Backend Requirements

This document describes the template rendering requirements, static file structure, and API endpoints required for the frontend to function.

1. Static Files

The backend must make static files available. These include javascript files, CSS, and images. These files will be delivered to the backend developer in a single folder, whose substructure should be preserved. A file in the root of this folder must be available in the root directory of the web domain, e.g. `static/robots.txt` must be accessible at `starmaps.com/robots.txt`.

No static file will clash names with any other page or API described in this document.

2. Page Rendering

Jinja templates will be used to server-side render HTML pages before being delivered to the browser.

This section will describe the structure of the templates as well as the data that must be provided in order to correctly render the templates

2.1. Template Content Specification Format

The format used to specify data required for rendering a specific template is hopefully intuitive, but this commented example should be used as a reference if any specification is unclear:

```
# A property named “title” which is a string, at the top-level
title: string
# A dictionary property, containing other properties
mountain: {
  name: string
  difficulty: number
  trail_count: number
  # A property that contains a list of data
  trails: [{
    name: string
    difficulty: number
  }]
}
# An optional property, which may or may not be present. A
# specification will describe when the attribute is expected
# to not be present.
website_link?: url
```

2.2. Parent Templates

Some templates extend other templates to reduce code duplication. Because of this, the backend must be able to handle the Jinja `extends` directive.

2.2.1. Page Base Content

Most pages require data for the page base template, which includes basic navigational data.

```
# List of pages to appear in the navigation bar
nav_links: [{
  title: string # Text to display in the navigation link
  page: string # Page slug
  to: url # HREF to use for the navigation link (relative URL)
}]
active_page: string # Page slug of currently opened page
```

The page slugs should be unique for each page in the website. This can be done by taking the relative URL of the page in most cases (e.g. `starmaps.com/about` would have the page identifier `about`). You should probably use `index` for the root page, instead of a blank slug.

2.3. Page Templates

2.3.1. Index Page

The template is named `index.jinja`. The rendered form of this page should be available at the root point of the domain, i.e. `/`.

This page requires the page base content and nothing else.

2.3.2. About Page

The template is named `about.jinja`. The rendered form of this template should be available at `/about`.

This page requires the page base content and nothing else

2.3.3. Mountain Search Page

The template is named `mountains.jinja`. The rendered form of this template should be available at `/search`.

This page uses the URL query string (QS) for storing the filter and search settings. These are not used for template rendering, but the backend must parse the QS to build the correct query for the database. The QS may have the following properties, with the defaults shown:

```
q?: string # Search string (default=””)
page?: integer # Search page number (default=0)
limit?: integer # Number of results shown per page (default=20)

# filters string has comma separated filters, looking like:
# filters=trailcount-10-100,near-nh-0-200
# which means a trail count between 10 and 100 and 0-200 miles
# from New Hampshire
# The exact list of filters is not specified in this document
filters?: string # (default=””)
```

The template content includes the page base content, along with the following:

```
mountains: [{
  name: string
  beginner_friendliness: number
  difficulty: number
  state: string
  trail_count: number
  vertical: number
  map_link: url # URL to interactive map page for the mountain
}]
pages: {
  prev?: url # Link to previous search result page (if it
exists)
  next?: url # Link to next search result page (if it exists)
}
```

2.3.4. Mountain Rankings Page

The template is named `rankings.jinja`. The rendered form of this template should be available at `/rankings`.

This page uses the query string (QS) to store the sort attribute and direction:

```
sort: “beginner” | “difficulty”
order: “desc” | “asc”
```

The template content includes the page base content, along with the following:

```
sort: string # Must match QS value
order: string # Must match QS value
mountains: [{
  name: string
  beginner_friendliness: number
  difficulty: number
  state: string
  map_link: url # URL to interactive map page for the mountain
}]
```

2.3.5. Map Page

The template is named `map.jinja`. The rendered form of this template should be available at `/map/:mountain_name`. `:mountain_name` should be replaced by the unique mountain name that references the mountain being displayed.

The template content is the following:

```
mountain: {
  unique_name: string # Unique mountain name, for API requests
  name: string
  statistics: {
    # A dictionary containing any string key to a string value.
    # These will be displayed verbatim on the page in the
mountain
    # overview section, so each key/value should be human
readable.
    [k: string]: string
  }
  trails: [{
    name: string
    difficulty: number
  }]
  lifts: [{
    name: string
  }]
}
```

3. API Endpoints

3.1. Mountain Data

`GET /data/:mountain_name/objects`

Fetch the trail data for the unique mountain name `:mountain_name`.

Response should be a 404 error if the unique mountain name does not exist, or on success return a JSON document in the following shape:

```
{
  “trails”: [
    {
      “id”: “trail-id-unique-for-this-mountain”,
      “name”: “Trail name”,
      “difficulty”: 0.3,
      “trail_length”: 400,
      “vertical_drop”: 200,
      “average_pitch”: 16.2,
      “steepest_pitch”: 22.6
    }
  ],
  “lifts”: [
    {
      “id”: “lift-id-unique-for-this-mountain”,
      “name”: “Lift name”,
      “lift_length”: 400,
      “vertical”: 200
    }
  ]
}
```

3.2. SVG Maps

`GET /data/:mountain_name/map.svg`

Fetch the SVG map data for the unique mountain name `:mountain_name`. Response should be a valid SVG file representing the map of the mountain, or a 404 error if the unique mountain name does not exist/does not have a map.

3.2. Path Data

`GET /data/:mountain_name/paths`

Fetch the 3D map data for the unique mountain name `:mountain_name`.

Response should be a 404 error if the mountain name does not exist, or on success return a JSON document in the following format:

```
{
  “trails”: [
    {
      “id”: “trail-id-unique-for-this-mountain”,
      “points”: “point-string”
    }
  ],
  “lifts”: [
    {
      “id”: “lift-id-unique-for-this-mountain”,
      “points”: “point-string”
    }
  ]
}
```

A “point-string” is a headerless CSV with three columns: latitude, longitude, and elevation (meters). Instead of newlines, the string should use vertical bars (|). It is suggested to round the data before sending it over the network, to reduce file size. Five decimal digits for latitude/longitude and one decimal digit for elevation works well. Example point-string, with two points:

```
“-41.723,72.123,213|-41.736,72.124,218”
```

4. Test Templates

A `test_templates.zip` archive is included containing templates to test most of the page rendering requirements. The test templates follow the naming conventions given in 2.3., and the real templates should be a drop-and-replace for the tests.

Not covered by the test templates are static files, API requests, and most of the content for map pages (2.3.5.). The search page also does not have a UI for searching, only displaying results (manually writing a query string will work instead).