# Blood Bridge
# Optimizing Lifesaving Resources using AWS services

**Project Description:**

Blood Bridge is a web-based blood bank management system designed to streamline and optimize blood donation and distribution processes. It leverages Amazon Web Services (AWS), including Amazon RDS for secure and scalable data storage and Amazon EC2 for reliable and efficient web hosting, ensuring high availability and security.

The system provides a user-friendly interface where individuals, hospitals, and blood banks can register, log in, and access a centralized dashboard to manage blood-related activities. Users can view real-time blood requests, submit their own requests, and specify blood type, quantity, and urgency. Blood Bridge enables donors to schedule donations, track their eligibility, and receive automated notifications, ensuring a seamless and efficient blood donation process.

To enhance accessibility and efficiency, Blood Bridge integrates cloud-based automation, real-time data tracking, and secure authentication protocols, ensuring data privacy and smooth communication between donors and recipients. Additionally, the platform encourages community engagement by connecting volunteers with critical blood donation opportunities, reducing response time in emergency situations. Future enhancements include AI-powered donor matching, predictive analytics for blood demand forecasting, and blockchain-based security for tamper-proof donor and recipient records. By leveraging cutting-edge cloud technology and an intuitive, data-driven approach, Blood Bridge improves healthcare outcomes, strengthens emergency response capabilities, and fosters a culture of life-saving contributions**.**

## Scenario 1: Emergency Blood Request:

During a critical situation, Sarah, a hospital administrator, logs into LifeLink to request a rare blood type for a patient in urgent need. Using her dashboard, she submits a high-priority request, specifying the blood type and quantity required. The system instantly notifies nearby potential donors, accelerating the matching process and minimizing response time, ultimately increasing the chances of saving the patient's life.

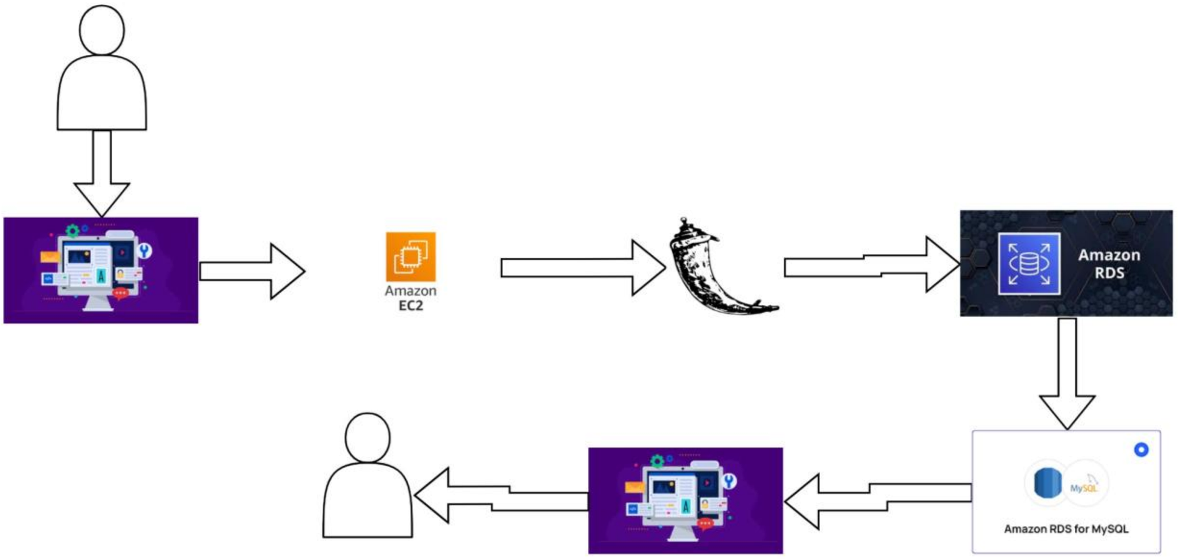
## Scenario 2: Regular Donor Management

John, a dedicated blood donor, logs into **Life Link**, a smart blood donation platform, to check his **eligibility status** for his next donation. As he navigates his personalized dashboard, he receives a **real-time alert** about an upcoming **community blood drive** in his area. With just a few clicks, he seamlessly **schedules his next donation**, ensuring his contribution reaches those who need it most.

## Scenario 3: Blood Bank Inventory Update

Lisa, a dedicated **blood bank manager**, logs into **Life Link**, a smart blood management platform, to **update and monitor blood inventory in real time**. Using her **specialized administrative account**, she efficiently inputs the latest **stock levels**, ensuring that the system reflects the most up-to-date availability for donors, hospitals, and emergency responders. As soon as she updates the inventory, the platform **instantly syncs the data across all connected users**, enabling **efficient blood distribution** by prioritizing **urgent requests for low-stock blood types**. With **real-time tracking, automated alerts, and intelligent demand forecasting**, Life Link helps Lisa **streamline operations**, reduce shortages, and ensure that **life-saving blood resources** reach patients **when and where they are needed most**.

## Architecture:



## Prior Knowledge:

1. AWS Account Setup: https://youtu.be/CjKhQoYeR4Q?si=ui8Bvk_M4FfVM-Dh

2.Web Application Stack : FLask || MySQL Connector using flask || HTML/JS/CSS

3. AWS EC2 Instance: https://www.youtube.com/results?search_query=aws+ec2+oneshot

4.RDS Database: https://www.youtube.com/results?search_query=rds+oneshot

5.MySQL: https://www.youtube.com/results?search_query=mysql+tutorial

6.RDSconnectsMySQL:https://www.youtube.com/results?search_query=mysql+connector+for+rds

7.CloneGit repo: https://www.youtube.com/results?search_query=clone+github+repository

8.AWS Cost Management: https://youtu.be/OKYJCHHSWb4?si=aY3DQl1v26CfZxXA

## Project Flow:

### Project Initialization:

- Define objectives, scope, and KPIs; set up the AWS environment.

### EC2 Instance Setup:

- Launch and configure an EC2 instance to host the web application.

### RDS Database Setup:

- Create and configure an RDS instance with MySQL engine.

### Web Application Development:

- Develop the web application with registration, login, and dashboard features.

### Database Integration:

- Connect the web application to the RDS database using appropriate drivers.

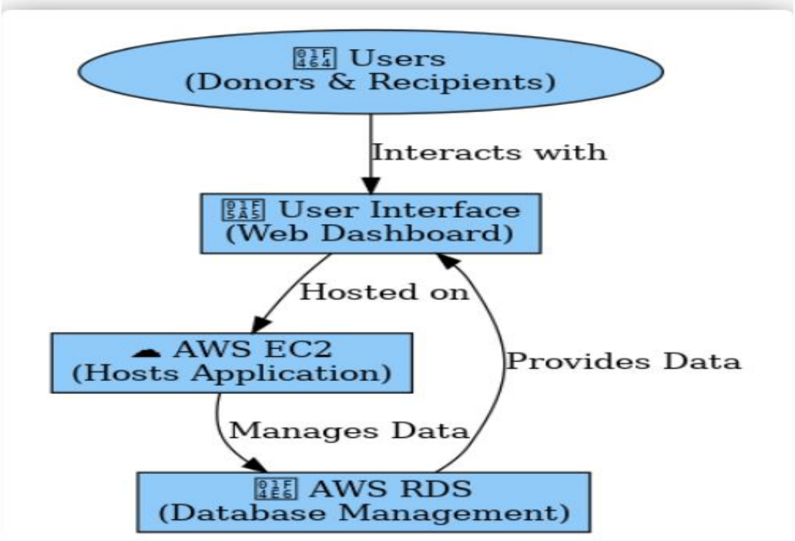
### User Interface Implementation:

- Create user-friendly interfaces for registration, login, and blood request management.

### Testing and Optimization:

- Conduct thorough testing of all features and optimize for performance.

Technical Diagram:

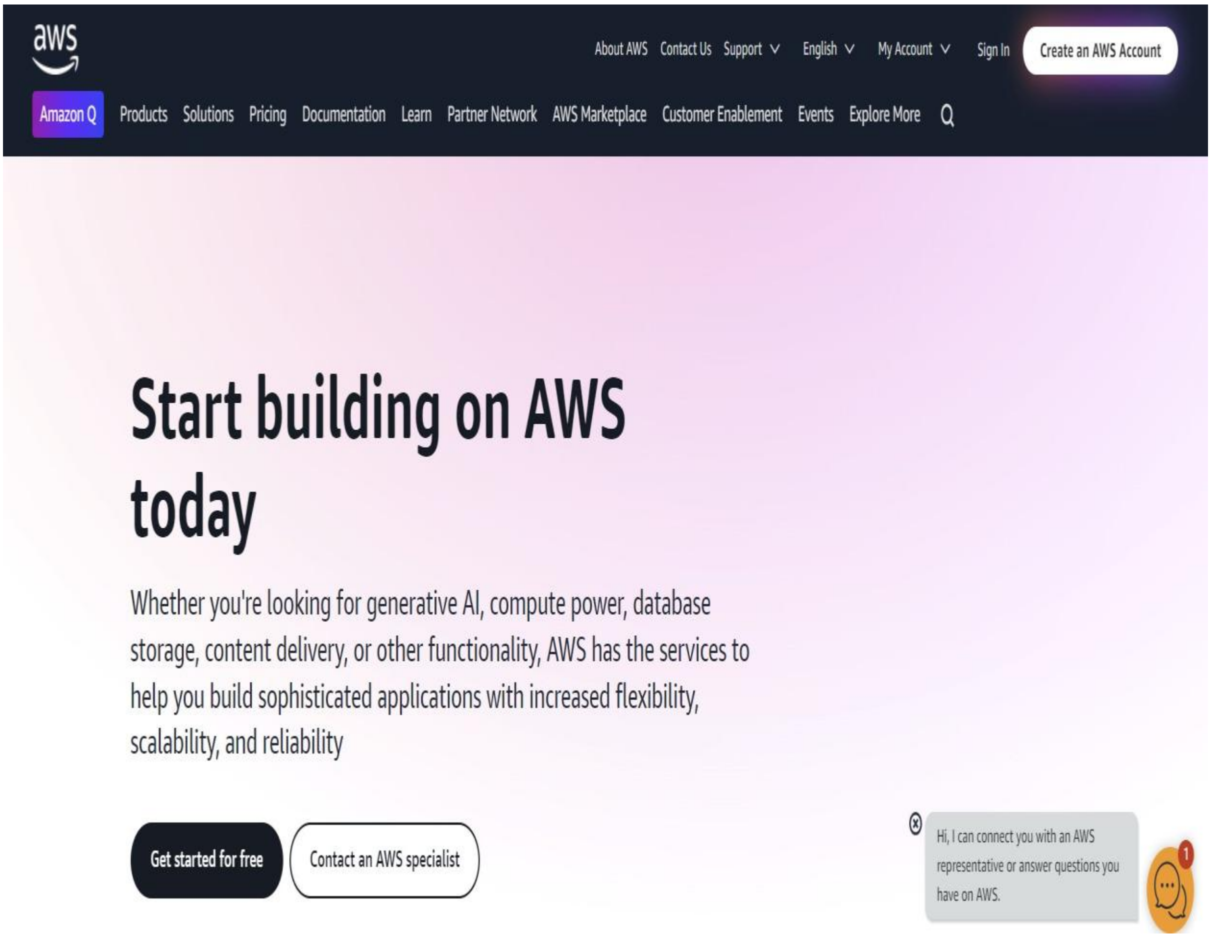


## Milestone 1: AWS Account Creation

In this milestone, we will set up an AWS account to access the necessary services for the BloodBridge project.

### Activity 1: Create AWS Account

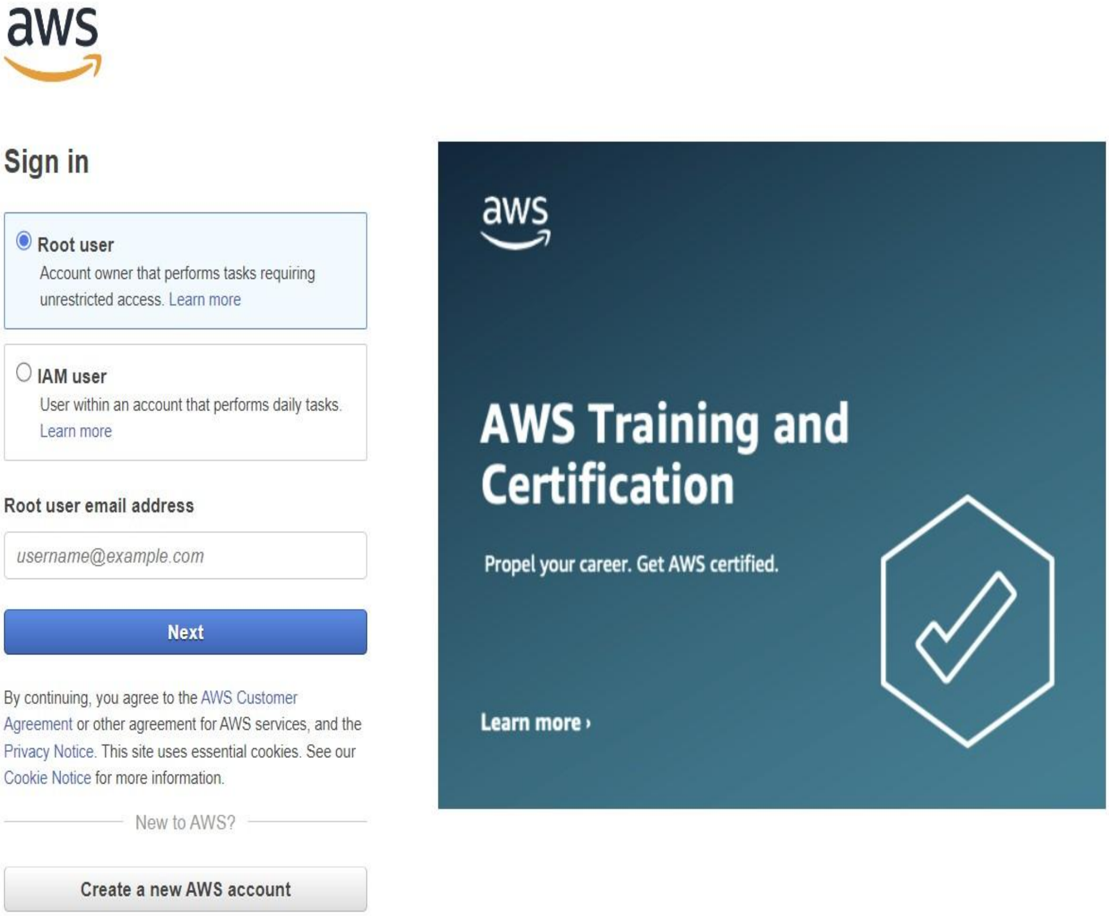
1. Go to the AWS website (https://aws.amazon.com/).
2. Click on "Create an AWS Account" button.
3. Follow the prompts to enter your email address and choose a password.
4. Provide the required account information, including your name, address, and phone number.
5. Enter your payment information. (Note: While AWS offers a free tier, a credit card or debit card is required for verification.)
6. Complete the identity verification process.
7. Choose a support plan (the basic plan is free and sufficient for starting).
8. Once verified, you can sign in to your new AWS account.



Start building on AWS today

Whether you're looking for generative AI, compute power, database storage, content delivery, or other functionality, AWS has the services to help you build sophisticated applications with increased flexibility, scalability, and reliability

Hi, I can connect you with an AWS representative or answer questions you have on AWS.



Sign in

● Root user
Account owner that performs tasks requiring unrestricted access. Learn more

○ IAM user
User within an account that performs daily tasks. Learn more

Root user email address

username@example.com

By continuing, you agree to the AWS Customer Agreement or other agreement for AWS services, and the Privacy Notice. This site uses essential cookies. See our Cookie Notice for more information.

New to AWS?

AWS Training and Certification

Propel your career. Get AWS certified.

## Milestone 2: Set Up AWS Environment

In this milestone, we will create and configure an EC2 instance to host the BloodBridge web application.

### Activity 1.1 Create and Configure an Amazon EC2 Instance

1. Access EC2 Console: In the AWS Management Console, go to the EC2 service.
2. Launch Instance: Click on "Launch Instance" and follow the wizard:
   - Choose an Amazon Machine Image (AMI) suitable for your web application (e.g., Amazon Linux 2).
   - Select an instance type (e.g., t2.micro for testing).
   - Configure instance details, including network settings.
   - Add storage as needed.
   - Add tags for better resource management.
   - Configure security group to allow HTTP/HTTPS traffic.

3. Review and Launch: Review your instance configuration and launch it, selecting or creating a key pair for SSH access.

### Activity 1.2: Configure Security Groups

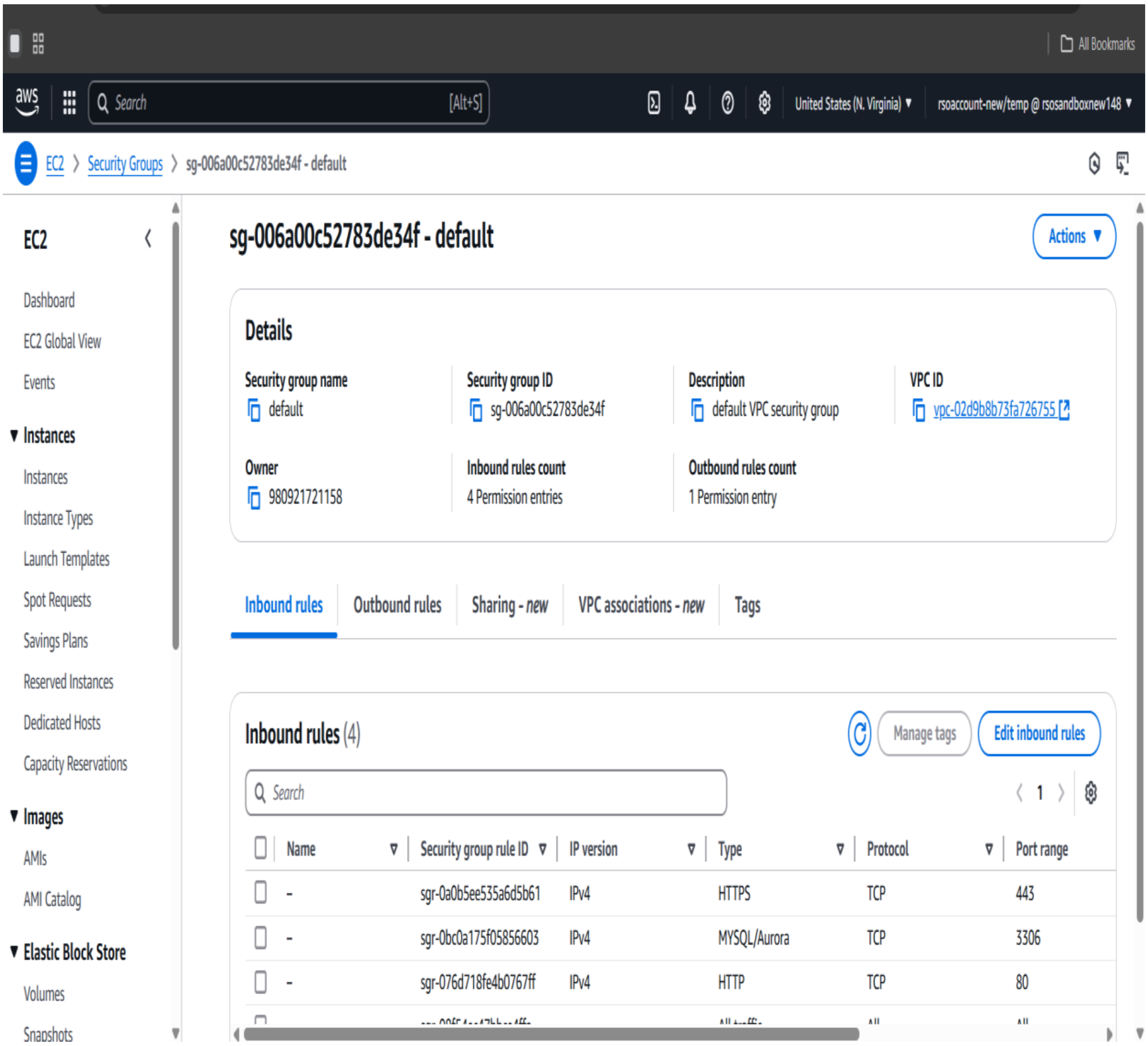
Add a Security Group:

Allow SSH (port 22) from your IP for remote access.

Allow HTTP (port 80) and HTTPS (port 443) to access your web application.

If you have other requirements (e.g., specific port for your Flask application), add those as well.

### Activity 1.3: Launch the Instance



sg-006a00c52783de34f - default

Details

Security group name: default

Security group ID: sg-006a00c52783de34f

Description: default VPC security group

VPC ID: vpc-02d9b8b73fa726755

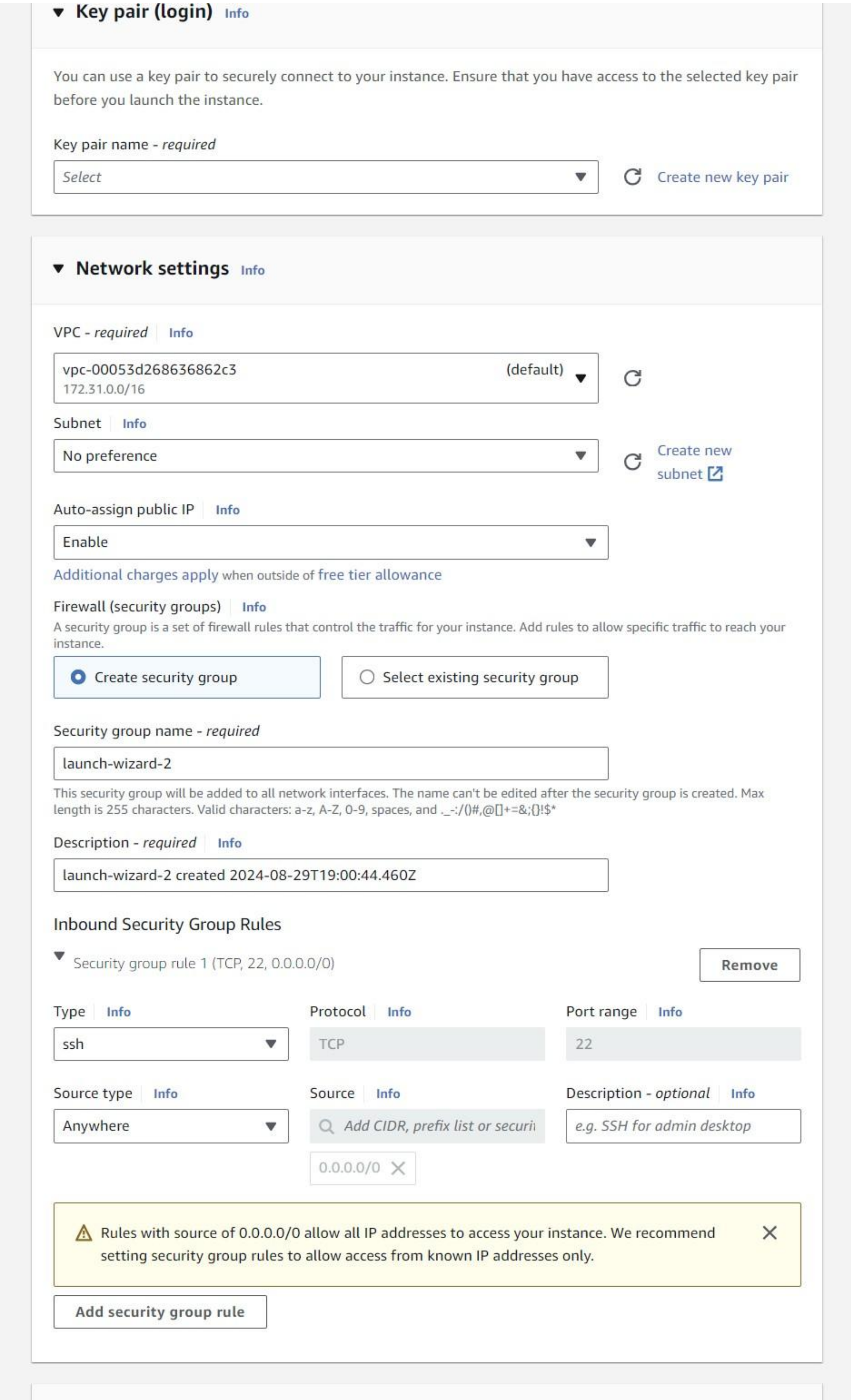
Owner: 980921721158

Inbound rules count: 4 Permission entries

Outbound rules count: 1 Permission entry

Inbound rules (4)

| □ | Name | Security group rule ID | IP version | Type | Protocol | Port range |
|---|---|---|---|---|---|---|
| □ | - | sgr-0a0b5ee535a6d5b61 | IPv4 | HTTPS | TCP | 443 |
| □ | - | sgr-0bc0a175f05856603 | IPv4 | MYSQL/Aurora | TCP | 3306 |
| □ | - | sgr-076d718fe4b0767ff | IPv4 | HTTP | TCP | 80 |
| □ |  | [illegible] |  | All traffic | All | All |



▼ **Key pair (login)** Info

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - *required*

*Select* ▼

▼ **Network settings** Info

VPC - *required* | Info

vpc-00053d268636862c3 (default) ▼
172.31.0.0/16

Subnet | Info

No preference ▼

Auto-assign public IP | Info

Enable ▼

Additional charges apply when outside of free tier allowance

Firewall (security groups) | Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

● Create security group ○ Select existing security group

Security group name - *required*

launch-wizard-2

This security group will be added to all network interfaces. The name can't be edited after the security group is created. Max length is 255 characters. Valid characters: a-z, A-Z, 0-9, spaces, and ._-:/()#,@[]+=&;{}!$*

Description - *required* | Info

launch-wizard-2 created 2024-08-29T19:00:44.460Z

Inbound Security Group Rules

▼ Security group rule 1 (TCP, 22, 0.0.0.0/0)

Type | Info: ssh ▼

Protocol | Info: TCP

Port range | Info: 22

Source type | Info: Anywhere ▼

Source | Info: *Add CIDR, prefix list or securi*
0.0.0.0/0 ✕

Description - *optional* | Info: *e.g. SSH for admin desktop*

⚠ Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only. ✕

## Milestone 3: Setting up RDS Database

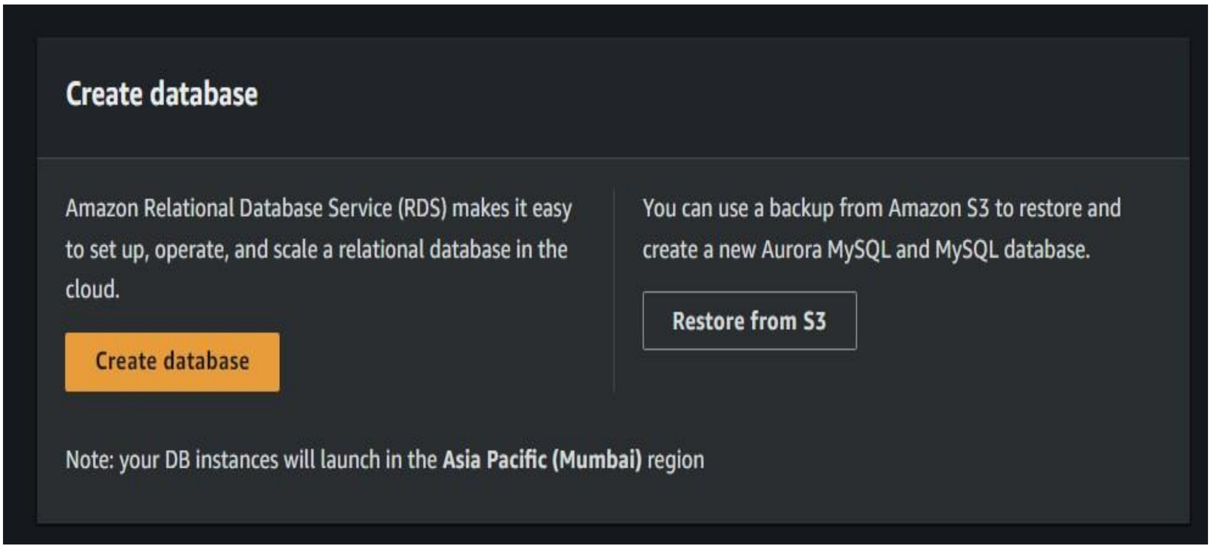
In this milestone, we will create and configure an RDS instance with MySQL to store and manage BloodBridge data.

### Activity 1: Create RDS Instance [RDS]

1. Access RDS Console: From the AWS Management Console, go to the RDS service.
2. Create Database: Click on "Create database" and follow the wizard: ○ Choose MySQL as the engine type.
   ○ Select the appropriate version and instance size.
   ○ Configure storage, network settings, and security groups.
   ○ Set up the master username and password.
   ○ VPC and Subnet: Ensure the RDS instance is in the same VPC as your EC2 instance.
   ○ Public Accessibility: Enable this option if you need direct access from outside the VPC (not recommended for production).
   ○ Security Group: Create or use an existing security group that allows MySQL traffic (default port 3306).
   ○ Set an initial database name (e.g., `bloodbank`).
3. Review and Create: Review your database configuration and create the instance.

### Activity 2: Configure Security Group

1. Once your RDS instance is created, go to its details page.
2. In the "Connectivity & security" tab, click on the VPC security group.
3. Add an inbound rule to allow MySQL/Aurora traffic (port 3306) from your IP address for now (we'll update this later to only allow traffic from the EC2 instance). And add rule to allow inbound traffic from everywhere.

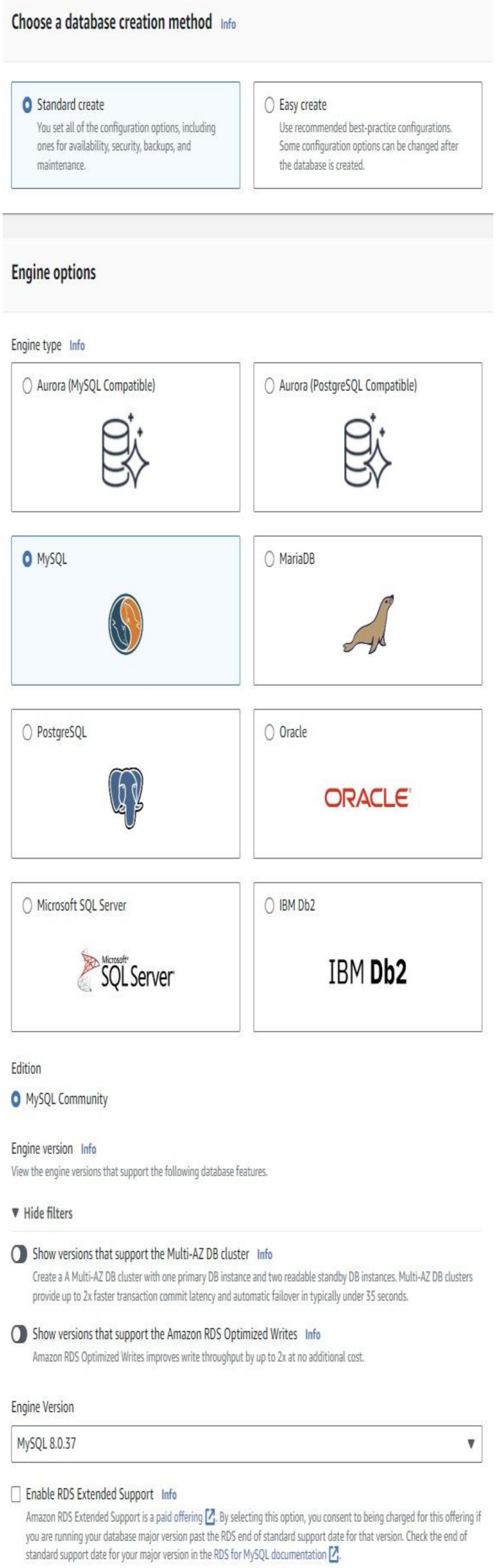


Create database

Amazon Relational Database Service (RDS) makes it easy to set up, operate, and scale a relational database in the cloud.

You can use a backup from Amazon S3 to restore and create a new Aurora MySQL and MySQL database.

Note: your DB instances will launch in the **Asia Pacific (Mumbai)** region



## Choose a database creation method Info

● Standard create
You set all of the configuration options, including ones for availability, security, backups, and maintenance.

○ Easy create
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

## Engine options

Engine type Info

○ Aurora (MySQL Compatible)

○ Aurora (PostgreSQL Compatible)

● MySQL

○ MariaDB

○ PostgreSQL

○ Oracle

○ Microsoft SQL Server

○ IBM Db2

Edition

● MySQL Community

Engine version Info
View the engine versions that support the following database features.

▼ Hide filters

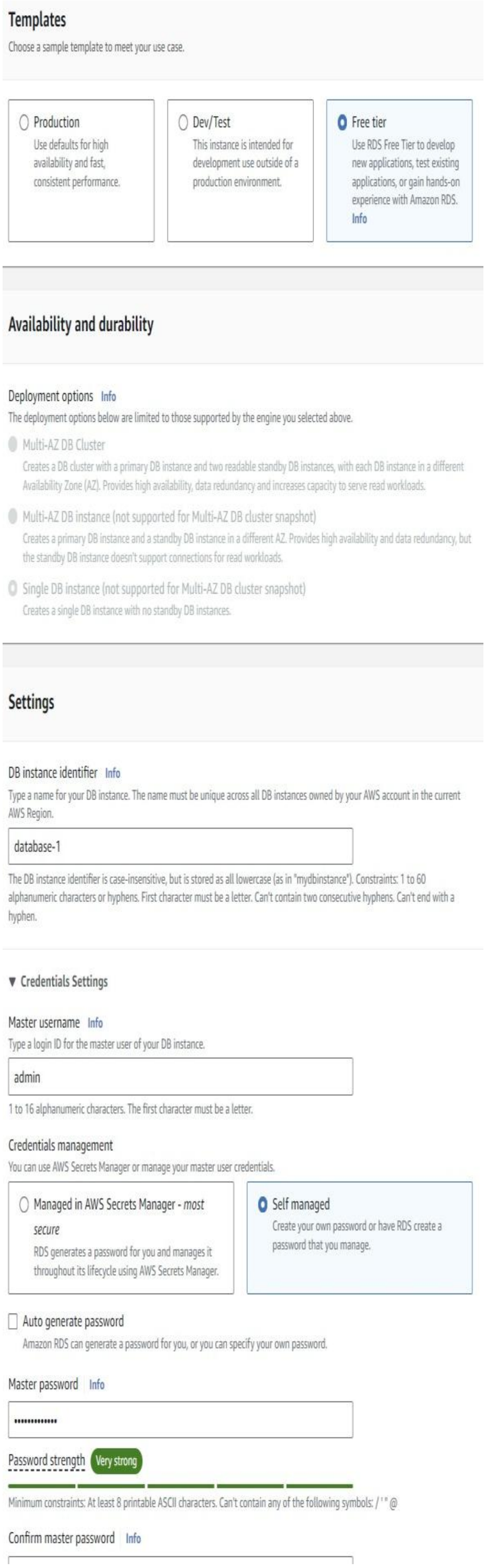
Show versions that support the Multi-AZ DB cluster Info
Create a A Multi-AZ DB cluster with one primary DB instance and two readable standby DB instances. Multi-AZ DB clusters provide up to 2x faster transaction commit latency and automatic failover in typically under 35 seconds.

Show versions that support the Amazon RDS Optimized Writes Info
Amazon RDS Optimized Writes improves write throughput by up to 2x at no additional cost.

Engine Version

MySQL 8.0.37 ▼

□ Enable RDS Extended Support Info
Amazon RDS Extended Support is a paid offering. By selecting this option, you consent to being charged for this offering if you are running your database major version past the RDS end of standard support date for that version. Check the end of standard support date for your major version in the RDS for MySQL documentation.



## Templates

Choose a sample template to meet your use case.

○ Production
Use defaults for high availability and fast, consistent performance.

○ Dev/Test
This instance is intended for development use outside of a production environment.

● Free tier
Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS. Info

## Availability and durability

Deployment options Info
The deployment options below are limited to those supported by the engine you selected above.

○ Multi-AZ DB Cluster
Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.

○ Multi-AZ DB instance (not supported for Multi-AZ DB cluster snapshot)
Creates a primary DB instance and a standby DB instance in a different AZ. Provides high availability and data redundancy, but the standby DB instance doesn't support connections for read workloads.

○ Single DB instance (not supported for Multi-AZ DB cluster snapshot)
Creates a single DB instance with no standby DB instances.

## Settings

DB instance identifier Info
Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

database-1

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username Info
Type a login ID for the master user of your DB instance.

admin

1 to 16 alphanumeric characters. The first character must be a letter.

Credentials management
You can use AWS Secrets Manager or manage your master user credentials.

○ Managed in AWS Secrets Manager - *most secure*
RDS generates a password for you and manages it throughout its lifecycle using AWS Secrets Manager.

● Self managed
Create your own password or have RDS create a password that you manage.

□ Auto generate password
Amazon RDS can generate a password for you, or you can specify your own password.

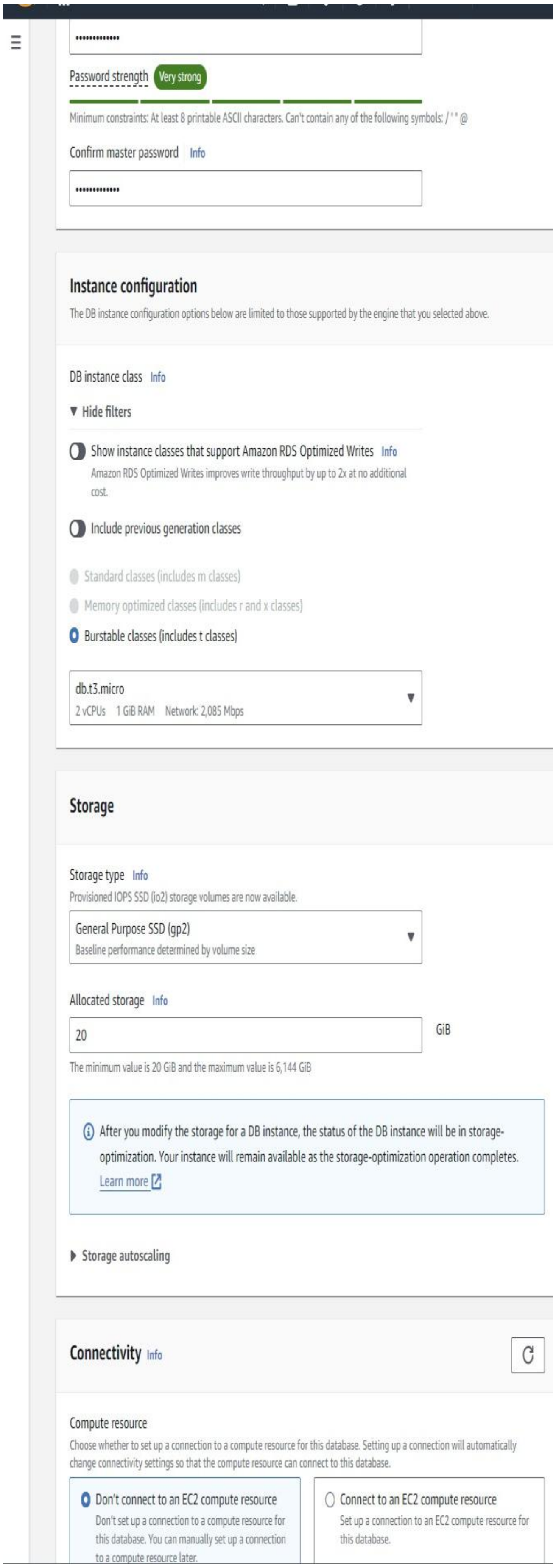
Master password Info

••••••••••••

Password strength Very strong

Minimum constraints: At least 8 printable ASCII characters. Can't contain any of the following symbols: / ' " @

Confirm master password Info



•••••••••••••

Password strength Very strong

Minimum constraints: At least 8 printable ASCII characters. Can't contain any of the following symbols: / ' " @

Confirm master password Info

•••••••••••••

## Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

▼ Hide filters

Show instance classes that support Amazon RDS Optimized Writes Info

Amazon RDS Optimized Writes improves write throughput by up to 2x at no additional cost.

Include previous generation classes

○ Standard classes (includes m classes)

○ Memory optimized classes (includes r and x classes)

● Burstable classes (includes t classes)

db.t3.micro ▼
2 vCPUs 1 GiB RAM Network: 2,085 Mbps

## Storage

Storage type Info

Provisioned IOPS SSD (io2) storage volumes are now available.

General Purpose SSD (gp2) ▼
Baseline performance determined by volume size

Allocated storage Info

20 GiB

The minimum value is 20 GiB and the maximum value is 6,144 GiB

After you modify the storage for a DB instance, the status of the DB instance will be in storage-optimization. Your instance will remain available as the storage-optimization operation completes. Learn more

▶ Storage autoscaling

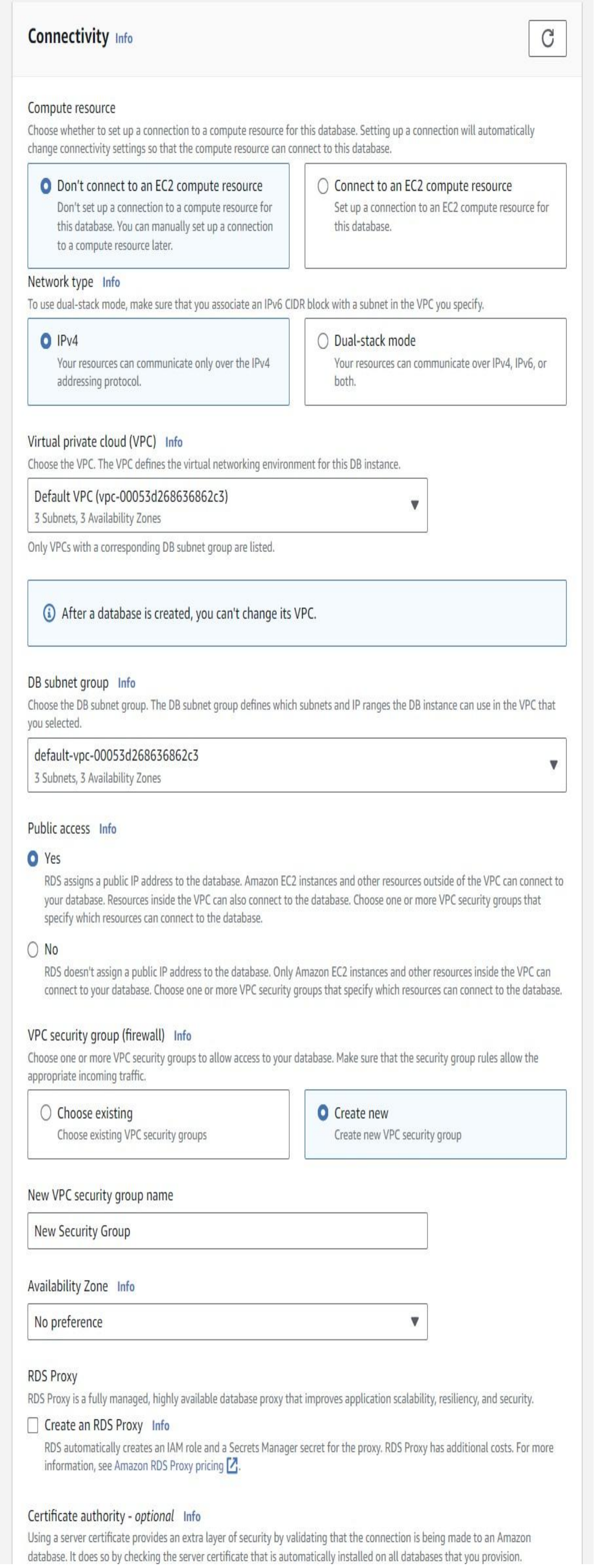
## Connectivity Info

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

● Don't connect to an EC2 compute resource
Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

○ Connect to an EC2 compute resource
Set up a connection to an EC2 compute resource for this database.



## Connectivity Info

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

● Don't connect to an EC2 compute resource
Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

○ Connect to an EC2 compute resource
Set up a connection to an EC2 compute resource for this database.

Network type Info

To use dual-stack mode, make sure that you associate an IPv6 CIDR block with a subnet in the VPC you specify.

● IPv4
Your resources can communicate only over the IPv4 addressing protocol.

○ Dual-stack mode
Your resources can communicate over IPv4, IPv6, or both.

Virtual private cloud (VPC) Info

Choose the VPC. The VPC defines the virtual networking environment for this DB instance.

Default VPC (vpc-00053d268636862c3) ▼
3 Subnets, 3 Availability Zones

Only VPCs with a corresponding DB subnet group are listed.

After a database is created, you can't change its VPC.

DB subnet group Info

Choose the DB subnet group. The DB subnet group defines which subnets and IP ranges the DB instance can use in the VPC that you selected.

default-vpc-00053d268636862c3 ▼
3 Subnets, 3 Availability Zones

Public access Info

● Yes
RDS assigns a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.

○ No
RDS doesn't assign a public IP address to the database. Only Amazon EC2 instances and other resources inside the VPC can connect to your database. Choose one or more VPC security groups that specify which resources can connect to the database.

VPC security group (firewall) Info

Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

○ Choose existing
Choose existing VPC security groups

● Create new
Create new VPC security group

New VPC security group name

New Security Group

Availability Zone Info

No preference ▼

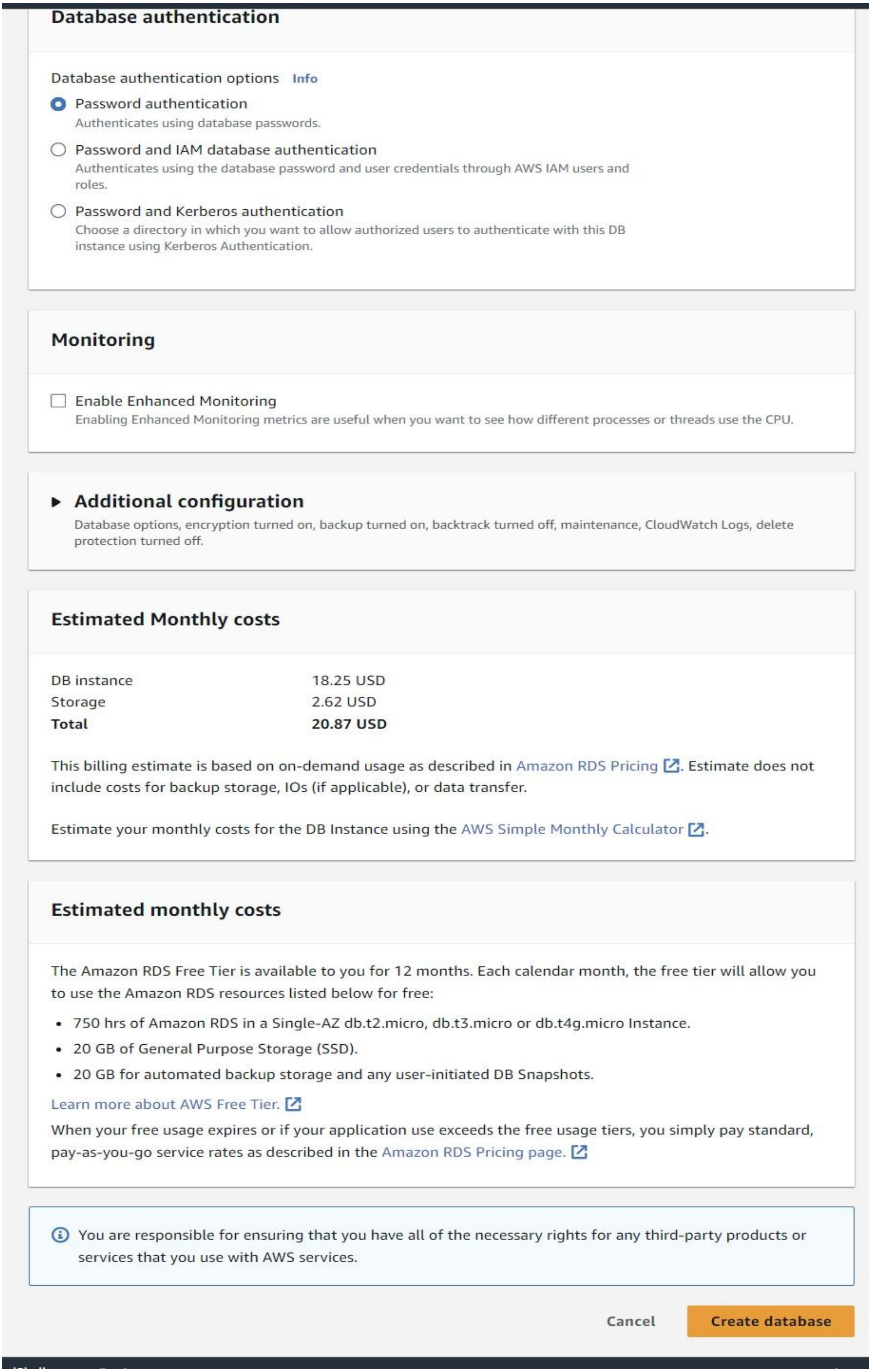
RDS Proxy

RDS Proxy is a fully managed, highly available database proxy that improves application scalability, resiliency, and security.

□ Create an RDS Proxy Info
RDS automatically creates an IAM role and a Secrets Manager secret for the proxy. RDS Proxy has additional costs. For more information, see Amazon RDS Proxy pricing.

Certificate authority - *optional* Info

Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.



## Database authentication

Database authentication options **Info**

● Password authentication
Authenticates using database passwords.

○ Password and IAM database authentication
Authenticates using the database password and user credentials through AWS IAM users and roles.

○ Password and Kerberos authentication
Choose a directory in which you want to allow authorized users to authenticate with this DB instance using Kerberos Authentication.

## Monitoring

□ Enable Enhanced Monitoring
Enabling Enhanced Monitoring metrics are useful when you want to see how different processes or threads use the CPU.

## ▶ Additional configuration

Database options, encryption turned on, backup turned on, backtrack turned off, maintenance, CloudWatch Logs, delete protection turned off.

## Estimated Monthly costs

| | |
|---|---|
| DB instance | 18.25 USD |
| Storage | 2.62 USD |
| **Total** | **20.87 USD** |

This billing estimate is based on on-demand usage as described in Amazon RDS Pricing. Estimate does not include costs for backup storage, IOs (if applicable), or data transfer.

Estimate your monthly costs for the DB Instance using the AWS Simple Monthly Calculator.

## Estimated monthly costs

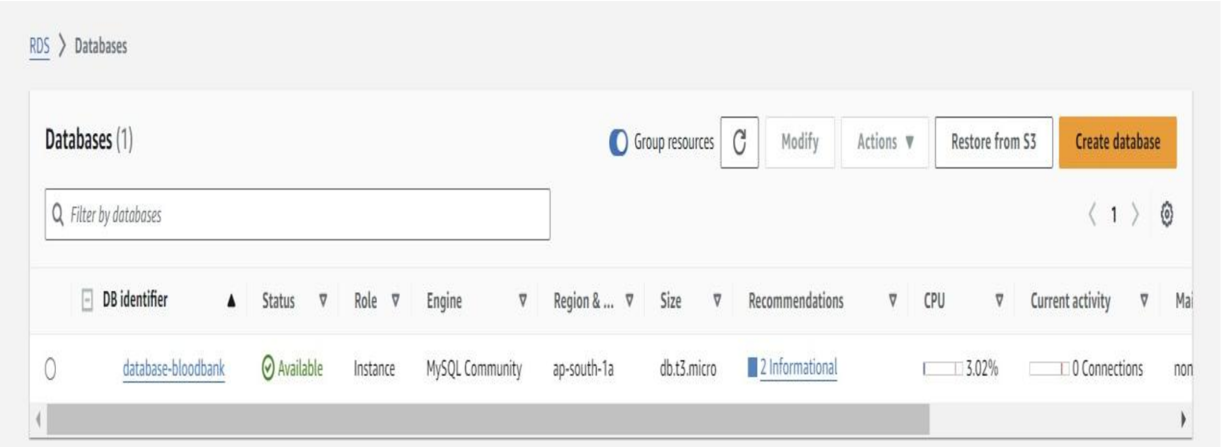
The Amazon RDS Free Tier is available to you for 12 months. Each calendar month, the free tier will allow you to use the Amazon RDS resources listed below for free:

- 750 hrs of Amazon RDS in a Single-AZ db.t2.micro, db.t3.micro or db.t4g.micro Instance.
- 20 GB of General Purpose Storage (SSD).
- 20 GB for automated backup storage and any user-initiated DB Snapshots.

Learn more about AWS Free Tier.

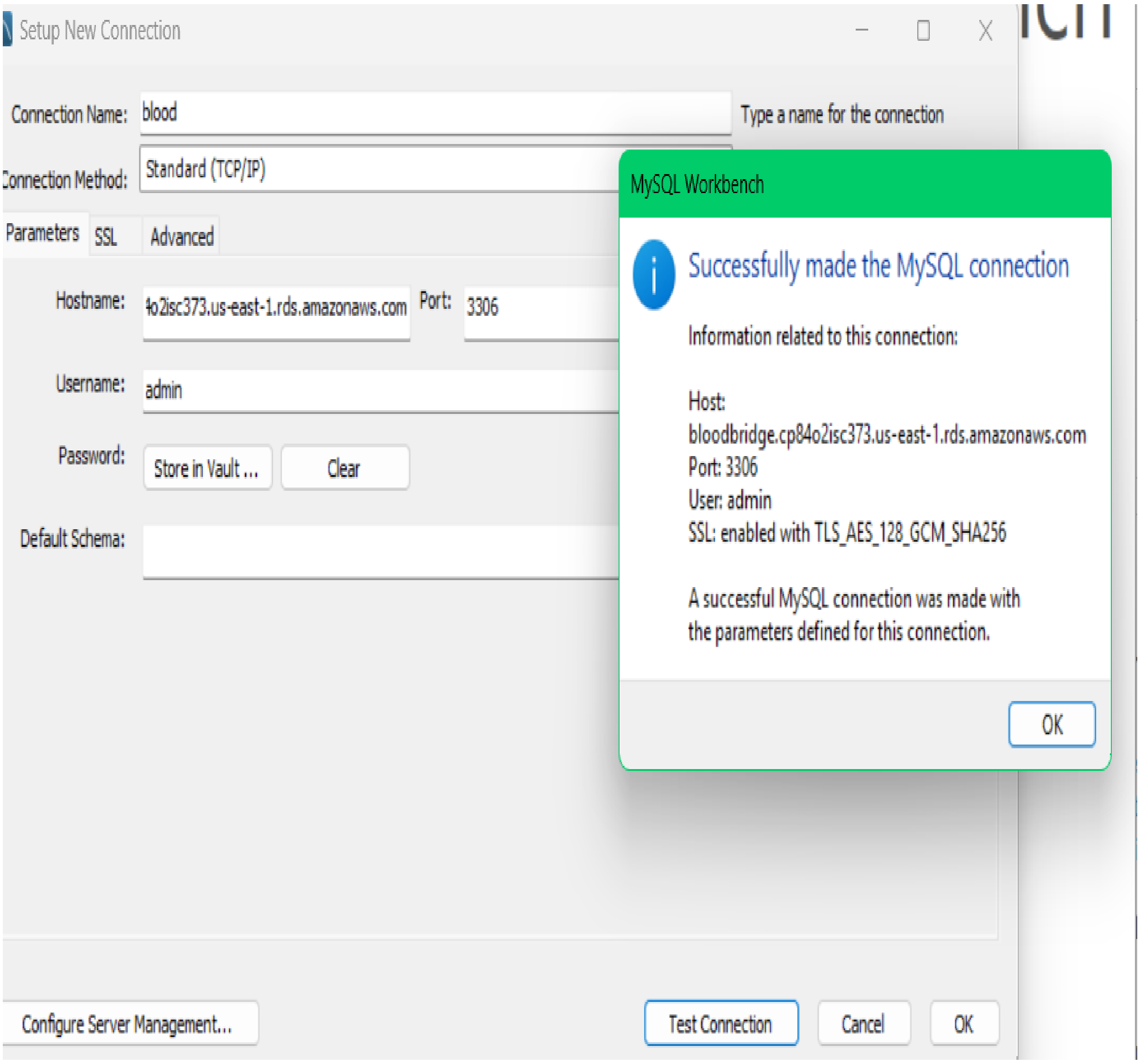
When your free usage expires or if your application use exceeds the free usage tiers, you simply pay standard, pay-as-you-go service rates as described in the Amazon RDS Pricing page.

ⓘ You are responsible for ensuring that you have all of the necessary rights for any third-party products or services that you use with AWS services.



Databases (1)

| | DB identifier | Status | Role | Engine | Region & ... | Size | Recommendations | CPU | Current activity | Mai |
|---|---|---|---|---|---|---|---|---|---|---|
| ○ | database-bloodbank | Available | Instance | MySQL Community | ap-south-1a | db.t3.micro | 2 Informational | 3.02% | 0 Connections | non |



Connection Name: blood — Type a name for the connection

Connection Method: Standard (TCP/IP)

Hostname: 4o2isc373.us-east-1.rds.amazonaws.com Port: 3306

Username: admin

Password:

Default Schema:

MySQL Workbench

Successfully made the MySQL connection

Information related to this connection:

Host: bloodbridge.cp84o2isc373.us-east-1.rds.amazonaws.com
Port: 3306
User: admin
SSL: enabled with TLS_AES_128_GCM_SHA256

A successful MySQL connection was made with the parameters defined for this connection.

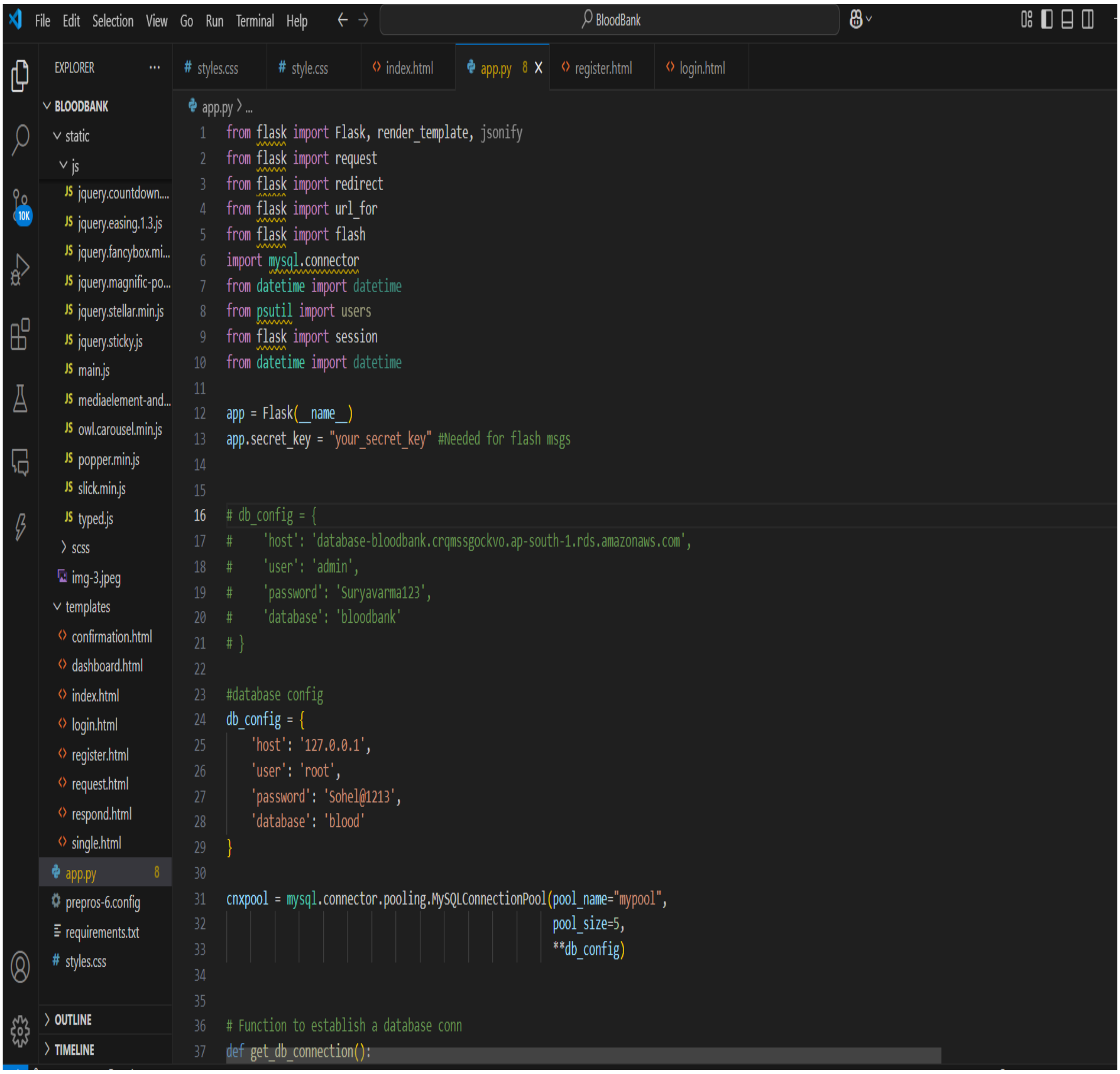
# Milestone 4: Develop Web Application

In this milestone, we will develop the BloodBridge web application with user registration, login, and dashboard features.

### Activity 1: Set Up Development Environment

Choose your preferred backend framework (e.g., Express.js for Node.js or Flask for Python).

Set up the project structure and install necessary dependencies.



```python
from flask import Flask, render_template, jsonify
from flask import request
from flask import redirect
from flask import url_for
from flask import flash
import mysql.connector
from datetime import datetime
from psutil import users
from flask import session
from datetime import datetime

app = Flask(__name__)
app.secret_key = "your_secret_key" #Needed for flash msgs


# db_config = {
#     'host': 'database-bloodbank.crqmssgockvo.ap-south-1.rds.amazonaws.com',
#     'user': 'admin',
#     'password': 'Suryavarma123',
#     'database': 'bloodbank'
# }

#database config
db_config = {
    'host': '127.0.0.1',
    'user': 'root',
    'password': 'Sohel@1213',
    'database': 'blood'
}

cnxpool = mysql.connector.pooling.MySQLConnectionPool(pool_name="mypool",
                                                      pool_size=5,
                                                      **db_config)



# Function to establish a database conn
def get_db_connection():
```

1. **Setting Up Flask Application:**

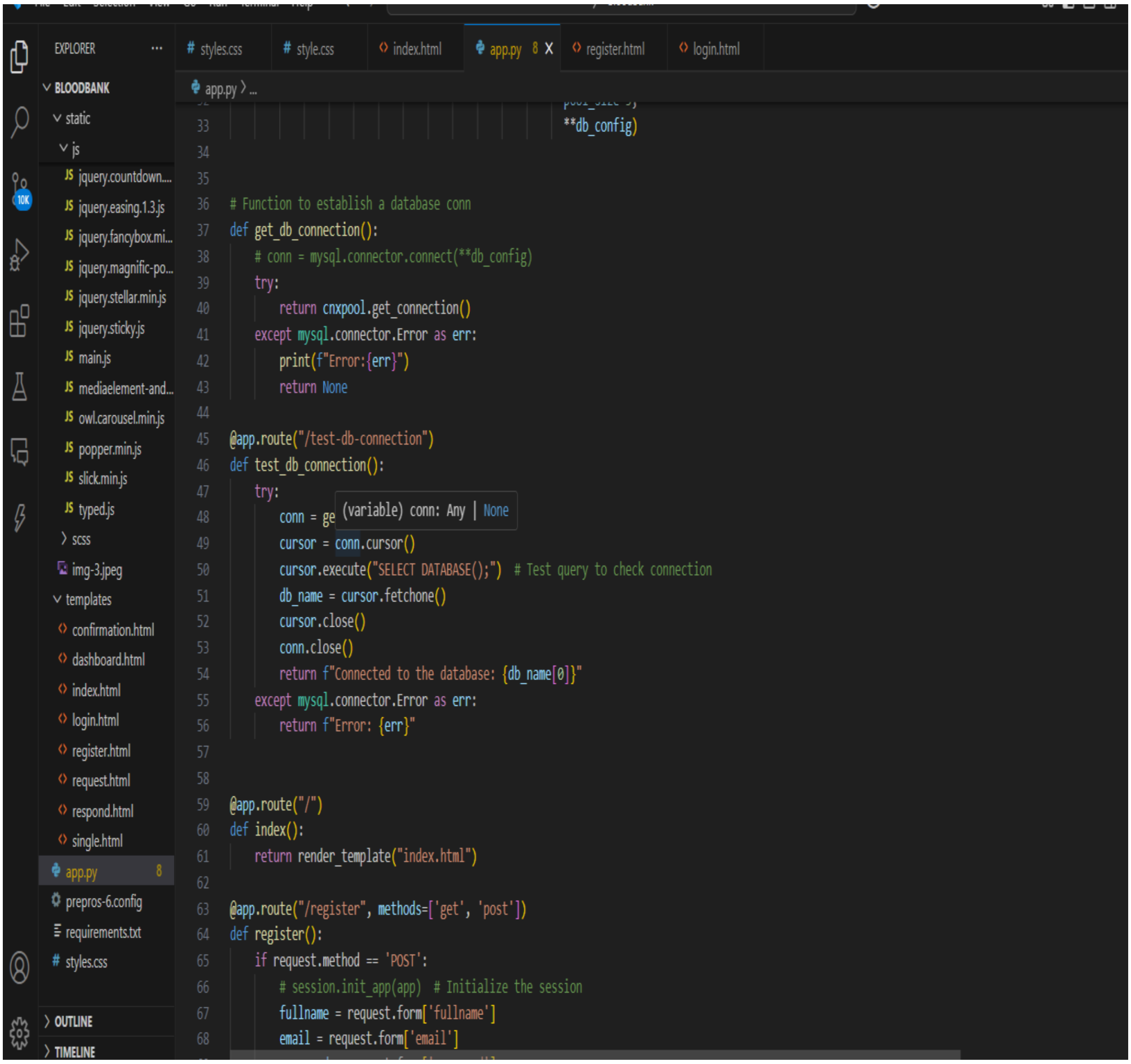
- The application starts by importing necessary modules such as Flask, MySQL connector, session handling, and other utilities like datetime.
- app = Flask(__name__): Initializes a Flask web application.

- app.secret_key = "your_secret_key": This key is necessary for securely handling sessions and flash messages (temporary notifications).

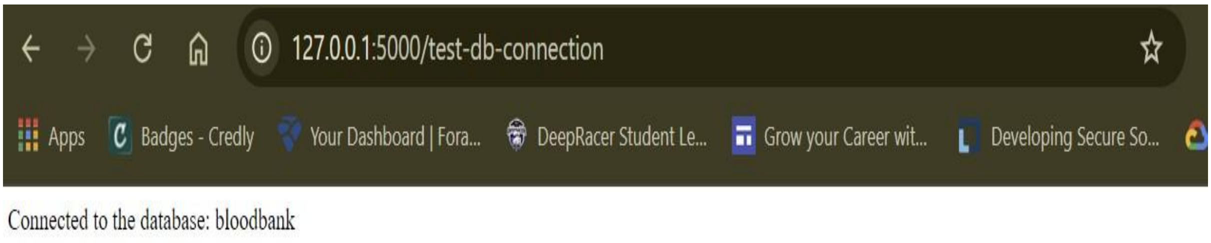
2. **Database Connection:**

- Two sets of database configuration (db_config) are defined: one for a local MySQL database (127.0.0.1) and another commented out version for a remote AWS RDS instance.
- A connection pool is created using mysql.connector.pooling.MySQLConnectionPool, which allows up to 5 simultaneous connections.

The get_db_connection() function retrieves a connection from the pool.



```python
                                      **db_config)


# Function to establish a database conn
def get_db_connection():
    # conn = mysql.connector.connect(**db_config)
    try:
        return cnxpool.get_connection()
    except mysql.connector.Error as err:
        print(f"Error:{err}")
        return None

@app.route("/test-db-connection")
def test_db_connection():
    try:
        conn = ge (variable) conn: Any | None
        cursor = conn.cursor()
        cursor.execute("SELECT DATABASE();")  # Test query to check connection
        db_name = cursor.fetchone()
        cursor.close()
        conn.close()
        return f"Connected to the database: {db_name[0]}"
    except mysql.connector.Error as err:
        return f"Error: {err}"


@app.route("/")
def index():
    return render_template("index.html")

@app.route("/register", methods=['get', 'post'])
def register():
    if request.method == 'POST':
        # session.init_app(app)  # Initialize the session
        fullname = request.form['fullname']
        email = request.form['email']
```



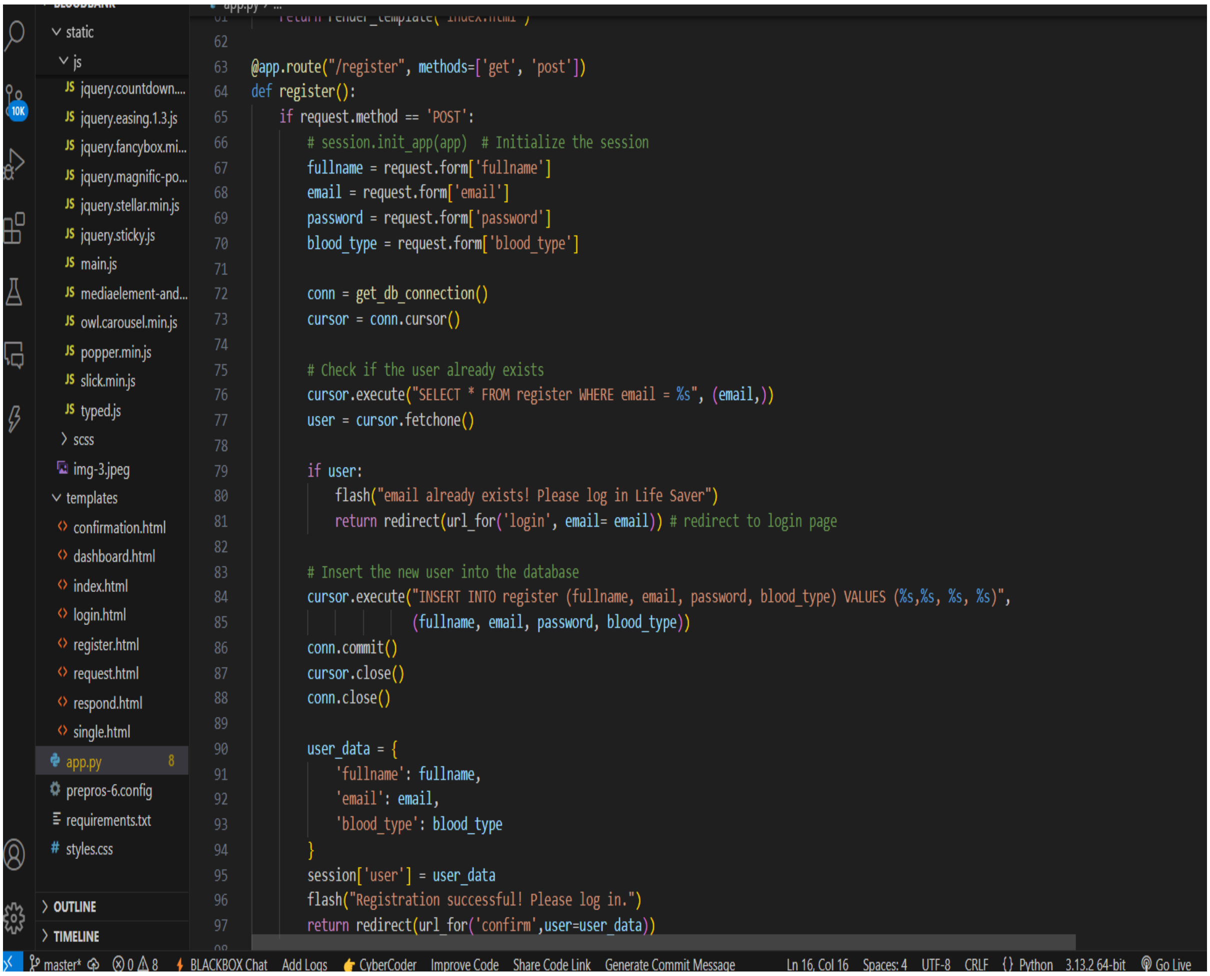
Connected to the database: bloodbank

**Testing the Database Connection:**

- A test route /test-db-connection is used to verify if the application can connect to the database by executing a simple SQL query: SELECT DATABASE();.

4. **Home Route (/):**

- The index() function returns the main HTML page (index.html).



```python
    return render_template('index.html')

@app.route("/register", methods=['get', 'post'])
def register():
    if request.method == 'POST':
        # session.init_app(app)  # Initialize the session
        fullname = request.form['fullname']
        email = request.form['email']
        password = request.form['password']
        blood_type = request.form['blood_type']

        conn = get_db_connection()
        cursor = conn.cursor()

        # Check if the user already exists
        cursor.execute("SELECT * FROM register WHERE email = %s", (email,))
        user = cursor.fetchone()

        if user:
            flash("email already exists! Please log in Life Saver")
            return redirect(url_for('login', email= email)) # redirect to login page

        # Insert the new user into the database
        cursor.execute("INSERT INTO register (fullname, email, password, blood_type) VALUES (%s,%s, %s, %s)",
                        (fullname, email, password, blood_type))
        conn.commit()
        cursor.close()
        conn.close()

        user_data = {
            'fullname': fullname,
            'email': email,
            'blood_type': blood_type
        }
        session['user'] = user_data
        flash("Registration successful! Please log in.")
        return redirect(url_for('confirm',user=user_data))
```

5. User Registration (/register):

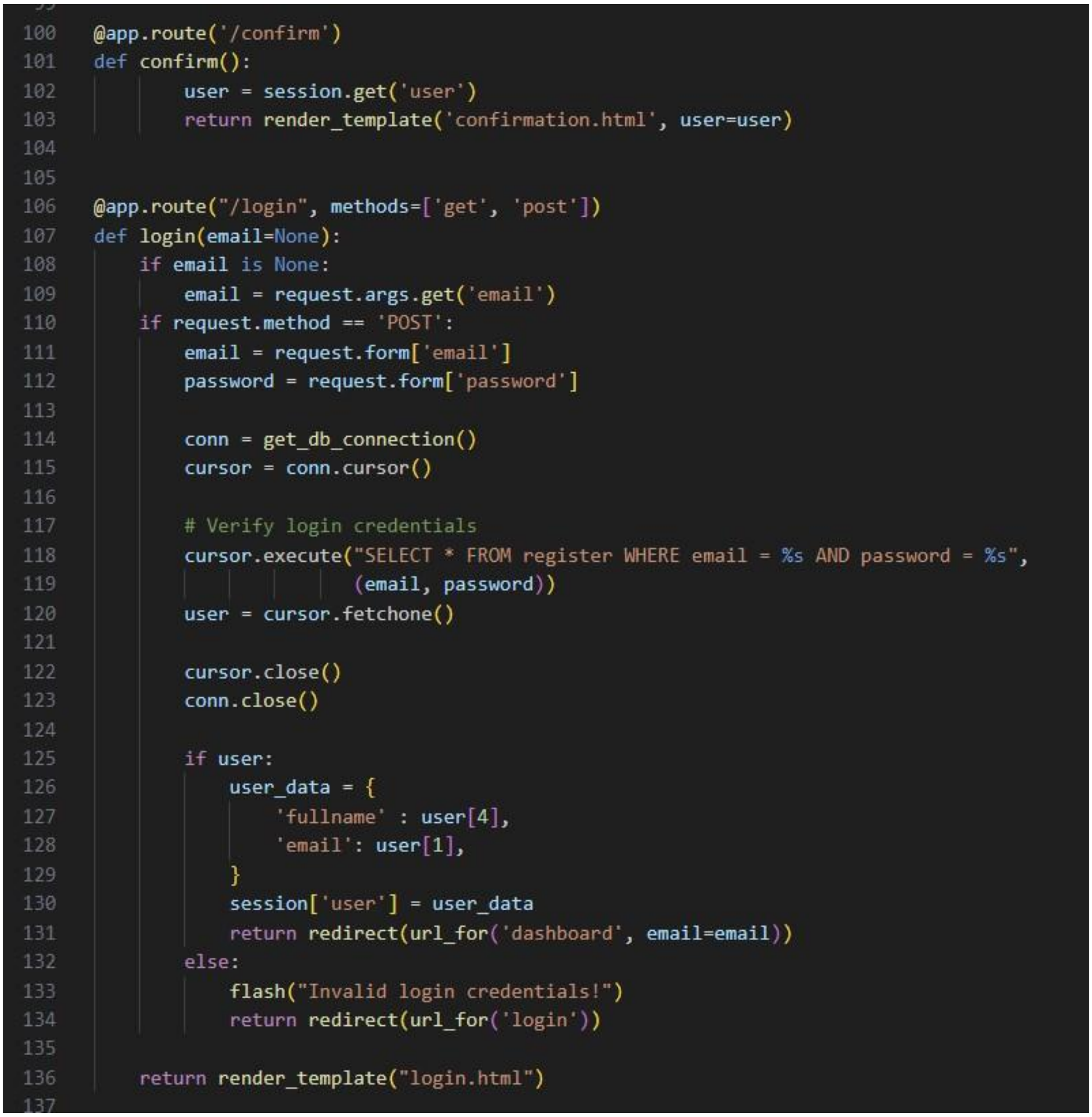
- The user submits a registration form with details like fullname, email, password, and blood_type.

It checks if the email already exists in the database using: python Copy code cursor.execute("SELECT * FROM register WHERE email = %s", (email,))

- If the email exists, the user is redirected to the login page, and a flash message is shown.

If the user is new, their details are inserted into the database: python Copy code cursor.execute("INSERT INTO register (fullname, email, password, blood_type) VALUES (%s,%s, %s, %s)", (fullname, email, password, blood_type))

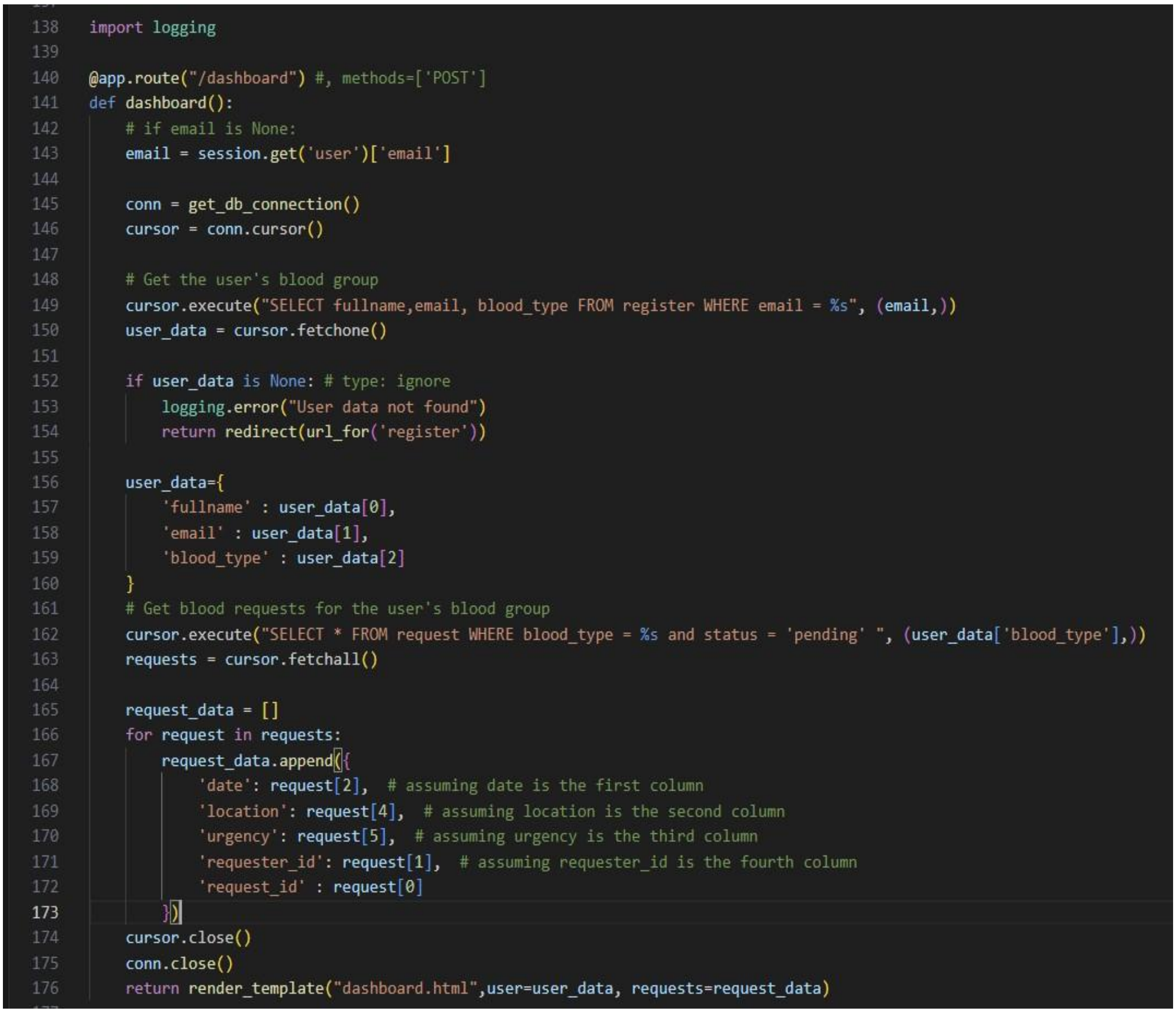
- Session data is created for the user, and the user is redirected to a confirmation page.



```python
@app.route('/confirm')
def confirm():
        user = session.get('user')
        return render_template('confirmation.html', user=user)


@app.route("/login", methods=['get', 'post'])
def login(email=None):
    if email is None:
        email = request.args.get('email')
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        conn = get_db_connection()
        cursor = conn.cursor()

        # Verify login credentials
        cursor.execute("SELECT * FROM register WHERE email = %s AND password = %s",
                       (email, password))
        user = cursor.fetchone()

        cursor.close()
        conn.close()

        if user:
            user_data = {
                'fullname' : user[4],
                'email': user[1],
            }
            session['user'] = user_data
            return redirect(url_for('dashboard', email=email))
        else:
            flash("Invalid login credentials!")
            return redirect(url_for('login'))

    return render_template("login.html")
```

6. **User Login (/login):**

- A login form accepts the email and password.
- These credentials are verified by querying the database. If they match, the user is redirected to the dashboard, and session data is stored: python Copy code cursor.execute("SELECT * FROM register WHERE email = %s AND password = %s", (email, password))



```python
import logging

@app.route("/dashboard") #, methods=['POST']
def dashboard():
    # if email is None:
    email = session.get('user')['email']

    conn = get_db_connection()
    cursor = conn.cursor()

    # Get the user's blood group
    cursor.execute("SELECT fullname,email, blood_type FROM register WHERE email = %s", (email,))
    user_data = cursor.fetchone()

    if user_data is None: # type: ignore
        logging.error("User data not found")
        return redirect(url_for('register'))

    user_data={
        'fullname' : user_data[0],
        'email' : user_data[1],
        'blood_type' : user_data[2]
    }
    # Get blood requests for the user's blood group
    cursor.execute("SELECT * FROM request WHERE blood_type = %s and status = 'pending' ", (user_data['blood_type'],))
    requests = cursor.fetchall()

    request_data = []
    for request in requests:
        request_data.append({
            'date': request[2],  # assuming date is the first column
            'location': request[4],  # assuming location is the second column
            'urgency': request[5],  # assuming urgency is the third column
            'requester_id': request[1],  # assuming requester_id is the fourth column
            'request_id' : request[0]
        })
    cursor.close()
    conn.close()
    return render_template("dashboard.html",user=user_data, requests=request_data)
```

**1. Session retrieval**

- It fetches the current user's email from the session (`email = session.get('user')['email']`).

**2. Database connection:**

- A connection to the database is established using `get_db_connection()`.

**3. Fetching user data:**

- The query fetches the full name, email, and blood type of the logged-in user from the `register` table based on their email.

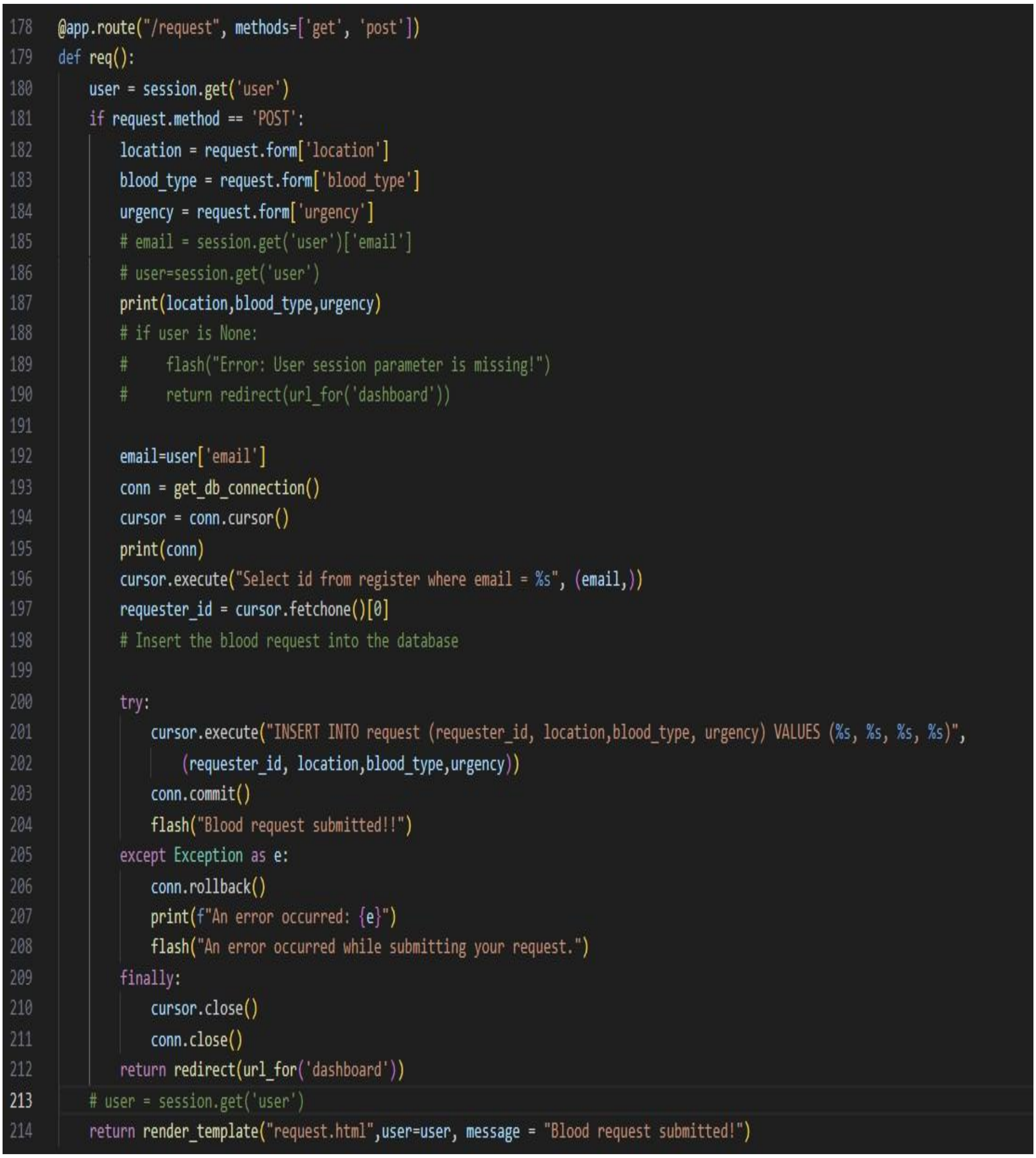
**4. Blood requests retrieval:**

- It retrieves all pending blood requests matching the user’s blood type from the `request` table. This helps users see relevant requests.

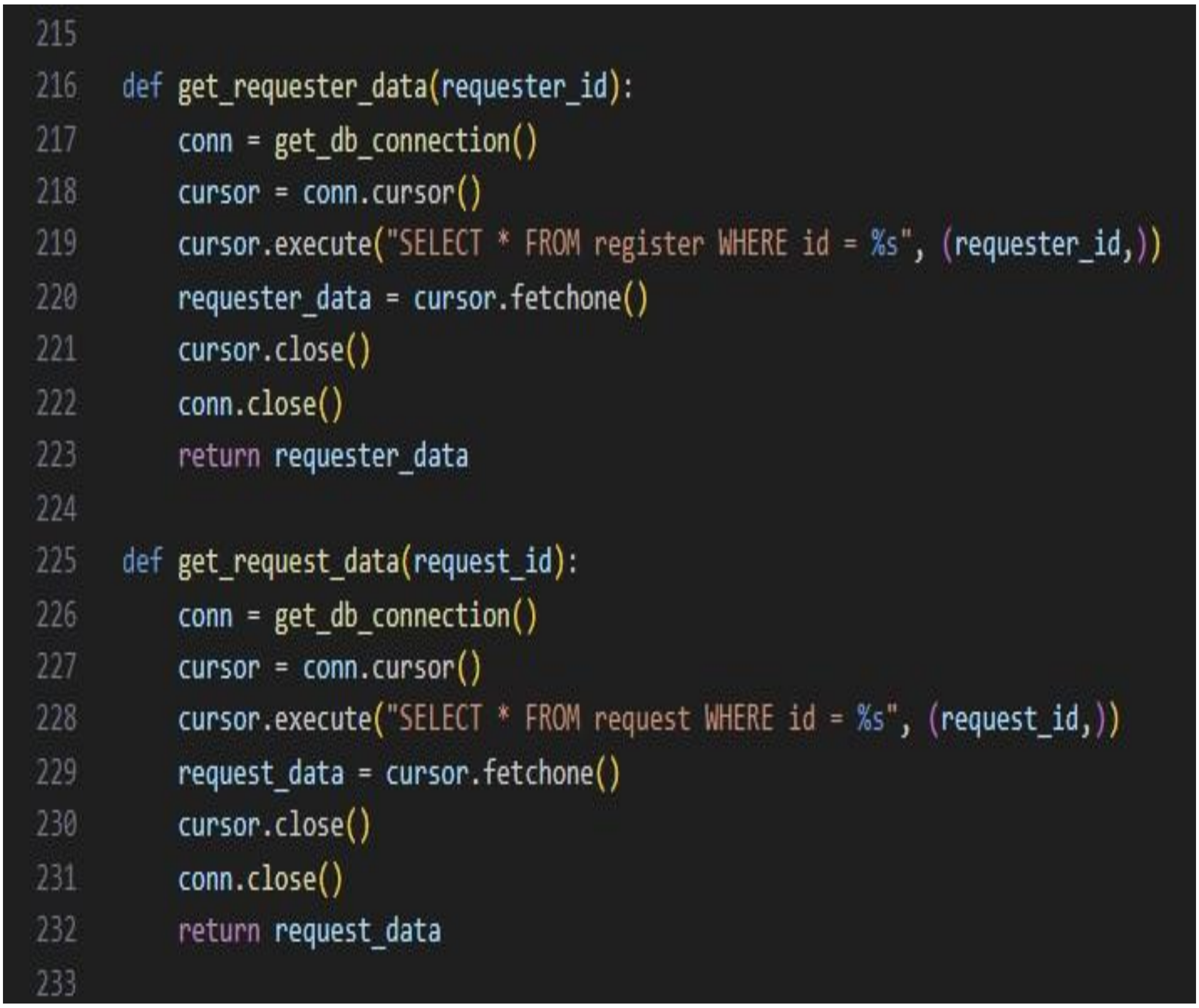
**5. Data display:**

- Finally, the `dashboard.html` template is rendered, showing the user data and matching blood requests.

The dashboard serves as a central page for users to view requests related to their blood type.



```python
@app.route("/request", methods=['get', 'post'])
def req():
    user = session.get('user')
    if request.method == 'POST':
        location = request.form['location']
        blood_type = request.form['blood_type']
        urgency = request.form['urgency']
        # email = session.get('user')['email']
        # user=session.get('user')
        print(location,blood_type,urgency)
        # if user is None:
        #     flash("Error: User session parameter is missing!")
        #     return redirect(url_for('dashboard'))

        email=user['email']
        conn = get_db_connection()
        cursor = conn.cursor()
        print(conn)
        cursor.execute("Select id from register where email = %s", (email,))
        requester_id = cursor.fetchone()[0]
        # Insert the blood request into the database

        try:
            cursor.execute("INSERT INTO request (requester_id, location,blood_type, urgency) VALUES (%s, %s, %s, %s)",
                (requester_id, location,blood_type,urgency))
            conn.commit()
            flash("Blood request submitted!!")
        except Exception as e:
            conn.rollback()
            print(f"An error occurred: {e}")
            flash("An error occurred while submitting your request.")
        finally:
            cursor.close()
            conn.close()
        return redirect(url_for('dashboard'))
    # user = session.get('user')
    return render_template("request.html",user=user, message = "Blood request submitted!")
```



```

def get_requester_data(requester_id):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM register WHERE id = %s", (requester_id,))
    requester_data = cursor.fetchone()
    cursor.close()
    conn.close()
    return requester_data

def get_request_data(request_id):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM request WHERE id = %s", (request_id,))
    request_data = cursor.fetchone()
    cursor.close()
    conn.close()
    return request_data

```

**Session retrieval:**

- It first checks the session for the logged-in user's details (using session.get('user')).

**Form submission (POST request):**

- If the request method is POST, the form data (location, blood type, urgency) is retrieved from the user input.

**Database connection and insertion:**

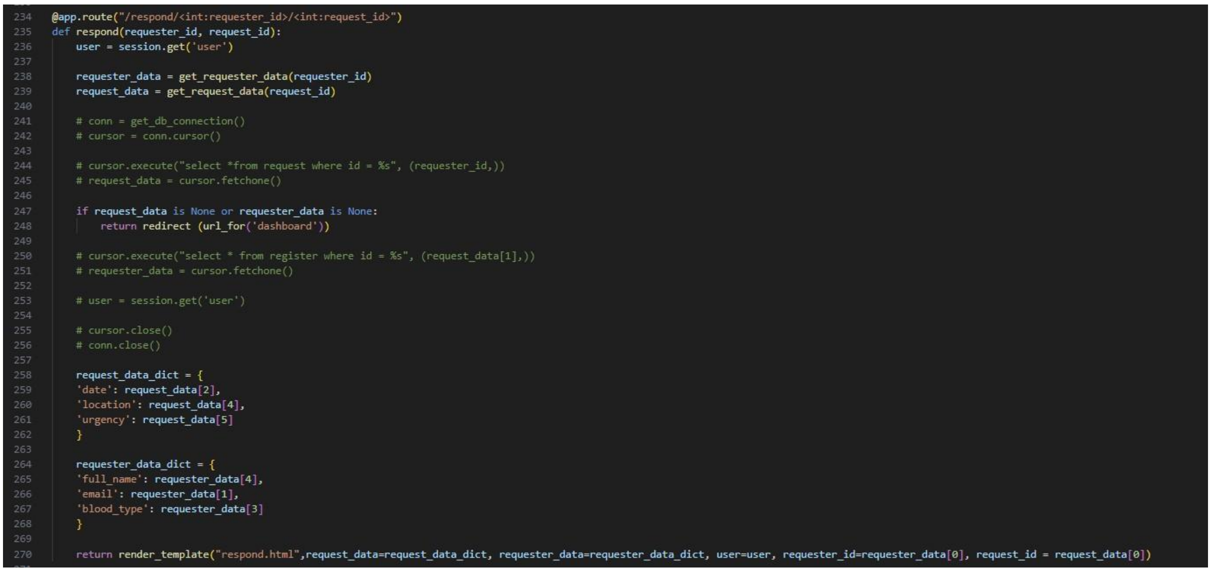
- A database connection is created using get_db_connection(). The code then fetches the requester_id (from the register table) based on the user's email. ● The user's blood request is inserted into the request table.

**Error handling and response:**

- The request is committed to the database, and if any error occurs, it is handled by rolling back the transaction.
- If the request is successful, a success message is flashed, and the user is redirected to the dashboard page.

**Template rendering (GET request):**

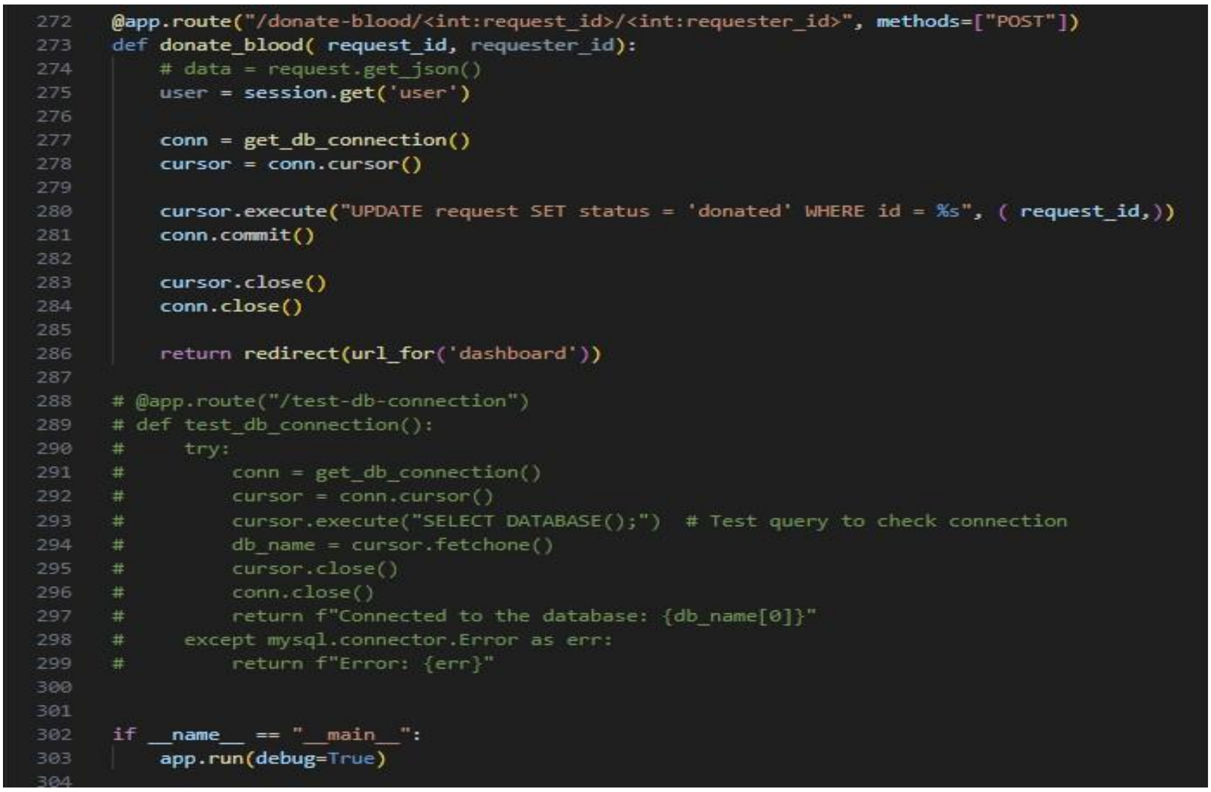
- For a GET request (when the page is first accessed), it renders the request.html template, displaying the form for submitting blood requests.



```python
@app.route("/respond/<int:requester_id>/<int:request_id>")
def respond(requester_id, request_id):
    user = session.get('user')

    requester_data = get_requester_data(requester_id)
    request_data = get_request_data(request_id)

    # conn = get_db_connection()
    # cursor = conn.cursor()

    # cursor.execute("select *from request where id = %s", (requester_id,))
    # request_data = cursor.fetchone()

    if request_data is None or requester_data is None:
        return redirect (url_for('dashboard'))

    # cursor.execute("select * from register where id = %s", (request_data[1],))
    # requester_data = cursor.fetchone()

    # user = session.get('user')

    # cursor.close()
    # conn.close()

    request_data_dict = {
    'date': request_data[2],
    'location': request_data[4],
    'urgency': request_data[5]
    }

    requester_data_dict = {
    'full_name': requester_data[4],
    'email': requester_data[1],
    'blood_type': requester_data[3]
    }

    return render_template("respond.html",request_data=request_data_dict, requester_data=requester_data_dict, user=user, requester_id=requester_data[0], request_id = request_data[0])
```

.Responding to a Request (/respond/<int:requester_id>/<int:request_id>):

- When another user wants to respond to a blood request, the respond() function fetches both the requester and request details using helper functions (get_requester_data() and get_request_data()).
- The data is passed to respond.html where the user can confirm their donation.



```python
@app.route("/donate-blood/<int:request_id>/<int:requester_id>", methods=["POST"])
def donate_blood( request_id, requester_id):
    # data = request.get_json()
    user = session.get('user')

    conn = get_db_connection()
    cursor = conn.cursor()

    cursor.execute("UPDATE request SET status = 'donated' WHERE id = %s", ( request_id,))
    conn.commit()

    cursor.close()
    conn.close()

    return redirect(url_for('dashboard'))

# @app.route("/test-db-connection")
# def test_db_connection():
#     try:
#         conn = get_db_connection()
#         cursor = conn.cursor()
#         cursor.execute("SELECT DATABASE();")  # Test query to check connection
#         db_name = cursor.fetchone()
#         cursor.close()
#         conn.close()
#         return f"Connected to the database: {db_name[0]}"
#     except mysql.connector.Error as err:
#         return f"Error: {err}"


if __name__ == "__main__":
    app.run(debug=True)
```

Donation Confirmation (/donate-blood/<int:request_id>/<int:requester_id>):

This route handles the blood donation confirmation by updating the request status to 'donated' in the database:
python
Copy code
cursor.execute("UPDATE request SET status = 'donated' WHERE id = %s", (request_id,))

11. **Session and Flash Messaging:**

- Flask's session is used to store user information temporarily across multiple routes.
- flash() is used to send feedback messages to the user, which are displayed when the page is rendered.

12. **Running the App:**

- The application runs with app.run(debug=True), meaning it will restart automatically if changes are made to the code during development.

## Activity 2: Implement User Authentication

1. Create registration and login routes and forms.
2. Implement user authentication logic, including password hashing and session management.

## Activity 3: Develop User Dashboard

1. Create a dashboard interface displaying user information and blood request options.
2. Implement blood request submission and tracking functionality.

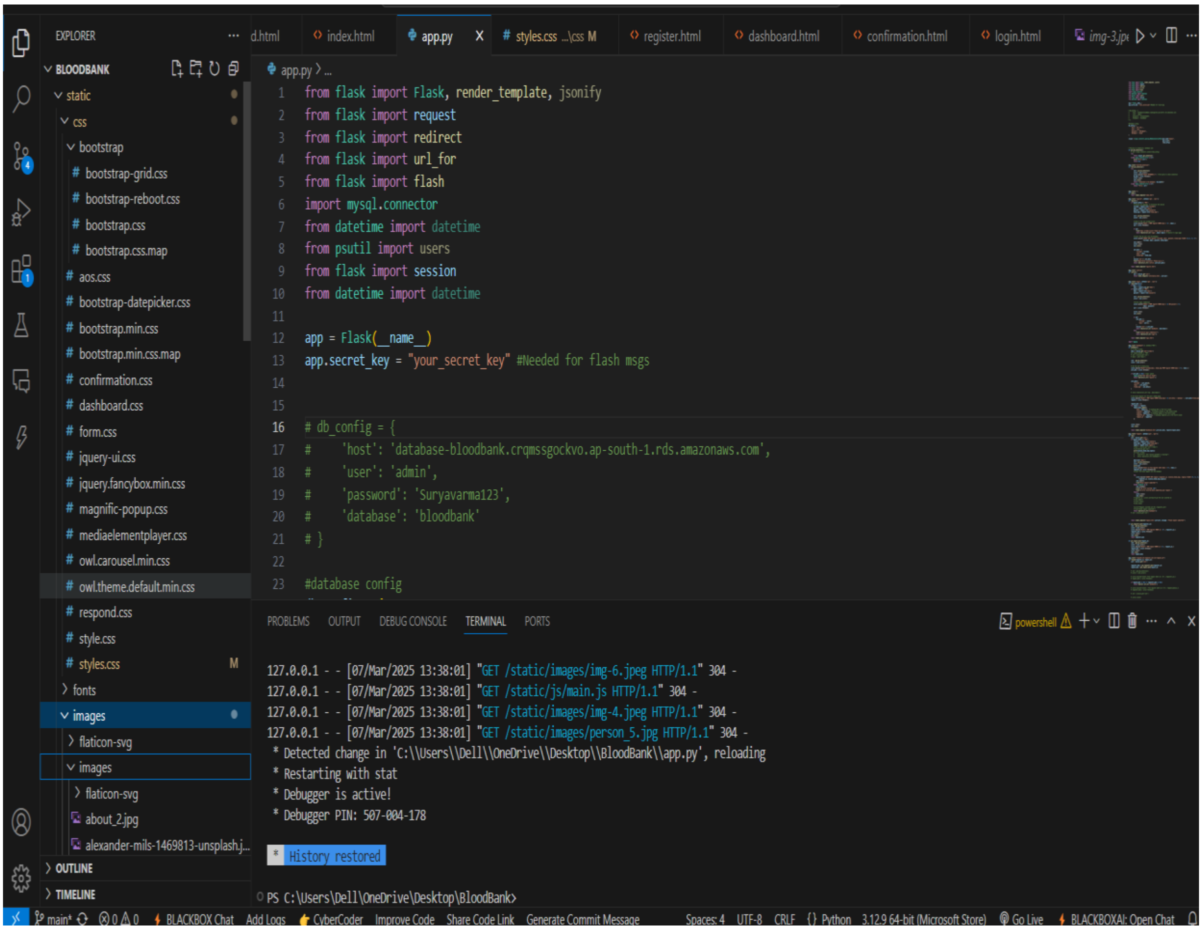
## Activity 4: Integrate with RDS Database

1. Set up database connection using appropriate drivers (e.g., mysql2 for Node.js). Implement database queries for user management and blood request handling.

# Milestone 4: Testing and Deployment

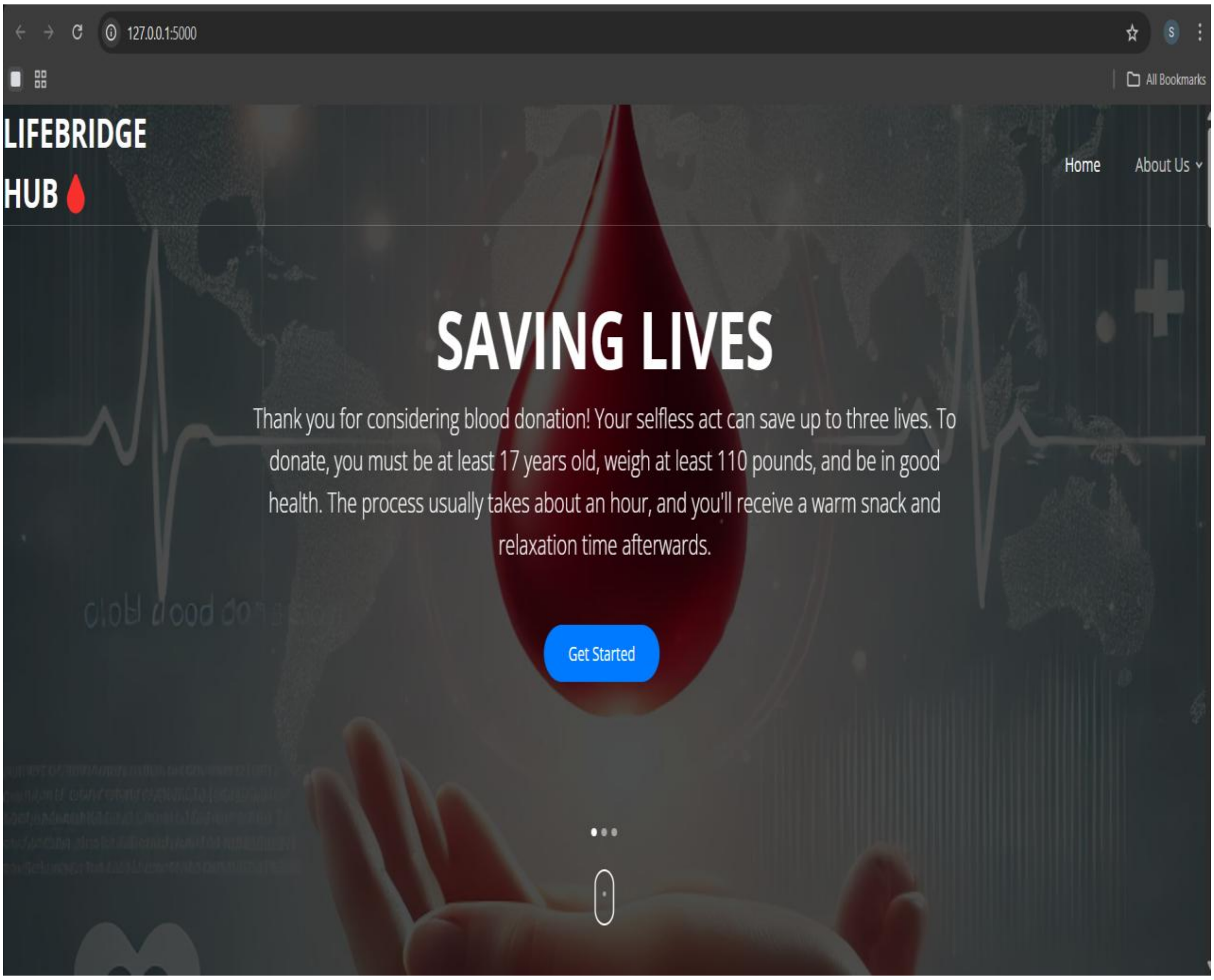
### Activity 1: Deploy to EC2

1. Transfer your application code to the EC2 instance.
2. Set up any necessary environment variables, including database connection strings.
3. Configure the web server to serve your application.
4. Start your application and ensure it's accessible via the EC2 instance's public IP or domain.
5. Run the below commands on ec2 terminal
6. sudo yum update -y
7. sudo yum install python3 -y
8. sudo pip3 install virtualenv
9. python3 -m venv venv
10. source venv/bin/activate
11. pip install flask
12. git clone https://github.com/your-repo/your-flask-app.git
13. cd your-flask-app
14. python3 app.py



```python
from flask import Flask, render_template, jsonify
from flask import request
from flask import redirect
from flask import url_for
from flask import flash
import mysql.connector
from datetime import datetime
from psutil import users
from flask import session
from datetime import datetime

app = Flask(__name__)
app.secret_key = "your_secret_key" #Needed for flash msgs


# db_config = {
#     'host': 'database-bloodbank.crqmssgockvo.ap-south-1.rds.amazonaws.com',
#     'user': 'admin',
#     'password': 'Suryavarma123',
#     'database': 'bloodbank'
# }

#database config
```

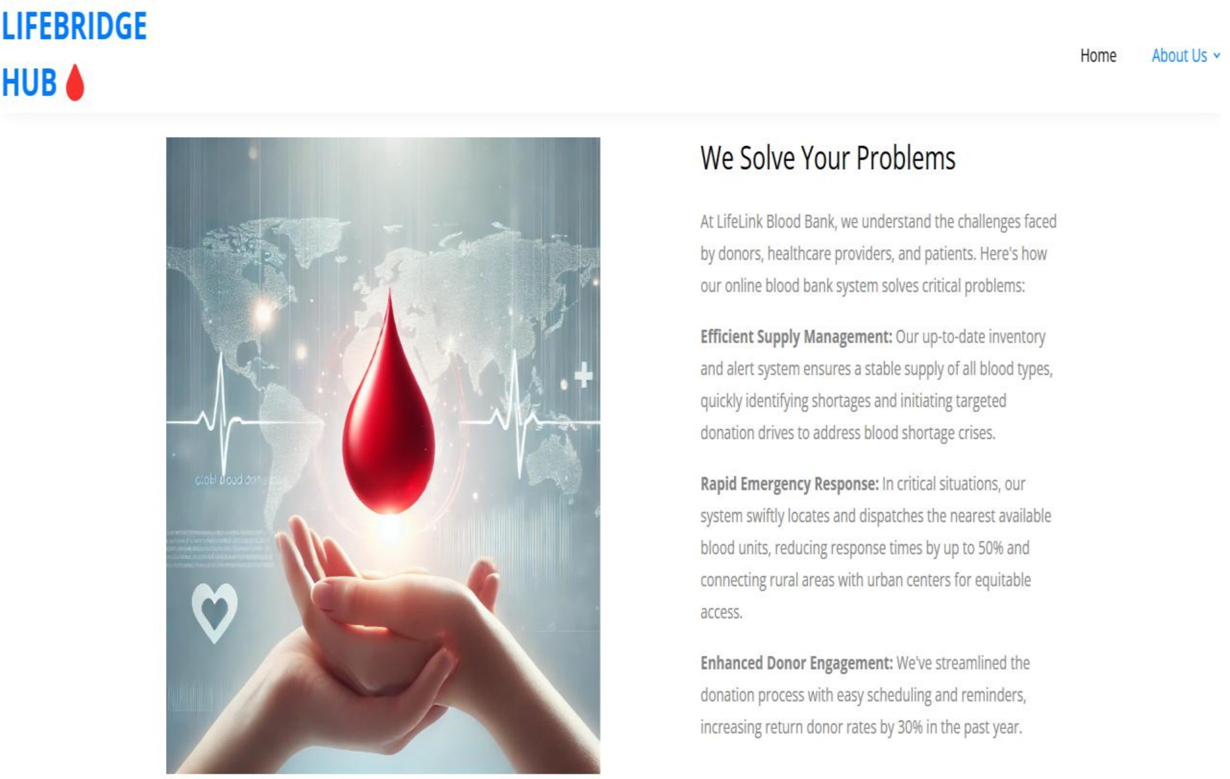
```
127.0.0.1 - - [07/Mar/2025 13:38:01] "GET /static/images/img-6.jpeg HTTP/1.1" 304 -
127.0.0.1 - - [07/Mar/2025 13:38:01] "GET /static/js/main.js HTTP/1.1" 304 -
127.0.0.1 - - [07/Mar/2025 13:38:01] "GET /static/images/img-4.jpeg HTTP/1.1" 304 -
127.0.0.1 - - [07/Mar/2025 13:38:01] "GET /static/images/person_5.jpg HTTP/1.1" 304 -
 * Detected change in 'C:\\Users\\Dell\\OneDrive\\Desktop\\BloodBank\\app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 507-004-178

 * History restored

PS C:\Users\Dell\OneDrive\Desktop\BloodBank>
```



LIFEBRIDGE HUB

# SAVING LIVES

Thank you for considering blood donation! Your selfless act can save up to three lives. To donate, you must be at least 17 years old, weigh at least 110 pounds, and be in good health. The process usually takes about an hour, and you'll receive a warm snack and relaxation time afterwards.



LIFEBRIDGE HUB

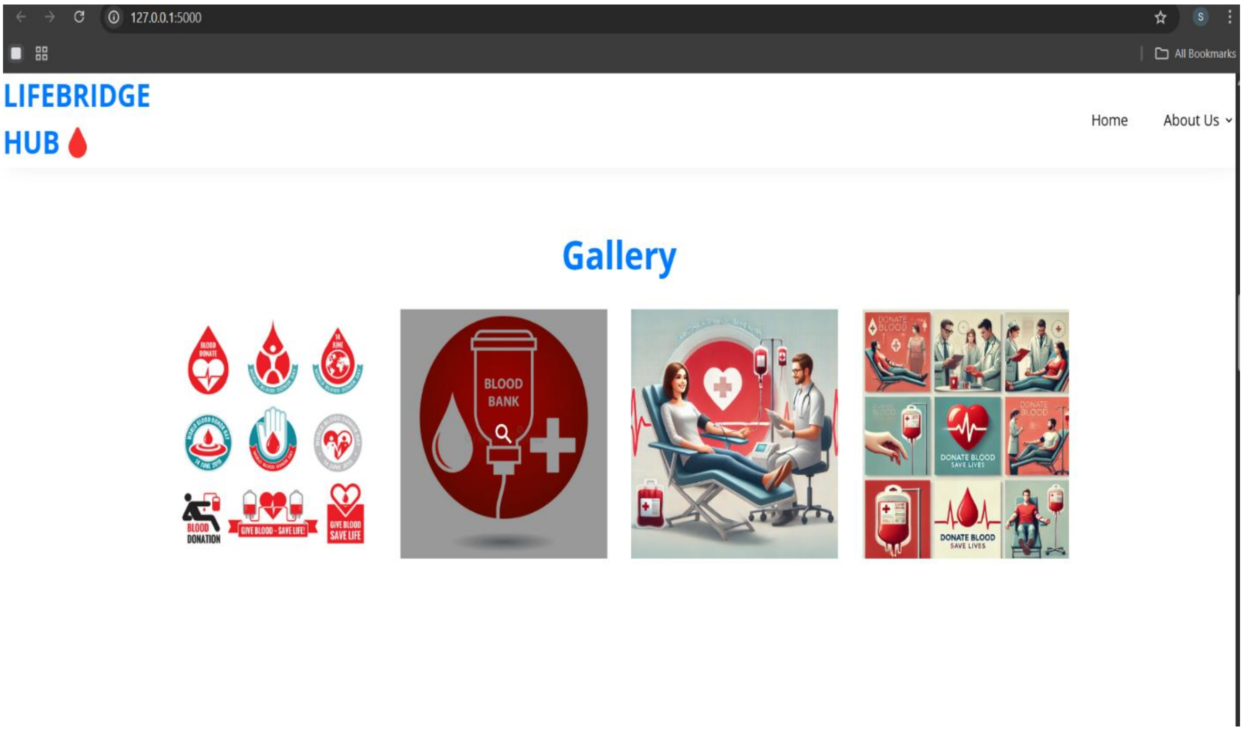
## We Solve Your Problems

At LifeLink Blood Bank, we understand the challenges faced by donors, healthcare providers, and patients. Here's how our online blood bank system solves critical problems:

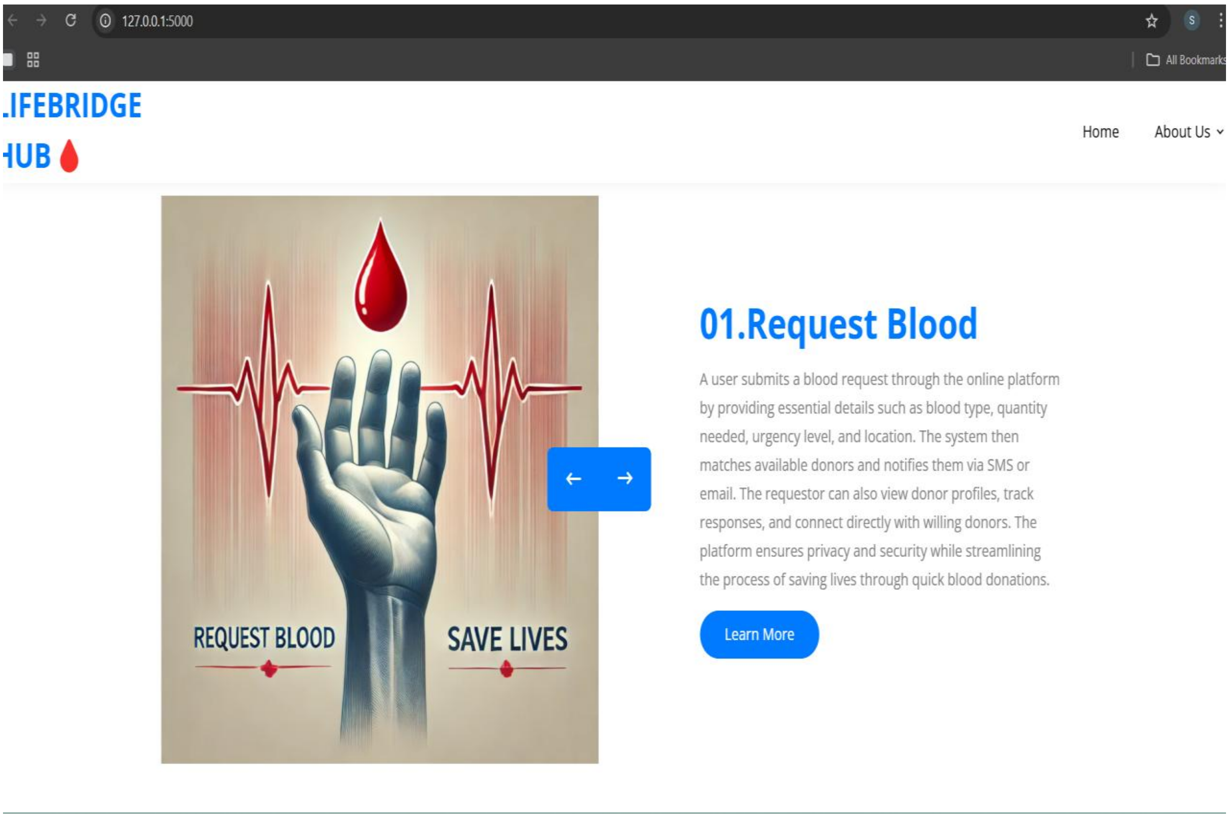
**Efficient Supply Management:** Our up-to-date inventory and alert system ensures a stable supply of all blood types, quickly identifying shortages and initiating targeted donation drives to address blood shortage crises.

**Rapid Emergency Response:** In critical situations, our system swiftly locates and dispatches the nearest available blood units, reducing response times by up to 50% and connecting rural areas with urban centers for equitable access.

**Enhanced Donor Engagement:** We've streamlined the donation process with easy scheduling and reminders, increasing return donor rates by 30% in the past year.

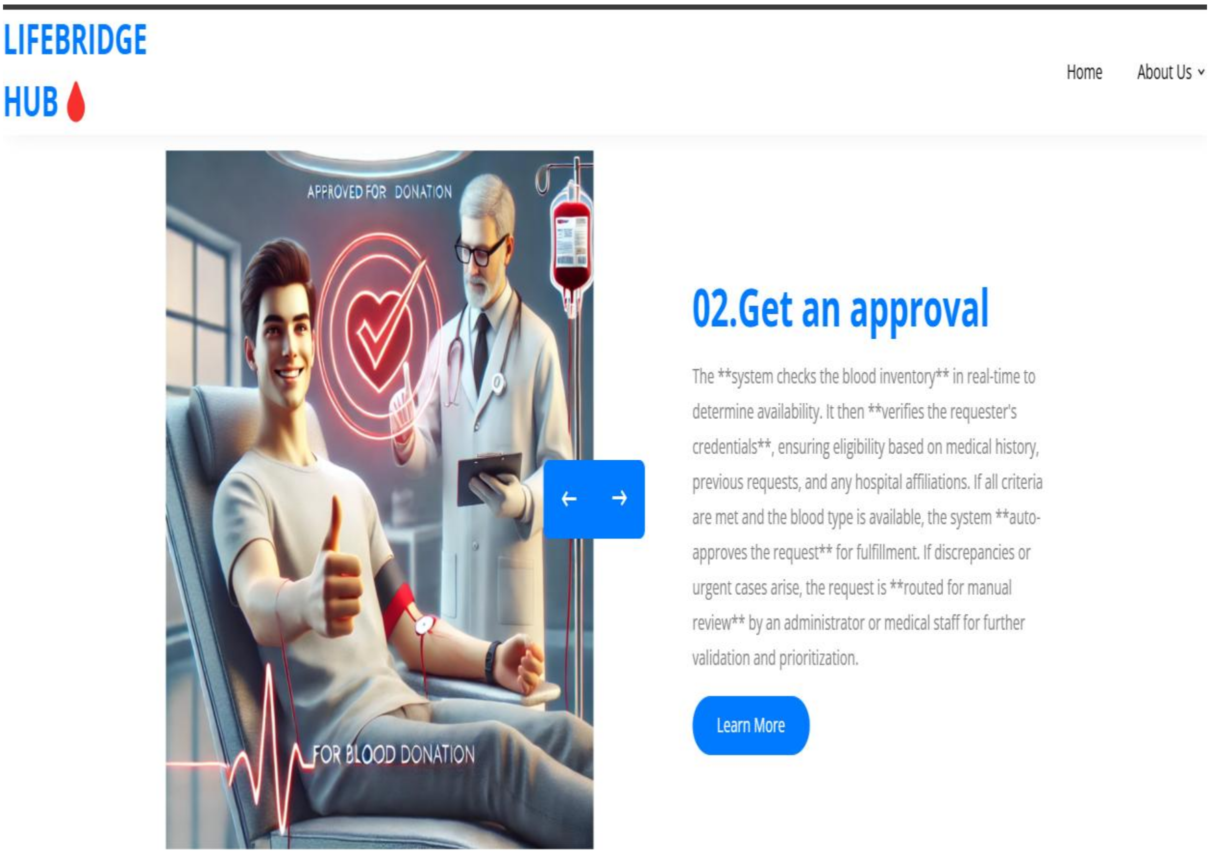


Gallery



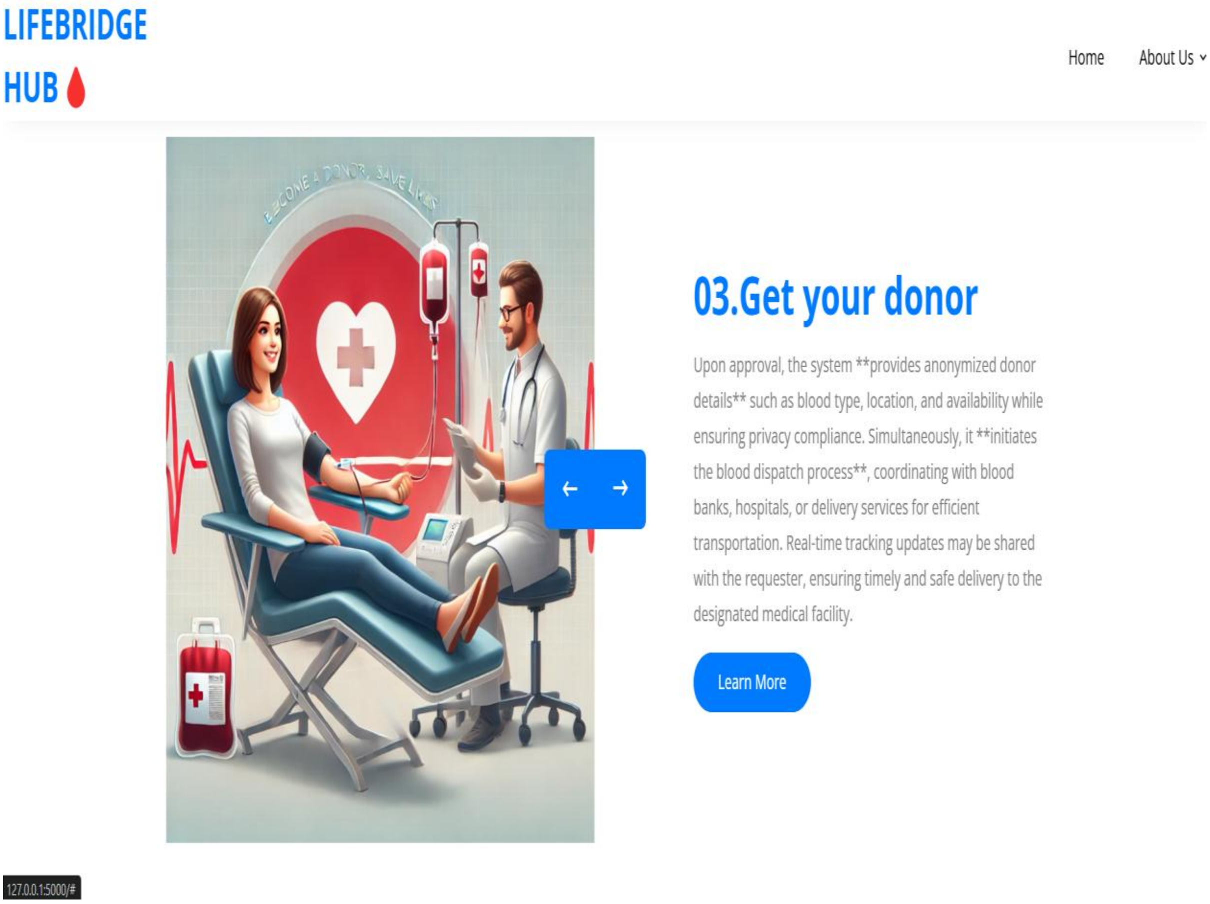
01.Request Blood

A user submits a blood request through the online platform by providing essential details such as blood type, quantity needed, urgency level, and location. The system then matches available donors and notifies them via SMS or email. The requestor can also view donor profiles, track responses, and connect directly with willing donors. The platform ensures privacy and security while streamlining the process of saving lives through quick blood donations.



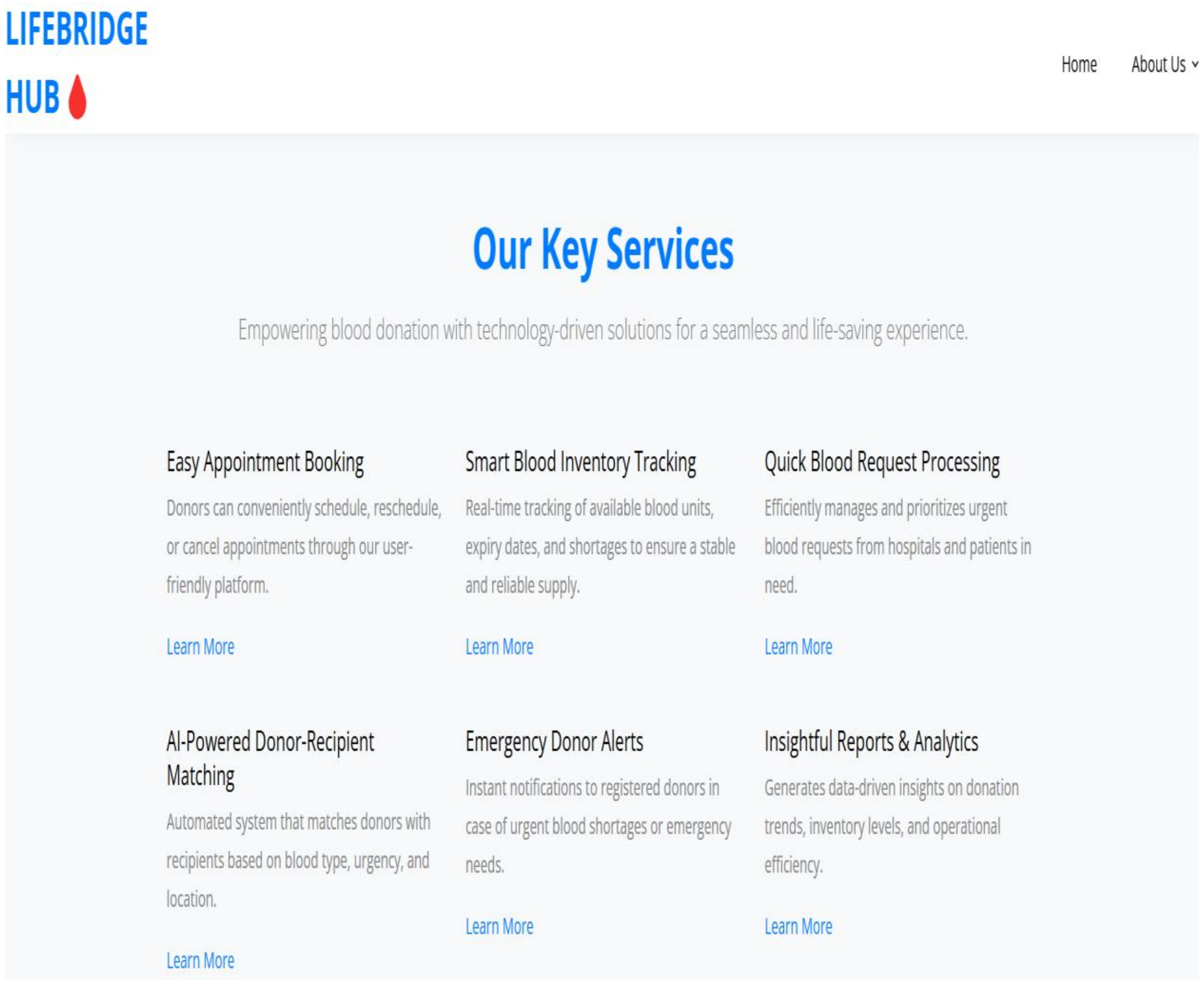
## 02.Get an approval

The **system checks the blood inventory** in real-time to determine availability. It then **verifies the requester's credentials**, ensuring eligibility based on medical history, previous requests, and any hospital affiliations. If all criteria are met and the blood type is available, the system **auto-approves the request** for fulfillment. If discrepancies or urgent cases arise, the request is **routed for manual review** by an administrator or medical staff for further validation and prioritization.



## 03.Get your donor

Upon approval, the system **provides anonymized donor details** such as blood type, location, and availability while ensuring privacy compliance. Simultaneously, it **initiates the blood dispatch process**, coordinating with blood banks, hospitals, or delivery services for efficient transportation. Real-time tracking updates may be shared with the requester, ensuring timely and safe delivery to the designated medical facility.



LIFEBRIDGE HUB

## Our Key Services

Empowering blood donation with technology-driven solutions for a seamless and life-saving experience.

### Easy Appointment Booking

Donors can conveniently schedule, reschedule, or cancel appointments through our user-friendly platform.

Learn More

### Smart Blood Inventory Tracking

Real-time tracking of available blood units, expiry dates, and shortages to ensure a stable and reliable supply.

Learn More

### Quick Blood Request Processing

Efficiently manages and prioritizes urgent blood requests from hospitals and patients in need.

Learn More

### AI-Powered Donor-Recipient Matching

Automated system that matches donors with recipients based on blood type, urgency, and location.

Learn More

### Emergency Donor Alerts

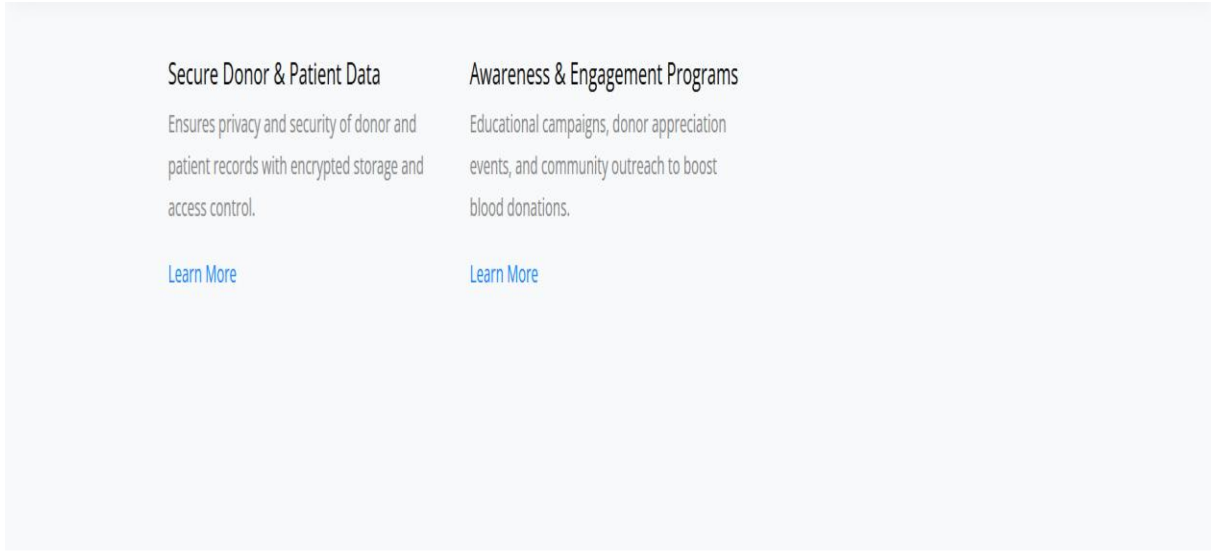
Instant notifications to registered donors in case of urgent blood shortages or emergency needs.

Learn More

### Insightful Reports & Analytics

Generates data-driven insights on donation trends, inventory levels, and operational efficiency.

Learn More



### Secure Donor & Patient Data

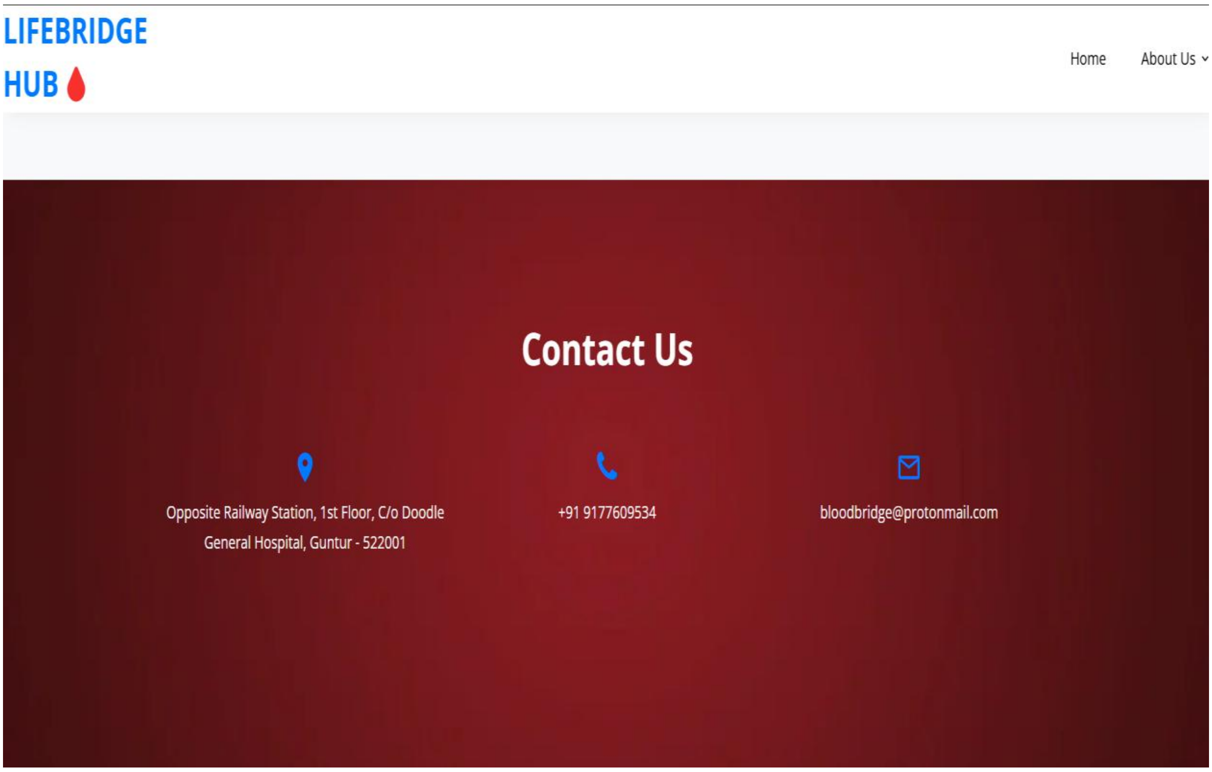
Ensures privacy and security of donor and patient records with encrypted storage and access control.

Learn More

### Awareness & Engagement Programs

Educational campaigns, donor appreciation events, and community outreach to boost blood donations.

Learn More

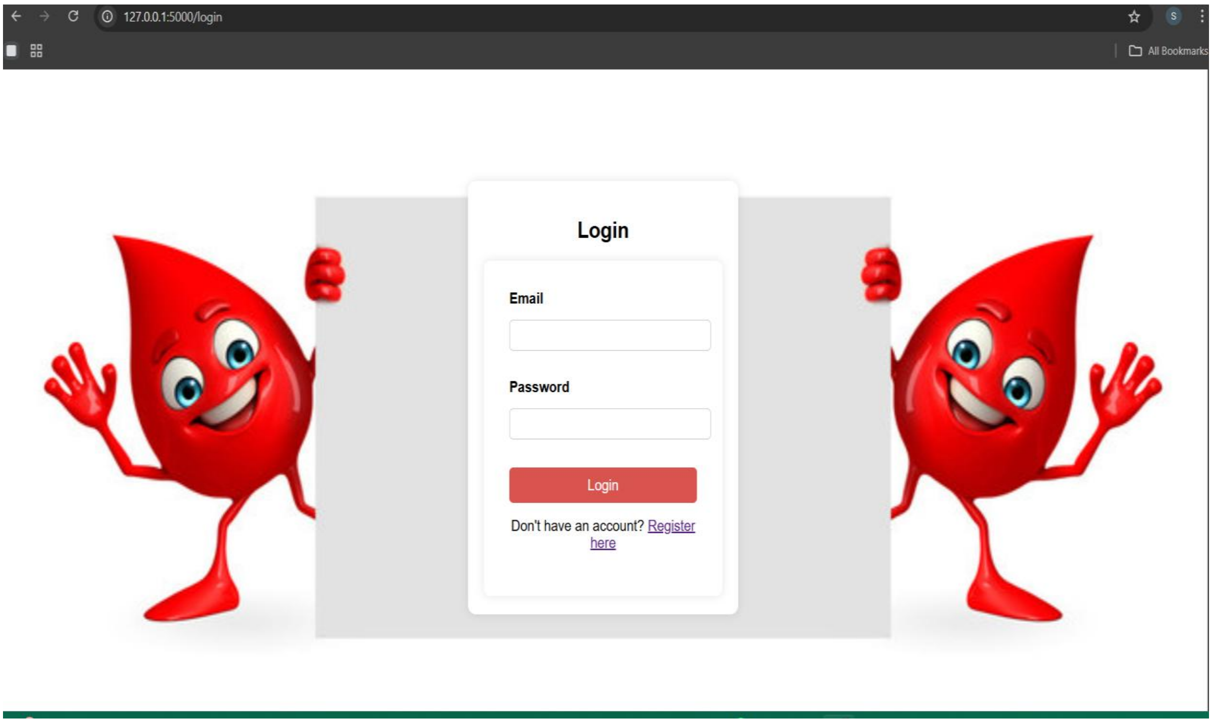


Contact Us

Opposite Railway Station, 1st Floor, C/o Doodle General Hospital, Guntur - 522001

+91 9177609534

bloodbridge@protonmail.com

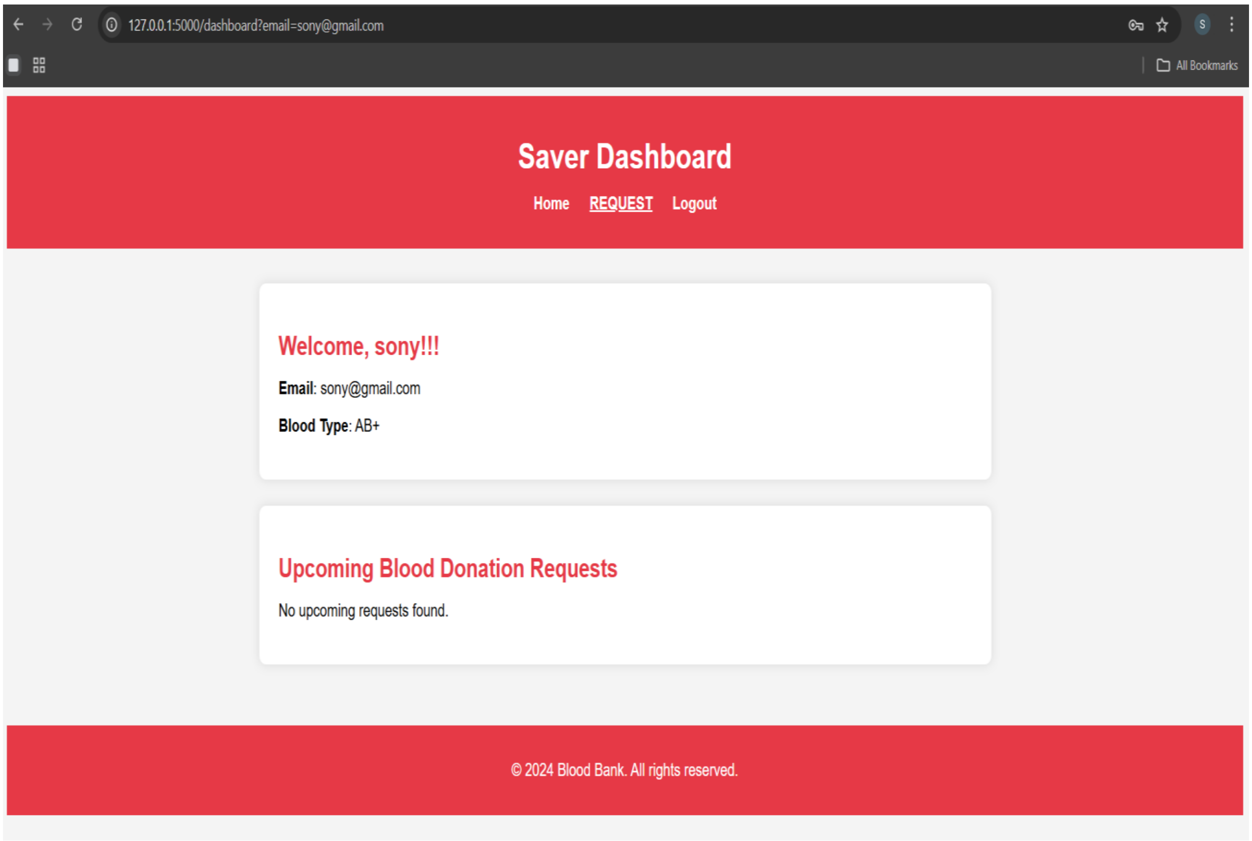


Login

Email

Password

Don't have an account? Register here



127.0.0.1:5000/dashboard?email=sony@gmail.com

Saver Dashboard

Home REQUEST Logout

Welcome, sony!!!

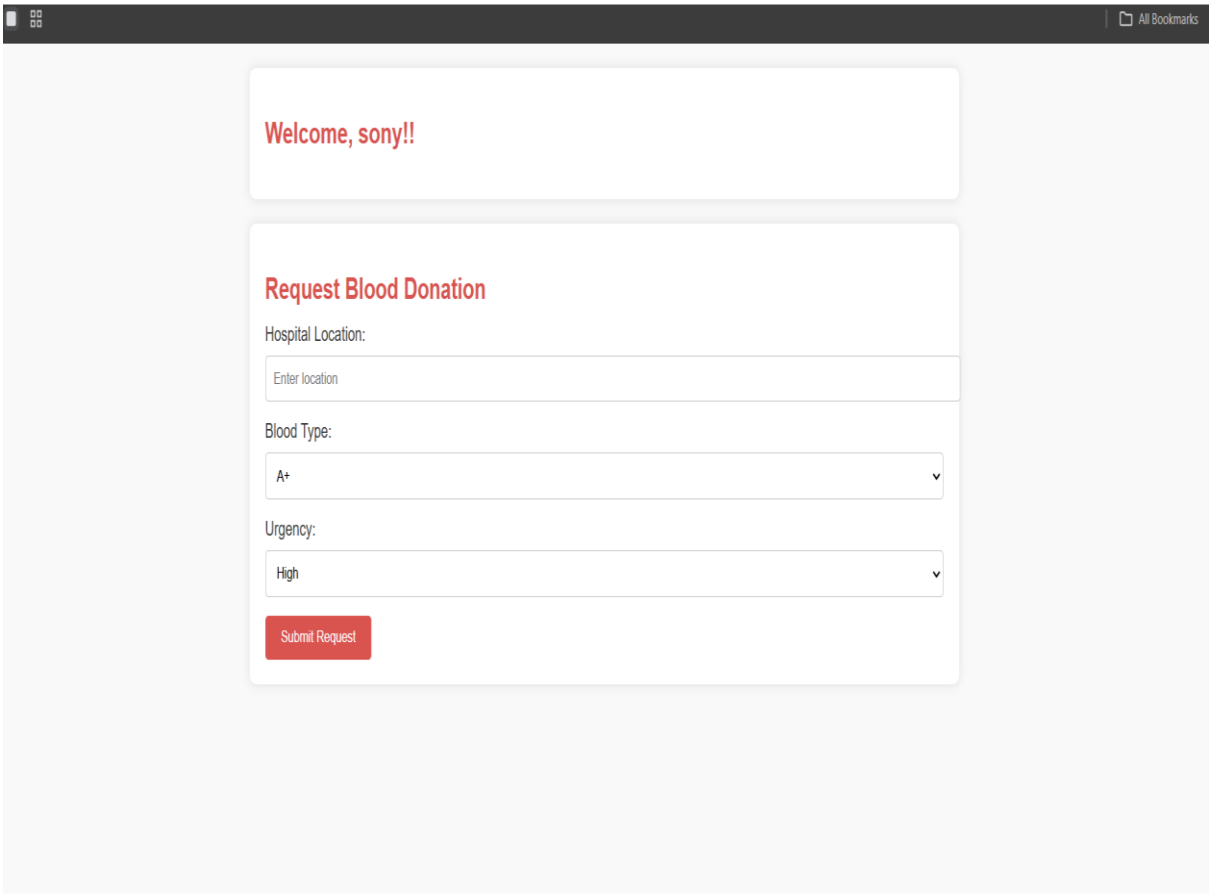
**Email**: sony@gmail.com

**Blood Type**: AB+

Upcoming Blood Donation Requests

No upcoming requests found.

© 2024 Blood Bank. All rights reserved.



Welcome, sony!!

Request Blood Donation

Hospital Location:

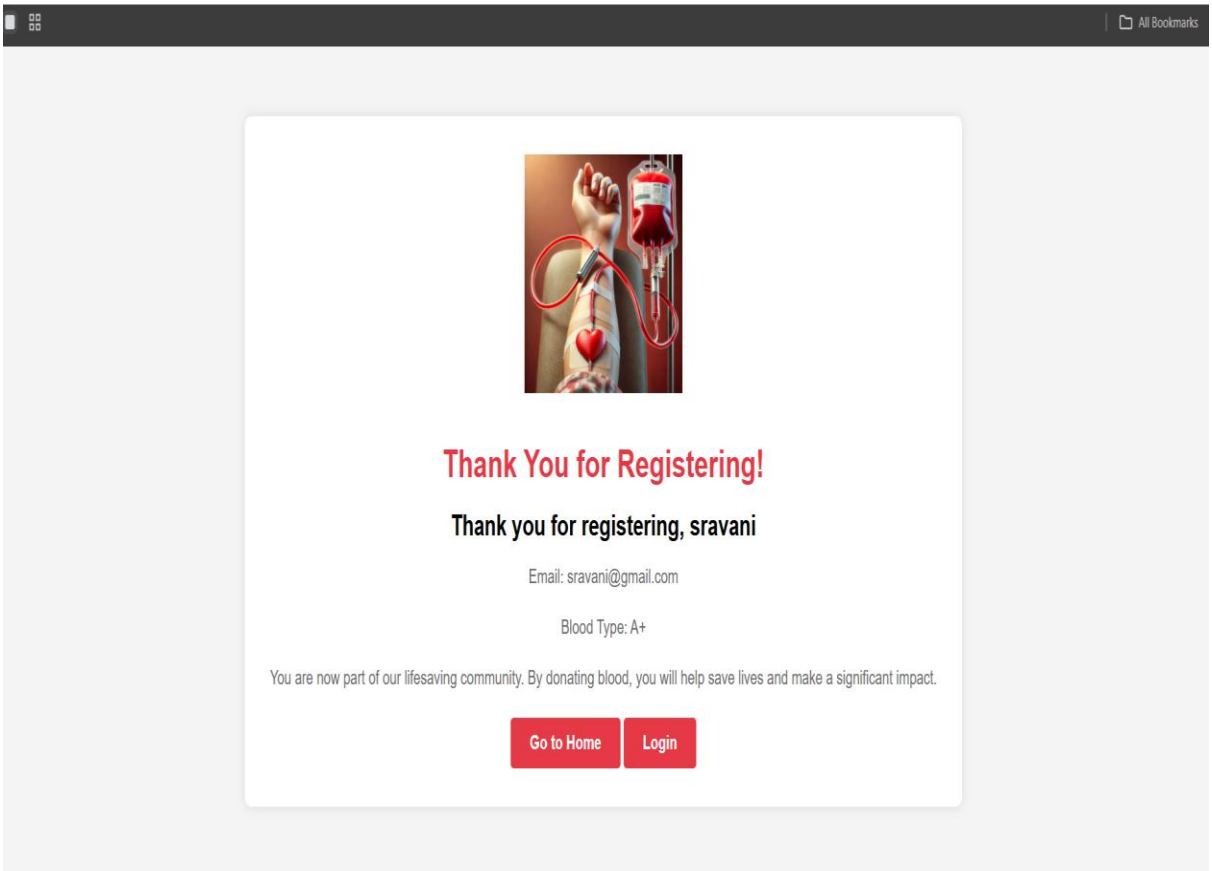
Enter location

Blood Type:

A+

Urgency:

High



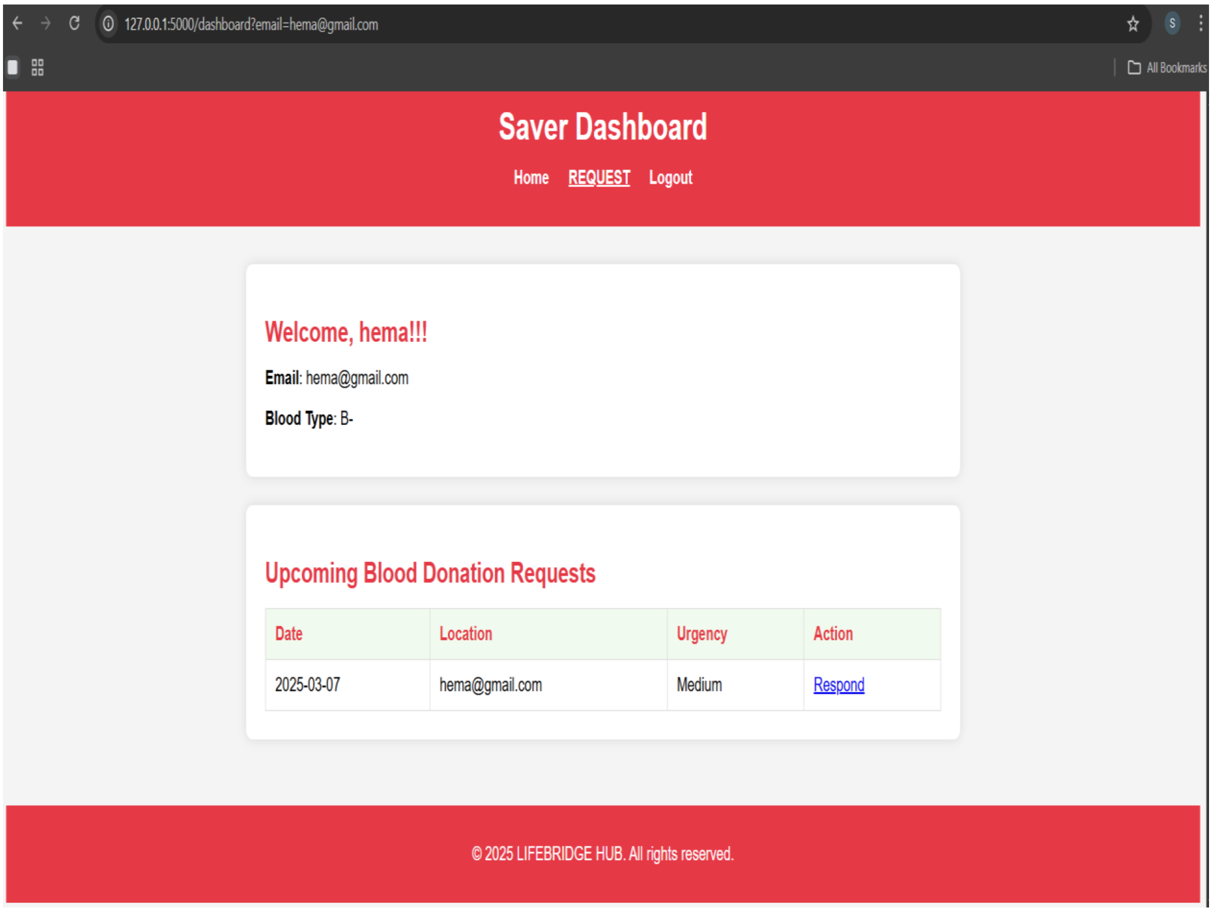
## Thank You for Registering!

### Thank you for registering, sravani

Email: sravani@gmail.com

Blood Type: A+

You are now part of our lifesaving community. By donating blood, you will help save lives and make a significant impact.



# Saver Dashboard

## Welcome, hema!!!

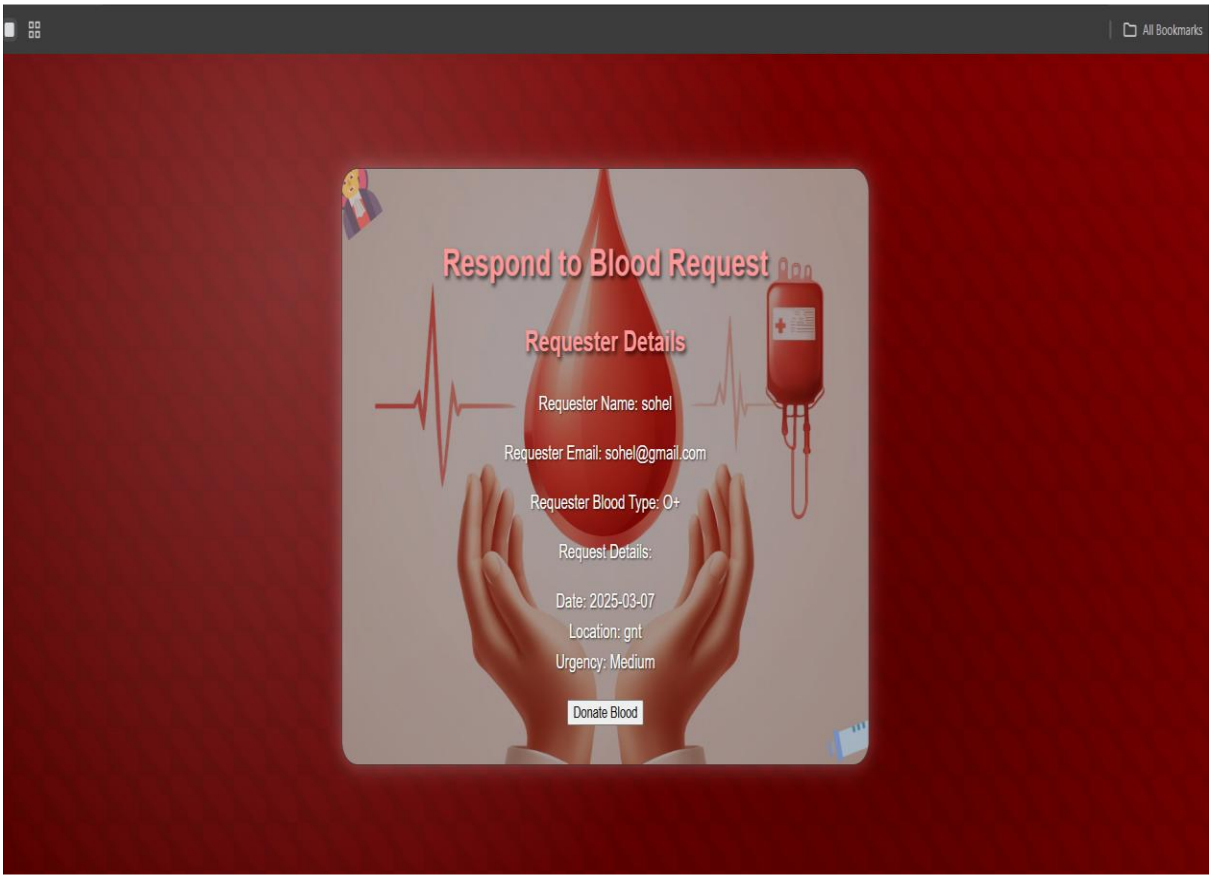
**Email**: hema@gmail.com

**Blood Type**: B-

## Upcoming Blood Donation Requests

| Date | Location | Urgency | Action |
|---|---|---|---|
| 2025-03-07 | hema@gmail.com | Medium | Respond |

© 2025 LIFEBRIDGE HUB. All rights reserved.



Respond to Blood Request

Requester Details

Requester Name: sohel

Requester Email: sohel@gmail.com

Requester Blood Type: O+

Request Details:

Date: 2025-03-07
Location: gnt
Urgency: Medium

## Conclusion:

This document provides a comprehensive guide for setting up the Blood Bridge platform using AWS services, emphasizing the key steps involved in creating an RDS database, developing a Flask application, and deploying it on an EC2 instance. By adhering to these milestones and activities, you can establish a robust, scalable web application tailored to efficiently manage blood donation requests. Leveraging EC2 for web hosting and RDS for database management ensures a high-performance, reliable infrastructure that can grow with the needs of your application.