LEC 01

Short Note: Software Frameworks

Programming vs. Software Engineering

- **Programming**: Writing code to create applications.
- **Software Engineering**: Broader process involving requirements analysis, design, QA, deployment, maintenance, and adherence to best practices.

Generic Functionalities in Software Development

- Common features (e.g., authentication, database connectivity) are reused across applications.
- Frameworks avoid reinventing the wheel for these functionalities.

What is a Framework?

- A reusable architecture providing tools, libraries, and best practices to accelerate development.
- Key characteristics:
    - **Inversion of Control**: Framework manages flow, not the developer.
    - **Extensibility**: Customize via user code without modifying core framework.

Framework vs. Libraries

- **Framework**: Dictates structure, controls flow (e.g., Spring Boot).
- **Library**: Provides specific functions called by the developer (e.g., jQuery).

Why Use Frameworks?

- Speed up development.
- Enforce standards (security, coding conventions).
- Reduce boilerplate code.
- Community and tooling support.

Advantages

- Code reusability, easier debugging, optimized processes.
- Compliance with security standards.

- Faster development with less code.

Limitations

- Learning curve.
- Design constraints and limited customization.
- Potential bloat from unused features.
- Requires staying updated with framework changes.

Examples

- **Backend**: Spring Boot (Java), Django (Python), Express.js (Node.js).
- **Frontend**: Angular (TypeScript), Vue.js (JavaScript).

Key Takeaway: Frameworks provide a structured foundation to focus on unique requirements, enhancing efficiency and quality in software engineering.

LEC 02

Short Note: Version Controlling & Git

What is Version Control?

- System to track file changes over time, enabling recall of specific versions.
- **Manual Methods**: Renaming files (e.g., numbers1.py, numbers2.py), cloud storage, log files.
    - **Issues**: Overwriting risks, collaboration complexity, lack of automation.

Evolution of Version Control Systems (VCS)

1. **Local VCS** (e.g., SCCS): Tracked individual files; no collaboration.
2. **Centralized VCS**: Client-server model with a central repository (network-dependent).
3. **Distributed VCS** (e.g., **Git**): Peer-to-peer; full local repositories, enabling offline work.

Git Basics

- Created by **Linus Torvalds** (2005) for Linux kernel development.

- **Key Features**:
    - **Distributed**: Every user has a full local repository.
    - **Non-linear Development**: Supports branching/merging.
    - **Commits**: Snapshots with unique SHA-1 hash IDs.
    - **Remote Repositories** (e.g., GitHub): Centralized "source of truth" for collaboration.

Core Concepts

- **Local Repository**: User-owned copy with full history.
- **Remote Repository**: Shared central location (e.g., GitHub).
- **Basic Commands**:
    - git add: Stage changes.
    - git commit: Save changes locally.
    - git push: Send local commits to remote.
    - git pull: Fetch updates from remote.
    - git merge: Combine branches.

Advantages Over Manual Methods

- **Automation**: Revert, compare, merge, track changes.
- **Collaboration**: Avoid overwrites with structured workflows.
- **History Tracking**: Logs with commit details (author, timestamp).

Git ≠ GitHub

- **Git**: Tool for version control.
- **GitHub**: Platform hosting remote repositories.

Key Takeaway: Git streamlines version control by automating tracking, collaboration, and history management, replacing error-prone manual methods.

LEC 03

Short Note: Git Branching & Workflows

Git Branching Basics

- **Purpose**: Enable isolated development (features, fixes) without disrupting the main codebase.
    - Avoid merge conflicts in collaborative work.
    - Example: Separate branches for features (Amith's feature-A, Binesh's feature-B).
- **Commands**:
    - git branch <name>: Create a branch.
    - git checkout -b <name>: Create + switch to a branch.

Git Workflows

1. **GitFlow**:
    - **Structure**: Multiple branches (main, develop, feature, release, hotfix).
    - **Use Case**: Enterprise projects with strict release cycles.
    - **Complexity**: Overkill for small teams.
2. **GitHub Flow**:
    - **Steps**:
        1. Create a feature branch from main.
        2. Develop, commit, and push changes.
        3. Create a **Pull Request (PR)** for review.
        4. Merge to main after approval.
        5. Delete the feature branch.
    - **Advantage**: Simplicity, ideal for continuous deployment.

Pull Requests (PRs)

- **Purpose**: Propose changes for team review before merging to main.
- Ensures code quality and collaboration.

Best Practices

- **Branching**:
    - Use short-lived, feature-specific branches.
    - Follow naming conventions (e.g., feature/user/task-name).
    - Delete merged branches.
- **Commits**:
    - Atomic, frequent commits.
    - Clear commit messages (e.g., "Fix login authentication bug").
- **General**:
    - Pull latest main before branching.
    - Push often (even unfinished work for backup).
    - Avoid direct commits to main.

Key Takeaway

Git workflows like GitHub Flow streamline collaboration by isolating changes, enabling reviews (PRs), and maintaining a stable main branch. Adopting best practices ensures efficient version control and minimizes conflicts.

LEC 04

Short Note: Web Application Architecture

Website vs. Web Application

- **Website**: Static content for consumption (e.g., blogs, portfolios).
- **Web Application**: Interactive, dynamic, and user-driven (e.g., e-commerce, social media).

Three-Tier Architecture

1. **Presentation Layer (Frontend)**:
    - **Technologies**: HTML/CSS/JavaScript, React, Angular, Vue.js.
    - **Architectural Patterns**:
        - **MVC** (Model-View-Controller): Separates data (Model), UI (View), and logic (Controller).
        - **MVVM** (Model-View-ViewModel): Uses data binding to sync View and Model.
2. **Application Layer (Backend)**:
    - **Technologies**: Java/Spring Boot, Node.js/Express, Python/Django.
    - **APIs**:
        - **REST** (stateless, HTTP methods).
        - **SOAP** (XML-based, strict standards).
        - **GraphQL** (client-driven queries).
    - **Architectural Styles**:
        - **Monolithic**: Single codebase (simple but rigid).
        - **Microservices**: Loosely coupled, independent services (scalable but complex).
3. **Data Layer**:
    - Databases (SQL/NoSQL), cloud storage.

Key Architectural Considerations

- **Monolithic Drawbacks**:
    - Scalability issues, slow deployment, high risk of system-wide failures.
- **Microservices Benefits**:
    - Independent scaling, polyglot programming, fault isolation.
    - **Challenges**: High infrastructure costs, debugging complexity, service coordination.

Anti-Patterns & Tools

- **Big Ball of Mud**: Poorly structured, tangled codebase.
- **Helpful Tools**:
    - **API Gateways** (manage requests, authentication).
    - **CDNs** (caching for faster content delivery).
    - **Redis** (caching), **Message Queues** (async communication).

Takeaway: Three-tier architecture enables scalable, modular web apps. Frontend focuses on user interaction, backend on logic and APIs, and data layer on storage. Choose Microservices for flexibility but manage complexity; avoid monolithic pitfalls and anti-patterns.

LEC 05

Short Note: REST APIs

Key Concepts

- **API Types**: REST (most popular), SOAP (XML-based), RPC (action-based), GraphQL (client-driven queries).
- **REST Basics**:
    - **Architectural Style** (not protocol) by Roy Fielding.
    - Uses **HTTP methods** (GET, POST, PUT, DELETE) for CRUD operations.
    - Responses in JSON/XML with **hyperlinks** (HATEOAS) for state transitions.

Six REST Architectural Constraints

1. **Client-Server**: Separation for scalability and independence.
2. **Stateless**: Each request contains all necessary info; no server-side session.
3. **Cacheable**: Responses labeled as cacheable to improve efficiency.
4. **Uniform Interface**:
    - **Self-descriptive messages**, **resource identification via URIs**, **HATEOAS** (hypermedia links guide client actions).
5. **Layered System**: Intermediaries (proxies, gateways) enhance security and performance.
6. **Code on Demand** (Optional): Clients execute scripts (e.g., JavaScript).

True REST vs. RPC

- **RPC-style APIs** lack HATEOAS, requiring clients to know endpoints in advance.
- **True REST** includes hyperlinks (e.g., "links": {"self": "http://..."}), enabling dynamic interaction.

Richardson Maturity Model

- **Level 0**: Single endpoint (POX/XML).
- **Level 1**: Resource-specific URIs.
- **Level 2**: Proper HTTP methods.
- **Level 3**: **HATEOAS** (highest maturity).

Facilitators of REST

- **HTTP**:
    - Methods (GET/POST/PUT/DELETE).
    - Status codes (2XX success, 4XX client errors, 5XX server errors).
    - Headers (e.g., Content-Type, Authorization).
- **JSON**: Lightweight data format.
- **HAL**: Standard for embedding hyperlinks.

Best Practices

- Use **meaningful URIs** (e.g., /users/1).
- Return **JSON** and proper HTTP status codes.
- Implement **authentication** (OAuth, JWT).
- Follow **HATEOAS** for discoverability.

Key Takeaway: REST APIs prioritize scalability, statelessness, and client-server decoupling. Adherence to constraints (especially HATEOAS) ensures true REST compliance, enabling robust and flexible web services.

LEC 06

Short Note: REST API Authentication & Authorization

Authentication vs. Authorization

- **Authentication**: Verifies user identity (e.g., username/password, JWT).
- **Authorization**: Determines user access rights (e.g., roles, scopes).

Methods for REST API Security

1. **HTTP Basic Authentication**:
    - Credentials sent as Base64 in headers.
    - **Pros**: Simple, lightweight (suitable for IoT).
    - **Cons**: Insecure without HTTPS.
2. **API Keys**:
    - Unique key passed in headers/query parameters.
    - **Pros**: Easy to implement.
    - **Cons**: No expiration; risk of theft. Always use with HTTPS.
3. **JSON Web Tokens (JWT)**:
    - Signed tokens containing user claims (e.g., roles).
    - **Pros**: Stateless, scalable, self-contained.
    - **Cons**: Token contents visible if intercepted (use HTTPS).
4. **OAuth 2.0**:
    - Authorization framework for delegated access (e.g., third-party apps).
    - **Key Roles**: Resource Owner, Client, Authorization/Resource Servers.
    - **Flows**: Authorization Code (server-side), Client Credentials (machine-to-machine).
5. **OpenID Connect (OIDC)**:
    - Extends OAuth 2.0 with authentication (ID tokens).
    - Enables social logins (e.g., "Sign in with Google").

Best Practices

- Use **HTTPS** to encrypt data in transit.
- **JWT**: Set short expiration times and use secure storage.
- **OAuth**: Limit scopes to least privilege.
- Avoid **Basic Auth** and **API Keys** for sensitive applications.

Key Takeaway: Authentication validates identity, while authorization controls access. Choose methods like JWT for stateless APIs, OAuth 2.0 for third-party access, and always prioritize security with HTTPS and proper token management.