UMD DATA605 - Big Data Systems

7.4: Big Data Architectures

Instructor: Dr. GP Saggese, gsaggese@umd.edu

Software Testing

- **Goal**
  - Evaluate functionality, reliability, performance, and security of a product to ensure it meets requirements
  - Software testing is critical in development
- **Adages**
  - *"If it's not tested, it doesn't work"*
  - *"Debugging is 2x harder than writing code"*
    - Corollary: *"If you do your best to write code, how can you debug it?"*
- **Many different types of testing**
  - What do you test?
  - From what point of view?
  - ...

What Are You Testing?

- **Unit testing**
  - Test individual components for correct functionality in isolation
- **Integration testing**
  - Ensure components work together as expected
  - Detect interface defects
- **System testing**
  - Evaluate a fully integrated system's compliance with requirements

How Are You Testing?

- **Smoke/sanity testing**
  - Quick check of functionalities to ensure main functions work
  - E.g., decide if a new build is stable
  - E.g., application doesn't crash on launch
- **Regression testing**
  - Ensure new changes don't affect existing functionality
- **Acceptance testing**
  - Final testing phase before release
  - More common in waterfall than Agile
- **Performance testing**
  - Load, stress, and spike testing
- **Security testing**
  - Identify vulnerabilities, threats, and risks
- **Usability testing**
  - Assess ease of use for end-users
  - E.g., uI/UX
- **Compatibility testing**
  - Check compatibility with browsers, database versions, OS, mobile devices

CI / CD

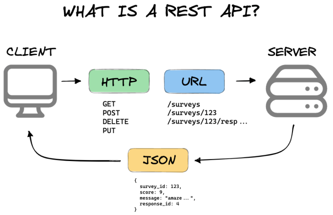
- **Continuous integration (CI)**
  - Merge code changes into a central repository multiple times a day
  - Automate build and test after each change
  - Add code with unit tests
  - *Goal*: Detect and fix integration errors quickly
- **Continuous deployment (CD)**
  - Automatically deploy code changes to production
    - Without human intervention
    - After build and test phases pass
  - *Goal*: Deliver features, bug fixes, and updates continuously
- **Examples**
  - GitHub Actions, GitLab Workflows, AWS Code, Jenkins

RESTful API

- **REST API**
  - REST = REpresentational State Transfer
  - Style of building API for web services / distributed systems
- **Uniform interface**
  - Refer to resources
    - E.g., document, services, URI, persons
  - Use HTTP methods
    - E.g., `GET`, `POST`, `PUT`, `DELETE`
  - Naming convention, link format
  - Response (XML or JSON)
- **Stateless**
  - Each request contains all necessary information
  - No shared state
  - Inspired by HTTP (modulo cookies)

RESTful API

- Cacheable
  - APIs label response data as cacheable or non-cacheable
  - Client can reuse cacheable responses
  - Increase scalability and performance
- Layered system
  - Each layer interfaces only with the immediate layer
  - E.g., in a tier application



```
{
  survey_id: 123,
  score: 9,
  message: "amaze ... ",
  response_id: 4
}
```

mannhowie.com

Stages of Deployment

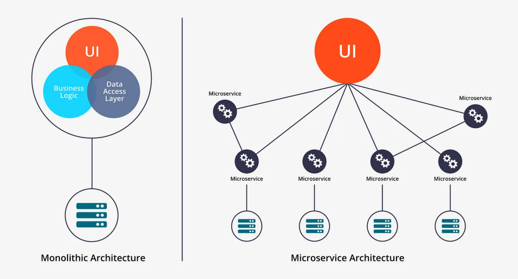
- **Software is deployed through several environments**
  - Each environment tests, validates, and prepares software for release to end user
- **Development environment (Dev)**
  - Individual for each developer or feature team
  - *Goal*: Developers write and initially test code
- **Testing**
  - Aka “Quality Assurance (QA)”
  - Mirrors production environment to perform under similar conditions
  - *Goal*: Systematic testing to uncover defects and ensure quality
- **Pre-prod**
  - Aka staging
  - Final testing phase before deployment to production
  - Replica of production environment for final checks and stakeholder review
- **Production (Prod)**
  - Live environment where software is available to end users
  - Optimized for security, performance, and scalability
  - Focus on uptime, user experience, and data integrity

Semantic Versioning

- **Semantic versioning** conveys meaning about changes to software
  - Systematic approach
  - Communicate to users potential impact of updating to a new version
- **Increment version**
  - Major Version `X.y.z`
    - Incompatible API changes
    - Significant updates that may break backward compatibility
  - Minor Version `x.Y.z`
    - Backward-compatible enhancements
    - Significant new features that don't break existing functionalities
  - Patch Version `x.y.Z`
    - Backward-compatible bug fixes that address incorrect behavior
  - Pre-release Version:
    - Denote a pre-release version that might not be stable
    - E.g., `1.0.0-alpha`, `1.0.0-beta`
    - Releases 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

- **Many styles of building complex systems**
  - *Monolith*: all features in one deployable unit
  - *Microservices*: many small services collaborating over the network



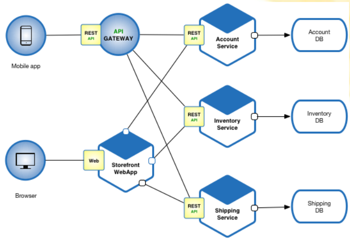
- **Heuristics**
  - Start from business domains, not technology layers
  - Align service boundaries with independent business capabilities
  - E.g., online shop with separate services for catalog, cart, payment, shipping

Monolithic Architecture

- **Monolith** = all features in one deployable unit
- **Pros**:
  - *Simplicity*: Simpler to develop, test, deploy, and scale as a single unit
- **Cons**
  - *Tightly coupled components*: Components run in the same process, leading to scalability and resilience issues
  - *Technology stack uniformity*: Developed with a single technology stack, limiting flexibility
  - *Deployment complexity*: Updates require redeploying the entire application
  - *Single point of failure*: Issues in any module can affect the entire application

Microservice Architecture

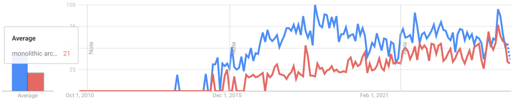
- **Microservices** = many small services collaborating over the network
- **Cons**
  - Complex deployment
  - Requires tooling



- **Pros**:
  - *Modularity*: Small, independently deployable services with specific business functionality
  - *Scalability*: Scale services independently for efficient resource use based on demand
  - *Technology diversity*: Use the best technology stack for each service's functionality
  - *Deployment flexibility*: Supports continuous delivery and deployment for faster updates
  - *Resilience*: Isolate and address faults; one service failure doesn't affect the entire system

Microservices vs Monolith: Hype

- Neither approach is a slam dunk!
    - You need to find the right granularity for your use case



"Microservice vs Monolithic" search over time