MAVERICK
SIMULATION SOLUTIONS

Auscultation Simulator

January 2024 - March 2024

By,

Kumar Laxmikant
Phone no: +91 9315198206
Email Id: laxmikantk2002@gmail.com

1. Introduction

1.1 Purpose

The Auscultation Simulator is a project designed to provide a realistic and immersive simulation environment for medical training and education. This innovative system aims to replicate authentic auscultation scenarios, allowing healthcare professionals, students, and educators to practice and refine their skills in a controlled and risk-free environment.

1.2 Scope

The scope of this documentation is to guide users, administrators, and developers through the setup, configuration, and usage of the Auscultation Simulator. It covers the integration of the Django web framework for the server-side and Raspberry Pi 4 Model B for handling sound playback and real-time simulation.

1.3 Audience

This documentation is intended for the following audience:

- **Users:** Healthcare professionals, medical students, and educators who will interact with the Auscultation Simulator for training and educational purposes.
- **Administrators:** Those responsible for setting up, configuring, and maintaining the Auscultation Simulator system.
- **Developers:** Individuals involved in extending or modifying the Auscultation Simulator codebase.

2. System Overview

2.1 Architecture

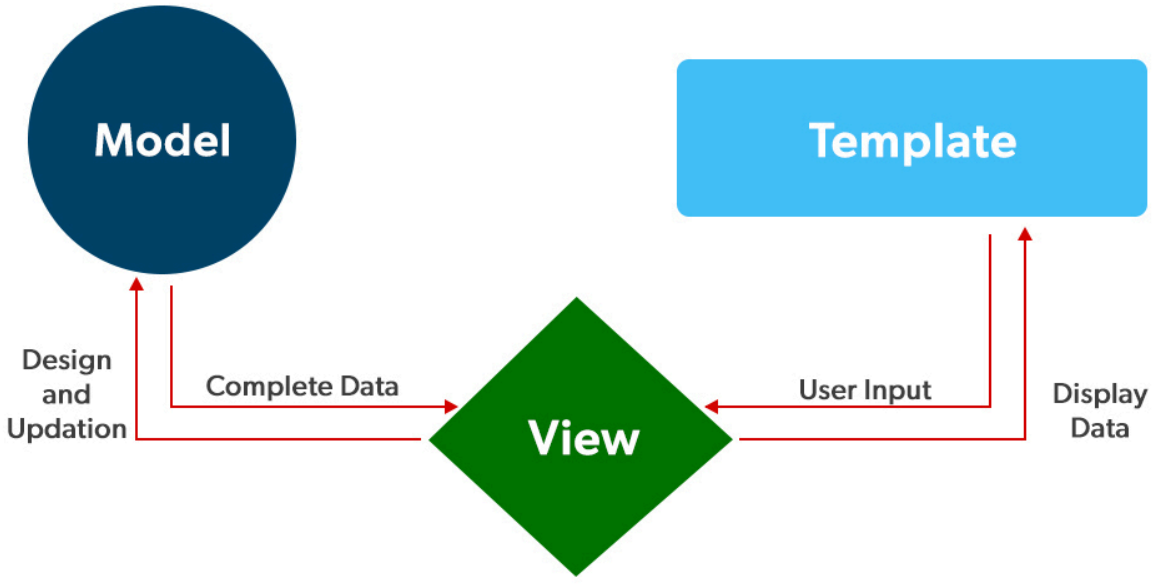
The Auscultation Simulator is built upon a client-server architecture, utilizing Django as the web framework for the server-side component and Raspberry Pi 4 Model B for handling sound playback and real-time simulation.

2.1.1 Client-Side (Web Interface):

- The web interface is developed using Django, a high-level Python web framework.
- Users interact with the simulator through a user-friendly interface accessible via web browsers.

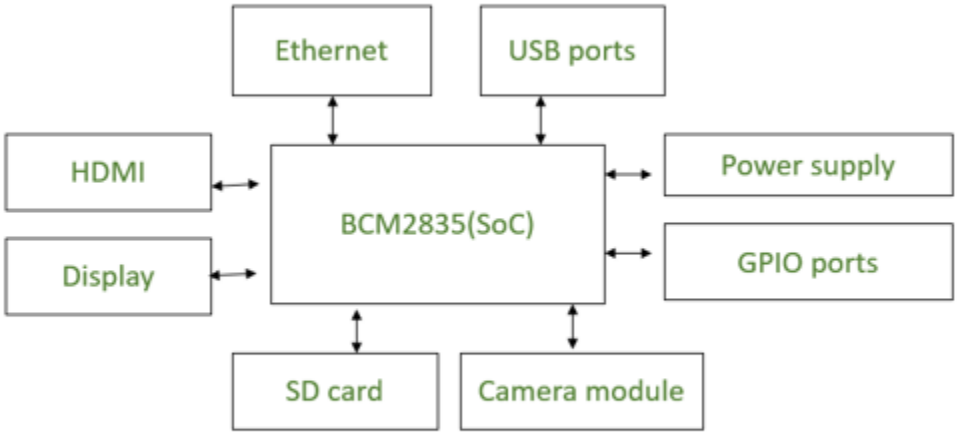
2.1.2 Server-Side (Django Server):

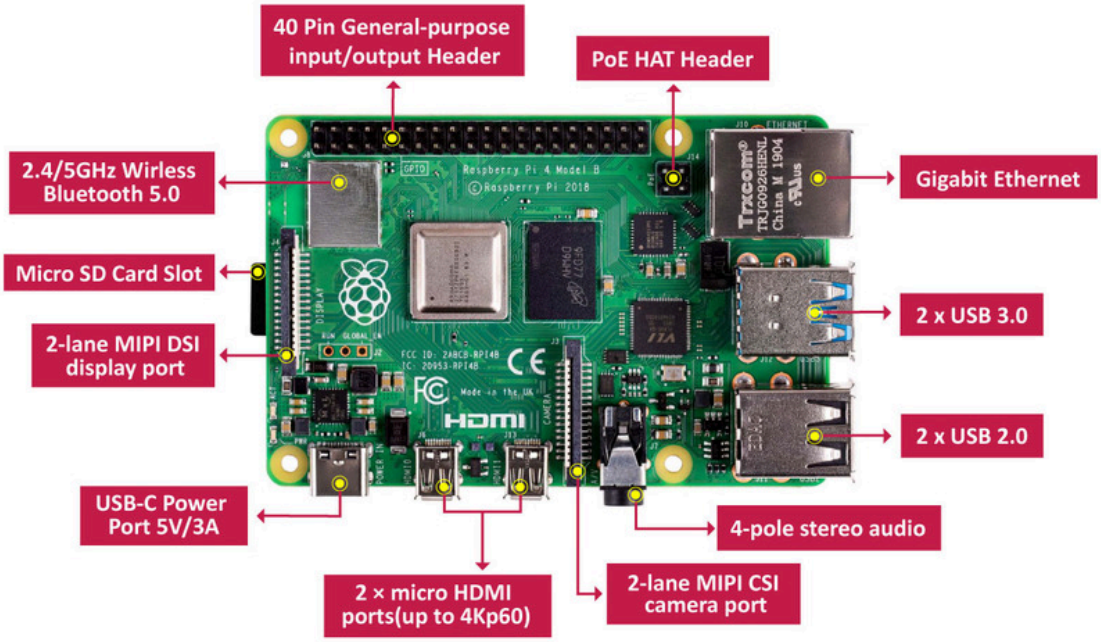
- The Django server manages user authentication, scenario selection, and communicates with the Raspberry Pi for sound playback control.
- The server hosts the database of auscultation sounds and handles user requests.



2.1.3 Raspberry Pi (Sound Playback):

- The Raspberry Pi acts as the sound playback device, responsible for generating realistic auscultation sounds in real-time.
- It communicates with the Django server to receive instructions on which sounds to play and at what parameters.





2.2 Components

- Raspberry Pi 4 Model B - 8GB RAM
- Raspberry Pi Official USB-C Power Supply for Raspberry Pi 4
- USB 3.0 7-Port Hub with 2-charging ports
- Audio Adapter External Sound Card USB to Audio 3.5mm Microphone Jack

3. Setup and Installation

3.1 Django Server

3.1.1 Prerequisites:

Before starting the installation process, make sure you have the following prerequisites installed on your server:

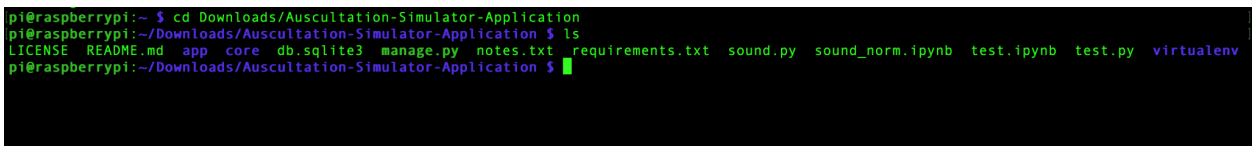
- Python (Version 3.9.2)
- Django (Version 4.2.9)
- Database System (SQLite)

3.1.2 Installation Steps:

This section provides a step-by-step guide for setting up and installing the Django server. Ensure that you follow these instructions carefully to have a successful deployment.

1. **Navigate to the Project Directory:**

   cd Auscultation-Simulator-Application



2. **Create a virtual environment:**

   Using Python 2: *python -m venv virtualenv*

   Using Python 3: *python3 -m venv virtualenv*

3. **Activate the virtual environment:**

   Windows: *virtualenv\Scripts\activate*

   MacOS/Linux: *source virtualenv/bin/activate*

```
pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $ source virtualenv/bin/activate
(virtualenv) pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $ ls
LICENSE  README.md  app  core  db.sqlite3  manage.py  notes.txt  requirements.txt  sound.py  sound_norm.ipynb  test.ipynb  test.py  virtualenv
(virtualenv) pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $
```

4. **Install the dependencies:**

   Automatically: *pip install -r requirements.txt*

   Manual: *pip install <package-name>*

```
(virtualenv) pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $ pip install pandas
Defaulting to user installation because normal site-packages is not writeable
Looking in indexes: https://pypi.org/simple, https://www.piwheels.org/simple
Requirement already satisfied: pandas in /home/pi/.local/lib/python3.9/site-packages (2.2.0)
Requirement already satisfied: numpy<2,>=1.22.4 in /home/pi/.local/lib/python3.9/site-packages (from pandas) (1.26.3)
Requirement already satisfied: python-dateutil>=2.8.2 in /home/pi/.local/lib/python3.9/site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /home/pi/.local/lib/python3.9/site-packages (from pandas) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.7 in /home/pi/.local/lib/python3.9/site-packages (from pandas) (2023.4)
Requirement already satisfied: six>=1.5 in /usr/lib/python3/dist-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
(virtualenv) pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $
```

5. **Migrate the Django project:**

   Using Python 2: *python manage.py migrate*

   Using Python 3: *python3 manage.py migrate*

6. **Run the application:**

   Local Server (Python 2): *python manage.py runserver*

   Local Server (Python 3): *python3 manage.py runserver*

```
(virtualenv) pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 4 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): app.
Run 'python manage.py migrate' to apply them.
January 29, 2024 - 04:06:39
Django version 4.2.9, using settings 'core.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

   Custom Server (Python 2): *python manage.py runserver <ip-addr>:<port-no>*

Custom Server (Python 3): *python3 manage.py runserver <ip-addr>:<port-no>*

```
(virtualenv) pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $ python manage.py runserver 192.168.100.108:8000
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 4 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): app.
Run 'python manage.py migrate' to apply them.
January 29, 2024 - 04:07:40
Django version 4.2.9, using settings 'core.settings'
Starting development server at http://192.168.100.108:8000/
Quit the server with CONTROL-C.
```

3.2 Raspberry Pi

1. **Open Terminal/cmd/Powershell/PuTTy**
2. **Ping Raspberry Pi's DNS:** *ping raspberrypi.local*

```
kumarlaxmikant@Kumars-MacBook-Pro ~ % ping raspberrypi.local
PING raspberrypi.local (192.168.100.108): 56 data bytes
64 bytes from 192.168.100.108: icmp_seq=0 ttl=64 time=3.196 ms
64 bytes from 192.168.100.108: icmp_seq=1 ttl=64 time=3.631 ms
64 bytes from 192.168.100.108: icmp_seq=2 ttl=64 time=3.674 ms
64 bytes from 192.168.100.108: icmp_seq=3 ttl=64 time=3.262 ms
64 bytes from 192.168.100.108: icmp_seq=4 ttl=64 time=3.156 ms
64 bytes from 192.168.100.108: icmp_seq=5 ttl=64 time=3.173 ms
64 bytes from 192.168.100.108: icmp_seq=6 ttl=64 time=3.504 ms
64 bytes from 192.168.100.108: icmp_seq=7 ttl=64 time=3.408 ms
^C
--- raspberrypi.local ping statistics ---
8 packets transmitted, 8 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 3.156/3.376/3.674/0.196 ms
```

3. **SSH into Raspberry Pi:** *ssh <user-name>@raspberrypi.local* (UserName: pi)
4. **Enter the Password:** (Password: 12345678)

```
kumarlaxmikant@Kumars-MacBook-Pro ~ % ssh pi@raspberrypi.local
pi@raspberrypi.local's password:
Linux raspberrypi 6.1.21-v8+ #1642 SMP PREEMPT Mon Apr  3 17:24:16 BST 2023 aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Jan 25 17:19:40 2024
pi@raspberrypi:~ $
```

5. **Navigate to Application directory:** *cd Downloads/Auscultation-Simulator-Application*

```
pi@raspberrypi:~ $ cd Downloads/Auscultation-Simulator-Application
pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $ ls
LICENSE  README.md  app  core  db.sqlite3  manage.py  notes.txt  requirements.txt  sound.py  sound_norm.ipynb  test.ipynb  test.py  virtualenv
pi@raspberrypi:~/Downloads/Auscultation-Simulator-Application $
```

3.3 Dependencies

The following additional dependencies or libraries is required for the Auscultation Simulator to function properly:

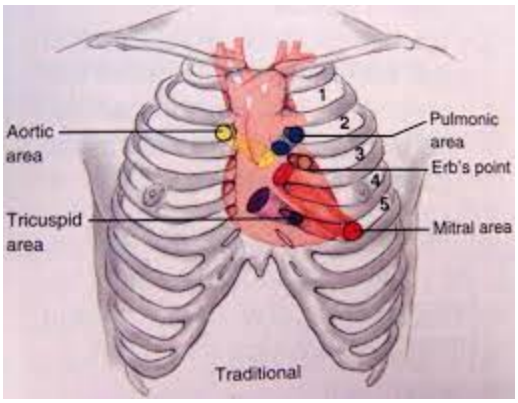
- **pandas (*Version: 2.1.4*):** A powerful data manipulation and analysis library for Python, providing data structures like DataFrame for efficient data handling.
- **numpy (*Version: 1.26.3*):** The fundamental package for scientific computing in Python, facilitating large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays.
- **plotly (*Version: 5.18.0*):** An interactive and open-source plotting library for Python, known for creating visually appealing and interactive plots, charts, and dashboards.
- **dash (*Version: 2.9.3*):** A productive framework for building web applications in Python, particularly well-suited for creating interactive, analytical dashboards.
- **sounddevice (*Version: 0.4.6*):** A Python library for playing and recording sound, providing a simple interface to interact with audio devices.
- **soundfile (*Version: 0.12.1*):** A library for reading and writing sound files in Python, supporting a variety of audio file formats.
- **neurokit2 (*Version: 0.2.7*):** A Python toolbox for neurophysiological signal processing, offering functionalities for processing physiological data, including ECG, EMG, and more.
- **django_plotly_dash (*Version: 2.2.2*):** A Django app for creating Plotly Dash applications within Django projects, allowing integration of interactive data visualizations into Django web applications.
- **pydub (*Version: 0.25.1*):** A Python library for audio file manipulation, providing high-level abstractions for tasks like converting between different audio formats and applying effects.
- **playsound (*Version: 1.3.0*):** A pure Python library for playing sound files, offering a simple and cross-platform way to play audio in Python applications.
- **threadpoolctl (*Version: 3.2.0*):** A library for controlling the number of threads used by native libraries that rely on thread-local storage, helping manage global threading resources in a Python program.

4. Sound Database

The Auscultation Simulator Sound Database is a curated collection of auscultation sounds designed to provide a diverse and realistic set of audio samples for medical training and education. The database includes sounds from the heart, lungs, abdomen and korotkoff, covering various physiological and pathological conditions.

4.1 Heart Sounds

Heart sounds are recorded from five locations, capturing the unique acoustics of each valve and Erb's point:



1. **Mitral Valve:** A heart valve located between the left atrium and left ventricle, regulating blood flow from the atrium to the ventricle.
2. **Aortic Valve:** A semilunar valve situated between the left ventricle and the aorta, controlling blood flow from the heart to the body.
3. **Pulmonary Valve:** A semilunar valve found between the right ventricle and the pulmonary artery, managing blood flow to the lungs.
4. **Tricuspid Valve:** Positioned between the right atrium and right ventricle, controlling blood flow from the atrium to the ventricle on the right side of the heart.
5. **Erb's Point:** A specific auscultation point on the chest used to listen to heart sounds, located at the third intercostal space at the left sternal border.

The 41 heart sounds include:

1. **Pulmonary Hypertension:** Abnormal heart sound associated with increased pressure in the pulmonary arteries.
2. **Patent Ductus Arteriosus:** Persistent opening between two major blood vessels leading from the heart.

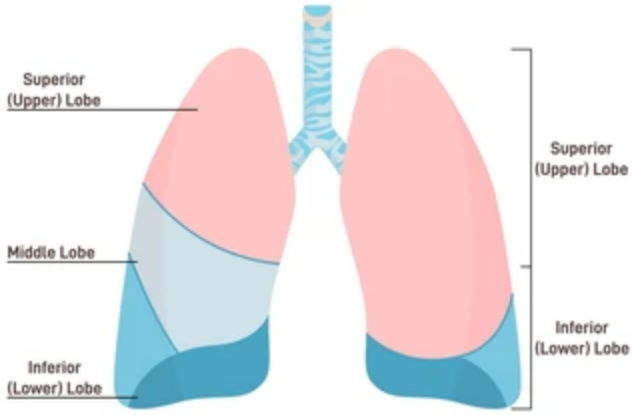
3. **Tetralogy of Fallot:** Congenital heart defect with four abnormalities affecting the heart's structure.
4. **Hypertrophic Cardiomyopathy:** Enlarged heart muscle leading to potential blockage of blood flow.
5. **Congestive Heart Failure:** Heart's inability to pump blood efficiently, causing fluid buildup.
6. **Holosystolic Murmur:** Continuous heart murmur throughout the entire contraction phase.
7. **Mitral Stenosis and Regurgitation:** Narrowing of the mitral valve and backward flow of blood.
8. **Tricuspid Valve Regurgitation:** Backward flow of blood due to incomplete closure of the tricuspid valve.
9. **Mitral Valve Stenosis:** Narrowing of the mitral valve, impeding blood flow.
10. **Normal Heart:** Standard heart sound, representing a healthy cardiac cycle.
11. **Opening Snap:** Sharp sound associated with the opening of the mitral valve.
12. **Pulmonary Valve Regurgitation:** Backward flow of blood through the pulmonary valve.
13. **Still's Murmur:** Innocent heart murmur often heard in children.
14. **Mitral Stenosis and Tricuspid Regurgitation:** Combined sounds of narrowed mitral valve and tricuspid regurgitation.
15. **Ventricular Septal Defect:** Hole in the heart's septum, leading to abnormal blood flow.
16. **Acute Pericarditis:** Inflammation of the pericardium causing friction during heartbeats.
17. **Fourth Heart Sound Gallop:** Extra heart sound associated with atrial contraction.
18. **Pericardial Rub:** Sound caused by inflamed pericardial surfaces rubbing against each other.
19. **Systemic Hypertension:** Elevated blood pressure within the arteries.
20. **Early Systolic Murmur:** Heart murmur occurring in the early part of systole.
21. **Atrial Septal Defect:** Hole in the heart's septum affecting blood flow.
22. **Congenital Aortic Stenosis:** Birth defect causing narrowing of the aortic valve.
23. **Ventricular Aneurysm:** Ballooning of the heart's ventricular wall.
24. **Austin Flint Murmur:** Low-pitched rumbling sound associated with aortic regurgitation.
25. **Functional Murmur:** Innocent heart murmur with no structural abnormalities.
26. **Aortic Valve Stenosis:** Narrowing of the aortic valve, restricting blood flow.

27. **Aortic Valve Regurgitation:** Backward flow of blood through the aortic valve.
28. **Continuous Murmur:** Uninterrupted sound heard throughout the cardiac cycle.
29. **Acute Myocardial Infarction:** Heart attack resulting from the blockage of blood flow to the heart muscle.
30. **Mitral Valve Prolapse:** Valve's abnormal backward displacement during contraction.
31. **Mid-Systolic Murmur:** Heart murmur occurring in the middle part of systole.
32. **Coarctation of the Aorta:** Narrowing of the aorta, restricting blood flow.
33. **Diastolic Murmur:** Heart murmur occurring during diastole.
34. **Aortic Stenosis and Regurgitation:** Combined sounds of narrowed aortic valve and regurgitation.
35. **Pulmonary Valve Stenosis:** Narrowing of the pulmonary valve, hindering blood flow.
36. **Split Second Heart Sound:** Splitting of the second heart sound, often heard in specific conditions.
37. **Ebstein's Anomaly:** Congenital heart defect affecting the tricuspid valve.
38. **Split First Heart Sound:** Splitting of the first heart sound, heard during inspiration.
39. **Dilated Cardiomyopathy:** Enlargement of the heart chambers, leading to reduced pumping ability.
40. **Mitral Valve Regurgitation:** Backward flow of blood through the mitral valve.
41. **Third Heart Sound Gallop:** Extra heart sound associated with rapid filling of the ventricles.

4.2 Lung Sounds

Lung sounds are recorded at eleven locations, encompassing both anterior and posterior aspects:

1. **LLB** - Left Lower Back
2. **LLF** - Left Lower Front
3. **LMB** - Left Middle Back
4. **LUB** - Left Upper Back
5. **LUF** - Left Upper Front
6. **RLB** - Right Lower Back
7. **RLF** - Right Lower Front



8. **RMB** - Right Middle Back
9. **RMF** - Right Middle Front
10. **RUB** - Right Upper Back
11. **RUF** - Right Upper Front

The 18 lung sounds include:

1. **Left-sided Pneumothorax:** Collection of air in the left pleural space, causing lung collapse.
2. **Harsh Respiration:** Coarse and rough breathing sounds.
3. **Rhonchi:** Continuous low-pitched wheezing sounds during breathing.
4. **Wheezes:** High-pitched whistling sounds during breathing.
5. **Right-sided Pneumothorax:** Collection of air in the right pleural space, causing lung collapse.
6. **Coarse Crackles:** Discontinuous, low-pitched cracking sounds during breathing.
7. **Bronchovesicular Respiration:** Mix of bronchial and vesicular breath sounds.
8. **Pleural Friction Rub:** Rough, grating sound during breathing due to inflamed pleura.
9. **Diminished Vesicular Respiration:** Reduced intensity of normal breath sounds.
10. **Fine Crackles:** Discontinuous, high-pitched crackling sounds during breathing.
11. **Asthma:** Respiratory condition characterized by wheezing and difficulty in breathing.
12. **COVID-19:** Respiratory condition associated with the coronavirus, causing various symptoms.
13. **Amphoric Respiration:** Hollow, echoing breath sounds.
14. **Pneumonia:** Inflammation of the lungs affecting air sacs and causing breathing difficulties.
15. **Bronchial Respiration:** Loud, high-pitched breath sounds heard over the large airways.
16. **Vesicular Respiration:** Normal breath sounds heard over the smaller airways.
17. **Stridor:** High-pitched, harsh sound during inspiration or expiration, often associated with airway obstruction.
18. **Gurgling Rhonchi:** Continuous low-pitched gurgling sounds during breathing.

4.3 Abdomen Sounds

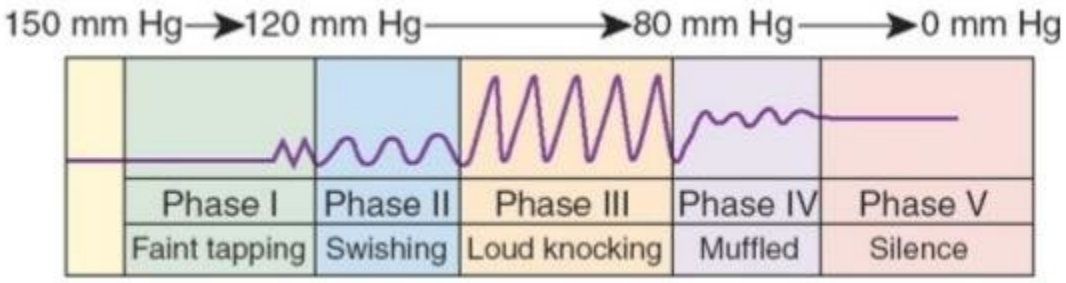
The abdomen sound library features a variety of gastrointestinal and abdominal conditions:

1. **Borborygmus:** Audible sounds produced by the movement of gas and fluids in the intestines.
2. **Bruits due to Renal Arteries Stenosis:** Abnormal sounds resulting from the narrowing of renal arteries.
3. **Capotement:** Audible splashing sounds associated with fluid and gas in the stomach.
4. **Constipation:** Difficulty in passing stools, often associated with reduced bowel movements.
5. **Crohn's Disease:** Inflammatory bowel disease causing abdominal pain, diarrhea, and weight loss.
6. **Diarrhea:** Frequent and loose bowel movements, often associated with gastrointestinal issues.
7. **Hyperactive Sounds:** Abnormally increased bowel sounds.
8. **Hypoactive Sounds:** Abnormally decreased bowel sounds.
9. **Irritable Bowel Syndrome:** Functional gastrointestinal disorder characterized by abdominal pain and altered bowel habits.
10. **Normal Bowel:** Regular and typical bowel sounds.
11. **Normal Bowel Sound With Bruits:** Regular bowel sounds accompanied by abnormal vascular sounds.
12. **Paralytic Ileus:** Obstruction of the intestines due to decreased or absent bowel movements.
13. **Peritoneal Friction Rub:** Audible rubbing or grating sounds due to inflammation of the peritoneum.
14. **Ulcerative Colitis:** Chronic inflammatory bowel disease causing inflammation and ulcers in the colon.

4.4 Korotkoff Sounds

There are 5 sounds to simulate Korotkoff:

1. **Tapping:** Sharp and distinct sounds heard during the initial appearance of blood flow.
2. **Swishing:** Continuous, turbulent sounds as blood flows more freely through the partially compressed artery.
3. **Knocking:** Sharp, repetitive sounds associated with changes in blood flow as the cuff pressure decreases.
4. **Muffling:** Decreased intensity or muffled sounds indicating continued blood flow with further cuff deflation.
5. **Silent:** Complete cessation of sounds, marking the point where the artery remains open throughout the cardiac cycle.



5. Technology Used

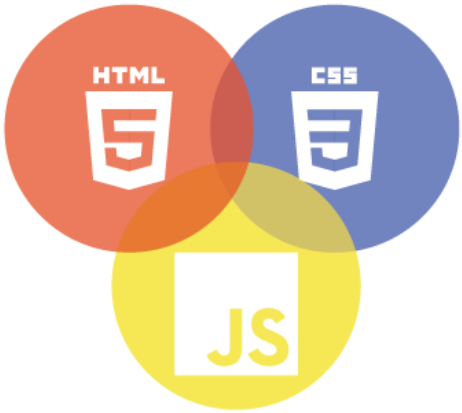
5.1 Software

5.1.1 Framework

- **Django:** Django is a powerful web framework for rapid development, known for its robustness and scalability. Built in Python, it comes with a wide range of features that accelerate development.

- **Bootstrap:** Bootstrap is a popular front-end framework for building responsive and mobile-first websites. It offers a collection of pre-styled components and a grid system, making it easy to create visually appealing and consistent user interfaces.

5.1.2 Front-End



- **HTML:** HTML provided the structural foundation, defining elements like headings, paragraphs, forms, and tables, ensuring clarity and accessibility.
- **CSS:** CSS enhanced visual presentation, handling styling aspects such as colors, fonts, layout, and responsiveness, aligning with branding guidelines for a cohesive interface.
- **JavaScript:** JavaScript added interactivity and dynamic behavior, enabling features like form validation, animations, and real-time updates, improving usability.

5.1.3 Back-End

- **Python:** Python serves as the backbone of the Auscultation Simulator's backend, chosen for its versatility, ease of development, and extensive libraries. Integrated seamlessly with the Django framework, Python enabled rapid development and efficient handling of data management, business logic implementation, and API development. Despite challenges such as performance optimization and security concerns, Python's flexibility and scalability allowed for the creation of a secure, scalable, and feature-rich application tailored to the needs of healthcare professionals and patients.

5.1.4 Database

- **SQLite:** SQLite, selected for the Auscultation Simulator's database, offers a lightweight and efficient solution for managing medical data. Its ease of deployment and integration with Django simplify data modeling and manipulation. Despite its lightweight nature, SQLite maintains strong performance, ensuring rapid data processing. While scalability and concurrency may present challenges as the application grows, SQLite's reliability and compatibility make it a solid choice for the Auscultation Simulator's database needs.

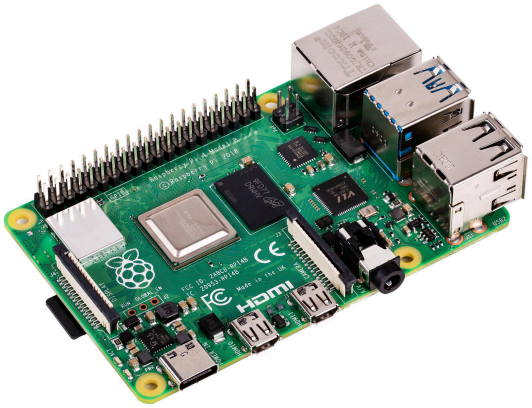
5.2 Operating System

- **Raspberry Pi OS:** The Auscultation Simulator operates on Raspberry Pi OS (*Legacy, 32-bit*), chosen for its seamless compatibility with Raspberry Pi hardware and optimized performance for embedded systems. Its lightweight nature ensures efficient resource utilization, essential for running the simulator on low-powered hardware. While the 32-bit architecture may pose limitations compared to 64-bit systems, optimizations and efficient resource management techniques mitigate these concerns, ensuring smooth operation. With access to a broad ecosystem of software and tools tailored for the Raspberry Pi platform, Raspberry Pi OS provides a stable and reliable operating

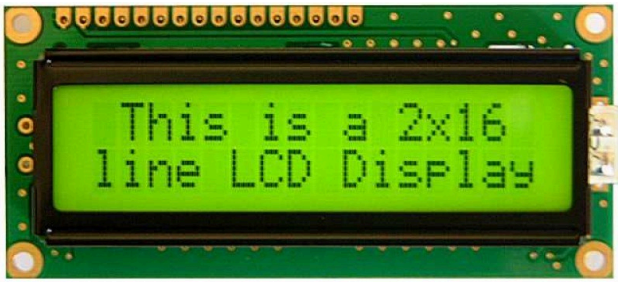
environment for the Auscultation Simulator, enabling immersive and realistic medical simulation training for healthcare professionals.

5.3 Hardware

- **Raspberry Pi 4 Model 4 (8GB RAM):** The Auscultation Simulator benefits from the robust capabilities of the Raspberry Pi 4 Model B with 8GB of RAM, ensuring optimal performance and responsiveness. This upgraded hardware configuration provides ample memory capacity for multitasking and running memory-intensive processes, essential for handling the computational demands of realistic auscultation simulations. With improved processing power and enhanced connectivity options, including Gigabit Ethernet and USB 3.0 ports, the Raspberry Pi 4 Model B facilitates seamless integration with peripherals and network devices, enabling accurate and immersive medical training experiences.



- **16x2 LCD Display:** The inclusion of a 16x2 LCD display in the Auscultation Simulator enriches user interaction by offering real-time updates on system metrics and network connectivity. Through the display's dynamic functionality, users can seamlessly monitor essential system resources such as CPU usage, RAM utilization, and Disk usage, followed by immediate visibility into the Raspberry Pi's IP address for network connectivity assessment.



- **Bootable USB SSD:** The integration of a bootable USB SSD with partitioned storage enhances the Auscultation Simulator's functionality, providing both portability and efficient data storage. With one partition hosting a bootable operating system and another dedicated to storage, the simulator benefits from streamlined setup across hardware configurations and ample space for medical data and simulation files.

- **Buzzer:** A buzzer has been integrated into the Auscultation Simulator, programmed to emit a brief beep lasting 0.1 seconds whenever updates are made to the heart rate or breath rate parameters. This auditory feedback mechanism enhances user engagement and provides immediate confirmation of parameter adjustments during simulation sessions.

6. Front-End Logic

6.1 HTML Code Structure

6.1.1 Parent Page display

6.1.1.1 Static file loading

```
{% load static %}
{% load plotly_dash %}
{% load crispy_forms_tags %}
```

These Django template tags are used to load static files, Plotly Dash, and crispy-forms functionalities into the template.

6.1.1.2 Favicon, Fonts and Icon Font Stylesheets

```
<link rel="icon" href="{% static 'image/dollar_icon.png' %}">
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link
href="https://fonts.googleapis.com/css2?family=Open+Sans:wght@400;600&family=Roboto:wght@500
```

```
;700&display=swap" rel="stylesheet">
```

This includes a link to the website's favicon and links to Google Web Fonts for 'Open Sans' and 'Roboto' fonts.

```
<link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.10.0/css/all.min.css"
rel="stylesheet">
<link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.4.1/font/bootstrap-icons.css"
rel="stylesheet">
```

These lines include the Font Awesome and Bootstrap Icons stylesheets for using icon fonts in the web page.

6.1.1.3 Bootstrap Stylesheets and Template Stylesheet

```
<link href="{% static 'css/bootstrap.min.css' %}" rel="stylesheet">
```

This links to the customized Bootstrap stylesheet.

```
<link href="{% static 'css/style.css' %}" rel="stylesheet">
```

This links to a custom stylesheet for additional styling.

6.1.1.4 Loading Spinner

```
<div id="spinner" class="show bg-dark position-fixed translate-middle w-100 vh-100 top-50
start-50 d-flex align-items-center justify-content-center">
    <!-- Spinner content -->
</div>
```

This section defines a loading spinner, which is initially visible and centered on the page.

6.1.1.5 JavaScript Libraries

```
<script src="https://code.jquery.com/jquery-3.6.4.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0/dist/js/bootstrap.bundle.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/3.8.0/chart.min.js"
integrity="sha512-sW/w8s4RWTdFFSduOTGtk4isV1+190E/GghVffMA9XczdJ2MDzSzLEubKAs5h0wzgSJOQTRYya
z73L3d6RtJSg==" crossorigin="anonymous" referrerpolicy="no-referrer"></script>
<script src="{% static 'js/main.js' %}"></script>
```

These lines include external JavaScript libraries for jQuery, Bootstrap, Chart.js, and the main JavaScript file for the template.

6.1.1.6 Various Includes for Scripts and Controls

```
{% include 'elements/Scripts/showVolumeControl.html' %}
{% include 'elements/Scripts/showLeftNavBar.html' %}
{% include 'elements/Scripts/footerButtons.html' %}
{% include 'elements/Scripts/soundPlay.html' %}
{% include 'elements/Scripts/heart_breath_update.html' %}
{% include 'elements/Scripts/volumeRangeControl.html' %}
{% include 'elements/Scripts/volumeUpdate.html' %}
{% include 'elements/Scripts/blink.html' %}
```

These lines include various scripts and controls for volume, left navigation bar, footer buttons, sound play, heart and breath updates, range input controls, volume updates, and blink controls.

6.1.2 Heart Rate display

6.1.2.1 Column 1: Button for Increasing Heart Rate

```
<div class="col-auto">
        <button type="submit" name="hr_plus" class="btn btn-link btn-outline-danger"
id="hr_plus" data-mdb-ripple-color="dark">
            <i class="fa fa-plus"></i>
        </button>
</div>
```

Column 1 contains a button styled as a link to increase the heart rate. It features a red outline and a plus icon, likely associated with a form submission or JavaScript function.

6.1.2.2 Column 2: Displaying Heart Rate

```
    <div class="col-auto">
        <div class="card text-info bg-dark">
            <div class="card-body">
                <p class="card-text font-weight-bold text-success" style="transition: 0.5s;"
id="hr_show">{{ hr_show }}</p>
            </div>
        </div>
    </div>
```

Column 2 displays the current heart rate in a dark-themed card with bold, green text. The value '*{{ hr_show }}*' suggests dynamic content updated from the server or client-side scripts.

6.1.2.3 Column 3: Button for Decreasing Heart Rate

```
   <div class="col-auto">
        <button type="submit" name="hr_minus" class="btn btn-link btn-outline-danger"
id="hr_minus" data-mdb-ripple-color="dark">
            <i class="fa fa-minus"></i>
        </button>
    </div>
```

Column 3 holds a button styled as a link to decrease the heart rate. Similar to Column 1, it has a red outline and a minus icon, linked to a form submission or JavaScript function.

6.1.3 Breadth Rate display

6.1.3.1 Column 1: Button for Increasing Respiratory Rate (rr)

```
   <div class="col-auto">
        <button type="submit" name="rr_plus" class="btn btn-link btn-outline-danger"
id="rr_plus" data-mdb-ripple-color="dark">
            <i class="fa fa-plus"></i>
        </button>
    </div>
```

The first column includes a button styled as a link to increase the respiratory rate. Similar to the heart rate column, it has a red outline and a plus icon. The button has an ID (rr_plus), a name attribute (rr_plus), and is likely associated with a form submission or JavaScript function.

6.1.3.2 Column 2: Displaying Respiratory Rate (rr)

```
   <div class="col-auto">
        <div class="card text-info bg-dark">
            <div class="card-body">
                <p class="card-text font-weight-bold" style="transition: 0.5s;"
id="rr_show">{{ rr_show }}</p>
            </div>
        </div>
    </div>
```

The second column displays the current respiratory rate within a dark-themed card. The card has an informative text color, and the respiratory rate value is presented in bold. The value '*{{ rr_show }}*' suggests dynamic content fetched from a server or updated through client-side scripts. The transition effect with a duration of 0.5 seconds implies a smooth visual change when the respiratory rate value is updated.

6.1.3.3 Column 3: Button for Decreasing Respiratory Rate (rr)

```
   <div class="col-auto">
        <button type="submit" name="rr_minus" class="btn btn-link btn-outline-danger"
id="rr_minus" data-mdb-ripple-color="dark">
            <i class="fa fa-minus"></i>
        </button>
    </div>
```

The third column includes a button styled as a link to decrease the respiratory rate. It resembles the corresponding heart rate column, featuring a red outline and a minus icon. Like the other buttons, this is likely linked to a form submission or JavaScript function. The data-mdb-ripple-color attribute suggests the use of the MDB library for a ripple effect during button clicks.

6.1.4 Left Navigation Bar display

```
<div class="navbar-nav w-100" id="heart_left_nav_bar">
    <a href="#" class="nav-item nav-link dropdown dropdown-toggle" data-toggle="dropdown"
id="dropdownmenu1" role="button" data-bs-toggle="dropdown" aria-expanded="false">
        <i class="fa fa-heartbeat me-2"></i>Heart tones</a>
    <div class="dropdown-menu dropdown-menu-end dropdown-menu-dark"
aria-labelledby="dropdownmenu1">
        <button type="submit" name="normal_heart_sound" id="normal_heart_sound" class="btn
dropdown-item text-wrap active">Normal heart sound</button>
        <button type="submit" name="split_first_heart_sound" id="split_first_heart_sound"
class="btn dropdown-item text-wrap">Split first heart sound</button>
        <button type="submit" name="split_second_heart_sound" id="split_second_heart_sound"
class="btn dropdown-item text-wrap">Split second heart sound</button>
        <button type="submit" name="third_heart_sound" id="third_heart_sound" class="btn
dropdown-item text-wrap">Third heart sound (gallop)</button>
        <button type="submit" name="fourth_heart_sound" id="fourth_heart_sound" class="btn
dropdown-item text-wrap">Fourth heart sound (gallop)</button>
    </div><!-- Rest of the dropdowns -->
</div>
```

The code creates a responsive UI with a dropdown menu for heart sounds. The navbar-nav class ensures proper styling, and the dropdown is triggered by a button with *nav-item*, *nav-link*, *dropdown*, and *dropdown-toggle* classes. The dropdown menu is styled with *dropdown-menu*, *dropdown-menu-end*, and *dropdown-menu-dark*. Each heart sound option is a button with *btn* and *dropdown-item* classes, and the text-wrap class allows text wrapping. The "Normal heart sound" button has the active class to indicate the default or selected option. Overall, this UI offers an intuitive way to choose heart sounds with consistent and visually appealing styling.

6.1.5 Right Navigation Bar display

```
<div id="normal_heart_sound_volume">
    <div class="row">
        <!-- Column 1: Button for Mitral Valve -->
        <div class="col-4">
            <button type="submit" name="normal_heart_sound_mitral_valve"
id="normal_heart_sound_mitral_valve" class="btn btn-link btn-square-md"
data-mdb-ripple-color="dark">Mitral valve</button>
        </div>

        <!-- Column 2: Adjustment Icon -->
        <div class="col-1">
            <i class="fa fa-adjust" aria-hidden="true"></i>
        </div>

        <!-- Column 3: Volume Up Icon -->
        <div class="col-1">
            <i class="fa fa-volume-up" aria-hidden="true"></i>
        </div>

        <!-- Column 4: Range Input for Volume -->
        <div class="col-5">
            <input type="range" class="form-range" min="0" max="100" value="0" step="1"
id="normal_heart_sound_mitral_valve_range">
        </div>

        <!-- Column 5: Display Volume Value -->
        <div class="col-1">
            <p id="normal_heart_sound_mitral_valve_range_value">0%</p>
        </div>
    </div>
    <!-- Rest of the code for other volume control panels -->
</div>
```

This code segment defines a volume control panel for the Mitral valve heart sound within a container labeled with the ID *normal_heart_sound_volume*. The panel comprises five columns in a row. The first column holds a button labeled "Mitral valve," likely linked to a form submission or JavaScript function associated with the Mitral valve heart sound. The second and third columns feature adjustment and volume up icons, respectively, serving as visual elements for control. The fourth column includes a range input styled as a form-range, allowing users to adjust the volume interactively. The fifth column displays the current volume value as a percentage. This structured interface enables users to control and monitor the volume specifically for the Mitral valve heart sound. Additional similar code sections may exist for controlling volumes of other heart sounds within the same container.

6.1.6 Special Volume Control display

```
<div class="bg-secondary text-center rounded px-2 py-2 gy-1">
    <div class="row">
        <div class="col-4">
            <button type="submit" name="mute_volume" id="mute_volume" class="btn btn-link
btn-outline-danger btn-square-md" data-mdb-ripple-color="dark">Mute All</button>
        </div>
        <div class="col-4">
            <button type="submit" name="default_volume" id="default_volume" class="btn
btn-link btn-outline-danger btn-square-md" data-mdb-ripple-color="dark">Default</button>
        </div>
        <div class="col-4">
            <button type="submit" name="full_volume" id="full_volume" class="btn btn-link
btn-outline-danger btn-square-md" data-mdb-ripple-color="dark">Max Volume</button>
        </div>
    </div>
</div>
```

This code snippet creates a compact control panel with three buttons for audio settings. Enclosed in a rounded secondary-colored background, the buttons—labeled "**Mute All**," "**Default**," and "**Max Volume**"—are evenly spaced within a centered row. Each button is associated with a specific function for muting all sounds, resetting to default settings, or maximizing the volume. The consistent styling, including the use of *btn*, *btn-link*, *btn-outline-danger*, and *btn-square-md* classes, maintains a cohesive design. The *data-mdb-ripple-color="dark"* attribute indicates the incorporation of MDB for a dark-colored ripple effect during button clicks. This user-friendly control panel offers convenient options for audio management.

6.1.7 Graph display container

```
{% load plotly_dash %}
<div class="row" id="hbr_div" style="display: none;">
    {% plotly_app name="blankhbrDash" ratio=0.3 %}
</div>
<!-- Code blocks for rest of the waveforms -->
```

This code snippet uses the Plotly Dash library within a Django template. It defines a hidden row (*hbr_div*) and includes a Plotly Dash app named "**blankhbrDash**" with a specific ratio of **0.3**. This likely represents a waveform related to heart rate. The comment suggests there are similar code blocks for displaying other waveforms, indicating a modular approach for handling various waveform types in the application.

6.1.8 Footer display

```
<div class="container-fluid px-3 pt-2 fixed-bottom">
    <div class="row">
        <div class="col-12">
            <div class="bg-secondary text-center rounded px-2 py-2 gy-1">
                <div class="row">
                    <div class="col-12">
                        <button type="button" class="btn btn-link btn-outline-danger
btn-square-md" data-mdb-ripple-color="dark" id="hbr_btn">HBR</button>
                        <button type="button" class="btn btn-link btn-outline-danger
btn-square-md" data-mdb-ripple-color="dark" id="ecg_btn">ECG</button>
                        <button type="button" class="btn btn-link btn-outline-danger
btn-square-md" data-mdb-ripple-color="dark" id="rsp_btn">RR</button>
                        <button type="button" class="btn btn-link btn-outline-danger
btn-square-md" data-mdb-ripple-color="dark" id="compare_btn">Compare</button>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

This code creates a fixed-bottom container with buttons for selecting medical simulation waveforms. The buttons, labeled "**HBR**," "**ECG**," "**RR**," and "**Compare**," are styled with MDB classes, providing a cohesive design. These likely serve as controls for displaying specific medical

waveforms, such as *heart rate* (HBR), *electrocardiogram* (ECG), *respiratory rate* (RR), and a comparison view.

6.2 Javascript Functionality

6.2.1 Heart Rate and Breadth Rate display blinking

```
var blink1 = document.getElementById('hr_show');
var blink2 = document.getElementById('rr_show');
setInterval(function() {
        blink1.style.opacity = (blink1.style.opacity == 0 ? 1 : 0);
        blink2.style.opacity = (blink2.style.opacity == 0 ? 1 : 0);
}, 500);
```

This JavaScript snippet creates a blinking effect for two HTML elements ('*hr_show*' and '*rr_show*') by toggling their opacity every **500 milliseconds**. This effect, achieved using setInterval, alternately makes the elements appear and disappear, commonly used for visual emphasis on a webpage.

6.2.2 Display Left Bar Panel

```
var heartLeftNavBar = document.getElementById("heart_left_nav_bar");
var lungs_front_nav = document.getElementById("lung_front_left_nav_bar");
var lungs_back_nav = document.getElementById("lung_back_left_nav_bar");
var bowel_nav = document.getElementById("bowel_left_nav_bar");
document.getElementById("heart_btn").onclick = function() {
   if (heartLeftNavBar.style.display === "none") {
        heartLeftNavBar.style.display = "block";
        lungs_front_nav.style.display = "none";
        lungs_back_nav.style.display = "none";
        bowel_nav.style.display = "none";

        document.getElementById("heart_btn").classList.add('active');
        document.getElementById("lungs_anterior_btn").classList.remove('active');
        document.getElementById("lungs_posterior_btn").classList.remove('active');
        document.getElementById("bowel_btn").classList.remove('active');
     }
};
// Logic for other Navigation buttons
```

This JavaScript code handles the interaction of a navigation bar with buttons for anatomical views. Clicking the "Heart" button displays the heart navigation bar and hides others, updating

button states for seamless switching between views. This same logic is followed for "**Lungs**" and "**Bowel**".

6.2.3 Graph switching implementation & Footer buttons activation and deactivation

```
hbr_btn.addEventListener("click", function() {
   this.classList.toggle("active");    // Toggle the 'active' class on the button
   if (currentDiv.style.display === "none") {
      currentDiv.style.display = "block";
   }
   else {
      currentDiv.style.display = "none";
   }
});
```

This JavaScript snippet toggles the "*active*" class on the "**hbr_btn**" button upon click. It also alternates the display state of a related element, switching between "*block*" and "*none*" for visibility control. This same logic is applied to other graph

6.2.4 Display Volume Control Panel

```
function getElement(id) {
    return document.getElementById(id + "_volume");
}

function hideAll() {
    var elements = [/* Sound ID's */]

    elements.forEach(function (element) {
        getElement(element).style.display = "none";
        document.getElementById(element).classList.remove('active');
    });
}

function showElement(id) {
    hideAll();
    getElement(id).style.display = "block";
    document.getElementById(id).classList.add('active');
}

document.getElementById("normal_heart_sound").onclick = function() {
    showElement("normal_heart_sound");
};
// More onclick functions for the rest of the buttons
```

This JavaScript code manages the visibility and active state of medical simulation elements related to heart and lung sounds. The *getElement*, *hideAll*, and *showElement* functions facilitate these actions. The functions handle a list of elements by toggling their display and active class based on user interactions. An *onclick* event is assigned to the "**normal_heart_sound**" element to trigger the display of the corresponding element. This modular structure allows efficient management of various sounds and associated UI components. There are more *onclick* functions defined to deal with button clicks associated with other audio and locations.

6.2.5 Volume range input and text synchronization

```
mitralRangeID.forEach((id,index) => {
    const rangeInput = document.getElementById(id);
    const valueParagraph = document.getElementById(mitralRangeID_value[index]);

    rangeInput.addEventListener('input', () => {
        const value = rangeInput.value;
        valueParagraph.textContent = value + '%';

        // Sync values among range inputs
        mitralRangeID.forEach((otherId,otherIndex) => {
            if (otherIndex !== index) {
                document.getElementById(mitralRangeID_value[otherIndex]).textContent = value
+ '%';
                document.getElementById(otherId).value = value;
            }
        });
    });
});
```

This JavaScript code utilizes the *forEach* method to handle multiple range inputs associated with mitral sounds. It dynamically connects each range input with its corresponding value paragraph based on provided IDs. Event listeners are added to each *range input*, responding to changes in input values.

When a user adjusts a range input, the associated value paragraph updates with the selected percentage. Moreover, the code ensures synchronization among all mitral range inputs. When one range input is adjusted, the values of other range inputs are updated accordingly, maintaining consistency across the user interface. This approach enhances the interactive and synchronized behavior of the mitral range inputs in the application.

6.3 Optimizing Data Interaction: AJAX and jQuery Implementation

6.3.1 Heart Rate updation request generation

```
$(document).ready(function () {
    $("#hr_plus").click(function () {
        $.ajax({
            type: 'POST',
            url: 'heartUpdate/',
            data: {'csrfmiddlewaretoken': '{{ csrf_token }}', 'hr_plus': true},
            success: function (response) {
                console.log(response);
                $("#hr_show").text(response.hr_show);
            },
            error: function (error) {
                console.log("Error:", error);
            }
        });
    });
});

$(document).ready(function () {
    $("#hr_minus").click(function () {
        $.ajax({
            type: 'POST',
            url: 'heartUpdate/',
            data: {'csrfmiddlewaretoken': '{{ csrf_token }}', 'hr_minus': true},
            success: function (response) {
                console.log(response);
                $("#hr_show").text(response.hr_show);
            },
            error: function (error) {
                console.log("Error:", error);
            }
        });
    });
});
```

This **jQuery** and **AJAX** code manages heart rate increment and decrement. Separate functions for "**hr_plus**" and "**hr_minus**" buttons send ***POST*** requests to '**heartUpdate/**' with the *CSRF token* and respective parameters. The server updates the heart rate, and the response updates the displayed value in the "**hr_show**" element. This implementation enables dynamic heart rate adjustments without page reload.

6.3.2 Breadth Rate updation request generation

```
$(document).ready(function () {
    $("#rr_plus").click(function () {
        $.ajax({
            type: 'POST',
            url: 'breathUpdate/',
            data: {'csrfmiddlewaretoken': '{{ csrf_token }}', 'rr_plus': true},
            success: function (response) {
                console.log(response);
                $("#rr_show").text(response.rr_show);
            },
            error: function (error) {
                console.log("Error:", error);
            }
        });
    });
});

$(document).ready(function () {
    $("#rr_minus").click(function () {
        $.ajax({
            type: 'POST',
            url: 'breathUpdate/',
            data: {'csrfmiddlewaretoken': '{{ csrf_token }}', 'rr_minus': true},
            success: function (response) {
                console.log(response);
                $("#rr_show").text(response.rr_show);
            },
            error: function (error) {
                console.log("Error:", error);
            }
        });
    });
});
```

This **jQuery** and **AJAX** code manages breath rate increment and decrement. Separate functions for "**rr_plus**" and "**rr_minus**" buttons send ***POST*** requests to '*breathUpdate/*' with the *CSRF token* and respective parameters. The server updates the breath rate, and the response updates the displayed value in the "**rr_show**" element. This implementation enables dynamic breath rate adjustments without page reload.

6.3.3 Sound play request generation

```
$(document).ready(function () {
    const Sounds = [/* All sound ID's */];

    function handleSoundClick(identifier, valveType) {
        $('#' + identifier + '_sound_' + valveType).click(function () {
            $.ajax({
                type: 'POST',
                url: 'soundPlay/',
                data: {'csrfmiddlewaretoken': '{{ csrf_token }}', [identifier + '_sound_' +
valveType]: true},
                success: function (response) {
                    // Handle success response
                    console.log('POST request successful for ' + identifier + '_sound_' +
valveType + ':', response);
                },
                error: function (error) {
                    // Handle error response
                    console.error('Error in POST request for ' + identifier + '_sound_' +
valveType + ':', error);
                }
            });
        });
    }

    Sounds.forEach(function (identifier) {
        handleSoundClick(identifier, 'mitral_valve');
        // Other handleSoundClick functions for other sounds and locations
    });
});
```

This jQuery code manages dynamic sound playback for various identifiers and valve types. The handleSoundClick function is defined to handle click events for each sound. Within the *$(document).ready* function, the code iterates through sound identifiers using a forEach loop, calling the handleSoundClick function for each. When a sound element is clicked, a ***POST*** request is sent to '*soundPlay/*' with the CSRF token and relevant parameters. Success or error responses are logged to the console. This implementation offers a modular approach for sound playback across different locations and valve types in the application.

6.3.4 Volume updation request generation

```
$(document).ready(function () {
    // Function to handle range input change for different heart sounds
    const heartSounds = [/* All sound ID's */];

    heartSounds.forEach(function (identifier) {
        $('#' + identifier + '_sound_mitral_valve_range').on('input', function () {
            // Get the current value of the range input
            var rangeValue = $(this).val();

            $.ajax({
                url: 'mitralVolumeChange/', // Replace with your backend endpoint
                method: 'POST',
                data: { 'csrfmiddlewaretoken': '{{ csrf_token }}', identifier: identifier,
rangeValue: rangeValue }, // Send the range value as data
                success: function (response) {
                    // Handle success response
                    console.log('POST request successful for ' + identifier + ':',
response);
                },
                error: function (error) {
                    // Handle error response
                    console.error('Error in POST request for ' + identifier + ':', error);
                }
            });
        });
    });
});
```

This jQuery code manages range input changes for different heart sounds. The heartSounds array contains all sound identifiers. The forEach loop iterates through each identifier, and for each, it attaches an 'input' event listener to the corresponding range input element ('***_sound_mitral_valve_range***'). When the range input changes, a ***POST*** request is sent to '*mitralVolumeChange/*' with the CSRF token and relevant parameters (identifier and rangeValue). Success or error responses are logged to the console. This code provides a flexible way to handle volume changes for various heart sounds in the application.

7. Back-end Logic

7.1 Heart Rate updation

```
def heartUpdate(request):
    global hr_show, current_mitral_valve_sound, current_aortic_valve_sound,
current_pulmonary_valve_sound, current_tricuspid_valve_sound, current_erb_valve_sound

    if request.method == 'POST':
        if 'hr_plus' in request.POST:
            hr_show += 1
            buzzer.on()
            time.sleep(0.1)
            buzzer.off()
            print('\nHeart Rate updated to: {}'.format(hr_show))
        elif 'hr_minus' in request.POST:
            hr_show -= 1
            buzzer.on()
            time.sleep(0.1)
            buzzer.off()
            print('\nHeart Rate updated to: {}'.format(hr_show))
        else:
            hr_show += 0

        if current_mitral_valve_sound is not None:
            start_mitral_thread(current_mitral_valve_sound)
        if current_aortic_valve_sound is not None:
            start_aortic_thread(current_aortic_valve_sound)
        if current_pulmonary_valve_sound is not None:
            start_pulmonary_thread(current_pulmonary_valve_sound)
        if current_tricuspid_valve_sound is not None:
            start_tricuspid_thread(current_tricuspid_valve_sound)
        if current_erb_valve_sound is not None:
            start_erb_thread(current_erb_valve_sound)
        return JsonResponse({'message': 'Success!', 'hr_show': hr_show})
    else:
        return HttpResponse("Request method is not a POST")
```

The provided Python code defines a Django view (**heartUpdate**) handling ***AJAX*** requests for heart rate updates. It increments or decrements the global variable **hr_show** based on received parameters ('**hr_plus**' or '**hr_minus**'). The function triggers a buzzer for a short duration and starts threads for valve-related sounds if present. The response includes a success message

and the updated heart rate. Global variables and thread usage may have considerations for thread safety and maintainability.

7.2 Breath Rate updation

```
def breathUpdate(request):
    global rr_show
    global current_lungs_sound, current_bowel_sound

    if request.method == 'POST':
        if 'rr_plus' in request.POST:
            rr_show += 1
            buzzer.on()
            time.sleep(0.1)
            buzzer.off()
            print('\nBreath Rate updated to: {}'.format(rr_show))
        elif 'rr_minus' in request.POST:
            rr_show -= 1
            buzzer.on()
            time.sleep(0.1)
            buzzer.off()
            print('\nBreath Rate updated to: {}'.format(rr_show))
        else:
            rr_show += 0

        if current_lungs_sound is not None:
            start_lungs_thread(current_lungs_sound)
        if current_bowel_sound is not None:
            start_bowel_thread(current_bowel_sound)
        return JsonResponse({'message': 'Success!', 'rr_show': rr_show})
    else:
        return HttpResponse("Request method is not a POST")
```

This Python code defines a Django view (**breathUpdate**) handling ***AJAX*** requests for breath rate updates. Similar to the previous code, it increments or decrements the global variable **rr_show** based on the received parameters ('**rr_plus**' or '**rr_minus**'). The function triggers a buzzer for a short duration and starts threads for lung and bowel-related sounds if present. The response includes a success message and the updated breath rate. As with the previous code, global variables and thread usage may have considerations for thread safety and maintainability.

7.3 Module to play audio on assigned speaker

```
def play_mitral(index, samples, samplerate):
    global speakers, stop_flag_mitral, current_mitral_valve_sound, hr_show
    delay_seconds = 60 / hr_show
    try:
        speaker = speakers[index]
    except:
        speaker = speakers[0]
    while not stop_flag_mitral.is_set():
        speaker.play(samples, samplerate)
        time.sleep(delay_seconds)
```

This Python code defines a function (**play_mitral**) to play a specific sound using a speaker. The function utilizes global variables (***speakers, stop_flag_mitral, current_mitral_valve_sound***, and ***hr_show***). The sound is played in a loop with a delay based on the heart rate (**hr_show**). Similar functions are implied for other locations like aortic, pulmonary, tricuspid, erb, lungs, and bowel, presumably handling sounds for those locations. Note that global variables and threading considerations may impact concurrent execution.

7.4 Module to activate thread to Play audio

```
def start_mitral_thread(sound_name):
    global playing_thread_mitral, stop_flag_mitral, hr_show, current_mitral_valve_sound
    if playing_thread_mitral and playing_thread_mitral.is_alive():
        stop_flag_mitral.set()  # Set the reload flag to signal the thread to stop
        playing_thread_mitral.join()
        print('Destroyed Mitral thread')
        stop_flag_mitral = threading.Event()

    # Start a new thread
    current_mitral_valve_sound = sound_name
    audio_path = df_heart.loc[(df_heart['sound_name'] == sound_name) &
(df_heart['sound_type'] == 'M'), 'audio_file_path'].values[0]
    heartbeat = AudioSegment.from_file(audio_path, format="wav")
    speed_multiplier = hr_show / 60.0  # Assuming 60 BPM as the baseline
    adjusted_heartbeat = heartbeat.speedup(playback_speed=speed_multiplier)
    exported_data = adjusted_heartbeat.export(format="wav").read()
    data, fs = sf.read(io.BytesIO(exported_data))
    playing_thread_mitral = threading.Thread(target=play_mitral, args=(1, data, fs))
    playing_thread_mitral.start()
```

This Python code defines a function (**start_mitral_thread**) that starts a new thread to play a specific heart sound. The function utilizes global variables (***playing_thread_mitral, stop_flag_mitral, hr_show***, and ***current_mitral_valve_sound***). Before starting a new thread, it checks if there is already an active thread and stops it. The thread plays the heart sound using the play_mitral function, adjusting the playback speed based on the heart rate (**hr_show**). Similar functions are implied for other locations like aortic, pulmonary, tricuspid, erb, lungs, and bowel, presumably handling sounds for those locations. Note that threading considerations and global variable usage may impact concurrent execution.

7.5 Module to update Sound Card Volume

```
def mitralVolumeChange(request):
    global current_mitral_valve_sound
    if request.method == 'POST':
        if request.POST['identifier'] == current_mitral_valve_sound:
            volume = request.POST['rangeValue']
            os.system('amixer -c 3 set Speaker {}%'.format(volume))
            print('Mitral Valve\'s Volume updated to {}%'.format(volume))
        return JsonResponse({'message': 'Success!'})
    else:
        return HttpResponse("Request method is not a POST")
```

This Python code defines a Django view function (**mitralVolumeChange**) that handles volume changes for a specific heart sound location (e.g., mitral valve) based on a ***POST*** request. It uses global variables (**current_mitral_valve_sound**) to identify the current sound being played. The function checks if the request method is ***POST*** and if the identifier in the request matches the current sound. If conditions are met, it extracts the volume value from the ***POST*** request and adjusts the volume using the os.system command. The function then prints a message and returns a ***JSON*** response. Similar functionality is implied for other locations such as aortic, pulmonary, tricuspid, erb, lungs, and bowel. Note that the use of os.system for volume control may have security implications and should be handled carefully in production environments.

7.6 Module to Mute all Sound Cards

```
def muteVolume(request):
    if request.method == 'POST':
        for i in range(3,10):
            os.system('amixer -c {} set Speaker 0%'.format(i))
        print('Mitral Valve\'s Volume updated to 0%')
        print('Aortic Valve\'s Volume updated to 0%')
        print('Pulmonary Valve\'s Volume updated to 0%')
        print('Tricuspid Valve\'s Volume updated to 0%')
        print('Erb\'s Point\'s Volume updated to 0%')
        print('Lungs\'s Volume updates to 0%')
        print('Bowel\'s Volume updates to 0%')
        return JsonResponse({'message': 'Success!'})
    else:
        return HttpResponse("Request method is not a POST")
```

The Python **muteVolume** function handles muting the volume for various sound locations in a Django app. Upon receiving a ***POST*** request, it iterates through different speaker indices (3 to 9) and sets their volume to **0%**. It prints volume update messages for Mitral Valve, Aortic Valve, Pulmonary Valve, Tricuspid Valve, Erb's Point, Lungs, and Bowel. The function then returns a JSON response indicating success. Note: Using *os.system* for volume control should be done cautiously in production environments due to security considerations.

7.7 Module to set all Sound Cards to default volume

```
def defaultVolume(request):
    if request.method == 'POST':
        for i in range(3,10):
            os.system('amixer -c {} set Speaker 50%'.format(i))
        print('Mitral Valve\'s Volume updated to 50%')
        print('Aortic Valve\'s Volume updated to 50%')
        print('Pulmonary Valve\'s Volume updated to 50%')
        print('Tricuspid Valve\'s Volume updated to 50%')
        print('Erb\'s Point\'s Volume updated to 50%')
        print('Lungs\'s Volume updates to 50%')
        print('Bowel\'s Volume updates to 50%')
        return JsonResponse({'message': 'Success!'})
    else:
        return HttpResponse("Request method is not a POST")
```

The **defaultVolume** function in Python handles setting the volume to **50%** for various sound locations in a Django app. Upon receiving a ***POST*** request, it iterates through different speaker indices (3 to 9) and sets their volume to **50%**. It prints volume update messages for Mitral Valve, Aortic Valve, Pulmonary Valve, Tricuspid Valve, Erb's Point, Lungs, and Bowel. The function then returns a ***JSON*** response indicating success. As mentioned earlier, using *os.system* for volume control should be done cautiously in production environments due to security considerations.

7.8 Module to set all Sound Cards to maximum volume

```
def fullVolume(request):
    if request.method == 'POST':
        for i in range(3,10):
            os.system('amixer -c {} set Speaker 100%'.format(i))
        print('Mitral Valve\'s Volume updated to 100%')
        print('Aortic Valve\'s Volume updated to 100%')
        print('Pulmonary Valve\'s Volume updated to 100%')
        print('Tricuspid Valve\'s Volume updated to 100%')
        print('Erb\'s Point\'s Volume updated to 100%')
        print('Lungs\'s Volume updates to 100%')
        print('Bowel\'s Volume updates to 100%')
        return JsonResponse({'message': 'Success!'})
    else:
        return HttpResponse("Request method is not a POST")
```

The **fullVolume** function in Python is designed to set the volume to **100%** for various sound locations in a Django app. Upon receiving a ***POST*** request, it iterates through different speaker indices (3 to 9) and sets their volume to **100%**. It prints volume update messages for Mitral Valve, Aortic Valve, Pulmonary Valve, Tricuspid Valve, Erb's Point, Lungs, and Bowel. The function then returns a ***JSON*** response indicating success. As mentioned earlier, using *os.system* for volume control should be done cautiously in production environments due to security considerations.

8. Operating System

The Auscultation Simulator operates on Raspberry Pi OS (Legacy, 32-bit), chosen for its seamless compatibility with Raspberry Pi hardware and optimized performance for embedded systems.

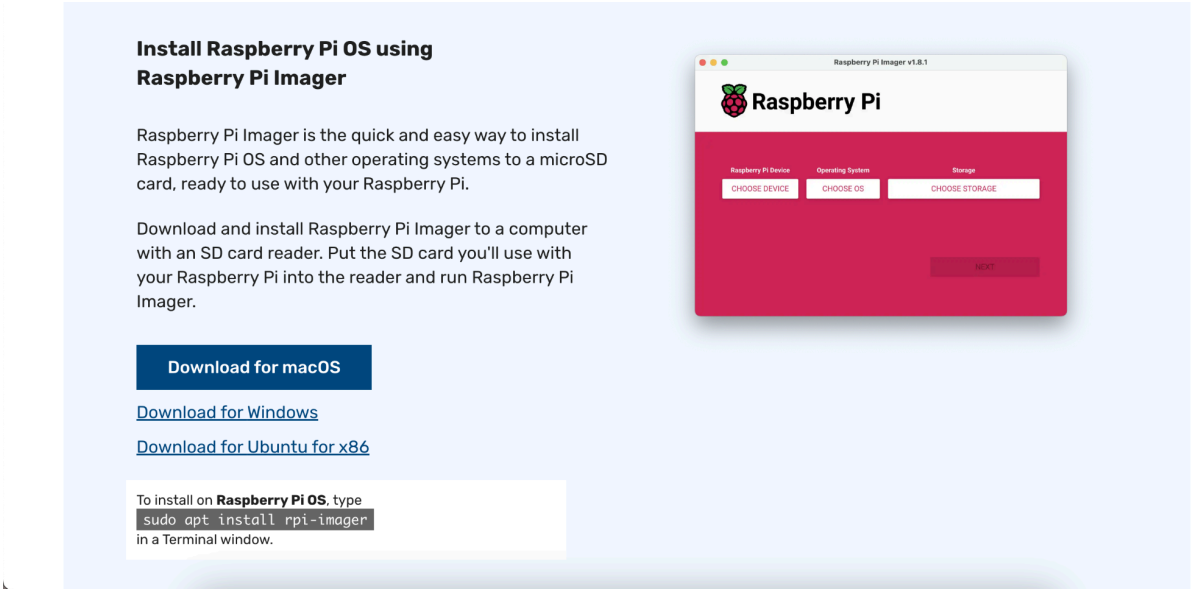
8.1 Installation of Raspberry Pi OS

Raspberry Pi OS, the official operating system for the Raspberry Pi, is based on the Debian Linux distribution. Installing Raspberry Pi OS is a straightforward process, and it can be accomplished using the Raspberry Pi Imager or by flashing the OS image onto an SD card manually.

Using Raspberry Pi Imager:

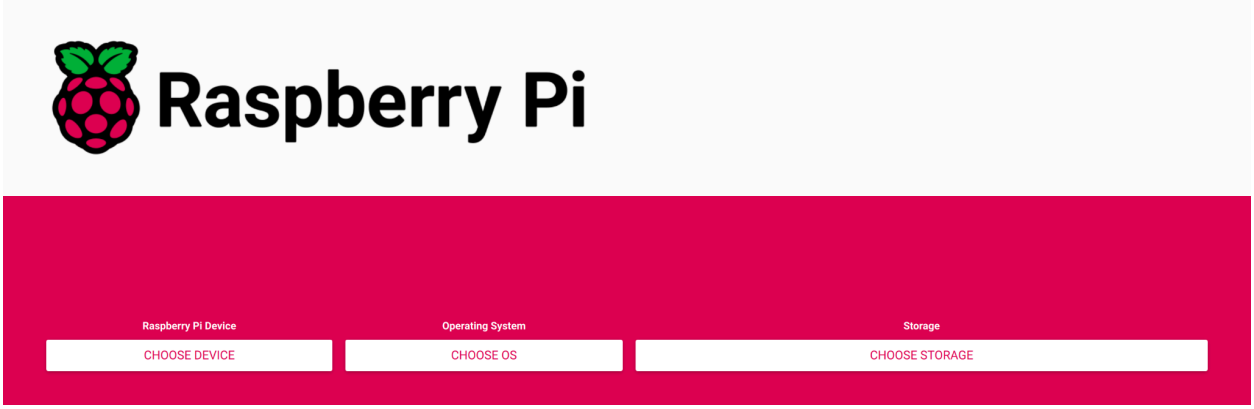
8.1.1 Download Raspberry Pi Imager:

- Visit the official Raspberry Pi website (https://www.raspberrypi.com/software/) and download the Raspberry Pi Imager suitable for your operating system (Windows, macOS, or Linux).



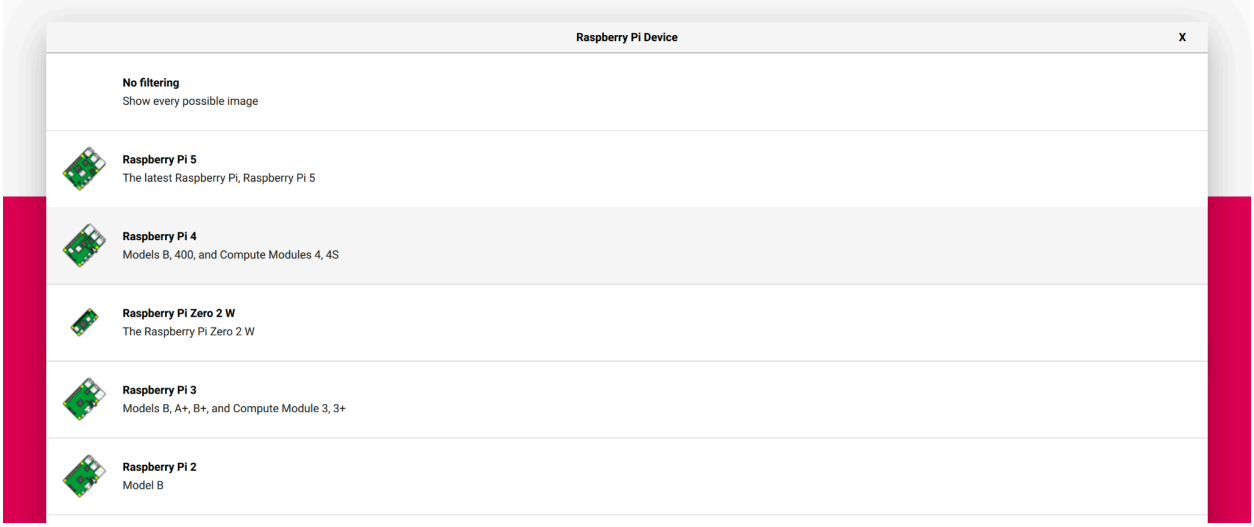
8.1.2 Install and Launch Raspberry Pi Imager:

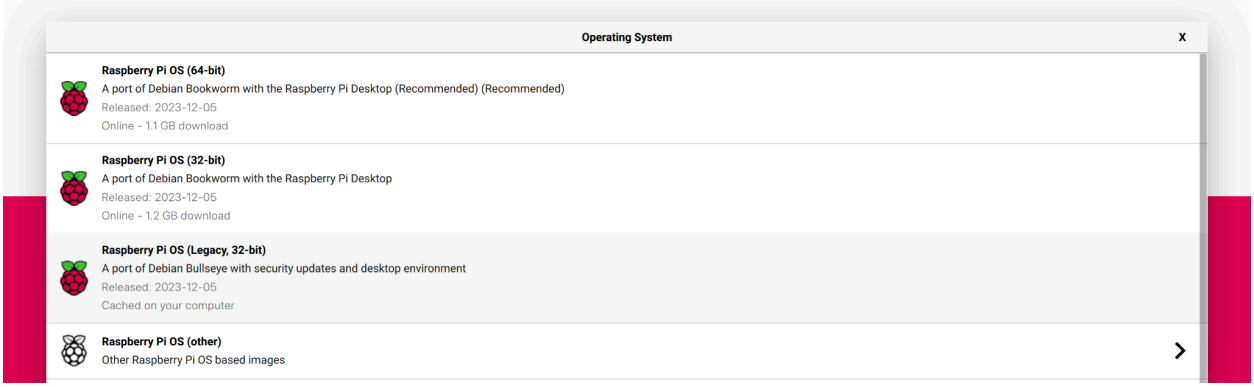
- Follow the installation instructions for your specific operating system.
- Launch the Raspberry Pi Imager once installed.

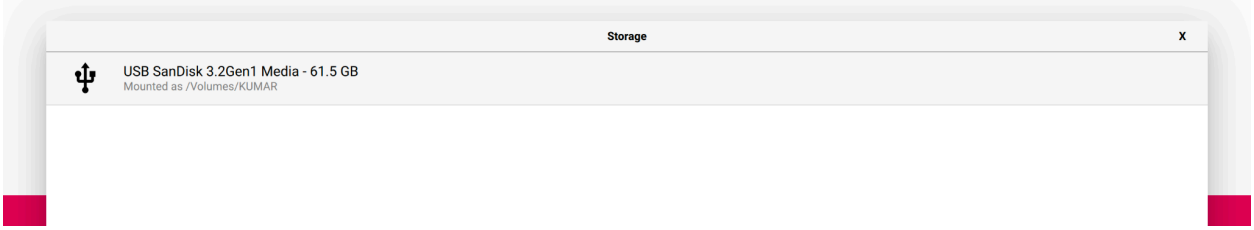


8.1.3 Select OS and Storage:

- In the Imager, click on "Choose OS" and select "**Raspberry Pi OS (Legacy, 32 bits)**" based on your Raspberry Pi model.
- Click on "Choose SD Card" and select the SD card you want to install the OS on.







8.1.4 Write the Image:

- Click on "Write" to begin the process. This will download the selected OS image and write it to the SD card.

8.1.5 Eject SD Card:

- Once the write process is complete, safely eject the SD card from your computer.

8.1.6 Insert SD Card into Raspberry Pi:

- Insert the prepared SD card into the Raspberry Pi's SD card slot.

8.1.7 Power Up:

- Connect peripherals (keyboard, mouse, etc. - IF REQUIRED) and power up the Raspberry Pi.

8.2 Raspberry Pi 4 Troubleshooting

8.2.1 Raspberry Pi ACT LED Error Patterns

If your Raspberry Pi board isn't booting, and the green 'ACT' LED is flashing, count the number of flashes to look up which of the following issues that indicates. Observe the LED warning flash codes:

Long Flashes	Short Flashes	Status
0	3	Generic failure to boot
0	4	start*.elf not found
0	7	Kernel image not found
0	8	SDRAM failure
0	9	Insufficient SDRAM
0	10	In HALT state
2	1	Partition not FAT
2	2	Failed to read from partition
2	3	Extended partition not FAT
2	4	File signature/hash mismatch - Pi 4
4	4	Unsupported board type
4	5	Fatal firmware error
4	6	Power failure type A
4	7	Power failure type B

So:

- If the ACT LED blinks in a regular four blink pattern, it cannot find the bootcode (start.elf).
- If the ACT LED blinks in an irregular pattern, then booting has started.

- If the ACT LED doesn't blink, then the EEPROM code might be corrupted. Try again without anything connected to make sure.

8.2.2 Raspberry Pi PWR LED Error

If the red light on your Raspberry Pi is blinking, this indicates that the power source it is connected to is not providing enough power. This occurs when the supplied voltage is below 4.63 Volts.

The recent models have been equipped with a **Brownout detector**. A brownout occurs when the AC power voltage decreases over time. This can be extremely harmful to the Raspberry Pi if not corrected quickly.

The brownout detector will cause the red LED light to turn off if the Raspberry Pi is not receiving enough power. The brownout detector will also cause the red LED light to turn off if a weak micro USB cable is used.

There are a few indicators that can show up when you're looking at your PWR function. See the list of reasons your red PWR light is flashing in the table below:

Red Light Status	Reason for Lighting
Consistent Blinking	Low power source
Light goes out	"Brownout" detected
Consistently Lit	Adequate power received

8.3 Crontab in Raspberry Pi

8.3.1 What is Cron Jobs?

Cron jobs are an easy way to schedule a script or program to run at specific dates and times in regular recurring cycles. These can be used for server maintenance tasks and scheduled backups, right out to real world applications for things like turning on or off lights and other automation.

8.3.2 Setting up Cron Jobs in Raspberry Pi

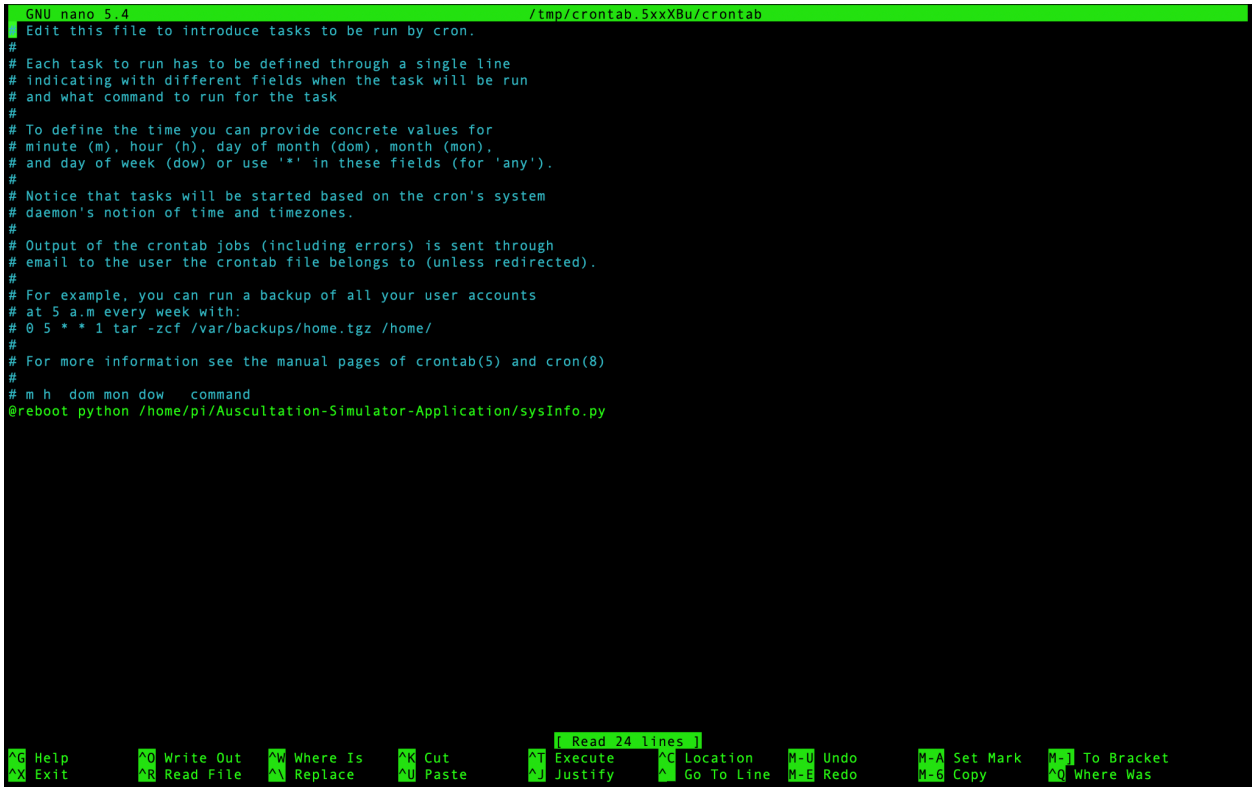
8.3.2.1 How To Create And Edit Cron Jobs

Start by firing up terminal and run the crontab command with the -e flag to edit the table of jobs:

```
crontab -e
```

8.3.2.2 Creating A Scheduled Task

The editor should have now opened – so we can create our first scheduled task.



8.3.2.3 Viewing

If you wish to view your scheduled tasks without editing you can use the command:

```
crontab -l
```

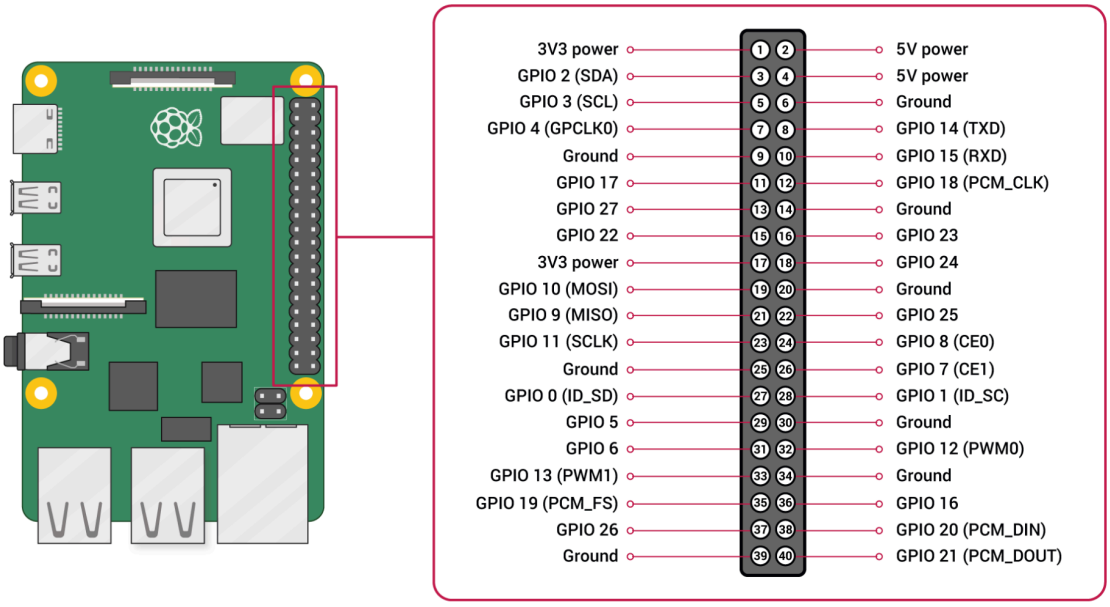
```
pi@raspberrypi:~ $ crontab -l
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
@reboot python /home/pi/Auscultation-Simulator-Application/sysInfo.py
pi@raspberrypi:~ $
```

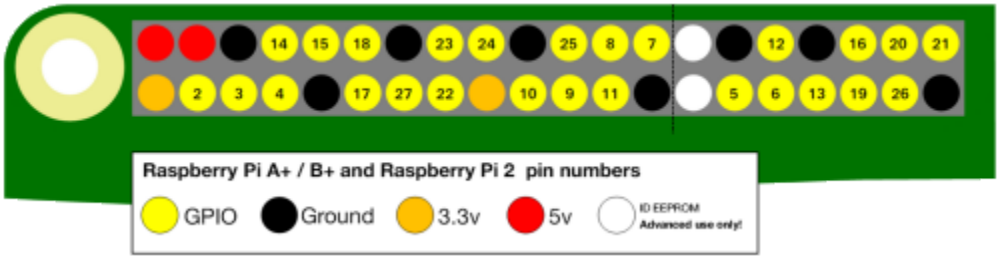
9. GPIO Connection and Pin Configuration

9.1 Introduction to Raspberry Pi's GPIO

9.1.1 What is GPIO?

A powerful feature of the Raspberry Pi is the row of GPIO (general-purpose input/output) pins along the top edge of the board. A 40-pin GPIO header is found on all current Raspberry Pi boards. The GPIO headers on all boards have a 0.1-inch (2.54 mm) pin pitch. Any of the GPIO pins can be designated in software as an input or output pin and used for a wide range of purposes.





9.1.2 Voltages

Two 5V pins and two 3.3V pins are present on the board, as well as a number of ground pins (GND), which can not be reconfigured. The remaining pins are all general-purpose 3.3V pins, meaning outputs are set to 3.3V and inputs are 3.3V tolerant.

9.1.3 Outputs

A GPIO pin designated as an output pin can be set to high (3.3V) or low (0V).

9.1.4 Inputs

A GPIO pin designated as an input pin can be read as high (3.3V) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software.

9.1.5 Other function

As well as simple input and output devices, the GPIO pins can be used with a variety of alternative functions, some are available on all pins, others on specific pins.

- **PWM (Pulse Width Modulation):**
    - Software PWM available on all pins
    - Hardware PWM available on GPIO12, GPIO13, GPIO18 and GPIO19.
- **SPI:**
    - SPI0: MOSI (GPIO10); MISO (GPIO9); SCLK (GPIO11); CE0 (GPIO8), CE1 (GPIO7)
    - SPI1: MOSI (GPIO20); MISO (GPIO19); SCLK (GPIO21); CE0 (GPIO18);CE1 (GPIO17); CE2 (GPIO16)
- **I2C:**
    - Data: (GPIO2); Clock (GPIO3)
    - EEPROM Data: (GPIO0); EEPROM Clock (GPIO1)
- **Serial:**
    - TX (GPIO14); RX (GPIO1)

9.2 Programming for 16x2 LCD Display

9.2.1 Imports:

```
import time
import psutil
from rpi_lcd import LCD
import threading
```

- **time:** Standard Python library for time-related functions.
- **psutil:** A cross-platform library for retrieving system information, including CPU, memory, disks, and network.
- **rpi_lcd:** A Python library for controlling LCD displays on Raspberry Pi.
- **threading:** Standard Python library for multi-threading.

9.2.2 Create LCD Object:

```
lcd = LCD()
```

- Creates an instance of the LCD class from the **rpi_lcd** library.

9.2.3 Functions for System Information:

```
def get_cpu_usage():
    ...

def get_ram_info():
    ...

def get_disk_info():
    ...

def get_ip_address():
    ...
```

- These functions use **psutil** to gather CPU usage, RAM usage, disk usage, and IP address information.

9.2.4 Function to Update and Scroll System Information:

```
def update_system_info(delay=1):
    ...
```

- Scrolls and updates the LCD with CPU, RAM, and disk information in a loop.

9.2.5 Function to Display IP Address:

```
def display_ip_address():
    ...
```

- Displays the IP address on the second line of the LCD in a loop.

9.2.6 Multithreading:

```
update_thread = threading.Thread(target=update_system_info)
ip_thread = threading.Thread(target=display_ip_address)
```

- Creates two threads for updating system information and displaying the IP address.

9.2.7 Thread Execution:

```
try:
    update_thread.start()
    ip_thread.start()
    update_thread.join()
    ip_thread.join()
except KeyboardInterrupt:
    lcd.clear()
```

- Starts the threads and waits for a keyboard interrupt to clear the LCD and exit gracefully.

9.3 Programming for Buzzer

9.3.1 Import Statements:

```
from gpiozero import Buzzer
```

- Imports the Buzzer class from the gpiozero library, which simplifies working with GPIO components.

9.3.2 Create Buzzer Object:

```
try:
    buzzer = Buzzer(17)
```

```
except:
    buzzer = None
```

- Attempts to create a Buzzer object connected to GPIO pin 17. If an exception occurs (for example, if the pin is already in use or not available), it sets the buzzer to None.

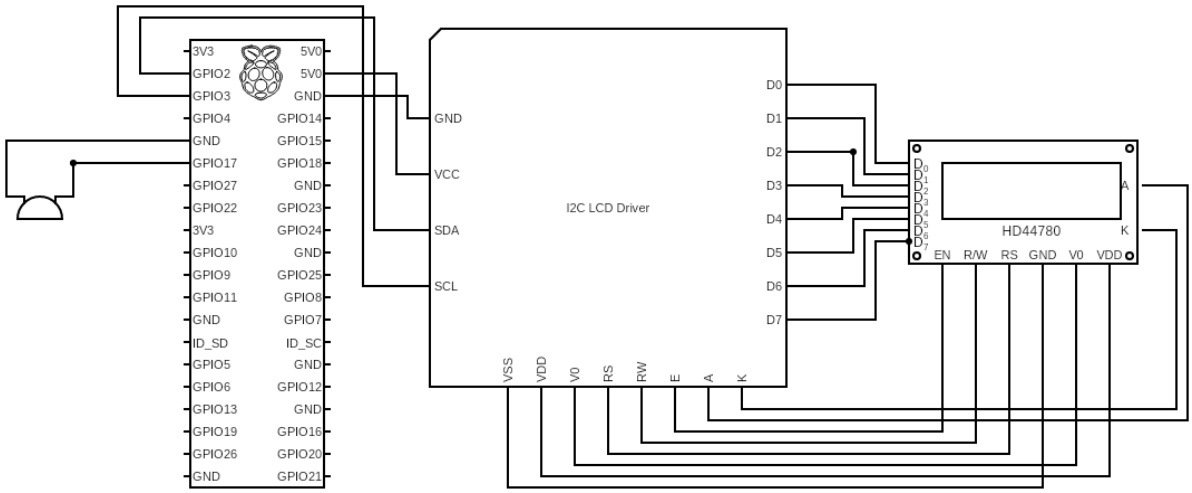
9.3.3 Turn Buzzer On and Off:

```
buzzer.on()
time.sleep(0.1)
buzzer.off()
```

- Turn the buzzer on for a short duration (0.1 seconds) and then turn it off.

NOTE: The time module needs to be imported for the time.sleep function. Ensure that you have import time at the beginning of your script.

9.4 GPIO Connection for 16x2 Display & Buzzer



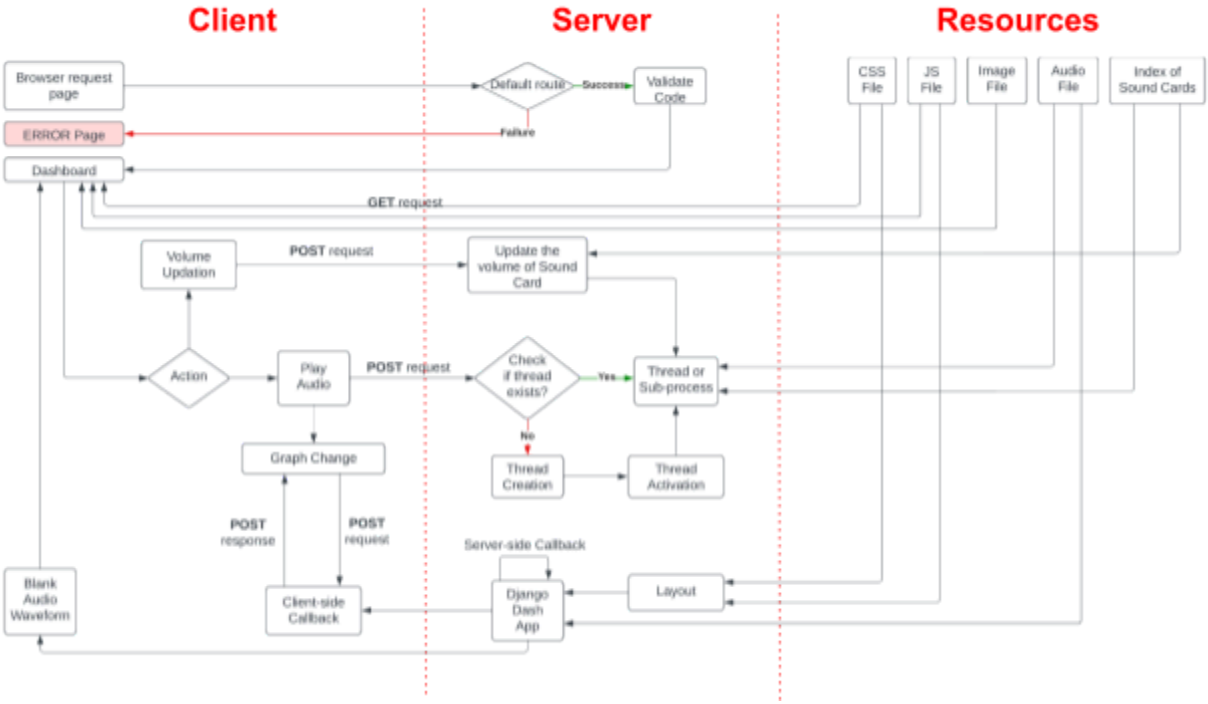
9.4.1 GPIO Connection for 16x2 Display:

- GND → GND (pin 6)
- VCC → 5V (pin 4)
- SDA → GPIO2 (pin 3)
- SCL → GPIO3 (pin 5)

9.4.2 GPIO Connection for Buzzer:

- GND → GND (pin 9)
- VCC → GPIO17 (pin 11)

10. Overall Workflow



10.1 HTTP Request

10.1.1 GET Request

- Retrieving data from a server, navigating through links, bookmarking, and search engine requests.
- **Purpose:** Used to request data from a specified resource.

- **Parameters:** Data is sent as part of the URL (query parameters). For example: example.com/resource?param1=value1&param2=value2
- **Security:** Parameters are visible in the URL, which can be a security concern for sensitive information.
- **Caching:** Responses can be cached, and the request is idempotent (multiple identical requests have the same effect as a single request).

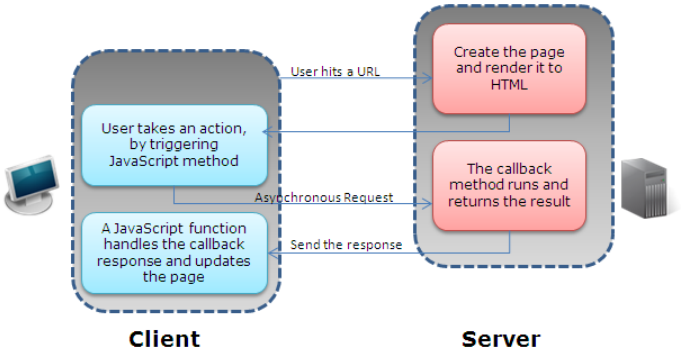
10.1.2 POST Request

- Submitting forms, uploading files, and any other situation where data needs to be sent to the server for processing.
- **Purpose:** Used to send data to be processed to a specified resource.
- **Parameters:** Data is sent in the request body. It can include various types of data, like form data, JSON, or binary data.
- **Security:** Parameters are not visible in the URL, making it more suitable for sensitive or large amounts of data.
- **Caching:** Responses are typically not cached, and the request is not idempotent.

10.2 Callbacks

10.2.1 Client-side Callback

Client-side callbacks refer to asynchronous operations or functions in a web application that are initiated and executed on the client side (in the user's browser). These operations are typically triggered by user interactions or events such as button clicks, form submissions, or timers. JavaScript, especially in combination with frameworks like React or Angular, is commonly used to implement client-side callbacks.



Examples of client-side callbacks include handling events like button clicks, making AJAX (Asynchronous JavaScript and XML) requests to fetch data from a server without reloading the entire page, or responding to user interactions in real-time.

10.2.2 Server-side Callback

Server-side callbacks, on the other hand, involve asynchronous operations that are initiated and executed on the server side. These operations are often triggered by events such as receiving an HTTP request from a client, processing data, interacting with a database, or performing other server-side computations.

In the context of web development, server-side callbacks are crucial for handling form submissions, updating databases, authenticating users, and other operations that require server-side processing. Server-side frameworks and languages like Flask (Python), Express (Node.js), Django (Python), Ruby on Rails, and many others are commonly used to implement server-side callbacks.

11. Future Works

Several potential enhancements and integrations are highlighted to further enhance the capabilities and features of the existing system. These future developments include:

11.1 Advanced IoT, Embedded, and Electronics Integration:

- **Proposal:** Integrating a custom board with multiple AUX ports to extend the functionality of the Raspberry Pi.
- **Integration with GPIO:** Connecting the custom board to Raspberry Pi's GPIO for enhanced input capabilities.
- Integration of cooling fan on Raspberry Pi to prevent overheating of device to avoid drop in performance due to heating.

- Usage of specialized casing for Raspberry Pi to avoid direct contact of Raspberry Pi with bare hands. Direct contact of Raspberry Pi with bare hands could lead to damage of Raspberry Pi due to static charge.

11.2 Implementation of Advanced Cybersecurity Features:

- **Proposal:** Strengthening the security of the IoT device, especially when connected to WLAN, by implementing advanced cybersecurity measures.
- **Focus on Protection:** Addressing potential external threats and ensuring the integrity and confidentiality of data.

11.3 Addition of Personalized Training Platform for Auscultation:

- **Proposal:** Introducing an additional dashboard for a personalized training platform in Auscultation.
- **Custom Study Courses:** Allowing users to design and practice their own study courses.
- **Testing Features:** Incorporating the ability to design and take tests for a more comprehensive learning experience.

11.4 Improvement in Graph Rendering and Audio Waveform:

- **Proposal:** Enhancing the rendering of graphs and audio waveforms to address current performance limitations.
- **Focus on User Experience:** Improving the visual representation of data for a smoother and more responsive user interface.

11.5 Implementation of AI and Machine Learning Features:

- **Proposal:** Integrating AI and machine learning features to enhance the overall performance of the application.
- **Adaptive Functionality:** Leveraging AI for adaptive learning experiences and intelligent insights.