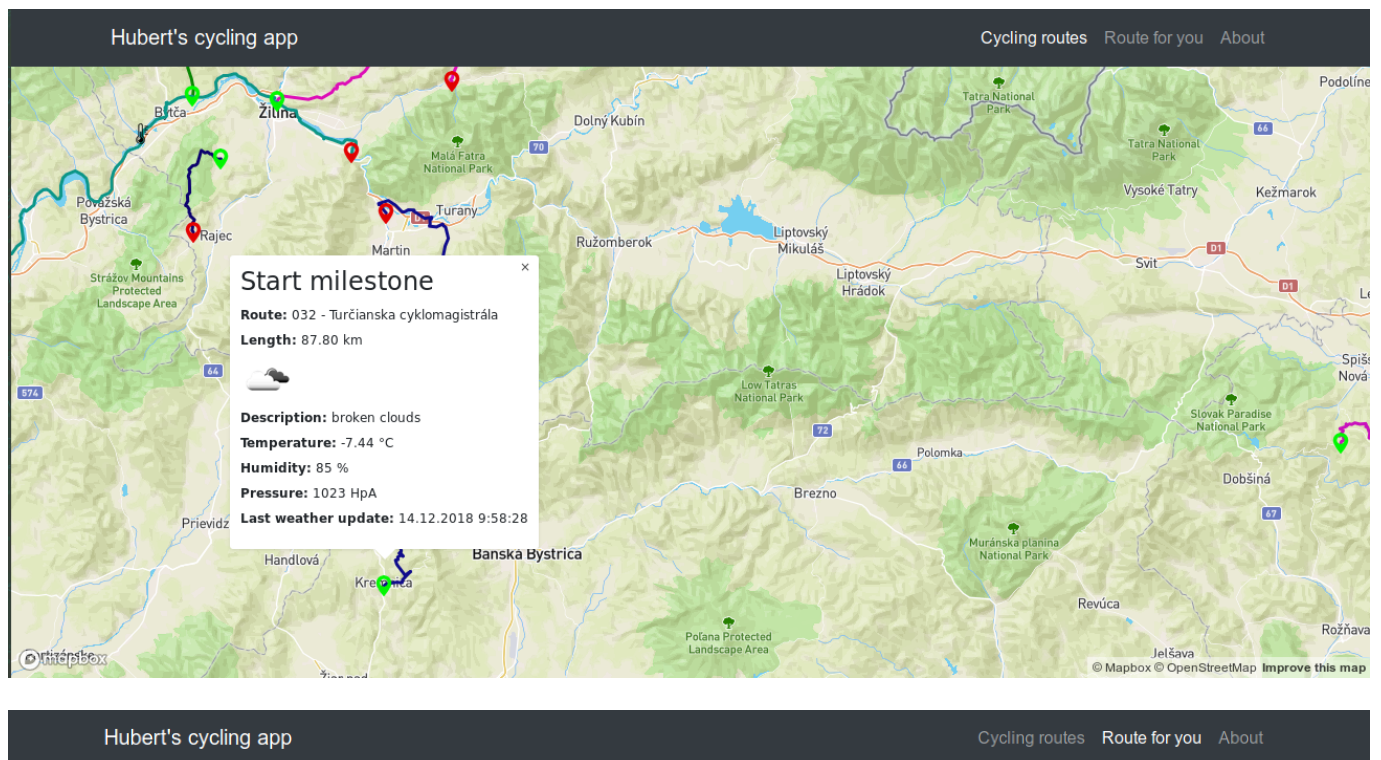


Overview

Application works with cycling routes data on the map and it's most important features are:

- show cycling routes and their data - name and length
- show their weather data - 1 route has more weather points based on it's length
- filtering cycling routes by desired minimal temperature and maximal humidity
- filtering cycling routes by their length
- show temperature heatmap of Slovak republic

This is it in action:



Find route matching your condition

Route length:

Minimal route length (km)

Maximum route length (km)

Find routes of your dreams

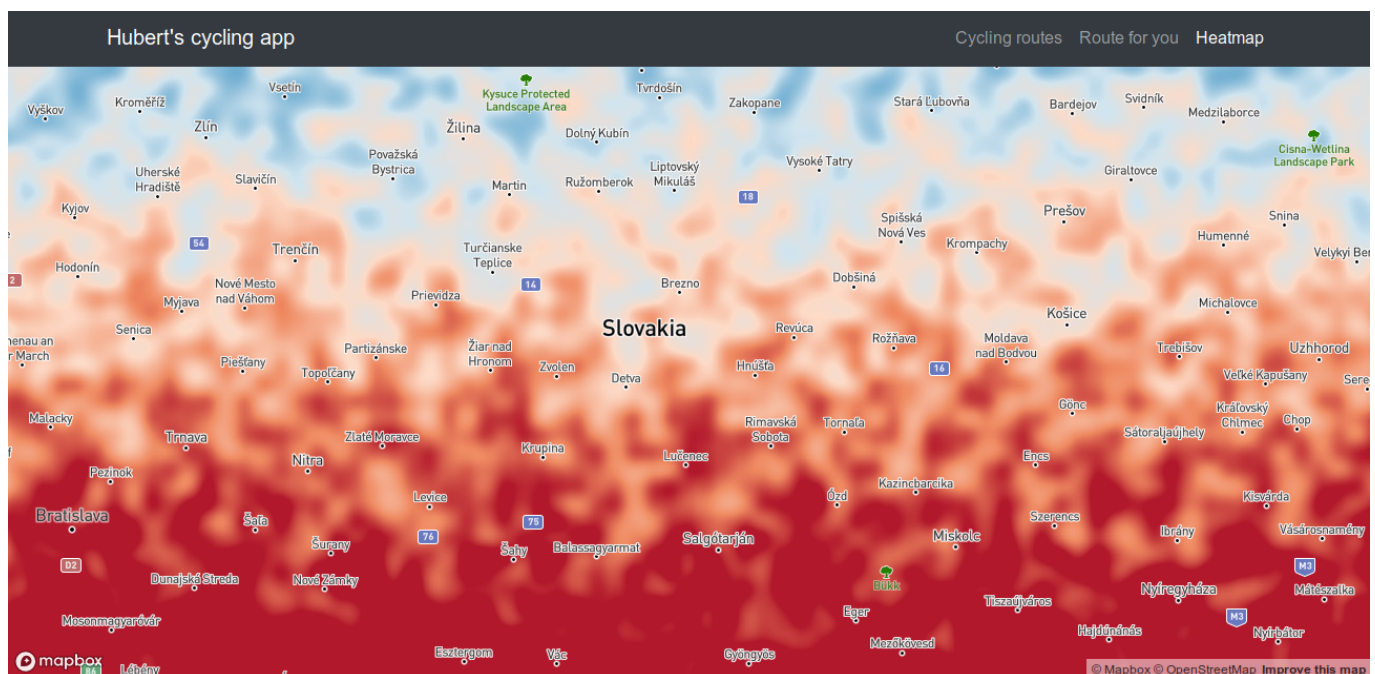
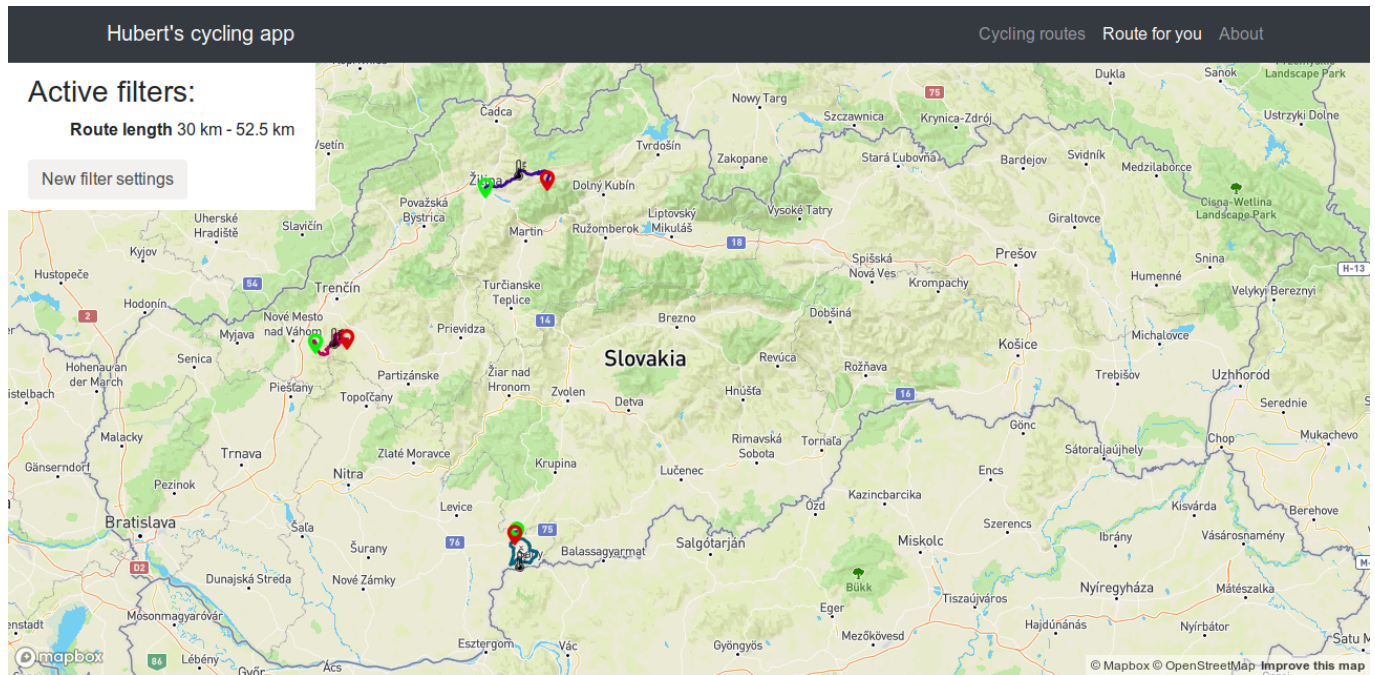
Find route matching your desired weather

Route weather stats:

Minimal desired average temperature (°C)

Maximal desired average humidity (%)

Find routes of your dreams



The application has 2 separate parts, the client which is a frontend web application using mapbox API and mapbox.js and the backend application written in Node.js, backed by PostGIS. The frontend application communicates with backend using an API. API is documented in interactive form with Swagger tool.

Frontend

The frontend application consists of two HTML pages: (`index.html`) and (`filter.html`). Both of the HTML files are displaying showing cycling routes geodata on the map via JS Mapbox library.

Filter page contains formular with selecting user's preferencies on routes and after submitting calls Backend's API with selected parameters. API will return routes that fit into user's preferences. All

dynamic JavaScript actions are done via JQuery library - mainly validating form inputs, sending API request via AJAX and initializing map with obtained cycling routes.

Index page contains a map with all cycling routes and their corresponding weather data.

Map initialization scripts are stored in (`js/map.js`) file. *Filtering validations and API calls* are stored in (`js/filter.js`) file. *Script file for index file* is stored in (`js/main.js`).

Route colors are generated randomly via a Color generator script.

Weather icons are obtained from *OpenWeatherMap repository*.

Backend

The backend application powered by Node.js and is responsible for:

- importing cycling routes data into POSTGIS database from gpx format
- continually gathering weather data relevant to saved routes data
- serving static files
- serving API
- communication with Postgres database
- serving API documentation
- logging application events into log files

Data

Cycling routes data

Cycling routes data are imported via an *bash* script that uses *ogr2ogr* tool for importing gpx formatted data into POSTGIS database. Before import, script runs DDL commands that create required tables. DDL file is stored in (`backend/data_definition/ddl.sql`). Import script consists of bash file (`backend/data_import/import.sh`) and 2 supporting SQL scripts (`backend/data_import/import1.sql`) and (`backend/data_import/import2.sql`).


Weather data

Weather data is obtained from *OpenWeatherMap API*. Count of weather query points for 1 route depends on route's length. Routes with length $< 30\text{km}$ are queried only for their starting and finishing points. Routes with length $\geq 30\text{km}$ and $< 100\text{km}$ are queried for starting, finishing points and for middle route point. Routes longer than 100km are queried for start, first quarter, middle, third quarter and finish points.

Gathering script runs every X milliseconds - according to configuration value stored in (`backend/config.json`). It queries every route for actual weather data and stores it into database.

Api

API is documented interactively through Swagger. When application runs, its interactive docs are accessible via URL: (localhost:3000/api-docs). There you can check all parameters needed and response value formats. You can also execute API calls from there as well.

 **swagger**

Hubert's cycling app API 1.0.0

[Base URL: localhost:3000/api]

Hubert's cycling app API documentation

Cycling routes - all

GET /cyclingRoutes Returns all cycling routes

Cycling routes - filtered

POST /cyclingRoutes/length Returns filtered list of cycling routes

POST /cyclingRoutes/weather Returns filtered list of cycling routes

Weather data points

GET /weatherPoints/{routeId} Returns weather data points with actual weather for specific route

Api methods

GET: /cyclingRoutes

Description: get all cycling routes

Parameters: none

Response format: [{ "fid": 0, "name": "string", "route": [{ "lat": 0, "lon": 0 }], "length": 0 }]

POST: cyclingRoutes/length

Description: get cycling routes filtered by route length in km

Parameters:

- minLength
- maxLength

Response format: [{ "fid": 0, "name": "string", "route": [{ "lat": 0, "lon": 0 }], "length": 0 }]

POST: cyclingRoutes/weather

Description: get cycling routes filtered by temperature and humidity

Parameters:

- minTemp
- maxHumidity

Response format: [{ "fid": 0, "name": "string", "route": [{ "lat": 0, "lon": 0 }], "length": 0 }]

Communication with database

All database communication is stored in *Database component*. It is located in (Backend/components/database/database.js).

Queries

Notes: - Ogr2ogr tool caused my lines to be of type MultiLineString => I needed to use ST_LineMerge everytime I wanted to use simple LineString methods. - cycling_routes_weather table contained weather data for all cycling_routes with historic data and more data point types => that's why I needed to use window function to prefilter them

Getting all cycling routes

```
SELECT fid, name, ST_AsGeoJSON(ST_LineMerge(route)) AS route,  
ST_Length(route::geography)/1000 as length  
FROM cycling_routes;
```

Explain:

```
"Seq Scan on cycling_routes  (cost=0.00..591.17 rows=210 width=362)"
```

Getting cycling routes filtered by length range

```
SELECT fid, name, ST_AsGeoJSON(ST_LineMerge(route)) AS route,  
ST_Length(route::geography)/1000 as length  
FROM cycling_routes  
WHERE ST_Length(route::geography)/1000 BETWEEN $1 AND $2
```

Explain:

```
"Seq Scan on cycling_routes  (cost=0.00..123.01 rows=1 width=362)"  
"  Filter: (((st_length((route)::geography, true) / '1000'::double precision) >=  
'20'::double precision) AND ((st_length((route)::geography, true) / '1000'::double  
precision) <= '50'::double precision))"
```

Getting cycling routes filtered by actual weather data (average route's temperature and humidity)


```

SELECT fid, name, ST_AsGeoJSON(route) AS route, ST_Length(route::geography)/1000 as
length FROM cycling_routes
JOIN (
    SELECT cycling_route_id,
    AVG((weather).temperature) AS avg_temperature,
    AVG((weather).humidity) AS avg_humidity FROM (
        SELECT cycling_route_id, weather,
        rank() OVER (
            PARTITION BY point_type, cycling_route_id ORDER BY measure_date DESC
        )
        FROM cycling_routes_weather
    ) actual_weather
    WHERE rank = 1
    GROUP BY cycling_route_id
) temp ON fid = cycling_route_id
WHERE avg_temperature >= $1 AND avg_humidity <= $2

```

Explain:

```

"Hash Join (cost=1542.02..1598.76 rows=16 width=362)"
"  Hash Cond: (cycling_routes.fid = temp.cycling_route_id)"
"    -> Seq Scan on cycling_routes (cost=0.00..12.10 rows=210 width=354)"
"    -> Hash (cost=1541.82..1541.82 rows=16 width=4)"
"      -> Subquery Scan on temp (cost=1540.85..1541.82 rows=16 width=4)"
"        -> GroupAggregate (cost=1540.85..1541.66 rows=16 width=20)"
"          Group Key: actual_weather.cycling_route_id"
"          Filter: ((avg((actual_weather.weather).temperature) >=
'-2'::double precision) AND (avg((actual_weather.weather).humidity) <= '90'::double
precision))"
"            -> Sort (cost=1540.85..1540.99 rows=57 width=56)"
"              Sort Key: actual_weather.cycling_route_id"
"              -> Subquery Scan on actual_weather
(cost=1141.06..1539.18 rows=57 width=56)"
"                Filter: (actual_weather.rank = 1)"
"                -> WindowAgg (cost=1141.06..1397.00 rows=11375
width=79)"
"                  -> Sort (cost=1141.06..1169.50 rows=11375
width=71)"
"                    Sort Key:
cycling_routes_weather.point_type, cycling_routes_weather.cycling_route_id,
cycling_routes_weather.measure_date DESC"
"                      -> Seq Scan on cycling_routes_weather
(cost=0.00..374.75 rows=11375 width=71)"

```

Getting milestones of specific cycling route

```

SELECT
    fid,
    ST_Length(route::geography)/1000 as length,
    ST_AsGeoJSON(ST_StartPoint(ST_LineMerge(route))) AS route_start,
    ST_AsGeoJSON(ST_Line_Interpolate_Point(ST_LineMerge(route), 0.25)) AS
route_first_quarter,
    ST_AsGeoJSON(ST_Line_Interpolate_Point(ST_LineMerge(route), 0.5)) AS

```

```
route_middle,  
    ST_AsGeoJSON(ST_Line_Interpolate_Point(ST_LineMerge(route), 0.75)) AS  
route_third_quarter,  
    ST_AsGeoJSON(ST_EndPoint(ST_LineMerge(route))) AS route_finish  
FROM cycling_routes  
WHERE fid = $1
```

Explain:

```
"Index Scan using cycling_routes_pkey on cycling_routes  (cost=0.14..21.68 rows=1  
width=172)"  
"  Index Cond: (fid = 12)"
```