Introduction to PWA

1. PWAs are web applications that use modern web capabilities to **deliver an app-like experience( Access and provide features that can be only be provided with apps).**
2. They **work offline, load quickly**, and can be **added to the home screen** of devices.
3. **Key Features**:
   * Responsive design
   * Offline capability
   * App-like experience
   * Safe (served over HTTPS)
4. A **Service Worker** in a Progressive Web App (PWA) is like a background helper that runs separately from your web page (Independent of web page). It’s **designed to manage** tasks **that keep your app working smoothly even when you’re offline** or **have a poor internet connection.**
   * It doesn’t use memory or CPU constantly. Instead, it wakes up only when needed, like when a network request is made, or a push notification arrives. (Event driven)
   * We can stop a service worker by unregistering it from browser.

**Q Can a progressive web app run withou service worker ?**

**Yes**, a Progressive Web App (PWA) can run without a Service Worker, but it will miss out on some of the key features that make PWAs special.

**What You Lose Without a Service Worker:**

1. **Offline Functionality**: Without a Service Worker, your app won't be able to provide offline access to cached content. This means that if a user loses their internet connection, they may not be able to use the app or see previously loaded data.
2. **Background Sync**: The app won’t be able to synchronize data in the background. This means that if a user performs an action while offline, such as submitting a form, it may not be sent to the server until the user is back online, and this process won't be automated.
3. **Push Notifications**: You won’t be able to send push notifications. Push notifications are a way to engage users even when the app is not open, which is managed by the Service Worker.
4. **Performance Enhancements**: The app won’t benefit from performance optimizations like caching resources to speed up loading times. Without caching, every request must go to the server, which can slow down your app and increase server load.
5. **Customizable Network Handling**: You’ll lose the ability to intercept and handle network requests in a customized way. This means that you can’t implement custom strategies for managing requests and responses.

**What You Still Get:**

* **Basic Web Functionality**: The app will still work as a regular web application. Users can navigate through pages, interact with content, and use features as long as they have an internet connection.
* **Add to Home Screen**: Users can still install the app to their home screen and access it in a full-screen mode, which is one of the core features of PWAs.

**Things to Set up the PWA project**

**Create a Project Folder**:

* Create a new folder on your computer named pwa-hello-world.

**Create Basic Files**:

* Inside the folder, create the following files:
  + index.html
  + style.css
  + app.js => **This file registers the Service Worker to Browser**
  + service-worker.js => **This file will handle caching for offline use.**
  + manifest.json => **The manifest file defines how your PWA behaves when installed.**

**Google Chrome**

1. **Clear Cache and Cookies**:
   * Open Chrome.
   * Click the three vertical dots (menu) in the top right corner.
   * Go to **More tools** > **Clear browsing data**.
   * In the pop-up, select the time range (e.g., "All time") to ensure everything is cleared.
   * Check **Cookies and other site data** and **Cached images and files**.
   * Click **Clear data**.
2. **Clear Service Workers**:
   * Open Chrome Developer Tools (press Ctrl+Shift+I or Cmd+Option+I on Mac).
   * Go to the **Application** tab.
   * In the left sidebar, select **Service Workers**.
   * Click **Unregister** for any service workers listed.
3. **Clear Local Storage and IndexedDB**:
   * In the **Application** tab, select **Local Storage** and **IndexedDB** in the left sidebar.
   * Right-click on the items and select **Clear** or **Delete**.
4. **Clear Web Storage**:
   * In the **Application** tab, select **Web Storage**.
   * Right-click on each domain listed and select **Clear** or **Delete**.

**Base Algorithm**

**Approach 1**

1. **Teacher Interaction:** When the user clicks the "Send Signal" button, the main.js script triggers a function that uses the Geolocation API to get the user's current position (latitude and longitude).
2. **Student Device Registration:** When the PWA is loaded in the student device, it checks if the user has already subscribed to push notifications. If not, it prompts the user to allow notifications. If permission is granted, the device subscribes to push notificationsand sends subscription info to the backend.
   1. For each device that registers for push notifications, you should store its location (latitude and longitude) along with the subscription details.
   2. When a signal is sent, you need to calculate the distance between the signal's location and the location of each registered device. (using **Haversine formula**)
   3. Based on the distance calculation, filter out the devices that fall within the specified radius.
3. **Sending a Signal :** When the teacher clicks the "**Send Signal**" button is clicked, the PWA sends a POST request to the backend at /send-signal, including the latitude and longitude. The backend processes it and determines which devices are within the specified radius.
4. **Sending Push(Signal) Notification :** When the backend sends a push notification, it uses **pywebpush** to dispatch the notification to all subscribed devices. The **subscription\_info** includes details about each device’s subscription.
5. **Handling Push Notification:** When a push notification is received, the service worker (registered via sw.js) handles the push event and displays the notification to the user.

**Note: Push Notifications with Location-Based Filtering or WebSockets or SSE is better for dealing with network delays and real-time interaction compared to Geolocation API.**

**Note:** Push notifications use **a targeted approach**, sending messages to specific devices or users who have opted in. They **do not involve traditional broadcasting technology,** which is used for mass communication. Instead, push notifications rely on a subscription model where messages are sent to devices based on their subscription details. However, **you can achieve a broadcasting-like effect by sending notifications to multiple devices that meet certain criteria,** such as being within a specific geographical area.

**Disadvantage: A continuous registration of students (with location) is required on a timely basis. Everything will be stored in one database so for every teacher -every student distance should be calculated. Involves use of multiple servers which will slow down the process.**

**Approach 2: BLE (Bluetooth low energy)**

1. **Scan for All Nearby Devices :** First, you need to scan for all nearby BLE devices. Each device should be advertising a service or characteristic that you can filter for.
2. **Connect to Each Device :** Once you have a list of nearby devices, connect to each one. This can be done in parallel to save time.
3. **Send the Message to Each Device :** After connecting, send the message to each device.

**Disadvantage:** For resending the acknowledgement each device needs to be connected to the central device which will lead to concurrency errors. Requires UUIDs which are difficult to determine.

**Approach 3: Web server**

1. Teacher will enter details of the lecture in the form and the session link will be transferred only to the required students of the class.
2. The students upon receiving the link will authenticate themselves for attendance.
3. The data will get transferred to the teacher in the form of report.
4. The teacher can manually alter the report if required

**Disadvantage: Requires some of the unique identity of the device to be stored in the database**

**Note :** BLE does not natively support broadcasting a message to multiple devices directly from a central device (like a PWA) in the way that traditional messaging systems do.

**Approach 4 : web scoket**

1. **Not following the push notification approach because it is too complex to implement and just helping to send the notifications once**
2. **Not Following the Bluetooth approach because range depends on the devices (limited range) and each device needs to be connected individually one after another to send messages which is time-consuming**
3. **Implementing the Web-Socket approach because it helps in forming the initial connection between the students and teachers (Connects large number of clients at the same time), helps in direct broadcasting of the messages to all the clients at the same time with the help of mapping.**

**Web socket base code**

1. **wss.on('connection', ...):** This event handler is triggered whenever a new WebSocket connection is established.
2. **URLSearchParams(req.url.slice(1)):** Parses the query string from the URL to extract parameters like role and teacherId.
3. **role and teacherId:** Determine whether the connection is from a teacher or a student and which teacher the student belongs to.
4. **handleTeacher(ws, teacherId) and handleStudent(ws, teacherId):** Call appropriate functions to handle teacher or student connections.

**Why to connect HTTP server with the Ws server ?**

1. The WebSocket protocol starts with an HTTP handshake to establish a connection. The client initially makes an HTTP request with an Upgrade header to switch the connection to WebSocket. An HTTP server is required to handle this handshake.
2. Most of the connection between ws client occur through http
3. By attaching the WebSocket server to the existing HTTP server, both HTTP and WebSocket traffic can be handled on the same port. This reduces the number of ports and resources required.
4. We can declare an independent web socket server but it is better to connect it with the http (otherwise we have to maintain separate http server)

**Diffrence between wss.on() and ws.on()**

1. **wss.on()** is used to handle events on the WebSocket Server (wss). It manages events related to the server as a whole, such as handling new connections or server-wide errors.
   1. 'connection': This event is triggered when a new WebSocket connection is established. It’s the primary way to handle incoming WebSocket connections and set up handlers for individual WebSocket clients.
   2. 'close': This event is triggered when the WebSocket server itself closes.
   3. 'error': This event is triggered when an error occurs on the WebSocket server.
2. **ws.on()** is used to handle events on individual WebSocket connections (ws). It manages events related to specific client connections, such as processing messages from that client or handling client-specific disconnections.
   1. 'message': This event is triggered when a message is received from the WebSocket client.It’s used to process incoming data from that particular client.
   2. 'close': This event is triggered when the WebSocket connection from the client is closed.
   3. 'error': This event is triggered when an error occurs on that specific WebSocket connection.

**Working**

1. Students fill the form and after clicking the submit button =>
2. websocket connection is initialized
3. the teacher initials and user type(student,teacher) are passed to the server
4. the server sends the acknowledgement of receiving the data
5. The Webserver checks type of user
6. If it is student then it places the student connection object in a set corresponding to its teacher initials in a dictionary (,aping teacher initials with the set of student connection object)
7. If it is teacher then the server sends an ‘Attendance started’ message to all the students corresponding the teacher initials (or t\_id)
8. Afetr students receive that message then the an authentication interface corresponding to the respective class will open on the screen

**MySQL Database Structure**

CREATE TABLE student\_details (

UID VARCHAR(255) PRIMARY KEY,

roll\_no INT NOT NULL,

name VARCHAR(255) NOT NULL

);

CREATE TABLE lecture\_details (

l\_id VARCHAR(255) PRIMARY KEY,

lecture\_name VARCHAR(255) NOT NULL,

branch VARCHAR(255) NOT NULL,

division VARCHAR(255) NOT NULL,

teacher\_initials VARCHAR(255) NOT NULL,

date DATE NOT NULL,

);

CREATE TABLE attendance ( UID VARCHAR(255), l\_id VARCHAR(255),

PRIMARY KEY (UID, l\_id), FOREIGN KEY (UID) REFERENCES student\_details(UID) ON DELETE CASCADE ON UPDATE CASCADE,

FOREIGN KEY (l\_id) REFERENCES lecture\_details(l\_id) ON DELETE CASCADE ON UPDATE CASCADE );

For connecting to mysql server using Azure SSL  
“**mysql -h classconnect-database.mysql.database.azure.com -P 3306 -u urmil -p --ssl**

**Enter password:** “

For disconnecting the connection

**QUIT;** or **EXIT;**

**For deleting data from tables**

Delete from student\_details ;

Delete from lecture\_details;

Delete from attendance;

Database connectivity in flask

#similar to just a single conection but it will allow other routes to use the connection simultaneausly

db\_password = os.environ.get('MYSQL\_PASSWORD')

pool = pooling.MySQLConnectionPool(

    pool\_name="mypool",

    pool\_size=5,

    user="urmil",

    password=db\_password,

    host="classconnect-database.mysql.database.azure.com",

    port=3306,

    database='classconnect',

    ssl\_ca="/certificate/DigiCertGlobalRootG2.crt.pem",

    ssl\_disabled=False

)

Setting Environment variables

set MYSQL\_USER=urmil

set MYSQL\_PASSWORD=class@1project

set MYSQL\_HOST=classconnect-database.mysql.database.azure.com

set MYSQL\_PORT=3306

set MYSQL\_DATABASE=classconnect

**For inserting data in the student\_details database using /update: use the JSON containing the list of students**

Firewall and protection

* There is also a firewall around the mysql server that only allow my azure server to communicate with the mysql server and block others
* The MySQl server in azure requires SSL certificate by default that helps client to ensure that the responding server is legitimate and also encrypts the transmission

“{

  "students": [

    {

      "UID": "21\_COMPB01\_25",

      "roll\_no": 1,

      "name": "John Doe"

    },

    {

      "UID": "21\_COMPB02\_25",

      "roll\_no": 2,

      "name": "Jane Smith"

    },

    {

      "UID": "21\_COMPB03\_25",

      "roll\_no": 3,

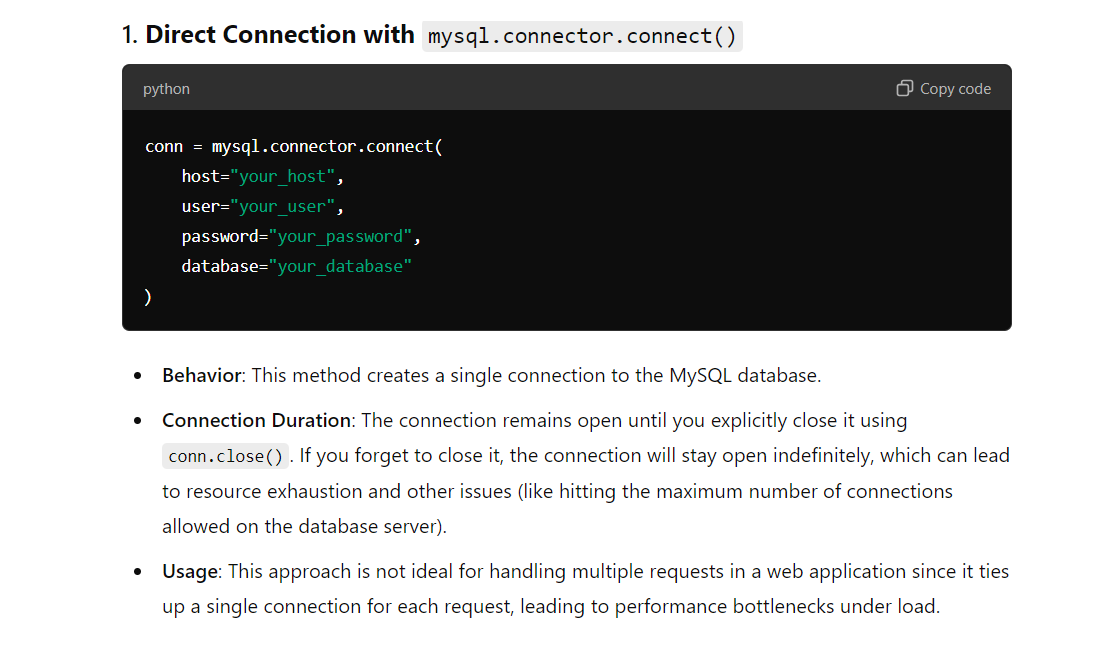
      "name": "Alice Johnson"

    }

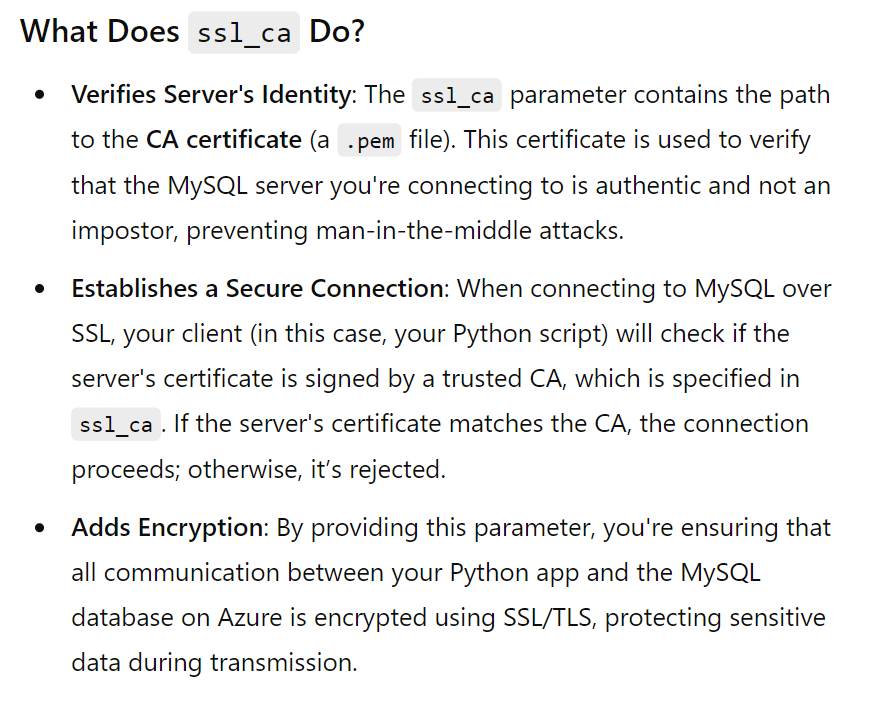
  ]

}

“



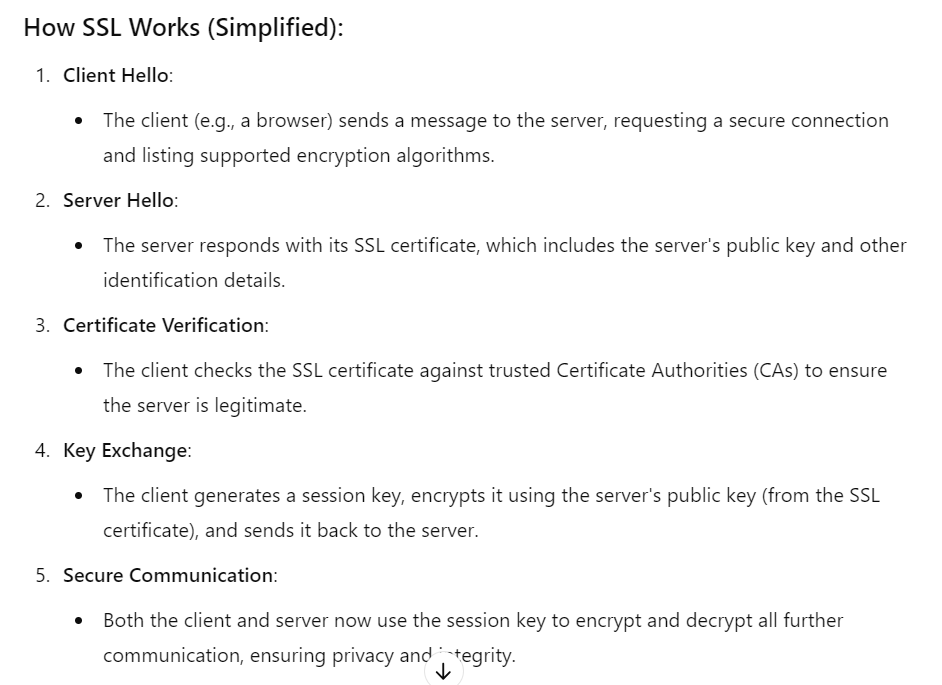
Use of SSL\_CA in conection



**Deploying Flask server on Azure**

1. Add a requirements.txt in root folder (not in the server folder). (In requiremenst.txt manually add requirements and specify only the versions that are currently running in the pc).
2. Add the .yml file (Automatically added by the azure) (Don’t Modify it)
3. Specify the startup command in the azure portal (if not then specify it in .yml file)
4. Just deploy normally and see the possible error in deployment logs or Log streams
5. While making requests using ‘postman’ use https

**Why HTTP changes to https on azure**

1. If you use **Azure App Service** to deploy your Flask app, HTTPS is enabled by default, and Azure manages the SSL certificates for you. You can access your app using <https://yourapp.azurewebsites.net>.
2. **SSL (Secure Sockets Layer)** is a security technology that establishes an encrypted link between a web server and a browser. It ensures that all data passed between the server and the client remains private and secure. SSL has now been largely replaced by **TLS (Transport Layer Security)**, but the term "SSL" is still commonly used to refer to this process
3. 

**Setting up a Load Balancer**

Load Balancer mechanism is generally available for ‘Virtual Machines’ in azure while for ‘App service’ the ‘Application Gateway’ is used that inherits the functionality of load balancing

**Introduction**

An **Application Gateway** is a web traffic load balancer that enables you to manage traffic to your web applications. Here are some key functions and features:

1. **Load Balancing**: Distributes incoming application traffic across multiple servers to ensure no single server is overwhelmed, improving performance and reliability.
2. **Web Application Firewall (WAF)**: Protects applications from common web vulnerabilities (e.g., SQL injection, cross-site scripting) by inspecting incoming traffic and blocking malicious requests.
3. **SSL Termination**: Offloads SSL encryption/decryption tasks from the backend servers, reducing their processing load and improving performance.
4. **URL-Based Routing**: Directs incoming requests to different backend pools based on the URL path, allowing for efficient resource usage and organization of services.
5. **Session Affinity**: Ensures that a user's session is directed to the same backend server for the duration of their session, improving user experience.
6. **Health Monitoring**: Continuously checks the health of the backend servers and routes traffic only to those that are healthy, ensuring high availability.
7. **Customizable Rules**: Allows users to define routing rules based on various criteria, enabling tailored handling of requests.
8. **Autoscaling**: Automatically adjusts resources based on traffic demand, ensuring optimal performance during peak loads.

-----------------------------------------------------------------------------------------------------------------

**Example Working**

Let’s go through an example scenario to illustrate how an **Application Gateway** works as a load balancer for a web application hosted on **Azure App Service**.

**Example Scenario**

**Web Application**: You have an online shopping application hosted on Azure App Service, called **ShopEasy**. The app has multiple instances to handle varying traffic.

**Architecture Overview**:

* **Frontend**: Users access the app via a domain name (e.g., www.shopeasy.com).
* **Backend**: The app is hosted on Azure App Service with multiple instances for scaling.
* **Application Gateway**: Sits in front of the App Service to manage traffic and load balance requests.

**Step-by-Step Flow**

1. **User Request**:
   * A user opens a web browser and types www.shopeasy.com.
   * The DNS resolves this domain to the public IP address of the Application Gateway.
2. **Application Gateway Receives the Request**:
   * The Application Gateway receives the incoming request.
   * It checks its configuration to determine how to handle the request.
3. **Routing the Request**:
   * The Application Gateway uses its **listener** configuration to understand what backend to route the request to.
   * It checks the **URL path** and decides which **backend pool** to use. For example, if the user requests /products, it routes to the backend pool configured for product services.
4. **Load Balancing**:
   * The Application Gateway uses its load balancing algorithm (e.g., Round Robin or Least Connections) to select one of the active instances of the App Service to handle the request.
   * Suppose it selects **Instance 1** of the App Service.
5. **Forwarding the Request**:
   * The Application Gateway forwards the request to **Instance 1** of the App Service using the **HTTP settings** defined (e.g., HTTPS on port 443).
   * This request is made directly to the internal IP of the App Service instance.
6. **Processing the Request**:
   * **Instance 1** processes the request, accesses the database if needed, and generates a response (e.g., a list of products).
7. **Response to Application Gateway**:
   * The response is sent back from **Instance 1** to the Application Gateway.
8. **Final Response to User**:
   * The Application Gateway receives the response from the backend and forwards it to the user’s browser.
   * The user sees the list of products rendered on their screen.

**Health Monitoring**

* **Continuous Health Checks**: The Application Gateway performs health probes on the App Service instances.
  + If **Instance 1** goes down (e.g., it becomes unresponsive), the Application Gateway stops sending requests to it and routes traffic only to the healthy instances (e.g., **Instance 2**).

**SSL Offloading**

* If your application uses HTTPS, the Application Gateway can terminate the SSL connection. This means it decrypts the HTTPS traffic before sending it to the backend, reducing the load on your App Service instances.

-----------------------------------------------------------------------------------------------------------------

**Internal management of Routes**

**What are Routing Rules?**

**Routing rules** are a set of instructions that the Application Gateway uses to determine how to handle incoming requests. They specify which backend pool (group of servers) should receive the request based on the request's URL. This allows you to direct traffic to different services or parts of your application based on the specific path in the URL.

**Example Scenario Explained**

Imagine you have a web application for an online store called **ShopEasy**. This application has different sections, each with its own functionality:

* **Homepage**: /
* **Products Page**: /products
* **Shopping Cart**: /cart
* **Checkout Page**: /checkout

Here’s how routing rules would work:

1. **User Request**:
   * A user visits the URL www.shopeasy.com/products.
2. **Application Gateway Receives the Request**:
   * The Application Gateway receives the request for the /products page.
3. **Routing Decision**:
   * The Application Gateway checks its routing rules to determine how to handle the request.
   * It has a rule set up that says:
     + If the request URL path starts with /products, route it to the **Products Backend Pool**.
4. **Backend Pool**:
   * The **Products Backend Pool** might contain several instances of the App Service specifically configured to handle product-related requests. For example, it could include:
     + Instance A (handling product listings)
     + Instance B (handling product details)
5. **Forwarding the Request**:
   * The Application Gateway then selects one of the instances in the **Products Backend Pool** (using load balancing algorithms) and forwards the request to that instance.

**Summary of Routing Rules in this Example**

* **Rule for Products**:
  + **Condition**: If the URL path is /products
  + **Action**: Route the request to the **Products Backend Pool**.
* **Additional Rules**:
  + You can set up other rules for different URL paths, for example:
    - **Rule for Cart**:
      * If the URL path is /cart, route it to the **Cart Backend Pool**.
    - **Rule for Checkout**:
      * If the URL path is /checkout, route it to the **Checkout Backend Pool**.

-----------------------------------------------------------------------------------------------------------------

**Detailed Explanation of Internal Management of Routes**

a **Flask server** and a **WebSocket server**, and only one instance of each, you can still set up an **Application Gateway** to manage traffic between them based on the type of requests. Here’s how you can configure it:

**Scenario Overview**

* **Flask Server**: Handles HTTP requests (e.g., REST APIs, web pages).
* **WebSocket Server**: Manages real-time communication (e.g., chat, notifications).

**Routing Setup**

1. **Application Gateway Configuration**:
   * You will set up routing rules in the Application Gateway to direct traffic to either the Flask server or the WebSocket server based on the type of request.
2. **Define Backend Pools**:
   * Create two backend pools in the Application Gateway:
     + **Flask Backend Pool**: Points to your Flask server.
     + **WebSocket Backend Pool**: Points to your WebSocket server.
3. **Routing Rules**:
   * Set up routing rules to decide how to handle incoming requests:
     + **HTTP Requests (Flask Server)**: For example, requests to paths starting with /api or /web can be routed to the Flask backend pool.
     + **WebSocket Requests (WebSocket Server)**: For WebSocket connections, you can use a path like /ws to direct traffic to the WebSocket backend pool.

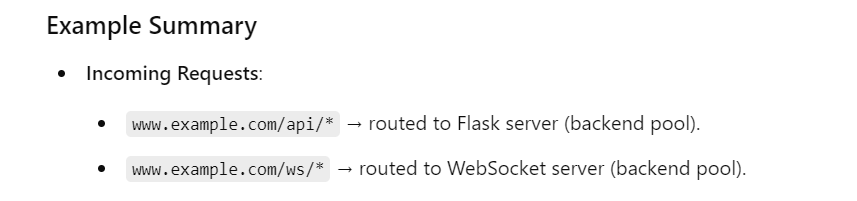
**Example Configuration**

Here’s a step-by-step breakdown of how this would look:

1. **Frontend IP Configuration**:
   * Configure the frontend IP for the Application Gateway. This will be the public IP address through which users will access your application.
2. **Create Backend Pools**:
   * **Flask Backend Pool**:
     + Add the Flask server instance (IP address or FQDN) to this pool.
   * **WebSocket Backend Pool**:
     + Add the WebSocket server instance (IP address or FQDN) to this pool.
3. **Create Routing Rules**:
   * **Flask Routing Rule**:
     + **Listener**: Listens on port 80 (HTTP) or port 443 (HTTPS).
     + **Condition**: URL path starts with /api or /web.
     + **Action**: Route to the **Flask Backend Pool**.
   * **WebSocket Routing Rule**:
     + **Listener**: Listens on the same ports (80 or 443).
     + **Condition**: URL path starts with /ws.
     + **Action**: Route to the **WebSocket Backend Pool**.

**Traffic Flow**

1. **User Makes an HTTP Request**:
   * A user accesses www.example.com/api/products.
   * The Application Gateway checks the routing rules and sees that the request matches the condition for the Flask server.
   * The request is forwarded to the Flask server.
2. **User Initiates a WebSocket Connection**:
   * A user connects to www.example.com/ws/chat.
   * The Application Gateway checks the routing rules and finds that this matches the WebSocket server's condition.
   * The connection request is forwarded to the WebSocket server.



-----------------------------------------------------------------------------------------------------------------

**Scaling the number of instances of a web app**

When you scale your application and add more instances of your Flask or WebSocket servers in Azure, the **Application Gateway** will automatically manage the traffic distribution across those instances. Here’s how it works and what you need to consider:

**Automatic Management of Multiple Instances**

1. **Dynamic Backend Pool Updates**:
   * When you add new instances to your backend pools (Flask or WebSocket), you can simply update the backend pool configuration in the Application Gateway.
   * The Application Gateway automatically recognizes the new instances and starts routing traffic to them based on the load balancing algorithm you have configured (like **Least Connections**).
2. **Load Balancing**:
   * With multiple instances, the Application Gateway distributes incoming traffic among all available instances in the backend pool according to the load balancing rules.
   * If you’re using **Least Connections**, it will send new requests to the instance with the least number of active connections, ensuring even distribution.

**Step-by-Step Example**

**Initial Setup**

1. **Backend Pools Configuration**:
   * **Flask Backend Pool**:
     + Instance 1: flask-server-instance-1 (IP: 10.0.0.1)
   * **WebSocket Backend Pool**:
     + Instance 1: websocket-server-instance-1 (IP: 10.0.0.2)
2. **Application Gateway Configuration**:
   * **HTTP Settings** for Flask:
     + Name: FlaskHttpSettings
     + Load Balancing Algorithm: **Least Connections**
   * **HTTP Settings** for WebSocket:
     + Name: WebSocketHttpSettings
     + Load Balancing Algorithm: **Least Connections**
3. **Routing Rules**:
   * **Flask Routing Rule**: Matches /api/\* paths, routes to FlaskHttpSettings.
   * **WebSocket Routing Rule**: Matches /ws/\* paths, routes to WebSocketHttpSettings.

**Traffic Flow**

1. **User Requests**:
   * A user accesses an API endpoint: https://example.com/api/products.
   * The Application Gateway checks the routing rules, matches it to the Flask routing rule, and forwards the request to the Flask server instance (10.0.0.1).
2. **WebSocket Connection**:
   * Another user connects to: https://example.com/ws/chat.
   * The Application Gateway matches this to the WebSocket routing rule and forwards the connection request to the WebSocket server instance (10.0.0.2).

**Scaling Up**

**Step 1: Add Instances**

After some time, you observe an increase in traffic. To handle this, you decide to scale up:

* **Flask Server Instances**:
  + Instance 2: flask-server-instance-2 (IP: 10.0.0.3)
  + Instance 3: flask-server-instance-3 (IP: 10.0.0.4)
* **WebSocket Server Instances**:
  + Instance 2: websocket-server-instance-2 (IP: 10.0.0.5)
  + Instance 3: websocket-server-instance-3 (IP: 10.0.0.6)

**Step 2: Update Application Gateway**

1. **Update Backend Pools**:
   * Add flask-server-instance-2 and flask-server-instance-3 to the **Flask Backend Pool**.
   * Add websocket-server-instance-2 and websocket-server-instance-3 to the **WebSocket Backend Pool**.
2. **Health Probes**:
   * Ensure that health probes are set up for both backend pools. These probes will regularly check if each instance is healthy and responsive.

**Step 3: Traffic Management**

Now that you have three instances of each server:

1. **Load Balancing in Action**:
   * When a new request comes in for https://example.com/api/products:
     + The Application Gateway checks active connections on all Flask instances.
     + If flask-server-instance-1 is busy with 5 connections, flask-server-instance-2 has 2, and flask-server-instance-3 has 1, the gateway routes the new request to flask-server-instance-3 (the least loaded).
2. **WebSocket Load Balancing**:
   * For a WebSocket connection request to https://example.com/ws/chat, if all three WebSocket instances are healthy, the Application Gateway will also use the least connections algorithm to direct the request to the instance with the fewest active connections.

**Monitoring and Adjustments**

* **Azure Monitor**:
  + Monitor metrics such as CPU utilization, memory usage, and response times for both the Application Gateway and your backend instances.
* **Adjustments**:
  + If you find that flask-server-instance-1 consistently has the highest load, you might consider optimizing your application logic, redistributing workloads, or scaling further.

Creating a virtual network

Go to ‘Virtual Networks’ create one and then add a subnet in it for web apps(with delegation Microsoft.Web/serverfarms….) and another subnet for private endpoints(without delegation)

In the app service web app, in networking select the appropriate VNet and the subnet for the web app

We don’t have to connect the MySQL server to the network, we can just establish a private endpoint for web apps connected in the same networks to communicate with the server (The web apps and the endpoint may be in different subnets)

We have to create a private endpoint in same VNet and dedicated subnet

We have to add thet endpoint in ‘networking’ of mysql server

**Updating the WebSocket Code**

Original WebSocket code:

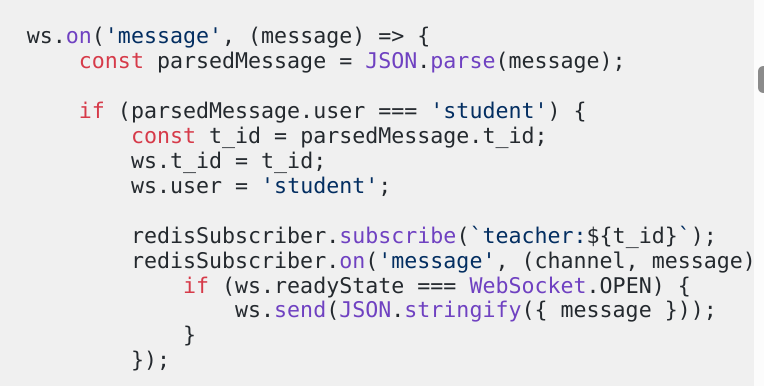
Uses a map of lists in which the key is teacher\_id and the values are the list of students' connections associated with the teacher\_id

When the student gets connected it checks if the map has the t\_id as a key, if not it creates one, and then its connection object is appended in the list corresponding to teaher\_id

When the teacher gets connected all the connection objects in the map gets he message one by one and then that list gets disconnected

**Why Redis Pub/Sub is infeccient to use :**

The placement of redisSubscriber.on('message', ...) inside or outside the WebSocket connection handler (wss.on('connection', ...)) changes how messages are handled for each



**Why It’s Inefficient**

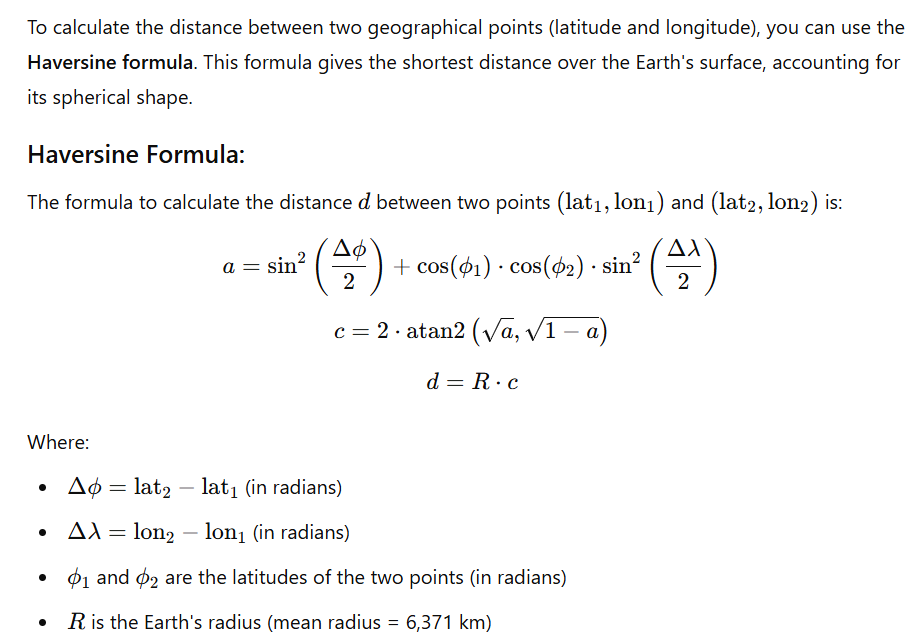
1. **Resource Overhead**: Each WebSocket client creates a separate on('message') listener, which takes up memory and processing power.
2. **Duplicate Processing**: When Redis publishes a message, every single listener responds to it, leading to duplicate message handling.
3. **Scaling Issues**: If more students subscribe, the number of listeners grows unnecessarily, degrading performance as each one is redundant for the same channel.

Typically, placing the redisSubscriber.on('message', ...) inside the handler is not recommended unless each WebSocket client needs a unique Redis subscription (for unique channels).



Disadvantage:  
Each client need to be sent one after the another in sequence like map

Calculating the diatance



**For Creating the docker redis app**

sudo apt update

sudo apt upgrade

sudo apt install docker.io

sudo systemctl enable --now docker

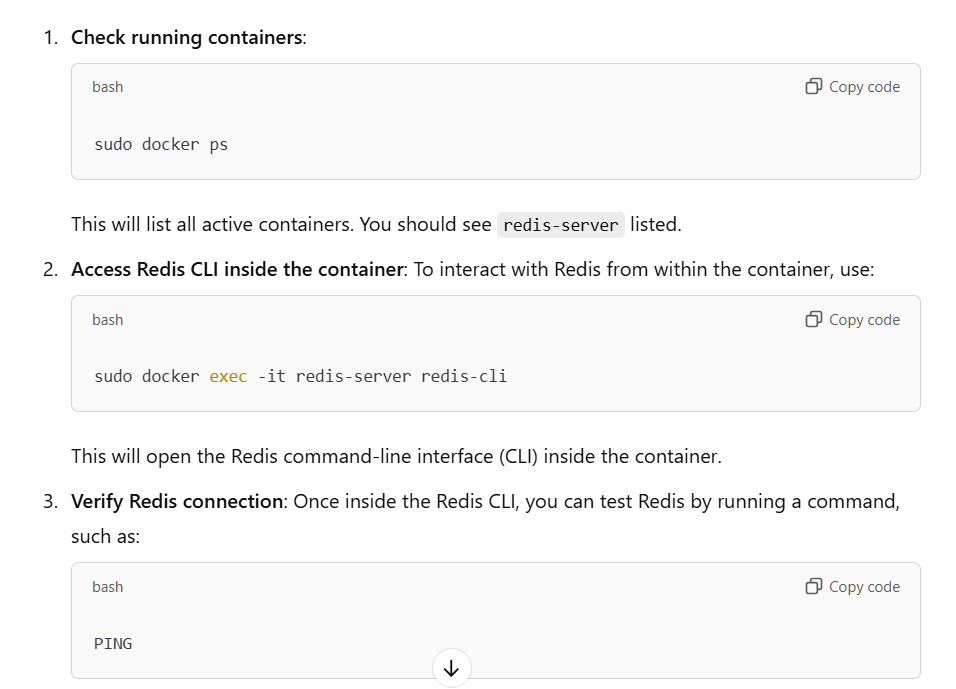
sudo apt install docker.io

sudo systemctl enable --now docker

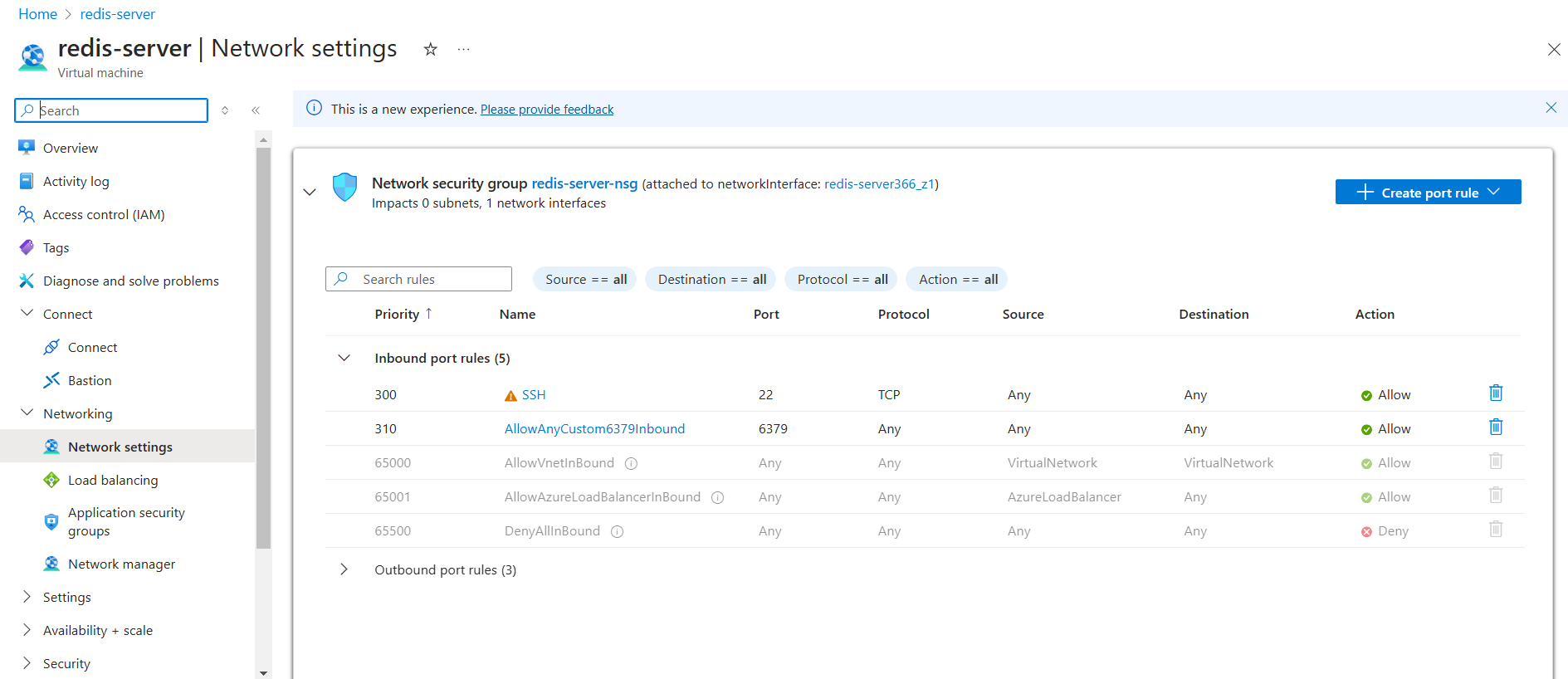
docker –version

sudo docker pull redis

sudo docker run -d --name redis-server -p 6379:6379 redis



And also do this



**For Connecting to the docker redis-app**

ssh [UrmilPawar@4.188.77.100](mailto:UrmilPawar@4.188.77.100)

sudo docker ps

**Starts the redis server if after restarting or starting-stopping the VM**

sudo docker start redis-server

Asynchronous Code

In the app.js of both teacher and students following are synchronous and asynchronous components

**Synchronous Sections:**

The following are **not asynchronous**:

1. **calculateDistance function:** This is a synchronous mathematical calculation.
2. **AuthenticateUser function:** This simply returns true, with no asynchronous calls.

**Summary of Asynchronous Components:**

1. **fetch calls** (to Flask server and WebSocket server).
2. **Geolocation API (navigator.geolocation.getCurrentPosition)**.
3. **WebSocket operations** (onopen, onmessage, etc.).
4. **Use of await (if included).**

The geolocation API is not included in async/await because there is no point in waiting till it gets completed. The location data is used very last by both students and teachers so it can be fetched parallelly to improve efficiency

The fetch() calls to store and fetch student data for students and teachers respectively are POST requests. From the student store request we aren’t fetching anything to wait and from the teacher’s fetch request though we are fetching we are displaying data (calling display function ) it is only after the data is fetched.

For loadbalancing fetch() request we require serverURL to e fetched before returning it and as fetch() is asynchronous the serverURL is returned before it is even fetched() there we have to use async/await instead of traditional .then()

For this we also have to do async in main function

document.getElementById('studentForm').addEventListener('submit', async function(event) { }

------------------------------------------------------------------------------------------------------------

**Ports**

**Websockets: 8080, 8081…**

**Flask : 8000….**

**Loadbalancers: 8080……**

**New Redis network**

**Redis Working:** There are many channels (communication lines on which message flows). Some users subscribe to the channels(wait and listen for any message) and some users publish to the channels (write messages on the communication). There is a central dedicated server that maintains these channels and the users subscribing and publishing to it.

**Working of WebSockets:** There is a single channel: **classconnect-channel** to which every websocket is connected. Least connection server is also connected to it. Every websocket subscribes to the channel and continuously wait from a message to the teacher.

When a teacher gets connected a message is published on the common channel so that every websocket connected gets the message.

There are two types of messages:

**Teacher clicking on connect**: Teacher gets added to the set in current websocket server and the message with **action=connect** is published so that other servers can also add the teacher in their respective ‘teacher’ set

**Teacher clicking on start**: message with **action=start** is published on the channel so that every WebSocket receives it and every student connected to the respective websocket receives the message of teacher getting connected . (If there is no student connected to the respective websocet then ‘no student connected ‘ gets printed to console.log)

Whenever any teacher or student is connected to the WebSocket or their connection is closed with the WebSocket a message with **action=‘check’** is sent on the channel. Even though all the WebSockets received the message only the **least connections load balancer** perform action for that message => **update the count of connection** for that websocket.