DES424 Lecture Summary

Cloud-based Application Development (Final Exam)
SIIT DE-ASD Y4T1/2022 – By Paphana Yiwsiw (@waterthatfrozen)

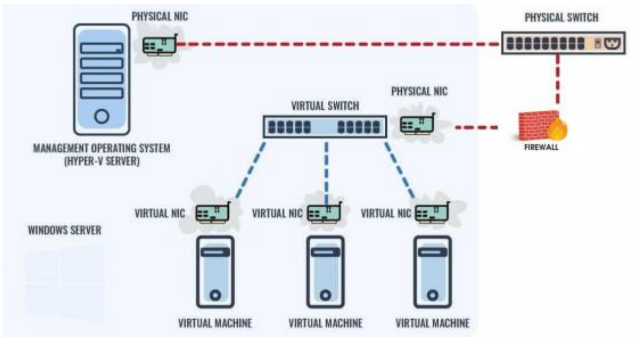
Contents

Lecture 6 – Cloud Network & Security

Traditional Approach

- It is usually performed by dedicated hardware, such as switches, routers, delivery controllers.
- SDN: Software-based, communicates via API, more freedom, manage resources virtually on the control plane.

Virtual Network (VN)

- Networking system that **emulates a physical network,** combining hardware and software to a single administrative unit. SDN can create and control VN or control hardware using software.
- Components:
  - vSwitch: software on host server, allow configuration of a VN, control and direct communication between physical network and virtual parts.
  - VN adapter: a gateway between networks, allow VMs to connect to a network.
  - Physical network: a host of VN infrastructure
  - VMs and devices: machines to connect to the networks
  - Servers: part of host infrastructure
  - Firewalls and security: monitor and prevent security threats in VN
  - Virtual NIC can connect to vSwitch or bind to physical NIC
    - Binding mode: Bridge (physical direct), NAT (Host IP shared), VN (VLAN, segment)
  - 

Network Address Translation: NAT

- Remapping single IP space to another by modifying network address info in IP headers during transiting across routing devices.
- Allow private IP devices to communicate with host on the internet by remapping IP/port

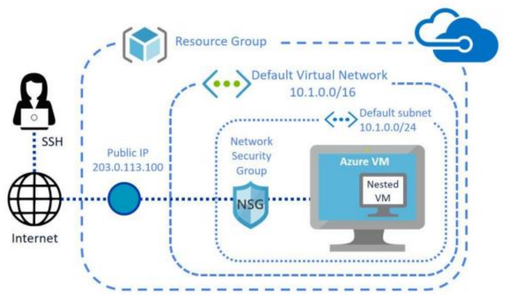
3 Classes of VN

- **VPN: Virtual Private Network**
  - Internet as a transport, connect 2 networks **or** remote client to central network
  - Create secure and encrypted tunnel between endpoints, obscuring traffic and ensure the privacy.
  - All services on both sides are presented as local services but hidden in between.
- **VLAN: Virtual Local Area Network**
  - Logically subdivides LAN into separate LAN.
  - Each VLANs act as a separate LAN, allow more security, monitoring, and management.
- **VXLAN: Virtual Extensible Local Area Network**
  - Strech layer 2 connections over layer 3 network, encapsulating ethernet frames in VXLAN packet which includes IP addresses for scalability problem in more extensible manner.

VN Benefits

- Management: segmenting and joining device virtually provide better access management.
- Security: established by traffic segmentation and access restriction by groups or devices.
- Simplification: replace rigid network and hardcoded routes with simplified optimized path.
- IP mobility: VMs can be moved to different hosts across the network easily while maintain the same IP, helps when recovering from disasters and balancing load.

VN on Cloud



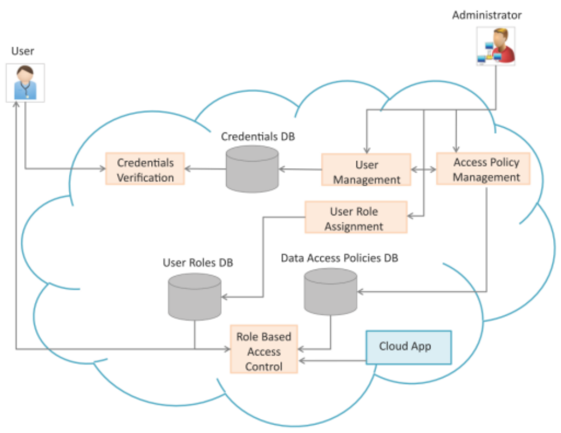
- Resource group: based on Azure account to control resources
- Default VN: default network assigned when create VM
- Network Security Group (NSG): control security for devices and services in the group
    - Activate rule or ACL (Access control list): Allow/Deny network traffic to VM in VN
    - NSGs can be associated with subnets or VM instances inside subnet.
- Azure VPN Gateway: VPN allow connecting on-premises with cloud network
- Azure NAT Gateway
    - Provides outbound internet connection for subnets in the VN.
    - NAT provides SNAT (source NAT) for that subnet after NAT gateway associated.
    - Specifies which static IP addresses VM used when creating outbound

Cloud Security Challenges

- Governance: Data ownership, quality, accessibility, policy
- Legal compliance: Standard regulation e.g., GDPR or PDPA
- Identity & access management: Authentication, authorization, user credentials.
- Data security: encryption at rest, in motion
- Availability: 24/7 all regions
- DR/BC planning: Disaster recovery/Business continuity

IAM: Identity and Access Management

- Mechanism that enforces policies necessary to control and track user identities and access privileges for IT resources, environments, and systems.
    - **Authentication:** username/password is the most common form of auth credentials managed by the IAM system.
    - **Authorization:** defines the correct granularity for access controls and supervise relationships between identities, access control rights, and resource availability.
    - **User Management:** responsible for creating new user identity and access group, resetting passwords, defining password policies, and managing privileges, related to system's administrative capabilities.
    - **Credentials Management:** establishes identities and access control rules for user accounts, mitigates threats of insufficient authorization.
- AAA: Authentication (valid user?), Authorization (permit resource?), Accounting (time and data used?)



Authentication Types

- API/Token
- Application
    - Internal: Username/password, maintain internally.
        - Password must be encrypted and make sure to use HTTPS or secure connection.
        - Password hashing: no plaintext password, stored hashed, compare hash code with the hashed password for authentication, owner doesn't know password. (Add salt to hashed password too. (pwd+salt)→ hash → stored salt and hash)
    - External: SSO(LDAP/AD/Cookie), OAuth (access with profile), Open ID (access only)
        - SSO: Single Sign-On
            - Enable users to authenticate with applications using one set of credentials
            - Advantages: no repeated password, policy enforcement, MFA, internal credential management rather than external storage.
            - Auth servers maintain passwords for all applications, pass an **authentication token** to external apps and services, stored in cookie.
        - AD: Active Directory
            - Allow IT dept. to manage and store info about users and devices within organization's network, provided centralized users management.
            - Advantages: easier to organize users, manage file access quicker, assign access right appropriately, maintain an overview of resources.
            - **Azure AD:** a cloud based IAM services, helps user access external resources such as Microsoft 365, Azure portal, and internal resources such as corporate intranet and cloud apps deployed for your own organization.
            - **MFA:** layered approach to securing data and applications, a system requires a user to present combinations of credentials to verify login identity of users. (Username + password then OTP/Google Authenticator)
        - OAuth 2.0: Open Authorization, open standard for access delegation, a way for users to grant websites or application to access to their information on others without giving them passwords. No password sending to other apps. Require redirect URI to redirect back to application after authentication finished.
            - OAuth refresh token: string that OAuth client used to get a new access token without user interaction. Must not allow the client to gain any access beyond the scope of original grant.
        - OpenID: SSO for consumer apps, login only, SAML 2.0: open standard for authorization and authentication, SSO for enterprise apps. XML format.

User Management Structure

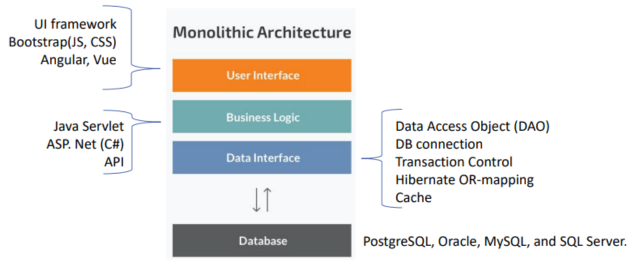
- RBAC (Role-based) or ABAC (Attribute-based)
- User Data
    - Masked password: type password and confirm password again.
    - Roles: support multiple roles for a user
- Role; Role, Role Access, Operation: show list of operation and stored in ROLE_ACCESS
- Operation: Support hierarchy and order of operation, link with menu system
- User mobile: store registration ID for sending notification, app version, device type, device version, last access date.
- Activity log: logging activities both web and API.
- Config: store key/value of configuration that can be changed e.g., mail server info, password policy, image path, app version

Simplified: Secure API with Token

- Mapping token with user account
- Control access of the token: from host IP, referrer domain, limit usage period, bypass mode for testing. Use hash for generating token, keep token usage log.

Lecture 7 – Microservices

Traditional Application Architecture



- Monolithic: built as single and indivisible unit
- Comprises of client-side UI, server-side application, and a database.
- Unified and functions are managed and served in one place.
- Drawbacks: tightly coupled between components, less usability, large code base, tough for dev and QA to understand, less scalable, doesn't follow SRP: Single responsibility people, more deployment and restart times, new technology barriers (app need to be rewritten)

Monolithic to Microservice

- A microservices architecture breaks down into a collection of smaller independent units.
- All service have their own logic and the database as well as perform the specific functions
- The application can access microservices through direct call
- Case
  - E-commerce; front: UI, back: check credit, maintain inventory, ship orders.
  - Uber
    - clear ownership of services to individuals' team, boost speed of development.
    - Fast scaling, focus only on the scalable services, update individual service without disruption, more reliable and fault tolerance.
  - Online Media: streaming media online, most liked, user info, uploading, trending, content, recommendation, localization, etc.

Advantages/Disadvantages of Microservices

- Advantages: Lower cost, more efficiency, increased agility and scalability, easier maintenance and updating, fault tolerance, independent deployment.
- Disadvantages: higher complexity, increase traffic, increased development time, deployment complexity, difficult global testing and debugging.

Microservice Architecture

- **Clients:** web browser, mobile, external, container client. Different consume functionality from multiple microservices and require frequent updates.
- **Identity Providers:** permit server-to-server and user-driven access to identity data.
- **JWT (JSON Web Token):** creating data with optional signature and encryption, payload contains JSON of some number of claims, tokens are signed either private secret or public/private key.
- **API Gateway:** single app entry point, routing request and translate protocol.
- **Messaging formats:**
  - Synchronous Message: **REST,** clients wait for responses, relies on stateless, client-server, and HTTP protocol.

        - Asynchronous Message: client doesn't wait for responses, use AMQP, STOMP, MQTT
            - Used in nature of messages is defined and interoperate between implementations
            - Popular message brokers are RabbitMQ and Apache Kafka.
            - Model: One-to-one or Publish-and-Subscribe
- **Data Handling/Databases:**
    - Each microservice owns a private database to capture their data and implement business functionality.
    - Maintain ACID properties: Atomicity, Consistency, Isolation, and Durability.
    - When using single database cluster
        - Leveraging the schemas or tables
        - RULE OF THUMB: writing is allowed from microservices that own the data; reading may be allowed to any services needed.
    - When using multiple database
        - Duplicate the data e.g., AWS DMS(Data Migration Service)
        - Writing is only allowed by microservice associated with the database; reading is allowed from any database where the data is duplicated.
- **Service Discovery**
    - Manage deployment and load distribution
        - Service provider: originates service instances over the network (API gateway)
        - Service registry: database that stores the available location of service instances
        - Service consumer: retrieves the location of a service instance from registry.
    - 2 Major discovery patterns:
        - Client-side discovery pattern; searches the service registry to locate a service provider, selects an available appropriate service instance using load balancing, then make a request.
        - Server-side discovery pattern; load balancer searches the service registry and forwards request accordingly once applicable service found.
- **Management:** allow business users to configure services during run-time
    - Docker / Kubernetes / Openshift / Istio
- **Static Content**
    - After microservices communicate within themselves, deploy static content to a cloud storage service that can deliver them directly to client using CDN
- **Logging**
    - Send all generated logs across the hosts to a centralized external place while microservices are running on multiple hosts.

Microservice Deployment

- Infrastructure View: can run on different infrastructure e.g., server, VM, container, or cloud.
- **Single Machine, Multiple Process**
    - Basic form of microservice, group of processes with load balancing
    - Benefits: Lightweight, convenience, easy troubleshooting, fixed billing
    - Limits: No scalability, single point of failure
    - App pack such as exe, binary, JAR, WAR
        - Source Code → CI/CD → Production Machine
        - When enter CI, build artifacts, retrieved in CD.

- **Multiple Machine, Multiple Process**
    - Scale up (upgrade server) or scale out (add more server) when the application’s capacity is exceeded.
- **Containers**
    - Running microservices directly as processes is very efficient
    - The server must be maintained with necessary dependencies and tools, runaway process can consume all memory/CPU, deploying and monitoring on many servers can be troublesome.
    - Containers: Packages that contain what program needed to run, provide just enough virtualization to run software in isolation.
    - Container image is a self-contained unit that can run on any server without installing any dependencies or tools.
    - Benefits: Isolation (No dependency conflicts), Concurrency (Multiple instances), Less overhead (No need to boot OS), No-install deployment, Resource control
    - Change process to container and distribute the load across numbers of machines.
- **Serverless Container**
    - CaaS e.g., AWS Fargate and Azure Container run containerized applications without dealing with servers.
    - ECS with Fargate allows us to run containers w/o renting servers, maintained by cloud providers.
- **Orchestrators**
    - Platforms specialized in distributing container workloads over a group of servers.
    - Most well-known: Kubernetes
        - Custom deployment scripts, codify the desired state with a manifest file and let the cluster take care of the rest.
- **Serverless Functions**
    - Use the cloud to build and run code on demand.
    - AWS Lambda and Azure functions handle all infrastructure details required for scalable and highly available services (easy to use, scale, pay-as-you-go).

Lecture 8 – Cloud Native & Docker

Type of Cloud Applications: Monolithic → Cloud-Enabled → Cloud-Based → Cloud-Native

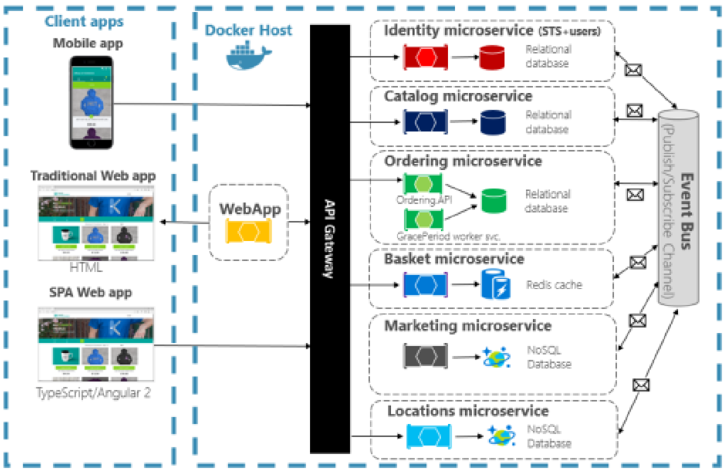
Cloud-Enabled Applications

- Built traditionally, migrated to cloud, designed in monolithic, depended on local resources H/W.
- Refactored to virtual resources, remained same underlying architecture.
- **No advantage** of shared services or resource pool, issue with scalability

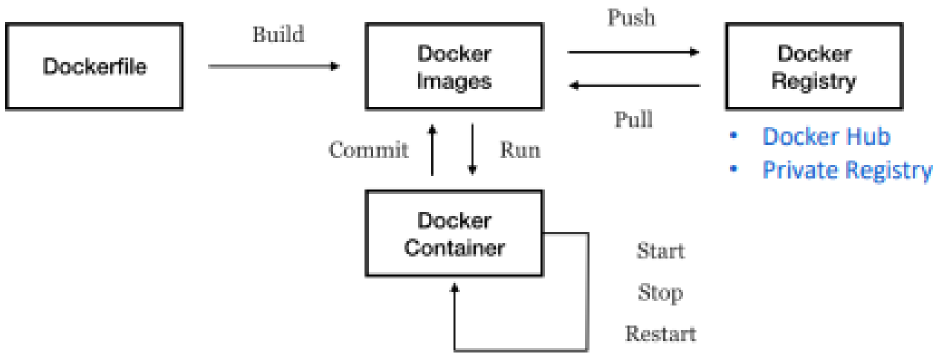
Cloud-Based Applications

- Middle ground b/w cloud-native and cloud-enabled
- Leverage some capabilities e.g., higher availability and scalability, but don't want to redesign whole applications to use cloud services
- Adopt IaaS, PaaS, SaaS. No need to maintain a server or worry about backup.

Cloud-Native Applications

- Architected to run in a public cloud using cloud tech, allow more accessibility and scalability.
- Comprised of CI, orchestrators, and container engines. Core is microservices.
- 

DOCKER!

- Standardized, executable components that combine app source code with OS and dependencies required to run in any environment. Like VM, but lightweight, smaller, faster, and easier to manage.
- Architecture: client-server; docker client talks to daemon (building, running, and distributing containers), via REST API over UNIX sockets or network interface. Daemon will pull images from registry and run those as containers. (Users interact with client, daemon execute command)
- Communicate with client using docker CLI; build, push, pull, run, start, stop.
- 

- Docker CLI command:
  - Run a new container:
    docker run IMAGE `[--name (container name),`
    `-p (host port:continaer port), -P (map all port),`
    `-d (detach, run in background), --hostname (assign to host name),`
    `-v (host directory: target directory),`
    `-it --entrypoint (EXECUTABLE IMAGE, change the entry point)]`
  - Mange container
    - List containers: **`docker ps [-a (all)]`**
    - Delete container: **`docker rm`** `[-f (force)]`
    - Start container: **`docker start`**
    - Stop container: **`docker stop`**
    - Copy file from container to host: `docker cp CONTAINER:SOURCE TARGET`
    - Copy file from host to container: `docker cp TARGET CONTAINER:SOURCE`
    - Start shell inside container: `docker exec -it CONTAINER EXECUTABLE`
    - Run command from container: **`docker exec CONTAINER COMMAND`**
    - Create image out of container: `docker commit CONTAINER`
  - Mange images
    - Download image: `docker pull IMAGE[:TAG]`
    - Upload image: `docker push IMAGE[:TAG]`
    - Remove image: `docker rmi IMAGE`
    - List all images: `docker images`
    - Build image from Dockerfile: **`docker build -t IMAGE[:TAR] DIRECTORY`**
    - Tag image: `docker tag IMAGE NEWIMAGE`
    - Save image to .tar file: `docker save IMAGE > FILE.tar`
    - Load image from .tar file: `docker load -i FILE.tar`
  - Info & stats
    - Log: `docker logs CONTAINER`
    - Stats of running container: `docker stats`
    - Processes of container: `docker top CONTAINER`
    - Version of docker: `docker version`
    - Show modified files in container: `docker diff CONTAINER`
    - Show mapped ports of container: `docker port CONTAINER`
  - CONTAINER → container name, IMAGE → image name, TAG → tag label (e.g., 1.0), TARGET → target file, DIRECTORY → directory path

- Docker registry: store docker images, docker hub=default image public storage location, docker pull request is made to pull image from docker registry, push command to store.
- Docker image: execute code in docker container, image = set of instructions to build docker container. Image is equivalent of a snapshot in VM, but more lightweight, act as starting point.
  - Where to store data: let docker mange the storage with own internal volume management, or create directory on host system and mount to the container from inside container
  - Docker image is made up from bundled files, with all essentials and dependencies, required to configure a fully container environment.
    - Interactive method: saving the result state as new image.
    - Dockerfile: provides the specification for creating a docker image.
      - Leftmost word = instruction, right is arguments of those instructions.

        - `FROM`: specify base image to be pulled from docker hub
        - `RUN`: execute command during image build process
        - `ENV`: set environment variables, both build and running time.
        - `COPY`: copy local files and directories to the image
        - `EXPOSE`: specify port to expose
        - `ADD`: can `COPY` from URL; not recommended, use `curl,` or get using `RUN`.
        - `WORKDIR`: set current working directory for this Dockerfile.
        - `VOLUME`: create or mount the volume of the container
        - `USER`: set username and UID when running container
        - `LABEL`: specify metadata of docker image
        - `ARG`: set build time variables, with key and value.
        - `CMD`: execute command in running container, only the last `CMD` is applied.
        - `ENTRYPOINT`: execute when docker container starts. Default: `/bin/sh -c`
      - Basic instructions in Dockerfile: `FROM`, `LABEL`, `RUN`, `WORKDIR`, `EXPOSE`, `CMD`
      - Good practice of Dockerfile: lightweight base image, install only what to use.
      - Use `.dockerignore` to exclude files/directories, push to docker hub using **`docker push username/package:version`**

- Docker container: instance of image that can be managed through docker API. Lightweight and independent. Any data will be scrapped when container is removed.
- Docker daemon: fulfilling container actions, mainly run-in background, manage docker network, storage, volume, container, services, and images. Build when requested.
- Docker networking: allow containers to communicate with others, provide services outside via host network, support multiple networks.
  - None: disable network systems, no connection with other containers
  - Bridge: default network driver, used when multiple containers connect w/ same host.
    create private network to the host, so container on this network can communicate.
    create docker network
    ```
    docker network create --driver bridge –subnet 172.17.0.0/16 mynet
    ```
    connect network to container:
    ```
    docker network connect mynet CONTAINER1
    ```
    specify IP address to container
    ```
    docker network connect --ip 172.17.0.10 mynet CONTAINER2
    ```
  - Host: Used when user doesn't require isolation between container and host.
    Use the host's IP address and TCP port space to expose service running inside container
    ```
    docker run -it --name web2 --net=host mycontainer
    ```
  - Overlay: allow communication between swarm services, when run on different hosts.
    Simplified version in multi-host networking, swarm scope driver, operate across swarm.
  - macvlan: make container look like physical driver, assigned MAC address and route traffic between containers through this MAC address.
    Connect container interface directly to host interfaces, addressed with routable IP addresses that on subnet of external network, map subnetwork to another via VLAN.
- Docker Storage: enable storage admin to configure and support application data storage within docker container deployments
  - Used storage drivers to manage the content of the image and writable container layer
  - By default, containers don't write data permanently to any location, storage must be configured if want to store data permanently. If stored outside, usable even removed.

- Layers: docker built one READ-ONLY layer for each instruction in the Dockerfile.
    - When command run is executed, docker builds READ-WRITE container layers
    - New file can be created on the container
    - Copy-on-Write Mechanism: Modify existing files in the image layers, local copy is created on the container layer, change only live on the container.
    - Files and modifications are destroyed when container is destroyed.
    - Use volume mapping techniques to persist the data.
- Types of storage
    - Docker Volumes (Permanent)
        - Area of isolated storage on host system contain running docker engine.
        - Default path is /var/lib/docker/volumes
        - Same volume can use for the next or different docker container.
    - Bind Mount (Permanent)
        - May be stored anywhere on the host system, even important directories.
        - Non-docker processes can modify any time. Mount using flag --mount or -v
        - `docker run -v /home/data:/var/www/html nginx:latest` or `docker run -mount type=bind, source=/home/data, target=/var/www/html nginx:latest`
    - Tmpfs (Non-persistence)
        - Data is written directly on host's memory and deleted upon container stop.
        - Useful when involved sensitive data or don't want it to be permanent.
        - `docker run -d --mount type=tmpfs,destination=/var/cache/ nginx:latest` or `docker run -d -tmpfs /var/run nginx:latest`

Docker Lifecycle

Build (docker build) → Push (docker push) → Run (docker run) → Upgrade (docker stop)

Running Container

- Run with default command: `docker run -i -t ubuntu` (`-i -t` for interactive container)
- Run with explicit command: `docker run alpine:3.5 /bin/ls -l`
- Set environment variables: `docker run -e PATH=/bin:/usr/bin alpine:3.5 ls`
- Open second terminal in a running container: `docker run -i -t --name blah ubuntu` **or** `docker exec -i -t blah /bin/bash`
- Inspect docker image `docker image inspect nginx`

Docker Swarm

- Container orchestration tool, allow user to manage multiple containers deployed across multiple host machines. Key benefits: high level of availability for applications.

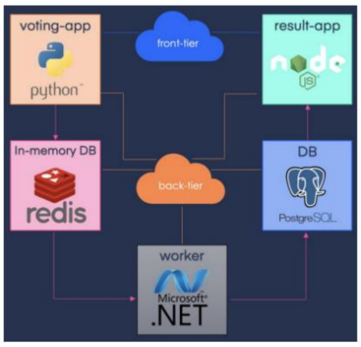
Example of Dockerfile (Web with nginx)

```
LABEL maintainer="contact@dev.com"
FROM ubuntu:18.04
RUN apt-get -y update && apt-get -y install nginx
COPY files/default /etc/nginx/sites-available/default
COPY files/index.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["/usr/sbin/nginx","-g",,"daemon off;"]
```

Lecture 9 – Docker Compose

- A tool in defining and sharing multi-container applications, define services in YAML file while tear the composed container in a single command.
- 3 steps process: Define environment with Dockerfile, define services that make your app, run docker compose up and docker compose starts and run the app.
- Example of docker-compose.yml

```
version: 3
services:
      redis:
            image: redis
            networks:
                  -back-tier
      db:
            image: postgres:9.4
            networks:
                  -back-tier  # only back-tier provided
      vote:
            build: ./vote
            ports:
                  - 5000:80
            networks:
                  - front-tier      # allow for both inside and outside
                  - back-tier
      result:
      worker:
networks:
      front-tier:                  # front-tier allow access from outside
      back-tier:                   # back-tier use internally
```



YAML (Yet Another Markup Language) [.yml / .yaml]

- Data serialization language for writing configuration files, human-readable & easy to understand
- Python-style indentation for nesting, Tab is not allowed, whitespaces are used instead.
- No format symbols e.g., braces, square, brackets, closing tag, quotation marks. # for comment.
- Node anchors: mark with &, reference with *

YAML for docker compose

- Usually contain at least 1 service, optional for volumes and network
- Services refer to containers' configuration
  - Service can be image from docker hub or build from Dockerfile
  - Ex. web app with front/back/database, split into 3 images and define as services.
- Volumes define physical disk space shared between host and container
  - 2 types of volume: named and host ones.
    - Named volume: docker managed, auto mounted in self-generated directory.
    - Host volume: specify existing folder in the host
    - Host volume at the service level, named volume at the global/outer level
- Networks define communication between containers and between container and host.
  - Service can communicate with another on the same network by referencing by container name and port (e.g. example-container:8080), with that port being exposed.

- Environment variables
    - Define static env variables and dynamic variables with ${} notation.
    - Methods to provide environment variable
        - .env file in the same directory: key=value
        - OS export before docker-compose up: export key=value → docker-compose up
        - Inline command: KEY=value docker-compose-up

Example of docker-compose.yml

```
version: 3
services:
      db:                                       # database service
            image: mysql:5.7                    # specify image
            volumes:
             - db_data:/var/lib/mysql
            restart: always                     # always restart
            environment:                        # set environment variables
               MYSQL_ROOT_PASSWORD: somewordpress
               MYSQL_DATABASE: wordpress
               MYSQL_USER: wordpress
               MYSQL_PASSWORD: wordpress
      wordpress:                                # wordpress service (Web app)
            depends_on:                         # set service dependency
             - db                               # depends on service db
            image: wordpress:latest             # specify image
            ports:
             - "8080:80"                        # Map port with host port
            restart: always
            environment:
             WORDPRESS_DB_HOST: db:3306
             WORDPRESS_DB_USER: wordpress
             WORDPRESS_DB_PASSWORD: wordpress
volumes:                                        # volume that is visible to all services
      db_data:

----
```

Lifecycle management

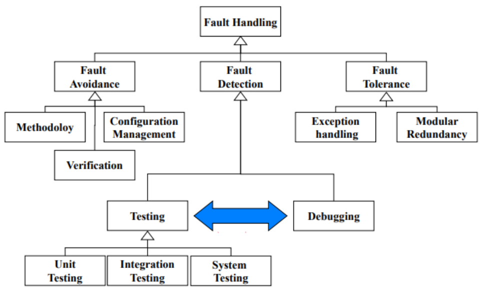
1. First time running
   `docker-compose up`
2. Start services (not the first time)
   `docker-compose start`
3. Different file name than default docker-compose.yml (-f for alternate file name)
   `docker-compose` **`-f custom-compose-file.yml`** `start`
4. Run in background as daemon (-d for detach mode)
   `docker-compose up` **`-d`**
5. Shutdown
   `docker-compose stop`
6. Deletion (delete everything)
   `docker-compose down`

Lecture 10 – Software Testing & CI/CD

Terminology

- **Failure**: deviation of observed behavior from specified behavior
- **Error state**: System in a state such further processing can lead to **failure**
- **Fault**: mechanical or algorithmic cause of an error a.k.a. bug
- **Validation**: checking activity of deviations between observed behavior and specification.
- Example of faults and errors:
  - Interface faults: client/server mismatch, requirement/implementation mismatch
  - Algorithmic faults: missing initialization, incorrect branching, null test missing.
  - Mechanical faults: operating temperature outside equipment specifications
  - Errors: null reference, concurrency, exceptions.
- Relative cost of bugs: cost to fix a bug increase exponentially, "found later, cost more".

How to deal with faults

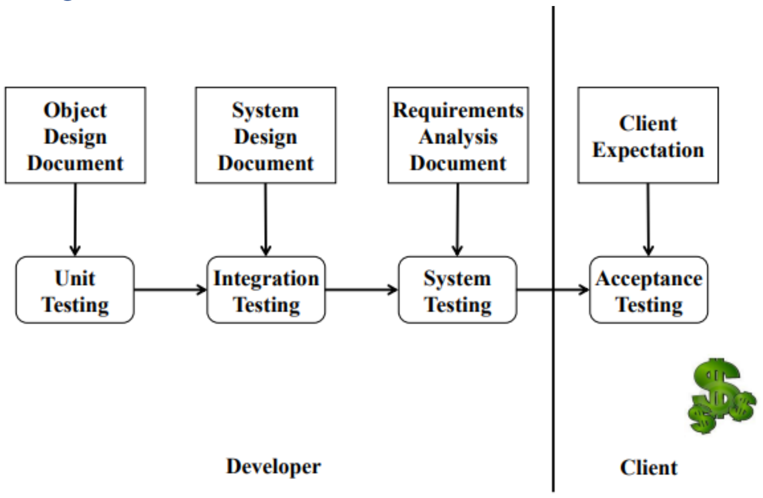


- Avoidance
  - Methodology to Reduce complexity
  - Use configuration management to prevent inconsistency
  - Verification to prevent algorithmic faults
  - Reviews
- Detection
  - Testing: provoke failure in planned way
  - Debugging: Find and remove cause of faults of observable failure
  - Monitoring: Deliver information about the state, used during debugging
- Tolerance
  - Exception handling, modular redundancy

Developing effective test

- You must have detailed understanding of the system, application and solution domain knowledge, testing techniques, skill to apply these techniques.
- Testing is done by independent testers. We often develop a certain mental attitude that program should behave in certain way but in fact it is not.

Testing Activities



Unit test: Object Design
Integration test: System Design
System test: Requirements analysis
Acceptance test: Client Expectations

Testing Types

- **Unit testing**
  - Testing done to individual components, either class or subsystem. Done by developers.
  - Goal: Confirm that component is correctly coded and carried out intended functionality
  - Dynamic testing at run time
    - Black-box testing
      - Focused on I/O behavior; can predict the output for any given input.
      - Goal: reduce test cases number by **equivalence partitioning**
        - Divide input condition equally, then choose test case for each.
        - Ex. Range(below, equal, above), Set (valid, invalid)
    - White-box testing
      - Path testing: all paths in modules must be used at least once.
      - Loop testing: start and end conditions works properly
      - Condition testing: execute, all possible outcomes come at least once.
  - Unit test heuristic
    - Create when object design is finished, black for functional, white for dynamic
    - Develop test case: find effective number of test cases
    - Cross-check test cases to eliminate duplicates
    - Desk check your source code
    - Create test harness: test drivers and stubs are needed for integration testing
    - Describe test oracle(mechanism to tell pass or fail test), usually first passed one.
    - Compare the results of the test with the test oracle, automate if possible.
  - JUnit: Java framework for writing and running unit tests. Written with “test first” and pattern-based development.
- **Integration testing**
  - Testing done to group of subsystems, eventually entire system. Done by developers.
  - Goal: Test the interface among the subsystems.
  - Determines the order in which subsystems are selected for testing and integration.
  - Because unit tests run only in isolation and many failures caused by interactions of subsystems. Without this, system testing will be very time-consuming.
  - Failures that are not discovered during this time, it will be discovered after deployment which is costly to fix.
  - Driver: component called tested unit, control test cases, main program
  - Stubs: tested unit dependent component, partial implementation, return fake values.
    - Example with login and admin page: Login can be driver (dummy login page), Admin page is stub, acting as called program from admin.
  - **Bottom-up testing strategy**
    - Subsystem in the lowest later of the call hierarchy are tested individually. Next subsystems are tested that call the previously tested subsystems.
    - Drivers are needed, No stubs. Suitable for object-oriented or real-time system
  - **Top-down testing strategy**
    - Test the top layer first, then combine all subsystems that are called by the tested system and test the resulting collection of those subsystems. Do this until all subsystems are incorporated into the test.
    - Many stubs are needed to do the testing, no drivers.
    - Test cases can be defined based on the functional requirements.

        - **Continuous Testing**
            - Continuous build: build/test/integrate from day one. Always runnable.
            - Required integrated tool support: continuous build server, high coverage automated test, refactoring supported tool, software configuration management, issue tracking .
        - Integration Testing Steps:
            - Select a component to be tested, unit test all classes in component first.
            - Combine selected component, preliminary fix-up, make drivers and stubs.
            - Test functional requirements: define test cases that go to all use cases
            - Test subsystem decompositions: define test case with all dependencies
            - Test nonfunctional components: performance tests
            - Keep records of test cases and activities.
            - Repeat until the system is fully tested.
            - Goal of integration testing: identify failures with component configuration.

- **System testing**
    - Testing done to entire system. Done by developers.
    - Goal: Test that system meets the functional and nonfunctional requirements
    - **Functional Testing:** validates functional requirements
        - Goal: test system functionality
        - Test cases designed from requirement analysis docs or user manual, centered around use cases (requirements/key functions)
        - Treated as black box, unit test can be reused along with newly-develop test cases
    - **Performance Testing:** validates non-functional requirements
        - Test how system behaves when overloaded; identify bottlenecks etc.
        - Try unusual execution order, check system's response to large volumes of data, amount of time spent in use cases
        - Type of performance testing:
            - Stress testing: test system limit
            - Volume testing: test handle large amounts of data
            - Configuration testing: test software and hardware configuration
            - Compatibility testing: test backward compatibility
            - Timing testing: test response time and function performing time
            - Security testing: test security requirements by violating it
            - Environmental testing: test for tolerances in physical environment
            - Quality testing: test reliability, maintainability, availability
            - Recovery testing: test loss of data response of the system
            - Human factor testing: test with end users.
        - Testing for web applications: Load/Stress/Volume/Lifecycle/Optimization/Soak
        - Tools: loader.io (run from US-East AWS), blazemeter.com(AWS EC2), JMeter (Apache, pure Java application)
- **Acceptance testing**
    - Evaluate the system from developers, **Done by client.** May involve executing typical transaction on site in a trial basis
    - Goal: Demonstrate that the system meets the requirements and **READY TO USE.**
    - **Alpha test:** test by client, use developer's environment in controlled settings. Developers are ready to fix bugs.

  - **Beta test:** test use client environment without developer supervision, gets a real work in target environment.

Software Tester Role

- Find and fix bugs as early as possible
- Well-understand the application
- Study detailed functionality where bugs is likely to occur, and study code to ensure that every line of code is tested.
- Create test cases to uncover the hidden bugs and ensure the software is usable and reliable

4 Testing Steps

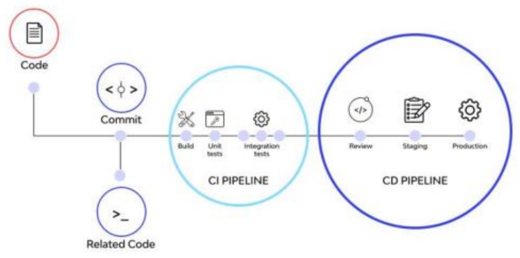
1. Select what to be tested: Analysis/Design/Implementation
2. Decide how to test: Review or inspect code/Proofs/Black or white box/Testing strategy
3. Develop test cases
   - Test case is a set of test data or situations that will be used in that unit for testing or about the attribute being measured
4. Create test oracle
   - Contain the predicted results for a set of test cases, written down before actual testing takes place.

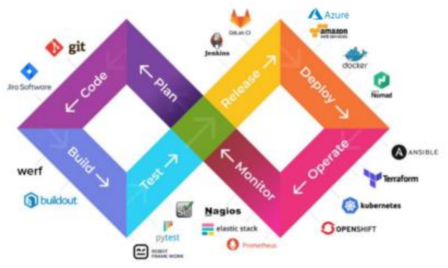
Test Documents

- **Test plan:** Quality objectives, resource need, schedules, assignments, methods.
- **Test cases:** inputs and expected outputs
- **Bug reports**
- **Test tools and automation**
- **Metrics, statistics, summaries:** number of unsolved bugs, mean time to repair

Continuous Integration/Continuous Delivery or Deployment (CI/CD)

- A method that frequently deliver apps to users with automation in stages in app development.
- A solution to the integration of new code problem that can cause dev and ops teams
- **CI:** practices that involves dev making small changes and check codes.
- **CD (Delivery):** automated delivery of completed code to testing environment
- **CD (Deployment):** next step of CD, every passed-test change automatically placed in production, resulting in many productions deployments





-
- **Pipeline**: code, commit, build, test, staging deploys, on-demand production deploys
- CI/CD is a set of development practices to deliver code changes rapidly and reliably. Focused on SDLC, highlight tools, emphasize automation.
- DevOps is a collection of ideas, practices, processes, and technologies that allow development and operations team work together to streamline product development. More focused on culture, highlight roles, emphasize responsiveness.

CI/CD Tools

- **Jenkins:** CI/CD automation software DevOps tool in Java, implement workflow/pipelines
- **Robot framework:** acceptance testing automation framework, test-driven development, uses keyword-driven testing approaches. Library: Selenium Library.

Robot Framework

- Remember that space in robot framework are tabs only.
- Example of robot test scripts (from QikVid project)

```
*** Comments ***        # Define Comments
This is a test script for login and logout function of QikVid app

*** Settings ***        # Import Library
Library    Selenium2Library
Library    XML

*** Variables ***       # Define Variables
${BROWSER}    chrome
${URL}    http://localhost:3000
${USERNAME}    ice
${PASSWORD}    12345678
${DELAY}    1.25

*** Test Cases ***            # Testing Steps
1. Open Browser
    Open Browser    ${URL}    ${BROWSER}
options=add_experimental_option("excludeSwitches", ["enable-logging"])
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}
2. Go to Log In Page
    Click Element    id=nav-user-icon
    ${PAGE_TITLE}    Get Text    xpath=//h1      # → Get text (xpath: element path in DOM)
    Log To Console    ${PAGE_TITLE}              # → Log to the console/terminal
    Should Contain    ${PAGE_TITLE}    Log In    # → Check for text in the variable
3. Enter Log In Account
    Input Text    id=login-username-input    ${USERNAME}          # → Input text
    Input Password    id=login-password-input    ${PASSWORD}
    Click Button    id=login-signin-btn
4. Go To Profile Page
    Click Element    xpath=//*[@id="nav-user-icon"]               # → Click element
    ${LOGIN_USERNAME}    Get Text    xpath=//h2
    Should Contain    ${LOGIN_USERNAME}    ${USERNAME}
    ${LOGOUT_BUTTON}    Get Text    xpath=//*[@id="user-logout-btn"]/div/p
    Should Contain    ${LOGOUT_BUTTON}    Log out
5. Log out
    Click Element    id=user-logout-btn
    ${PAGE_TITLE}    Get Text    xpath=//h1
    Should Contain    ${PAGE_TITLE}    Log In
6. Close Browser
    Close Browser
```

Lecture 11 – Scaling Services

Dapr Integration

- Dapr is a system designed to support cloud native and serverless computing.
- Portable, serverless, and event-driven runtime. Easy for dev to build resilient, stateless, stateful cloud microservices and embraces language diversity and developer frameworks

Sidecar pattern

- About separating cross-cutting operations from microservice to reduce internal complexity.
- Dapr exposes HTTP and gRPC API as a sidecar pattern, either container or process, not requiring the code to include any dapr runtime code. \

Azure Application Insights

- When building microservices application with many distributed services that communicate with each other and use different infrastructure, then it is required the mechanism to observe and trace from end to end.
- **Application Insights** is a tool to monitor azure container apps under the same container app environment and collect service telemetry and usage and engagement via integrated analytics tools.
    - **Telemetry:** data collected to observe application, 3 categories
        - **Distributed tracing:** insights into traffic between services in distributed transactions. Ex. Front-end talks with back-end to retrieve data.
        - **Metrics:** insights into performance of a services and resource utilization. Ex. CPU and memory utilization of background processor, help us to understand when we need to increase the number of replicas.
        - **Logging:** insight into code execution and occurred errors.
- Investigate tab in application insight
    - Services connectivity: for distributed tracing via application map
    - Live metrics: near real-time status of distributed application graph.
    - Transaction search: find and explore telemetry items e.g., exceptions, requests, events.
    - Failure: view frequency of failures across different operations
    - Performance: display performance details, identify longest duration, diagnose problems.

Azure CLI Command

Login to azure cloud:	`az login`	Verify login account:	`az account show`

Install extension:	`az extension add -n containerapp –upgrade # container app`
	`az extension add -n application-insights  # app insights`		

Register namespace:	`az provider register --namespace Microsoft.App`
	`az provider register --namespace Microsoft.OperationalInsights`		

Create resource group:	`az group create --name myrg --location “eastus”`		
Create ACR instace:	`az acr create --resource-group myrg --name ACRName --sku Basic --admin-enabled true`		
Build docker image and push to ACR:	`az acr build --registry ACRName --image “IMAGEName” --file “PathtoDockerfile” .`		
Create container app environment:	`az containerapp env create --name ContainerAppName --resource-group myrg --location “eastus” --dapr-instrumentation-key InstrumentKeyfromAppInsight`		

Update environment variable to container app

```
az containerapp update --name "ContainerAppName" --resource-group myrg
--set-env-vars ENV_VAR=VALUE
```

Add environment variable to container app

```
az containerapp env dapr-component set --name "ContainerAppName"
--resource-group myrg –dapr-component-name statestore --yaml "File.yaml"
```

Create and deploy container

If in windows powershell, use **** as multiline command.
FQDN = fully qualified domain name

```
az containerapp create
--name ContainerAppName                      # container app name
--resource-group myrg                        # resource group name
--environment ContainerAppName               # container app environment
--image "ACRName.azurecr.io/ImageName"       # container image in ACR
--registry-server "ACRName.azurecr.io"       # registry server
--target-port 80                             # target port (listen for incoming request)
--ingress 'external'                         # external = from public, assigned FQDN
                                             # internal = within Azure container Environment
--min-replicas 1 --max-replicas 2 --cpu 0.25 --memory 0.5Gi --query configuration.ingress.fqdn
# If you use dapr
--enable-dapr --dapr-app-id DAPRAppID --dapr-app-port 80
```

Azure CosmosDB Command

Check Azure CosmosDB storage account if exists

```
az cosmosdb check-name-exists --name CosmosDBName
```

Create Azure CosmosDB storage account for SQL

```
az cosmosdb create --name CosmosDBName --resource-group myrg
```

Create SQL database

```
az cosmosdb sql database create --account-name CosmosDBName --resource-group myrg
--name DatabaseName
```

Create SQL API Container

```
az cosmosdb sql container create --account-name CosmosDBName --resource-group myrg
--database-name DatabaseName --name SQLAPIContainerName --partition-key-path "/id"
--throughput 400
```

List Azure CosmosDB keys

```
az cosmosdb keys list --name CosmosDBName --resource-group myrg
```

Add secret to container app

```
az containerapp secret set --name ContainerAppName --resource-group myrg
--secret "KEY=VALUE"
```

Deploy new images and push to ACR

```
az containerapp update --name ContainerAppName --resource-group myrg
--image "ACRName.azurecr.io/ImageName" --revision-suffix RevisionSuffix
--cpu 0.25 --memory 0.5Gi  --min-replicas 1 --max-replicas 2
--set-env-vars "ApplicationInsight__InstumentationKey=secretref:appinsight-key"
```