Outline

Shape Runner is a health and fitness web application that allows users to design, generate, and save custom running routes in creative shapes. The app combines interactive map drawing with automated route generation, enabling users to click points on a map to outline a shape, and then automatically snap that shape onto real-world roads using OpenRouteService. Users can generate routes, view turn-by-turn instructions, save them to their personal profile, and search previously created routes.

In addition to route generation, the application supports fitness tracking through a run logging system. Users can record completed runs with details such as date, duration, rating, and notes, and search their history using filters.

The system includes secure account creation and authentication, input validation and sanitisation, and persistent storage using a MySQL database. The user interface is built using EJS templates and styled with a clean responsive layout. Altogether, Shape Runner provides a practical fitness tool that blends creativity, mapping, and health tracking.

Architecture

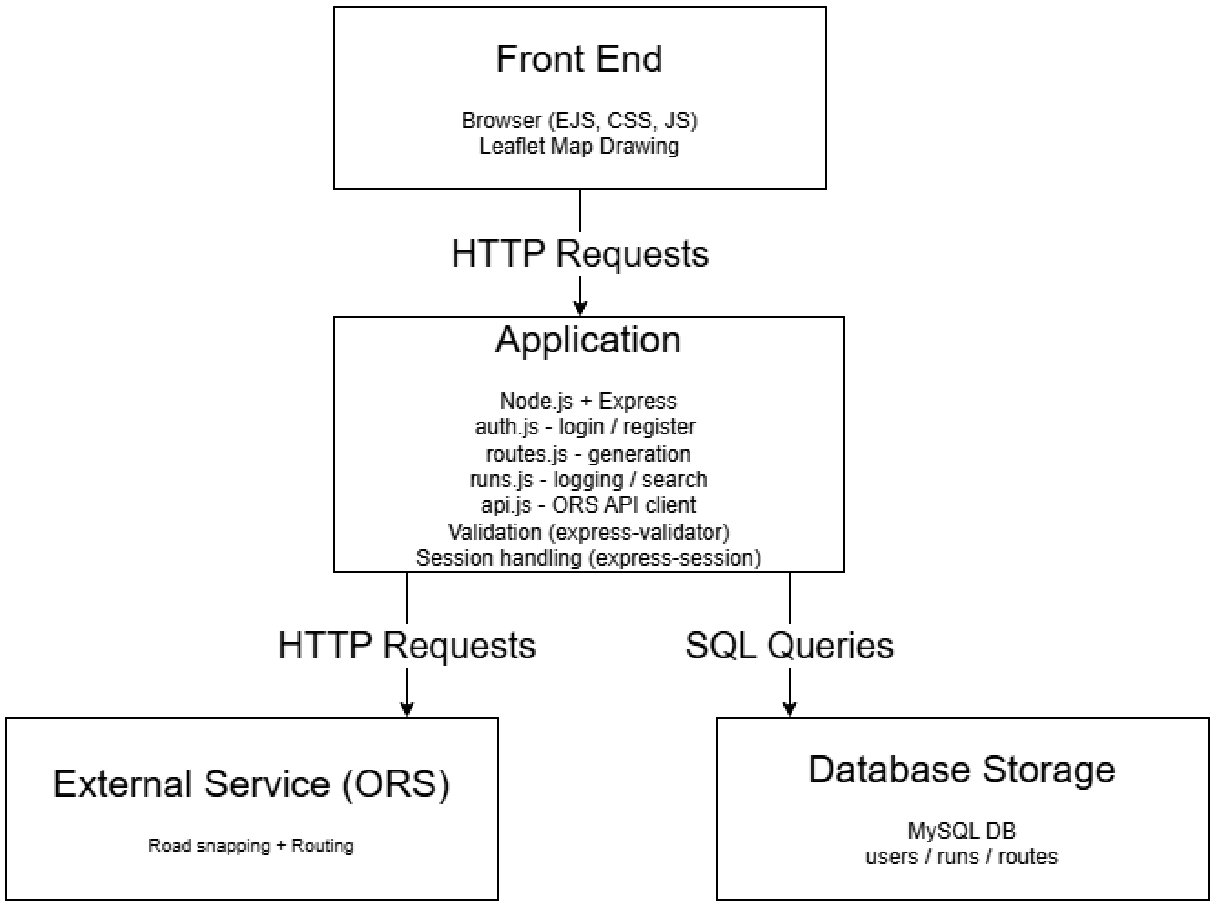


Figure 1 - Three-tier architecture showing front-end, application tier, and data/external service integrations.

Shape Runner uses a three-tier architecture. The presentation tier runs in the browser and consists of EJS-rendered pages styled with a shared CSS file, along with Leaflet for map drawing and interaction. The application tier is a Node.js Express server that handles routing, authentication, validation, session management, and calls to the OpenRouteService API for route snapping. It includes modular controllers such as auth.js, routes.js, runs.js, and api.js. The data tier is a MySQL database that stores users, generated routes, and logged runs. Communication between tiers occurs via HTTP requests and parameterised SQL queries, ensuring security and separation of concerns.

Data Model

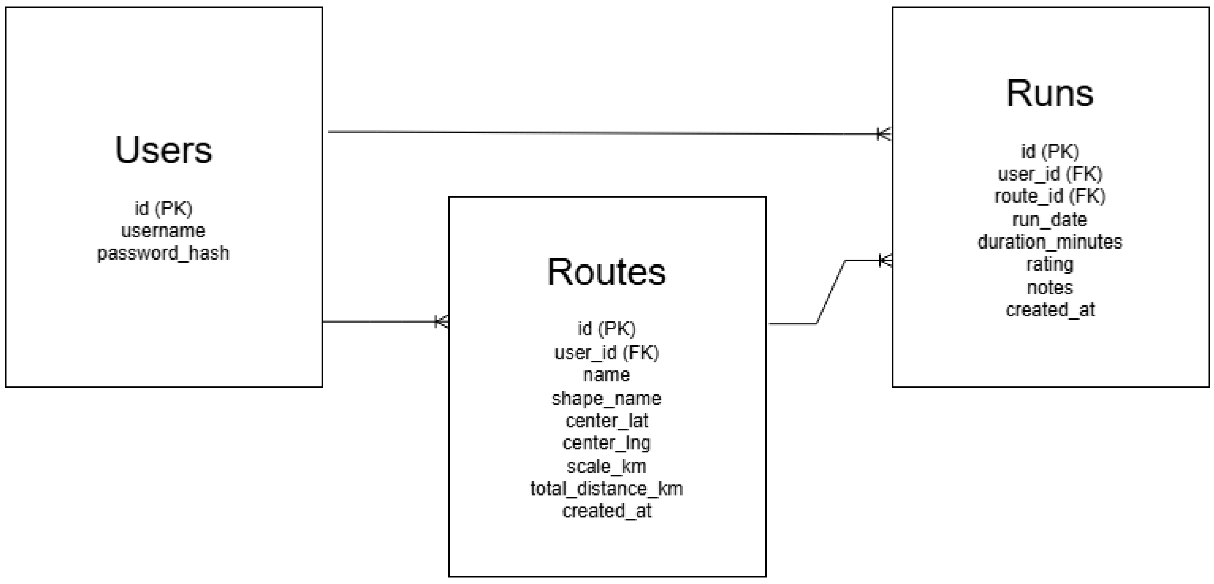


Figure 2 - Entity–relationship diagram showing Users, Routes, and Runs tables with primary and foreign key relationships.

The application uses a relational MySQL database with three core tables: Users, Routes, and Runs.

Users stores account credentials and is referenced by both child tables. Routes contains user-generated running shapes including centre coordinates, scale, total estimated distance, and creation timestamps. Each route belongs to exactly one user. Runs stores individual running activity logs, including date, duration, rating, and notes. Each run references both a user and the route followed.

The relationships are:

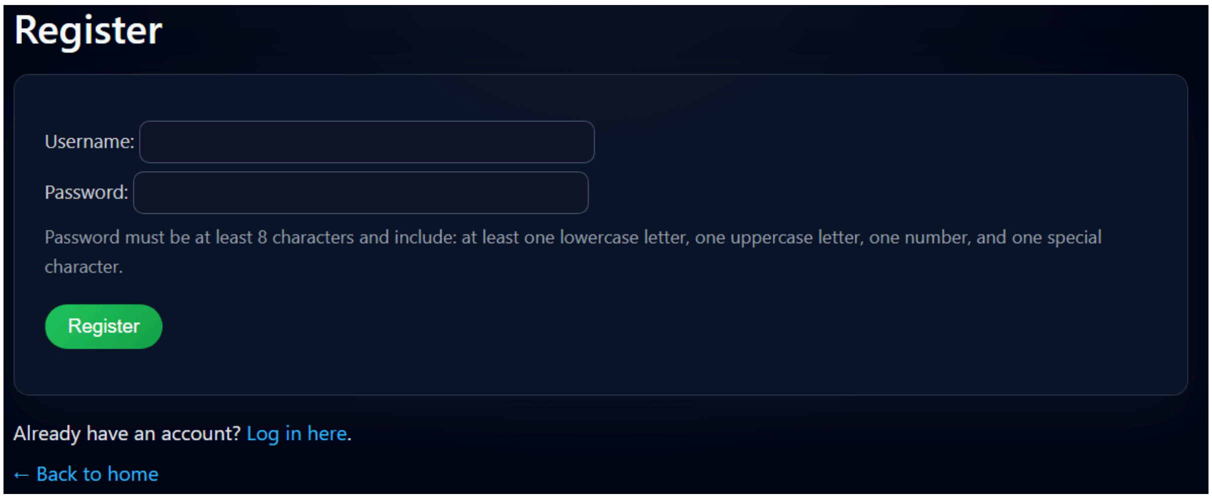
- Users 1-to-many Routes
- Users 1-to-many Runs
- Routes 1-to-many Runs

This structure cleanly separates accounts, saved routes, and logged activities.

Functionality

Shape Runner is a web-based fitness application that lets users design, save, and run custom GPS art routes. The system is built using Node.js, Express, EJS, MySQL, and Leaflet, and all features operate through clean, user-friendly web pages.

User Registration and Authentication


Register

Username:

Password:

Password must be at least 8 characters and include: at least one lowercase letter, one uppercase letter, one number, and one special character.

Register

Already have an account? Log in here.

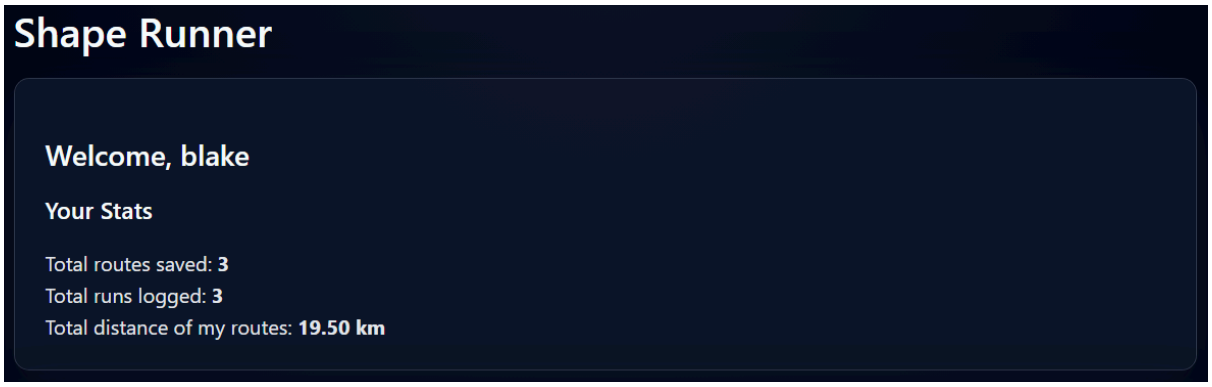
← Back to home

Figure 3 - User registration form with password validation rules.

Users create an account via a secure registration form that includes password validation rules (uppercase, lowercase, number, and special character). All passwords are hashed using bcrypt before being stored. Once registered, users can log in and are redirected to a personalised home dashboard that displays their statistics, including total routes created, total runs logged, and total distance across all their saved routes.

All authenticated pages use Express sessions to keep users logged in.

Home Dashboard


Shape Runner

Welcome, blake

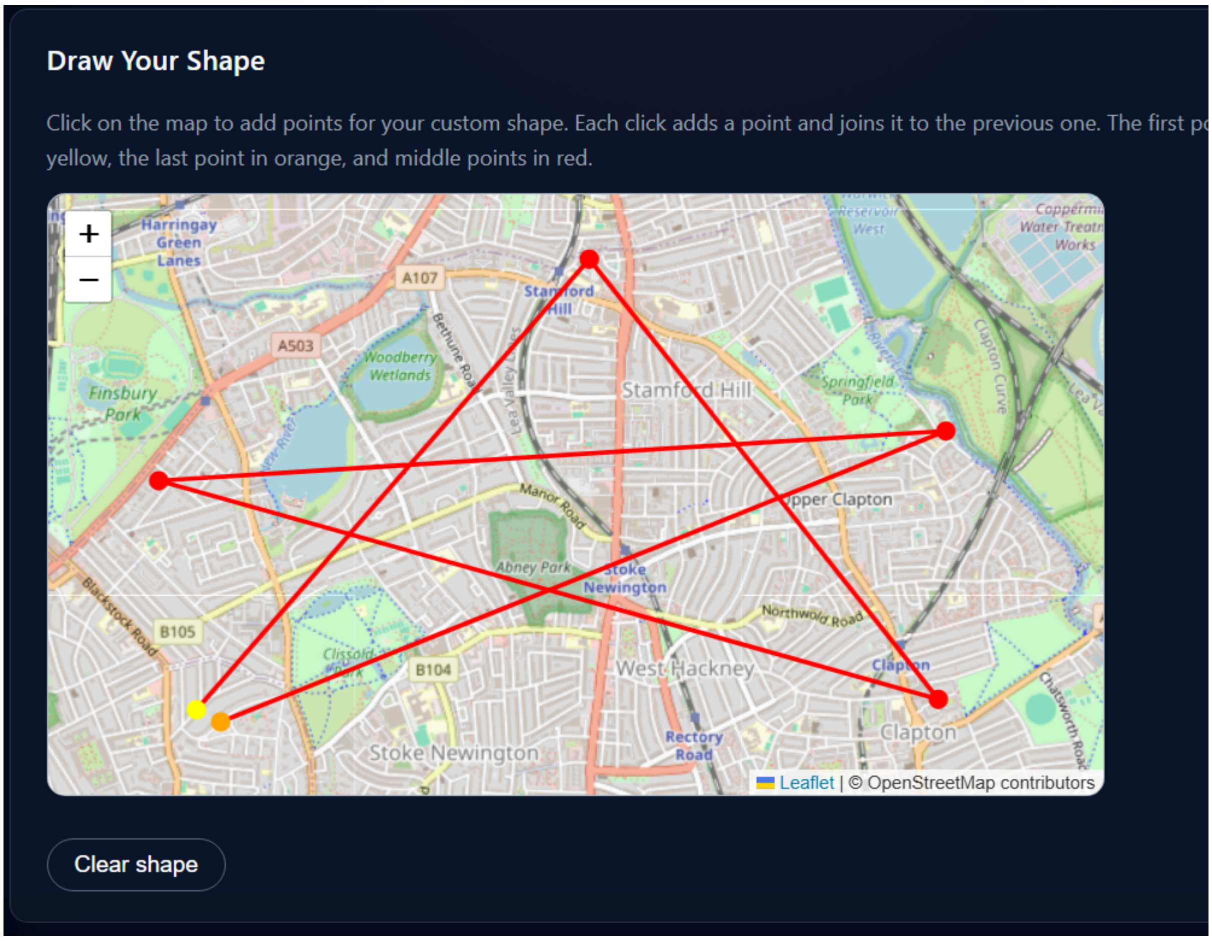
Your Stats

Total routes saved: **3**
Total runs logged: **3**
Total distance of my routes: **19.50 km**

Figure 4 - The home page shown after logging in, providing navigation to route generation, saved routes, run logging, and run history.

After logging in, the home page acts as the entry point for all application features. It displays navigation links, user stats, and access to route generation, route searching, run logging, and run history. A consistent navigation bar and modern CSS theme provide a clean experience across all pages.

Route Drawing & Generation


Draw Your Shape

Click on the map to add points for your custom shape. Each click adds a point and joins it to the previous one. The first po
yellow, the last point in orange, and middle points in red.

Figure 5 - Leaflet map where users click to draw custom shapes. Points appear in colour and are connected to form a free-drawn route shape.

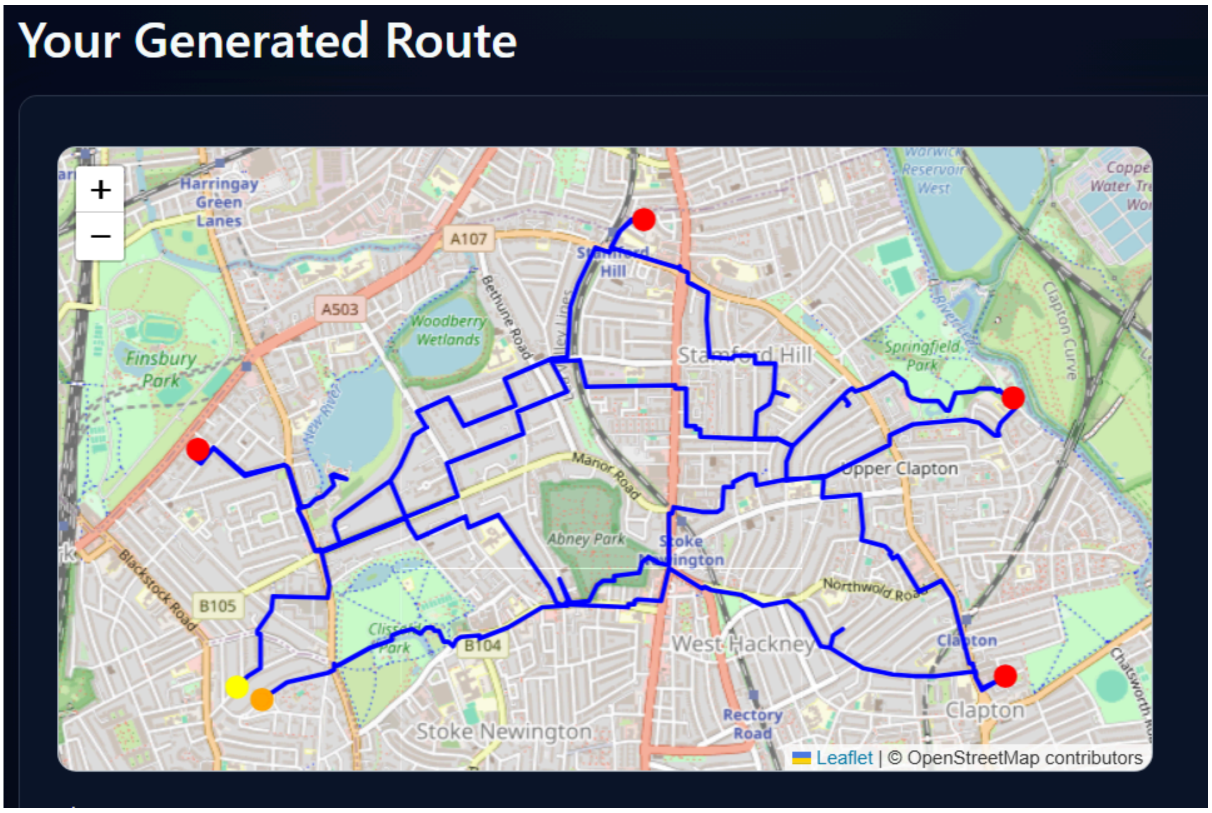


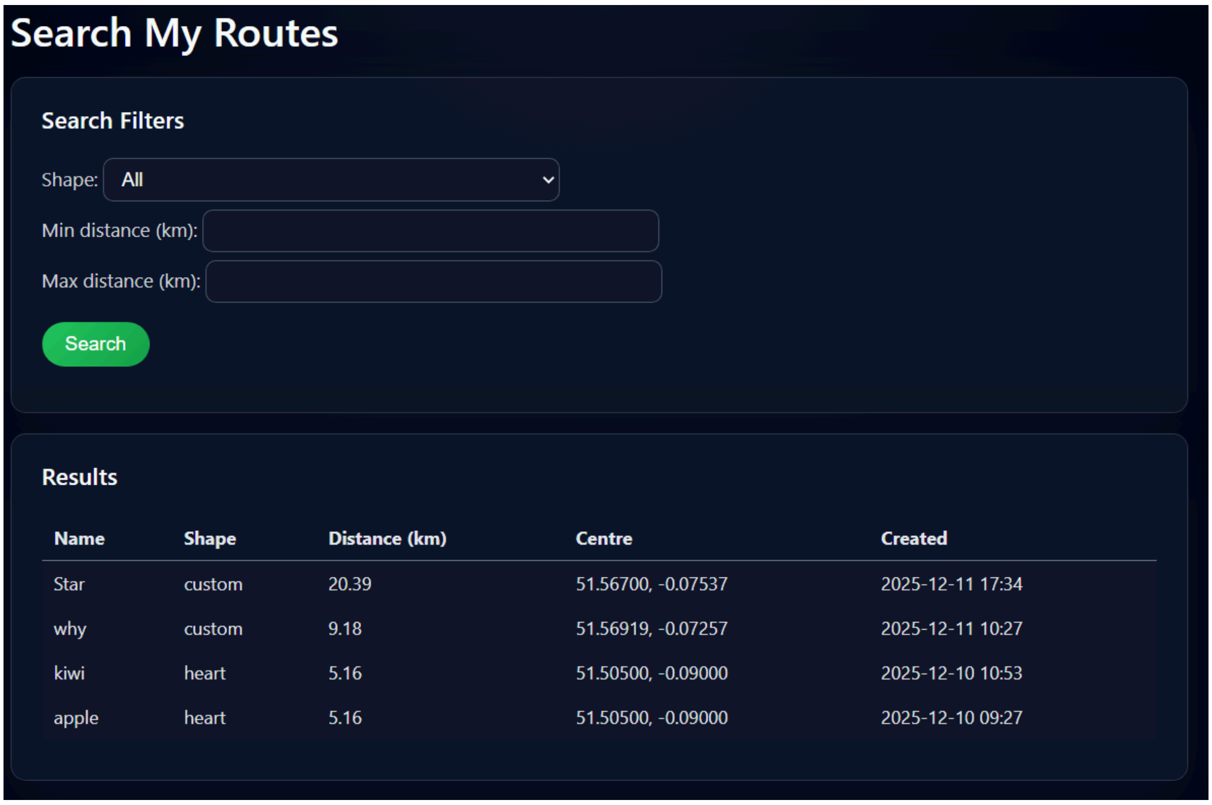
Figure 6 - The system's generated running route snapped to the road network using OpenRouteService routing.

The core feature of the app allows users to design their own running route. Using Leaflet, the user clicks on a map to place points — these represent vertices of the desired route shape. Points appear as small markers on the map, with the first point highlighted in yellow and the last point in orange for clarity.

Once at least two points are placed, the user can generate a route. The server receives the drawn points and sends them to OpenRouteService (ORS), which snaps them to real-world roads and generates a runnable polyline. The returned polyline is displayed on a results page as an accurate road-based route overlay, with all vertices shown. Users may then save the generated route to their profile.

If ORS fails, the system gracefully falls back to a straight-line approximation using geographical distance calculations.

Saving and Searching Routes


Search My Routes

Search Filters

Shape: All

Min distance (km):

Max distance (km):

Results

Name	Shape	Distance (km)	Centre	Created
Star	custom	20.39	51.56700, -0.07537	2025-12-11 17:34
why	custom	9.18	51.56919, -0.07257	2025-12-11 10:27
kiwi	heart	5.16	51.50500, -0.09000	2025-12-10 10:53
apple	heart	5.16	51.50500, -0.09000	2025-12-10 09:27

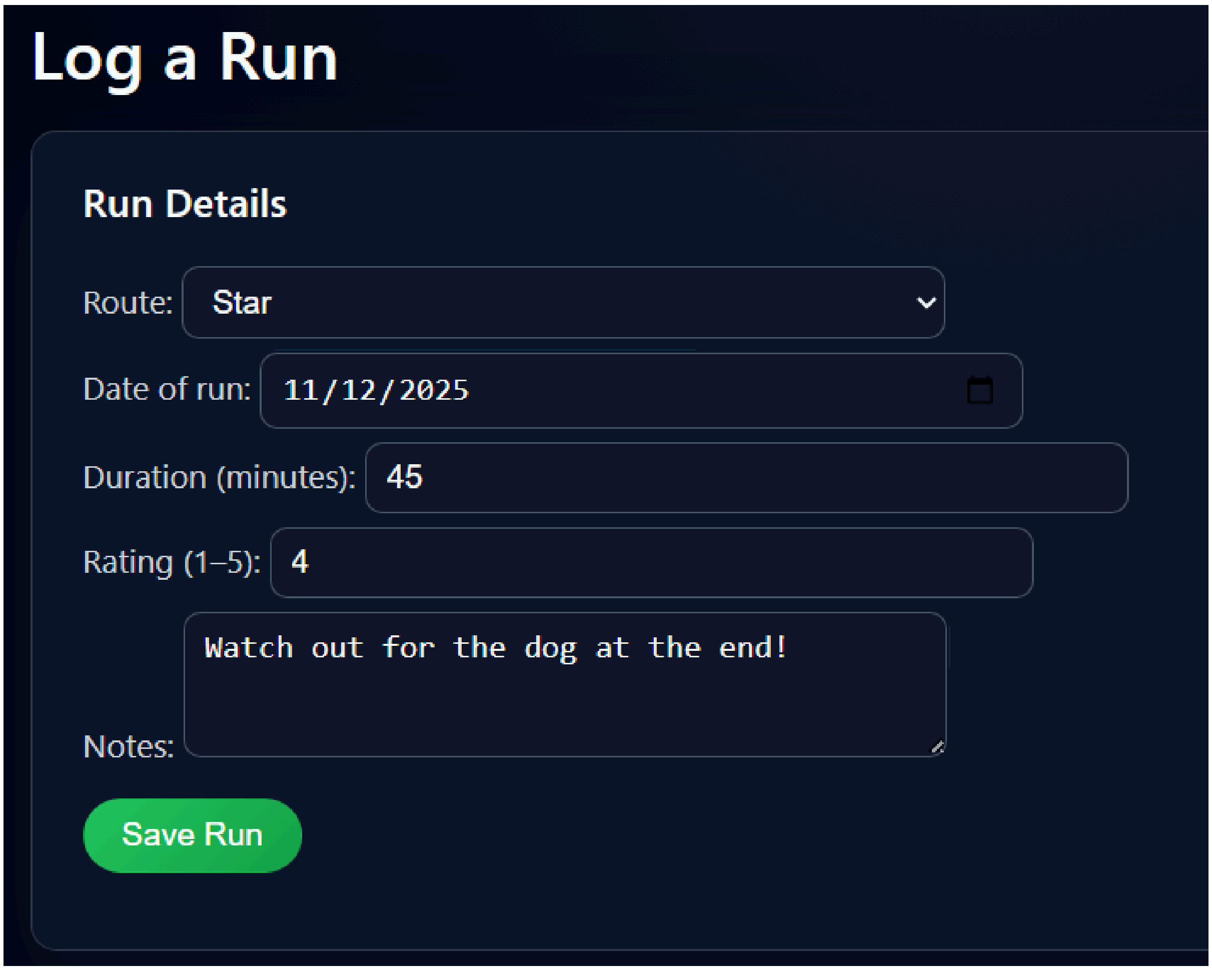
Figure 7 - Authenticated user view for filtering saved routes by shape type and distance.

After generating a route, users can save it by providing a name. The saved route includes shape metadata, centre coordinates (for preset shapes), total distance, and timestamp.

The route search page allows users to filter their saved routes using shape name, minimum distance, and maximum distance.

Results appear in a styled table showing route name, shape, distance, centre coordinates, and creation date.

Logging Runs



Log a Run

Run Details

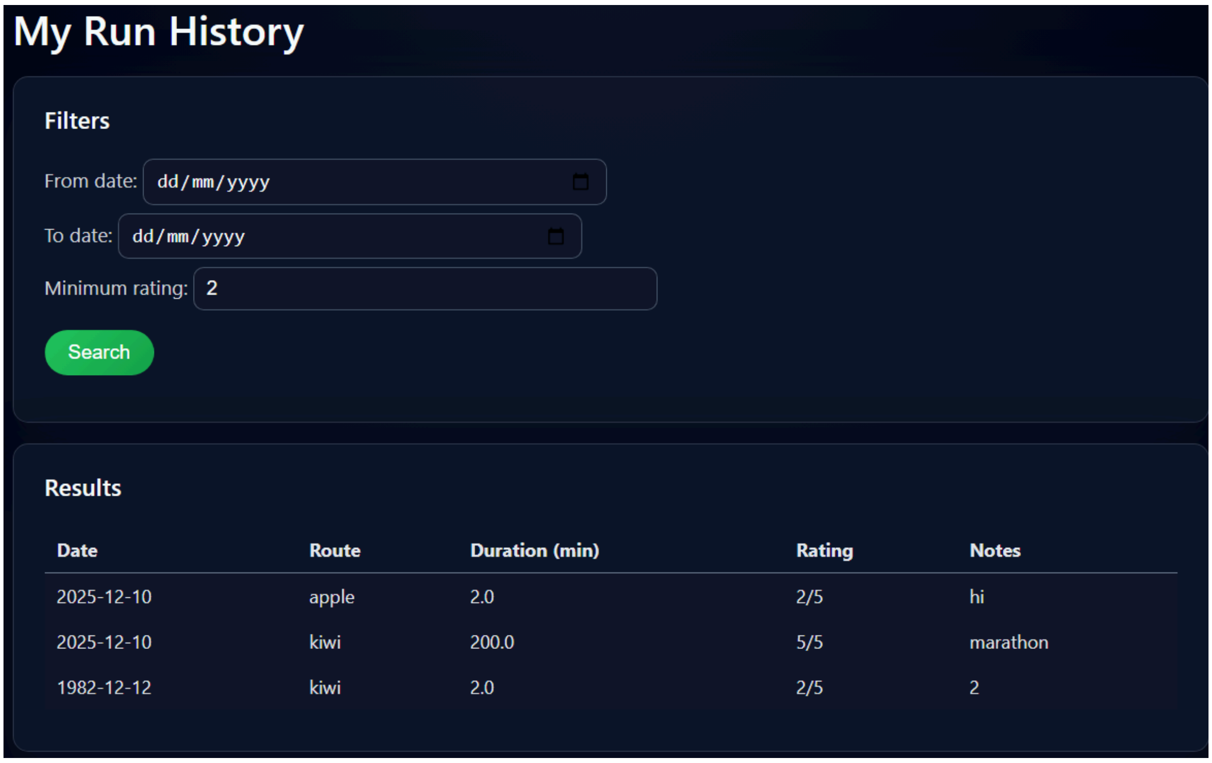
Route:	Star
Date of run:	11/12/2025
Duration (minutes):	45
Rating (1–5):	4
Notes:	Watch out for the dog at the end!

Figure 8 - Form used for logging a real-world run.

Users can log real runs performed using any of their saved routes. A run entry includes route selection, date of run, duration (minutes), rating (1–5), and optional notes.

After saving, a confirmation screen offers shortcuts to log another run or view run history.

Run History & Filtering


My Run History

Filters

From date: dd/mm/yyyy

To date: dd/mm/yyyy

Minimum rating: 2

Results

Date	Route	Duration (min)	Rating	Notes
2025-12-10	apple	2.0	2/5	hi
2025-12-10	kiwi	200.0	5/5	marathon
1982-12-12	kiwi	2.0	2/5	2

Figure 9 - Searchable table showing historical runs, dates, durations, ratings, and notes.

The run history page provides filters for date range (from/to) and minimum rating.

Filtered results appear in a table listing distance, duration, rating, notes, and the associated route name. This enables users to track progress and analyse performance over time.

Advanced Techniques

The Shape Runner project incorporates several advanced backend and geospatial techniques that go beyond standard CRUD operations. These features demonstrate technical depth in routing algorithms, API integration, validation, geolocation, dynamic map rendering, and fallback logic.

Road Snapping and Routing via OpenRouteService (ORS)

The application supports snapping hand-drawn user shapes to real-world road networks. This is achieved by sending a sequence of clicked map points to the ORS Directions API, which computes an optimal walking route passing through these points.

This involves constructing a JSON request body, invoking an external API, parsing encoded polylines, and returning a runnable GPS path.

Code Snippet — ORS Routing Request (Code from routes.js)

```
// Call OpenRouteService to snap points to roads
function routeShapeOnRoads(points, callback) {
  var apiKey = process.env.ORS_API_KEY;

  // If no key configured, just skip snapping and use straight lines
  if (!apiKey) {
    console.log('No ORS_API_KEY set, using straight lines only.');
    return callback(null, null);
  }

  if (!points || points.length < 2) {
    return callback(new Error('Not enough points to route'));
  }

  // Cap number of points to avoid ORS limits
  if (points.length > 40) {
    var step = Math.ceil(points.length / 40);
    var reduced = [];
    for (var i = 0; i < points.length; i += step) {
      reduced.push(points[i]);
    }
    // Ensure the final point is included
    var last = points[points.length - 1];
    var lastReduced = reduced[reduced.length - 1];
    if (lastReduced.lat !== last.lat || lastReduced.lng !== last.lng) {
      reduced.push(last);
    }
    points = reduced;
    console.log('Downsampled routing points to', points.length, 'coordinates');
  }

  // ORS expects [lng, lat]
  var coords = points.map(function (p) {
    return [p.lng, p.lat];
  });

  var options = {
    url: 'https://api.openrouteservice.org/v2/directions/foot-walking/geojson',
    method: 'POST',
    headers: {
      'Authorization': apiKey,
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      coordinates: coords
    })
  };

  request(options, function (err, response, body) {
    if (err) {
      console.error('ORS error:', err);
      return callback(err);
    }

    try {
      var data = JSON.parse(body);

      if (!data.features || data.features.length === 0) {
        return callback(new Error('No route returned from ORS'));
      }

      var feature = data.features[0];

      // geometry.coordinates is [[lng, lat], [lng, lat], ...]
      var routedCoords = feature.geometry.coordinates;

      var routedPoints = routedCoords.map(function (c) {
        return { lat: c[1], lng: c[0] };
      });

      // distance is in metres
      var distanceMeters = feature.properties && feature.properties.summary &&
feature.properties.summary.distance;

      var distanceKm = distanceMeters ? distanceMeters / 1000 : null;

      callback(null, {
        routedPoints: routedPoints,
        distanceKm: distanceKm
      });
    } catch (e) {
      console.error('Error parsing ORS response:', e);
      callback(e);
    }
  });
}
```

This algorithm enables dynamic road-aligned route generation, transforming freehand shapes into viable running paths.

Densification of User Points

When users draw simple shapes with few points, routing between only two points can cause unwanted shortcuts. To solve this, the system generates intermediate points along each line segment, improving routing accuracy and preserving the intended shape.

Code Snippet — Densifying Shape Vertices (Code from routes.js)

```
// Create extra points between each pair of points to better preserve shape
function densifyPoints(points, segmentsPerEdge) {
  if (!points || points.length < 2) return points;

  var result = [];
  for (var i = 0; i < points.length - 1; i++) {
    var a = points[i];
    var b = points[i + 1];

    // Always include the start of the segment
    if (i === 0) {
      result.push({ lat: a.lat, lng: a.lng });
    }

    // Add intermediate points
    for (var k = 1; k <= segmentsPerEdge; k++) {
      var t = k / segmentsPerEdge;
      var lat = a.lat + t * (b.lat - a.lat);
      var lng = a.lng + t * (b.lng - a.lng);
      result.push({ lat: lat, lng: lng });
    }
  }

  return result;
}
```

This improves ORS routing accuracy significantly, particularly when users draw large shapes with minimal points.

Geolocation Integration

The app uses the browser’s geolocation API to auto-centre the map on the user's real location. This improves convenience and helps users generate local routes.

Code Snippet — Geolocation for Map Start Position (Code from routes_generate.ejs)

```
// Geolocation to move map + centre inputs near the user
      if (!navigator.geolocation) {
```

```
    statusEl.textContent = 'Geolocation is not supported by this browser.';
  } else {
    statusEl.textContent = 'Trying to detect your location...';

    navigator.geolocation.getCurrentPosition(
      function (position) {
        var lat = position.coords.latitude;
        var lng = position.coords.longitude;

        latInput.value = lat.toFixed(5);
        lngInput.value = lng.toFixed(5);

        map.setView([lat, lng], 14);

        statusEl.textContent = 'Location detected. Click on the map to draw your shape.';
      },
      function (error) {
        console.log('Geolocation error:', error);
        statusEl.textContent = 'Could not get your location; using default centre.';
      }
    );
  }
```

Validation Pipeline using express-validator

Registration enforces password strength rules using a declarative validation chain, ensuring strong authentication practices.

Code Snippet — Password Validation Rules (Code from auth.js)

```
check('password')
  .isLength({ min: 8 })
  .withMessage('Password must be at least 8 characters.')
  .matches(/(?=.*[a-z])/)
  .withMessage('Password must contain at least one lowercase letter.')
  .matches(/(?=.*[A-Z])/)
  .withMessage('Password must contain at least one uppercase letter.')
  .matches(/(?=.*\d)/)
  .withMessage('Password must contain at least one number.')
  .matches(/(?=.*[^A-Za-z0-9])/)
  .withMessage('Password must contain at least one special character.')
```

This protects user accounts and demonstrates secure backend form handling.

Graceful Fallback Routing

If ORS fails or cannot produce a valid route, the app computes distance using spherical trigonometry and displays a straight-line approximation rather than breaking.

Code Snippet — Fallback Computation (Code from

```
  } else {
    console.log('Falling back to straight-line distance and points.');
    // Build straight-line distance
    totalDistanceKm = 0;
    for (var i = 1; i < straightPoints.length; i++) {
      var prev = straightPoints[i - 1];
      var curr = straightPoints[i];
      totalDistanceKm += distanceKm(prev.lat, prev.lng, curr.lat, curr.lng);
    }
  }
```

This ensures the system remains fully functional even without external API availability.

AI Declaration

I tried not to use AI for this project but I searched Google for the best way to calculate geographic distance between two points in javascript I had forgotten to add ‘-ai’, and the Google AI answered with essentially the exact code I needed (the Haversine formula), so I adapted it to my code (changing only the function name and the terms used for longitude and latitude).

Here is the code:

```
function distanceKm(lat1, lng1, lat2, lng2) {
  var R = 6371;
  var dLat = (lat2 - lat1) * Math.PI / 180;
  var dLng = (lng2 - lng1) * Math.PI / 180;
  var a =
    Math.sin(dLat / 2) * Math.sin(dLat / 2) +
    Math.cos(lat1 * Math.PI / 180) *
      Math.cos(lat2 * Math.PI / 180) *
      Math.sin(dLng / 2) *
      Math.sin(dLng / 2);

  var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
  return R * c;
}
```