UMD DATA605 - Big Data Systems

Orchestration with Airflow Data wrangling Deployment

Instructor: Dr. GP Saggese - gsaggese@umd.edu**
v1.1

Software testing

- Evaluate the functionality, reliability, performance, and security of a product and ensure it meets specified requirements
  - Software testing is a critical phase in the development process
- Adage:
  - "If it's not tested, it doesn't work"
  - "Debugging is 2x harder than writing code"
    - Corollary: if I'm doing my best to write code, how I can possibly debug it?
- **Different types of testing**
- **Unit testing**: test individual components to ensure that each part functions correctly in isolation
- **Integration testing**: ensure that components work together as expected (e.g., detect interface defects)
- **System testing**: evaluate a fully integrated system's compliance with its specified requirements

Software testing

- **Smoke/sanity testing**: A quick, non-exhaustive run-through of the functionalities to ensure that the main functions work as expected
  - E.g., decide if a new build is stable enough to use
  - E.g., the application doesn't crash upon launching
- **Regression testing**: ensure that the new changes have not adversely affected existing functionality
  - Confusing: regressing in the sense of "getting worse"
- **Acceptance testing**: final phase of testing before the software is released
  - More used in waterfall workflows than Agile development
- **Performance testing**: load testing, stress testing, and spike testing
- **Security testing**: identify vulnerabilities, threats, and risks in the software
- **Usability testing**: assess how easy it is for end-users to use the software application
  - E.g., UI/UX
- **Compatibility testing**: check if the software is compatible with different browsers, database versions, operating systems, mobile devices, ...

CI / CD

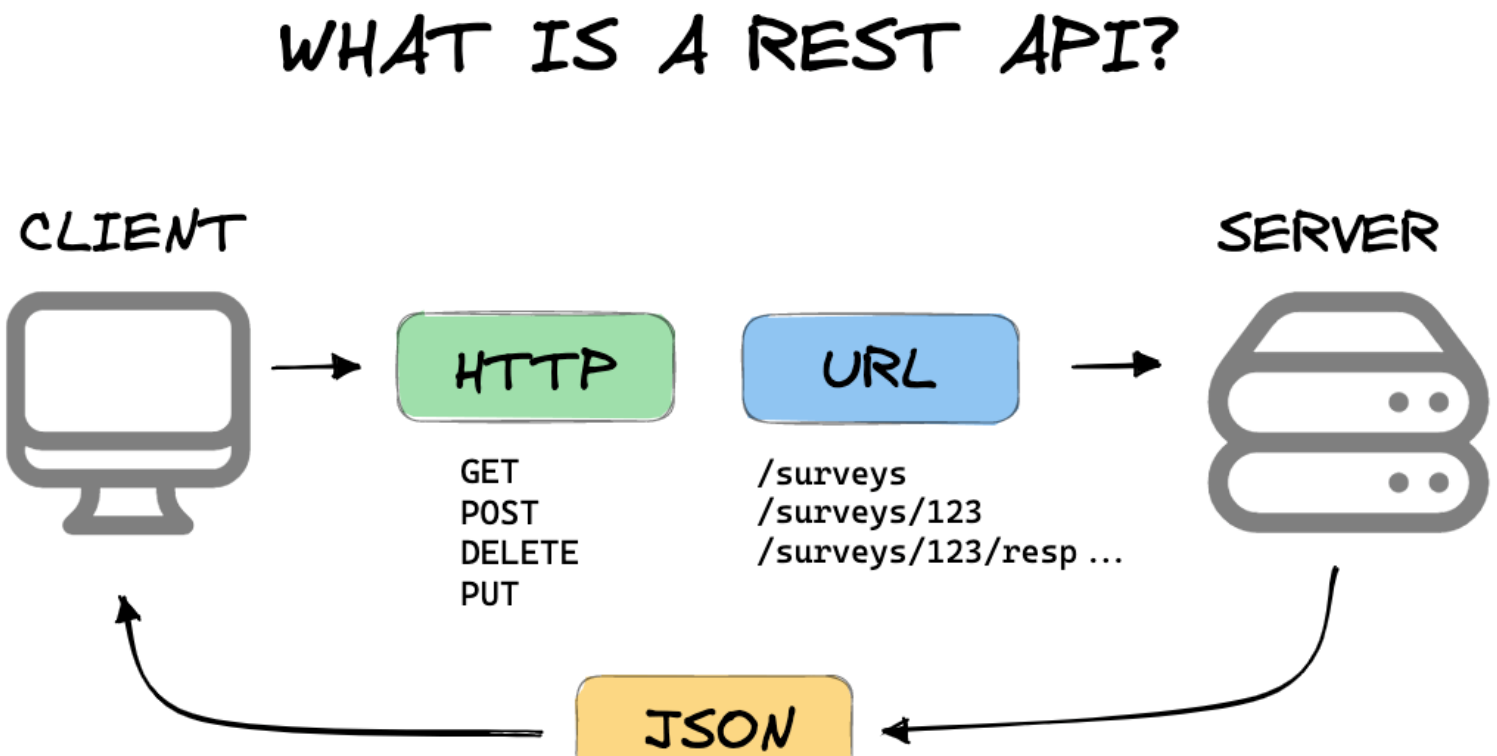
- **Continuous integration (CI)**
  - Devs merge their code changes into a central repository, often multiple times a day
  - Automated build and test code after each change
  - **Goal**: Detect and fix integration errors quickly
  - Need unit tests! Add code together with tests!
- **Continuous deployment (CD)**
  - Automatically deploy all code changes to a production environment
    - without human intervention
    - after the build and test phases pass
  - **Goal**: new features, bug fixes, and updates are continuously delivered to users in real-time
- E.g., GitHub actions, GitLab workflows, AWS Code, Jenkins

RESTful API

- REST API
  - = Web service API conforming to the REST style
  - REST = REpresentational State Transfer
  - Style to develop distributed systems
- **Uniform interface**
  - Refer to resources (e.g., document, services, URI, persons)
  - Use HTTP methods (GET, POST, PUT, DELETE)
  - Naming convention, link format
  - Response (XML or JSON)
- **Stateless**
  - Each request from client to server must contain all of the information necessary
  - No shared state
  - Inspired by HTTP (modulo cookies)

RESTful API

- **Cacheable**
  - The data in a response should be labeled as cacheable or non-cacheable
  - If cacheable, the client can reuse the response later
  - Increase scalability and performance
- **Layered system**
  - Each layer cannot "see" beyond the immediate layer that they interface
  - E.g., in a tier application



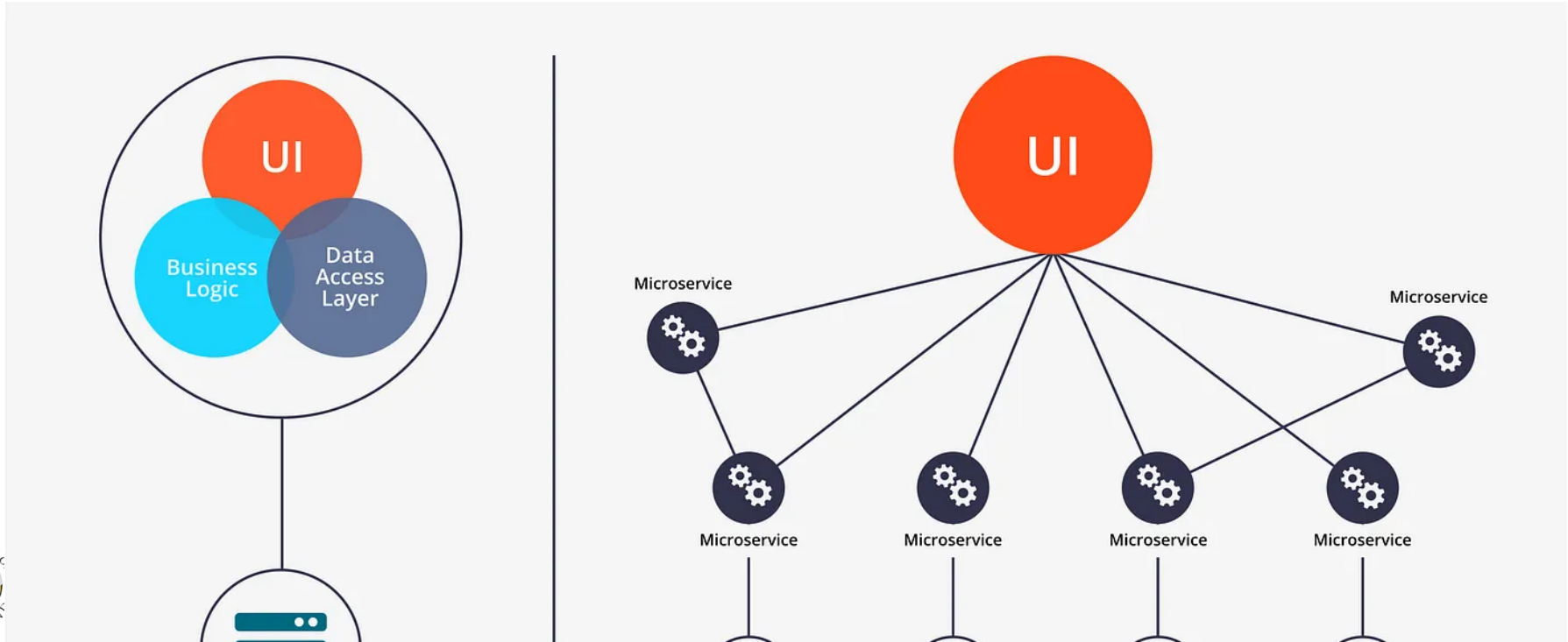
Stages of deployment

- The deployment of software progresses through several environments
  - Each environment is designed to progressively test, validate, and prepare the software for release to end user
- **Development environment (Dev)**
  - Individual for each developer or feature team
  - Goal: Where developers write and initially test the code
- **Testing or Quality Assurance (QA) environment**
  - Mirrors the production environment as closely as possible to perform under conditions similar to production
  - Goal: systematic testing of the software to uncover defects and ensure quality
- **Staging/Pre-Prod environment**
  - Final testing phase before deployment to production
  - Replica of the production environment used for final checks and for stakeholders to review the new changes
- **Production Environment (Prod)**
  - It is the live environment where the software is available to their end users
  - Highly optimized for security, performance, and scalability
  - Focus is on uptime, user experience, and data integrity

Semantic versioning

- Semantic versioning is a versioning scheme for software that aims to convey meaning about the underlying changes
  - Systematic approach
  - Understand the potential impact of updating to a new version
- Major Version (**X**.y.z)
  - Incremented for incompatible API changes or significant updates that may break backward compatibility
- Minor Version (x.**Y**.z)
  - Incremented for backward-compatible enhancements and significant new features that don't break existing functionalities
- Patch Version (x.y.**Z**)
  - Incremented for backward-compatible bug fixes that address incorrect behavior
- Pre-release Version:
  - Label to denote a pre-release version that might not be stable (e.g., `1.0.0-alpha`, `1.0.0-beta`)
  - These releases are for testing and feedback, not for production use
- Build Metadata
  - Optional metadata to denote build information or environment specifics
  - E.g., 1.0.0+20210313120000 or 1.0.0+f8a34b3228c

Microservices vs Monolithic Architecture



Microservice Architecture

- **Modularity**: composed of small, independently deployable services, each implementing a specific business functionality
- **Scalability**: services can be scaled independently, allowing for efficient use of resources based on demand for specific features
- **Technology diversity**: each service can be developed using the most appropriate technology stack for its functionality
- **Deployment flexibility**: allows for continuous delivery and deployment practices, enabling faster iterations and updates
- **Resilience**: failure in one service doesn't necessarily bring down the entire system; easier to isolate and address faults
- **Cons**
- Complexity to deploy
- Needs tooling

Monolithic Architecture

- **Simplicity**: (initially) simpler to develop, test, deploy, and scale as a single application unit
- **Tightly coupled components**: all components run within the same process, leading to potential scalability and resilience issues as the application grows
- **Technology stack uniformity**: the entire application is developed with a single technology stack, which can limit flexibility
- **Deployment complexity**: Updates to a small part of the application require redeploying the entire application
- **Single point of failure**: Issues in any module can potentially affect the availability of the entire application