Aptech COMPUTER EDUCATION
Unleash your potential

TECHWIZ 4 A GLOBAL I.T. COMPETITION

Techwiz 4 – A Global IT Competition

StreamTrace

Web Application Development

Developed and designed by
Hacktivists of Aptech Qatar

Table of Contents

Problem

The challenge is to develop "StreamTrace," a user-friendly web application enabling users to efficiently manage their streaming subscriptions. The application must offer functions like adding, modifying, and removing streaming services, marking favorites, setting reminders for payments and shows, and allowing searches, sorting, and filtering. This responsive web application aims to provide subscription tracking, renewal dates, payments, and favorite shows for individuals, ensuring an organized and convenient streaming experience.

Solution

To build the web application, we chose to use the **Flask**, a **Python** framework to build small web applications. For storing data, we integrated **MongoDB** with the help of **PyMongo** to connect our Flask web app with the Online database. The reason for us to choose these two is that they are extremely easy and fast to build any kind of web application in a short period of time. The frontend was designed using **Bootstrap**, **jQuery**, **DataTables**, and **Chart.js**, which helped us to create a visually pleasing and responsive interface. These technologies together ensure that tracking subscriptions is smooth and that users have an improved experience on the StreamTrace platform.

Design Specifications

1. Frontend

- The user interface is structured using Bootstrap components, following a clean and organized layout ensuring responsiveness and consistent design principles.
- The admin dashboard is structured using CSS Grid layout module.

2. Backend

- The Flask application is organized using software development standards, emphasizing cohesion and coupling.
- Different functionalities are handled through various endpoints, including user registration, authentication, account management, admin functions, and data scraping.

- Endpoints seamlessly interact with the frontend to serve data and manage user requests.

3. Database

- Defined collections include Users, Services, Subscriptions, Watchlist, and UserWatchlist.
- Data stored includes User Details, Streaming Service Provider Details, Subscription Details, and User Watchlist (movies and TV shows).
- Data retrieval, insertion, and modification are handled using Flask PyMongo, ensuring efficient data management.

4. Data Handling

- Data is managed using a combination of client-side and server-side validation.
- User passwords are securely hashed using MD5 encryption.

5. Subscription Tracking and Management

- Users can conveniently add, modify, or delete subscriptions via their personalized dashboard.
- Reminders for payments and shows are implemented by calculating and displaying the remaining days before subscription expiration.
- Backend processes utilize JavaScript to calculate and display remaining days for reminders.

6. UI Components (Bootstrap, jQuery, DataTables, Chart.js)

- UI components are integrated by creating individual components in separate files and unifying them using Jinja's extend and include functions.
- Bootstrap simplifies CSS styling, jQuery enhances JavaScript functionality, DataTables facilitates tabular data presentation with search, sort, and filter options, and Chart.js enables data visualization.

7. Responsive Design

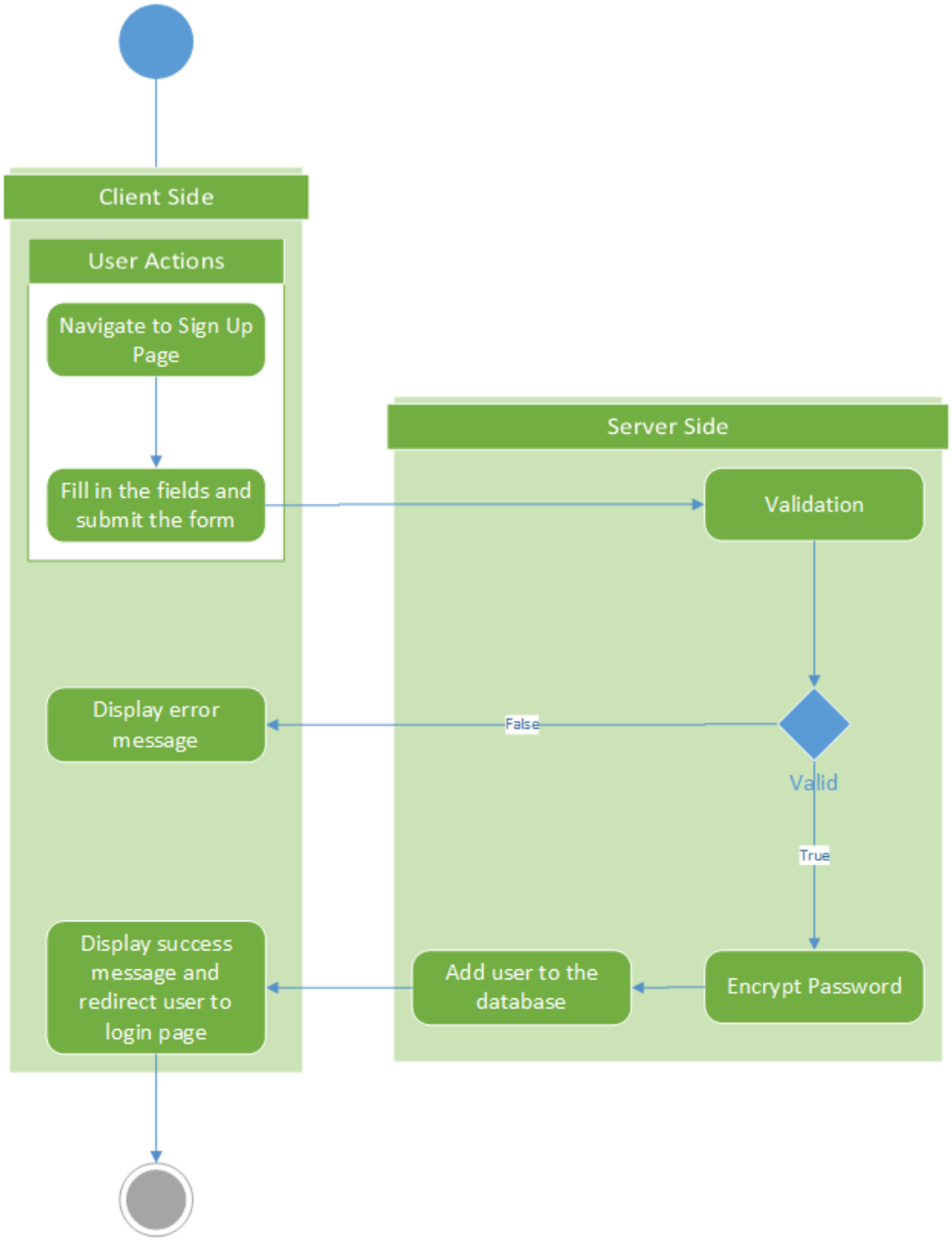
- Responsive design is ensured through the implementation of media queries along with Bootstrap classes, making the application usable across diverse devices and screen sizes.

8. Error Handling

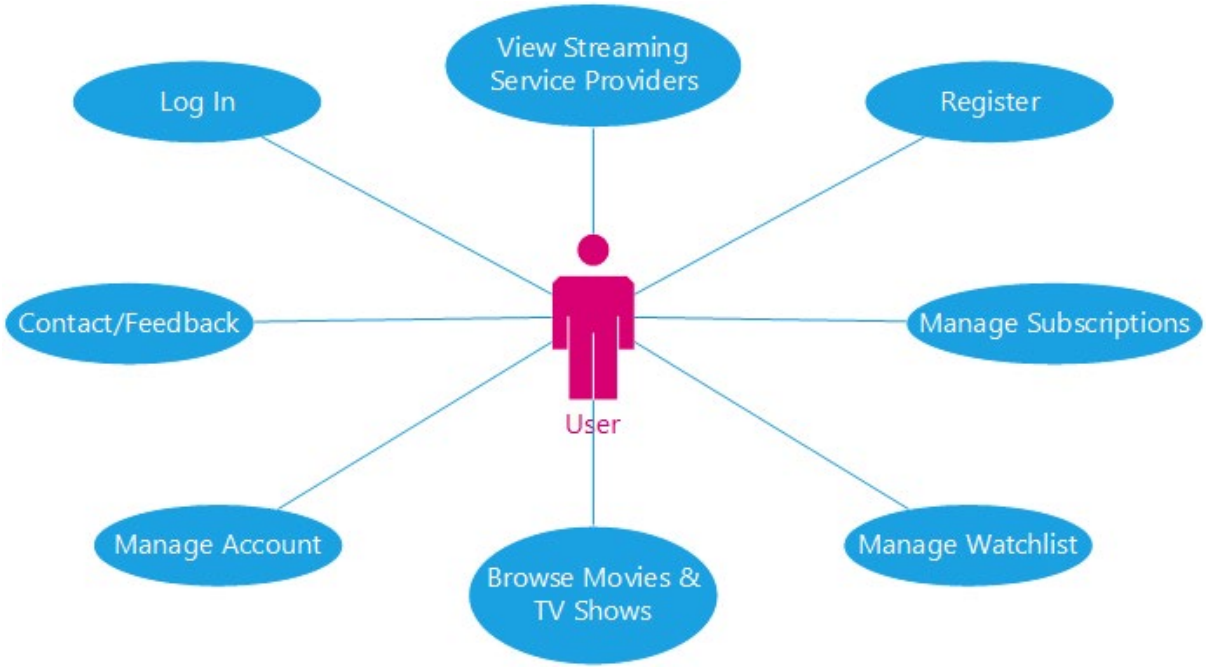
- Robust error handling is maintained using Python's exception handling (try and except) mechanism across functions, contributing to a smooth user experience.

Diagrams

1. Activity Diagram – Account Registration

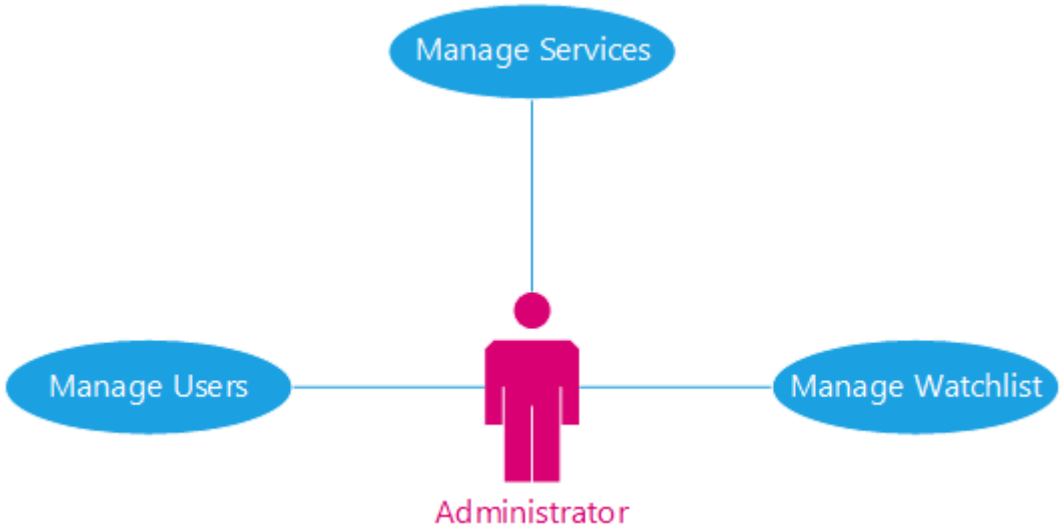


2. Use Case Diagram – User



s

3. Use Case Diagram – Admin



Database Design

Collection	Field	Field Type
Users	_id	ObjectId
	email	String
	username	String
	password	String
	age	Int32
	first_name	String
	last_name	String
	phone	Int64
	roles	Array
Services	_id	ObjectId
	name	String
	logo	String
	logo_wide	String
Subscriptions	_id	ObjectId
	user_id	ObjectId
	service_id	ObjectId
	name	String
	Amount	Double
	expiry	Date
	archived	Boolean

Watchlist	_id	ObjectId
	type	String
	title	String
	year	Int32
	rating	Double
	poster	String
UserWatchlist	_id	ObjectId
	user_id	ObjectId
	watchlist_id	ObjectId
	date_added	Date

1. **Users Collection:**

- This collection stores user profile information.
- Each user document has a unique _id generated by MongoDB, along with attributes like email, username, hashed password, age, first name, last name, phone, and roles (such as "staff").
- Sample data:


```
_id: ObjectId('64d508b69a1ac3658532de8b')
email: "demo@example.com"
username: "DemoUser"
password: "7fa87c0bed13bdbd5bd8d37adc6f5e15"
age: 22
first_name: "Demo"
last_name: "User"
phone: 123456789
▾ roles: Array
    0: "staff"
```

2. Services Collection:

- This collection holds details about streaming service providers.
- Documents include an _id, service name, and logos stored as base64 image strings.
- Sample data:


```
_id: ObjectId('64d3e7cec004ea42a475de5a')
name: "Netflix"
logo: "iVBORw0KGgoAAAANSUhEUgAAAgAAAAIABAMAAAAGVsnJAAAAJFBMVEVHcEziCBPbCBLdCB…"
logo_wide: "/9j/4AAQSkZJRgABAQAAAQABAAD/2wCEAAsLCwsLCwwNDQwREhASERgWFBQWGCUaHBocGi…"
```


3. Subscriptions Collection:

- This collection tracks users' subscriptions.
- Documents contain an _id, subscription name, amount, expiry date, a reference to the associated service (from the Services collection), and the user who owns the subscription.
- Sample data:


```
_id: ObjectId('64d6a5f6f73b5f2a66def9b4')
name: "Demo"
amount: 64.99
expiry: 2023-07-27T00:00:00.000+00:00
service: ObjectId('64d3f0ea0f897427dde2bd91')
user: ObjectId('64d508b69a1ac3658532de8b')
archived: true
```

4. Watchlist Collection:

- This collection records movies or TV shows that users want to watch.
- Each document contains an _id, type (movie or TV show), title, year, rating, and a link to the poster.
- Sample data:

```
_id: ObjectId('64d3c429c8b627c12d6064f2')
type: "Movie"
title: "The Shawshank Redemption"
year: 1994
rating: 9.3
poster: "https://m.media-amazon.com/images/M/MV5BNDE3ODcxYzMtY2YzZC00NmNlLWJiND…"
```

5. UserWatchlist Collection:

- This collection links users to the items they've added to their watchlist.
- Documents have an _id, a reference to the user (from the Users collection), a reference to the watchlist item (from the Watchlist collection), and a date indicating when the item was added.
- Sample data:

```
_id: ObjectId('64d66ce049c2c4646d104a39')
user_id: ObjectId('64d508b69a1ac3658532de8b')
watchlist_id: ObjectId('64d3c429c8b627c12d606570')
date_added: 2023-08-11T21:15:19.893+00:00
```

In MongoDB, each collection acts as a container for documents, which are essentially JSON-like objects. The _id field is automatically generated by MongoDB as a unique identifier for each document. The design allows to store related data in separate collections while maintaining the ability to link them through references (such as user IDs, service IDs, etc.).

MongoDB's schema flexibility enables to add or modify fields within documents without needing to adhere to a strict structure. This is when the app may evolve over time.

Overall, this design ensures efficient data retrieval and management, as well as the ability to perform queries that involve multiple collections through references. MongoDB calls this aggregation which helps in combining two or more collections, searching, sorting, and filtering easily. This enhances the performance and scalability of the web application.

Source Code

This report only discusses the **functional requirements** mentioned in the Software Requirements Specification (SRS). To see the whole project source code, check out the GitHub link provided with this document.

1. Account Registration

1.1. Frontend

```html
<form method="post" class="needs-validation" novalidate>
  <div class="mb-4">
    <label for="email" class="form-label">Email</label>
    <input name="email" id="email" type="email" placeholder="Enter your email address"
      class="form-control" required autocomplete="off">
    <div class="invalid-feedback">Please enter a valid email address.</div>
  </div>

  <div class="mb-4">
    <label for="username" class="form-label">Username</label>
    <input name="username" id="username" type="text" placeholder="Create a username"
      class="form-control" required autocomplete="off" minlength="5" pattern="[A-Za-z0-9]{5,}">
    <div class="invalid-feedback">Please choose a 5+ characters long alphanumeric username.</div>
  </div>

  <div class="mb-4">
    <label for="password" class="form-label">Password</label>
    <input name="password" id="password" type="password" placeholder="Create a password"
      class="form-control" required autocomplete="off" minlength="6">
    <div class="invalid-feedback">Please choose a 6+ characters long password.</div>
  </div>

  <button type="submit" class="btn btn-lg btn-purple bg-gradient px-5">Sign Up</button>
</form>
```

This form takes email, username, password input from client side and sends a post request to the server.

1.2. Backend

```python
@auth.route("/signup/", methods=["GET", "POST"])
def signup():
    form_fields = {"email", "username", "password"}

    if request.method == "POST":
        try:
            form = request.form.to_dict()
            validated = True

            # Check if all fields exist in the form data
            if set(form.keys()) == form_fields:
                # Validate email
                if not validate_email(form["email"]):
                    validated = False
                    flash("Email is either not valid or already exists.")

                # Validate username
                if not validate_username(form["username"]):
                    validated = False
                    flash("Username is not valid.")

                # validate password
                if not validate_password(form["password"]):
                    validated = False
                    flash("Password is not valid.")
            else:
                validated = False
                flash("Please fill all fields.")

            if validated:
                form["password"] = encrypt_message(form["password"])
                db.Users.insert_one(form)
                flash("Successfully created a new account. Please login to continue.")
                return redirect(url_for("auth.login"))
        except:
            flash("Something went wrong. Please try again later.")
            pass

    return render_template("auth/signup.html")
```

In the server-side, each input is validated by functions given below. Once validated, the code encrypts the user password using MD5 hash algorithm and saves their data in the database.

```python
# Function to get user by email
def get_user(email):
    user = None
    try:
        user = db.Users.find_one({"email": email})
    except:
        pass
    return user


# Function to validate email address during signup
def validate_email(email):
    valid = False
    if email:
        if not get_user(email):
            valid = True
    return valid


# Function to validate username during signup
def validate_username(username):
    valid = False
    if username:
        if len(username) >= 5 and username.isalnum():
            valid = True
    return valid


# Function to validate password during signup
def validate_password(password):
    valid = False
    if password:
        if len(password) >= 6:
            valid = True
    return valid
```

These are the functions used to validate user during the account registration process.

The below function is used to encrypt the password.

```python
# Function to encrypt a string using MD5 hashing
def encrypt_message(message):
    encrypted_message = hashlib.md5(message.encode()).hexdigest()
    return encrypted_message
```

2. Login

2.1. Frontend

```html
<form method="post" class="needs-validation" novalidate>
  <div class="mb-4">
    <label for="email" class="form-label">Email</label>
    <input name="email" id="email" type="email" placeholder="Enter your email address"
      class="form-control" required>
    <div class="invalid-feedback">Please enter your registered email address.</div>
  </div>

  <div class="mb-4">
    <label for="password" class="form-label">Password</label>
    <input name="password" id="password" type="password" placeholder="Enter your password"
      class="form-control" required>
    <div class="invalid-feedback">Please enter your password.</div>
  </div>

  <button type="submit" class="btn">Log In</button>
</form>
```

This form takes email and password from the html input elements and posts a request to the server for verification.

2.2. Backend

```python
@auth.route("/login/", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        try:
            user = {}
            email = request.form.get("email")
            password = request.form.get("password")

            ...

        except:
            flash("Something went wrong. Please try again later.")
            pass

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

```python
validated = True
# Check if email and password exists
if email and password:
    # Find user
    user = get_user(email)
    if user:
        password = encrypt_message(password)
        # Validate password
        if not user["password"] == password:
            validated = False
            flash("Password in incorrect.")
    else:
        validated = False
        flash("Email not found.")
else:
    validated = False
    flash("Please fill all fields.")

if validated:
    user["_id"] = json_util.dumps(user["_id"]).split('"')[3]
    session["user"] = user
    session.permanent = True
    flash("Successfully logged in.")
    return redirect(url_for("account.dashboard"))
```

This validates if the user exists and if their password match with saved password.

Once validated, the user is added to the session and is redirected to the user dashboard page.

3. User Dashboard

3.1. Frontend – HTML

```html
<table id="user-subs" class="table">
  <thead>
    <tr>
      <th class="text-center">Streaming Service</th>
      <th class="text-center">Account</th>
      <th class="text-center">Amount</th>
      <th class="text-center">Expiry</th>
      <th class="text-center" data-type="@data-value">Status</th>
      <th class="text-center">Manage</th>
    </tr>
  </thead>
  <tbody></tbody>
  <tfoot>
    <tr>
      <th colspan="2">Total</th>
      <th id="total-amount" class="text-center"></th>
      <th colspan="3"></th>
    </tr>
  </tfoot>
</table>
```

This is a simple table that will list of all user subscriptions.

3.2. Frontend – JavaScript

```javascript
$(document).ready(async function () {
  await get_user_subs()
});

async function get_user_subs() {
  const url = "/user-subs";
  sub_table = $("#user-subs").DataTable({ ajax: { url: url}});
}
```

This JS function uses DataTables ajax feature to get user subscriptions from the server and add them to the HTML table.

3.3. Backend

```python
# Function to get User Subscriptions
@account.route("/user-subs")
def user_subs():
    subs = []
    try:
        user_id = session["user"]["_id"]
        filter = {"user": ObjectId(user_id)}
        pipeline = [
            {"$match": filter},
            {
                "$lookup": {
                    "from": "Services",
                    "localField": "service",
                    "foreignField": "_id",
                    "as": "service",
                }
            },
            {"$sort": {"expiry": 1}},
        ]
        subs = list(db.Subscriptions.aggregate(pipeline))
        subs = json_util.dumps(subs)
    except:
        pass
    return subs
```

This function is used to get user subscriptions from the database.

The function filters the database records by user id and looks up on the Services collection to get the streaming service provider details of each subscription.

4. Account Settings

4.1. Personal Details - Frontend – HTML

```html
<form id="update-personal" class="needs-validation" novalidate>
  <label for="username" class="form-label fw-medium text-secondary">Username</label>
  <input value="{{session.user.username}}" name="username" id="username" type="text"
    placeholder="enter a username" class="form-control" required autocomplete="off"
    minlength="5" pattern="[A-Za-z0-9]{5,}">
  <div class="invalid-feedback">Please choose a 5+ characters long alphanumeric username.</div>

  <label for="first_name" class="form-label">First name</label>
  <input value="{{session.user.first_name}}" name="first_name" id="first_name" type="text"
    placeholder="Your first name" class="form-control" autocomplete="off">

  <label for="last_name" class="form-label">Last name</label>
  <input value="{{session.user.last_name}}" name="last_name" id="last_name" type="text"
    placeholder="Your last name" class="form-control" autocomplete="off">

  <label for="phone" class="form-label">Phone number (with country code)</label>
  <input value="{{session.user.phone}}" name="phone" id="phone" type="number"
    placeholder="Your phone number" class="form-control form-control-lg" autocomplete="off">

  <label for="age" class="form-label">Age</label>
  <input value="{{session.user.age}}" name="age" id="age" type="number" placeholder="Your age"
    class="form-control" autocomplete="off">

  <button type="submit" class="btn">Update</button>
</form>
```

This form takes username, first and last name, phone, and age inputs from the user.

4.2. Personal Details – Frontend – JavaScript

```javascript
$('form#update-personal').on('submit', async function (e) {
  e.preventDefault()
  const url = `/update-personal`
  await send_request(this, url)
})
```

This function is executed when the HTML form is submitted.

The previous calls the below function.

```
async function send_request(form, url) {
  const form_btn = $(form).find('button[type="submit"]')
  const data = $(form).serialize()
  if (form.checkValidity()) {
    $.post(url, data).done(function (response) {
        notify(response)
      });
  }
}
```

This function sends a request to the server to update user data. This function is also used to update user email and password which will be discussed below.

4.3. Personal Details – Backend

```
@account.route("/update-personal", methods=["POST"])
def update_personal_details():
    response = ""
    try:
        data = request.form.to_dict()
        valid = True

        if not validate_username(data["username"]):
            valid = False
            response = "Username is not valid."

        data["phone"] = int(data["phone"])
        data["age"] = int(data["age"])

        if valid:
            user_id = session["user"]["_id"]
            db.Users.update_one({"_id": ObjectId(user_id)}, {"$set": data})
            session["user"].update(data)
            response = "Personal details updated successfully!"
    except:
        response = "Failed to update."
    return response
```

4.4. User Email – Frontend – HTML

```html
<form id="update-email" class="needs-validation" novalidate>
  <label for="email" class="form-label">Email address</label>
  <input value="{{session.user.email}}" name="email" id="email" type="email"
    placeholder="enter your email" class="form-control" required autocomplete="off">
  <div class="invalid-feedback">Please enter a valid email address.</div>

  <button type="submit" class="btn">Update</button>
</form>
```

4.5. User Email – Frontend – JavaScript

```javascript
$('form#update-email').on('submit', async function (e) {
  e.preventDefault()
  const url = `/update-email`
  await send_request(this, url)
})
```

4.6. User Email – Backend

```python
@account.route("/update-email", methods=["POST"])
def update_email():
    response = ""
    try:
        email = request.form.get("email")
        if validate_email(email):
            user_id = session["user"]["_id"]
            db.Users.update_one({"_id": ObjectId(user_id)}, {"$set": {"email": email}})
            session["user"]["email"] = email
            response = "Email updated successfully!"
        else:
            response = "Email already exists."
    except:
        response = "Failed to update."
    return response
```

This function applies the same logic to validate email as the account registration process.

4.7. Password – Frontend – HTML

```html
<form id="update-password" class="needs-validation" novalidate>
  <label for="old_password" class="form-label">Old password</label>
  <input name="old_password" id="old_password" type="password" placeholder="enter your old password"
    class="form-control" required autocomplete="off">
  <div class="invalid-feedback">Please enter your old password.</div>

  <label for="new_password" class="form-label">New password</label>
  <input name="new_password" id="new_password" type="password" placeholder="enter your new password"
    class="form-control" required autocomplete="off">
  <div class="invalid-feedback">Please enter your new password.</div>

  <button type="submit" class="btn">Update</button>
</form>
```

4.8. Password – Frontend – JavaScript

```javascript
$('form#update-password').on('submit', async function (e) {
  e.preventDefault()
  const url = `/update-password`
  await send_request(this, url)
})
```

4.9. Password – Backend

```python
@account.route("/update-password", methods=["POST"])
def update_password():
    response = ""
    try:
        form = request.form.to_dict()
        valid = True
        if encrypt_message(form['old_password']) != session["user"]["password"]:
            valid = False
            return "Old password is incorrect."
        if not validate_password(form['new_password']):
            valid = False
            return "Password is not valid."

        if valid:
            user_id = session["user"]["_id"]
            data = {"password": encrypt_message(form['new_password'])}
            db.Users.update_one({"_id": ObjectId(user_id)}, {"$set": data})
            session["user"]["password"] = new_password
            response = "Password updated successfully!"
    except:
        response = "Failed to update."
    return response
```

This also uses the same logic as the sign-up process.

5. Streaming Service Provider Management

5.1. Frontend – HTML

```html
<form method="post" enctype="multipart/form-data">
  <div>
    <label for="name" class="form-label">Name:</label>
    <input name="name" id="name" type="text" class="form-control" placeholder="Netflix" required>
  </div>

  <div>
    <label for="logo" class="form-label">Logo:</label>
    <input name="logo" id="logo" type="file" accept="image/*" class="form-control" required>
    <span>Transparent background (square icon) (1:1)</span>
    <div id="logo-preview"></div>
  </div>

  <div>
    <label for="logo_wide" class="form-label">Logo (wide):</label>
    <input name="logo_wide" id="logo_wide" type="file" accept="image/*" class="form-control" required>
    <span >With background (16:9)</span>
    <div id="logo-wide-preview"></div>
  </div>

  <button type="submit" class="btn btn-success">Add new service</button>
</form>
```

This form takes service provider name and logos from the client-side.

5.2. Frontend – JavaScript

```javascript
$('input[name="logo"]').on('change', function () {
    previewFile(this, 'logo-preview')
  })

$('input[name="logo_wide"]').on('change', function () {
  previewFile(this, 'logo-wide-preview')
})
```

These two functions are executed when client chooses a logo.

The previous functions call the below function that displays the uploaded logos to the client.

```javascript
function previewFile(fileInput, previewId) {
  const preview = document.getElementById(previewId);

  for (const file of fileInput.files) {
    const reader = new FileReader();
    reader.onload = function (e) {
      if (file.type.startsWith('image/')) {
        const img = `<img src="${e.target.result}">`
        preview.innerHTML = img
      }
    };
    reader.readAsDataURL(file);
  }
}
```

5.3. Backend

```python
@admin.route("/services/new/", methods=["GET", "POST"])
def add_new_service():
    try:
        if request.method == "POST":
            name = request.form.get("name")
            logo = request.files.get("logo")
            logo_wide = request.files.get("logo_wide")

            # Convert images to base64
            logo = convert_img_to_base64(logo)
            logo_wide = convert_img_to_base64(logo_wide)
            db.Services.insert_one(
                {
                    "name": name,
                    "logo": logo,
                    "logo_wide": logo_wide,
                }
            )
            flash("Successfully added a new service!")
            return redirect(url_for("admin.service_manager", search=name))
    except:
        pass
    return render_template("admin/manager/new-service.html")
```

The server-side function takes the input from client-side and the streaming service provider data to the database.

Before adding the images to the database, the server converts the images to base64. Below is the function used to convert images to base64.

```
1 def convert_img_to_base64(image):
2     image = image.read()
3     image = base64.b64encode(image).decode("utf-8")
4     return image
```

The process of updating these records is the same as inserting new records.

6. Subscription Management

6.1. Frontend

```
<form method="post" class="needs-validation" novalidate>
  <div>
    <label for="name" class="form-label">Account Name</label>
    <input name="name" id="name" placeholder="Your account name" type="text"
      class="form-control form-control-lg" autocomplete="off" required>
    <div class="invalid-feedback">Please enter your {{service.name}} account name.</div>
  </div>

  <div>
    <label for="amount" class="form-label">Amount</label>
    <input name="amount" id="amount" placeholder="Subscription amount" type="number" step="0.01"
      class="form-control form-control-lg" autocomplete="off" required>
    <div class="invalid-feedback">Please enter a valid {{service.name}} subscription amount.</div>
  </div>

  <div>
    <label for="expiry" class="form-label">Expiry Date</label>
    <input name="expiry" id="expiry" placeholder="Expiry date" type="date" class="form-control"
      autocomplete="off" required>
    <div class="invalid-feedback">Please enter your {{service.name}} subscription's expiry date.</div>
  </div>

  <button type="submit" class="btn btn-lg btn-purple">Add new subscription</button>
</form>
```

This form is used to take subscriptions details like streaming service provider, account name, amount, and expiry date of the service.

6.2. Backend

```python
@account.route("/subscriptions/new/<service_id>", methods=["GET", "POST"])
def new_sub(service_id):
    service = db.Services.find_one({"_id": ObjectId(service_id)})
    if not service:
        return redirect(url_for("views.services"))

    try:
        form_fields = {"name", "amount", "expiry"}
        if request.method == "POST":
            form = request.form.to_dict()
            valid = True

            if set(form.keys()) == form_fields:
                # Check if subscription already exists
                if len(check_duplicate_sub(service_id, form["name"])) != 0:
                    valid = False
                    flash("A subscription with the same account already exists.")

                form["amount"] = float(form["amount"])
                form["expiry"] = datetime.strptime(form["expiry"], "%Y-%m-%d")

                if valid:
                    form["service"] = ObjectId(service_id)
                    form["user"] = ObjectId(session["user"]["_id"])
                    db.Subscriptions.insert_one(form)
                    flash("Subscription added successfully!")
                    return redirect(url_for("account.dashboard"))
            else:
                flash("Please fill all field.")
    except:
        flash("Something went wrong. Please try again later.")
    return render_template("account/new-sub.html", service=service)
```

This process uses the logic and error handling as my previous codes.

The process of updating and deleting records are also similar.

The process of **Watchlist Management** is similar to Subscription Management.

7. Search, Sort and Filter

Many parts of the web application offer to search, sort, and filter data. This document will not cover each and every part. This document has taken the process behind the **Browse** page to demonstrate how these functions work.

Step 1: Get Arguments and Set Parameters

```python
@views.route("/browse/")
def browse():
    page = request.args.get("page", default=1, type=int)
    search = request.args.get("search")
    title_type = request.args.get("type")
    sort_by = request.args.get("sort")
    page_details = {"page": page, "search": search, "type": title_type, "sort": sort_by}
    watchlist = get_watchlist(page, search, title_type, sort_by)
    return render_template(
        "browse.html", watchlist=watchlist, page_details=page_details
    )
```

The above code helps in getting arguments from requests made by client. Arguments include search query like keywords, sorting and filter details.

After collecting this information, a request is made to the database with the given parameters using the next function.

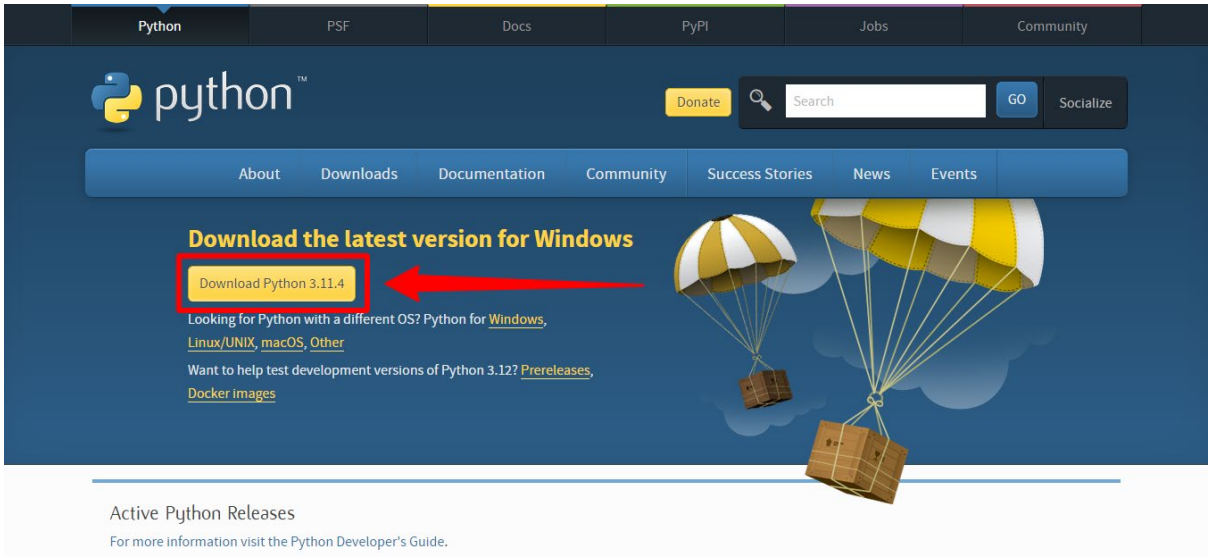
```python
def get_watchlist(page=1, search=None, title_type=None, sort_by=None):
    watchlist = []
    try:
        page = int(page)
        limit = 100
        skip = (page - 1) * limit

        filter = {}
        sort = [("year", -1)]
        if search:
            query = [{"title": {"$regex": i, "$options": "i"}} for i in search.split()]
            filter = {"$and": query}
        if title_type in ["Movie", "TV"]:
            filter["type"] = title_type
        if sort_by == "oldest":
            sort = [("year", 1)]
        elif sort_by == "rating_high":
            sort = [("rating", -1)]
        elif sort_by == "rating_low":
            sort = [("rating", 1)]
        elif sort_by == "title_asc":
            sort = [("title", 1)]
        elif sort_by == "title_desc":
            sort = [("title", -1)]

        watchlist = list(db.Watchlist.find(filter).sort(sort).skip(skip).limit(limit))
    except:
        pass
    return watchlist
```

This function helps to retrieve the filtered and sorted data from the database which is displayed to the user.

Installation Guide

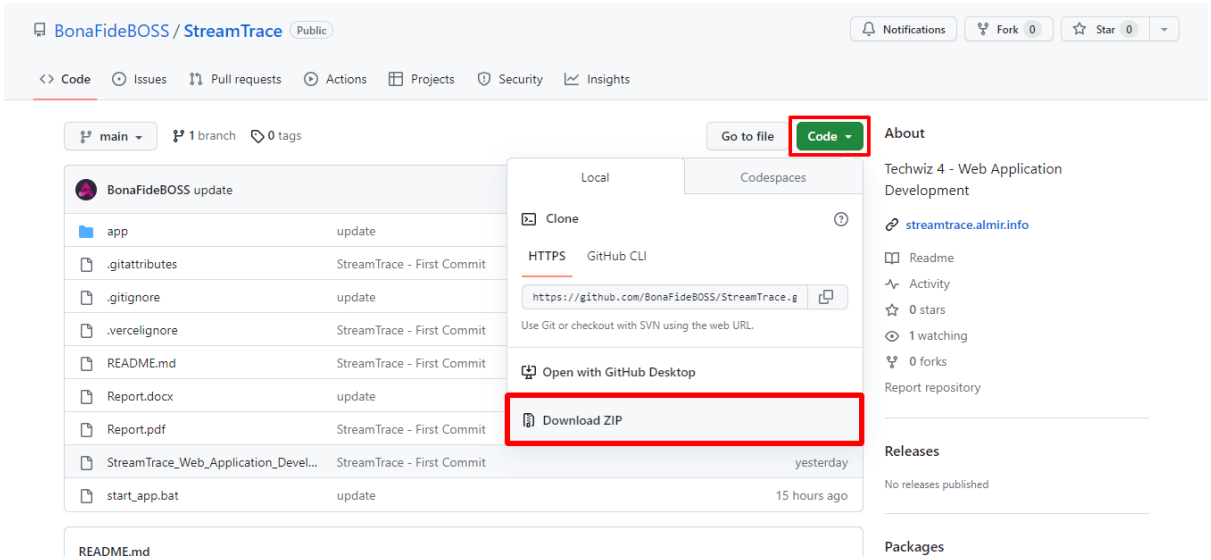
Step 1: Download and Install Python (latest version)

Link: https://www.python.org/downloads/



Step 2: Download project files from GitHub.

Link: https://github.com/BonaFideBOSS/StreamTrace



Step 3: Set up virtual environment

A virtual environment is not provided with project due to its huge file size.

1. Extract the zip file
2. Open command prompt inside the folder
3. Execute the command given below

```
python -m venv venv
```

4. Activate virtual environment

```
venv\scripts\activate
```

Step 4: Install requirements

1. Move inside the app folder

```
cd app
```

2. Run the command shown below

```
python -m pip install -r requirements.txt
```

Step 5: Run Flask

The last step is to run the command below, then open a browser and go to http://127.0.0.1:5000/.

```
python -m flask run
```

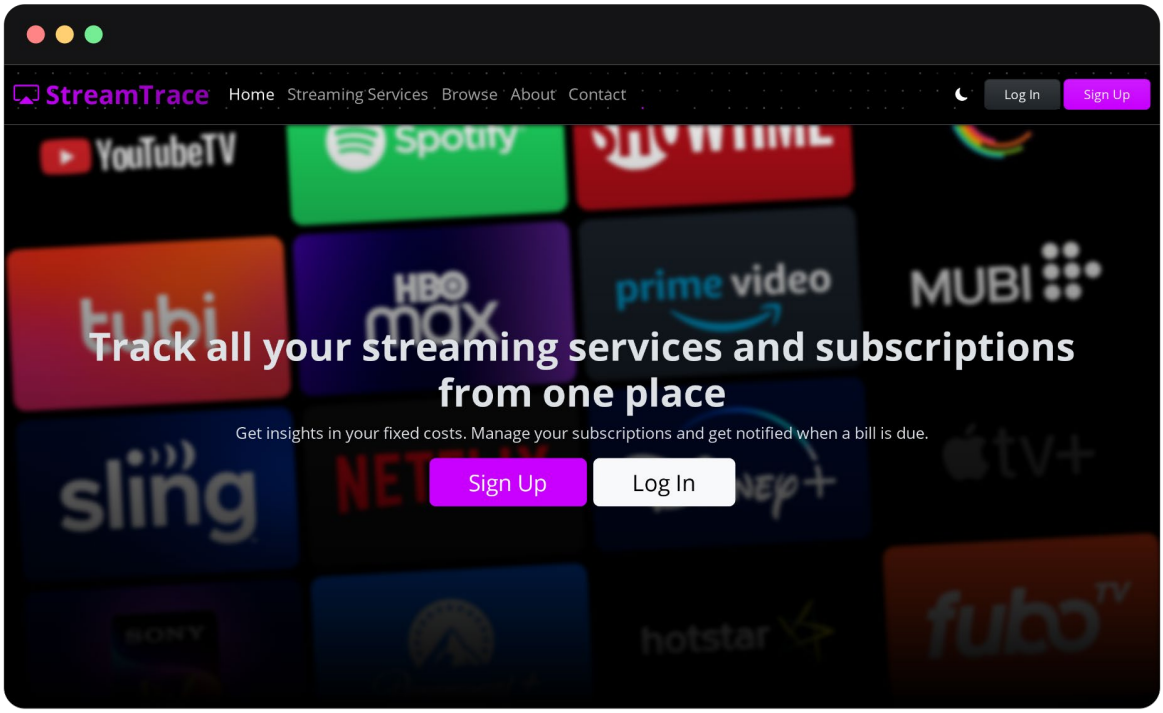
User Manual

This is a complete guide on how to use the web application.

Live website link

The quickest way to use the web app is by visiting the live website hosted online which is accessible to everyone, anywhere.

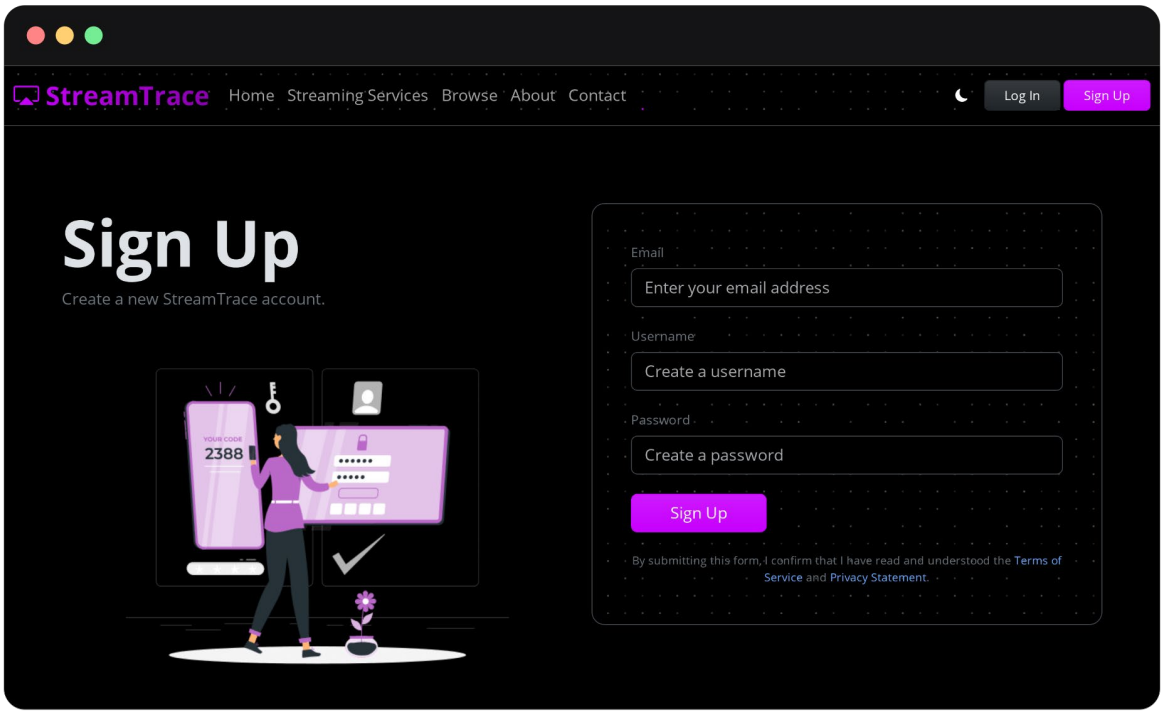
Link: https://streamtrace.almir.info/



1. Sign Up

Click on the sign-up button to go to the signup page.

Link: https://streamtrace.almir.info/signup/



Fill in all the fields to create a new account.

If you're too lazy to do that, take the demo account provided on the next page and move to the login page.

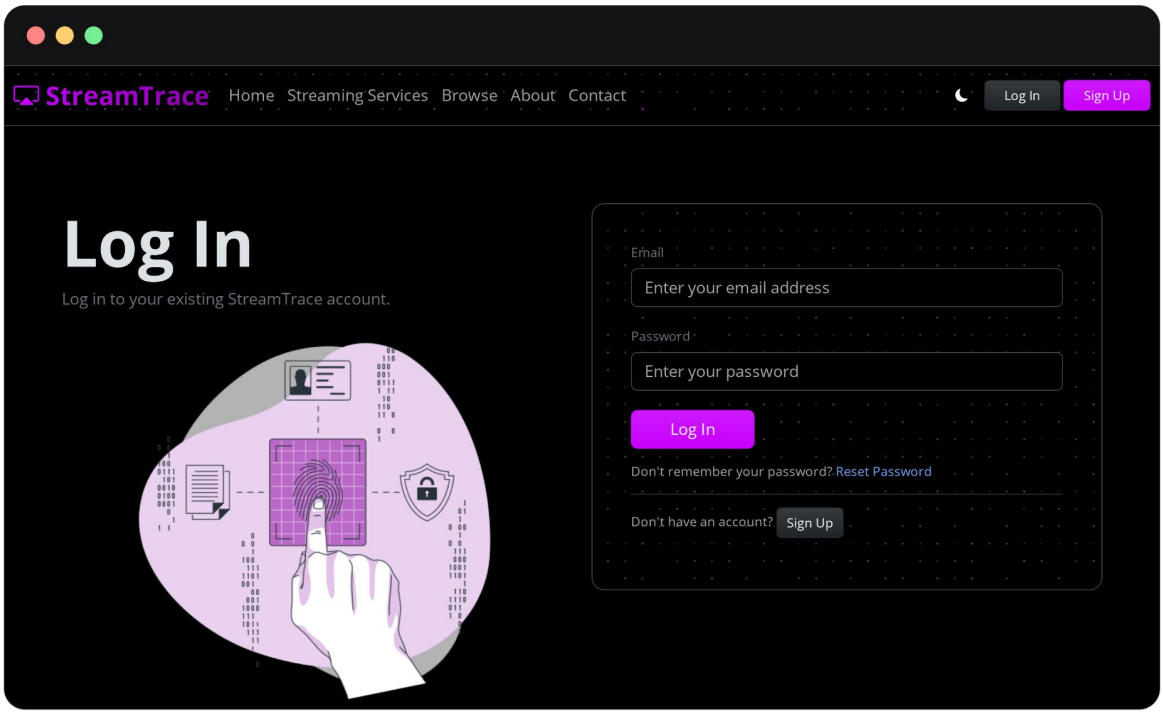
2. Log In

This page lets you log in to the web application and track your subscriptions.

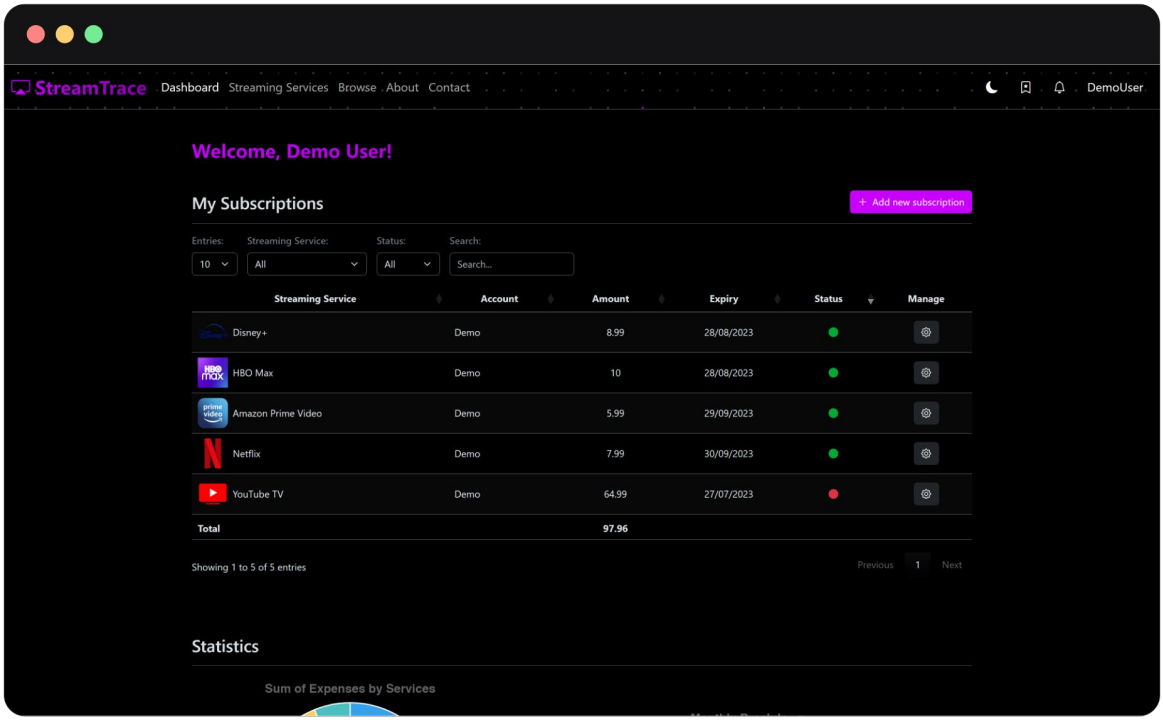
2.1. Demo Account

Email	demo@example.com
Password	$Demo@123

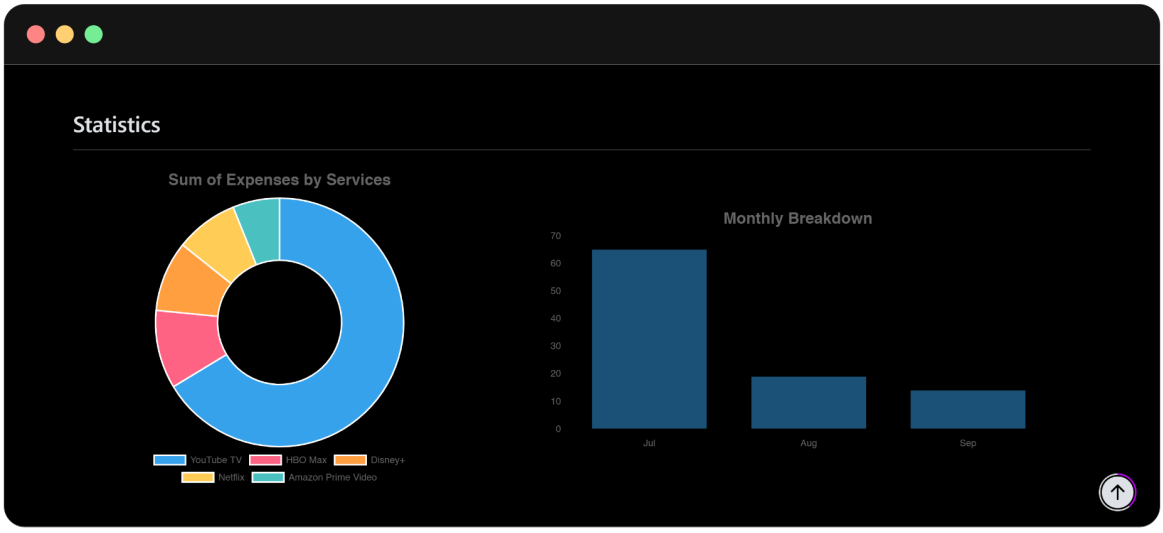
If you haven't created your own account, you can use the demo account provided above to log in to the website.



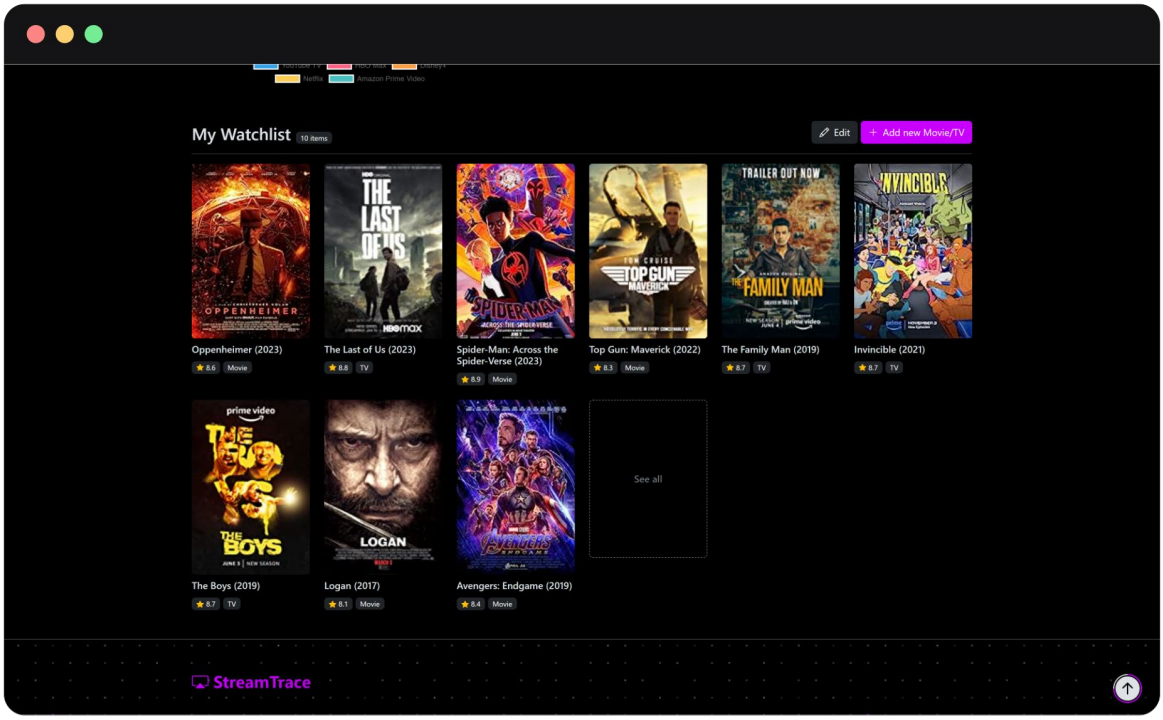
3. User Dashboard



On this page, users can see the list of subscriptions they have added to their account. You can also **search**, **filter**, and **sort**. This page also visualizes your data to show some statistics.



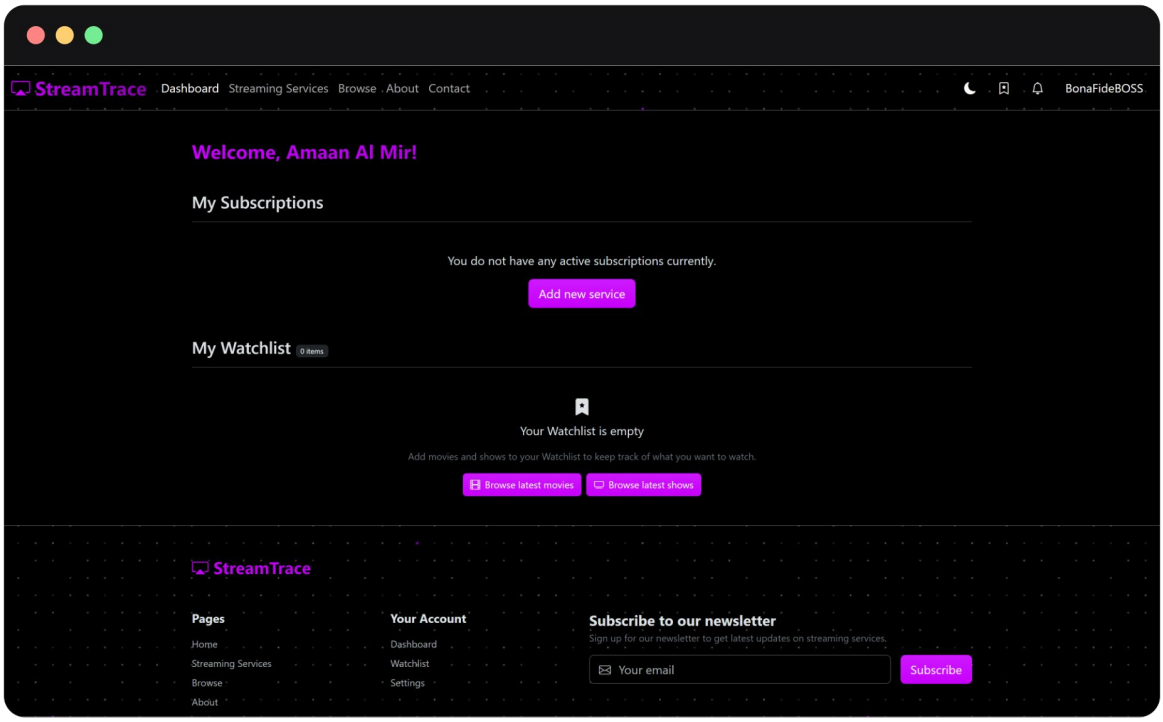
At the bottom of the page, you will see some recently items you have added to your watchlist.



Clicking on the **See all** button or the **edit** button will you take you the Watchlist page where you can see all the movies and TV shows in your watchlist.

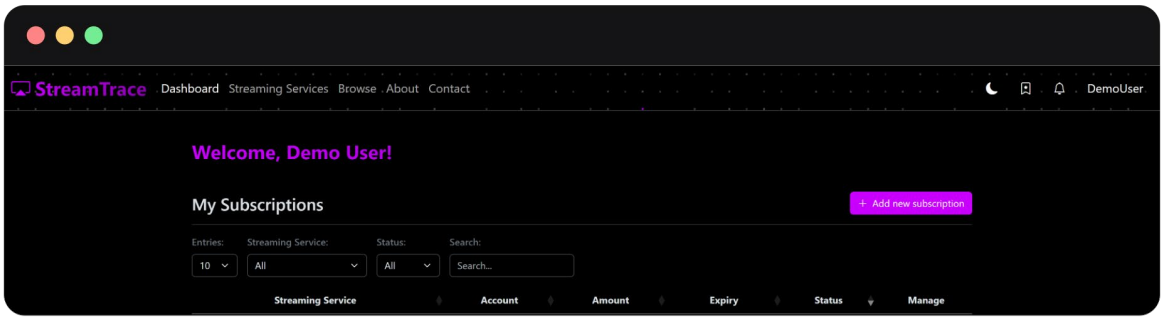
4. Add New Subscription

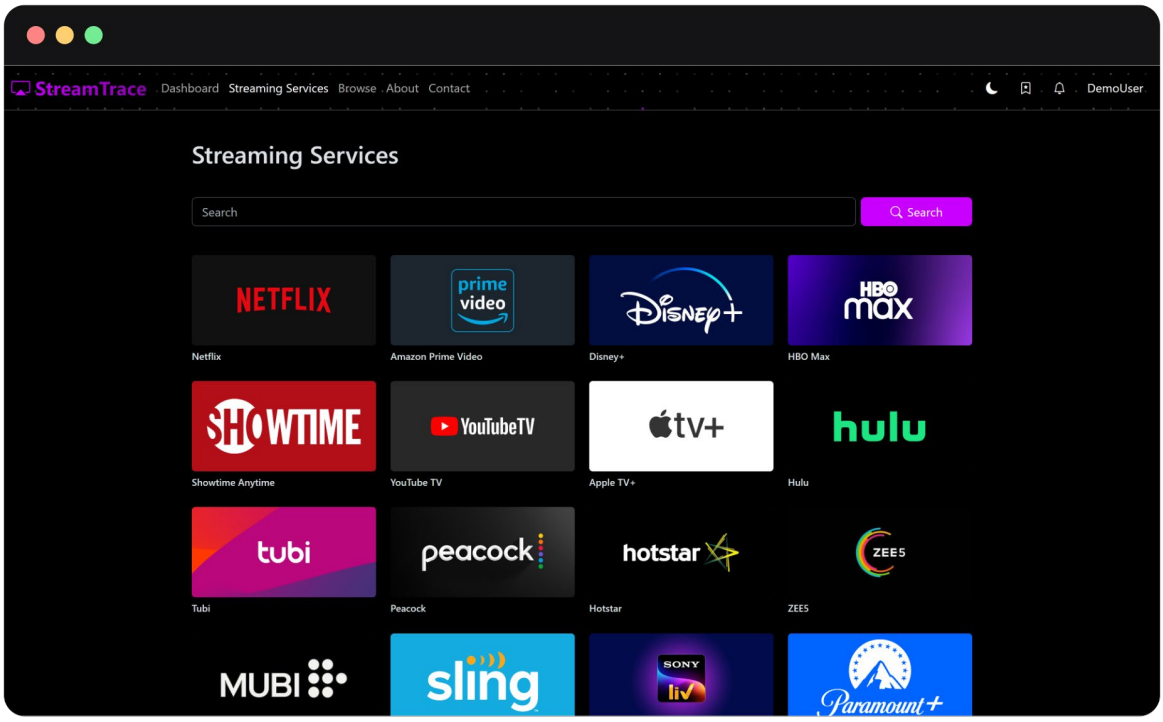
Now, if you're new the website, user dashboard would look this.



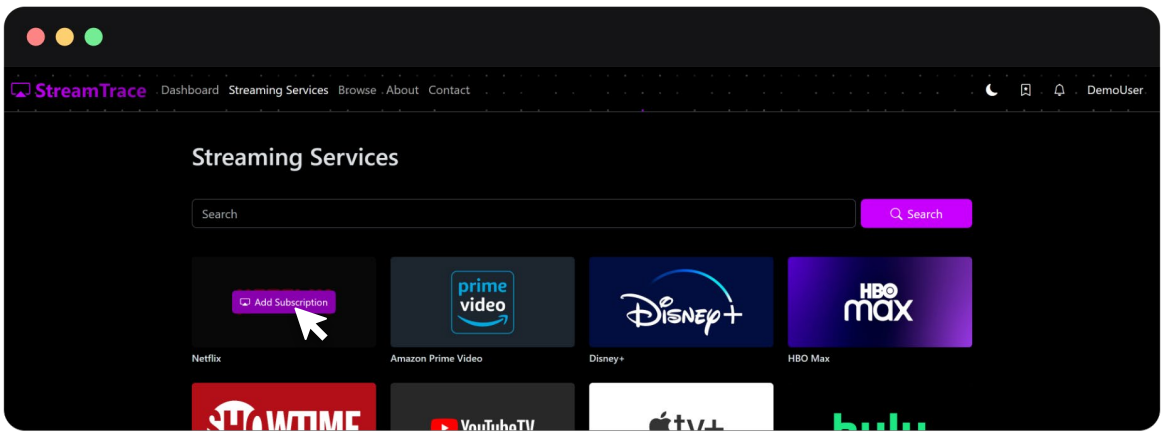
Here, you can simply click on **Add new service** which will open the list of all available streaming service providers.

If you wish to add another subscription, click on the button with plus symbol right to the My Subscriptions heading.

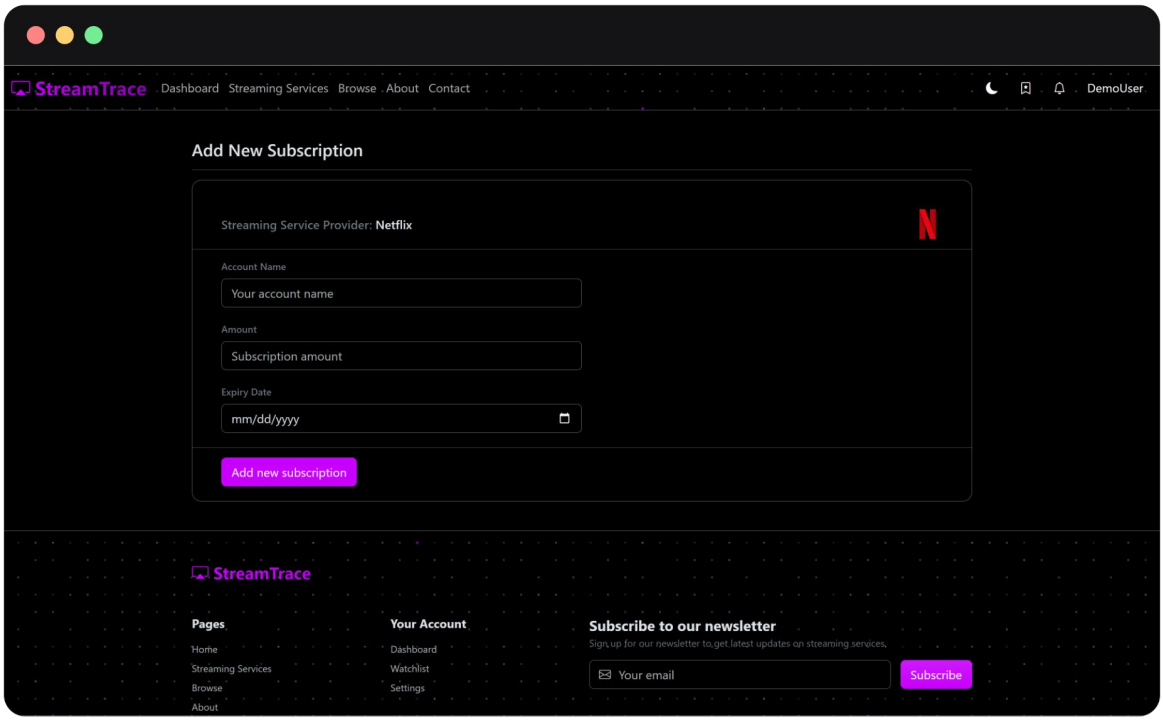




From the list of **Streaming Services**, you may select your streaming service provide by clicking on the streaming service card.



This will take you to another where you can fill in details about your subscription.



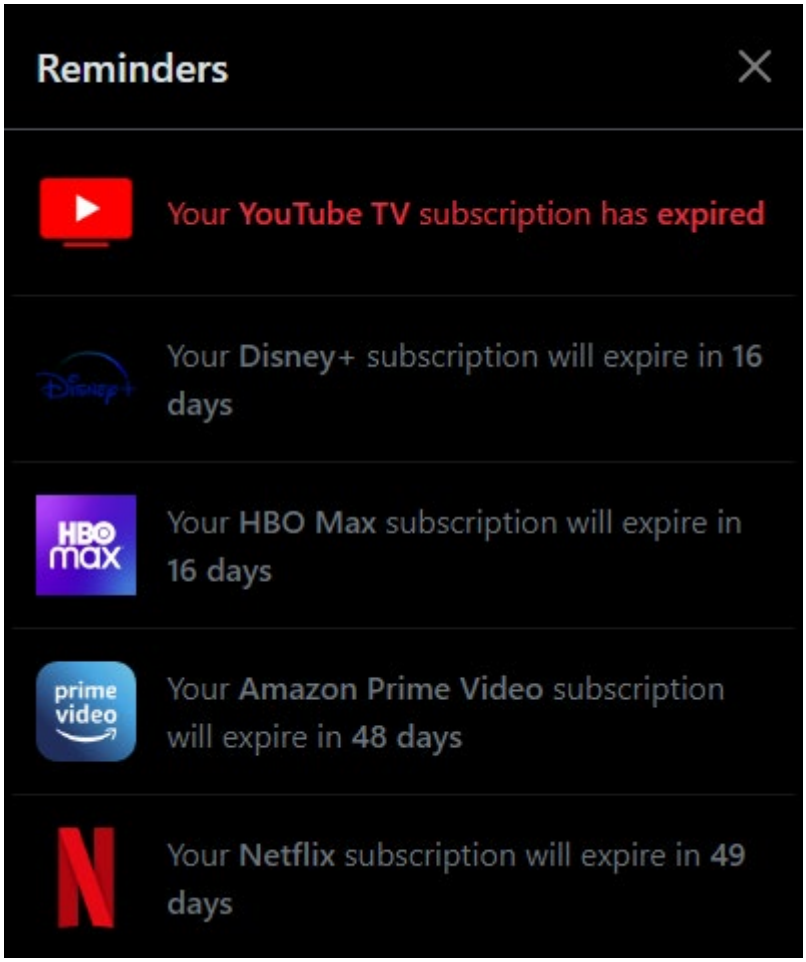
Here, add your subscription details like your **account name**, your subscription cost **amount**, and the **last date** of your subscription service.

Once done, click on **Add new subscription** to add it your StreamTrace account which you can later track and get reminders.

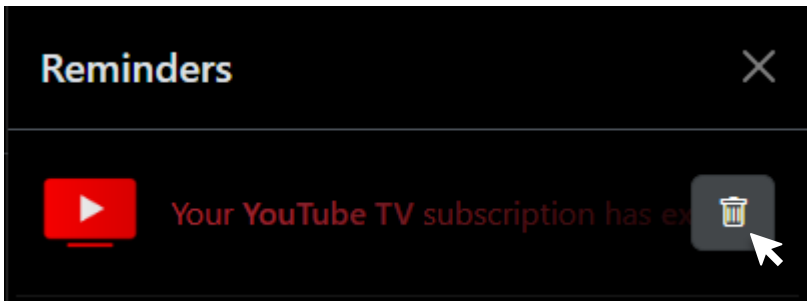
5. Reminders

To see reminders, click on the bell icon on top-right of the screen.

This will open the Reminders tab that will show you long before your subscriptions expire.



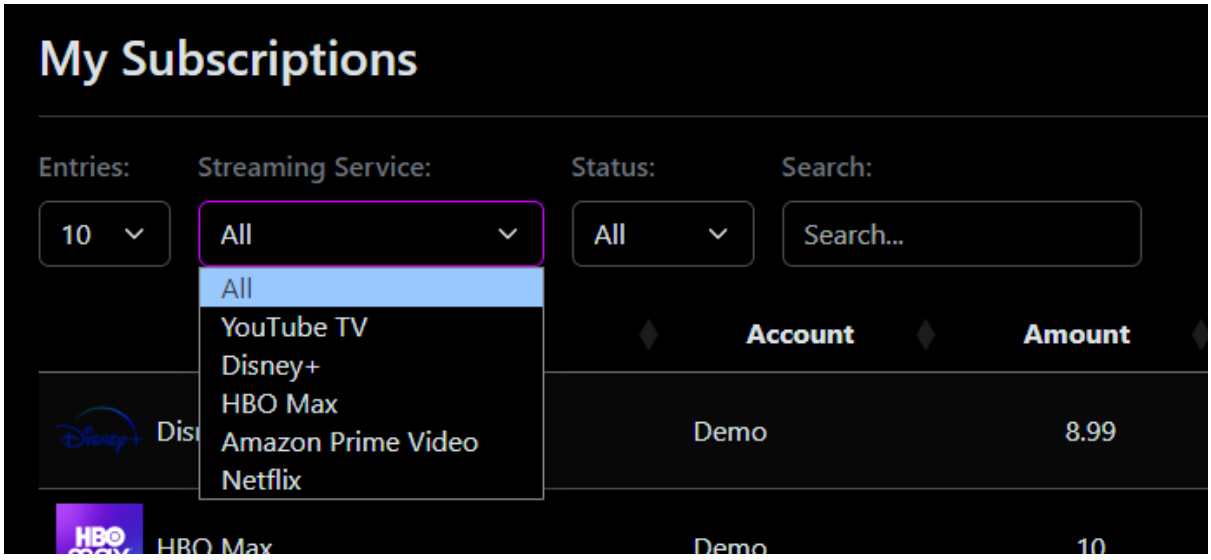
If you subscription service has expired, you can choose to archive it from your list of reminders.



This will hide the reminder for that particular subscription.

6. Search, Filter & Sort Subscriptions

You can search, filter, and/or sort your subscriptions from your dashboard.

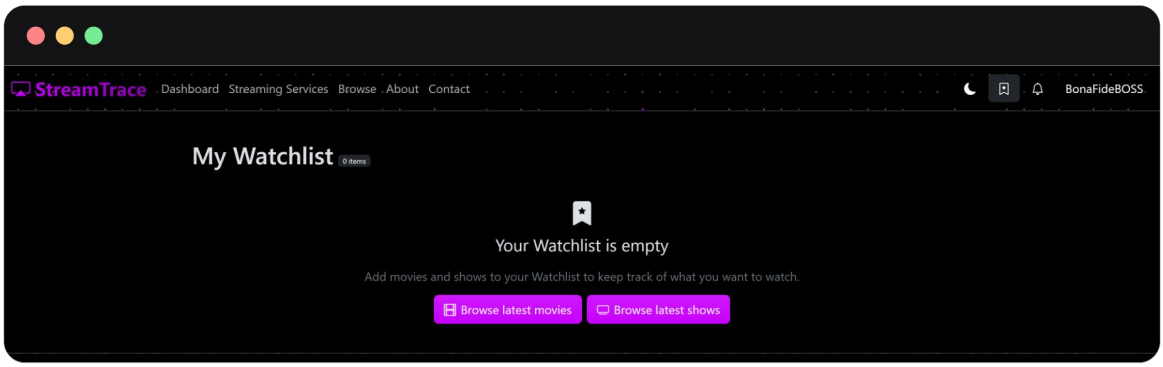


7. User Watchlist

You can add movies and TV shows to your Watchlist to keep track of what you want to watch.

To go to your watchlist page, click on the bookmark icon on top-right corner of desktop screen.

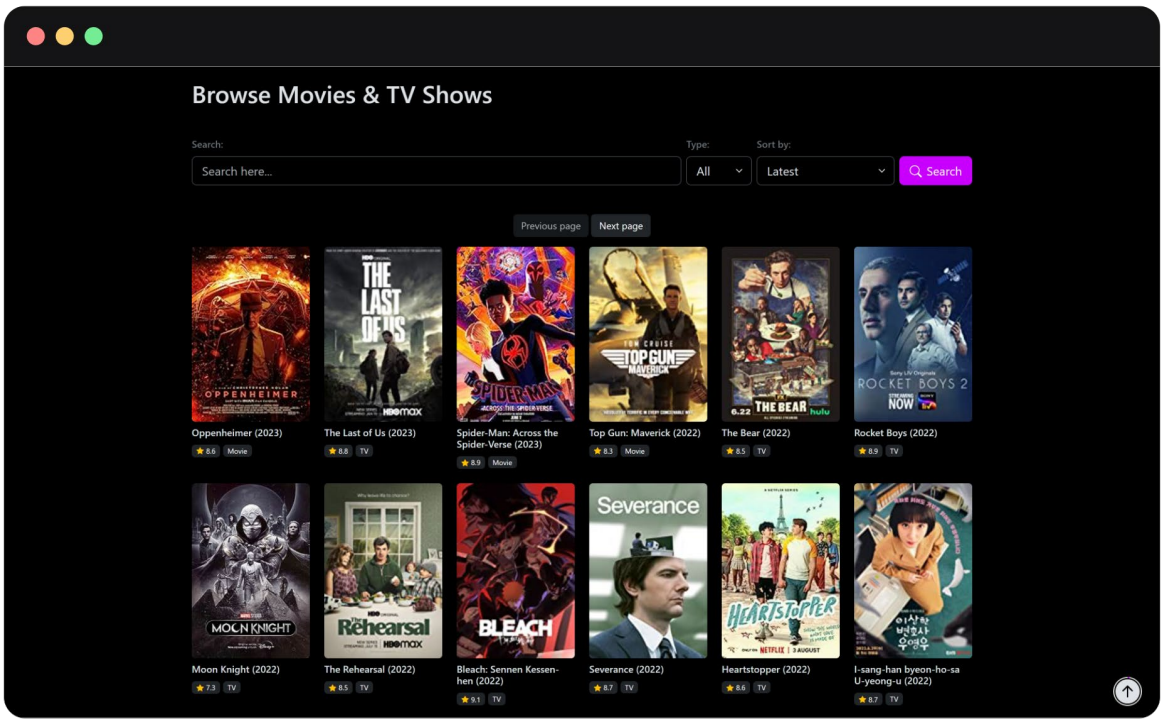
If you're watchlist is empty, you should see the below screen.



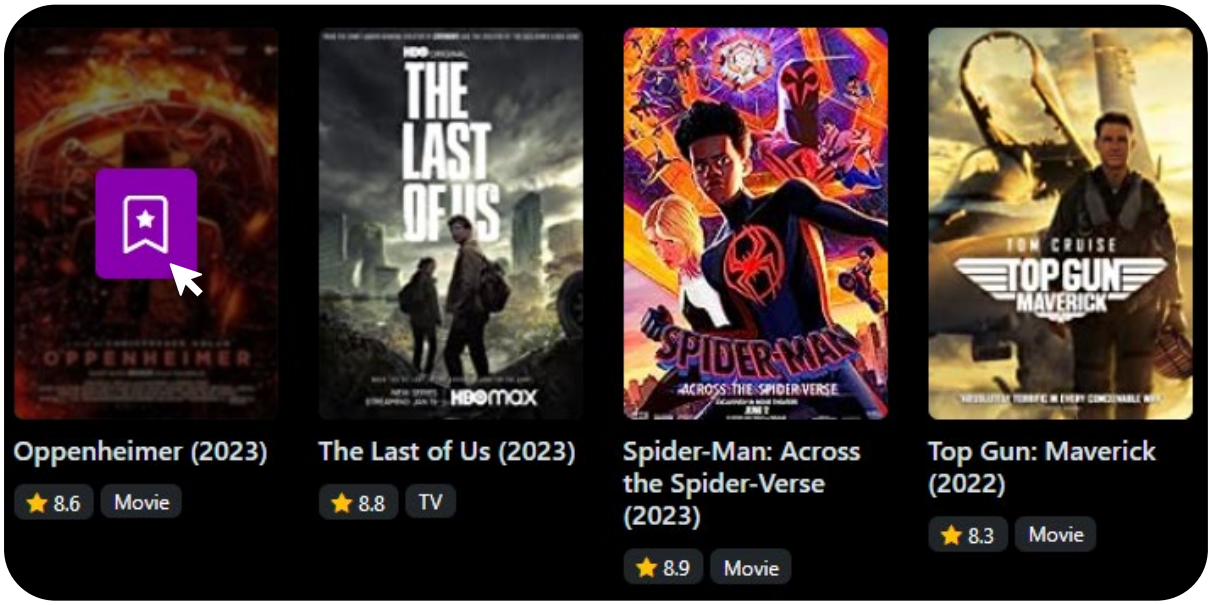
This will provide you with links to browse latest movies and TV shows.

You can also browse all movies and shows by going to the **Browse** page.

7.1. Browse Movies and TV Shows

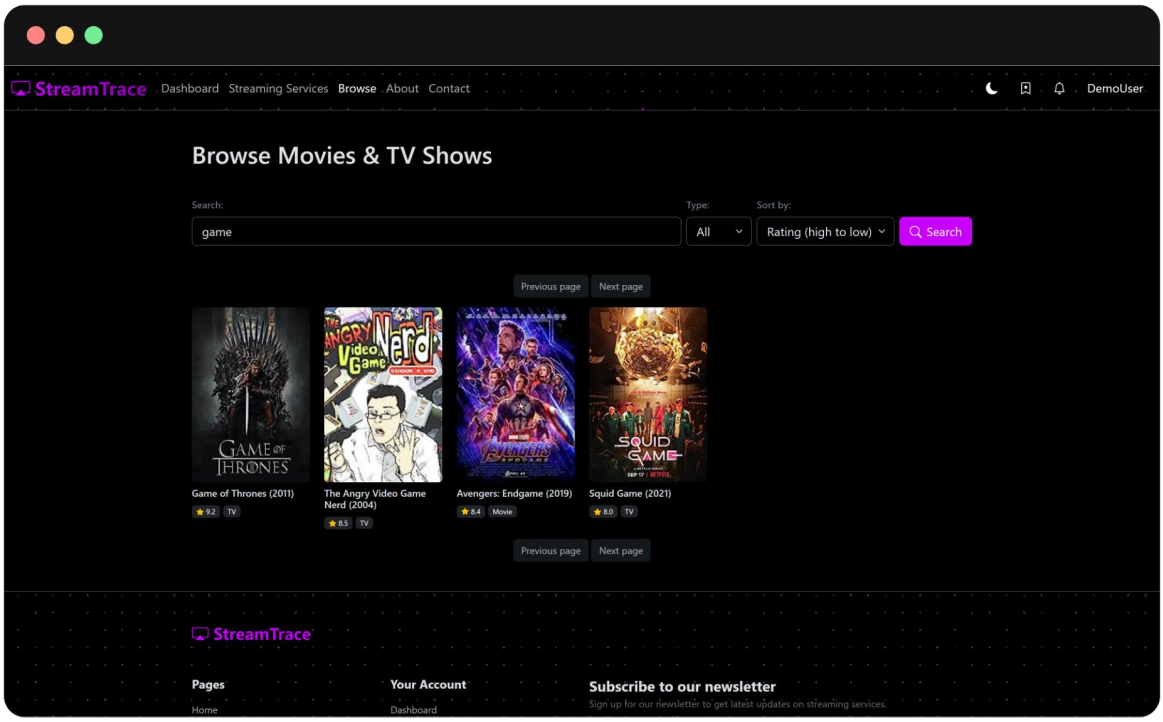


This page offers you many movies and shows to choose from for you to watch on your streaming service platform. Click on an item to add it to your watchlist.



7.2. Search, Filter & Sort Movies and TV Shows

You can search, sort and/or filter movies and shows from your watchlist or the browse page.

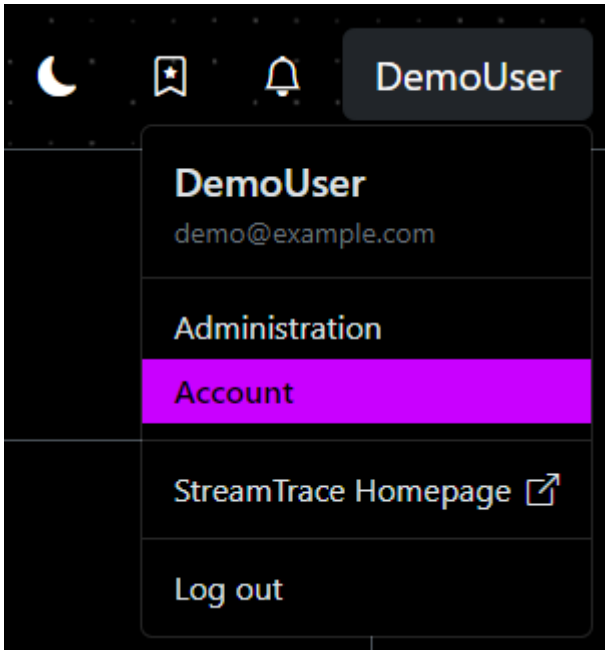


This allows to easily find your favorite movie or show.

8. User Account

If you wish to configure your account settings, head over to the account page where you will find various options to manage your account details.

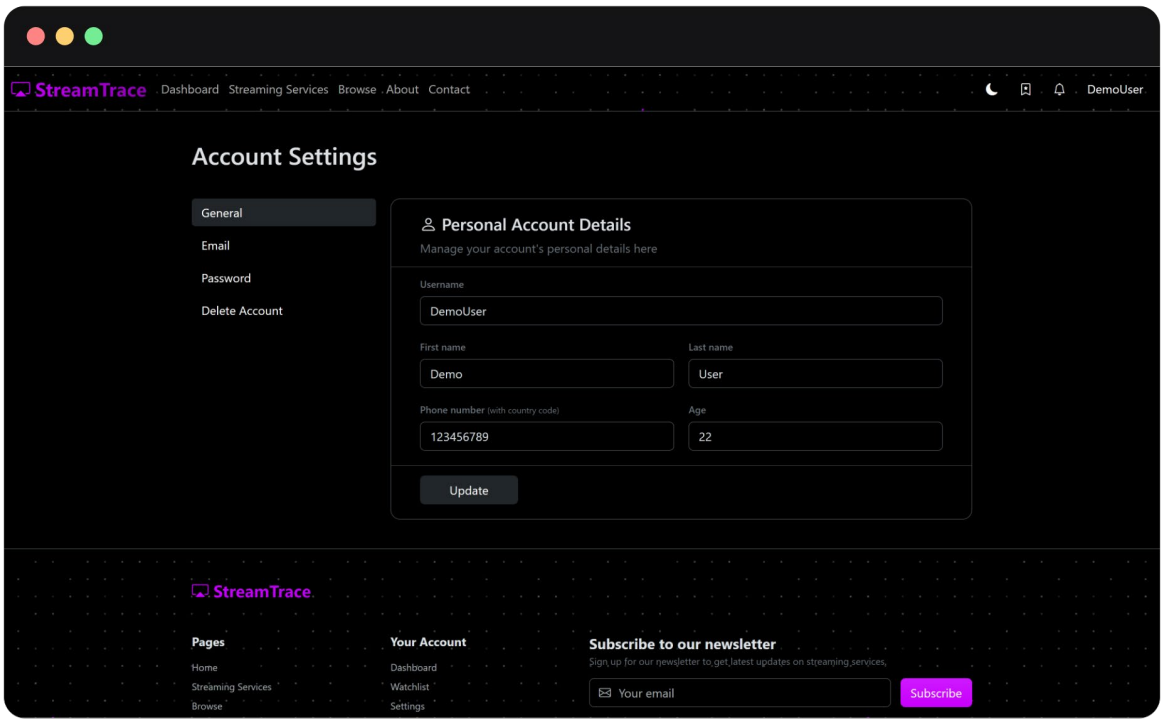
To open your account page, click on your username of top-right corner of the screen, then click on **Account**.



This will open your account page where can:

1. Update Personal Account Details
2. Update your email address
3. Update your password
4. Delete your account

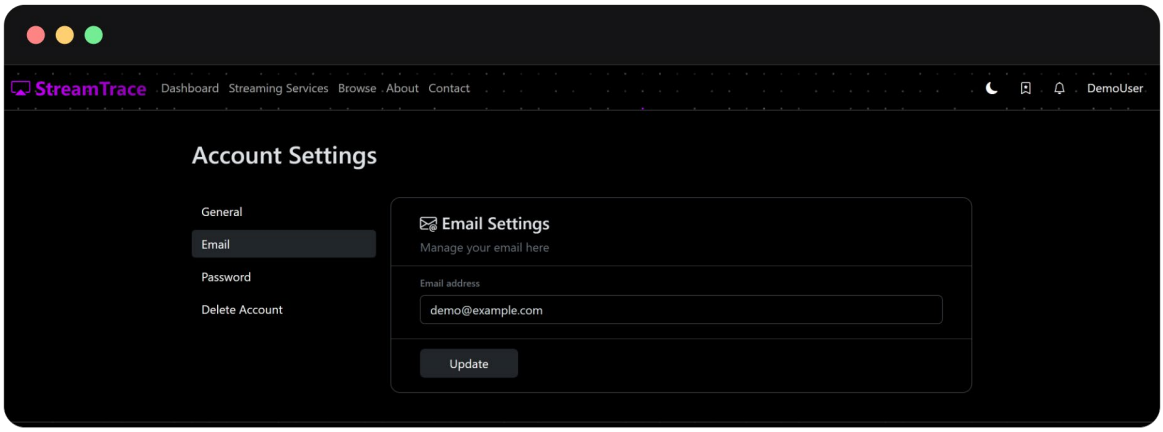
8.1. Update Personal Account Details



On this page, you update your **username** along with your personal details like **first and last name**, your **phone number**, if we need to contact you, and **your age** that can help us recommend you more personalized content.

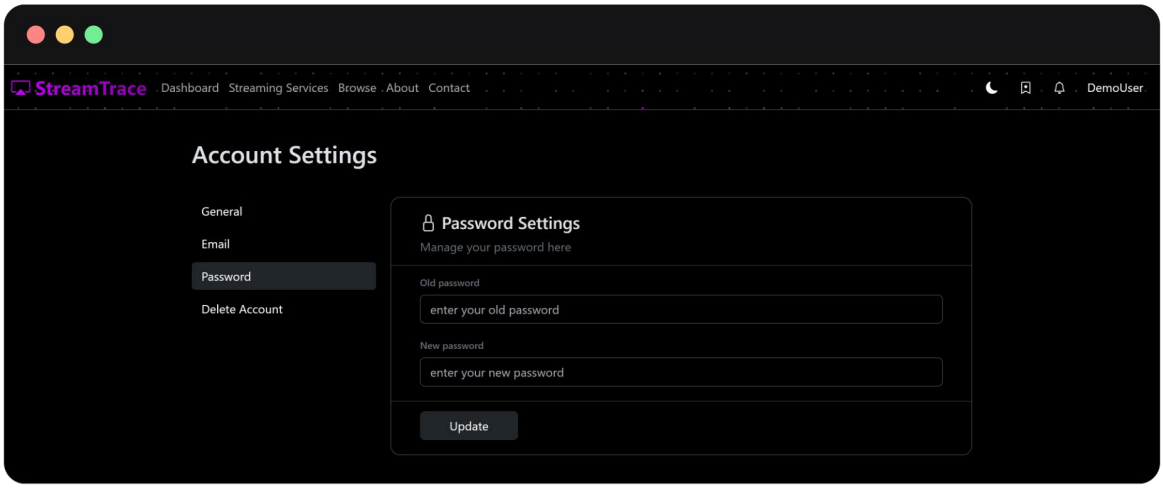
8.2. Update Email Address

If you wish to change your email address, this is where you do it.



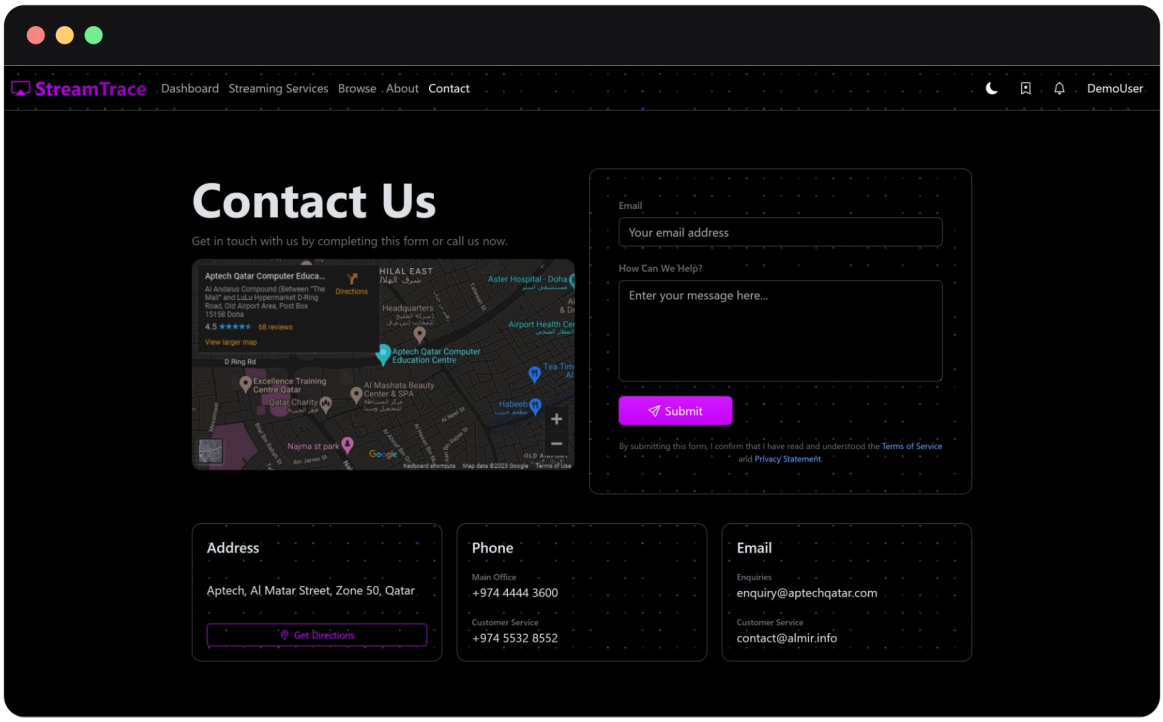
8.3. Update Password

If you wish to update your log in password, you change it on this tab.



9. Contact Page

If you wish to contact us for support or send **feedback** about, please feel free to use the contact page to get in touch with us or drop feedback about our website.

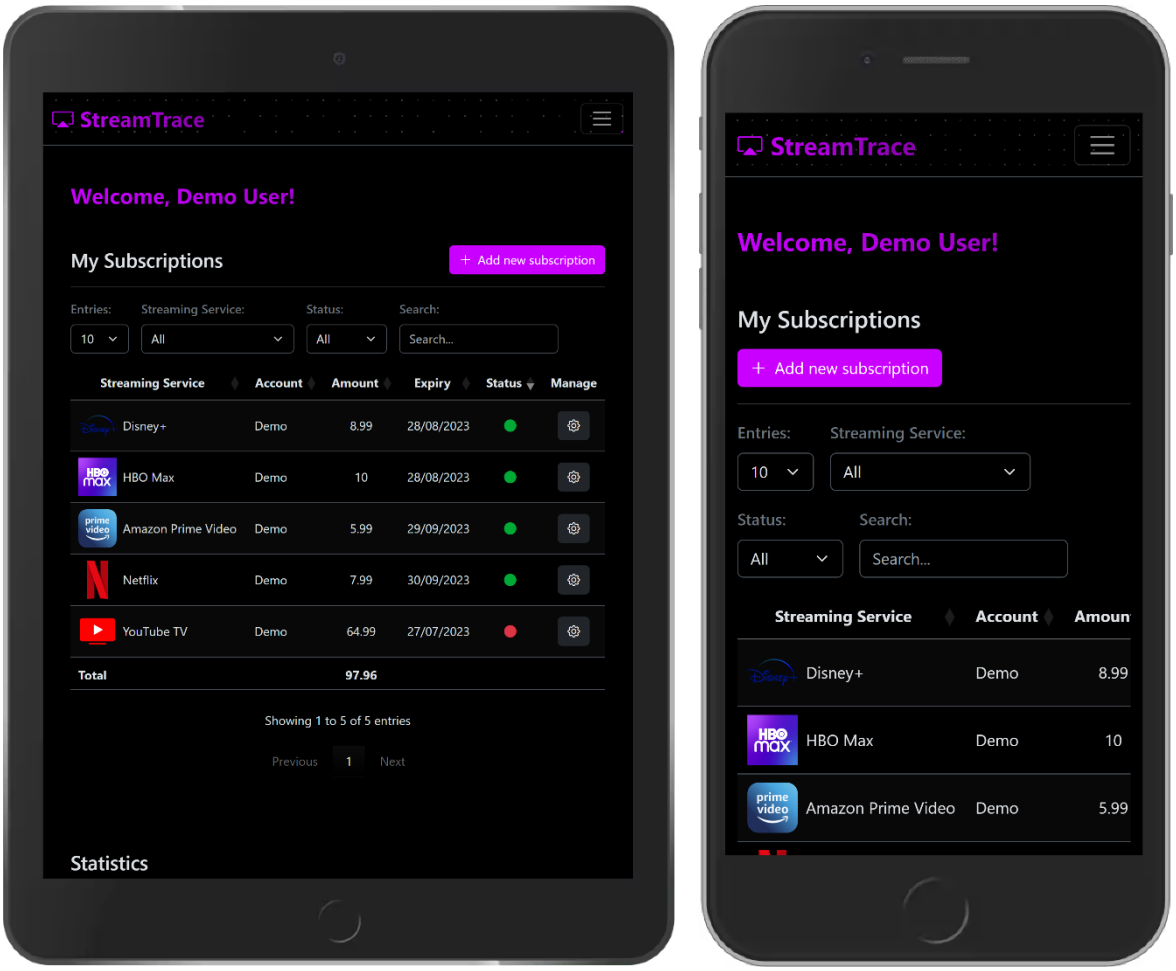


On this page, enter your email address that we'll use to contact you back if needed and your message for us.

Users can also reach out to us using the **contact details** provided on this page.

10. Mobile and Small Screen Compatibility

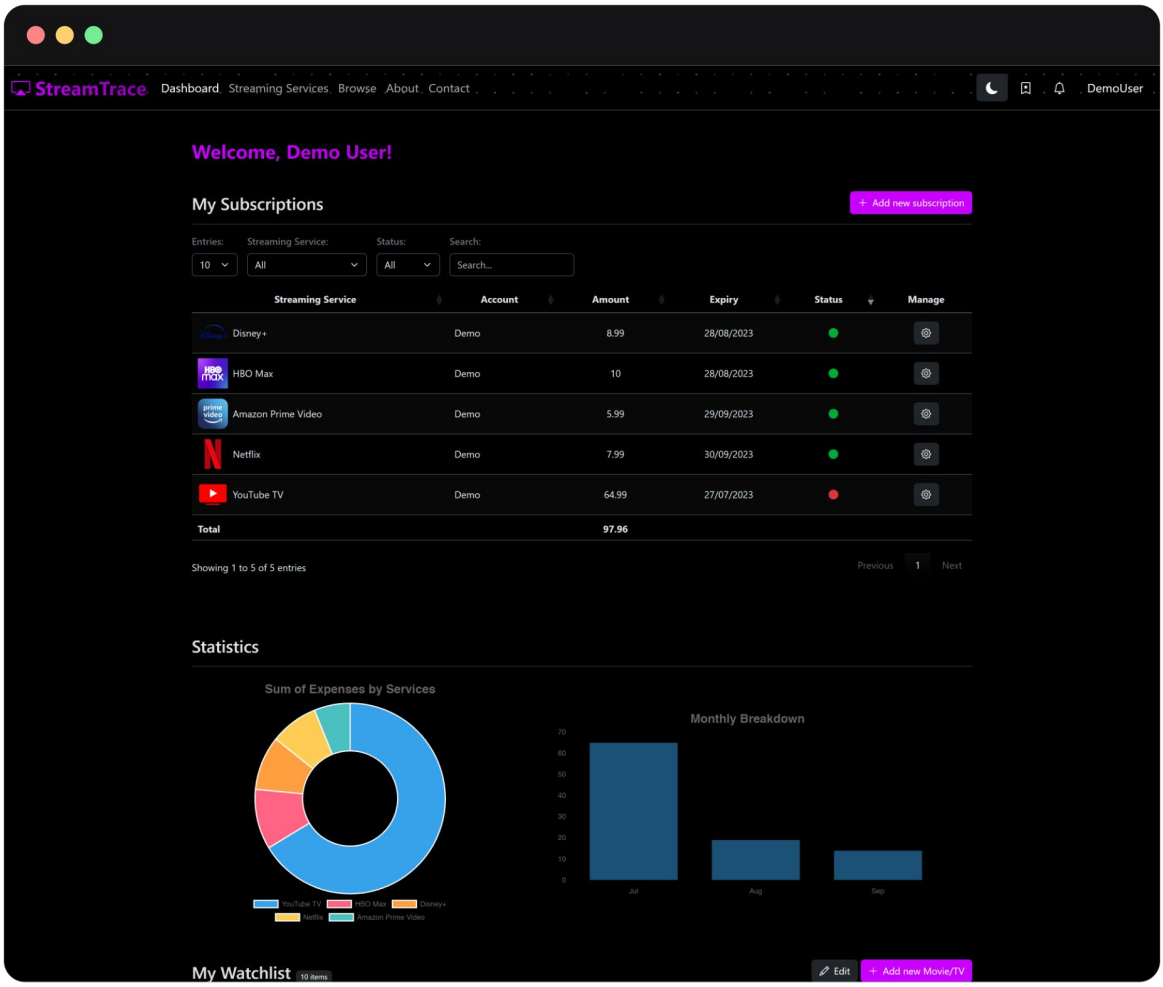
All the features discussed above will also work smoothly on small screen devices like tables and mobiles.



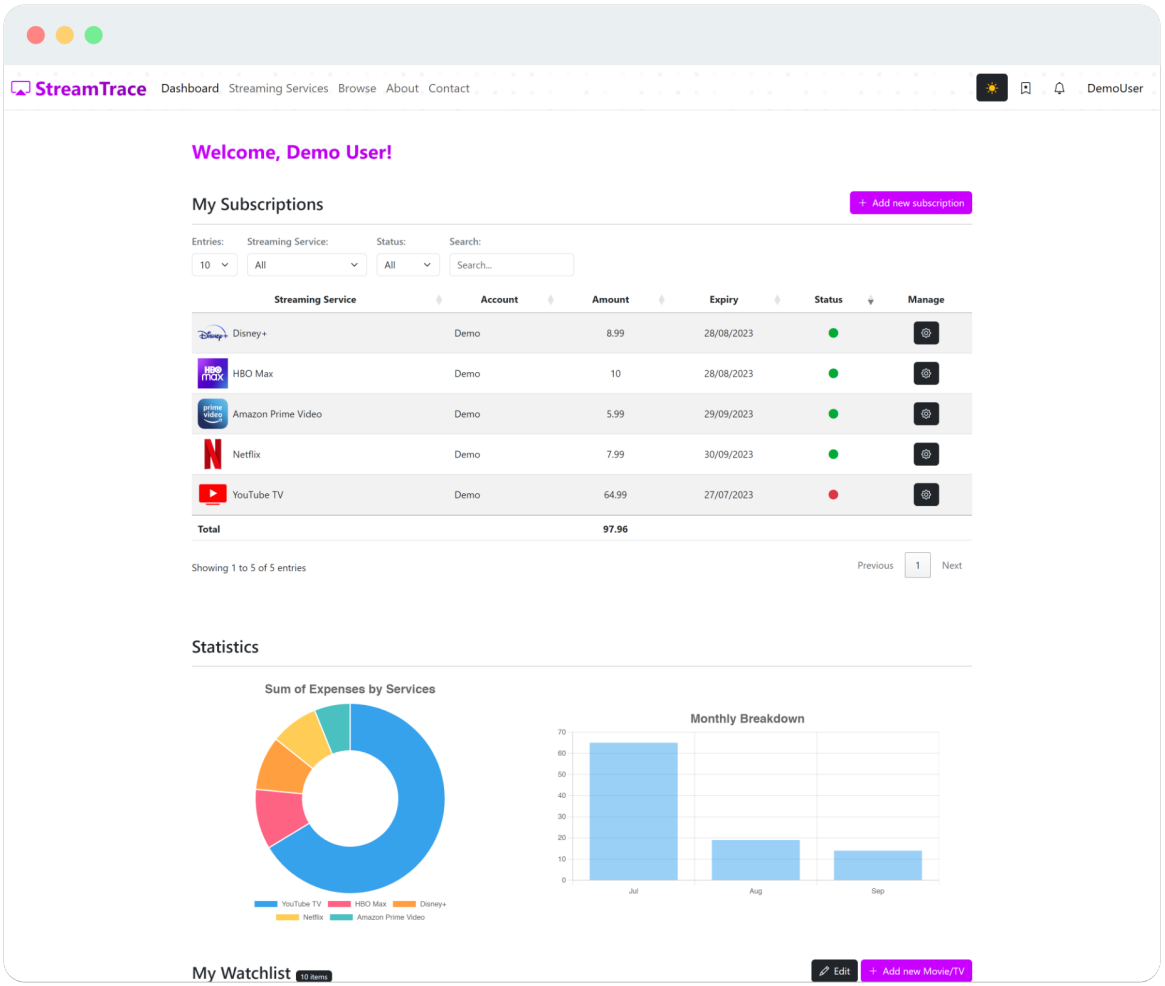
11. Website Theme

The default theme of the website is dark mode but did you know you can also use the light mode in case you're outside and want things to be bright.

11.1. Dark Mode



11.2. Light Mode



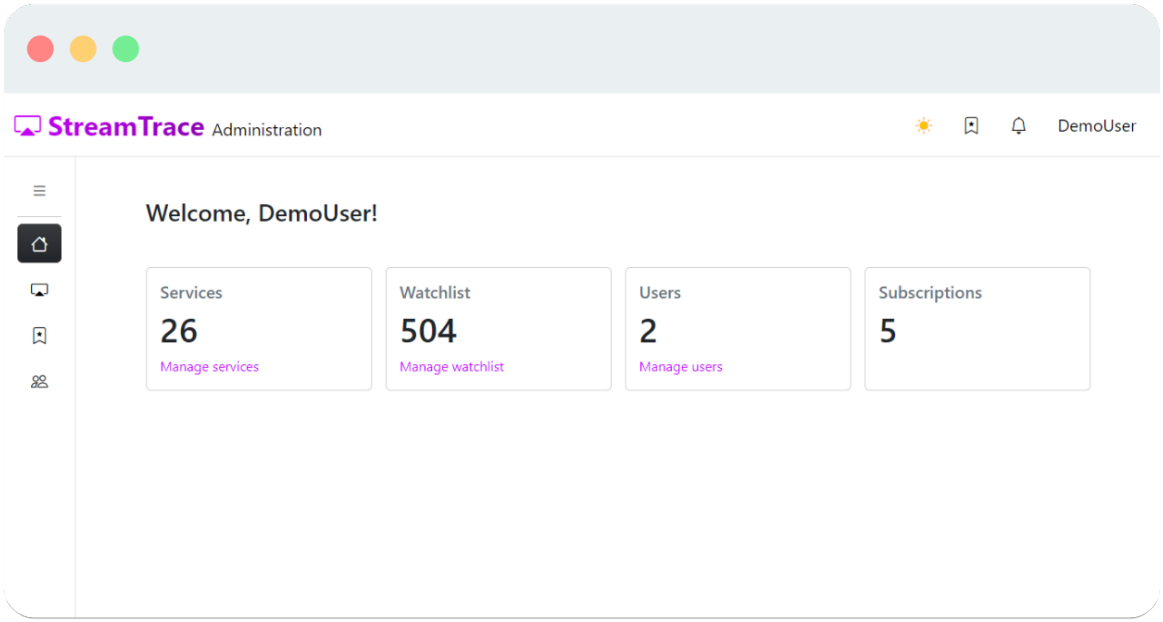
You can change the website theme by clicking on the Moon/Sun icon on the top-right corner of the screen.

Admin User Manual

This guide is for the administrators of the website.

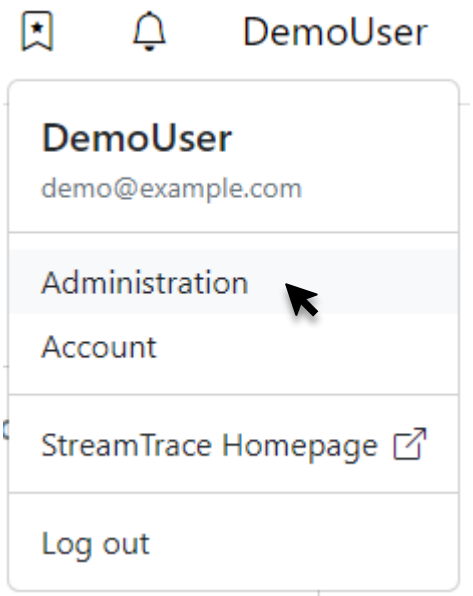
The Demo Account provided earlier on this document has enough permission to view the admin site and create and add new data. But it cannot edit or delete any data.

1. Dashboard



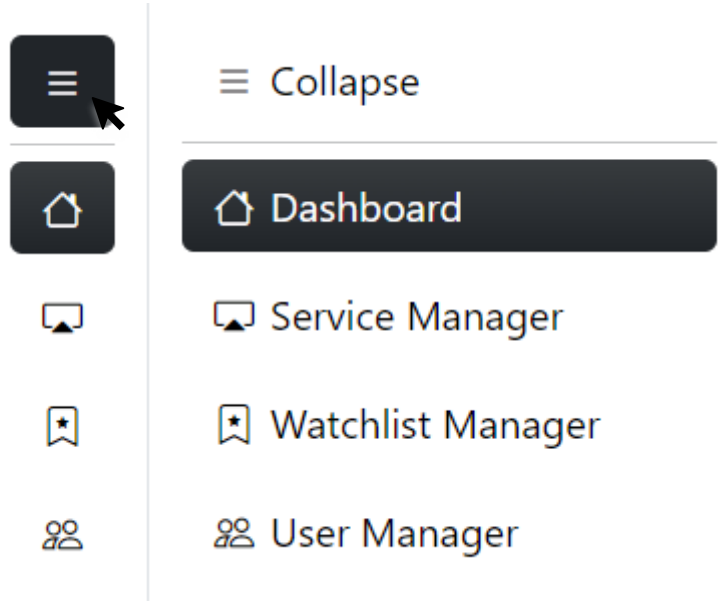
The dashboard display count of records in each database collection. It also provides links to manage different collections.

To open the Administrator dashboard, click on Administration from navbar menu.

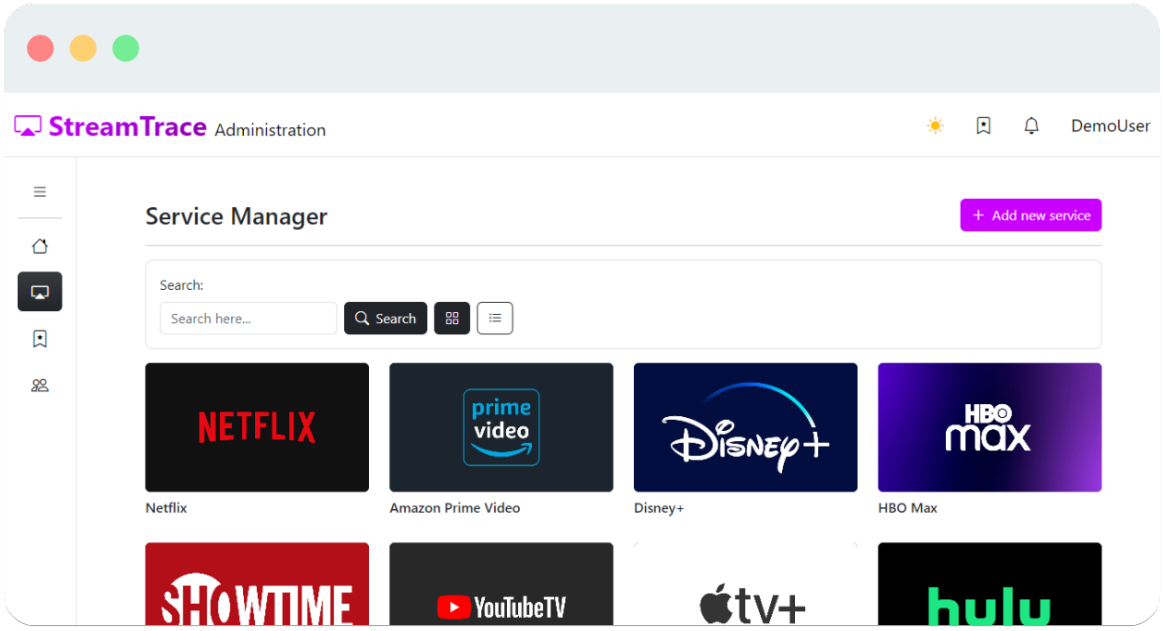


The administration page includes a sidebar navigation that allows to navigate through different administrative pages.

Click on the Collapse button to expand the sidebar.



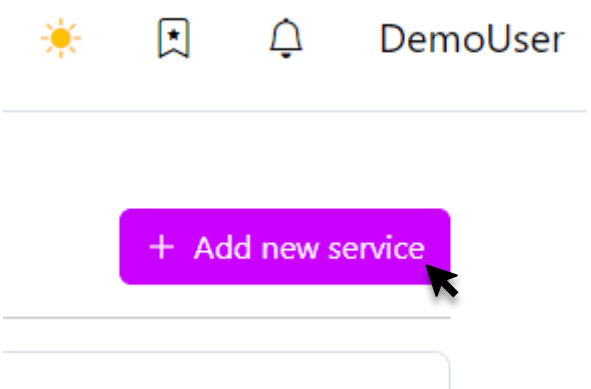
2. Service Manager



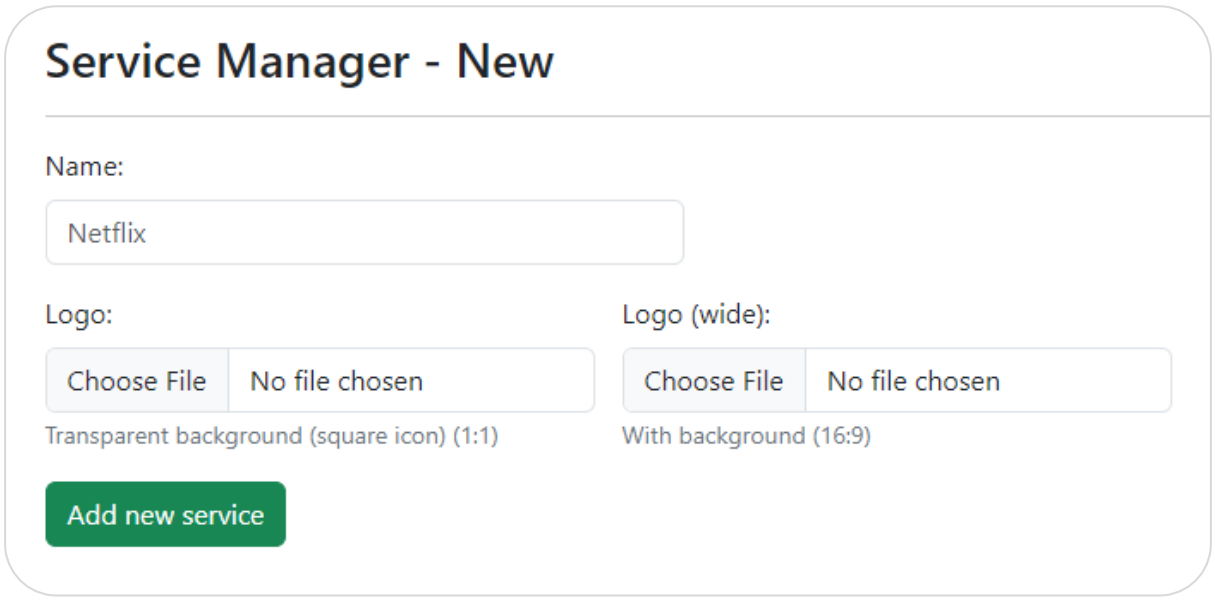
Admins can **create**, **edit**, and **delete** streaming service providers from this page.

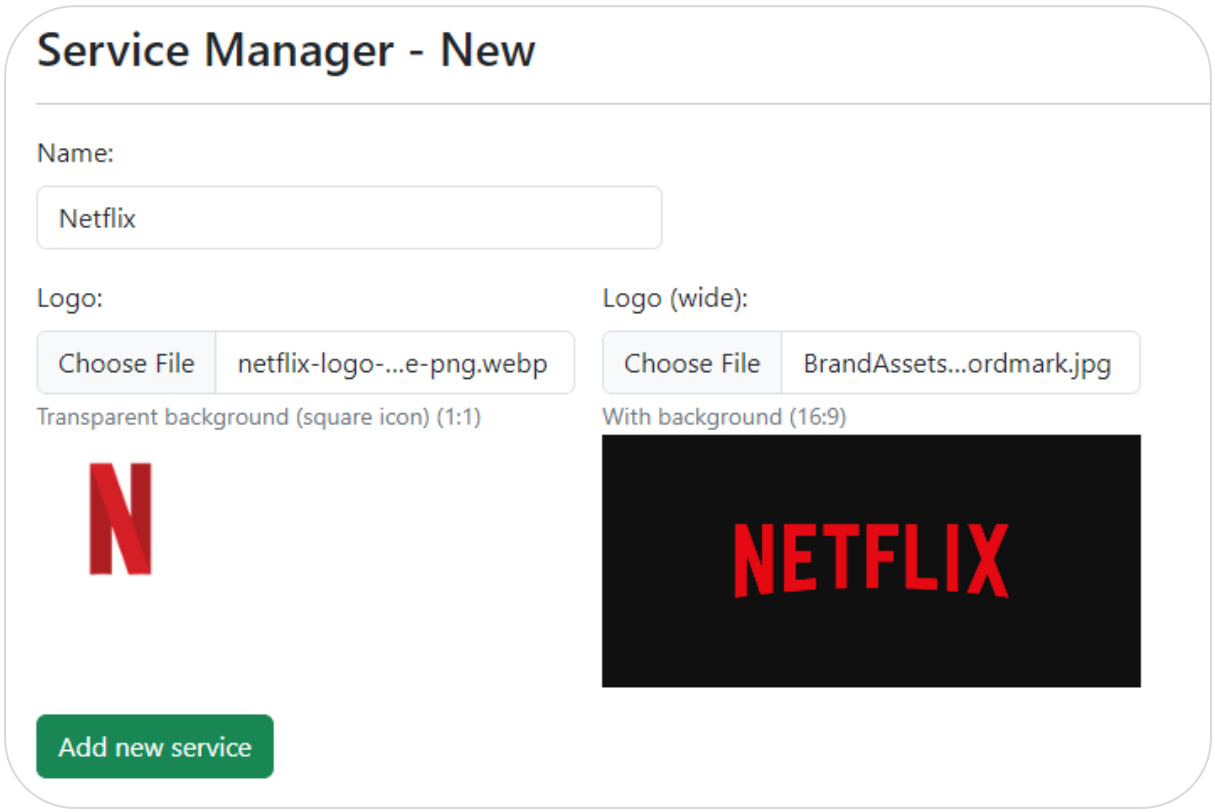
2.1. Add New Service

Click on the button with plus symbol to add a new streaming service provider to the database.



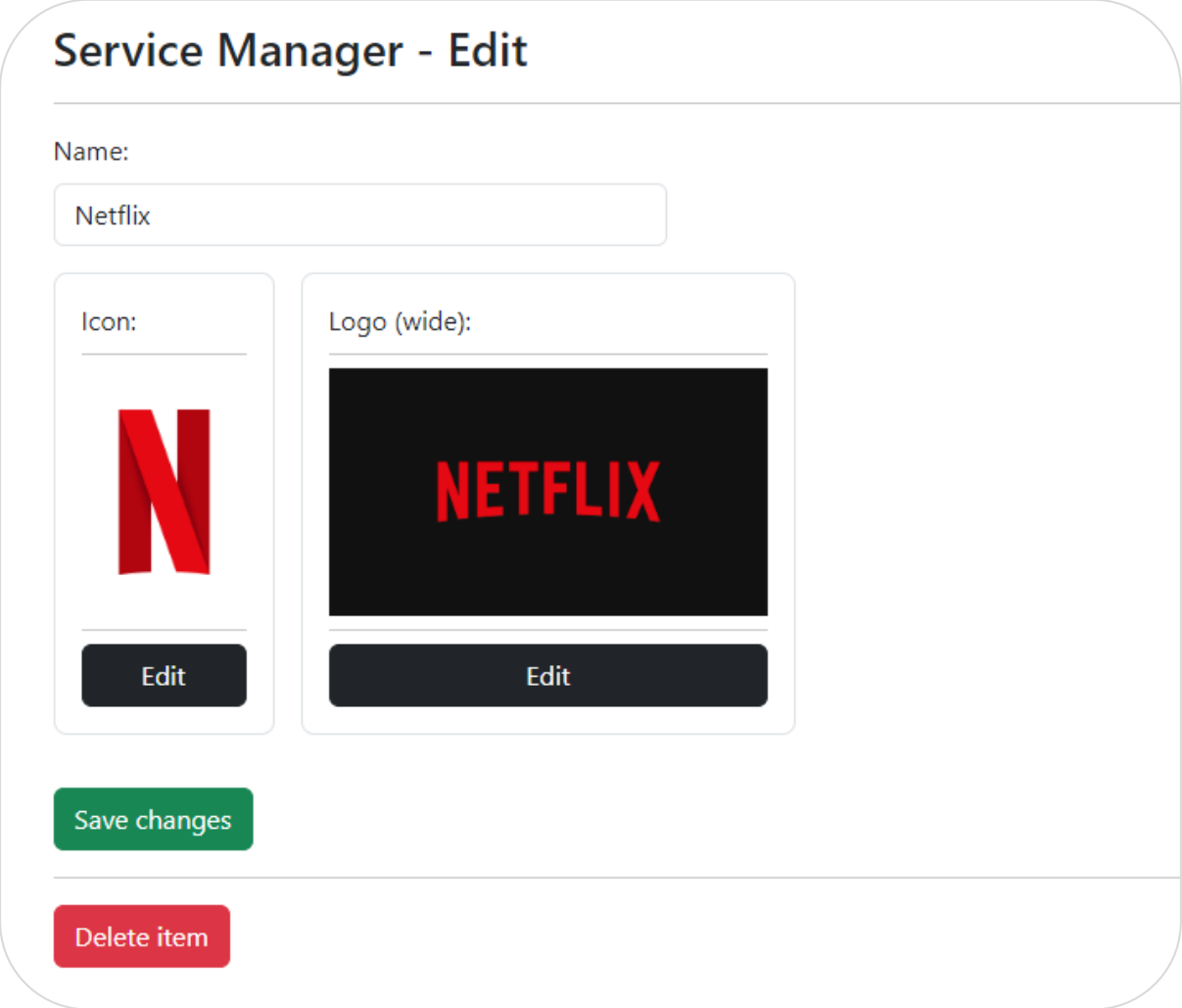
On this page, you can insert name of the service provider and their logo.





2.2. Edit or Delete a Service Provider

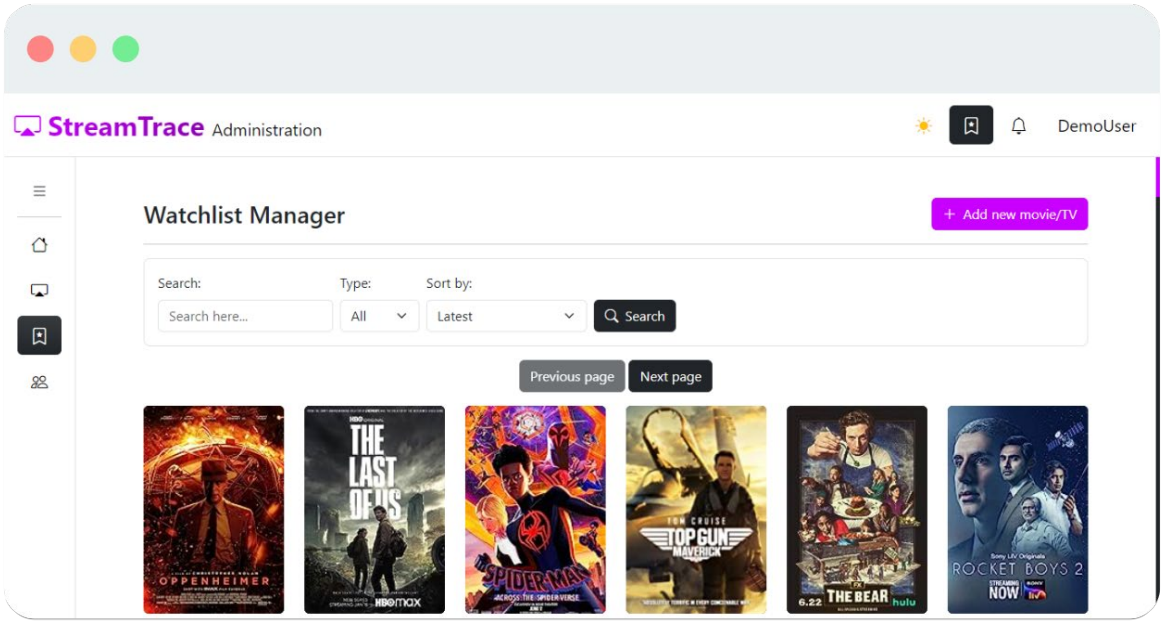
To edit or delete a streaming service provider, click on any service to go to their edit page where you can perform these actions.



Note: The demo account cannot perform these actions as it does not have enough permissions to do so.

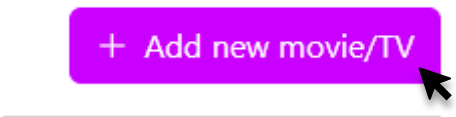
3. Watchlist Manager

This page lets you add movies and TV shows to the website for users to add them to their watchlist.

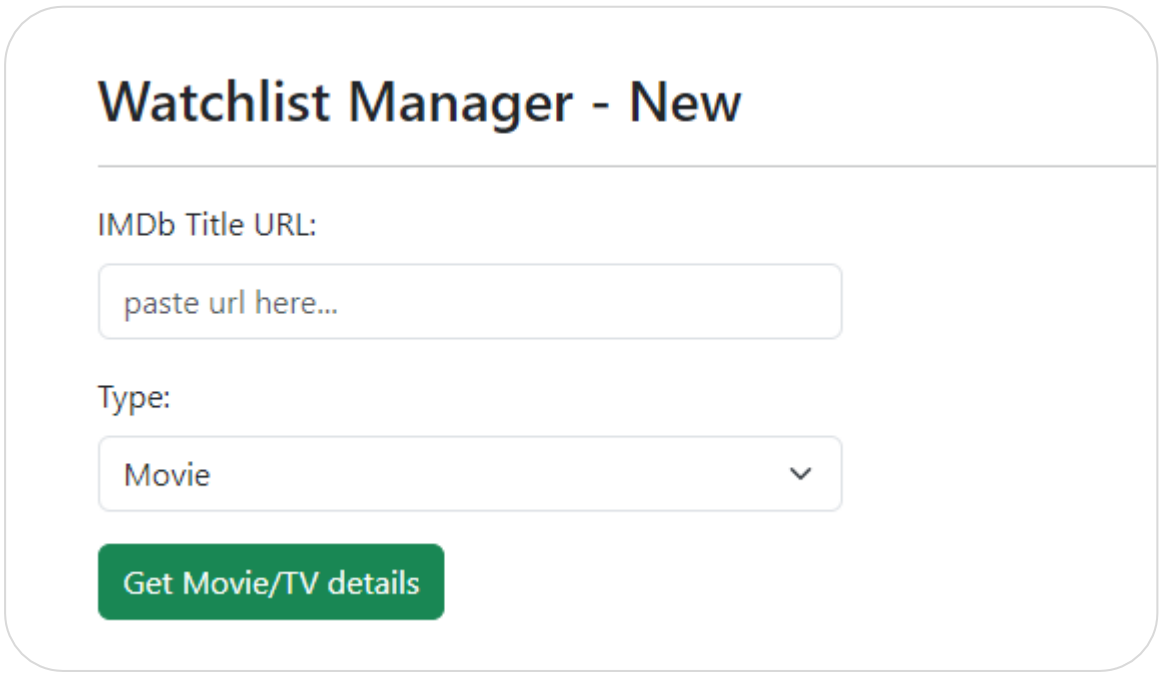


Here also admins can perform CRUD functions on the Watchlist collection.

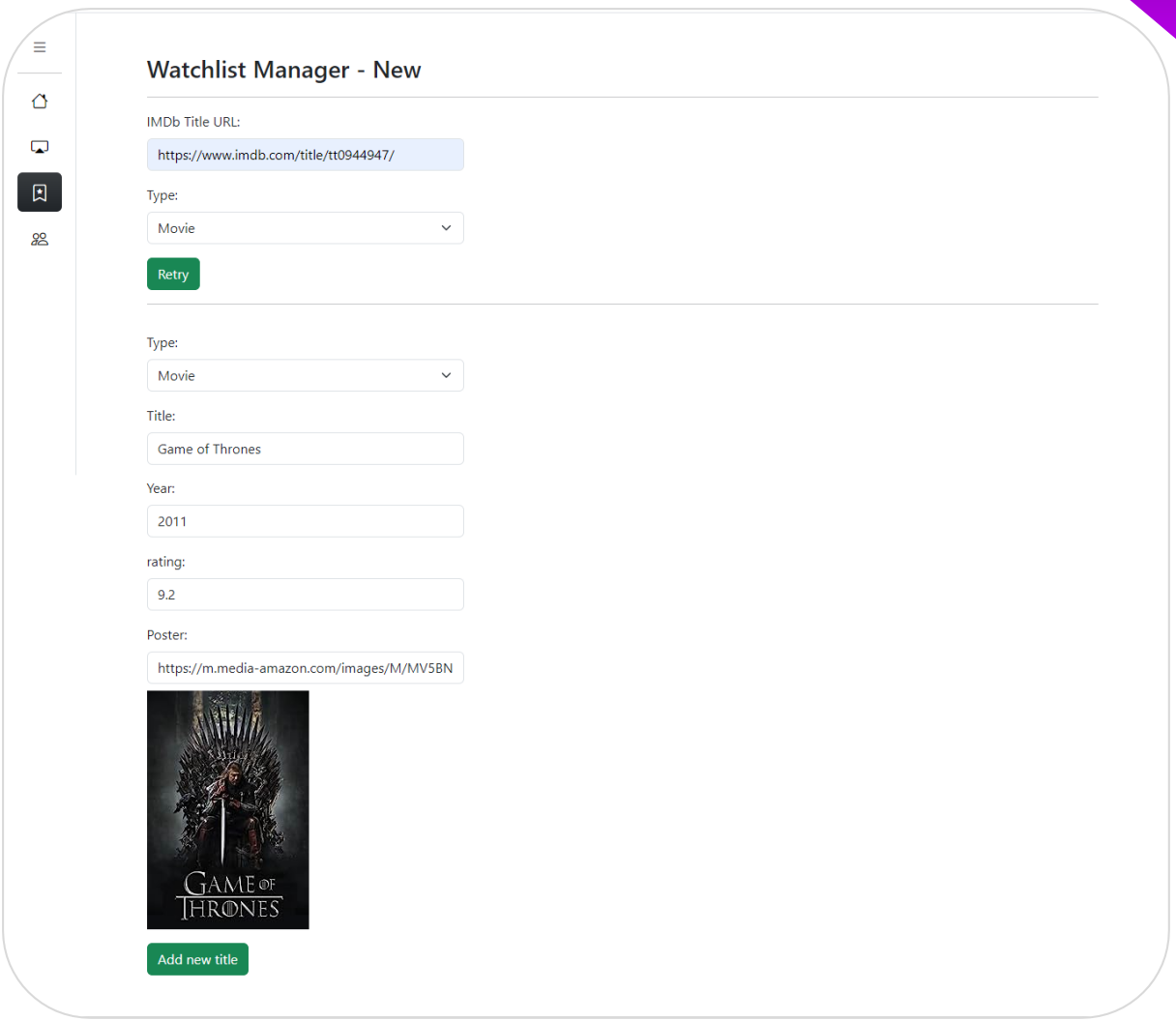
To Add a new movie or TV, click on the plus icon as shown below.



3.1. Add New Movie or TV Show



On this page, you can get a **movie/tv show details** like its **rating and poster** from **IMDb title page**. Simply, paste a link to IMDb title page and click on the get details button.



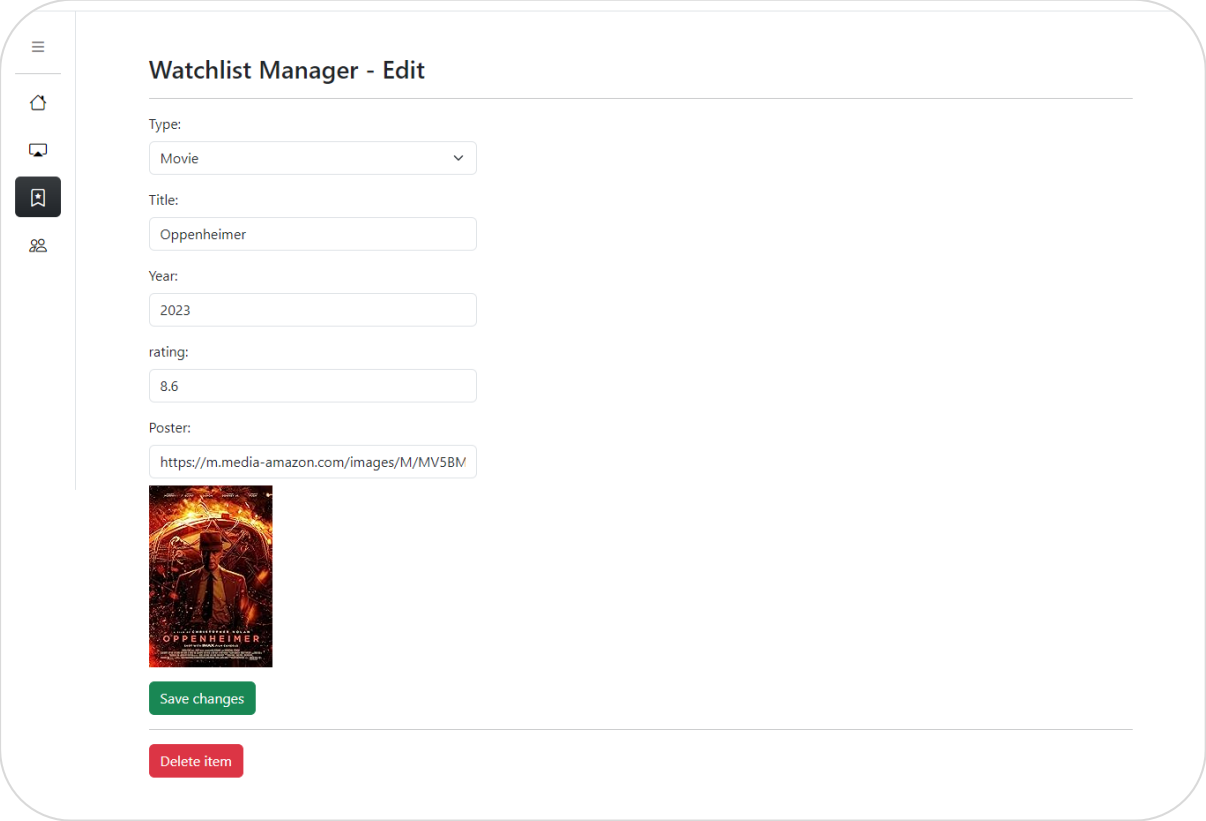
This will get all the necessary details like movie/TV title, year, rating, and poster URL.

Important: You need to set the type of the title (Movie/TV) manually.

Note: The demo account can perform this action.

3.2. Edit or Delete a Movie or TV Show

To edit or delete a movie or TV show from the database, click on the title you want edit or delete.

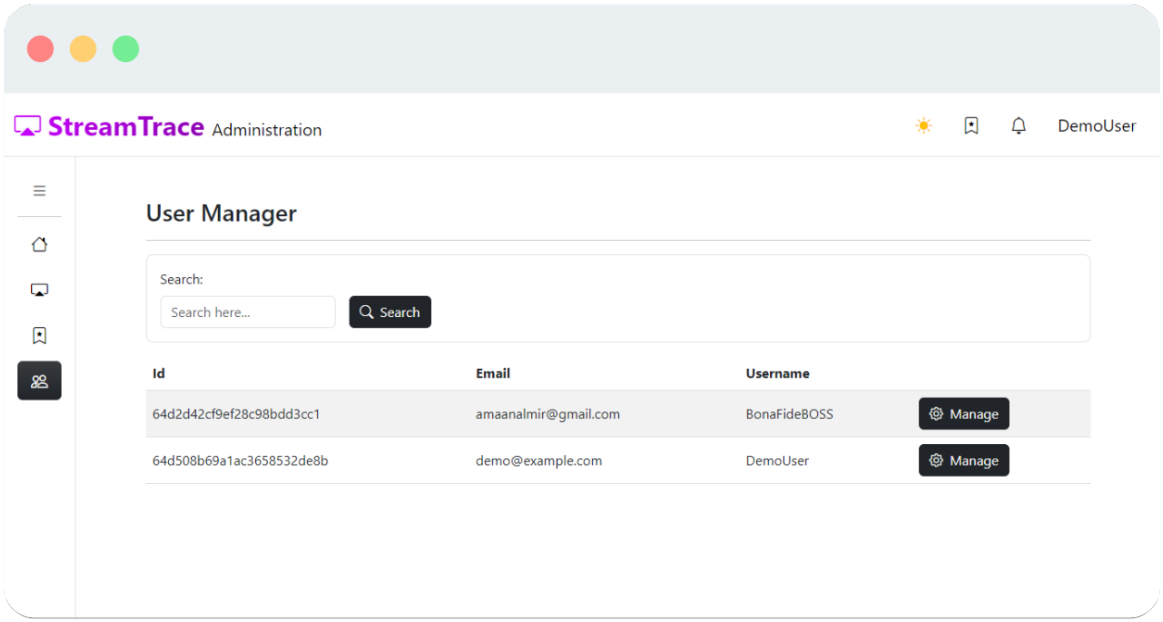


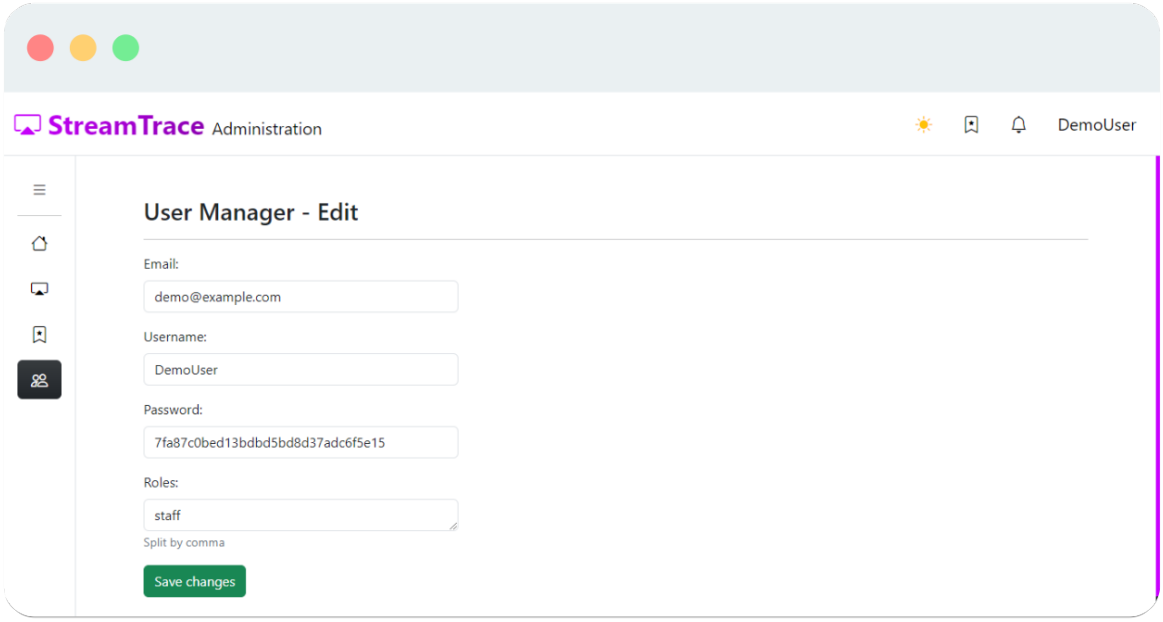
Here, admins can edit the title to update its details or permanently remove it from the database.

Note: The demo account cannot perform this action.

4. User Manager

To **update a user details**, go the user manager page and click on the manage button next to the user you wish to edit and make the changes. **Note:** The demo account does not have permission to edit a user details.





References

Otto, M. & Thornton, J., 2023. *Bootstrap 5.3 Documentation.* [Online]
Available at: https://getbootstrap.com/docs/5.3/

Ronancher, A., 2023. *Flask 2.3.x Documentation.* [Online]
Available at: https://flask.palletsprojects.com/