practices come under software engineering?
  - Analyzing user requirements and designing the software to meet those requirements.
  - Communicating with all the stakeholders to resolve issues and update them on the progress.
  - Choosing the software architectures, patterns, algorithms, coding techniques, technologies used to realize the software design based on sound design considerations.
  - Quality Assurance (QA) of the developed software to ensure that the software meets the client requirements and acceptable reliability, performance and security considerations.
  - Deploying/ releasing the software for actual use.
  - Maintaining the developed software as needed after release/ deployment.

# GENERIC FUNCTIONALITIES IN SOFTWARE DEVELOPMENT

- Generic functionalities in Software Development
  - Most (web) applications have a lot of features in common.
    - Eg: User Authentication, routing, Database connectivity, Scheduled jobs etc.
  - There is little point in rewriting all the functionalities (including the common ones) from scratch every time new software is developed.
  - What if the generic functionalities can be collected and made reusable?

# FRAMEWORKS: WHAT ARE THEY?

- A framework is an integrated set of software artifacts (such as classes, objects, and components) that collaborate to provide a reusable architecture for a family of related applications [1].
- A framework enables a coding environment that contains low-level libraries to address conventional coding issues. The objective of a framework is to deliver faster development of an application. This includes everything we need to build large-scale applications, such as templates based on best practices [4].
- A framework is a foundation for developing software applications. Software engineers and developers use a framework as a template to create websites and applications. Developers do this by adding code to a framework, then personalizing it for their specific purpose [5].
- A software framework is an abstraction in which software, providing generic functionality, can be selectively changed (**extended**) by additional user-written code, thus providing application-specific software [6].

# HOW DOES FRAMEWORKS HELP WITH THE SOFTWARE ENGINEERING APPROACH?

- How does Frameworks help with the Software Engineering Approach?
  - Architectural Patterns (eg: MVC, MVVM)
  - Design Patterns
    - Inversion of Control
  - Standards/ Best Practices (Naming conventions, Coding standards, Directory structures etc.)
  - Tooling Support (Package managers, Dependency managers, Boilerplate code generation)
  - Testing Support

# FRAMEWORK VS. LIBRARIES

- Certain features make a framework different from other library forms, including the following [7]:
  - Default Behavior: Before customization, a framework behaves in a manner specific to the user's action.
  - Inversion of Control: Unlike other libraries, the global flow of control within a framework is employed by the framework rather than the caller.
  - Extensibility: A user can extend the framework by selectively replacing default code with user code.
  - Non-modifiable Framework Code: A user can extend the framework, but not modify the code.

# WHY USE FRAMEWORKS?

- The designers of software frameworks aim to facilitate software developments by allowing designers and programmers to **devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time** [6].
- The aim of frameworks is to provide a common structure so that developers don't have to redo it from scratch and can reuse the code provided. In this way, frameworks allow us to cut out much of the work and save a lot of time.
- To summarize: there's **no need to reinvent the wheel**.

# ADVANTAGES OF USING FRAMEWORKS

- Improved coding, easy code reusability and ease of debugging code compared with from-scratch developments.
- Community support.
- Accelerated development (if the programmer is familiar with the framework).
- Often provides caching and optimized processes for resource intensive tasks.
- Enables faster methods for development with less code (boilerplate code etc.).
- Better compliance to accepted security standards.

# LIMITATIONS OF USING FRAMEWORKS

- Learning the Framework and not the programming language prevent programmers from gaining an in-depth understanding of the programming language.
- Users of a framework are forced to respect the limitations and conventions imposed by its design.
- Options to tweak functionalities are limited.
- A framework will come with everything required to satisfy a wide range of use cases, not all these features will be used for a given project.
- The right framework for the application should be chosen, or else performance and user experience may be impacted negatively.
- We must be up-to-date with new/deprecated features in every version.

# SOME EXAMPLES

- Spring Boot (Java)
- Express.js (with Node.js)
- Laravel (PHP)
- .NET Core (C# etc.)
- Angular (Typescript)
- Vue.js (JavaScript)

# REFERENCES


1. Software Engineer vs. Programmer: What's the Difference?
2. Frameworks: Why They Are Important and How to Apply Them Effectively
3. What is a Web Framework, and Why Should I use one?
4. The Difference Between a Framework and a Library
5. What Is a Framework? (Definition and Types of Frameworks)
6. Software framework
7. Techopedia Explains Software Framework

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# VERSION CONTROLLING & WORKFLOWS WITH GIT - I

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

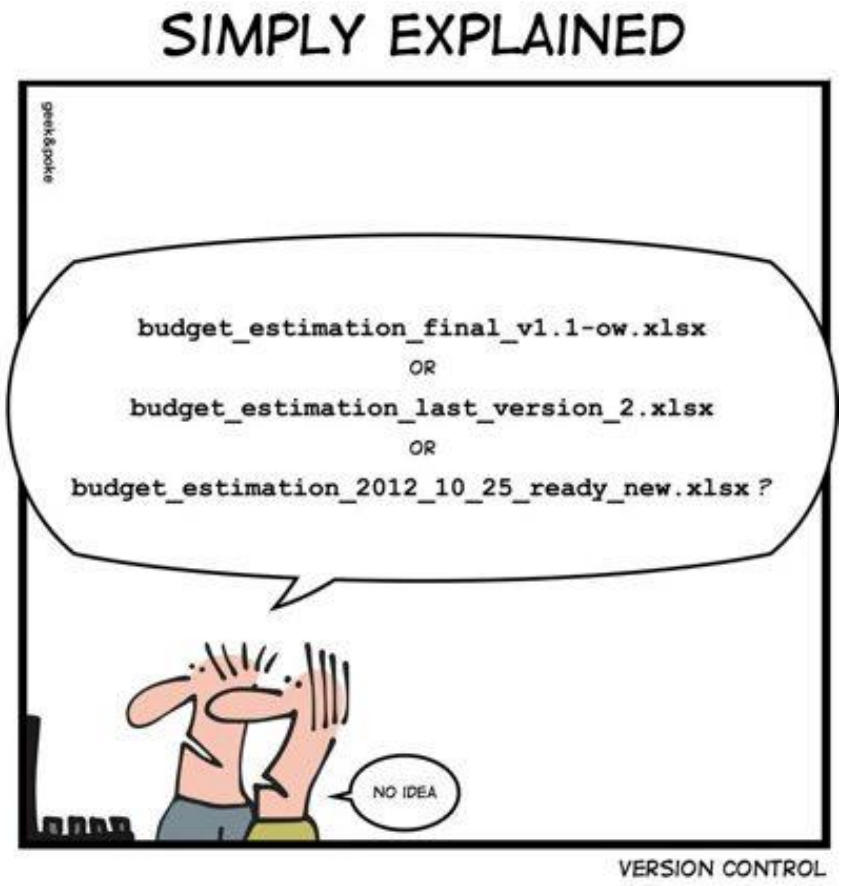
# CONTENTS

- Version Controlling: What?
- What you know about Version Controlling
- Version Controlling: Scenario Discussion
- A bit of history on Version Controlling Systems (VCS)
- Git: the de-facto standard today in VCS
- Git Basic Concepts
- Conclusion

# VERSION CONTROLLING: WHAT?

- “Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.” [1].

# WHAT YOU KNOW ABOUT VERSION CONTROLLING

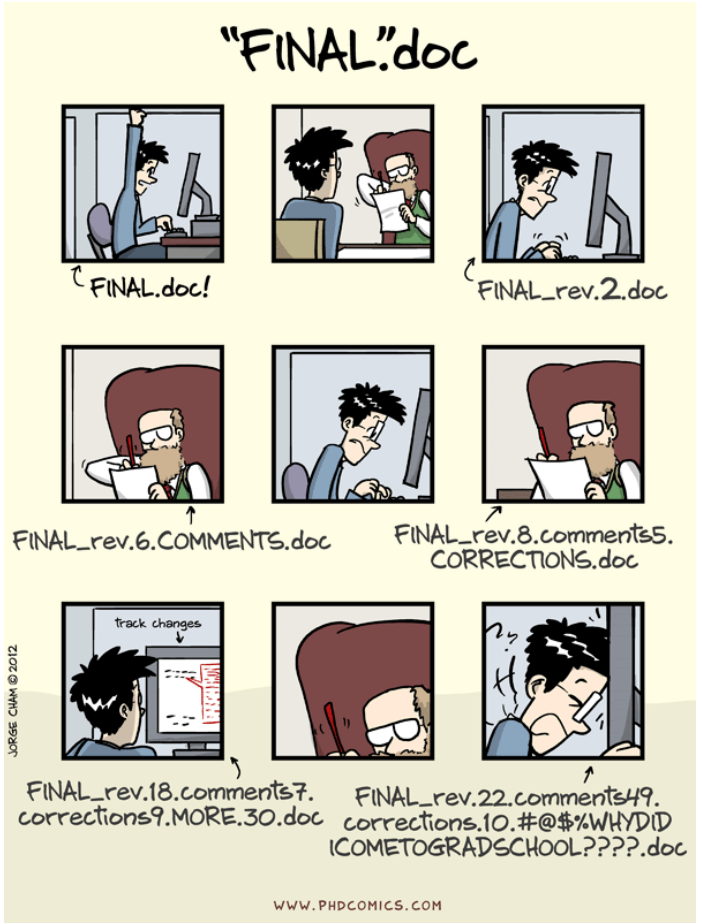



Source: https://www.datamation.com/trends/tech-comics-version-control/

# WHAT YOU KNOW ABOUT VERSION CONTROLLING

- Q: What methods of Version Controlling are you familiar with?
  - Keeping multiple copies of files with different names
  - Google Drive/ OneDrive
  - Undo/ Redo Buffer

- Now, let's see how one student named Amith does version controlling.




Source: https://swcarpentry.github.io/hg-novice/01-basics/

# VERSION CONTROLLING: SCENARIO DISCUSSION

- Suppose Amith is working on one file called numbers.py for an assignment.
- Amith has worked on the file for several days now, and has saved multiple copies of the file, i.e., numbers1.py, numbers2.py, numbers3.py and numbers.py.
- numbers.py is the latest version of the file. Whenever a new update is to be made to the file, first the existing numbers.py is renamed to numbersX.py (X = {1, 2, 3, 4…}) and then the numbers.py is saved with the most recent change.
- Amith needs to submit the assignment now, but at the last moment he discovers that the numbers.py (most recent version) throws a serious error.

# VERSION CONTROLLING: SCENARIO DISCUSSION

- **Q1:** What can Amith do?
- **A:** Amith can submit the one before (e.g.: numbers10.py).

- **Q2:** What if the most recent version (numbers.py) has some good changes too, which Amith thinks needs to be in the submitted file?
- **A:**
  - Compare numbers10.py and numbers.py and Copy the good parts.
  - Paste the copied code as fit in the appropriate places in numbers10.py.
  - Replace the numbers.py with this modified file.

# VERSION CONTROLLING: SCENARIO DISCUSSION

- The assignment was submitted. Now, Amith needs to keep this numbers.py and all its previous versions for future submissions too.
- However, Amith has a nagging feeling about what will happen if the HDD of his laptop fails.

- **Q3:** What can Amith do address this very valid concern?
- **A:** Copy all the versions of numbers.py to a Cloud storage location (OneDrive/ Google Drive).

# VERSION CONTROLLING: SCENARIO DISCUSSION

- What are the pros of above approach?
  - All the versions are safely stored.
  - The files can be accessed from anywhere now.
- The cons?
  - Considering accessing from anywhere: If Amith develops two versions of numbers.py (both named the same), one at the computer lab in his university (then uploaded to the cloud) and another numbers.py separately on his laptop, Amith runs the risk of replacing the version on laptop with the other if he uploads the first file without the name being changed when his laptop syncs with the cloud storage.

# VERSION CONTROLLING: SCENARIO DISCUSSION

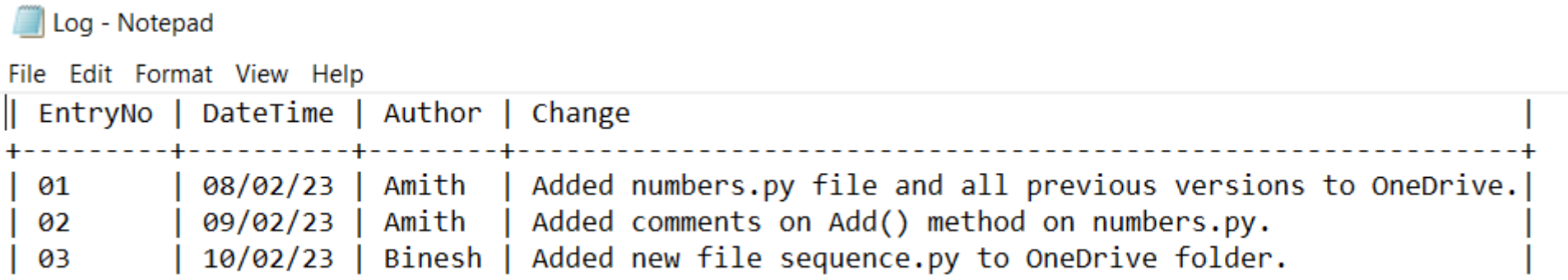
- All in all, Amith now has a working Version Controlling System.
- His system can,
  - **Revert** to a past version.
  - **Compare** two file versions.
  - **Push** changes done from anywhere to a secure remote location on the cloud.
  - **Pull** changes from the cloud to anywhere.
  - **Merge** different versions of files.

# VERSION CONTROLLING: SCENARIO DISCUSSION

- Now, Binesh joins Amith as the future assignments are to be done as pair work.
- It is more important than ever to follow a strict discipline when using Amith's system of version controlling as there are other people involved (than Amith).
- However, just focusing on naming the files correctly (to avoid overwrites) is not enough.
- Just seeing the files will not give each developer a high-level idea about what each other is doing.

# VERSION CONTROLLING: SCENARIO DISCUSSION

- Solution?
  - A log file!
- How?
  - The log file should contain information on **who** did **what when** so everybody would be aware of what is happening.



# VERSION CONTROLLING: SCENARIO DISCUSSION

- Does Amith's method solve the problem of version controlling he had earlier?
  - Yes.
- Does that mean it is an easy system to follow?
  - Not really. There are several specific rules the users of the system must follow in order to make it work.
  - As the number of users of this system increase, so does the chance of someone making a mistake and ruining the system for everybody.
- Should he follow this approach then?
  - Yes, it's a good system. But the management of the system should not be given to the users. It should be automated.
- Are there such automated systems already?
  - Yes! That's what version controlling systems like GIT, SVN and Mercurial do.

# A BIT OF HISTORY ON VERSION CONTROLLING SYSTEMS (VCS)

- The first proper Version Control System (VCS), named *Source Code Control System (SCCS)* was released in 1973 by Bell Labs.
- It was a first-generation VCS, which are now collectively known as **Local Version Control Systems**.
- Local VCS were intended to track changes for individual files and checked-out files could only be edited locally by one user at a time.
- Facilitating collaboration among large teams was problematic.

# A BIT OF HISTORY ON VERSION CONTROLLING SYSTEMS (VCS)

- To address the problem of collaboration, a new type of VCS were developed.
- This second generation of VCS are collectively known as **Centralized Version Control Systems.**
- Centralized VCS allowed users to refer files stored on a centralized repository through a network and allowed multiple users to use the files concurrently.
- Follows a *Client-Server* approach.
- However, they would all have to commit their code to the centralized repository, which would require network access.
- A central authority is required to maintain the Centralized repository.

# A BIT OF HISTORY ON VERSION CONTROLLING SYSTEMS (VCS)

- With the advent of open-source software came the third generation of VCS. They are collectively known as **Distributed Version Control Systems**.
- The earlier Centralized VCS required a central authority to maintain the centralized repository. No such authorities are usually found in open-source projects.
- Instead, there are just contributors who may develop features which may or may not be relevant to the main open-source project.
- Therefore, Distributed VCS have *peer-to-peer (P2P)* approach to version controlling.
- Distributed VCS are widely used today.
- De-facto standard in Distributed VCS now?
  - **Git!**

# GIT: THE DE-FACTO STANDARD TODAY IN VCS

- Created by Linus Torvalds (creator of Linux) in 2005.
- Pronounced as *Ghi-T*. Not *Ji-T (*A "git" is a cranky old man. According to online sources, Linus meant himself).
- Originally designed to do version control on Linux kernel.
- Git supports non-linear development (thousands of parallel branches) and is fully distributed.

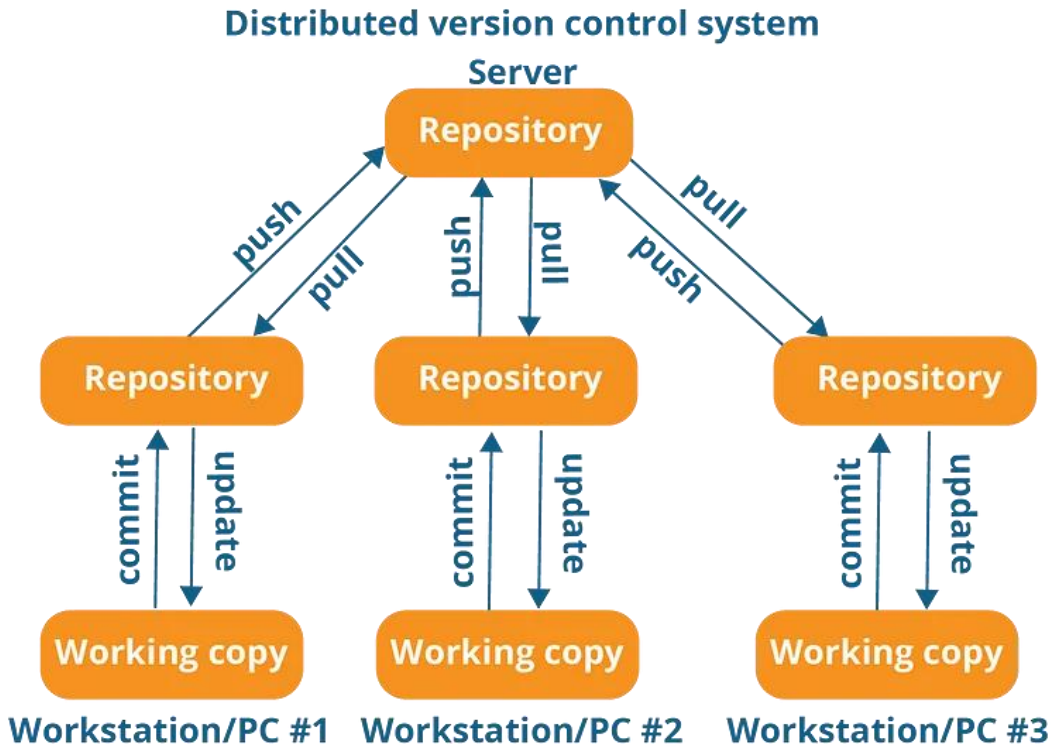
# GIT BASIC CONCEPTS

- As mentioned, git does not necessarily need a Centralized Repository to work (It can work in somewhat centralized manner too, which we will see).
- Every user maintains a **local repository** of their own.
- When starting out they will obtain this local repository,
  - from another user or,
  - they may **clone** it from a central location or,
  - They may even initialize a brand-new repository from scratch.
- This repository will contain what others have done originally, what the user has done themselves and even work of others as selected by the user.
- They can **commit** and **update** their local repository without any interference from any others.

# GIT CONCEPTS

- Git has two types of repositories.
  - Local repositories – which is owned by each user.
  - Remote repository – a centralized location accessible to all users. A single source of truth.
    - The users will **push** their new updates to this repository so those can be shared with others as well.
    - The users will **pull** new updates from others from this repository.
  - Why do we need Remote repositories?
    - Scenario Discussion: The problem with Distributed Version Controlling. Which project is the "right" project when multiple versions of the project exist?
    - Solution? Remote repositories as a single source of truth!
  - Git != GitHub (Git is not Github)!

# GIT CONCEPTS




Source: https://medium.com/@sahoosunilkumar/how-does-git-works-5cc8444ea383

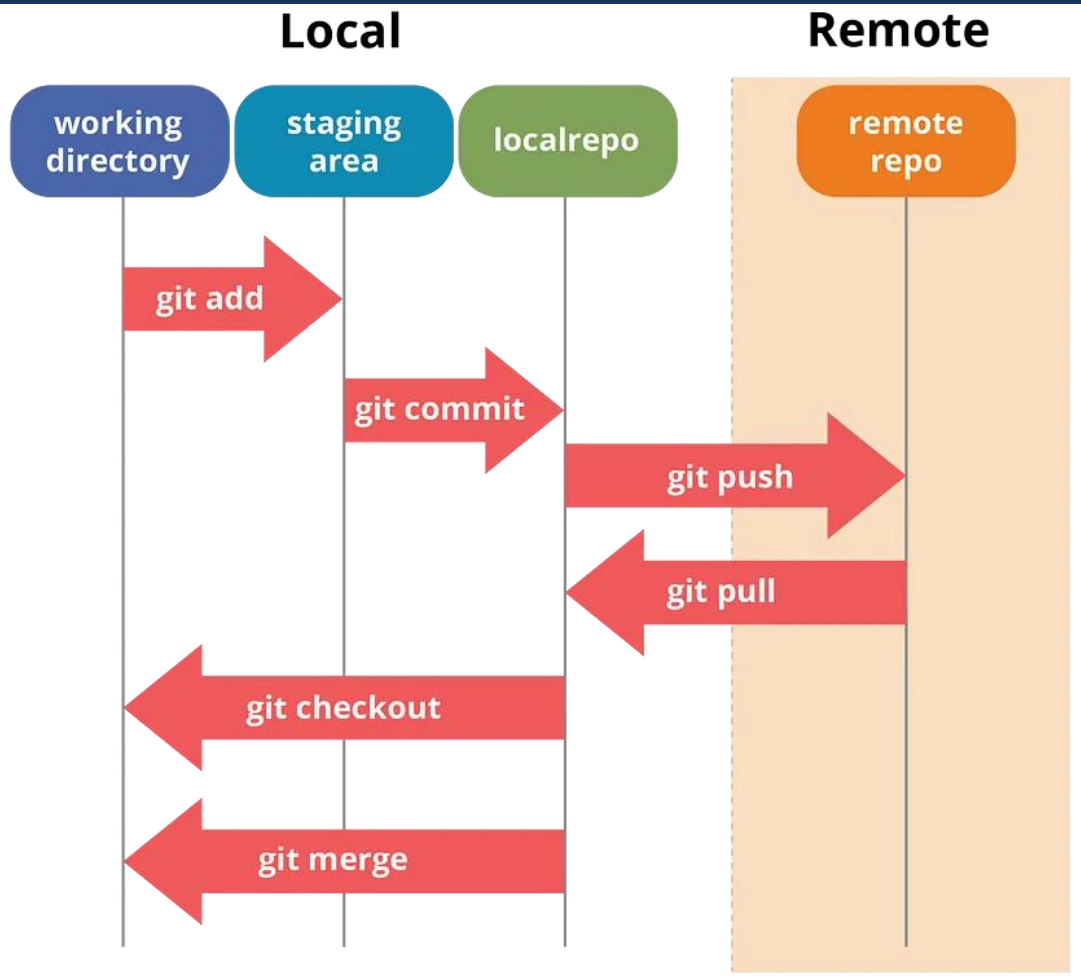
# GIT CONCEPTS

- Git works on **Commits**.
- A Commit is very simply **a snapshot of all changes in the local repository** (more correctly, the working directory).
- Each Commit has a unique Commit ID.
- A Commit ID is a unique SHA-1 hash that is created whenever a new commit is recorded.
- The most recent commit is called the **Head** (more correctly, it refers to the currently checked-out branch's latest commit).
- For now, we will only deal with **main**/ master **Branch**.
- We will extensively deal with Commit IDs for many operations on git (more on this later).

Source: https://medium.com/@sahoosunilkumar/how-does-git-works-5cc8444ea383

# GIT CONCEPTS

1. **Add** a new file to the local Repo.
2. Do the required changes to the file.
3. **Commit** the file to the local repo.
4. **Pull** the changes from the remote repo.
5. **Push** the local changes to the remote repo.




Source: https://medium.com/@sahoosunilkumar/how-does-git-works-5cc8444ea383

# SCENARIO REVISITED

- Refer back to Amith's version controlling method in the beginning. It could,
  - **Revert** to a past version.
  - **Compare & Merge** two different versions of files.
  - **Pull** changes from a remote location to anywhere.
  - **Push** changes done from anywhere to a secure remote location.
  - Use the **Log file** to keep track of all the events.

- All these actions can be easily accomplished using Git.
  - Revert – git revert (this reverts a commit that was made to the repository).
  - Compare & Merge – Automatically handled by git. No user intervention required most of the time (unless a conflict occurs).
  - Push and Pull – Update the **Upstream** and get updates from upstream.
  - Git has many ways of automatically keeping track of all the events (e.g.: git status, log, blame etc.)

# NEXT WEEK

- Branching
- Best Practices
- Pull Requests
- Git Workflows

# REFERENCES


1. Pro Git - Scott Chacon and Ben Straub [Free eBook]
2. Dangit, Git!?! [Git quick reference]
3. Reading 5: Version Control [MIT Lecture]
4. Distributed Version Control [Wikipedia]
5. History of Version Control Systems

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# VERSION CONTROLLING & WORKFLOWS WITH GIT - II

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

# LEARNING OUTCOMES

- After completing this lecture, you will be able to,
  - Apply the basic concepts of Git for version controlling
  - Apply an efficient branching strategy for a team
  - Describe what a git workflow is and why a workflow is important.
  - Apply a simple git workflow to your own projects.

# CONTENTS

# FROM THE PREVIOUS WEEK

- Version Controlling: Scenario Discussion
- A bit of history on Version Controlling Systems (VCS)
- Git Basic Concepts
  - Repositories – Local and Remote
  - Each user has their own local repository
    - This local repository can be obtained by,
      - Copying from another user (P2P)
      - **Cloning** it from a remote repository (on GitHub, BitBucket etc.)
      - Initializing their own repository.
  - Commits
  - Basic git operations (git add, commit, push, pull)

# GIT BRANCHING: THE BASICS

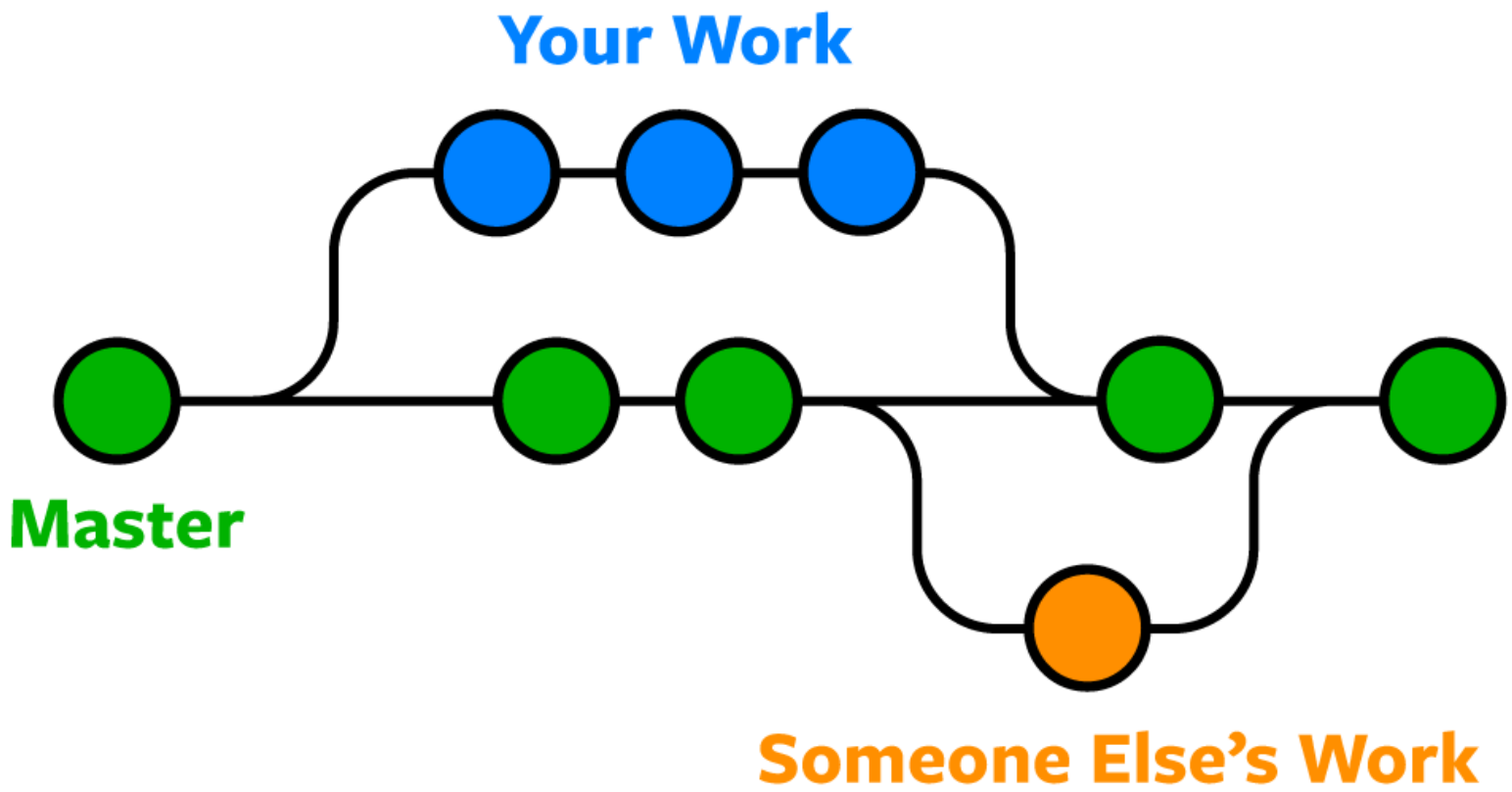
- Think of a tree. What is a branch in a tree?
  - An offshoot of the trunk of a tree.
- What is a branch in Git?
  - The default branch in Git is the ***master*** branch (It is like the trunk of a tree).
  - Very simply put, a branch is a detour you make from the main path of development.

Source: https://www.rainforest-alliance.org/wp-content/uploads/2021/06/kapok-tree-profile-1.jpg.optimal.jpg

# GIT BRANCHING: THE BASICS

- Why is branching needed?
  - Branching lets developers work independently inside a single code repository.
    - E.g., In product X, Amith works on feature A, while Binesh works on feature B.
    - If both have their code in just their local repositories, then no problem.
    - Now, it is a good practice to push your changes to the remote repository regularly.
      - What sort of problems can be expected when they do that?
        - **Merge Conflicts!**
  - Any solution to the above problem?
    - Yes, let each of them work on their own branch.

# GIT BRANCHING: THE BASICS




Example on git branching. Source: https://www.nobledesktop.com/learn/git/git-branches

# GIT BRANCHING: THE BASICS

- Why is branching needed? (continued)
  - Allows a developer to switch among different tasks easily.
    - Suppose while Amith is working on feature A, he gets a request to attend to a major issue in the product X.
      - If Amith is working on his local master branch, he will now have to,
        - Undo all the changes he did.
        - Fix the issue.
        - Restart the work on feature A.
      - If Amith had been working on a separate branch even if he was on his local repository, he could have,
        - Stopped the work on the feature and switch back to master.
        - Created new branch for the **"hotfix"** and work on it.
        - Pushed the hotfix branch to the remote repo (and merge to remote master)
        - Switched back to his feature branch and continue where he left off.

# GIT BRANCHING: THE BASICS

- Referring to the previous scenario,
  - Using git branches during development time is clearly more efficient rather than not using any.
  - Using branches is a must for collaborations with multiple developers.
  - Whatever you do on a branch is isolated from the rest of the code – great when you want to experiment!
  - It is not a bad idea to use branching for individual projects too (see the second bullet point on why branching is needed).
- Branching on Git is a very lightweight operation. Therefore, use it whenever possible.

# HOW DO WE CREATE A BRANCH ON GIT?

- Very simple.
  - *git branch <branch_name>* - just creates the branch based on your current branch.
  - *git checkout <branch_name>* - checks out the branch.

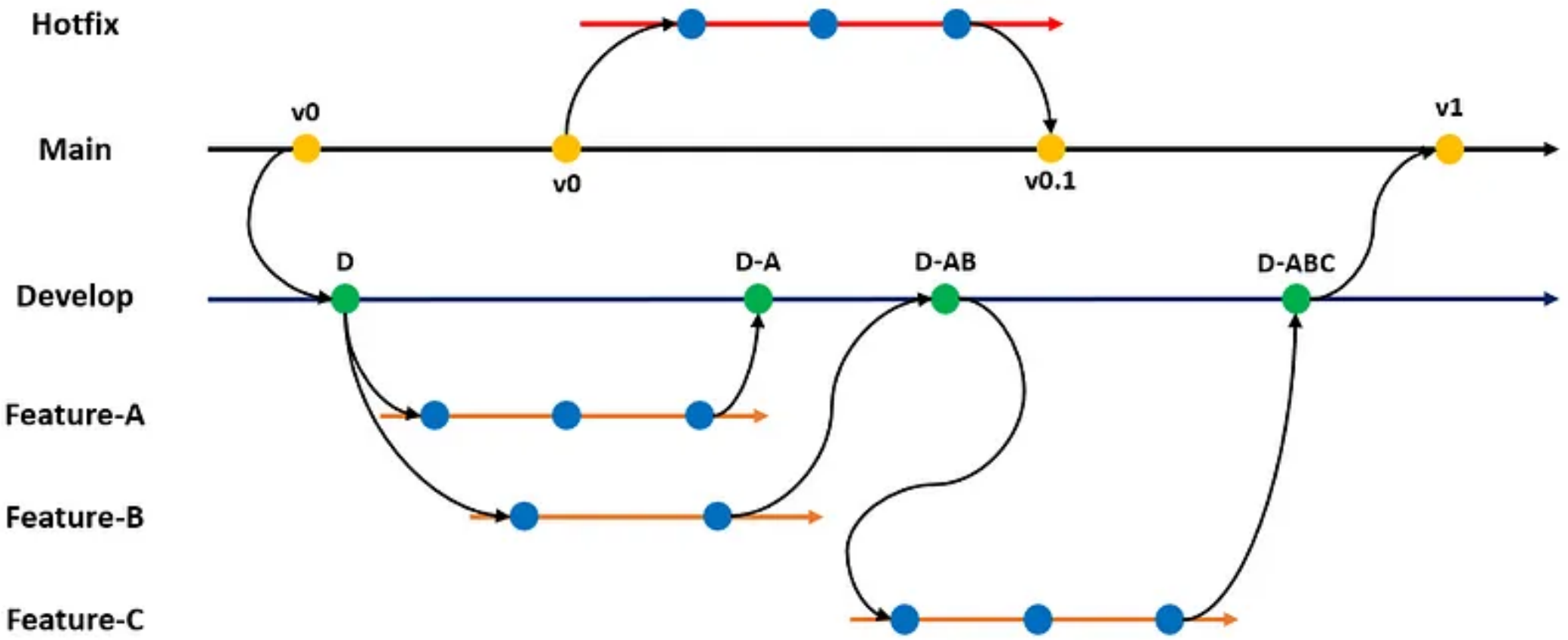
OR

  - *git checkout –b <branch_name>* - creates the branch, and switches to the new branch (checks it out) from your current branch immediately.

# WORKFLOWS (BRANCHING STRATEGIES)

- Git is very flexible when it comes to how-to branch. There is no one standard method, but some well accepted how-to branch 'recipes' are there.
- They are called workflows or branching strategies.
  - A Git workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner.
- Some popular workflows are,
  - GitFlow
  - GitHub Flow
  - GitLab Flow
  - Trunk Based Development (goes with CI/ CD)

# GIT BRANCHING: AN EXAMPLE (GITFLOW)




Source: https://betterprogramming.pub/git-branching-strategy-for-better-team-collaboration-aacb5f235d05

# A SIMPLER GIT WORKFLOW

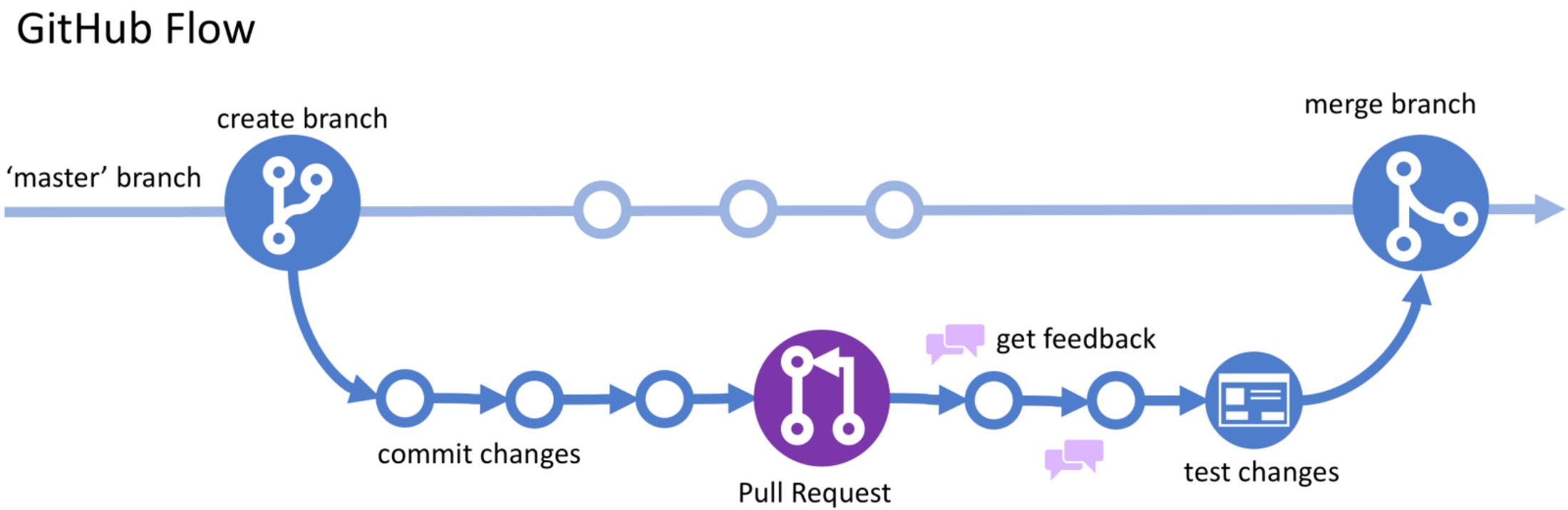
- The GitFlow is very comprehensive.
  - Can become unnecessarily complex for small projects.
- It is more suited to Enterprise grade software development efforts.
  - For most use cases, it is overkill.
- Therefore, a simpler alternative is preferable for most use cases.
  - One such workflow is the GitHub Flow which can be used by anybody.

# THE GITHUB FLOW

- This is a GitHub based simple workflow.
  1. Create a new feature branch from the main (master) branch.
  2. Make the necessary changes to the feature branch – continue until you think your feature is ready.
  3. Once you think the feature is done, ask for feedback from the other developers – create a Pull Request (PR) for this.
  4. Address their review comments received on the Pull Request (PR).
  5. Merge your Pull Request (PR) to the main branch.
  6. Delete your feature branch.

Refer the official documentation here.

# THE GITHUB FLOW




Source: https://build5nines.com/introduction-to-git-version-control-workflow/

# PULL REQUESTS (PR)

- A Pull Request (PR) is used to propose a new change to a repository (usually to the main branch).
- Usually, the proposed changes are in a feature branch, which ensures that the main branch will only contain code which is tested and working.
- The other collaborators will have to verify that the proposed change (PR) is OK first.
- If the PR is OK, they will accept it. Or else, they will suggest changes until it is acceptable.
- Once the PR is accepted, the changes in the feature branch are merged automatically to the main branch, bringing the proposed change to the main branch.
- Find the official documentation on Creating PRs here.

# GIT BRANCHING: BEST PRACTICES

1. Create a branch for each feature. Do not work on multiple features in a single branch.
2. Ideally, only one person should work on one feature branch.
3. Feature branches should be short-lived. They **should not exist** once the feature is complete.
4. Delete the local and remote feature branch after merging to the master branch (see point 3).

# GIT BRANCHING: BEST PRACTICES

5. Merge early and often to avoid merge conflicts.
6. Keep the master branch clean. Only *production ready* code should be in the master branch. It is better to **avoid** working directly on the master branch.
7. Use a good naming convention to name your feature branches. Everyone should stick to the same naming convention.
   - E.g.:
     - feature/<username>/task-name-in-lowercase-spaces-replaced-with-dashes
       - feature/vishan.j/awesome-feature-x-to-our-app

# GIT GENERAL BEST PRACTICES

1. Make small, incremental changes.
   - Don't make huge changes in one go. If you have a big feature to add, break it down to smaller features and add one at a time.
2. Keep commits atomic.
   - It is better if each commit is focused on one part of the feature. This will minimize the damage if you must revert a commit.
3. Commit often.
   - Commit your changes often. Even if the work is not done.
4. Pull the changes in origin/main branch before you create a new branch from your local/main.
   - This ensures that the new feature branch is created with all the latest changes in master. This reduces the possibility of merge conflicts later.

# GIT GENERAL BEST PRACTICES

5. Push local feature branch commits to Remote often.
   - Even if it is an unfinished feature, push it. That's good for safekeeping.
6. Use branches.
   - See the section on branching for a comprehensive set of reasons.
7. Write good commit messages.
   - Make the lives of other Software Engineers, DevOps engineers, QA engineers easier.
8. Select one branching strategy (Git Workflow) and stick with it.
   - Select one that suits you and use it well.

# SUMMARY

- Git branching
  - The basics
  - How do we create a branch on git?
  - Workflows (Branching strategies)
  - An Example (GitFlow)
  - A Simpler Git Workflow
    - The GitHub Flow
    - Pull Requests (PRs)
  - Best practices for branching
- Git general best practices

# ADDITIONAL TOPICS

Read on these topics to be up to date on some current trends.

- Continuous Integration and Continuous Deployment (CI/ CD).
- Compare Trunk based development workflow with GitFlow workflow.
  - Understand when to use which.
- Read on Pull Requests.
- Read on Git Rebase and see why it's a better alternative to Git Merge.

# REFERENCES


1. Pro Git - Scott Chacon and Ben Straub [Free eBook]
2. Dangit, Git!?! [Git quick reference]
3. Branch in Git
4. What Are the Best Git Branching Strategies
5. 10 Git Branching Strategy Best Practices
6. Trunk-based Development vs. Git Flow
7. How to Use Git and Git Workflows – a Practical Guide

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# WEB APPLICATION ARCHITECTURE – AN OVERVIEW

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

# LEARNING OUTCOMES

- After completing this lecture, you will be able to,
  - Describe the main components of three tier architecture for web development.
  - Identify some suitable technologies to develop the presentation layer and the application layer of a web application.
  - Apply suitable architectural components to design the presentation layer and the application layer of a Three tier web application.

# CONTENTS

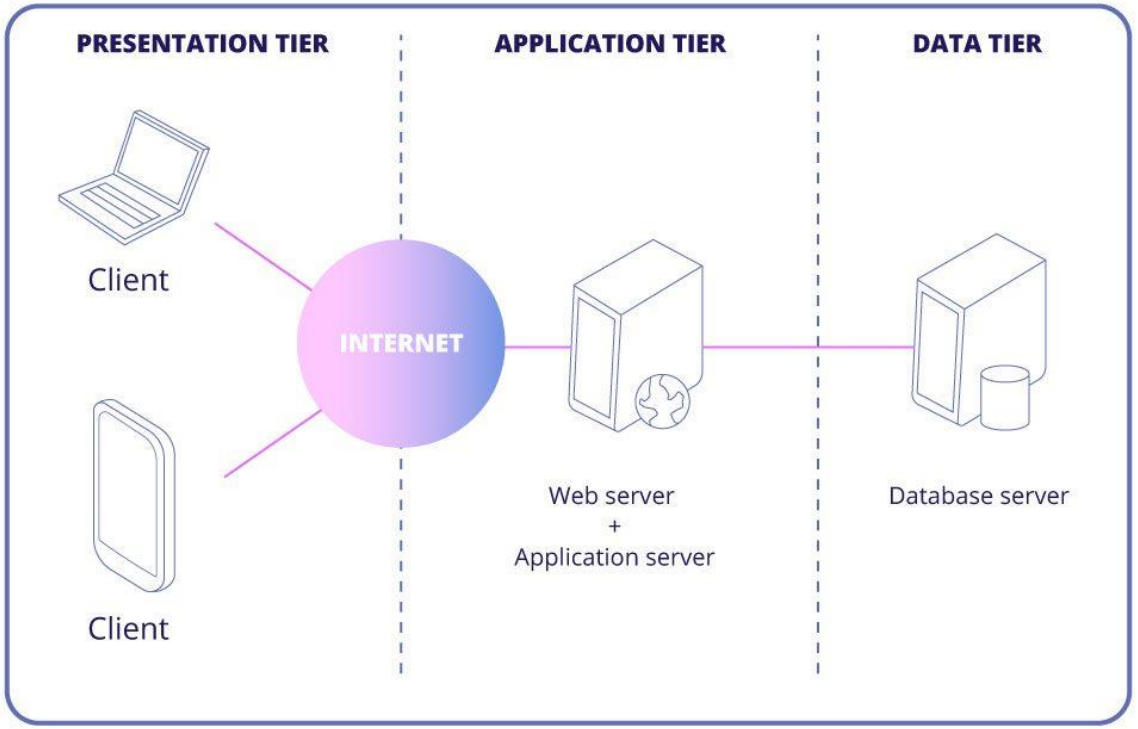
- Website vs Web Application
- Three Tier/ Three Layer Architecture
- Presentation Layer
  - Some technologies
  - Architecting the Presentation Layer
- Application Layer
  - Some technologies
  - Application Programming Interface (API)
  - Architecting the Application Layer
  - Big Ball of Mud: an architectural anti-pattern
- Other Helpful Technologies
- Summary

# WEBSITE VS. WEB APPLICATION

- Q: What is the difference between a Website and a Web Application?
  - A Website provides static content which can be consumed by the end users.
  - A **Web Application** is designed for **interaction** with the end users.

- Our focus is on **Web Applications**.

# THREE TIER ARCHITECTURE




Source: https://mobidev.biz/blog/web-application-architecture-types

mobidev

# THREE TIER ARCHITECTURE

- Modern web applications are engineered based on this architecture.

- Has three tiers.
  - Presentation tier/ layer
  - Application tier/ layer
  - Database tier/ layer

- The most popular form of n-tier (multitier) architecture.

# THREE TIER ARCHITECTURE

- Three tier architecture is a good choice for the development of web applications.

- Reasons?
  - Each tier/ layer runs on its own infrastructure.
  - Each tier/ layer can be developed independently/ in parallel by different teams.
  - Allows updating and scaling up/ down each tier independently without impacting the other tiers.
  - Better security when compared with two tier architecture (due to the isolation of data and logic from the presentation tier).

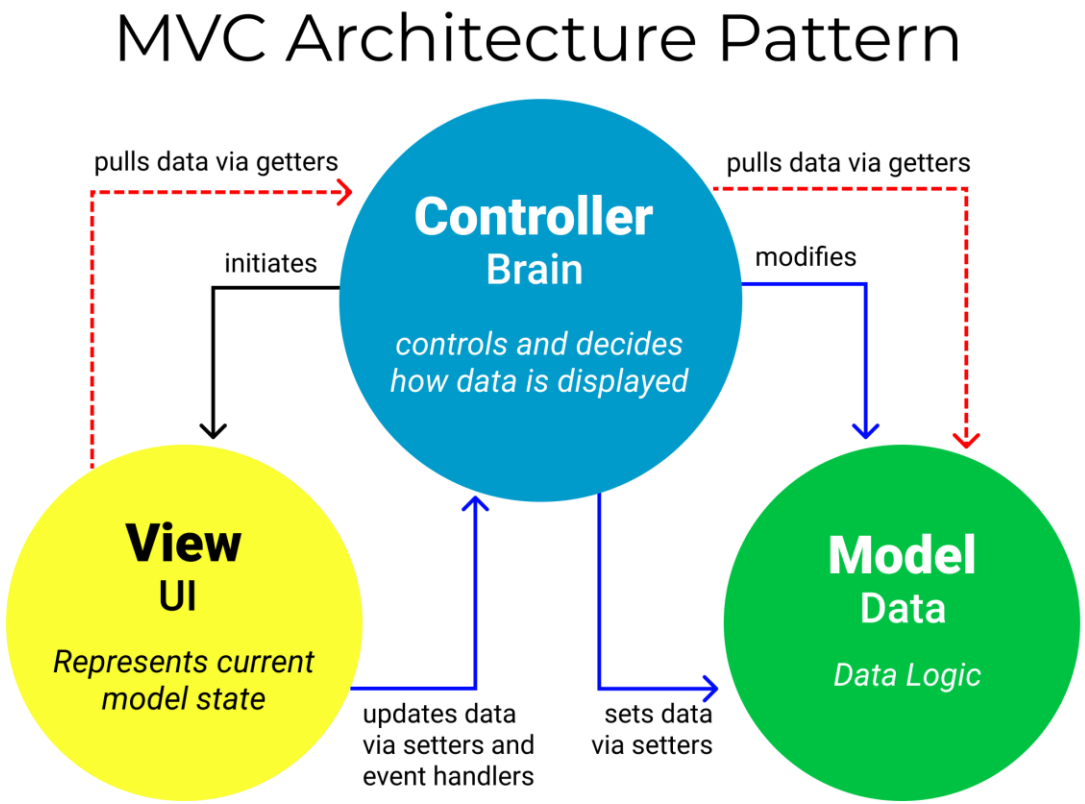
# PRESENTATION LAYER

- Also called the client side or the frontend.
- Technologies used to implement include,
  - HTML/ CSS/ JavaScript
  - React.js
  - Angular
  - Vue.js
  - Some legacy technologies
    - jQuery and AJAX

# ARCHITECTING THE PRESENTATION LAYER

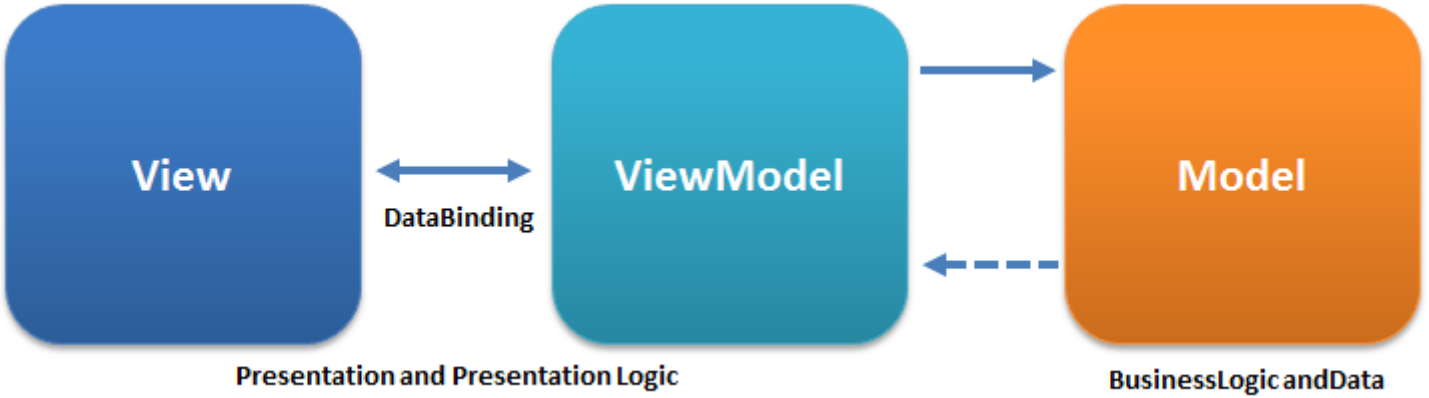
- Some well known examples
  - Model-view-controller (MVC)
  - Model-view-viewmodel (MVVM)
- Some more examples
  - Model-view-presenter (MVP)
  - Component Architecture
  - Micro frontends

# ARCHITECTING THE PRESENTATION LAYER: MVC




Source: https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/

# ARCHITECTING THE PRESENTATION LAYER: MVVM



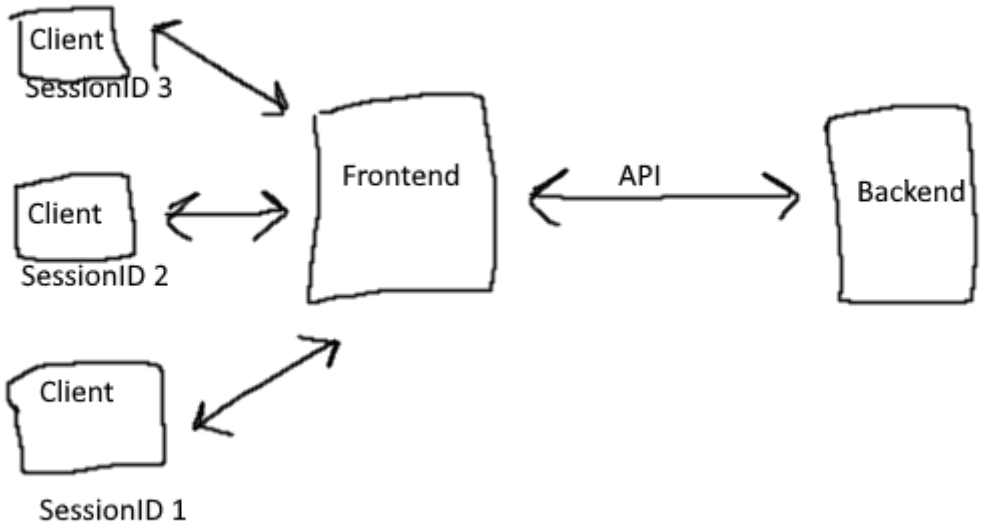

Source: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel

# APPLICATION LAYER

- Also called the server side or the backend.
- Technologies used to implement include but not limited to,
  - Programming languages and frameworks
    - Java (frameworks: Spring/ Spring Boot)
    - Node.js (frameworks: express.js/ Koa)
    - Python (frameworks: Django/ Flask)
    - PHP (frameworks: Laravel, Symphony)
    - .Net Framework
  - Virtual Machines (VMs)
  - Serverless Computing
  - Containers

# APPLICATION LAYER

- How does the Backend and the frontend communicate?




Source: https://stackoverflow.com/q/71033687

# APPLICATION LAYER: APIS

- Application Programming Interface (API)!

- What is it?
  - A software interface which facilitates communication between two or more applications.
  - Abstracts away the complexities of application integrations.
  - A contract of services offered.

# APPLICATION LAYER: APIS

- Benefits of APIs?
  - developers don't have to create everything from scratch - can use existing functions exposed as an API resulting in more productivity.
  - APIs significantly reduce development costs by reducing development efforts.
  - Improves collaboration and connectivity across the ecosystem.

# APPLICATION LAYER: APIS

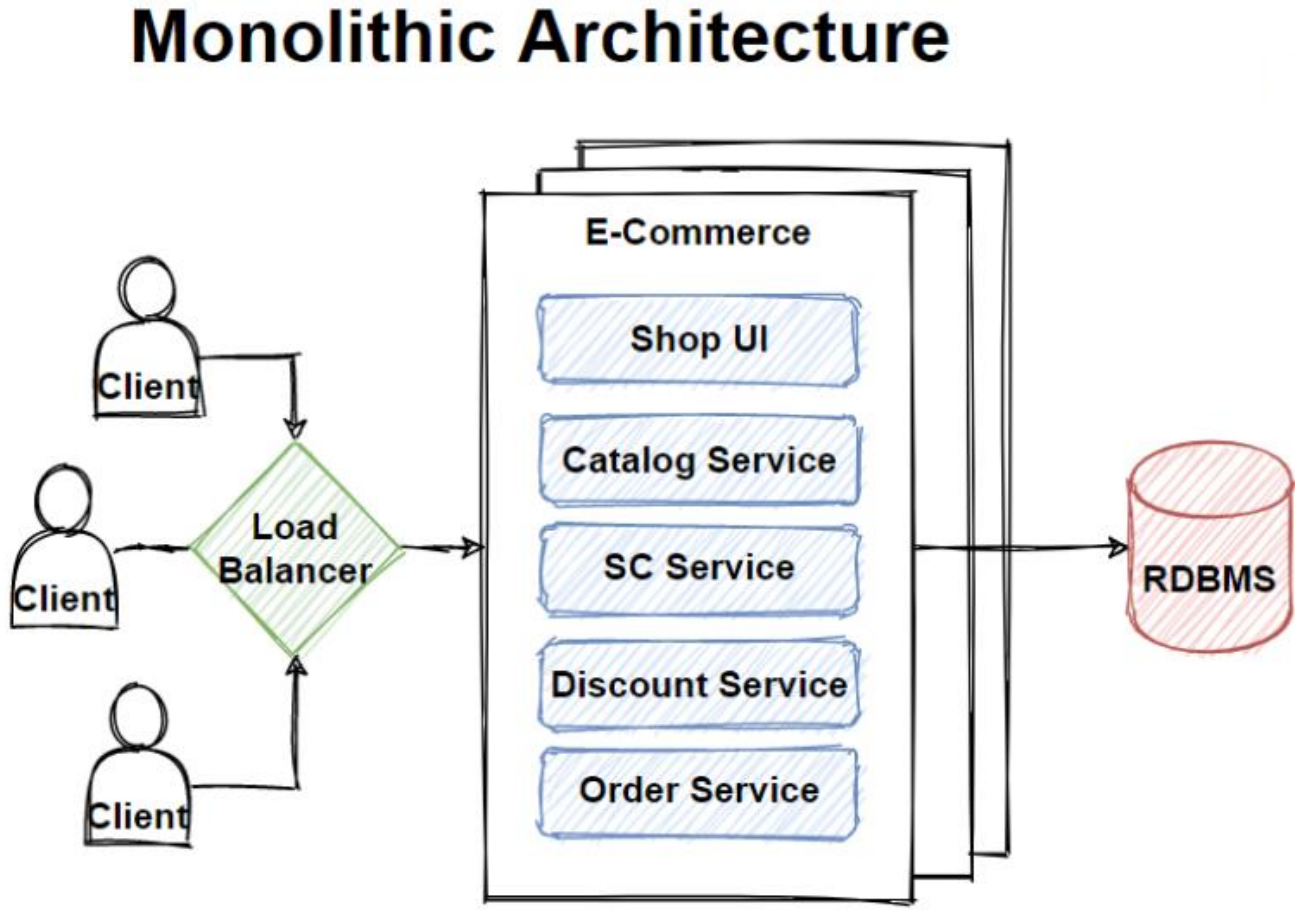
- Different types of APIs?
  - Representational State Transfer API (REST API)
  - Simple Object Access Protocol API (SOAP API)

More on these in a separate lecture.

# ARCHITECTING THE APPLICATION LAYER

- Some well known examples
  - Monolithic Architecture
  - Microservices Architecture

# ARCHITECTING THE APPLICATION LAYER: MONOLITHIC




Source: https://medium.com/design-microservices-architecture-with-patterns/monolithic-architecture-is-still-worth-at-2021-98bfc112dc24

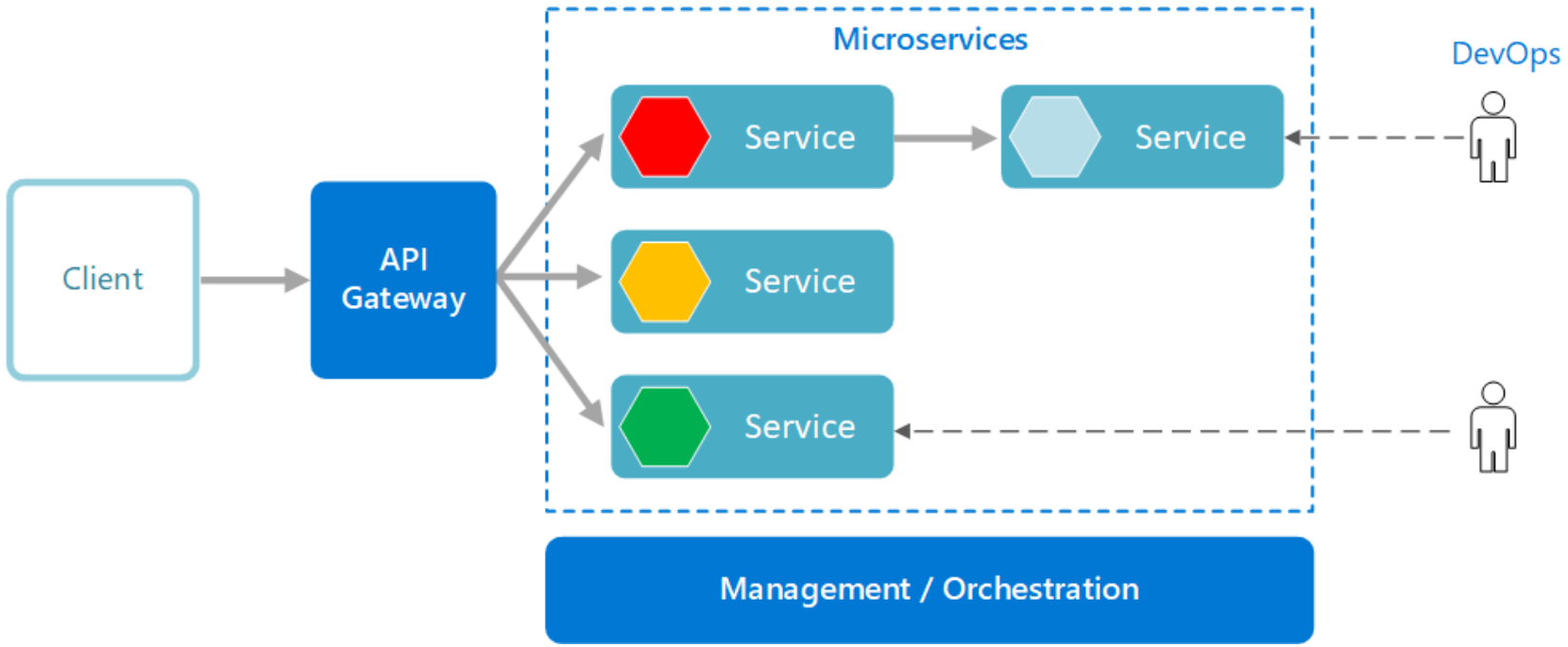
# ARCHITECTING THE APPLICATION LAYER: MONOLITHIC

- Drawbacks in Monolithic Architecture?
  - Slower development speed due to **local complexity.**
  - Can't scale individual components.
  - Reliability – if there's an error in any module, it could affect the entire application's availability.
  - Barrier to technology adoption – any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.
  - Lack of flexibility – a monolith is constrained by the technologies already used in the monolith.
  - A small change to a monolithic application requires the redeployment of the entire monolith.

# ARCHITECTING THE APPLICATION LAYER: MICROSERVICES

- "Microservices architecture is an approach in which a single application is composed of many loosely coupled and independently deployable smaller services." (Source – IBM)

# ARCHITECTING THE APPLICATION LAYER: MICROSERVICES




Source: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

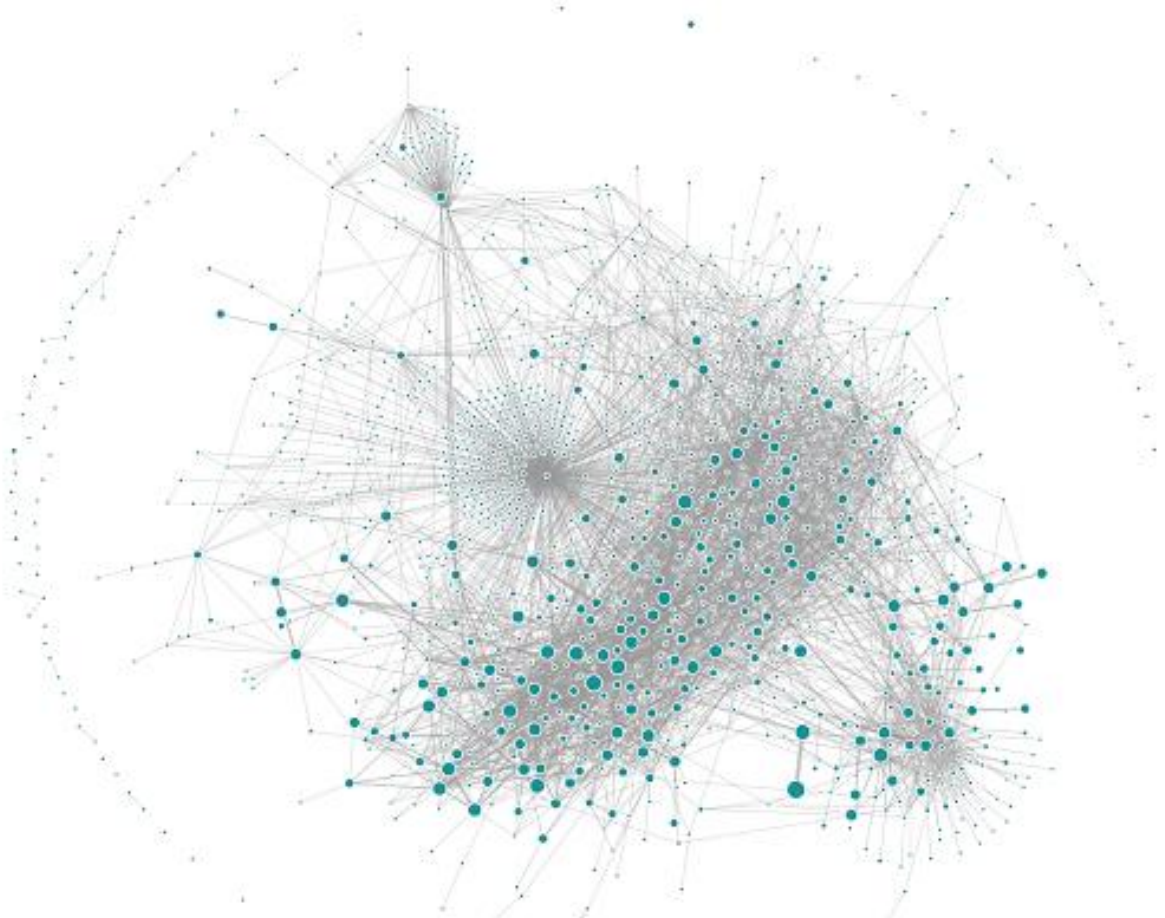
# ARCHITECTING THE APPLICATION LAYER: MICROSERVICES

- Some benefits of Microservices
  - Microservices are **small, independent, and loosely coupled** (therefore, a single team can write/ maintain a single service).
  - Services can be **deployed independently** (can update an existing service without rebuilding/ redeploying the entire application).
  - Can **scale** each service independently
  - High **reliability**
  - Supports **polyglot programming**

# ARCHITECTING THE APPLICATION LAYER: MICROSERVICES

- Is everything great with Microservices?

# ARCHITECTING THE APPLICATION LAYER: MICROSERVICES



Uber's microservice architecture circa mid-2018

Source: https://www.uber.com/en-LK/blog/microservice-architecture/

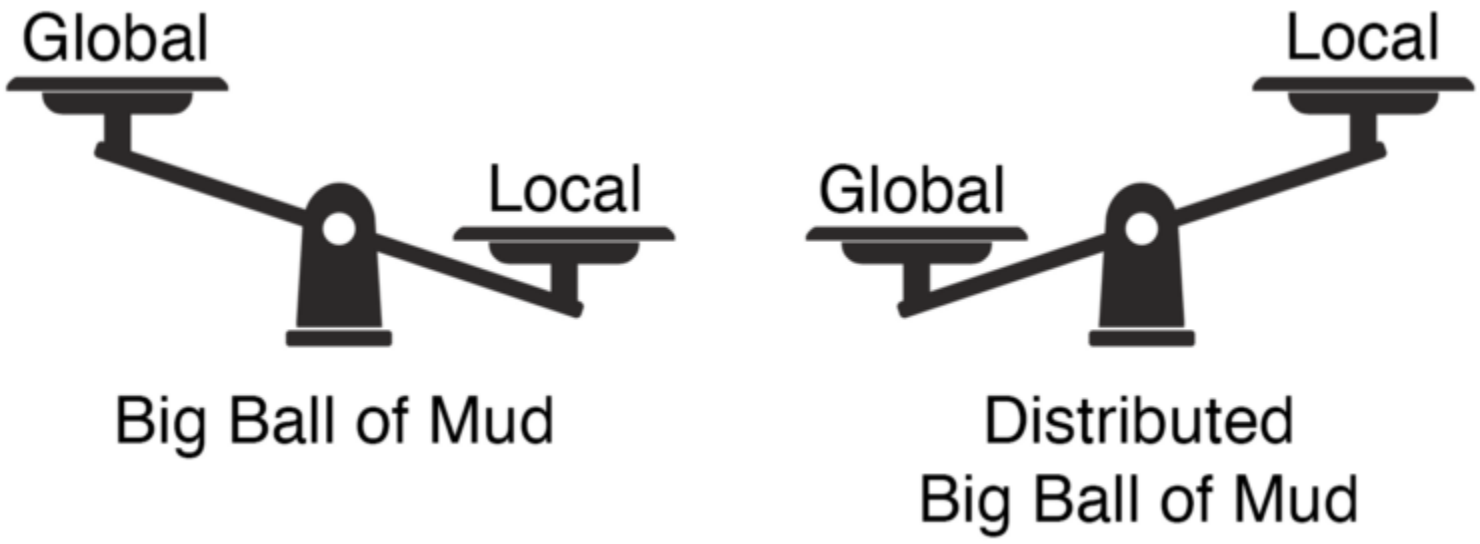
# ARCHITECTING THE APPLICATION LAYER: MICROSERVICES

- **High global complexity** is the biggest problem when in comes to Microservices.
  - *Local complexity* depends on the **implementation of a service**; *global complexity* is defined by the **interactions and dependencies** between the services.
- Managing complexity is a constant challenge in Microservices.

# ARCHITECTING THE APPLICATION LAYER: MICROSERVICES

- What are some other downsides of Microservices?
  - High infrastructure costs
  - Debugging challenges
  - A lot more…

# BIG BALL OF MUD: AN ARCHITECTURAL ANTI-PATTERN




Source: https://www.doit.com/untangling-microservices-or-balancing-complexity-in-distributed-systems/

# OTHER HELPFUL TECHNOLOGIES

- API Gateways (authentication, throttling, orchestration, caching, statistics etc.)
- Content Delivery Networks (CDN)
- Caching Tools (e.g., Redis)
- Load Balancers
- Message Queues
- Cloud Computing
  - Cloud Storage
  - Virtual Machines (VMs)
  - Many more resources…

# SOME INTERESTING TOPICS

- Component Based Architecture
- Micro Frontends – Microservices for the frontend
- BFF: Backend For Frontend also known as BIF: Backend In Frontend

# REFERENCES


1. Web application vs. website: finally answered
2. What is three-tier architecture?
3. SOAP APIs vs REST APIs
4. Micro Frontends: What are They and When to Use Them?
5. Model–view–presenter
6. An In-Depth Guide to Component-Based Architecture: Features, Benefits, and more
7. The Complete Guide To BFF (Backend For Frontend)
8. Microservices vs. monolithic architecture
9. Untangling Microservices or Balancing Complexity in Distributed Systems
10. Big Ball of Mud - The Daily Software Anti-Pattern

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# REST APIS

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

# LEARNING OUTCOMES

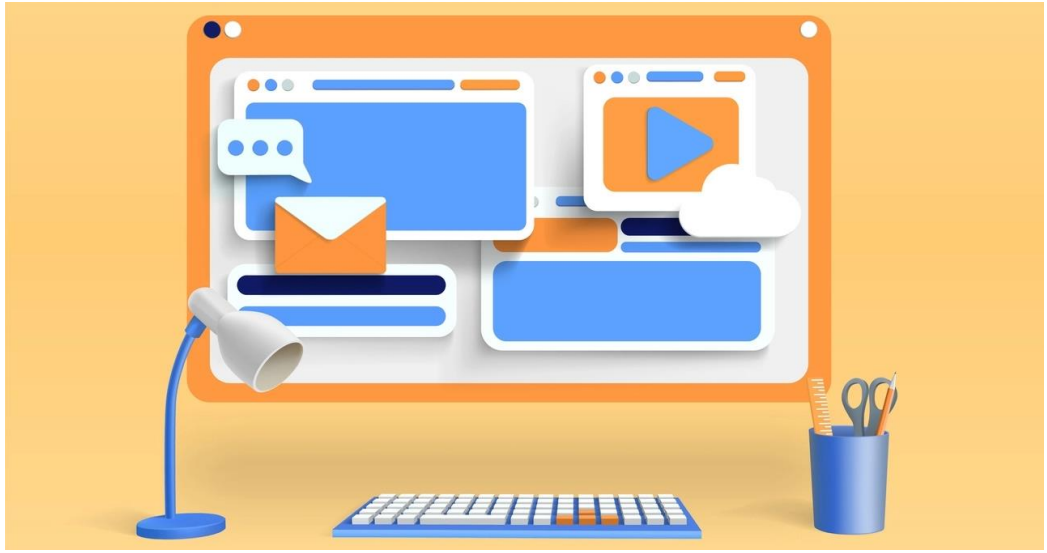
- After completing this lecture, you will be able to,
  - Describe the six REST architectural constraints.
  - Apply the knowledge gained in this lecture in designing proper REST APIs.
  - Evaluate if an API is a REST API or not.
  - Evaluate a given API to identify how much it conforms to the REST constraints using the Richardson Maturity Model.

# CONTENTS

- Consuming web resources - Human vs. programs
- Application Programming Interface (API)
  - Examples
- REST
  - Six Architectural Constraints of REST
  - True REST vs. RPC (Remote Procedure Calls)
  - Richardson Maturity Model
- Some Facilitators of REST
  - HTTP, JSON, HAL
- Best Practices

# CONSUMING WEB RESOURCES - HUMAN VS. PROGRAMS

- How do you consume web resources?
  - Web browsers
  - Web applications
  - Mobile applications etc.



Source: https://illuminatingfacts.com/a-timeline-of-the-rise-and-fall-of-popular-web-browsers/

# CONSUMING WEB RESOURCES - HUMAN VS. PROGRAMS

- How do programs consume web resources?
  - Web APIs!
  - API stands for **Application Programming Interface**


Source: https://stileex.xyz/en/difference-computer-program-software/

# APPLICATION PROGRAMMING INTERFACE (API)

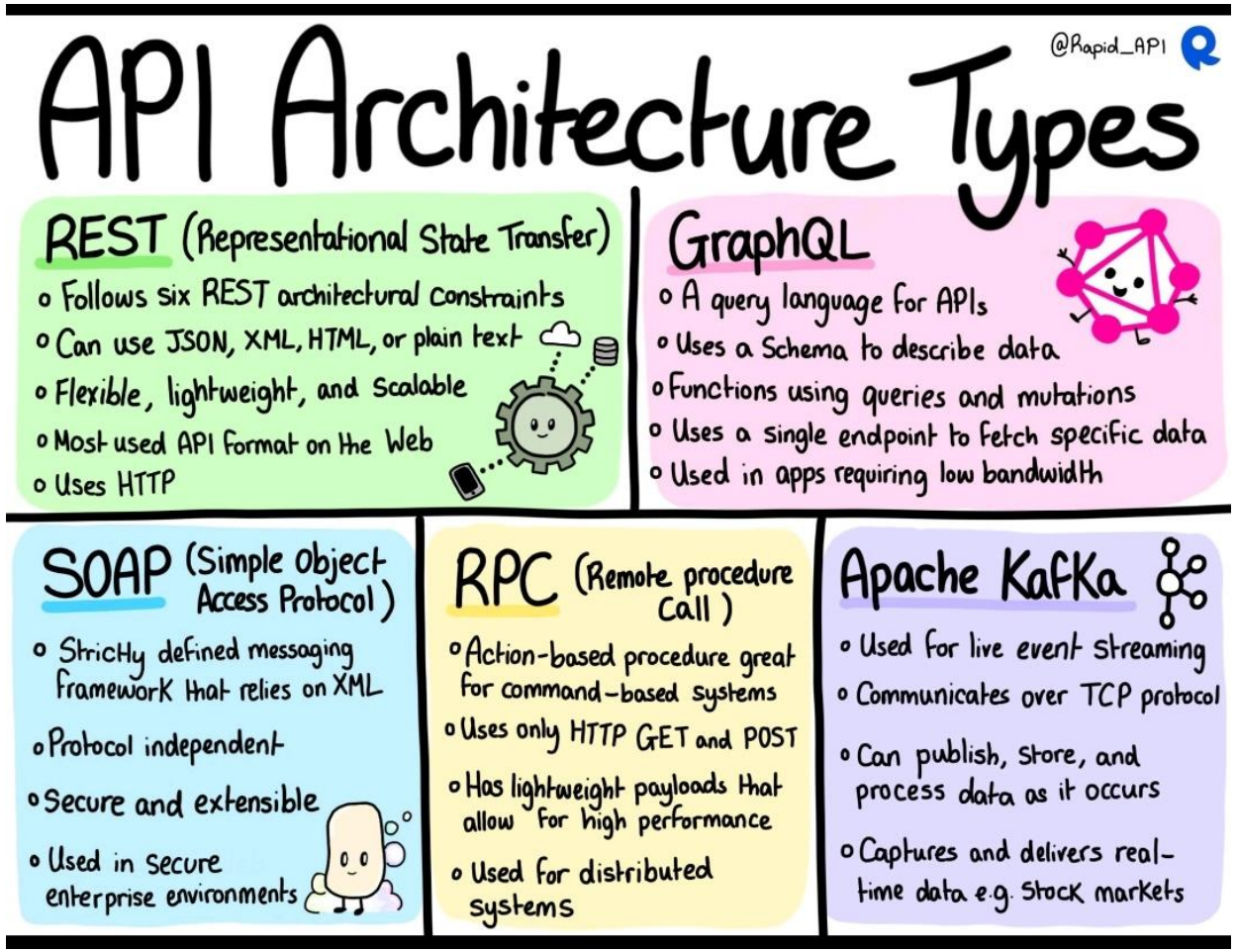
- What is an API?
  - A software interface which facilitates communication between two or more applications.
  - Abstracts away the complexities of application integrations.
  - A contract of services offered.

# APPLICATION PROGRAMMING INTERFACE (API)

- Examples:
  - RPC (Remote Procedure Calls)
  - CORBA Implementations (Common Object Request Broker Architecture)
  - SOAP (Simple Object Access Protocol)
  - **REST** (Representational State Transfer)

Mostly legacy technologies

# APPLICATION PROGRAMMING INTERFACE (API)




Source: https://ee.linkedin.com/posts/theo-borek-higgins_devops-api-devopslearn-activity-7028421145682944000-0YgL

# REST (REPRESENTATIONAL STATE TRANSFER)

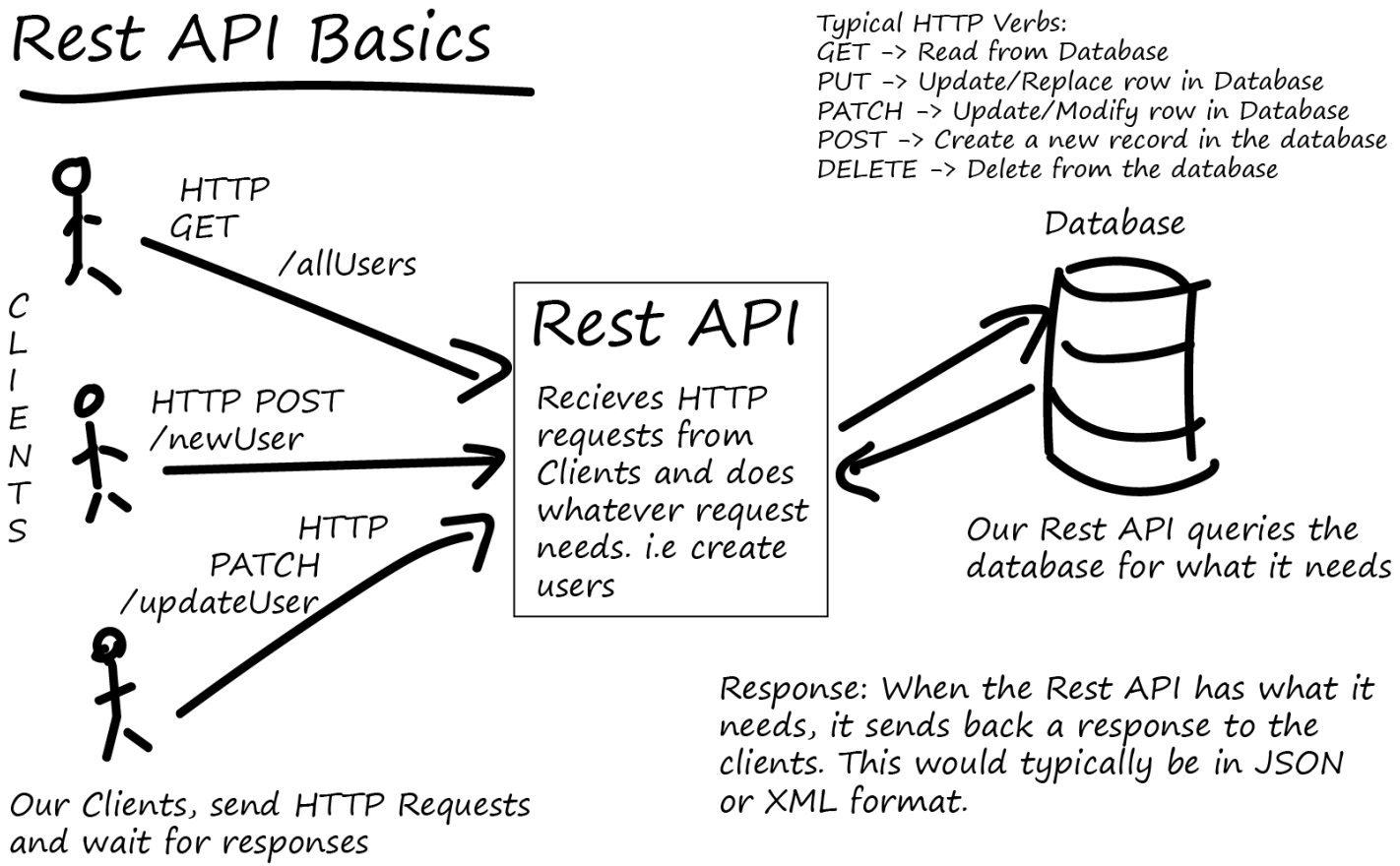
- What is REST?
  - an **architectural style** for creating web services.
  - Introduced by Roy Fielding in 2000 in his now famous Doctoral dissertation.
  - Arguably the most popular approach to creating Web APIs now.


Source: https://medium.com/@liams_o/15-fundamental-tips-on-rest-api-design-9a05bcd42920

# REST (REPRESENTATIONAL STATE TRANSFER)

- Why is REST Popular?
  - Just an **architectural style** (not a technology or a full-blown protocol like SOAP)
  - If the API aligns with the six REST constraints, it is a REST API
  - No strict rules, very flexible
  - Simple to use and learn compared with previously used approaches (e.g., SOAP/ CORBA etc.)
  - Runs on HTTP  Protocol

# REST (REPRESENTATIONAL STATE TRANSFER)




Source: https://tutorialedge.net/software-eng/what-is-a-rest-api/

# REST (REPRESENTATIONAL STATE TRANSFER)

- An example
  - A proper REST response to a GET call to http://localhost:8080/employees/1 REST endpoint.


```
{
  "id": 1,
  "firstName": "Bilbo",
  "lastName": "Baggins",
  "role": "burglar",
  "name": "Bilbo Baggins",
  "_links": {
    "self": {
      "href": "http://localhost:8080/employees/1"
    },
    "employees": {
      "href": "http://localhost:8080/employees"
    }
  }
}
```


Source: https://spring.io/guides/tutorials/rest/

# REST (REPRESENTATIONAL STATE TRANSFER)

- Design Principles
  - REST is designed to facilitate **strong decoupling** between the client and the server to facilitate internet-scale usage.
  - Follows a Request/ Response style of communication.
  - Some Terms:
    - Client – the consumer of services via the API
    - Server – the provider of services and the API that exposes these services
    - Resource – any content that the server responds with to a client request

# REST (REPRESENTATIONAL STATE TRANSFER)

- Design Principles (contd.)
  - When a **client** requests for a **resource**, the **server** will respond with
    - the **representation** of the resource in its current **state** (Usually JSON or XML)
    - and relevant **hypermedia links (URIs)** that can be used to change the **state** of the resource

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Client-server architecture
- Stateless
- Cacheable
- Uniform Interface
- Layered system
- Code on demand (Optional)

Let's examine each constraint.

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Client-server architecture
  - Maintain the clear separation of the client and the server,
    - To promote portability of the user interface
    - To promote low coupling between the client and the server
    - To improve scalability of the server components
    - To allow clients and servers grow independently

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Stateless
  - Communication between clients and server must be stateless in nature
  - Each request from client to server must contain all the information necessary to understand the request and cannot take advantage of any stored context on the server.
  - Session state is kept entirely on the client.
  - Improves visibility, reliability, and scalability.

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Cacheable
  - Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.
  - If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
  - Caching can eliminate requirement for some interactions thereby improving efficiency, scalability, and user-perceived performance.

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Uniform Interface
  - Implementations are decoupled from the services they provide, which encourages independent evolvability.
  - Continued in the next slide…

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Uniform Interface
  - Has four sub-constraints.
    - **Self-descriptive messages** - Each message going back and forth between the client and the server should contain enough information to process the message.
    - **Identification of resources** - Individual resources are identified in requests/ responses. This is done through URIs (Uniform Resource Identifiers). The resources themselves on the server are conceptually separate from the representations that are returned to the client.
    - **Resource manipulation through representations** - When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource's state
    - **Hypermedia as the engine of application state** (**HATEOAS**) – continued in the next slide…

# REST: SIX ARCHITECTURAL CONSTRAINTS

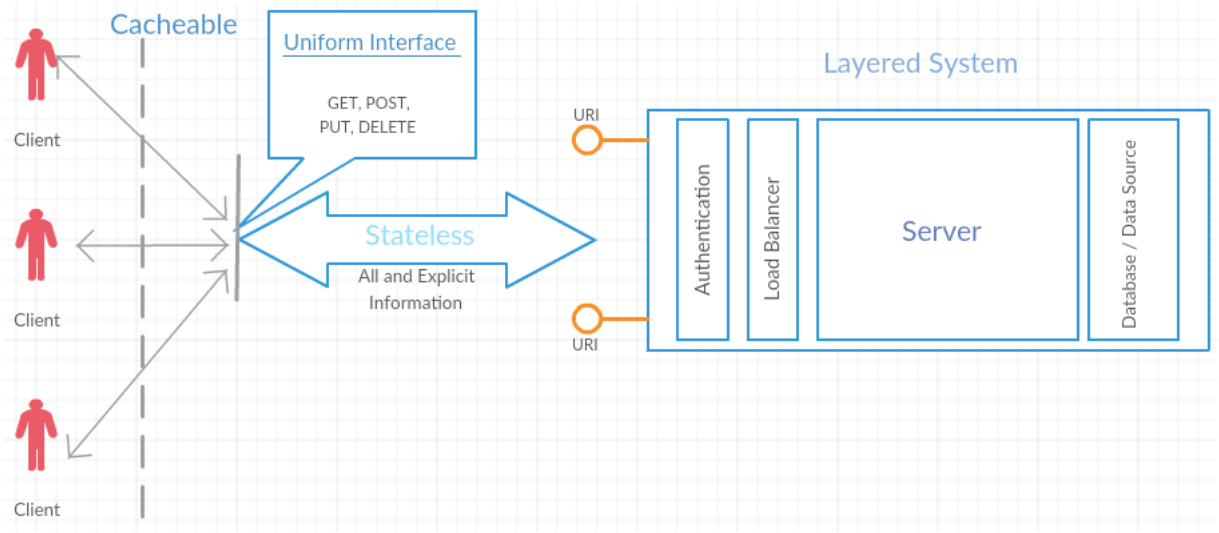
- Uniform Interface (continued…)
  - **Hypermedia as the engine of application state (HATEOAS)** - (Hey-tee-oh-s)
    - Having accessed an initial URI for the REST application a REST client should then be able to use server-provided links dynamically to discover all the available resources it needs.
    - As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available (Dynamic API).
    - There is **no need** for the client to be hard-coded with information regarding the structure of the server.
    - This is akin to a human user having to know only the main URL of a website. The rest of the URLs are discovered when consuming the site.

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Layered system
  - Each layer only interacts with the layers above and below them.
  - The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
  - The layered system constraints enable network-based intermediaries such as proxies and gateways to be transparently deployed between a client and server using the Web's uniform interface.
    - A network-based intermediary will intercept client-server communication for a specific purpose. Network-based intermediaries are commonly used for enforcement of security, response caching, and load balancing.

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Layered system (continued…)




Source: REST API Design Rulebook (Mark Masse, Oreilly, 2011)

# REST: SIX ARCHITECTURAL CONSTRAINTS

- Code on demand (Optional)
  - REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.
  - This simplifies clients by reducing the number of features required to be pre-implemented.
  - This is the only optional constraint in REST.

# TRUE REST VS. RPC (REMOTE PROCEDURE CALLS)

- Pretty URLs like /employees/10 aren't REST.
- Merely using HTTP methods such as GET, POST etc. is not REST.
- Having all the CRUD operations laid out isn't REST.

- In such an API there is no way to know **how to interact with the service** without explicit help from the developers.
  - HATEOAS constraint is missing!

# TRUE REST VS. RPC (REMOTE PROCEDURE CALLS)

The response to a GET call to http://localhost:8080/employees/1 endpoint.


```
[
  {
    "id": 1,
    "name": "Bilbo Baggins",
    "role": "burglar"
  }
]
```

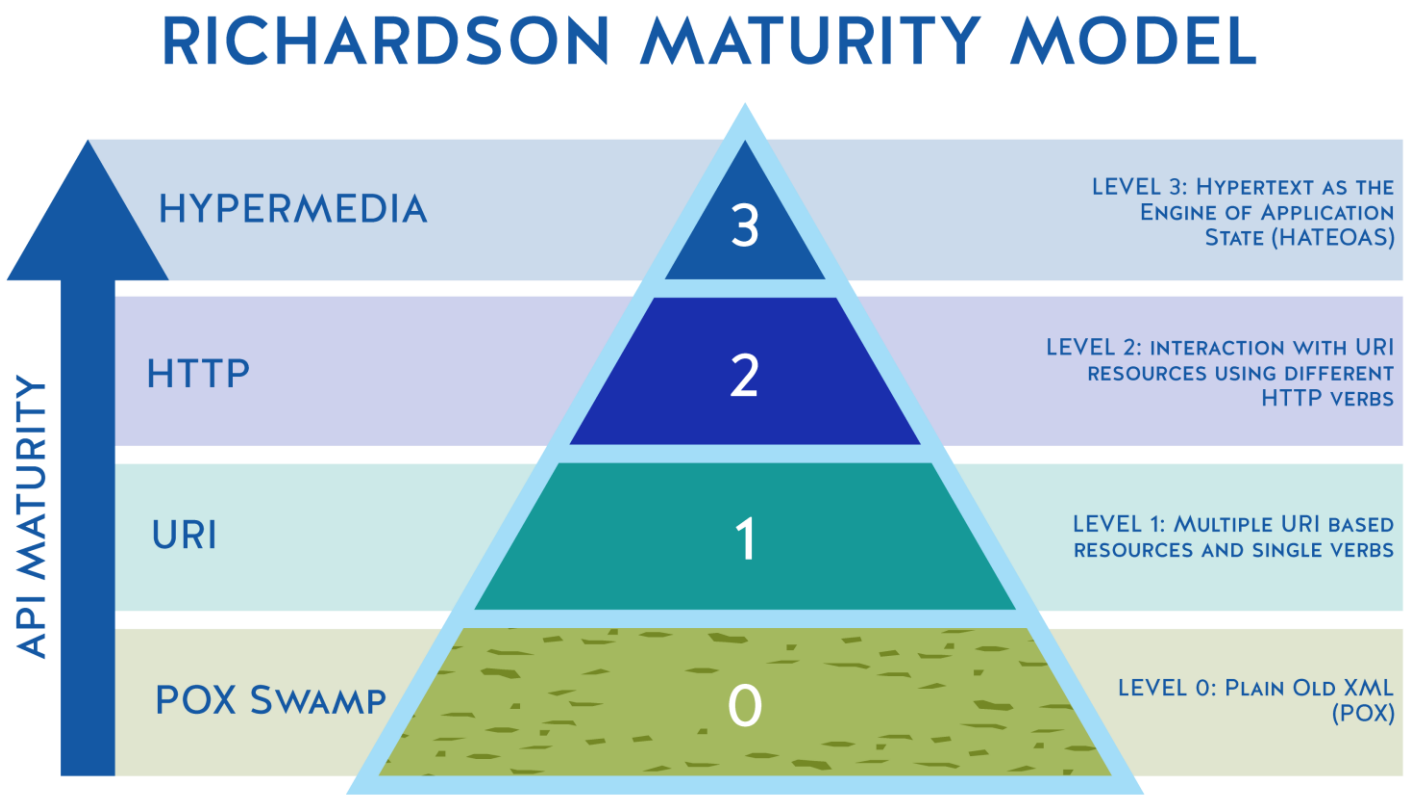

Just a JSON entity. No hyperlinks – Results in RPC


```
{
  "id": 1,
  "name": "Bilbo Baggins",
  "role": "burglar",
  "_links": {
    "self": {
      "href": "http://localhost:8080/employees/1"
    },
    "employees": {
      "href": "http://localhost:8080/employees"
    }
  }
}
```


RESTful representation of the same entity

# RICHARDSON MATURITY MODEL

- It is a four-level scale that indicates extent of API conformity to the REST constraints.
- Proposed by Leonard Richardson.




Source: https://devopedia.org/richardson-maturity-model

# RICHARDSON MATURITY MODEL

- Level-0: Swamp of POX
  - Usually exposes just one URI for the entire application. Uses HTTP POST for all actions, even for data fetch. SOAP or XML-RPC-based applications come under this level. POX stands for Plain Old XML. Could be JSON too.
- Level-1: Resource-Based Address/URI
  - Introduces resources and allows to make requests to individual URIs (still all typically POST) for separate actions instead of exposing one universal endpoint (API).
- Level-2: HTTP Verbs (Methods)
  - Fully utilize all HTTP commands or verbs such as GET, POST, PUT, and DELETE.
- Level-3: HATEOAS
  - Full maturity. Utilizes URI, HTTP Methods and HATEOAS.

# SOME FACILITATORS OF REST

- HTTP
  - HTTP Methods – REST APIs utilize these to indicate the type of operation being performed.
    - GET – Reading a resource
    - POST – Writing a resource
    - PUT – Updating a resource
    - DELETE – Deleting a resource
  - HTTP Status Codes – REST APIs utilize these to denote the status of a particular request.
    - 2XX – Successful responses range (e.g., 200 OK)
    - 4XX – Client errors range (e.g., 404 NOT FOUND)
    - 5XX – Server errors range (e.g., 500 INTERNAL SERVER ERROR)
    - List of HTTP status codes: HTTP response status codes

# SOME FACILITATORS OF REST

- HTTP (continued…)
  - HTTP Headers – These are used to pass along additional information/ meta data between the client and the server.
    - Content-type
    - Cache-control
    - Authorization
    - List of HTTP Headers: HTTP headers
- It is important that proper HTTP methods, Headers and Status Codes are used in REST requests/ responses.

# SOME FACILITATORS OF REST

- JSON (JavaScript Object Notation) – A lightweight human readable format for storing and transporting data as key-value pairs.
- HAL (Hypertext Application Language) – An open specification that provides a structure to represent RESTful resources


```
{
  "id": 1,
  "firstName": "Bilbo",
  "lastName": "Baggins",
  "role": "burglar",
  "name": "Bilbo Baggins",
  "_links": {
    "self": {
      "href": "http://localhost:8080/employees/1"
    },
    "employees": {
      "href": "http://localhost:8080/employees"
    }
  }
}
```


Source: https://spring.io/guides/tutorials/rest/

# SOME BEST PRACTICES

- Use correct HTTP methods, headers and status codes according to the operation being performed.
- Accept and respond with JSON.
- Always use proper authentication (e.g., OAuth, JWT etc.)
- Give meaningful names to the URIs.
  - E.g.,
    - https://abcd.com/users/
    - https://abcd.com/users/2
- Keep the API consistent.

# MORE BEST PRACTICES

- Self study activity
  - Read and follow this list of best practices from Microsoft: RESTful web API design
  - Make sure you apply these in your assignment as well!

- More hands-on experience on REST APIs during the practical session.

# SUMMARY

- Consuming web resources - Human vs. programs
- Application Programming Interface (API)
  - Examples
- REST
  - Six Architectural Constraints of REST
  - True REST vs. RPC (Remote Procedure Calls)
  - Richardson Maturity Model
- Some Facilitators of REST
    - HTTP, JSON, HAL
- Best Practices

# REFERENCES


1. An Introduction to Representational State Transfer (REST)
2. Architectural Styles and the Design of Network-based Software Architectures
3. Representational state transfer
4. Richardson Maturity Model
5. RESTful web API design

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# REST APIS - AUTHENTICATION AND AUTHORIZATION

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

# LEARNING OUTCOMES

- After completing this lecture, you will be able to,
  - Know the difference between authentication and authorization.
  - Apply different authentication and authorization mechanisms for REST APIs as appropriate to given scenarios.

# CONTENTS

- Authentication vs. Authorization
- Importance of Authentication and Authorization
- Some methods for Authentication and Authorization in REST APIs
  - HTTP Basic Authentication
  - API Keys
  - JSON Web Tokens (JWT)
  - OAuth 2.0
    - OpenID Connect (OIDC)
- Summary

# AUTHENTICATION VS. AUTHORIZATION

- Should everyone have free access to your resources?
  - No! Only known users should.

- How do we identify if a user is a known user?
  - **Authentication!**
  - [YouTube] Session vs Token Authentication

# AUTHENTICATION VS. AUTHORIZATION

- Should all known users be allowed to access all resources?
  - No! They should only have access to what we explicitly want them to access.
- How do we control what they can access?
  - **Authorization!**

# AUTHENTICATION VS. AUTHORIZATION

**Authorization**

What you can do

**Authentication**

Who you are

Source: https://blog.restcase.com/4-most-used-rest-api-authentication-methods/

# AUTHENTICATION VS. AUTHORIZATION

- **Authentication** is when an entity proves an **identity**.
  - Authentication proves that you are who you say you are.

- **Authorization** is when an entity proves a **right to access.**
  - Authorization proves you have the right to make a request.

- In summary:
  - Authentication: Refers to proving correct identity.
  - Authorization: Refers to allowing a certain action.

# AUTHENTICATION VS. AUTHORIZATION

| Authentication | Authorization |
| --- | --- |
| Determines whether users are who they claim to be | Determines what users can and cannot access |
| Challenges the user to validate credentials (for example, through passwords, answers to security questions, or facial recognition) | Verifies whether access is allowed through policies and rules |
| Usually done before authorization | Usually done after successful authentication |
| Generally, transmits info through an ID Token | Generally, transmits info through an Access Token |
| Generally governed by the OpenID Connect (OIDC) protocol | Generally governed by the OAuth 2.0 framework |
| Example: Employees in a company are required to authenticate through the network before accessing their company email | Example: After an employee successfully authenticates, the system determines what information the employees are allowed to access |

Source: https://auth0.com/docs/get-started/identity-fundamentals/authentication-and-authorization

# SOME METHODS FOR AUTH IN REST APIS

- HTTP Basic Authentication
- API Keys
- JSON Web Tokens (JWT)
- OAuth 2.0
  - OpenID Connect (OIDC)

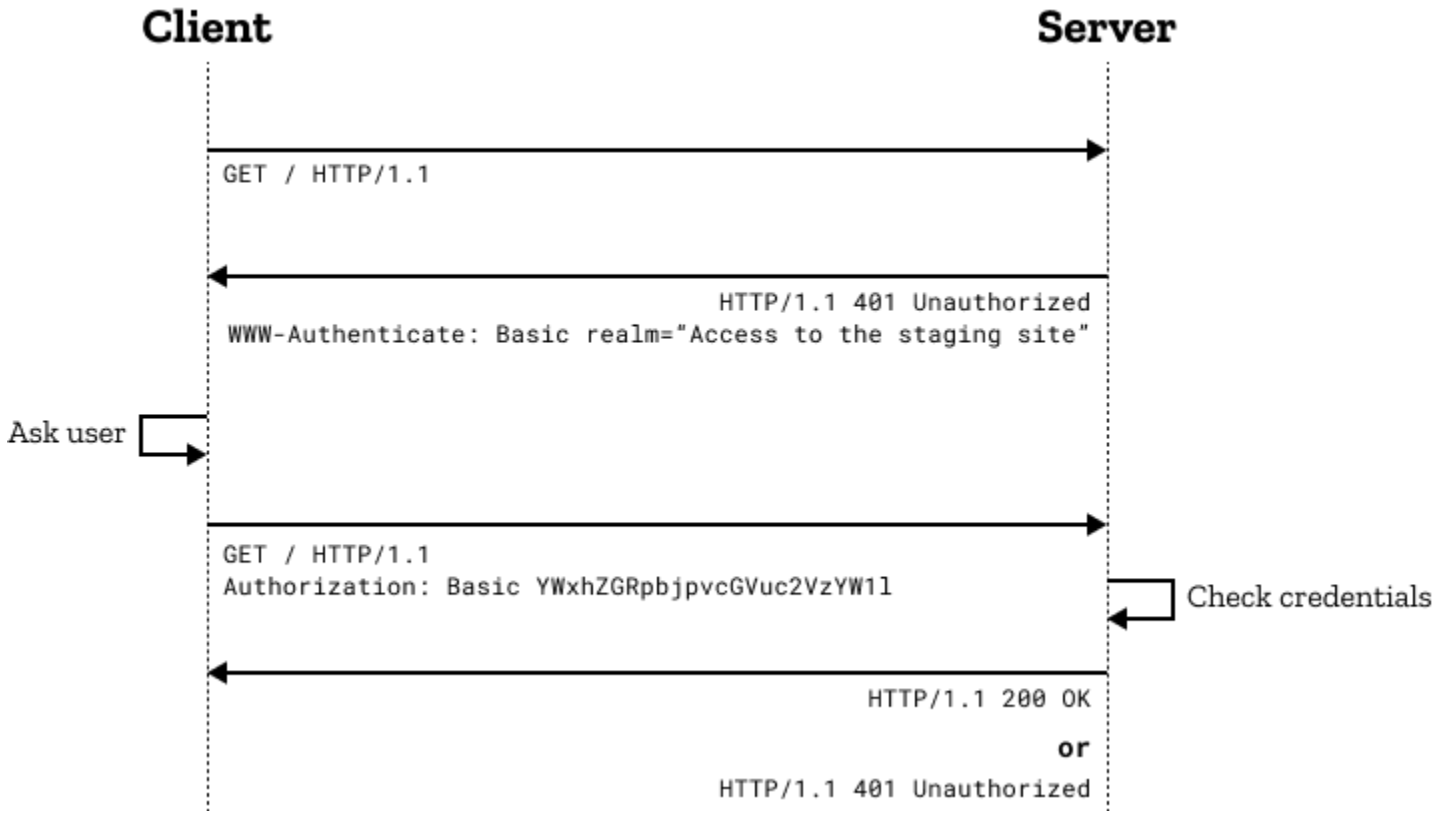
# HTTP BASIC AUTHENTICATION

- Most basic form of authentication in APIs.
- The client sends the username and the password of the user encoded in Base64 encoding technique to the server.
- The credentials are sent in the HTTP header with every request.
  - *Authorization: Basic <username:password>*

# HTTP BASIC AUTHENTICATION

- Very simple to use as it does not require cookies, session identifiers, login pages.
- Lightweight.
  - Good for use-cases like IoT.
- Rarely recommended due to its inherent security vulnerabilities.
  - For better security use with HTTPS over SSL/ TLS.

# HTTP BASIC AUTHENTICATION




Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

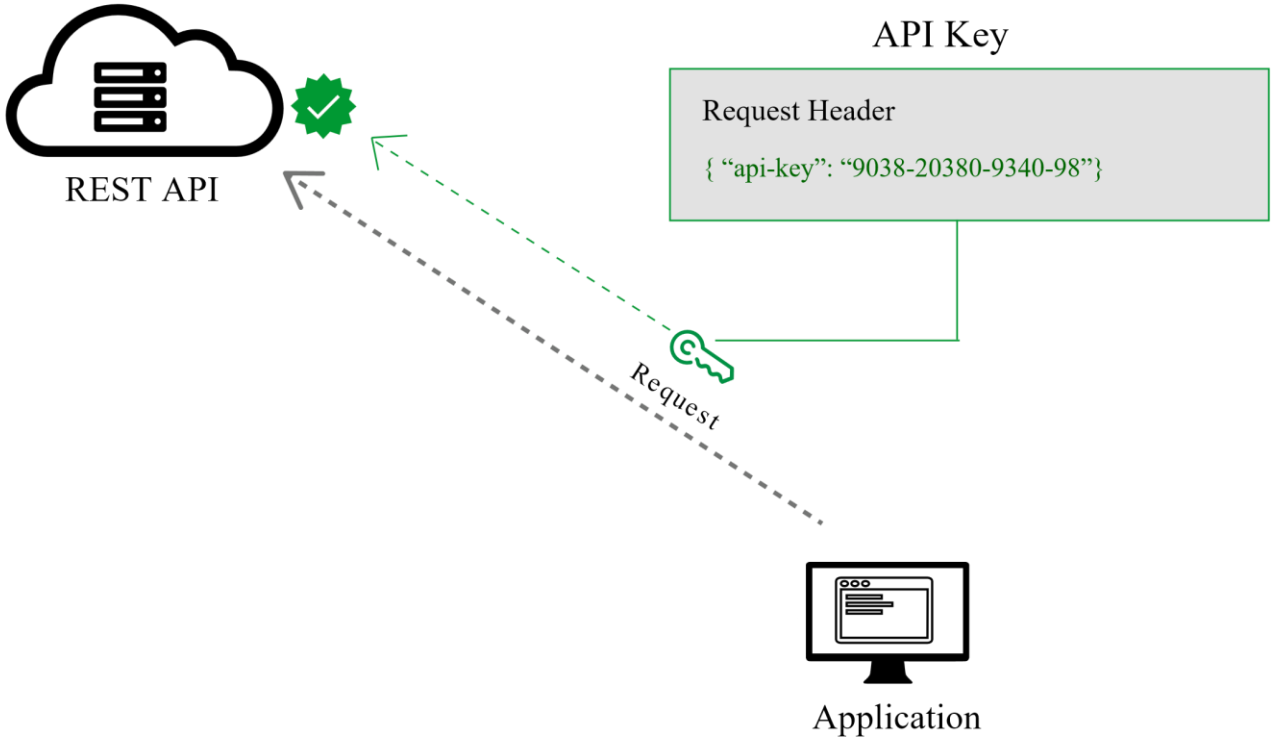
# API KEYS

- A **unique generated key value** is assigned for each first-time user.
- Thereafter, the user sends all requests with the key as
  - a query parameter or
  - in the HTTP header (the standard way)
- Simple, fast and convenient.

# API KEYS

- A method of authentication, not authorization.
- No expiration.
  - Once the key is stolen, it may be used indefinitely unless revoked.
- As with Basic HTTP authentication, not recommended to be used as is.
  - For better security use with HTTPS over SSL/ TLS.

# API KEYS

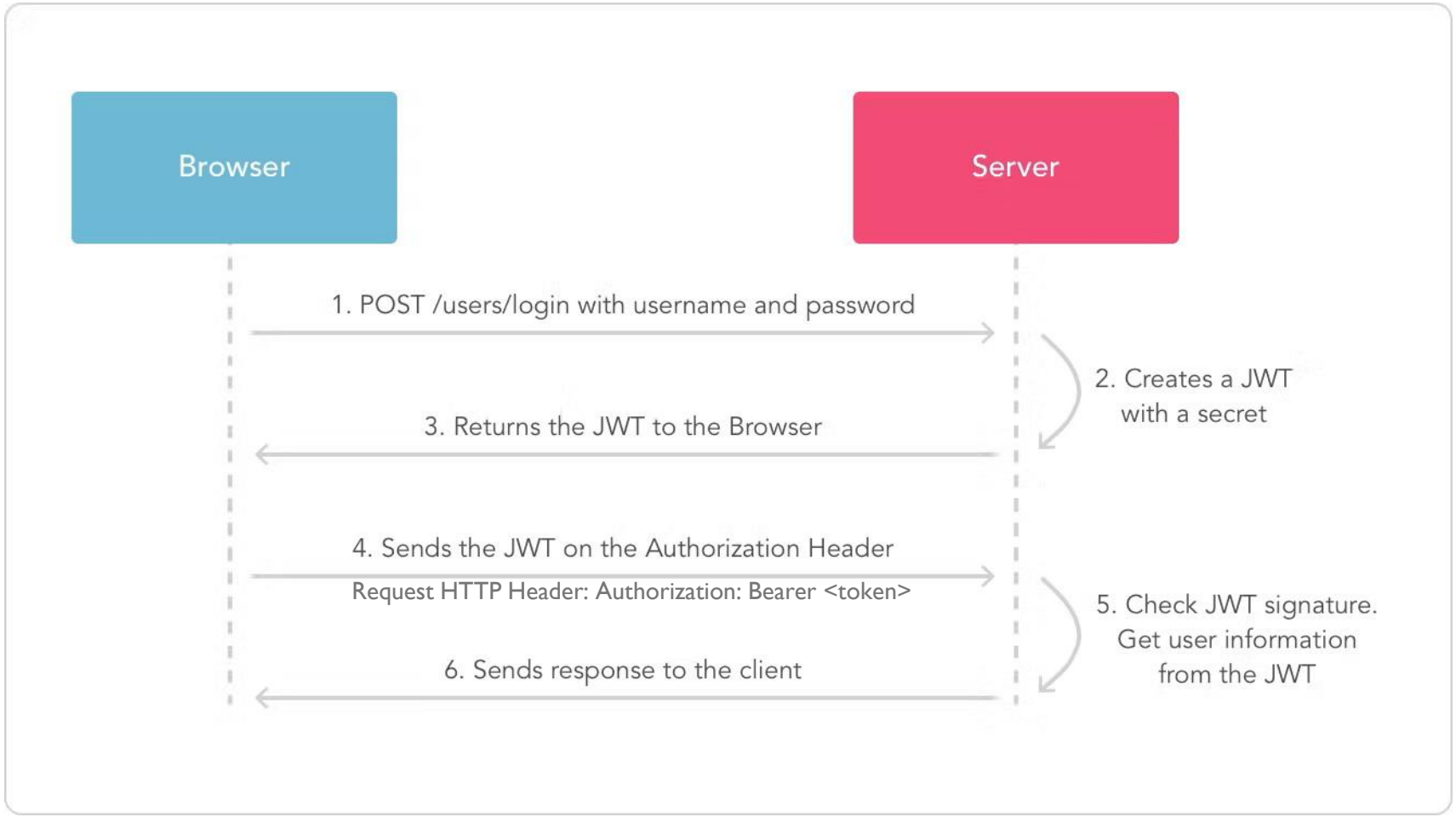



Source: https://blog.restcase.com/4-most-used-rest-api-authentication-methods/

# JSON WEB TOKENS (JWT)

- A compact and a self-contained means to securely exchange information between parties as a JSON object.
- Digitally signed, therefore can be verified and trusted.
  - Using secrets
  - Using public/ private key pairs
- Used for user **authorization** after authentication.
- Very popular!

# JSON WEB TOKENS (JWT)



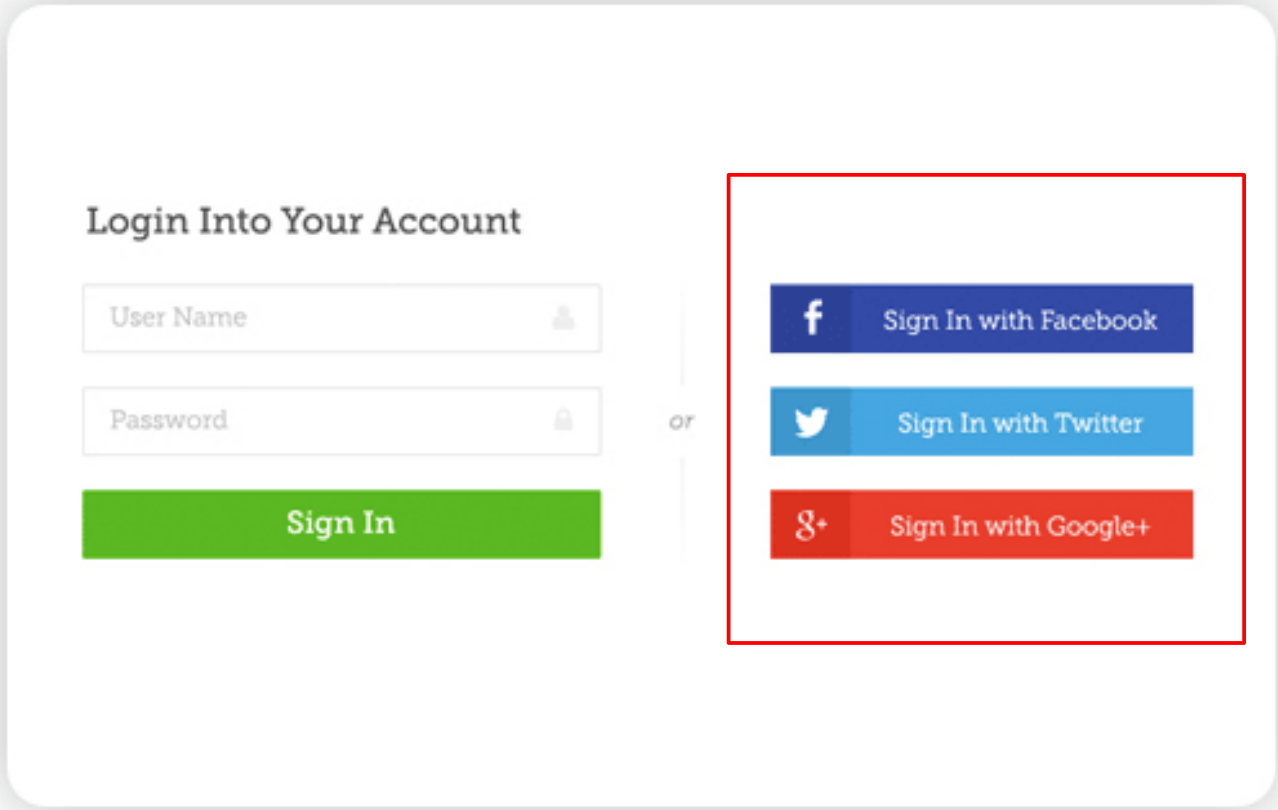

Source: https://lo-victoria.com/introduction-to-rest-api-authentication-methods

# JSON WEB TOKENS (JWT)

- JWT Decoder – example JWT demo
- JWT guarantees that data is not tampered because of the signature.
- The contents of a JWT can be seen by anyone that intercepts the token.
  - It is just serialized, not **encrypted**.
  - It is recommended to **use HTTPS** to avoid this problem.
- JWT is pronounced as "Jot".
- JWT is good for implementing stateless APIs.

# OAUTH 2.0

Have you seen this before?

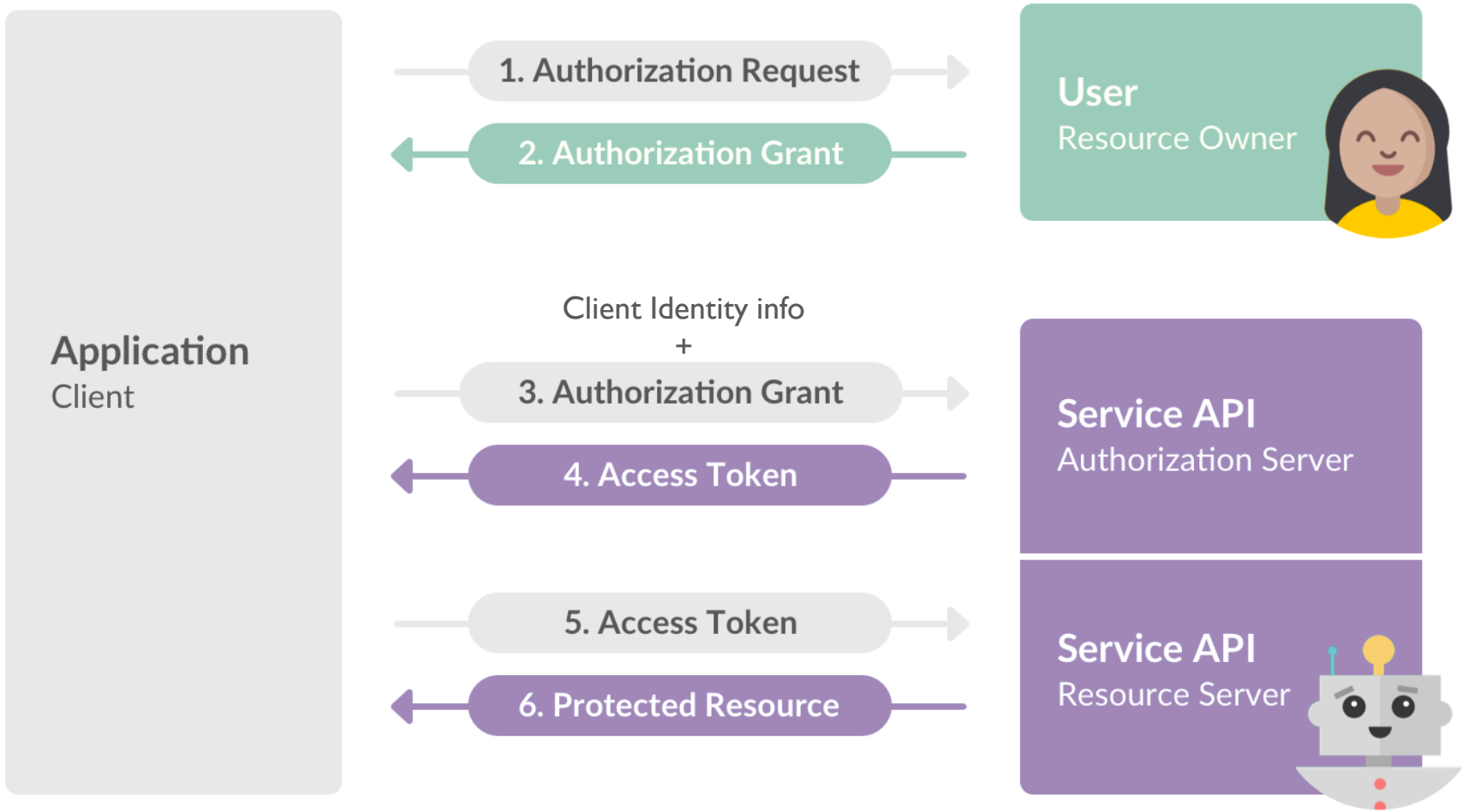



Source: https://lo-victoria.com/introduction-to-rest-api-authentication-methods

# OAUTH 2.0

- It is an authorization framework/ protocol.
- If an implementation also conforms to OpenID Connect specification, it will also support authentication.
  - This is how social authentication works!
- OAuth works by,
  - **Delegating user authentication** to a service that hosts a user account
  - **Authorizing third-party applications** to access that user account

# OAUTH 2.0




Source: https://lo-victoria.com/introduction-to-rest-api-authentication-methods

# OAUTH 2.0

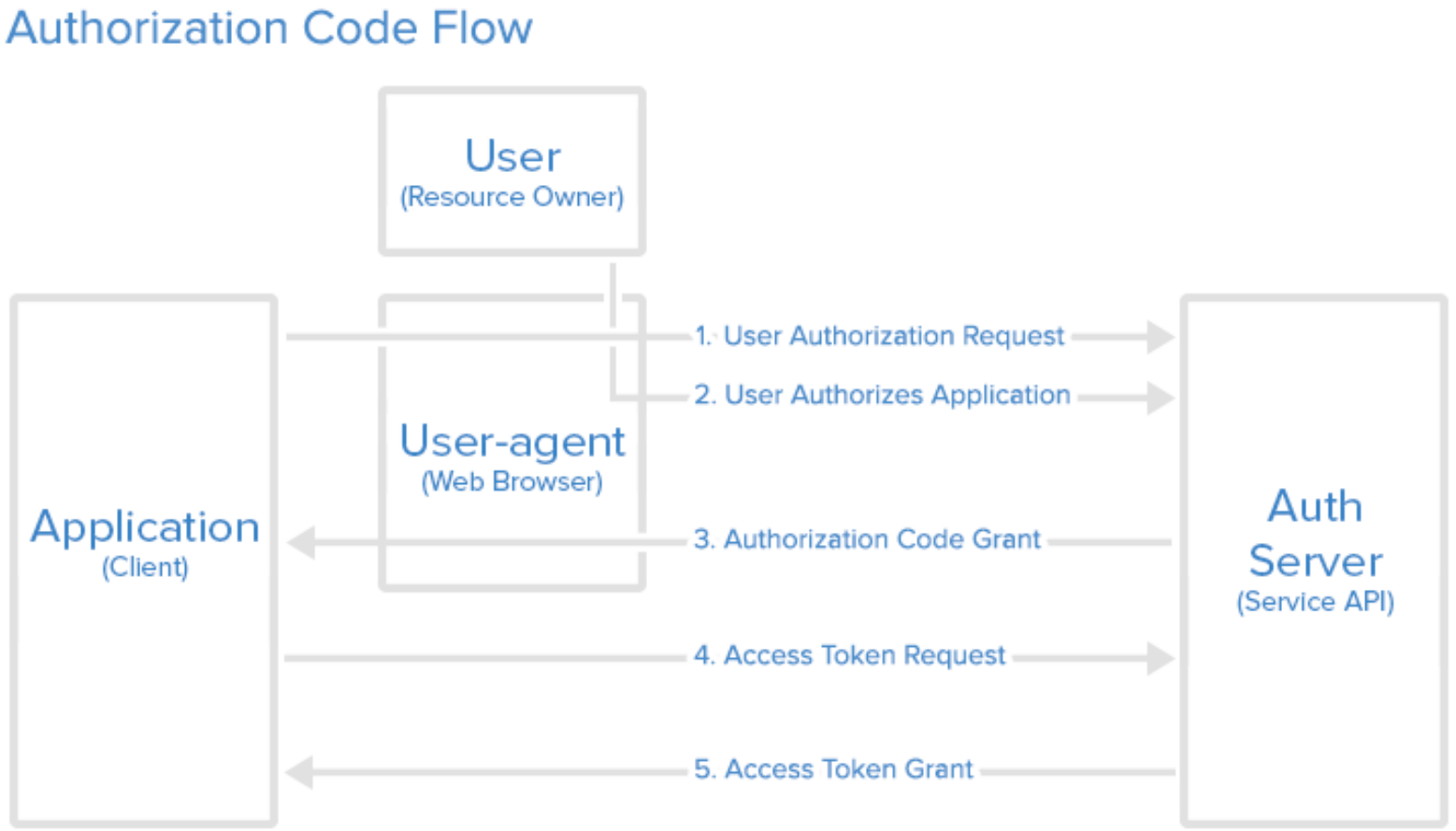
- OAuth Roles
  - Resource Owner: The resource owner is the user who authorizes an application to access their account.
  - Client: The client is the application that wants to access the user's account.
  - Authorization Server: The authorization server verifies the identity of the user then issues access tokens to the client application.
  - Resource Server: The resource server hosts the protected resources.

# OAUTH 2.0

- An OAuth 2.0 flow (grant type) is a way of retrieving an Access Token.
- OAuth 2.0 supports different authorization flows (also known as grant types).
  - Authorization Code Flow – Widely used for server-side and mobile web applications.
  - PKCE
  - Client Credentials flow etc.

# OAUTH 2.0

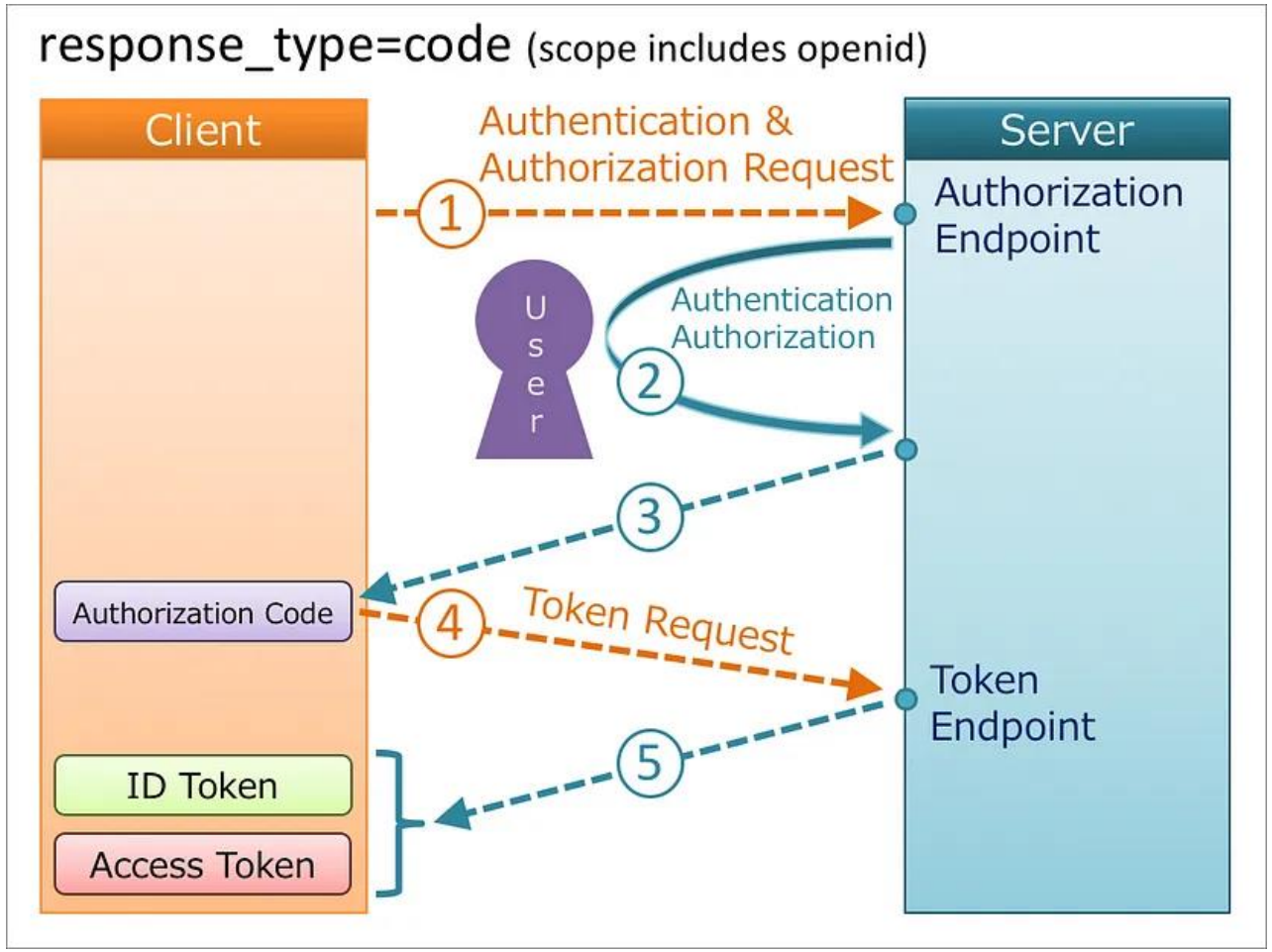
## Authorization Code Flow




Source: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2

# OPENID CONNECT

Authorization Code Flow
+ with Authentication




Source: https://darutk.medium.com/diagrams-of-all-the-openid-connect-flows-6968e3990660

# OAUTH 2.0

- Why use OAuth 2.0?
  - It enables an application to obtain limited access (scopes) to a user's data without the user having to share their password.
- Why use OpenID Connect (which is an extension of OAuth 2.0)?
  - Accelerate Sign Up Process at Your Favorite Websites
  - No need to maintain multiple usernames/ passwords.
  - Minimize password security risks.

# OAUTH 2.0

- [YouTube] OAuth 2.0: An Overview

# SELF-STUDY ACTIVITY

- There is much more in OAuth 2.0.
  - Read this introduction to OAuth 2.0 to get a more detailed understanding.
- Spring Boot and OAuth 2.0
  - Check out these official tutorials from Spring.

# SUMMARY

- Authentication vs. Authorization
- Importance of Authentication and Authorization
- Some methods for Authentication and Authorization in REST APIs
  - HTTP Basic Authentication
  - API Keys
  - JSON Web Tokens (JWT)
  - OAuth 2.0
    - OpenID Connect (OIDC)

# REFERENCES


1. 3 Common Methods of API Authentication Explained
2. 4 Most Used REST API Authentication Methods
3. How does HTTPS actually work?
4. HTTP authentication
5. API Keys
6. Introduction to JSON Web Tokens
7. Authorization Code Flow

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# FRONTEND DEVELOPMENT OVERVIEW

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

# LEARNING OUTCOMES

- After completing this lecture, you will be able to,
  - Describe the main concepts associated with frontend web development.
  - Apply the knowledge gained in developing more robust frontend web applications.
  - Apply the learned concepts to create more secure web applications.

# CONTENTS

- What is frontend development?
- Why is frontend development important?
- HTML, CSS, JS
- Document Object Model (DOM)
- State in Web apps
  - State Management
- Web Application Security
  - OWASP Top 10 Security Risks & Vulnerabilities
  - HTTP Security Headers
- Wrap up
- Summary

# WHAT IS FRONTEND DEVELOPMENT?

- Frontend development is the development of visual and interactive elements of a website that users interact with directly.
- Also known as Client-side development.

# WHY IS FRONTEND DEVELOPMENT IMPORTANT?

- Backend vs. Frontend
  - Frontend web development focuses on **creating a good look and feel for the user.**
  - Backend web development focuses on engineering the web application's structure, logic and data.

# WHY IS FRONTEND DEVELOPMENT IMPORTANT?

- Frontend development can be considered under two aspects.
  - **Designing** a good look and feel
    - HCI Design, UI/ UX etc.
  - **Engineering** the designed good look and feel
    - Frontend technologies
      - E.g., HTML, CSS, JS, Different frameworks, libraries etc.

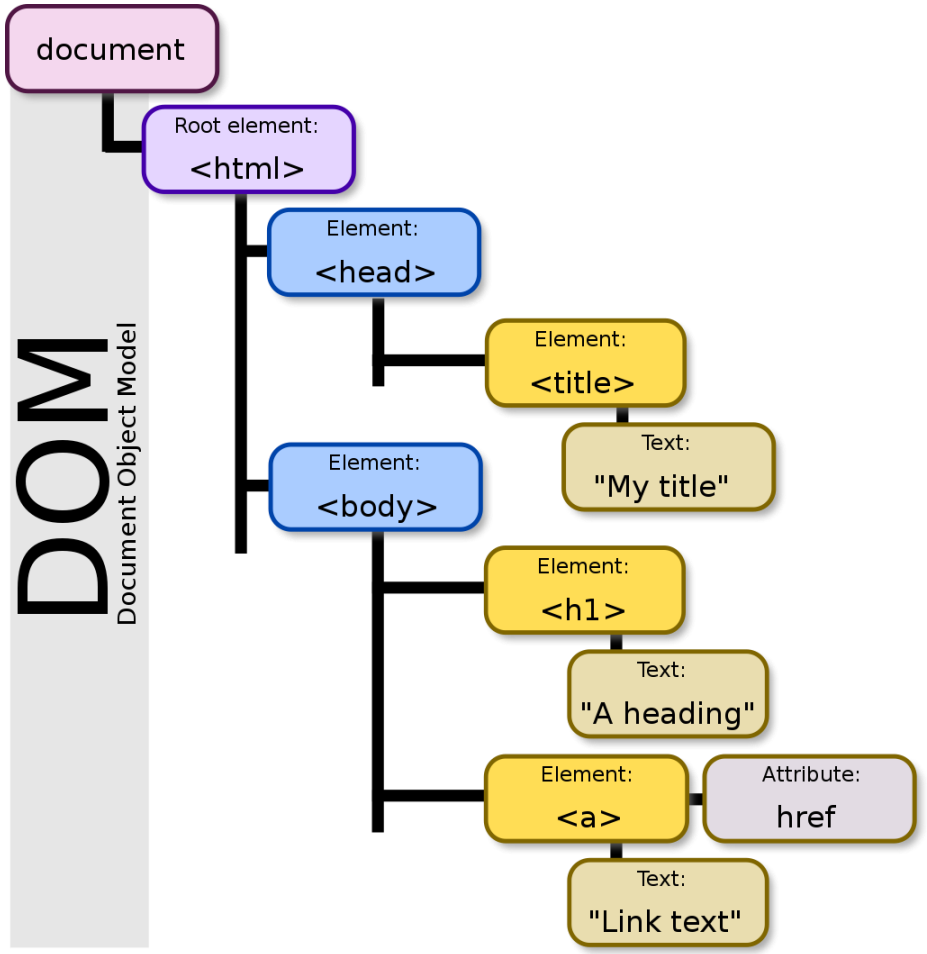
# HTML, CSS, JS

- Frontend development is done with a combination of technologies in which,
  - **HTML** provides the structure
  - **CSS** provides the styling and layout
  - **JavaScript** provides the dynamic behavior and interactivity

# DOCUMENT OBJECT MODEL (DOM)

- What do we actually manipulate with these technologies?
  - Document Object Model (DOM)!

# DOCUMENT OBJECT MODEL (DOM)

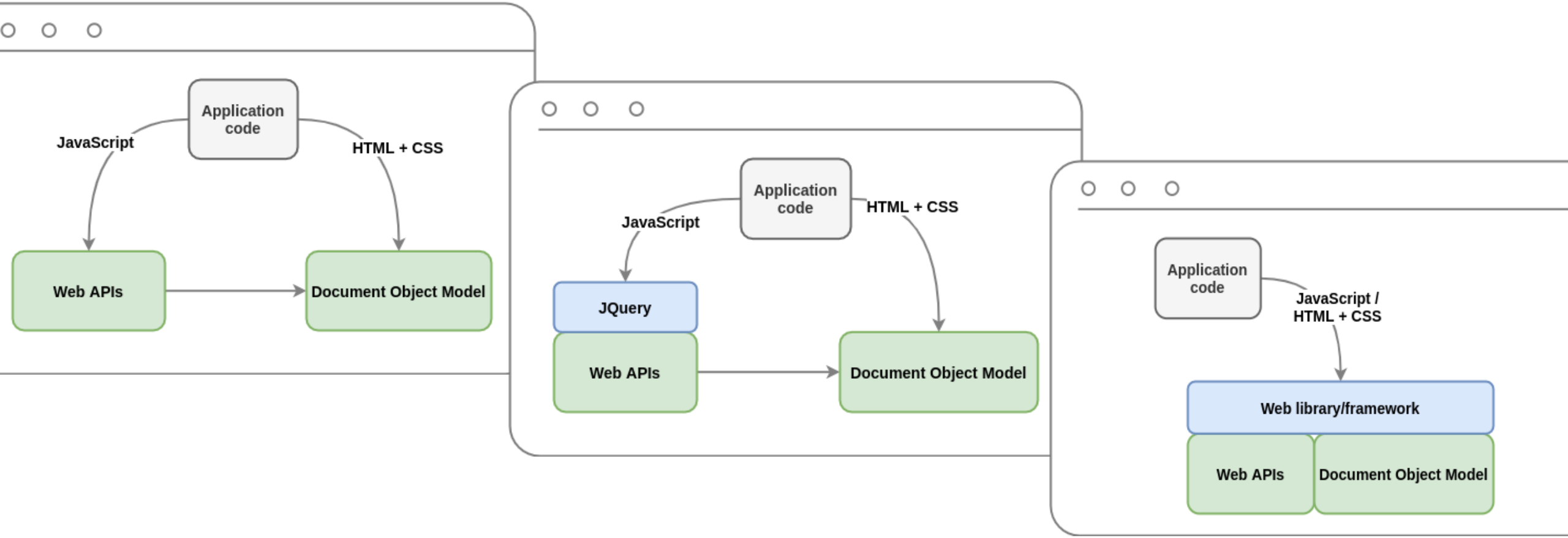



Source: https://en.wikipedia.org/wiki/Document_Object_Model

# DOCUMENT OBJECT MODEL (DOM)

- Document Object Model (DOM) – An object-oriented representation of the HTML structure
- DOM is not a JS functionality.
  - JS is the mostly used language to manipulate it, but other languages can also be used.
- It is just a Web API.
- [YouTube] The DOM in 4 minutes

# EVOLUTION OF FRONTEND DEVELOPMENT




Source: https://www.epineda.net/the-evolution-of-front-end-development/amp/

# DOCUMENT OBJECT MODEL (DOM)

- What initiates modifications to the Document Object Model (DOM)?
  - The **Application State**!


Source: https://transforming.com/2019/05/07/current-state-future-state-and-embracing-change/

# STATE IN WEB APPS

- **State** refers to the current condition or data of the application at a given point in time.
- State is important in web applications because it affects the behavior and appearance of the application.
- [YouTube] What is "State" in Programming?

# STATE MANAGEMENT IN WEB APPS

- **State management** is a concept in web development that refers to the management and manipulation of data or information within an application.
- In simple terms, it involves **controlling the data flow** and ensuring that the application's **data is consistent** and **up-to-date**.

# STATE MANAGEMENT IN WEB APPS

- Proper management of state is important for the performance and reliability of web applications.
- Poorly managed state can result in slow application performance, data inconsistencies, or security vulnerabilities.

# STATE MANAGEMENT IN WEB APPS

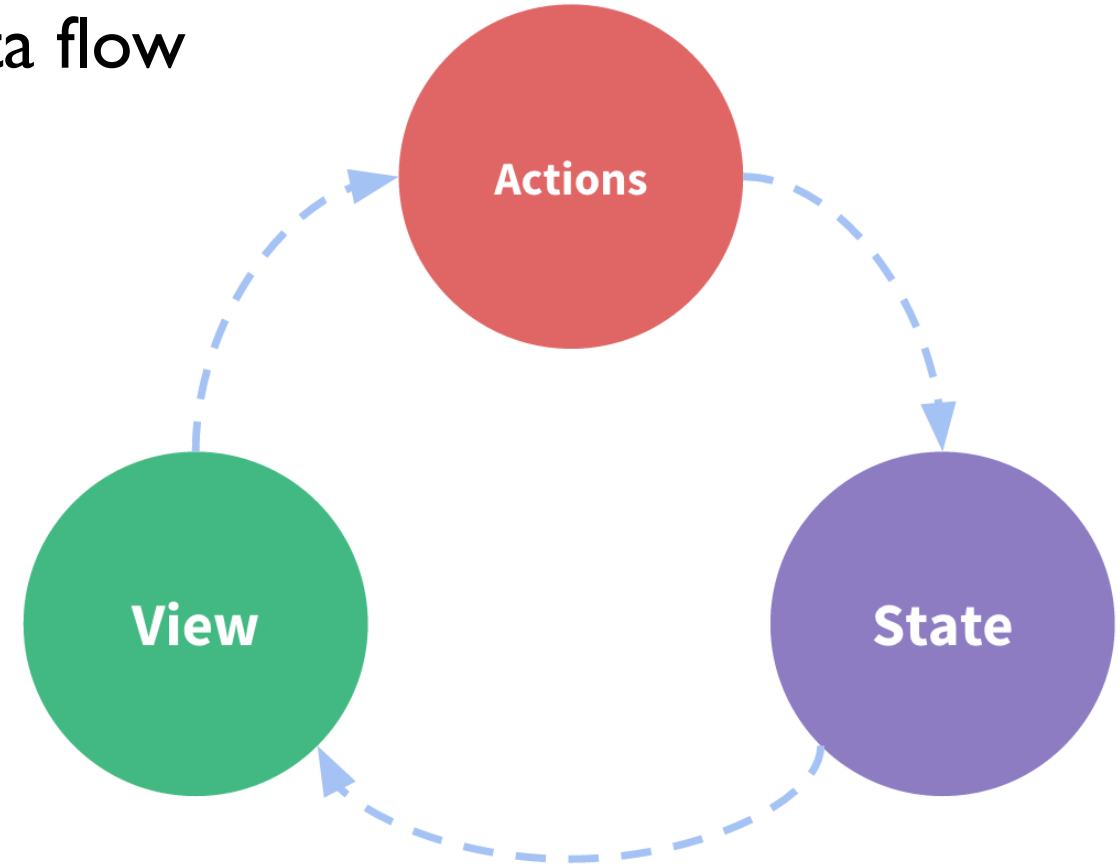
- How is State managed in web apps?
  - Local state management
  - Global state management
    - Redux

# STATE MANAGEMENT IN WEB APPS

- Redux
  - Redux is a pattern and library for managing and updating application state.
  - Redux helps you manage "global" state - state that is needed across many parts of your application.
  - It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.
  - Redux is commonly used with React.
  - Read more on Redux here.

# STATE MANAGEMENT IN WEB APPS

## Redux one-way data flow




Source: https://redux.js.org/tutorials/essentials/part-1-overview-concepts

# STATE MANAGEMENT IN WEB APPS

- Redux is not always needed.
  - **Do not use Redux until** you have problems with vanilla React.
- When to use Redux?
  - You have **large amounts of application state** that are needed in many places in the app.
  - The app state is updated frequently over time.
  - The logic to update that state may be complex.
  - The app has a medium or large-sized codebase and might be worked on by many people.

# WEB APPLICATION SECURITY

- OWASP, or the Open Web Application Security Project, is a nonprofit organization focused on software security.
- OWASP Top 10 is a list of the 10 most common web application security risks.

# WEB APPLICATION SECURITY

- OWASP Top 10 Vulnerabilities 2021
  1. Broken Access Control
  2. Cryptographic Failures
  3. Injection
  4. Insecure Design
  5. Security Misconfiguration
  6. Vulnerable and Outdated Components
  7. Identification and Authentication Failures
  8. Software and Data Integrity Failures
  9. Security Logging and Monitoring Failures
  10. Server-Side Request Forgery

# WEB APPLICATION SECURITY

- Testing an application for the OWASP vulnerabilities is a **crucial step** in application development.
- [YouTube] What is The OWASP Top 10? | A Radware Minute
- [Mandatory] Read OWASP Top Security Risks & Vulnerabilities 2021.

# WEB APPLICATION SECURITY

- HTTP Security Headers
  - HTTP security headers are a subset of HTTP headers that is related specifically to security.
  - Setting suitable headers in your web applications and web server settings is an easy way to greatly improve the resilience of your web application against many common attacks.
  - Read this link on HTTP Security Headers.
  - Some additional information can be found here too.

# WRAP UP

- See this link for a step-by-step guide on becoming a frontend developer.
- Here's a nice overview on Frontend development: [YouTube] Frontend web development - a complete overview

# SUMMARY

- What is frontend development?
- Why is frontend development important?
- HTML, CSS, JS
- Document Object Model (DOM)
- State in Web apps
  - State Management
- Web Application Security
  - OWASP Top 10 Security Risks & Vulnerabilities
  - HTTP Security Headers

# REFERENCES


1. What is the Document Object Model?
2. Introduction to the DOM
3. The Best React State Management Tools for Enterprise Applications
4. Redux Essentials, Part 1: Redux Overview and Concepts
5. OWASP Top 10 Vulnerabilities

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# DEVOPS AND CI/ CD - AN INTRODUCTION

PROGRAMMING APPLICATIONS AND FRAMEWORKS (IT3030)

# LEARNING OUTCOMES

- After completing this lecture, you will be able to,
  - Describe the failures in the traditional software deployment methods, and the reasons for the rise of DevOps.
  - Describe the main components of DevOps.
  - Describe what CI/ CD is and how its application is beneficial in modern software development.
  - Describe the importance of Test Automation in DevOps.
  - Apply the concepts learnt in solving real world problems

# CONTENTS

- Journey so far
- Deploying an app
  - Typical Deployment Scenario
  - Failures
- DevOps
  - Core DevOps principles
  - Continuous Integration (CI)
  - Continuous Delivery and Continuous Deployment (CD)
  - Software Testing and Test Automation
- Wrap Up
- Summary

# JOURNEY SO FAR

- What was discussed so far,
  - Software Frameworks
  - Version controlling with Git
  - Git Workflows
  - Web application architectures
  - REST APIs
  - An overview on Frontend development

# DEPLOYING AN APP

- We have discussed how we engineer a web application so far.
- However, engineering the application is not enough.

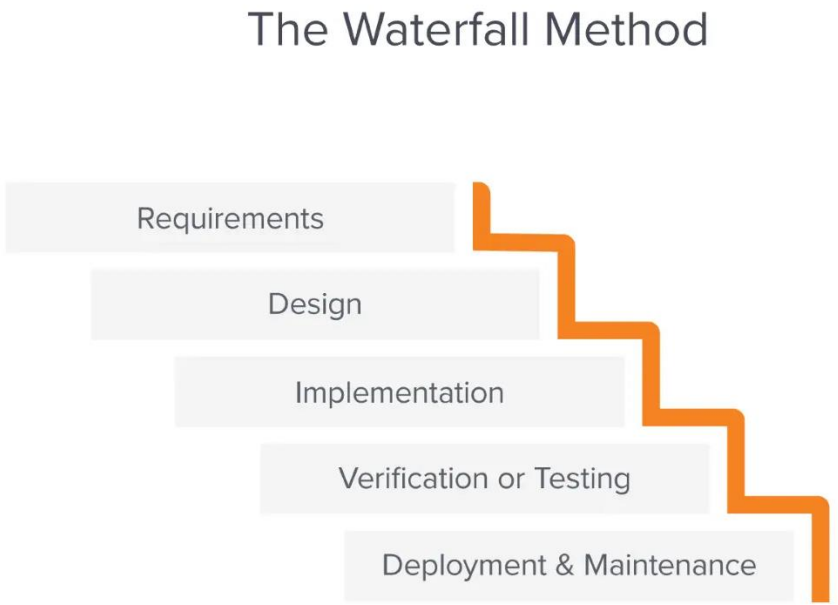
- It must be **deployed**!
  - Why? So that actual users can use the application.

# DEPLOYING AN APP

- Application Deployment, also known as Software Deployment, is the process of installing, configuring, updating, and enabling one application or suite of applications that make a software system **available for use.** (Source)

# TYPICAL DEPLOYMENT SCENARIO

- How were deployments handled traditionally?
  - Waterfall model was the most common SDLC model used back in the day.
    - SDLC - Software Development Life Cycle
  - In Waterfall model, each stage is carefully planned and executed sequentially by a dedicated team.
    - E.g.: Business stakeholders, Business Analysts, Architects, Developers, Quality Assurance, IT Operations




Source: https://business.adobe.com/blog/basics/waterfall

# TYPICAL DEPLOYMENT SCENARIO

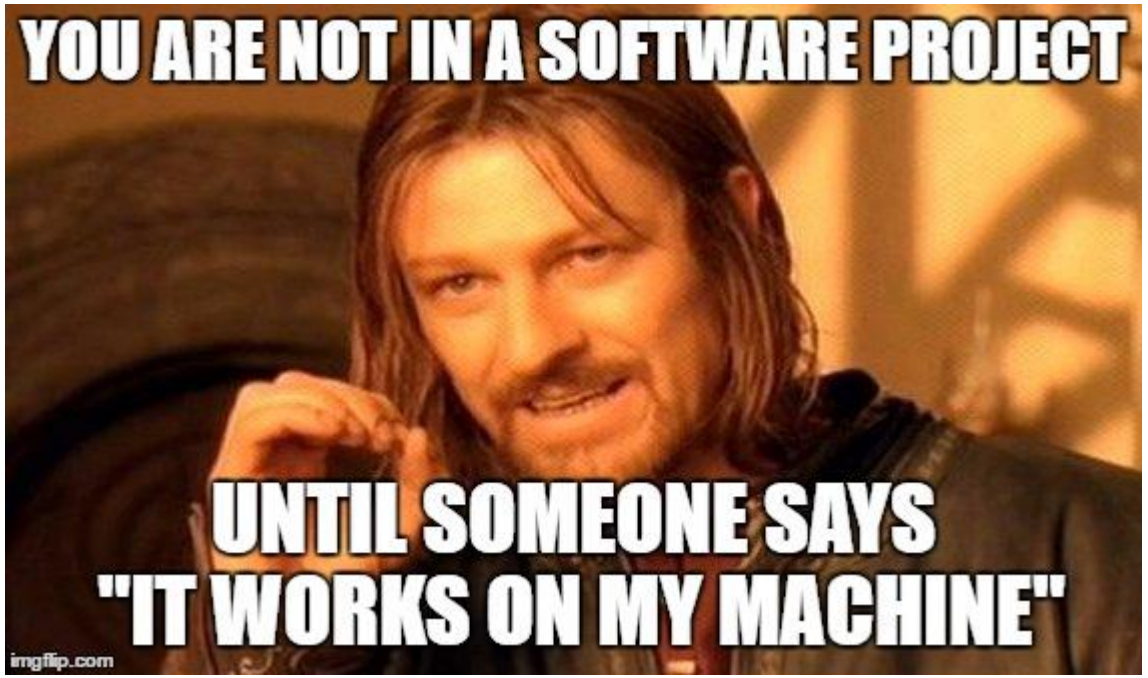
- A release would be carried out every few months.
- First, the developers would develop the solution.
- Then the QA team would start release testing the application leading to the release.

# TYPICAL DEPLOYMENT SCENARIO

- The QA team would then discover bugs.
  - QA team would frantically report the bugs.
  - Developers would frantically fix the bugs.
- Now the QA team has to retest the fixes.
  - This is on top of the functionality remaining to be tested!
- Typically, the developers and/ or QA teams would get overwhelmed with the workload as the release date becomes closer.

# TYPICAL DEPLOYMENT SCENARIO

- Then the Developers, QA and the IT Operations teams would recommend to push ahead the release date given the problems.
  - Business stakeholders would not agree!
  - They dictate that release be carried out as planned as they have their own strategic goals to meet.
- IT Ops would also run into various problems deploying the application.
  - Famous example: It works on my machine!




Source: https://elbruno.com/you-are-not-in-a-software-project-until-someone-says-it-works-on-my-machine/

# TYPICAL DEPLOYMENT SCENARIO

- Finally, the release would be carried out as planned even with the issues.
  - It will be rolled back in the next couple of hours after doing everything to make the application work fails.
    - Redeploy again with several components disabled.
  - Fixes will be carried out for some time after the release to stabilize the product.
  - The product is actually usable only after few days/ weeks after the release.
- **Rinse and repeat** for the next release!

# TYPICAL DEPLOYMENT SCENARIO: FAILURES

- Each team (Dev/ QA/ IT Ops) worked separately and had competing objectives ("siloed").
- Each team had their own goals.
  - Resulted in botched releases and unhappy customers.
- Lots of firefighting/ finger pointing - Very stressful environment!

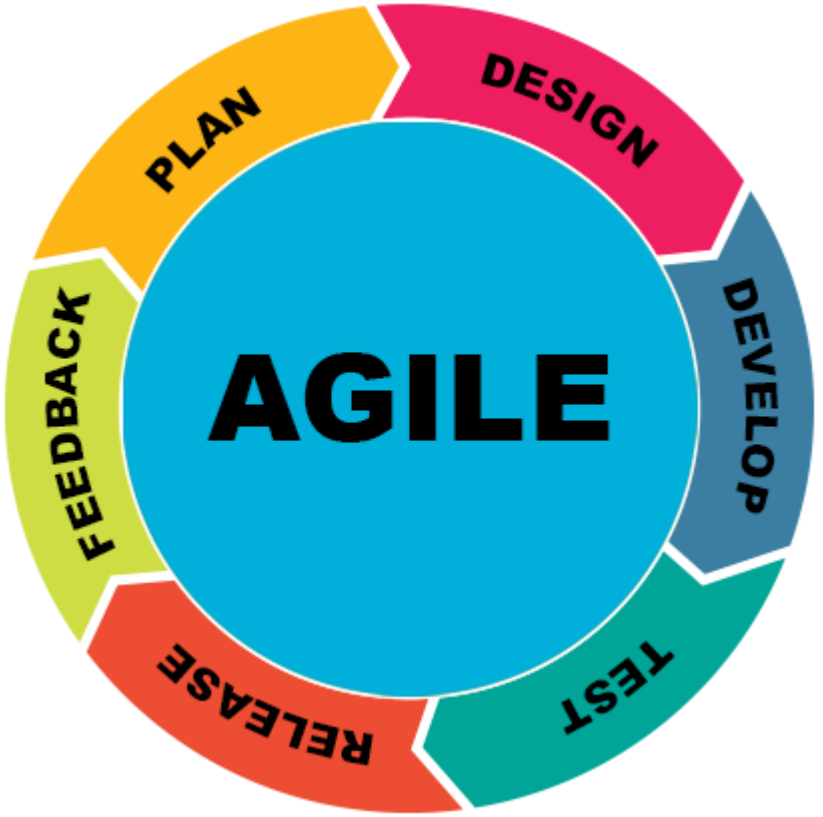
Source: https://resources.biginterview.com/behavioral-interviews/behavioral-interview-questions-conflict/

# DEVOPS

- Somewhere between 2007-2009 different communities started raising concerns about the level of dysfunction in doing releases this manner.
- There had to be a better way of doing things!
  - Gradually a movement appeared against the traditional mindset of keeping different stakeholders separated in different silos.
- This is the origin of **"DevOps"**.
- [YouTube] What is DevOps?

# DEVOPS

- DevOps stands for "Developer-Operations".
- The concept behind DevOps is simple.
  - Bring the Developers, QA, IT Operations (IT Ops) teams together to work collaboratively on planning, building, testing and releasing software with shared responsibility for successful business outcomes.
  - Based on **Agile approach** to software development!
  - DevOps is not just a method for Software Development. It's a **culture**.

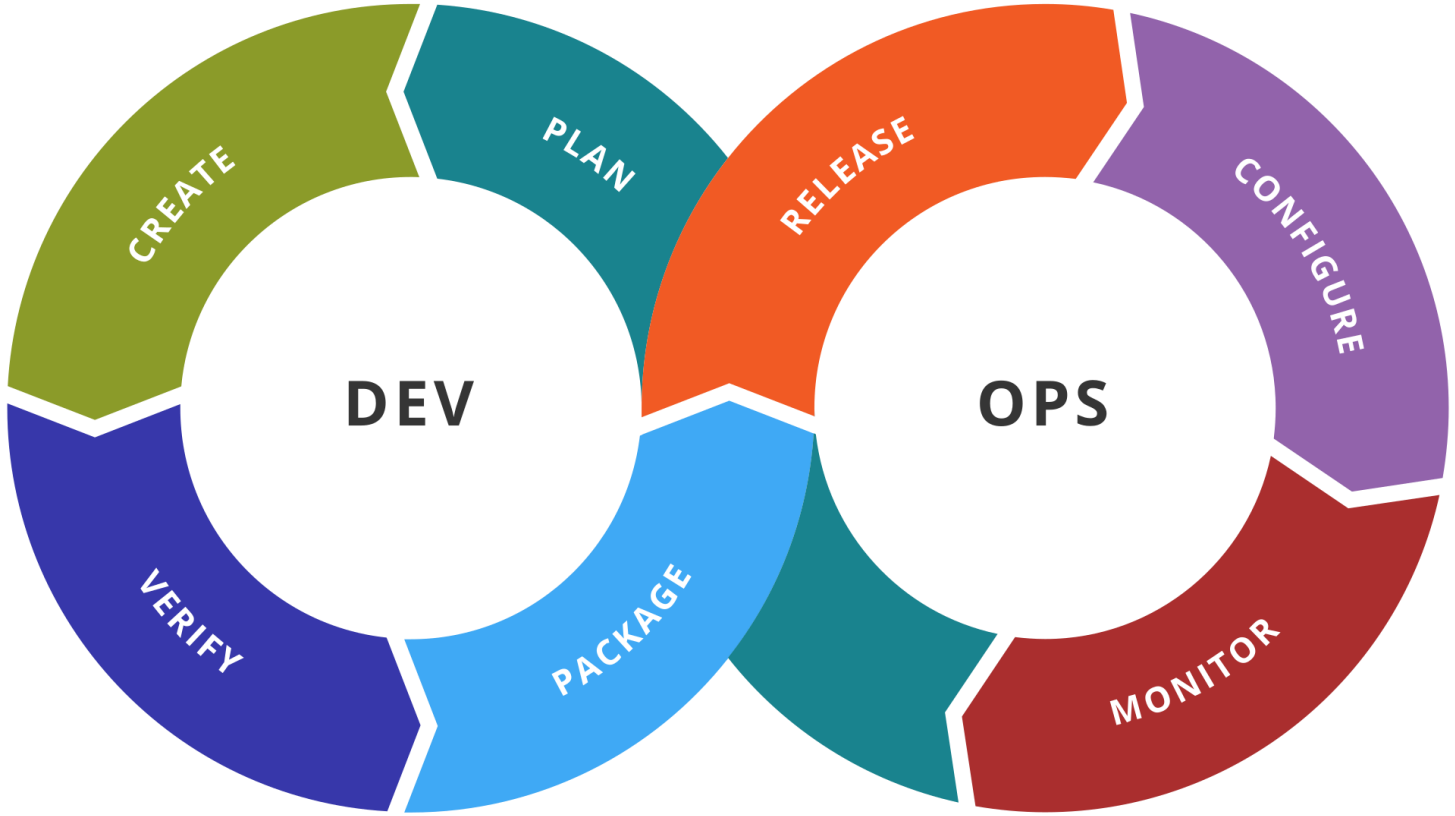



Source: https://technology.berkeley.edu/tpo/agile

# DEVOPS

- DevOps is a combination of software development (dev) and operations (ops). It is defined as a software engineering methodology which aims to integrate the work of development teams and operations teams by facilitating a culture of collaboration and shared responsibility. (Source)

# DEVOPS STAGES




Source: https://en.wikipedia.org/wiki/DevOps_toolchain

# CORE DEVOPS PRINCIPLES

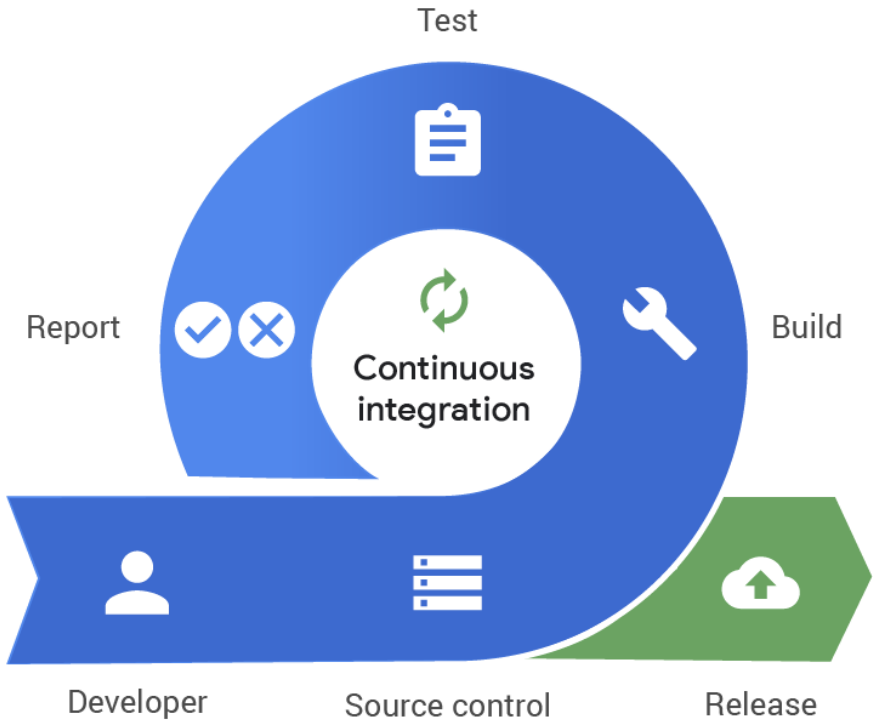
- Automation of the software development lifecycle
  - Automating the SDLC as much as possible increases productivity by reducing the introduction of human errors and freeing up humans from repetitive tasks.
  - Mainly consists of,
    - **Continuous Integration (CI)**
    - **Continuous Delivery and Continuous Deployment (CD)**
    - Infrastructure Automation (Automated provisioning of Environments and other resources)
    - Test Automation - **Shift Left** Testing (Read this)

# CORE DEVOPS PRINCIPLES

- Collaboration and communication
  - Developers, QA, stakeholders, IT Ops should communicate clearly to collaborate efficiently. At the end of the day, it is everyone's responsibility to deliver software which satisfies business requirements.
- Continuous improvement
  - It is the practice of focusing on experimentation, minimizing waste, and optimizing for speed, cost, and ease of delivery.
- Customer-centric action
  - DevOps teams use short feedback loops with customers and end users to develop products and services centered around user needs.

# CONTINUOUS INTEGRATION (CI)

- Continuous integration is the practice of,
  - Frequently pushing all code changes into a designated main branch of a code repository,
  - automatically kicking off a build whenever there is a new change(s),
  - And running automatic tests against the build to verify the change.
    - Developers need to create the tests too if they don't exist already.




Source: https://www.pagerduty.com/resources/learn/what-is-continuous-integration/

# CONTINUOUS INTEGRATION (CI)

- By merging changes frequently and triggering automatic testing and validation processes,
  - the issues can be discovered early and fixed then and there.
  - Integration challenges that can happen when waiting for release day to merge changes into the release branch can be avoided.
  - Quality of the product can be verified from the early days of development rather than near the release date (Shift-left testing).

# CONTINUOUS DELIVERY (CD)

- **Continuous delivery** is an extension of continuous integration.
- It is the deployment of all built and automatically tested deliverables (basically executables) to a further testing and/or production environment after the CI stage via an automated process.
- This **takes away the stress on a team** to prepare for a delivery for days.

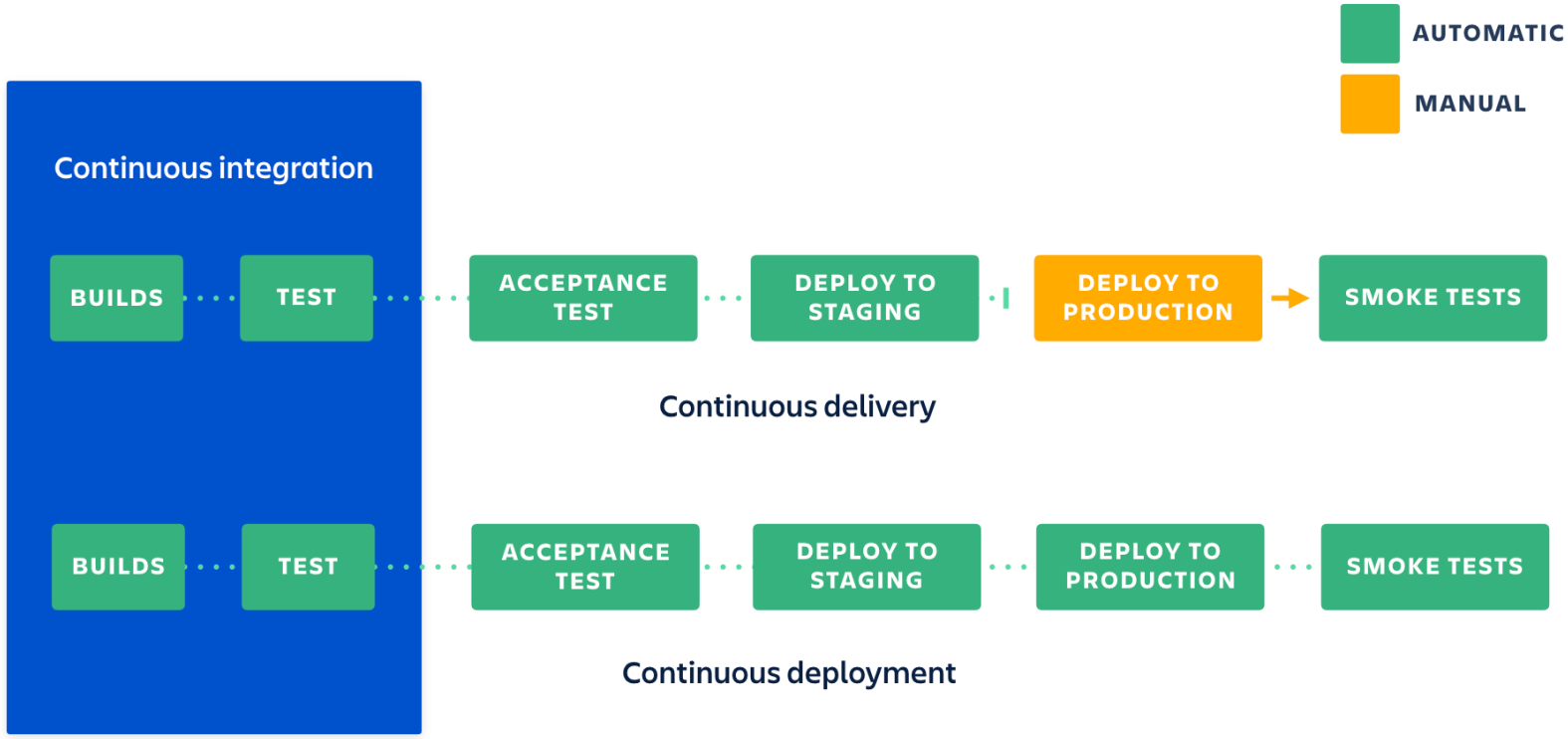
# CONTINUOUS DELIVERY (CD)

- This deployment process is triggered manually. Once the process is triggered, the deployment happens automatically.
- The user has the control over when the automatic deployment should be triggered with just a click of a button.
- **Feature flags** are used to disable/ hide incomplete features from affecting customers in production.

# CONTINUOUS DEPLOYMENT (CD)

- **Continuous Deployment** is a further extension of Continuous Delivery.
- With Continuous Deployment, every change that passes all stages of the production pipeline is released to the customers.
- There is **no human intervention**, and only a failed test will prevent a new change to be deployed to production.
- Feature flags are an inherent part in Continuous Deployments.

# CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY VS. CONTINUOUS DEPLOYMENT




Source: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

# SOME POPULAR TOOLS FOR CI/ CD

**Open Source**


Jenkins | GoCD | GitLab CI | Drone CI | Spinnaker | Buildbot

**SaaS**

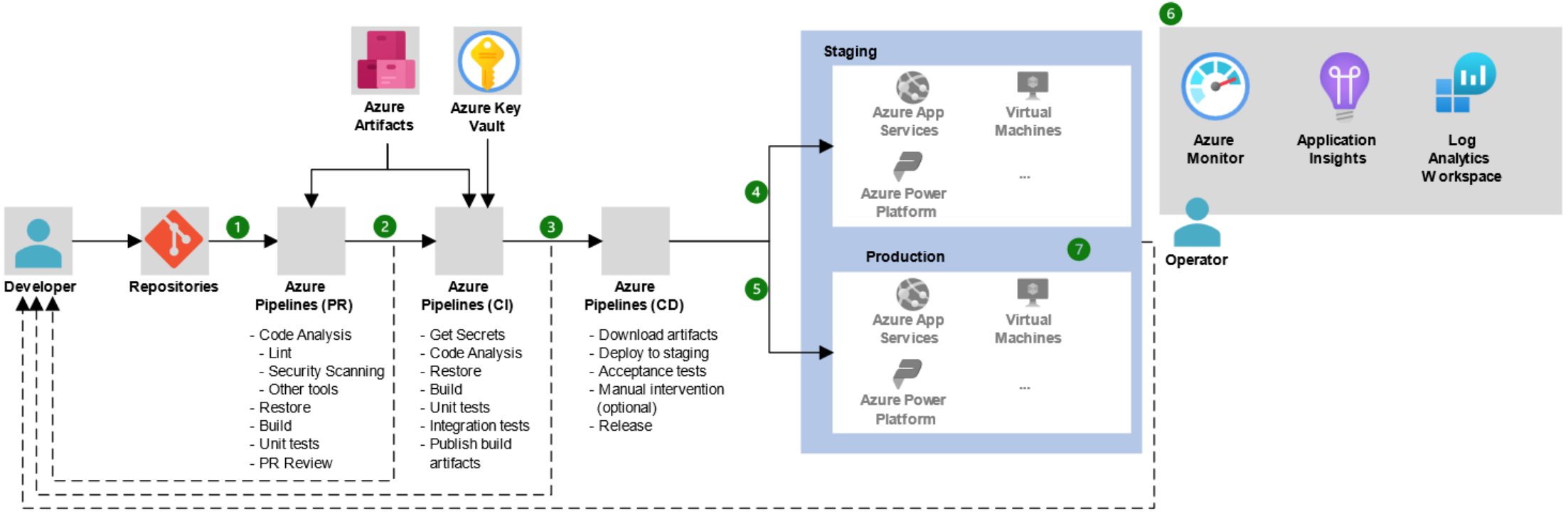

Codeship | Travis CI | TeamCity | CircleCI | GitHub Actions | Semaphore

**Cloud Services**

Azure DevOps | AWS CodePipeline | Google Cloud Build

Source: https://livebook.manning.com/book/pipeline-as-code/chapter-1/135

# EXAMPLE CI/ CD WORKFLOW ON AZURE DEVOPS




Source: https://learn.microsoft.com/en-us/azure/architecture/example-scenario/apps/devops-dotnet-baseline

# BENEFITS OF CONTINUOUS INTEGRATION

- Less bugs get shipped to production as regressions are captured early by the automated tests.
- Building the release is easy as all integration issues have been solved early.
- Less context switching as developers are alerted as soon as they **break the build** and can work on fixing it before they move to another task.
- Testing costs are reduced drastically – your CI server can run hundreds of tests in the matter of seconds.
- Your QA team spends less time testing and can focus on significant improvements to the quality culture.

# BENEFITS OF CONTINUOUS DELIVERY

- The **complexity of deploying software has been taken away**. Your team doesn't have to spend days preparing for a release anymore.
- You can release more often, thus **accelerating the feedback loop** with your customers.

# BENEFITS OF CONTINUOUS DEPLOYMENT

- You can develop faster as there's no need to pause development for releases. Deployments pipelines are triggered automatically for every change.
- Releases are less risky and easier to fix in case of problem as you deploy small batches of changes.
- Customers see a continuous stream of improvements, and quality increases every day, instead of every month, quarter or year.

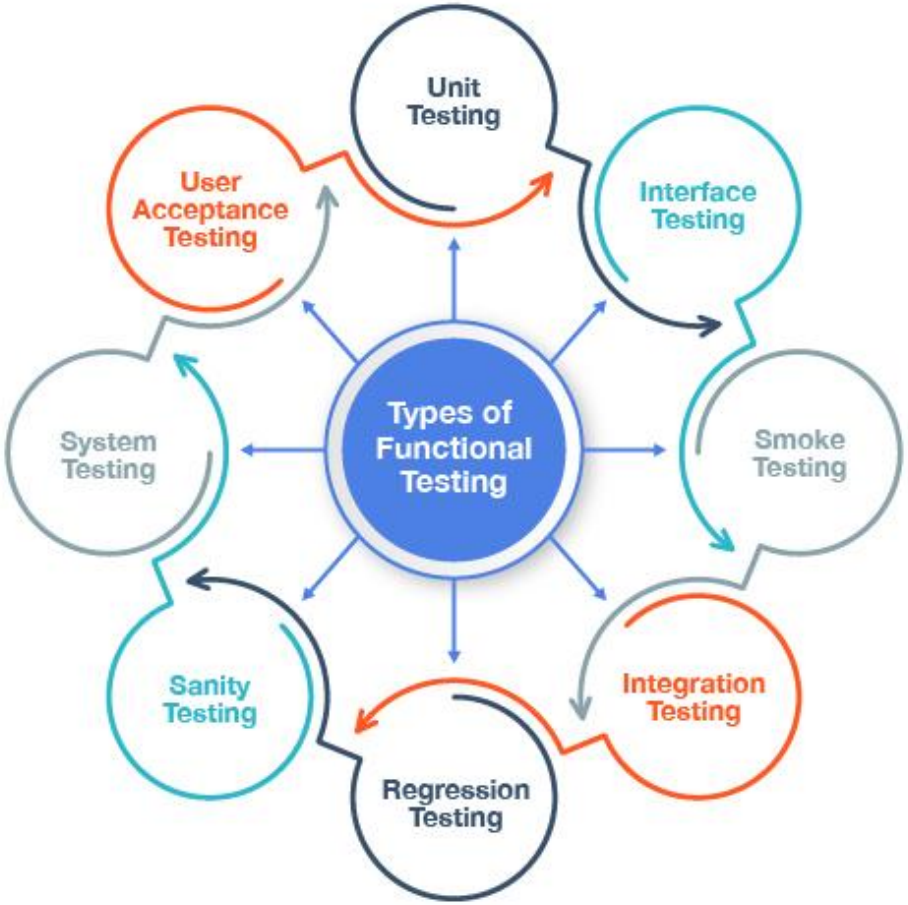
# CI/ CD BEST PRACTICES

- Self-study activity: Read on best practices here.

# SOFTWARE TESTING

- Software testing can be broadly categorized into two types.
  - Functional tests – verifies if functional requirements are met
  - Non-functional tests – verifies if non-functional requirements are met

# FUNCTIONAL TEST TYPES




Source: https://testsigma.com/blog/the-different-software-testing-types-explained/

# NON-FUNCTIONAL TEST TYPES


Source: https://testsigma.com/blog/the-different-software-testing-types-explained/

# TEST AUTOMATION

- In continuous testing, various types of tests are performed within the CI/CD pipeline. These can include:
  - **Unit testing**, which checks that individual units of code work as expected.
  - **Integration testing**, which verifies how different modules or services within an application work together.
  - **Regression testing**, which is performed after a bug is fixed to ensure that specific bug won't occur again.
- Self study activity: Read this article on Automated software testing.

# WRAP UP

- [YouTube] DevOps CI/CD Explained in 100 Seconds
- Try out GitHub Actions: a CI/ CD Platform on GitHub.

# SUMMARY

- Deploying an app
  - Typical Deployment Scenario
  - Failures
- DevOps
  - Core DevOps principles
  - Continuous Integration (CI)
  - Continuous Delivery and Continuous Deployment (CD)
  - Software Testing and Test Automation

# REFERENCES


1. What is CI/CD?
2. 4 Must-know DevOps principles
3. Manifesto for Agile Software Development
4. How Is Netflix SO GOOD at DevOps?
5. What Is Shift Left Testing?
6. Continuous integration vs. delivery vs. deployment
7. The different types of software testing

# THANK YOU

VISHAN.J@SLIIT.LK

NELUM.A@SLIIT.LK

# Insights from the Industry

Udam Liyanage
Software Engineer
Buckhill Software

# The Flow

1. Industry Overview
2. Busting Common Industry Myths
3. Creative Problem Solving
4. The Bright Side
5. Activity

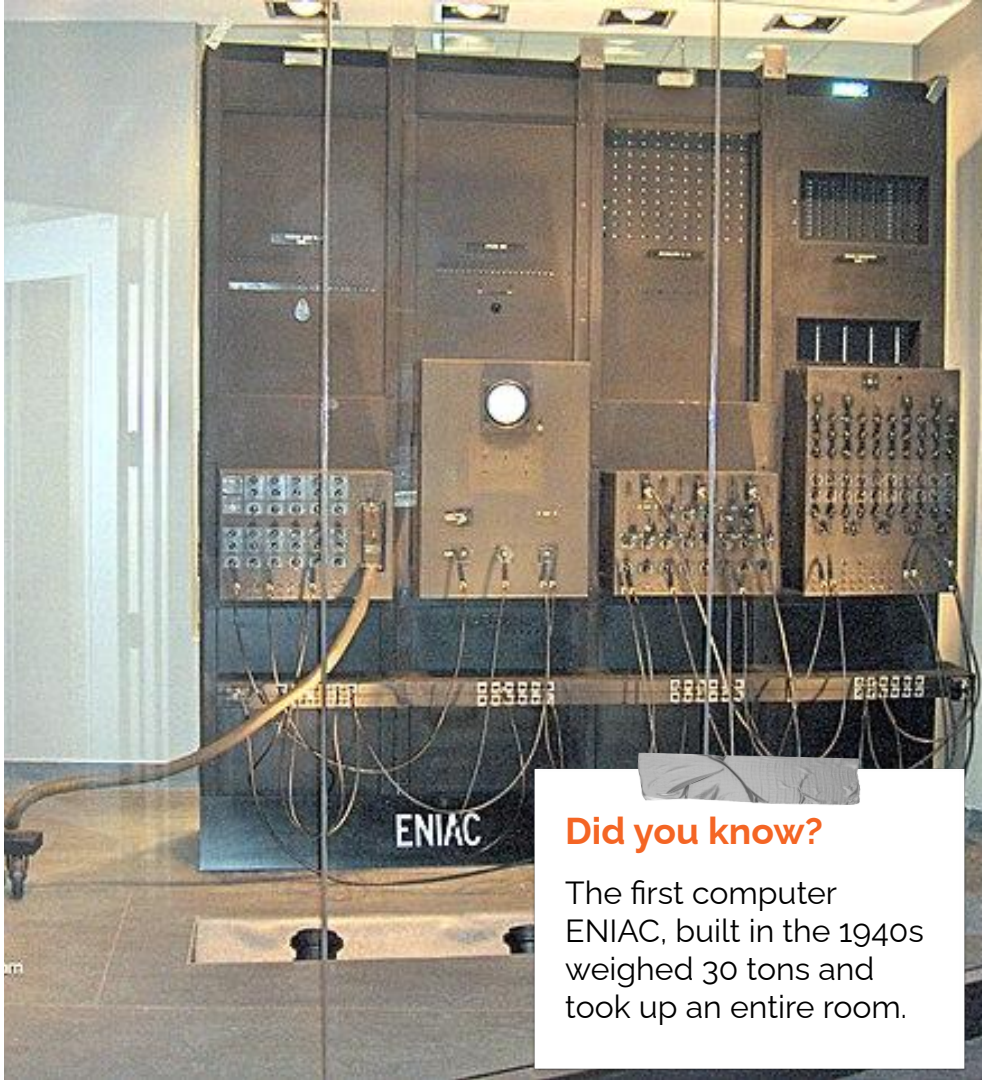
# 1. Industry Overview

- ➔ How IT came to be - an overview of the past 50 years
- ➔ Silicon Valley - Place or Mindset
- ➔ Simply Complicated - Rapid Miniaturisation
- ➔ PaaS and SaaS

# An Overview of the past 50 years

The IT industry has seen vast changes in the past 50 years. We came from a computer the size of a 2 storey house to a computer that can fit on our palms. With all the hardware advancements came the software advancements needed to run on that hardware. The industry saw rapid changes - some good and some bad in the past years. Some influential innovation of the past 50 years:

- ➔ Automated Teller Machine (ATMs)
- ➔ DNA Testing and Sequencing
- ➔ Fibre Optics
- ➔ Barcodes and Scanners
- ➔ GPS




**Did you know?**

The first computer ENIAC, built in the 1940s weighed 30 tons and took up an entire room.

Silicon Valley is not a place. It's a mindset. Good engineers can make any location Silicon Valley.

# Silicon Valley - Place or Mindset

Silicon Valley grew up in the area between San Jose, California, and San Francisco as a result of Frederick Terman, the legendary dean of Stanford engineering school during the 1940s and 1950s. The transistor was invented and manufactured in Silicon Valley, which gave the area a leg up in the radio and telegraph industries. ***Early engineers were not after money, they were about making things possible that had never been possible before, such as space travel.***

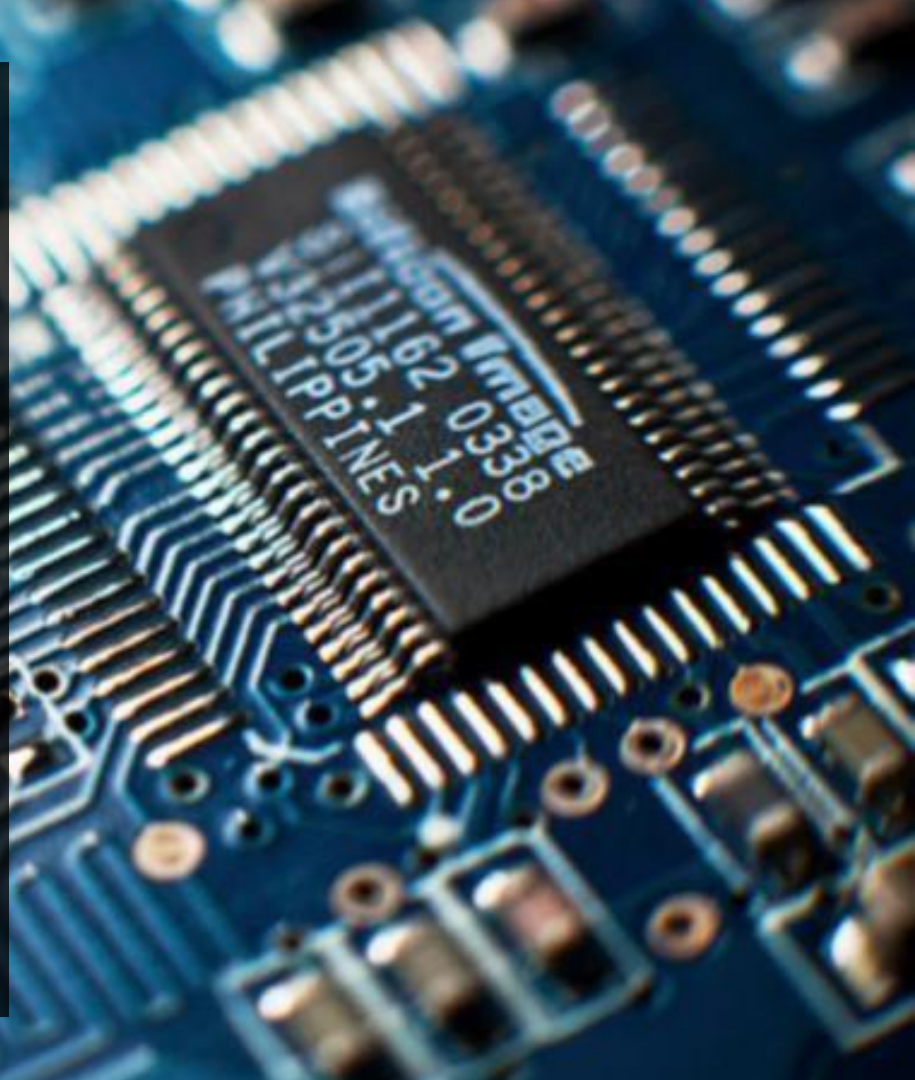
Silicon Valley turned from an orchard to the tech capital of the world in a couple of decades. What made all this possible? It was the people that made Silicon Valley what it is today.

# Simply Complicated - Rapid Miniaturisation

There's a rule of thumb in almost every engineering principle: big = more power. This is completely unrealistic in the IT industry. In Computing, we managed to make our devices more powerful while making the hardware that runs them smaller.

The continuous miniaturization of technology will among other things make technology more accessible to everyone. Ultimately the goal is to have computers everywhere by making components smaller and more powerful.

Technology created in the future will all have a few similar characteristics, they will be smaller, lightweight, durable, reliable, tamper-proof, and consume very little power.



# PaaS

- A complete development and deployment environment in the cloud
- The cloud services provider hosts, manages and maintains all the hardware and software included in the platform.
- Users access the PaaS through a graphical user interface (GUI).
- Development or DevOps teams can collaborate on all their work across the entire application lifecycle.
- You manage the applications and services you develop, and the cloud service provider typically manages everything else.
- E.g. AWS, Windows Azure, Google App Engine

# SaaS

- Is cloud-hosted, ready-to-use application software.
- Users pay a monthly or annual fee to use a complete application from within a web browser, desktop client or mobile app.
- The application and all of the infrastructure required to deliver it - servers, storage, networking, middleware, application software, data storage - are hosted and managed by the SaaS vendor.
- The vendor manages all upgrades and patches to the software, usually invisibly to customers.
- E.g. Netflix, Google Drive, Microsoft Office 365

# 2. Busting Common Industry Myths

- ➔ Industry Anti-patterns
- ➔ Glorification in the IT industry
- ➔ Good Leaders vs Bad Bosses
- ➔ Bureaucracies and Micromanagement
- ➔ Sprints and Workloads
- ➔ Neck on the line - The cutthroat approach to delivering software

# Industry Anti-patterns

Most major organizations today have embarked on transformation programs in response to changes in customer, competitive, and regulatory landscapes. Whether the transformations are labeled agile, digital, or DevOps, their fundamental premise is to build value by establishing short, iterative, and continuous feedback loops between product and customers that dramatically improve both the product and its time to market.

# Force-fitting technology solutions

Watch out when technology decisions do not attract business scrutiny beyond cost and a cursory discussion of "scalability/strategic alignment." For instance, we hear in many organizations about a "microservice-first" approach. While microservices are a critical component of many IT modernization journeys, they don't fit the bill in all circumstances.

At one major corporation, the architects of the transformation suggested an approach that built microservices for a *client-side* application. But microservices is fundamentally a *server-side* architecture. The architects were simply responding to an organizational push to become technologically modern.

Leaders need to raise their hands to ask "silly" questions to fully understand the rationale and purported benefit of the recommended technology choices.

# Adopting cutting-edge tech that's not fully mature

With stability and scalability as two core elements of any IT organization's focus, very careful due diligence and decision making are needed to avoid adopting technologies that haven't fully matured.

A leading bank launched a major redesign for its customer-facing application, using the latest web front-end framework as the software solution stack. It was touted as "future-proof" technology that would attract new talent. The project suffered serious setbacks and cost overruns because staff didn't have the right capabilities to support it, resulting in time and money to upskill them. It was finally delivered after two years—18 months behind schedule.

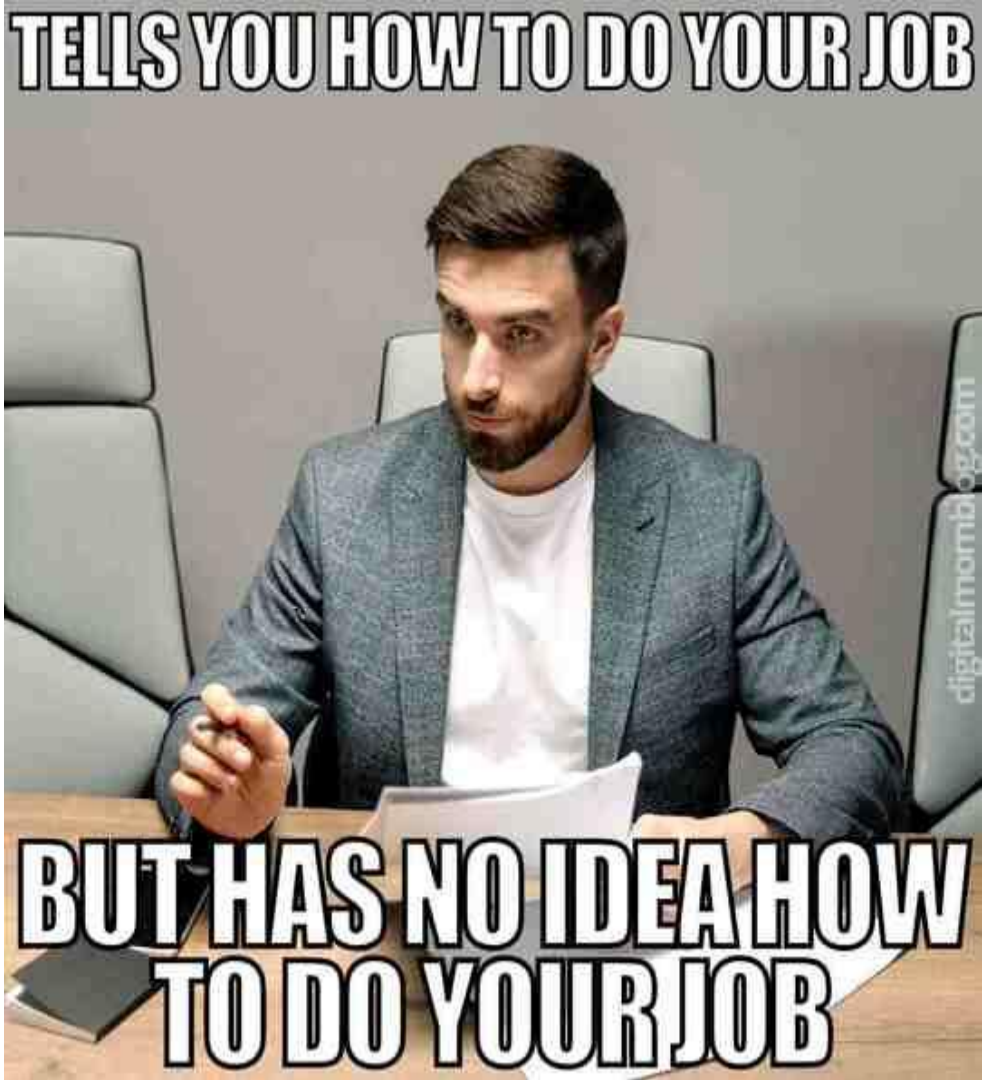
Choose simple, proven technologies with which your people are familiar.

# Building up an army of managers

Career growth in most organizations usually entails people management. Gradually, talented employees who once showed great technology promise spend more time on managing people and administrative activities than on practicing the craft of engineering. They become full-time managers. Over time, they lose the ability to engage in deep technical conversations with their teams, to role-model technical problem solving and innovation, and—most damaging—to effectively manage team performance based on detailed technical merit.

One major financial organization launched a program to overhaul its performance-management process and discovered that, on average, managers spent less than an hour a week in technical discussions with their teams.

Give technology managers specific responsibilities for tech delivery. Encourage a culture of tech expertise for technology managers.




# The Golden Hammer

Assume that a software development team has attained a high level of proficiency in a specific vendor product or solution, referred to as the "Golden Hammer" in this context. Because of this, every new product or development endeavour is seen as a problem that can be solved most effectively using that product. The Golden Hammer is often the wrong remedy for the issue, but little effort is made to consider other options. This AntiPattern leads to the incorrect use of a favourite tool or idea. As a result, developers and managers are content with an existing strategy and reluctance to learn and use a more effective one.

Always choose the right tool for the job. But keep disruptions at a minimum. Not every tool is meant to do every job. Fish cannot fly; birds cannot swim.

# Glorification in the IT Industry

Our generation is obsessed with work, busyness, and productivity. We look forward to downtime not because it gives us an opportunity to rejuvenate, but because it gives us more time to work toward passion projects, freelance gigs, and side hustles. And when we're not dedicating every waking minute to work and the pursuit of productivity, we feel guilty.

The obsession with long work hours rose out of the tech industry, where high output and productivity is a prized and cherished virtue. The workaholism culture has been perfected in companies like Google and Facebook, who keep their employees in the office and at their desks longer by offering outrageous perks and accommodations such as free food, massages, shuttles, nap zones, on-call doctors, happy hours, and on-site housing. And the idea of giving up nights and weekends to code software and build your own business is something that has been normalized (and even fetishized) in tech culture for several years now. It's the glorification of hard work — **the "hustle" or the "struggle"** — itself, rather than the goals and success the work aims to achieve.

Furthermore, working more hours does not necessarily mean you'll see the results you intended. Overworking leads to a state of chronic stress and mental fatigue, which impacts your ability to interact with others, communicate clearly, manage your emotions, and make decisions. So all the extra hours and sacrificing nights and weekends for work does not necessarily equate to more output and may actually have negative effects on productivity.

To alleviate this condition, we have to learn how to "switch off" from work from time to time. We have to be able to give ourselves uninterrupted downtime to recover from the stress and pressure of our work routine. Because while it may seem like more working hours would give us greater opportunity to work toward professional goals, in actuality proper rest, self-care, and personal well-being will allow us to work more effectively and efficiently.

# Good Leaders

- Influence - Authority comes from their ability to influence others.
- Listen - Supports and guides through a task.
- Mentor - understand that employees benefit from encouragement and mentorship
- Delegate Authority - skilled at results by enabling their team to figure out what to do
- Part of the team - let go of this hierarchical distinction and view their team members as equal contributors.

# Bad Bosses

- Command - Authority comes from their position.
- Explain - Ensures the work is understood and leaves it in the subordinates hand.
- Discipline - more likely to use a reward-and-punishment system
- Delegate Tasks - gets results by telling people what to do and is concerned with doing it right.
- Above the team - view their team members as subordinates

# Bureaucracies and Micromanagement

Bureaucracies and micromanagement happen at different levels in different companies. Some companies glorify and endorse this. Bureaucracies and micromanagement are two sides of the same coin. Both, hinders progress, makes employees unhappy, and threatens entire businesses to close down. Every industry, from retail to financial services to healthcare, is experiencing shifts and disruptions that outpace the capability of the traditional org chart. In simplest terms, that's because the traditional hierarchy was designed to solve the predictable problems of a complicated world, not the unpredictable problems of a complex world.

## Silos

While teams and functional departments may excel at their piece of the problem, siloing prevents true teamwork at the enterprise level. Complex problems require the very cross-functional relationships that silos prevent, while tribalism, geographic dispersion, and a lack of face-to-face communication exacerbate the issue.

## Lack of alignment

Alignment across large organizations is difficult to maintain. Senior executives have competing visions for the way forward. Functional groups often work at cross-purposes, unaware of what other teams are executing on. Employees lower down in the organization often lack awareness of the company's objectives and strategies, and are unsure how their contributions impact the organization at large.

## Lack of adaptable leadership

Although the world has changed dramatically, many leaders are still managing the same way. Micromanaging and mastering all the details is unrealistic and ineffective. Leaders today need to focus on fostering teamwork, collaboration, and communication

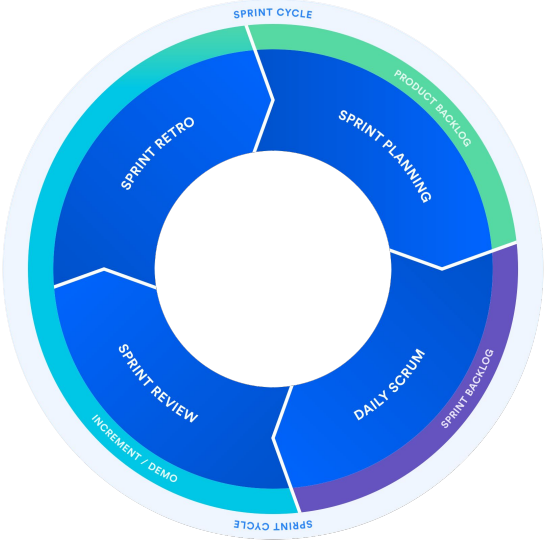
## Antiquated decision-making

The information-up, decision-down cycle of a traditional bureaucracy cannot compete with the speed and interconnectedness of the modern environment. Creating a system based on inclusion and transparency was critical to this endeavor.

# Sprints and Workload

A sprint is a short, time-boxed period when a scrum team works to complete a set amount of work. With scrum, a product is built in a series of iterations called sprints that break down big, complex projects into bite-sized pieces. A sprint can be 1 to 4 weeks. The most common sprint duration is 2 weeks. The workload of every developer/engineer is calculated and allocated based on a concept called **story points**. Also with this, the team shall identify priorities and start tasks in that order.

- ➔ **Sprint planning** is where the upcoming sprint is planned.
- ➔ During a sprint, the team checks in during the **daily scrum**, also called the **standup** meeting, about how the work is progressing. The scrum master/PMs usually join this meeting. Note that this is a very short meeting - approx 15 mins.
- ➔ After a sprint, the team demonstrates what they've completed during the **sprint review**. This stage will be skipped based on the company.
- ➔ Finally, the **sprint retrospective** is held for the team to identify areas of improvement.



# Neck on the line - A cutthroat approach to delivering software

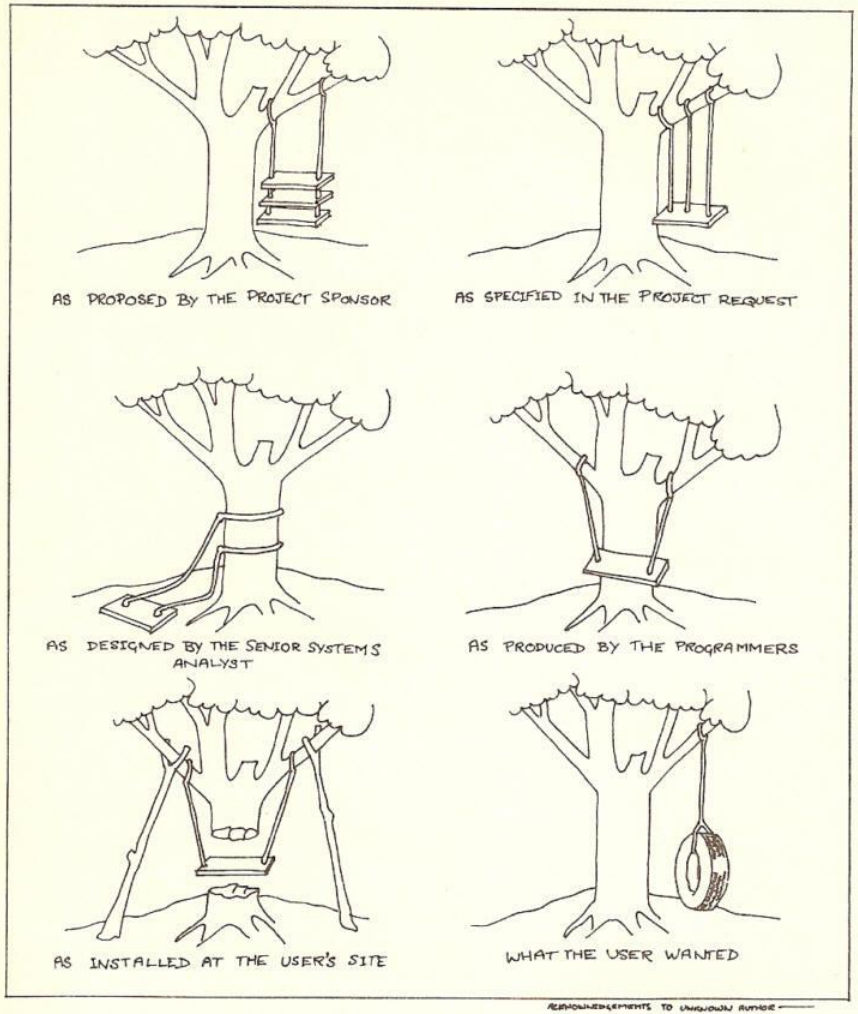
As discussed earlier, the software industry is very big on unhealthy practices. Delivery time is favored above the quality and the efficiency of the developers. What we have to understand is that missing the deadline of a project is fine as long as there is a justification for it. Not all tasks go as planned, you may estimate a task to take 2 days but ends up spending 4 days on it, which reduces the capacity you have to work on the other tasks. This does not mean that you have spend sleepless nights trying to finish that unless of course the company wants it urgently, in which case they will support you by giving you support and more resources. But most of the time, chances are that the end-user (client) and the company will understand why something could not be done.

Your peak performance comes when you are well rested. If you keep working through nights, you will eventually get tunnel vision and try to solve problems with zero creativity and - just for the sake of solving problem. Overworking results in burning out and losing interest in work.

Although it seems boring from the outside, being a software engineer can be a very exciting job. You, like any other engineer, will spend your day trying to tackle complex problems and coming up with creative solutions for them. If you're ever in a workplace where the cutthroat approach to delivering software is glorified: think twice.

# 3. Creative Problem Solving

- ➔ Technical Requirements
- ➔ Technical Decisions
- ➔ Tech Stacks
- ➔ Discussion - Vague technical requirements and how to use creative problem solving to workaround obstacles




From the *University of London Computer Centre Newsletter* No. 53, March 1973

# Technical Requirements

Technical requirements, in the context of software development and systems engineering, are the factors required to deliver a desired function or behavior from a system to satisfy a user's standards and needs.

Setting clear technical requirements is an essential step in the software and system development process.

When working on a project or creating software, technical requirements describe the technical aspects and issues that you need to address for the project or software to work and execute successfully.

Technical requirements are important because they describe how software should function and what its behavior should be. This helps developers and users to understand the best way to use the software. A document of clearly defined specifications helps to create a project or software that has a proper process for implementation. Developers and other technicians refer to this as technical requirement documentation.

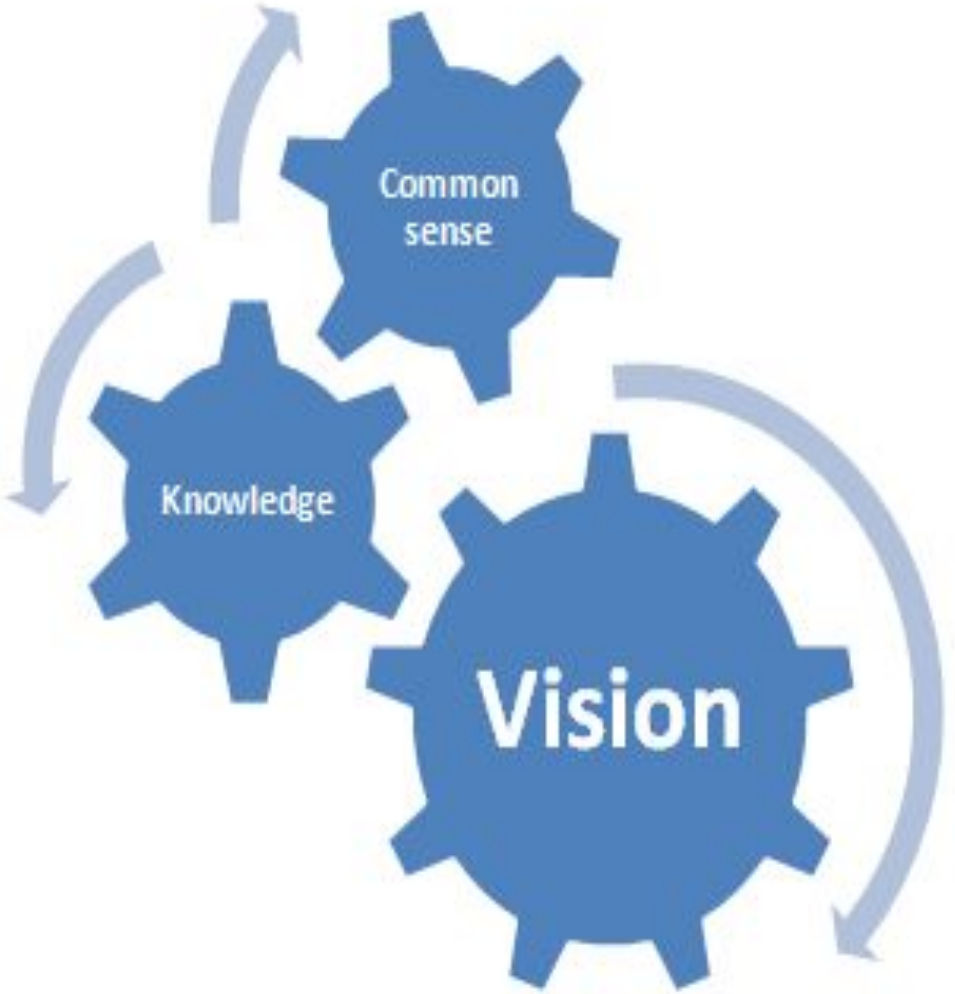
# Technical Decisions

Every technology-driven business, no matter the size, needs the right technical and architectural experience to choose their technology stack effectively and balance the present and the future of their software product. From a developer's perspective, **choosing the right technology stack is a factor of success** for every software project.

The way you build your tech stack has much influence on your company, product, and development team. There's **plenty to consider where technology adoption is concerned**. Each technology stack has its advantages and disadvantages, so it's important to make sure to **select the right tools that could help you the most with your project**.

In the end, **everything you choose for your project has to fit your specific needs and all of the elements must work well together**.Pairing the right technologies together is one of the most critical decisions because it could make all the difference when it comes to increasing your business's productivity and success.

By making the right choice, you help your **developers accomplish more in less time**.



# Tech Stacks

A tech stack is the set of technologies used to develop an application, including programming languages, frameworks, databases, front-end and back-end tools, and APIs. Investing in the various elements of your tech stack is a crucial step to finding success as a software company, since your tech stack gives your product team the tools it needs to build and maintain your product, and to make sure it continues to meet customer needs. Before the days of ubiquitous SaaS products and services, tech stacks were relatively simple: there was LAMP (Linux, Apache, MySQL, PHP), an older standard for building PHP-based web applications, and non-open source alternatives like WAMP (for those that preferred Windows to Linux). Some basic categories of a tech stack are:

- **Operating systems and programming languages** - You'll choose these based on the environment you're most comfortable developing in as well as the type of application you want to optimize for. You may end up with several, depending on how you want to build the backend and the user experience, and what devices you're building for.
- **Servers and load balancing** - This category include servers, content distribution networks, routing, and caching services that let your applications send and receive requests, run smoothly, and scale capacity as needed.
- **Data storage and querying** - This layer of the stack consists of relational and non-relational databases, data warehouses, and data pipelines that allow you to store and query all of your real-time and historical data.
- **Backend Frameworks** - Frameworks often include some of the basic functionality you'll need to build an app, and provides structure for things like organizing and communicating with your database, handling requests from users, and sending out registration or password reset emails.
- **Frontend Frameworks** - The services and frameworks you use to build the user experience, including the user interface and all the client-side functionality in your product.
- **API Services** - The applications that help you connect to the tools that make up your extended tech stack.

# Why is getting your tech stack right so crucial?

The way you build your tech stack influences much about your company: what kind of products you'll be able to build, how efficiently you'll be able to work, and even what type of engineers you'll hire. The process always involves trade-offs—some technologies save time but allow for less customization, others are better for certain audience segments (iPhone users, say), still others are more scalable but require more ongoing maintenance. However, it is possible to assemble a stack that can both meet your needs now and evolve as your company matures. When possible, start building with tools that can scale as you grow. Many well-known backend solutions, like AWS, give you the option automatically add additional servers as you need them rather than having to estimate usage and pay for capacity upfront. The decisions regarding your future tech stack are also going to influence:

- ➔ Your system's performance
- ➔ How easy it is to scale
- ➔ TTM - Time To Market
- ➔ The ability to add the right skilled people to your development team
- ➔ Cost of maintenance of the product
- ➔ How users will interact with you application
- ➔ How easily you can make changes to your project as it evolves.

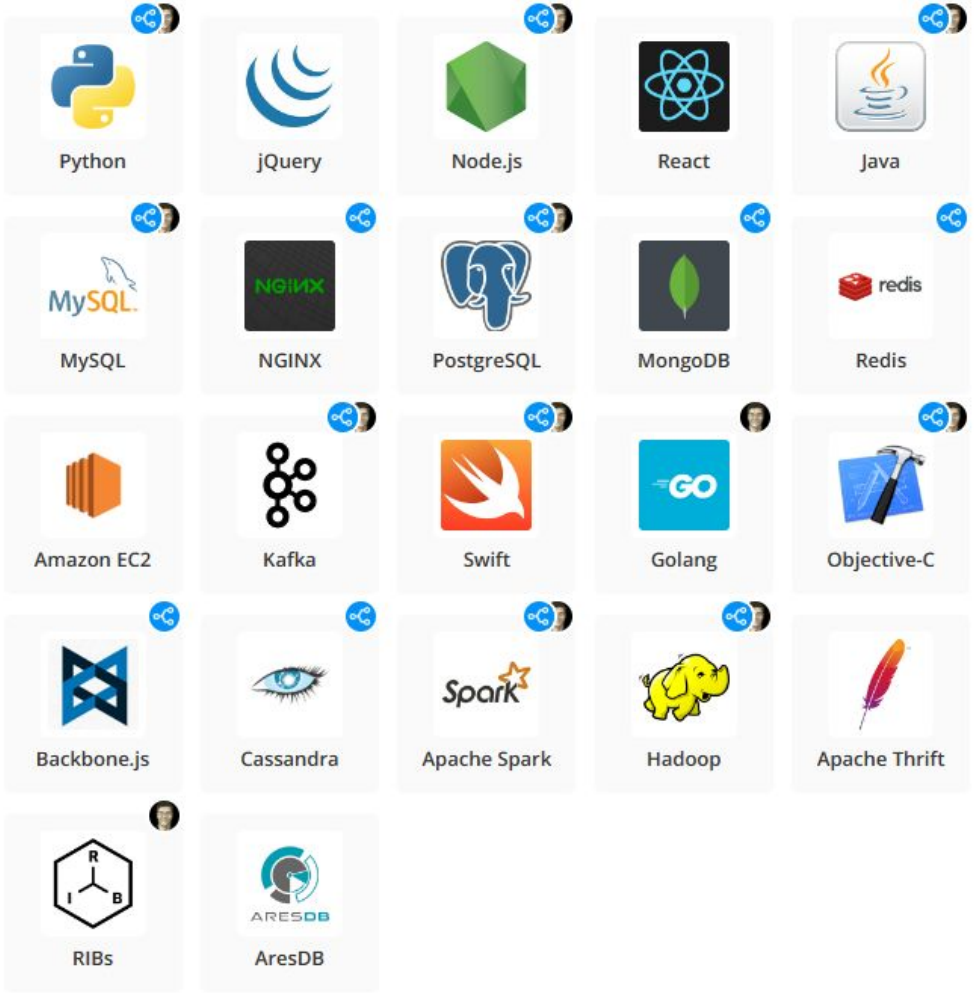
# Case Study - Uber

Let's take a look at Uber, one of the fastest-growing companies in Silicon Valley history.
There are many competitors in the taxi industry, but the company has brought a revolution in car services all across the world. Uber's use case proves that effective personalisation and relevant customer engagement strategies work. **Data is very important for Uber**. They build scalable streaming pipelines for near-real-time features. Their pipelines are fed with data about all rides, car requests, they even track drivers going through the city without passengers to know the traffic patterns.
On the product front, Uber's data team is behind all the predictive models powering the ride sharing cab service, making the company the leader in the industry. Uber transforms and scales faster by leveraging the right technology to offer a unique customer experience.

# Uber - Application and Data Stack

It can be seen that Uber uses multiple programming languages, frameworks, database technologies and big data tools in their tech stack.

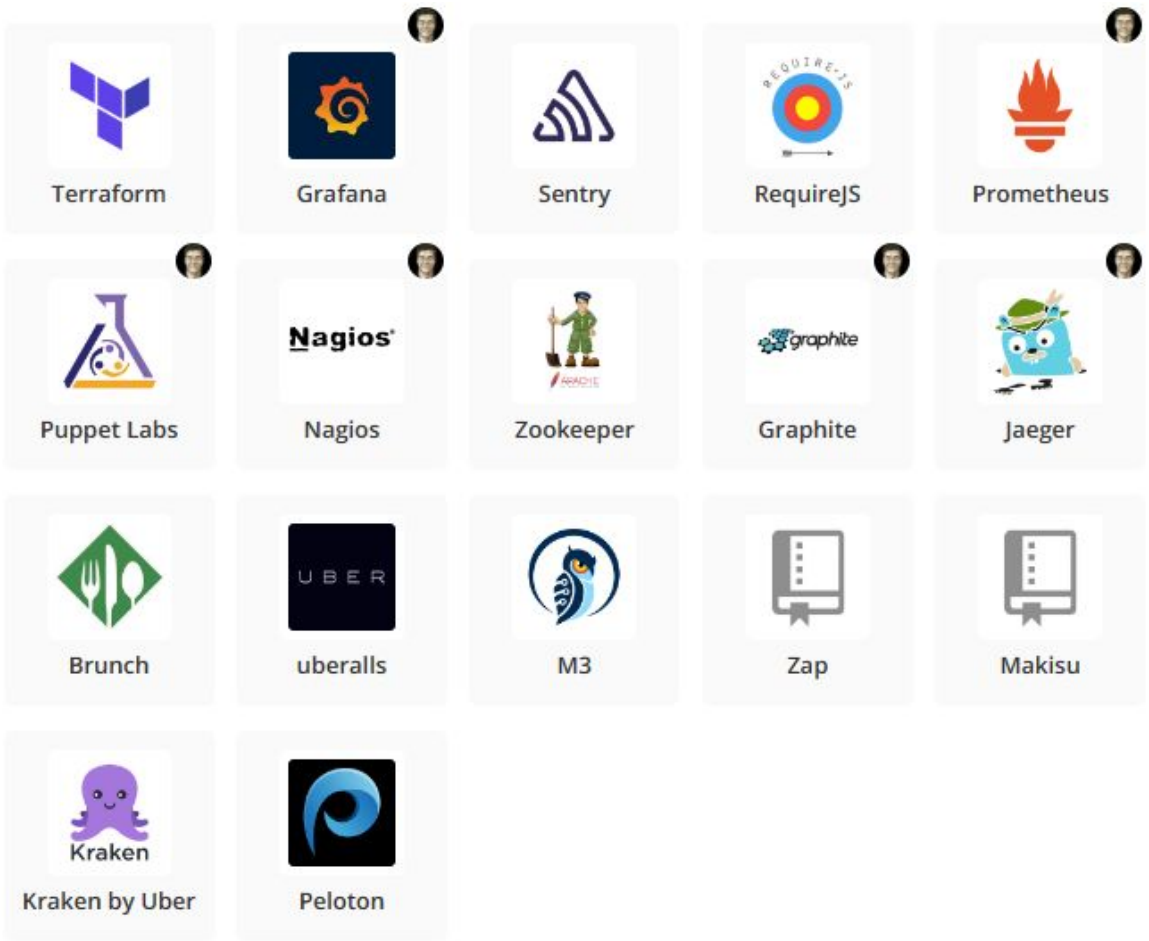
This is a very good example of a mature tech stack because they have not forced the engineers to use the golden hammer to solve the problems. They have used the ***right*** tool to get the job done rather than force-fitting technology solutions.



# Uber - DevOps Stack

Uber uses a range of DevOps tools to make their global DevOps strategy resilient.

Sometimes, Uber does not find the tool to do the job and their build their tools internally, this is very good technical decision because they're not trying to use an existing tool to do something it's not meant to do. Rather, when need be, they create a tool for that. A good example is Jaeger, which is a distributed tracing system Uber built to track requests across distributed systems.



# Vague technical requirements and how to use creative problem solving to workaround obstacles.

A software engineers daily life is filled with unknowns. Sometimes, when the specifications are defined very cleary, the job is easy. But this is rarely the case. Most of the time, the technical requirements are very vague and force you to think outside the box.

Technical Requirement

“There’s a portion in the frontend application that allows the user to upload an excel sheet of address data. The application should Geocode these addresses for an internal geospatial calculation. Google Maps Geocoding API is used to geocode the addresses. The geocodes are sent to the backend for the geospatial calculation. The average number of locations per Excel sheet is 350. Right now, the whole Geocoding process takes around 4 minutes to complete and this is big hit to the user experience of the system. This should be remodeled to reduce latency. End goal is to geocode 500 addresses under 10 seconds. ”

The above technical requirement is vague. It tells you what the problem is and what the end goal should be. How you get there, is up to you. And you have to make the proper technical decisions, taking into account the trade-offs and achieve the end goal. Keep in mind, that this too should be achieved within the estimated time frame and budget.

# Solving the Problem

There are a couple of steps to follow to tackle this problem:

1. **Identify the core problem** - The Geocoding as it is right now is slow and the geocoding is done by third-party API, the latency of which we cannot control.
2. **Investigate why the core problem is happening** - It was found that Google takes approximately 450ms to respond with a geocode. So, 450ms * 500 = 225000ms which converts to 3.75 minutes. This is taking time because the requests to the Google API is made sequentially. If we make the API calls parallel this problem would be solved and we can ideally geocode 500 addresses in under 1 second.
3. **Choose the technologies to use** - The main stack is PHP but PHP does not support concurrency out of the box, We need a language that supports concurrency as a first-class citizen. We decided to go with Golang because it is lightweight, easy to deploy and manage while providing unmatched performance.
4. **Build the first PoC (Proof Of Concept)** - Build a proof of concept for the suggested solutions. This is when you start to see holes in your concept. The problem we had was that Google was rate-limiting the API requests at 50 requests per second. So now we could not parallelly call the Geocoding API with 500 locations.
5. **Workaround the obstacle** - Now we had a problem we never anticipated in the first place, but that's part of the job. Again, we identify the problem, and try to think out of the box to solve. Solution we came up with was to use multiple Geocoding API keys and bind the API keys to the thread at runtime. No thread is allowed to surpass the hard limit of 50 requests per second. When the thread completes, the Geocoding API key is released to the pool.
6. **Final solution** - With 1 service (API key) we were able to geocode 50 locations in ~1.4s (network latency + json encoding/decoding). With 5 API keys - 50*5 = 250 requests/1.4s. Therefore, 1.4s*2 = 2.8s/500 requests. Sub 3 seconds for 500 locations. Expected was 10 seconds.

# 4. The Bright Side

## It's not as bad as it sounds

Let's face it, the industry is not as bad as it sounds. True, there are some parts of the industry that could really promote good habits. But as with all things, good comes with a little bad. We are working in an industry that has given the world so much in the past few decades alone and continues to do so at a rapid pace. This can one of the most exciting and promising industries one can work in.  We change the world everyday.

## It's okay to break things

It's perfectly okay to break things; it happens when you move forward. There are no perfect engineers who never made mistake in their lives. There are engineers who made mistakes and learned from them. If you make a mistake, acknowledge, rectify and continue. It's not the end of the world. Everything can be repaired back to how it was.

## 'Perfect' technical decisions - They don't exist

Don't worry about making perfect technical decisions. They don't exist. Rather aim to make adaptable and extensible decisions. With the rate at which technology is expanding, new technologies are outdated pretty soon. Therefore, perfect technical decisions do not exist. We should aim to make technical decisions that can be refactored and upgraded in the future.

## Pitfalls of future-proofing

Over-engineering and future-proofing are two pitfalls for any engineering tool. If a tool can be built as a one-off, don't worry about over-engineering the solution. Follow the KISS principle - Keep It Simple Stupid. Also, don't waste too much time on future-proofing your solutions. Chances are they will be outdated in a couple of months anyway.

# Good luck!

All the best with your education and internships.

Hope to see you all creating ripples in the industry that we all call home.

Cheers!

I'm always reachable here - udamsliyanage@gmail.com