PAGE No.
DATE / /

Module No-3

Monolithic Scenario

- Imagine you have massive web application with various functions including a static website embedded within in. The entire web application is deployed as a single Java EE application archive. So when we need to fix a spelling mistake in the static website, we need to rebuild the entire web application archive and deploy it again.
- Instead of that, firstly we have to figure out which part of the colossal code base is responsible. So you create a branch in your version control system, corresponding to the code currently running in production. Then, you make the correction, build a new artifact, slap on a shiny new version, and finally deploy it.
- Since it's all bundled up, any change in the monolith affects the entire business critical system. If something goes wrong during deployment, revenue may be lost rapidly. The complexity of the monolith makes it difficult to guarantee that the change only affects the intended component.
- Additionally, seemingly harmless changes like modifying version strings during artifact production introduce uncertainties.
- Due to the intricacies of the monolithic system, comprehensive testing and manual verification become essential for every change, adding to the complexity of the deployment chain.
- In summary, monolithic architecture complicates continuous delivery process, demanding careful considerations of potential repercussions and thorough testing for every minor modifications.

PAGE No
DATE / /

Architecture rules of thumb

- Architecture rules of thumb are guiding principles that help in designing and organizing software systems effectively.
- They provide a set of guidelines to address challenges and avoid undesirable situations, such as those presented in the earlier scenario.

i) Separation of Concern -

- Introduced by the renowned Dutch scientist Edsger Dij
- This principle suggests that different aspects of a system should be considered separately, allowing for more efficient organization of thought.
- It serves as a fundamental rule in software desi influencing other rules and guiding the structuring of systems.
- It is one of the crucial rules in software desig

ii) The principle of cohesion -

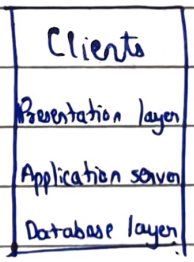
- Cohesion measures the degree to which elements within a software module being together.
- Strong cohesion indicates a strong relationship betwee functions in a module, aligns with separation of concerns.
- It emphasizes the importance of keeping related functionalities closely connected within a module.

ii) Coupling -

- Coupling means degree of dependency between two Modules.
- Low coupling is desirable, as it minimizes the interdependence of modules.
- This rule too connect with the separation of concerns, promoting a design where modules are loosely connected, making the system more flexible and easier to maintain.

★ Three tier system:

- Matangle customer database is a very typical three tier system, CRUD (Create, Read, Update, Delete) type of system.
- These types of systems are common and has been in use for decades.



i) Presentation layer -

- This tier represents the user interface and is implemented using the React web framework.
- Deployment involves distributing a set of Javascript and static HTML files.
- The React framework, can be substituted with other frameworks like Angular based on organizational preferences.

- Despite different frameworks, the deployment and build processes for most Javascript framework are similar, providing flexibility in technology choices.

ii) Logic Tier -

- Logic Tier manages the business logic and is implemented using the Clojure language on the Java platform
- The choice of the Java platform is common in large organizations, while smaller ones may opt for platforms based on Ruby or Python.
- Our example, based on Clojure, contains a little bit of both worlds.

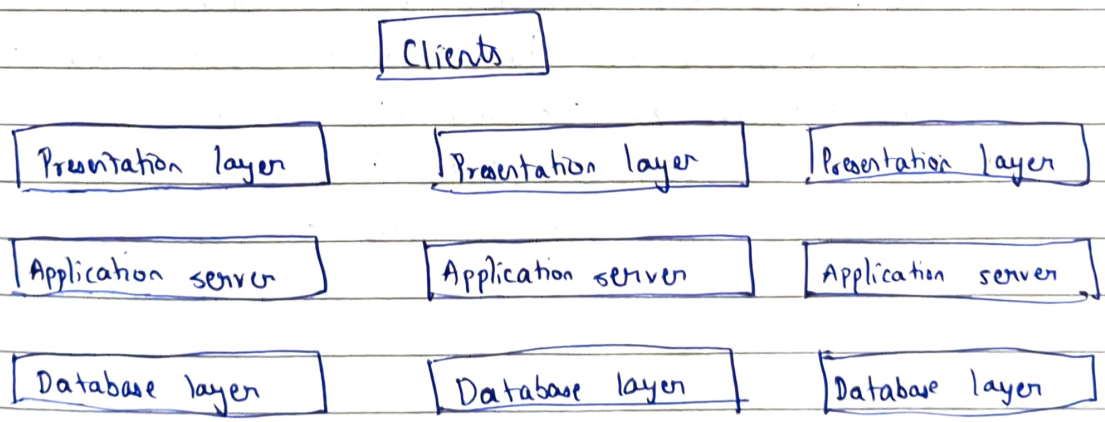
iii) Data Tier -

- The data tier involves the database, implemented using the PostgreSQL, relational database management system.
- PostgreSQL is chosen for its robustness, even though larger enterprises might prefer alternatives like Oracle databases.
- The database is responsible for storing and retrieving data, supporting the CRUD operations.

PAGE No.
DATE / /

Microservices

- Microservices has emerged to describe systems that differ from the traditional three-tier pattern in the composition of the logic tier.
- In a microservices architecture, the logic tier is comprised of several smaller services that communicates using language agnostic protocols.
- While HTTP based protocols, often using JSON REST, are common, they are not the only option, but there are various possibilities exists for protocol layer.
- This architectural design patterns aligns with a Continuous Delivery approach.
- Unlike monolithic systems, where deploying changes can be complex, microservices makes it easier to deploy a set of smaller, standalone services independently.
- The modularity and independence of microservices facilitate a more seamless and continuous delivery of updates, making it an attractive choice for modern software development.



PAGE No.
DATE / /

Interlude - Conway's Law

- The structure of an organization that designs software winds up copied in the organization of the software. This is called as Conway's Law
- For instance, the three-tier pattern commonly aligns with how many IT departments are organized:
  i) Database Administrator (DBA) team
  ii) Backend developer team
  iii) Frontend developer team
  iv) Operations team

We can see the clear resemblance between the architecture pattern and the organization.

- In context of DevOps, the primary aim is to bridge different roles, ideally bringing them together in cross functional teams.
- Conway's Law suggests that the organization's structure is mirrored in the software design.
- Therefore, when different roles collaborate in cross functional teams, this collaborative structure tends to reflect in the software they create.
- The microservices pattern happens to mirror a cross-functional team quite closely

PAGE NO
DATE / /

DevOps, architecture, and resilience

- In the context of DevOps and software architecture, the microservice offers a significant advantages in accelerating the deployment of new features to users. However, it also introduces challenges, particularly regarding the system's reliability and resilience.
- Microservices, due to their higher number of integration points and increased complexity, possess a higher probability of failure compared to monolithic systems.
- Automated testing is crucial in ensuring the quality of deployed changes; it doesn't entirely resolve the risk of services suddenly failing for various reasons.
- With multiple running services in a microservice pattern, the statistical likelihood of a service failure increases.
- To mitigate such risks, effective monitoring and automated response mechanism become essential. A strategy to ensure resilience in the face of potential service ~~could~~ failure could involve:-

i) Employing multiple application servers running the application.
ii) Offering a specialized monitoring interface via JSON REST.
iii) Implementing a monitoring daemon that periodically checks this interface.
iv) Configuring a load balancer to redirect traffic away from a failed server.

This strategy serves as a basic example, illustrating the challenges in designing resilient systems comprising multiple interconnected parts.

- The need for an application specific monitoring interface arises from the limitations of generic system monitoring.
- While standard monitoring covers basic system health metrics like CPU load, disk space, server status, it might not provide insights into the specific functionality of individual services.
- For instance, a service might fail due to a broken database access configuration, which generic monitoring might not detect. Therefore, a service specific health probing interface helps assess the proper functioning of individual services by attempting to interact with their dependencies, such as contacting the database and reporting the status of the connection.
- It is best if organization can agree on a common format for the probing return structure. The structure will also depend on the type of monitoring software used.

★ The need for source code control.

- Terence Mckenna says that everything is code, some may agree or not, but in DevOps a lot can be indeed be expressed in coded form.
- Applications, infrastructure, documentation, hardware functionalities can be represented as software.
- Given the fundamental role of code in DevOps, the source code repository plays a central role in an organization, as nearly everything produced in the software development and deployment process passes through it at some stage in its lifecycle.

Source Code Management System:

- Source code management system, crucial in development continue to evolve.
- Currently Git is widely used source code management system that was initially created to facilitate the development of the Linux kernel.
- It gained prominence when the license for the previous SCM tool, Bitkeeper changed making unsuitable for the Linux kernel development.
- Git was designed to support the complex workflow of the Linux Kernel project and is technically proficient for most organizations.
- Git has one of the key advantage compared to other SCM system, that is distributed version control system (DVCS).
- It means that it allows developers to work efficiently even when not connected to a network.
- Git doesn't require constant server connectivity for operations, making it faster in many scenarios.
- It also enables developers to work privately until are ready to share their work, and it supports multiple remote logins, reducing the risk of a single point failure.

Other distributed version control systems apart from Git:-

i) Bazaar - Supported by Canonical, company behind Ubuntu. Launchpad is a Canonical's code hosting service supports Bazaar.

ii) Mercurial - Notable open source projects such as Firefox and OpenJDK use Mercurial.

- Git is somewhat complex, compensates for this complexity with its speed and efficiency.
- It may be challenging to understand at first, there are user-friendly frontends available that simplify its use and facilitate various development tasks.

☆ Source Code Management System Migrations

- When migrating from one source code management system to another, maintaining the entire history of code changes is a common challenge.
- For certain projects, especially open source projects, preserving the data is crucial and worth the time and effort.
- However, for many organizations, the resources required to retain complete historical data might not be justifiable. In such cases, keeping the old source code management system accessible online for reference when needed might be a pragmatic approach. Eg:- VisualSource Safe and ClearCase.
- Not all migrations entail such extensive efforts. Some transitions, like moving from Subversion to Git is straightforward and do not necessitate sacrificing historical accuracy.
- In these cases, the migrations process can be more streamlined, and the complete history of changes can be retained without significant additional effort.