Summary of Presentation: HTTP Requests, Cloud Computing, Containers, and 12-Factor Apps ENb

HTTP Requests "Classic"

1. **DNS Resolution:**
2.
    - **Client:** Where is www.zli.ch?
    - **DNS Server:** Resolves to 195.141.80.206.
3. **HTTP Request to Web Server:**
    - **Client:** GET https://www.zli.ch/index.html
    - **Web Server (195.141.80.206):** Responds with HTML content.

Cloud Computing

- **Definition:** Cloud computing is a model enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources.

- **Characteristics (NIST):**
  1. On-demand self-service
  2. Broad network access
  3. Resource pooling
  4. Rapid elasticity
  5. Measured service
- **Service Models:**
  - **▼ IaaS (Infrastructure as a Service):**
    - **Examples:**
      - Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and IBM Cloud exemplify Infrastructure as a Service (IaaS). These cloud providers furnish infrastructure services like virtual machines, storage, and networking, offering a foundation for building and deploying various applications.
      - A concrete example involves deploying a VPN (Virtual Private Network) service on AWS, leveraging the infrastructure components provided by the cloud platform.
    - **Managed Aspects:**
      - **Virtualization:** The virtualization layer, which abstracts hardware resources to create virtual machines, is fully managed by the IaaS provider.
      - **Server:** The physical servers (in the case of dedicated instances) or virtual servers (in the case of virtual machines) are managed by the IaaS provider.

    - **Disk Space:** Storage and disk space provisioning and management, including the choice of storage types, are handled by the IaaS provider.
    - **Network:** The underlying network infrastructure, including bandwidth and connectivity, is managed by the IaaS provider.
  - **Unmanaged Aspects:**
    - **Application:** Developers have full control over the application, including its code, logic, and functionality.
    - **Data:** While the infrastructure provides storage services, developers are responsible for defining and structuring their application's data.
    - **Runtime:** The execution environment for the application, including language runtimes, is the responsibility of the developers.
    - **Middleware:** Developers are responsible for selecting and configuring middleware components required by their applications.
    - **Operating System (OS):** The choice of operating system and its configuration are decisions made by the developers.

▼ **PaaS (Platform as a Service):**

- **Examples:**
  - Heroku, Google App Engine, and Microsoft Azure App Service exemplify Platform as a Service (PaaS). These platforms empower developers to build, deploy, and scale applications without dealing with the intricacies of managing the underlying infrastructure.
  - An illustrative scenario involves deploying a Java SpringBoot application on Heroku, where developers can focus on the application's code and functionality rather than the infrastructure.

- **Managed Aspects:**
  - **Runtime:** The runtime environment necessary for application execution is fully managed by the PaaS provider.
  - **Middleware:** Middleware components, if required for the application, are handled and maintained by the PaaS provider.
  - **Operating System (OS):** The underlying operating system, including updates and security patches, is managed by the PaaS provider.
  - **Virtualization:** If virtualization is employed, the provider manages the virtualization layer.
  - **Server:** The physical or virtual servers supporting the application are fully managed.
  - **Disk Space:** Storage and disk space provisioning and management are taken care of by the PaaS provider.
  - **Network:** The network infrastructure and connectivity needed for the application are managed by the PaaS provider.
- **Unmanaged Aspects:**
  - **Application:** Developers retain control over the application code, logic, and functionality.
  - **Data:** While the platform manages aspects like data storage and retrieval, developers are responsible for defining and structuring their application's data.

▼ **SaaS (Software as a Service):**

Examples:

- Salesforce, Microsoft Office 365, Google Workspace, and Dropbox are prime examples of SaaS. These services grant users access to software applications over the internet without the need for installation or maintenance.
- Listening to Spotify via the web player or engaging in conversations on WhatsApp Web are additional examples of SaaS.

Managed Aspects:

- **Application:** The software or application is fully managed by the SaaS provider. Users interact with the application without dealing with underlying complexities.
- **Data:** Data storage, backup, and management are handled by the SaaS provider, ensuring data integrity and security.
- **Runtime:** The runtime environment required for the application to function is managed by the SaaS provider.
- **Middleware:** Middleware components, if necessary for the application, are maintained and managed by the SaaS provider.
- **Operating System (OS):** The underlying operating system, including updates and security patches, is the responsibility of the SaaS provider.
- **Virtualization:** If virtualization is used, the provider manages the virtualization layer.
- **Server:** The physical or virtual servers hosting the application are fully managed.
- **Disk Space:** Storage and disk space provisioning and management are handled by the SaaS provider.
- **Network:** The network infrastructure and connectivity needed for the application are managed by the SaaS provider.

▼ **FaaS (Function as a Service):**

- **Example:**
  - AWS Lambda, Azure Functions, and Google Cloud Functions are examples of Function as a Service (FaaS) providers. These platforms empower developers to execute individual functions or code snippets in response to events without the need to manage the entire underlying infrastructure.
  - A practical example involves deploying a Link Cleaner API using FaaS, where the platform handles the execution of specific functions triggered by incoming events.
- **Managed Aspects:**
  - **Middleware:** Any necessary middleware components are managed and provided by the FaaS provider.
  - **Runtime:** The runtime environment required for executing functions is fully managed by the FaaS provider.
  - **Operating System (OS):** The underlying operating system, including updates and security patches, is managed by the FaaS provider.
  - **Virtualization:** If virtualization is utilized, the provider manages the virtualization layer.
  - **Server:** The physical or virtual servers supporting the functions are fully managed by the FaaS provider.
  - **Disk Space:** Storage and disk space provisioning, as required for the functions, are handled by the FaaS provider.
  - **Network:** The network infrastructure needed for the functions, including connectivity, is managed by the FaaS provider.
- **Unmanaged Aspects:**
  - **Functions (Web Controller in Java):** Developers maintain control over the specific functions or code snippets that are executed. In the provided example, the Web Controller in Java represents the custom

code written by developers to perform a specific function in response to events.

▼ Deployment Models:

- **Public:**
  - Public cloud deployment involves hosting applications and services on cloud infrastructure that is accessible to the general public. The resources are owned and operated by a third-party cloud service provider.
- **Private:**
  - Private cloud deployment involves hosting applications and services on cloud infrastructure that is dedicated to a single organization. The resources can be owned and operated by the organization or a third-party provider.
- **Hybrid:**
  - Hybrid cloud deployment involves a combination of both public and private cloud resources. It allows data and applications to be shared between them. This model provides greater flexibility and optimization of existing infrastructure.
- **Community:**
  - Community cloud deployment involves infrastructure shared by several organizations with common concerns (e.g., security, compliance). It is suitable for communities of users with similar needs.

▼ Benefits:

- **Pay-as-you-go:**

    - Cloud services typically follow a pay-as-you-go model, where organizations pay for the resources and services they use. This can lead to cost savings as compared to traditional infrastructure.
- **Low Maintenance:**
    - Cloud service providers handle infrastructure maintenance tasks, including hardware upgrades, security patches, and system updates. This reduces the burden on organizations for maintaining and managing physical infrastructure.
- **Global Accessibility:**
    - Cloud services are accessible from anywhere with an internet connection. This enables global accessibility, allowing users to access applications and data from various locations.
- **High Flexibility:**
    - Cloud environments offer high flexibility in terms of scaling resources up or down based on demand. This agility allows organizations to adapt quickly to changing business needs.

▼ Drawbacks:

- **Complexity:**
    - Implementing and managing cloud solutions can be complex, especially for large-scale deployments. Integration with existing systems, data migration, and ensuring security add to the complexity.
- **Less Control:**
    - Organizations may have less direct control over the underlying infrastructure in a cloud environment. This lack of control can be a concern for certain businesses, especially those with specific compliance or regulatory requirements.

Container Technology

- **What is it?**
  - A standardized software unit for application deployment.
  - Allows running applications independently of the host OS.
- **Pros and Cons (Virtual Servers vs. Containers):**
  - Vertical scaling (VMs) vs. Horizontal scaling (Containers)
  - Resource consumption
  - Portability
  - Efficiency
- **Container Runtimes:**
  - Docker, Containerd

Container Orchestration

- **Why Needed?**
  - Provisioning, deployment, scaling, load balancing, monitoring, and configuration of containerized applications.
- **Technologies:**
  - Kubernetes, Docker Swarm, Amazon ECS
- **Scaling:**
  - Vertical (Scale up): Increase resources per container.

- Horizontal (Scale out): Add more instances of containers.

Kubernetes

▼ YAML files

```
# Config.yaml example
apiVersion: v1
kind: ConfigMap
metadata:
  name: mongo-config
data:
  mongo-url: mongo-service
```

```
# Secret.yaml example
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
type: Opaque
data:
  mongo-user: bW9uZ291c2Vy
  mongo-password: bW9uZ29wYXNzd29yZA==
```

```
# Deplyoment.yaml example
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment
  labels:
```

```
    app: webapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: nanajanashia/k8s-demo-app:v1.0
        ports:
        - containerPort: 3000
        env:
        - name: USER_NAME
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: mongo-user
        - name: USER_PWD
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: mongo-password
        - name: DB_URL
          valueFrom:
            configMapKeyRef:
              name: mongo-config
              key: mongo-url
---
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
```

```
spec:
  type: NodePort
  selector:
    app: webapp
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
      nodePort: 30100
```

YAML Breakdown

- `apiVersion`: Specifies the Kubernetes API version. In this case, it's using the "apps/v1" API version, which is appropriate for Deployments.
- `kind`: Specifies the type of Kubernetes resource. Here, it's "Deployment," indicating that this configuration file is defining a Deployment.
- `spec`: This section defines the desired state of the Deployment.
- `replicas: 3`: Specifies that you want to run three replicas of your application.
- `selector`: Describes the selector to match pods managed by this Deployment.
- `matchLabels`: Specifies the labels that the Replica Set created by the Deployment should use to select the pods it manages. In this case, pods with the label `app: example` are selected.
- `template`: Defines the pod template used for creating new pods.
- `metadata`: Contains the labels to apply to the pods created from this template. In this case, the pods will have the label `app: example`.
- `spec`: Describes the specification of the pods.
- `containers`: This section specifies the containers to run in the pod.
- `name: example-container`: Assigns a name to the container.
- `image: example-image`: Specifies the Docker image to use for this container.
- `ports`: Defines the ports to open in the container.
- `containerPort: 8080`: Indicates that the container will listen on port 80.

Docker

▼ YAML file

```
version: '3'
services:
  app:
    image: node:latest
    container_name: app_main
    restart: always
    command: sh -c "yarn install && yarn start"
    ports:
      - 8000:8000
    working_dir: /app
    volumes:
      - ./:/app
    environment:
      MYSQL_HOST: localhost
      MYSQL_USER: root
      MYSQL_PASSWORD:
      MYSQL_DB: test
  mongo:
    image: mongo
    container_name: app_mongo
    restart: always
    ports:
      - 27017:27017
    volumes:
      - ~/mongo:/data/db
```

```
volumes:
  mongodb:
```

YAML file Breakdown

- `version` refers to the docker-compose version (Latest 3)
- `services` defines the services that we need to run
- `app` is a custom name for one of your containers
- `image` the image which we have to pull. Here we are using `node:latest` and `mongo`.
- `container_name` is the name for each container
- `restart` starts/restarts a service container
- `port` defines the custom port to run the container
- `working_dir` is the current working directory for the service container
- `environment` defines the environment variables, such as DB credentials, and so on.
- `command` is the command to run the service

▼ Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 5000 available to the world outside this container
EXPOSE 5000

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Dockerfile Breakdown

- `FROM python:3.8-slim`: Use the official Python 3.8 image as the base image.
- `WORKDIR /app`: Set the working directory inside the container to `/app`.
- `COPY . /app`: Copy the current directory contents into the container at `/app`.
- `RUN pip install --no-cache-dir -r requirements.txt`: Install Python dependencies specified in `requirements.txt`. This assumes you have a file named `requirements.txt` in your project with the necessary dependencies.
- `EXPOSE 5000`: Expose port 5000 to the outside world. This is the port on which the Flask application will run.
- `ENV NAME World`: Set an environment variable named `NAME` with the default value "World".
- `CMD ["python", "app.py"]`: Specify the command to run when the container starts. In this case, it runs the `app.py` script using Python.

12-Factor Apps

- **Key Factors:**
  - ▼ (1) **Codebase:**
    - A single codebase tracked in version control, with multiple deployments running the same code. This ensures consistency and ease of management.
    - Example:
      - Tracking in version Control

```
git clone https://github.com/isaaclins/isaaclins.git
```

▼ (2) **Dependencies:**

- Explicitly declare and isolate dependencies. The application should rely on declared dependencies and not make assumptions about the system it runs on.
- Example:
  - Explicitly declare and isolate Dependencies

```
import discord
import os
import pyautogui
import asyncio
import sys
import requests
import subprocess
import string
from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume
from datetime import datetime
from uuid import getnode as get_mac
```

▼ (3) **Configuration:**

- Store configuration in the environment. Avoid hard-coding configuration settings in the code to make the application easily configurable across different environments.
- Example:

- Store configurations in environment variables.

```
export API_KEY=your_api_key
```

▼ (4) **Supporting services:**

- Treat backing services (databases, caches, etc.) as attached resources. The application should be designed to work seamlessly with these services, which can be swapped or upgraded without affecting the app.
- Example:

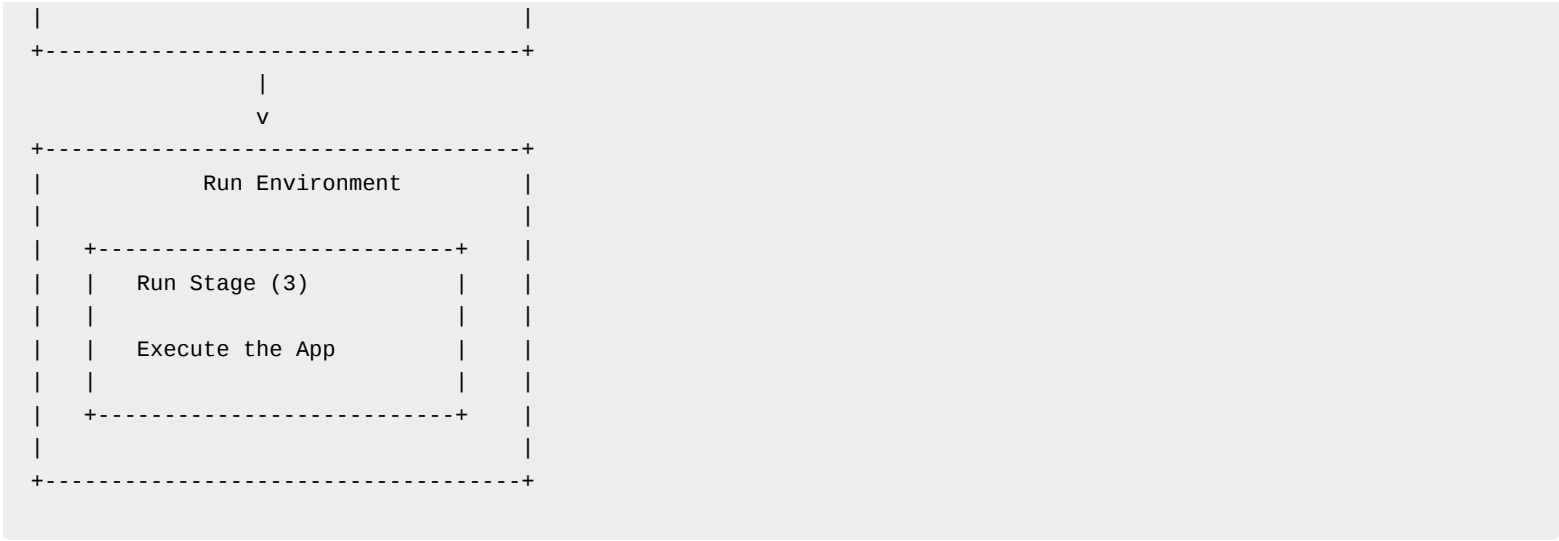
```
+-----------------------+       +-----------------------+
|    Web Application    |       |    Web Application    |
|                       |       |                       |
|   +---------------+   |       |   +---------------+   |
|   |   Database 1  |   |       |   |   Database 2  |   |
|   +---------------+   |       |   +---------------+   |
|                       |       |                       |
+-----------------------+       +-----------------------+

Web Application is connected to Database 1 by default.
Database 1 goes down.

+-----------------------+       +-----------------------+
|    Web Application    |       |    Web Application    |
|                       |       |                       |
|   +---------------+   |       |   +---------------+   |
|   |   Database 1  |   |       |   |   Database 2  |   |
|   +---------------+   |       |   +---------------+   |
|           |           |  ==>  |                       |
|           v           |       |                       |
|   +---------------+   |       |   +---------------+   |
|   |   Database 2  |   |       |   |   Database 2  |   |
```

```
|   +---------------+   |       |   +---------------+   |
|                       |       |                       |
+-----------------------+       +-----------------------+
The Web Application switches to Database 2, because Database 1 is down.
```

▼ (5) **Build, release, run:**

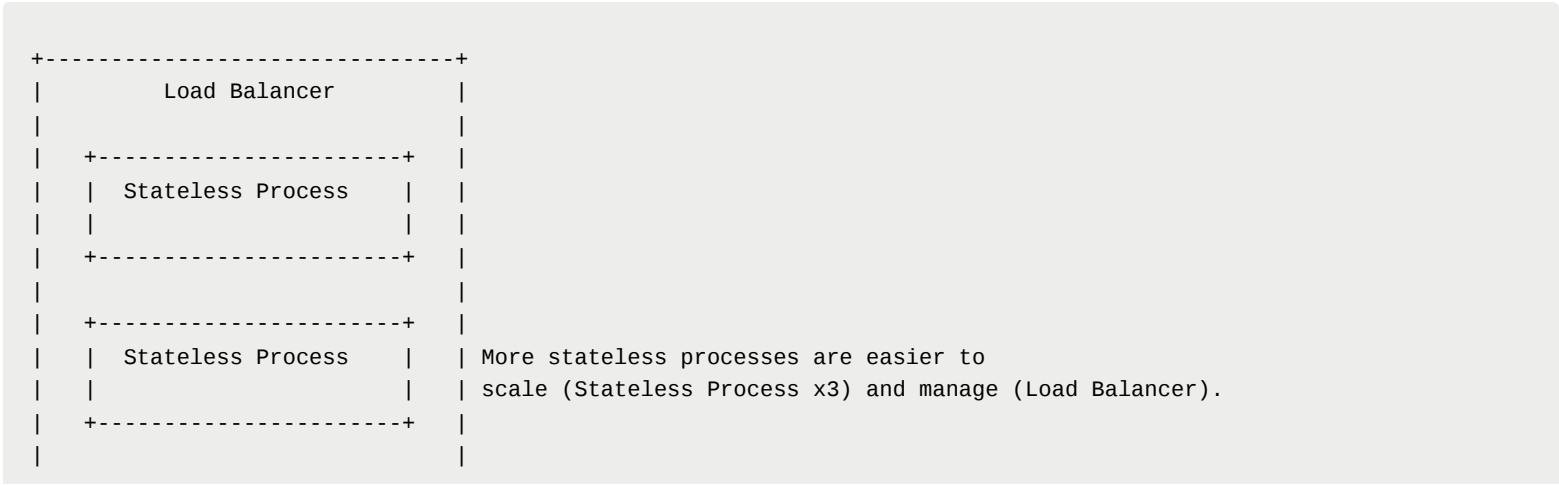
- Strictly separate the build and run stages. The build stage creates a deployable artifact, the release stage combines the build with the environment's configuration, and the run stage executes the app.
- Example:

```
+-------------------------------------+
|            Build Pipeline          |
|                                     |
|   +---------------------------+     |
|   |   Build Stage (1)         |     |
|   |                           |     |
|   |   Create Artifact         |     |
|   |                           |     |
|   +---------------------------+     |
|                                     |
+-------------------------------------+
                 |
                 v
+-------------------------------------+
|            Release Pipeline        |
|                                     |
|   +---------------------------+     |
|   |   Release Stage (2)       |     |
|   |                           |     |
|   |   Combine with Config     |     |
|   |                           |     |
|   +---------------------------+     |
```



▼ (6) **Processes:**

- Execute the app as one or more stateless processes. Stateless processes are easier to scale and manage.
- Example:

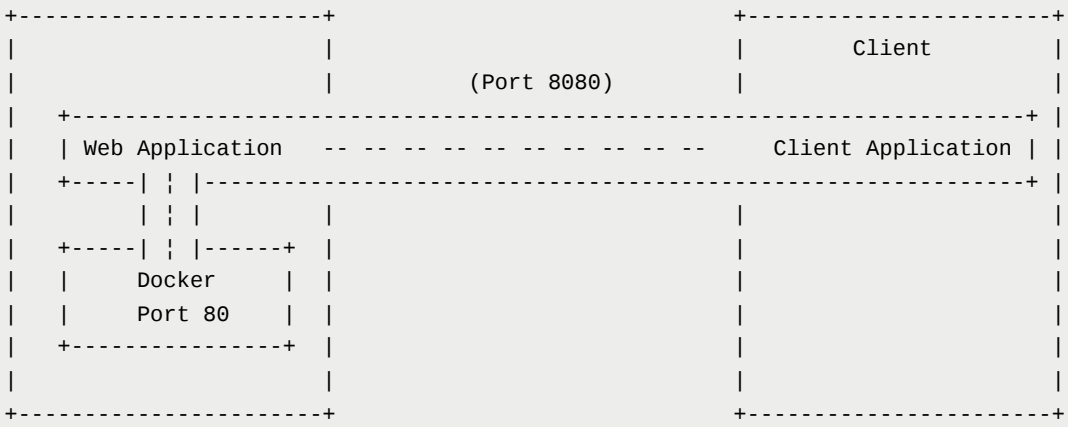


```
|   +-----------------------+   |
|   |  Stateless Process    |   |
|   |                       |   |
|   +-----------------------+   |
|                               |
+-------------------------------+
```

▼ (7) **Port binding:**

- Export services via port binding. The app should be self-contained and export services through a port, making it accessible to other services.
- **Example:**
  - Export services by binding ports.

```
docker run -p 8080:80 your_image
```



▼ (8) **Concurrency:**

- Scale out via the process model. The application should be designed to scale horizontally by adding more instances of the app.

▼ (9) **Disposability:**

- Maximize robustness with fast startup and graceful shutdown. Apps should start quickly and shut down gracefully, making them more resilient to failures and easier to manage.

▼ (10) **Dev-prod parity:**

- Keep development, staging, and production environments as similar as possible. Reducing the differences between these environments helps prevent bugs and reduces deployment issues.

▼ (11) **Logs:**

- Treat logs as event streams. Logs should provide visibility into the state of the application, and treating them as event streams enables easier analysis.

▼ (12) **Admin processes:**

- Run admin/management tasks as one-off processes. Administrative tasks should be separate from the normal application processes and run as isolated, short-lived tasks.
- Example:
  - Treat admin tasks as one-off processes.

```
heroku run rake db:migrate
```

- **Definition:**
  - Methodology for building Software-as-a-Service (SaaS) apps with factors like codebase, dependencies, configuration, etc.