# IFQ715: Web Development Practice
# Assignment 2: Develop a web application with React

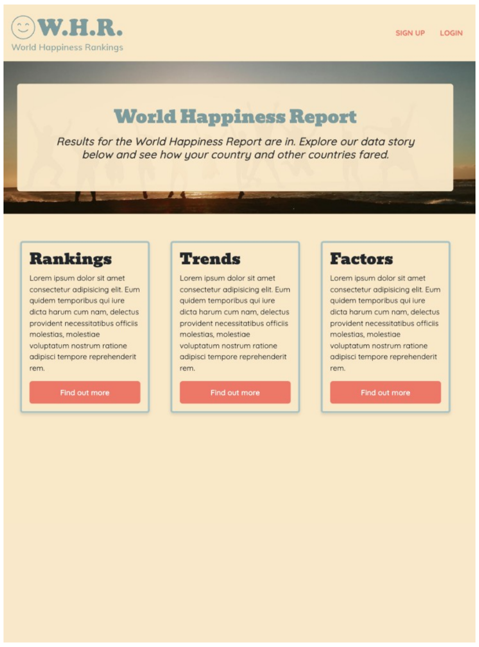
Matthew Campbell
20.09.2024
Queensland University of Technology

## Introduction

World Happiness Rankings is a Single Page Application (SPA) built in React that consumes World Happiness Report data from an API. It is a data visualisation project that utilises two libraries for charting: AG Grid and Recharts. Design and UI (User Interactivity) elements were carried out using a combination of React-Bootstrap, Bootstrap 5.3 and CSS with some custom properties for overriding Bootstrap styles. In this report I describe my approach to UI and design and address the technical aspects of the application, including its architecture, how I approached creating components and how I utilised the available endpoints. I will also describe the application's limitations and some of the challenges I faced.

## UI/Design

This application leans heavily on React-Bootstrap and Bootstrap 5.3 for responsiveness and looks mostly fine across most devices. Responsiveness testing was conducted with Chrome's DevTools and care was made to ensure responsivity in landscape and portrait modes for all devices. One exception was for the iPad Pro in portrait mode, where large amounts of blank space can be seen. An attempt to fix this using dynamic viewport height (Gayan, 2023) was unsuccessful, so a task for future improvements might involve media queries (W3 Schools, n.d.) or adding more content to the page.



World Happiness Report

Results for the World Happiness Report are in. Explore our data story below and see how your country and other countries fared.

Rankings

Lorem ipsum dolor sit amet consectetur adipisicing elit. Eum quidem temporibus qui iure dicta harum cum nam, delectus provident necessitatibus officiis molestias, molestiae voluptatum nostrum ratione adipisci tempore reprehenderit rem.

Trends

Lorem ipsum dolor sit amet consectetur adipisicing elit. Eum quidem temporibus qui iure dicta harum cum nam, delectus provident necessitatibus officiis molestias, molestiae voluptatum nostrum ratione adipisci tempore reprehenderit rem.

Factors

Lorem ipsum dolor sit amet consectetur adipisicing elit. Eum quidem temporibus qui iure dicta harum cum nam, delectus provident necessitatibus officiis molestias, molestiae voluptatum nostrum ratione adipisci tempore reprehenderit rem.

*Figure 1: Blank space in iPad Pro portrait mode*

Design elements like the logo, colour palette and font are variously custom or sourced elsewhere. The logo was created using Adobe's logo generator (Adobe, 2024). To create a colour palette that would match the logo, I used the Geco Colorpick (Geco, n.d.) Chrome extension on the logo during the creation process to identify background and primary colours and experimented with different colours for the rest of the palette (secondary, text, accent, etc). Fonts were sourced from Google Fonts (Google, n.d.).

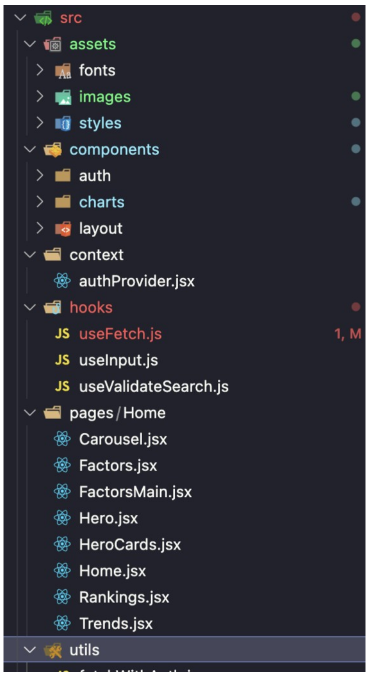
To the best of my ability, I tried to keep the code for styling as clean and modular as possible, however the opinionated nature of Bootstrap/React-Bootstrap at times got in the way of this. Where no Bootstrap class (that I knew of) allowed me to customise an element as I wanted, I used CSS, for example when styling the background image and mask in the hero section. Overriding colour palettes and styling for elements like buttons was for the most part fairly simple with CSS custom properties. This, however—and for reasons I still don't understand—proved near impossible with React-Bootstrap's NavLink component. Here, unfortunately, I had to resort to using "!important", which is generally considered bad practice (Senadheera, 2023).

Another problem I faced and was ultimately unable to solve before deadline was meaningfully displaying data fetching to the user. This in theory is handled by the "fetchPending" state which is defined in the useFetch hook and imported into the chart components. There, the logic is set up so that if fetchPending is true, a React-Bootstrap spinner should appear. This was actually working reasonably well while I was testing the application with the caching and abort controller logic (refer to next section) commented out, but when the logic is active, it doesn't appear to work at all.

The effect is that the user doesn't get any feedback that their request is being processed, and must discern this from looking at the data UI. If, for example in the rankings section, a query for 2015 with no country is entered, then another query for 2016 is entered after that, the change in state from one year to the next is only perceptible as a quick refreshing of the table. If the query is cached, it's barely perceptible at all. This problem is especially noticeable in the factors section, where there isn't even a column for "year", which might have made changes in the data being displayed more noticeable. Future work on the app would look into what's causing the spinner not to work, and perhaps also add a legend to display what the user has searched for.

## Application Architecture & Technical Details

For this application I preferred a logic-first approach (Olanrewaju, 2024) and tried to follow DRY ("Don't Repeat Yourself") principles (srinam, 2024) as much as possible, creating reusable custom hooks and—to a smaller extent—components. Application architecture followed React best practice (Dhanani, 2024) whereby assets, including images and stylesheets; components; hooks, including for fetching API data and sanitising requests; pages, namely the homepage and its various components; and utilities (utils) are kept in separate directories (*Figure 2*) to keep the project structure clean and easy to understand (Dhanani, 2024).



```
src
  assets
    fonts
    images
    styles
  components
    auth
    charts
    layout
  context
    authProvider.jsx
  hooks
    useFetch.js        1, M
    useInput.js
    useValidateSearch.js
  pages/Home
    Carousel.jsx
    Factors.jsx
    FactorsMain.jsx
    Hero.jsx
    HeroCards.jsx
    Home.jsx
    Rankings.jsx
    Trends.jsx
  utils
```

*Figure 2: Directory structure overview*

The app is divided into four sections: the "hero" section, a "rankings" section, a "trends" section and a "factors" section. Components—like Rankings.jsx—containing UI and form elements are found in the pages/Home directory, while related tables or charts are kept in components/charts.

Building web applications in the past, I've found myself guilty of writing the same functions multiple times. Beginning with a logic-first approach forced me to consider early on how I wanted to handle the data and which aspects of doing so could be made more modular so that I didn't end up rewriting logic within the components themselves.

These functions and hooks came about as a result of research and experimentation. I acknowledge here that once the application grew in complexity, some functions—especially surrounding authentication—began to overlap with others or fall into disuse, like in the case of fetchWithAuth.js (utils). I would put this down to some oversights during the planning stage. I will now take the time to describe in brief how these hooks work.

### <u>Authentication provider (authProvider.jsx):</u>

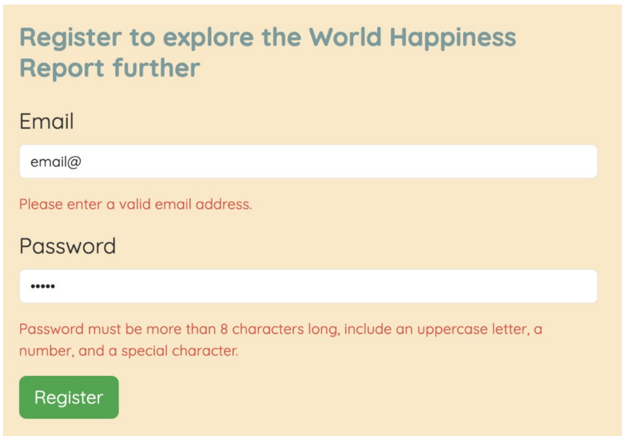
Context is created in React to make certain data available throughout an application, as opposed to having to pass it down from parent to child (React, n.d.) which can become cumbersome. This authentication provider is based on Arya's DEV blog post (Arya, 2023). It sets up a JSON web token (JWT) authentication context to manage user state, including setting tokens for accessing the "factors" data when the user logs in and managing logout state. Without this context, logic for setting JWTs would need to be repeated across components. (I'm aware that JWTs are only used for one of the endpoints, but I mention this for the sake of argument). The context also makes logic for logging out workable wherever the user is within the app.

Future refactoring might improve this function to also handle setting the API key in headers, as currently this is handled in the various chart components.

## <u>**Fetch hook (useFetch.js):**</u>

This hook contains the logic for fetch calls, including asynchronous data fetching (Joy, 2023), caching for performance (Adegbuyi, 2020) and an abort controller (Chamochumbi, 2021) for when asynchronous requests need to be cancelled, typically as a result of user behaviour (Labaran, 2024). State variables and their subtending functions for fetching data, fetch pending status, errors and header setting are constructed within the function and then returned. useFetch is then imported into the "chart" components where its state variables are destructured (W3 Schools, n.d.) according to the needs of the particular component and the function is called with the appropriate URL.

## <u>**Input validation for sign up and login (useInput.js):**</u>


Register to explore the World Happiness Report further

Email

email@

Please enter a valid email address.

Password

•••••

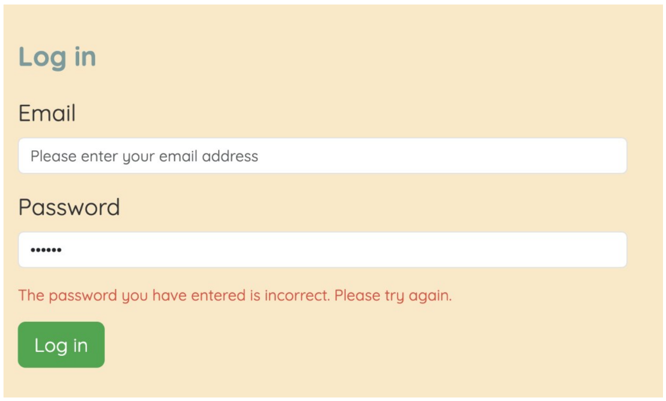
Password must be more than 8 characters long, include an uppercase letter, a number, and a special character.

Register

*Figure 3: Good user feedback - authentication UI for signing up*

useInput sanitises input values for emails and passwords. The validation logic utilises regex codes to check that emails entered comply with validation standards (Odyntsov, 2024) and that passwords are strong enough (Ibrahim, 2022). A host of other functions are included in line with Fulness's guide to creating input validation hooks on Medium (Fulness, 2023) which contribute to better user experience (*Figure 3*), including displaying errors if an input field is clicked out of (if it has been "touched") and there's an invalid entry and for resetting input values, for example when forms have been submitted.

Upon reviewing the implementation of the Login.jsx component, I realised that applying the same password validation logic to the sign up and login endpoints was an oversight. While checking for strong passwords is obviously important at the sign up stage, it is redundant at login where verifying credentials should be the focus.


Log in

Email

Please enter your email address

Password

••••••

The password you have entered is incorrect. Please try again.

Log in

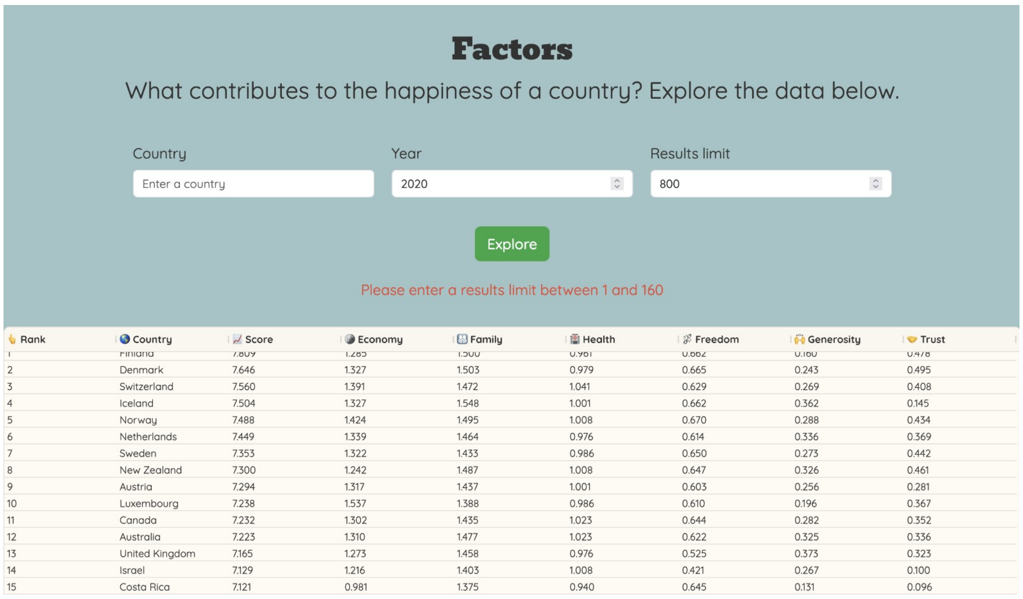
*Figure 4: Invalid implementation of verification logic at the login endpoint*

## <u>**Data query validation (useValidateSearch.js)**</u>

useValidateSearch sanitises queries with values that fall outside the range of data available and provides feedback to the user when a particular search doesn't yield results (*Figure 5*). The function accepts arguments for country, year and limit. Simple if statements are used to check if year and limit are within the allowed range; if not, relevant error state is set. For validating country inputs, a fetch call (using useFetch) is used for the "countries" endpoint. An if-statement then checks if the country input is *not* empty (a condition which is acceptable for the rankings and factors section) *and* does *not* match (.includes() method) a country in a mapped version of the countries data:

```
if (country.trim() !== "" &&
!countriesData.map((c) => c.toLowerCase())
.includes(country.trim().toLowerCase())
) {
setError("Please enter a valid country name.");
return false;
}
```

The useValidateSearch hook works well in the application, however the validity of this approach may be questionable given that not all parameters are used for all data queries (for example, "limit" is only used in the factors section). Furthermore, in its current state, the hook is not suited for use in the trends section, which *does* in fact require "country" as its only input. The current behaviour for the trends section when white space is entered is simply to default to the value set in the query state. This will be a task for future refactoring.


Factors
What contributes to the happiness of a country? Explore the data below.
Explore

Country: Enter a country
Year: 2020
Results limit: 800

Please enter a results limit between 1 and 160

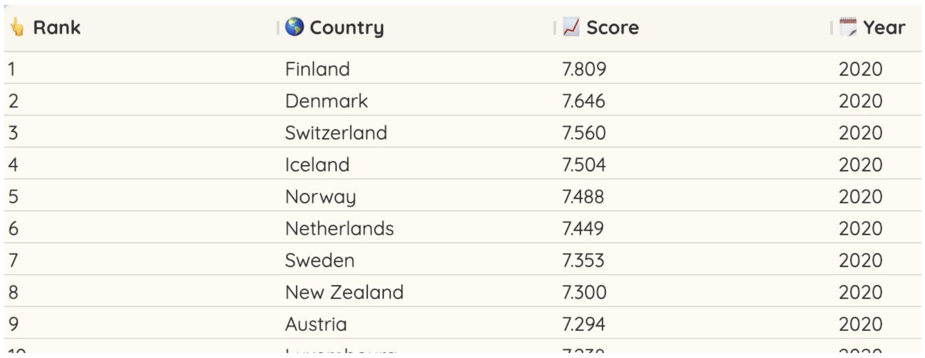
| Rank | Country | Score | Economy | Family | Health | Freedom | Generosity | Trust |
|---|---|---|---|---|---|---|---|---|
| 1 | Finland | 7.809 | 1.285 | 1.500 | 0.961 | 0.662 | 0.160 | 0.478 |
| 2 | Denmark | 7.646 | 1.327 | 1.503 | 0.979 | 0.665 | 0.243 | 0.495 |
| 3 | Switzerland | 7.560 | 1.391 | 1.472 | 1.041 | 0.629 | 0.269 | 0.408 |
| 4 | Iceland | 7.504 | 1.327 | 1.548 | 1.001 | 0.662 | 0.362 | 0.145 |
| 5 | Norway | 7.488 | 1.424 | 1.495 | 1.008 | 0.670 | 0.288 | 0.434 |
| 6 | Netherlands | 7.449 | 1.339 | 1.464 | 0.976 | 0.614 | 0.336 | 0.369 |
| 7 | Sweden | 7.353 | 1.322 | 1.433 | 0.986 | 0.650 | 0.273 | 0.442 |
| 8 | New Zealand | 7.300 | 1.242 | 1.487 | 1.008 | 0.647 | 0.326 | 0.461 |
| 9 | Austria | 7.294 | 1.317 | 1.437 | 1.001 | 0.603 | 0.256 | 0.281 |
| 10 | Luxembourg | 7.238 | 1.537 | 1.388 | 0.986 | 0.610 | 0.196 | 0.367 |
| 11 | Canada | 7.232 | 1.302 | 1.435 | 1.023 | 0.644 | 0.282 | 0.352 |
| 12 | Australia | 7.223 | 1.310 | 1.477 | 1.023 | 0.622 | 0.325 | 0.336 |
| 13 | United Kingdom | 7.165 | 1.273 | 1.458 | 0.976 | 0.525 | 0.373 | 0.323 |
| 14 | Israel | 7.129 | 1.216 | 1.403 | 1.008 | 0.421 | 0.267 | 0.100 |
| 15 | Costa Rica | 7.121 | 0.981 | 1.375 | 0.940 | 0.645 | 0.131 | 0.096 |

*Figure 5: Error message - data search UI*

## Charts & Use of Endpoints

In this section I will discuss how the data at each endpoint provided by the API was utilised in making two tables (one for the rankings section and one for factors) and a graph (for the trends section).

### Rankings (RankingsTable.jsx)



| Rank | Country | Score | Year |
|---|---|---|---|
| 1 | Finland | 7.809 | 2020 |
| 2 | Denmark | 7.646 | 2020 |
| 3 | Switzerland | 7.560 | 2020 |
| 4 | Iceland | 7.504 | 2020 |
| 5 | Norway | 7.488 | 2020 |
| 6 | Netherlands | 7.449 | 2020 |
| 7 | Sweden | 7.353 | 2020 |
| 8 | New Zealand | 7.300 | 2020 |
| 9 | Austria | 7.294 | 2020 |

*Figure 6: Rankings table*

The rankings section contains a table built with AG Grid (AG Grid, n.d.) and styled using AG Grid's theme builder (AG Grid, n.d.), which provides a downloadable custom CSS file that is saved into the project's styles directory and imported into the relevant component. For creating URLs, a local variable (apiURL) is defined and then has the relevant query parameters appended onto it based on the conditions of there being queries for one of either "country" or "year" or both. This URL is then fetched with the useFetch hook after which the returned data is fed into the rows of the table. Columns are defined in line with the AG Grid documentation. The component—and indeed all of the chart components—contains a useEffect hook which sets headers, including the API key and Content-Type, which as mentioned could have been better handled perhaps in the authentication context provider.

### Trends (TrendChart.jsx)

The trends area chart also makes use of the rankings endpoint and was an attempt at experimenting with charting. It shows one particular country's happiness "score" across the years that are represented in the data, also accessed from the rankings endpoint. The graph was built using Recharts' (Recharts, n.d.).

In this component, only one query for country is required. I had initially set it up to accept up to three countries, but this was making the years along the x axis render as many times as there were countries queried. At the time of writing this report, I'm still working out a solution for this, but for the moment the chart works sufficiently well, excepting the UI issues mentioned in the UI/Design section.

Another issue I had here initially was that the dates were arranging from most recent to oldest. This was solved easily using the "slice()" and "reverse()" array methods (Ibaba, 2022).

In hindsight, an area chart was perhaps not the right choice for representing this particular data, mostly because happiness scores—at least from what I've observed in this data—don't tend to change that dramatically (see Figure 7) from year to year. A more interesting-looking chart may have been possible if more years were represented in the data, or if multiple countries could be entered as I originally wanted to accomplish.

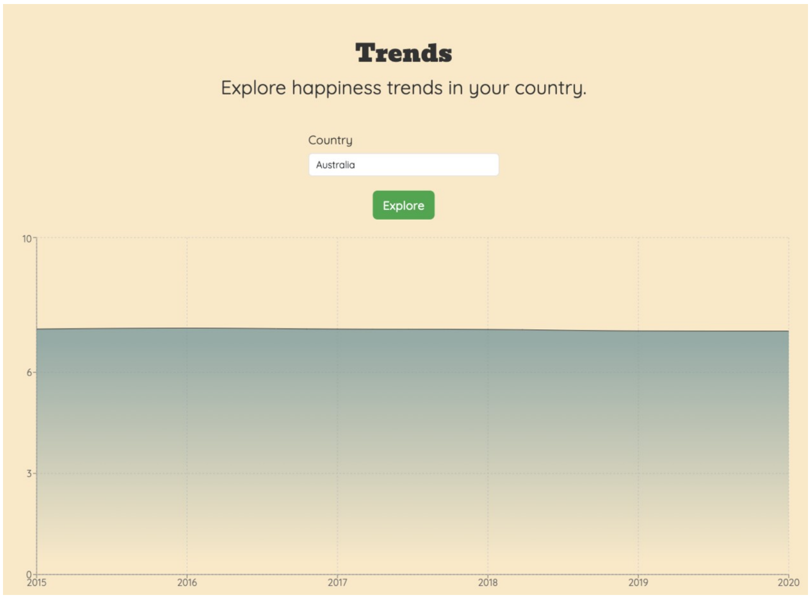
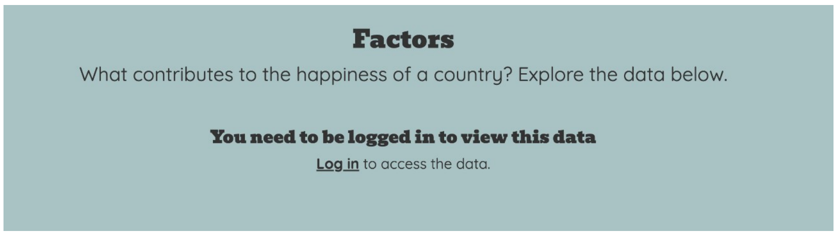


*Figure 7: A rather uninteresting area chart*

**<u>Factors (FactorsTable.jsx & FactorsMain.jsx)</u>**

AG Grid was put into service again for the factors section (which utilises the factors endpoint), but only after a long and ultimately untriumphant battle trying to create a radar chart with Recharts (Recharts, n.d.).

Since this section requires the user to be logged in to view the data, the FactorsMain.jsx component conditionally renders a message to the user. The token is retrieved with a function (useAuth) written in authProvider.jsx that provides access to the authentication context. A constant variable (isLoggedIn) is then set up with its value being a conditional check if the token exists in the current user state or not. Ternary conditional rendering (React, n.d.) is then used to render either the login message (*Figure 8*) or the table once the user is logged in.


Factors

What contributes to the happiness of a country? Explore the data below.

You need to be logged in to view this data

Log in to access the data.

*Figure 8: Conditionally rendered login component*

The only thing that otherwise differentiates the component logic in FactorsTable.jsx from RankingsTable.jsx is that I decided to utilise URLSearchParams API to construct the URL (Grafana Labs, n.d.). Reusing the approach from the rankings table would have worked fine, but might have become a little ungainly because of the extra parameters that need to be considered when constructing the URL.

## Concluding remarks

This was an interesting project that helped me understand important React concepts, like state, hooks and context. It also provided a good introduction to charting libraries.

In this report I've chosen only to focus on components and hooks that I managed to get working to some degree and that feature in the application itself. Some components however were abandoned for various reasons, like the radar chart mentioned above, or with the carousel that I couldn't get working on smaller screens no matter how much CSS wrangling I tried. These components can be found in the unused-components directory included as part of my assignment submission. The components were not abandoned for their lack of feasibility, but rather for deadline reasons. I look forward to returning to them in future work on this project once I have a better understanding of React.

# Appendix

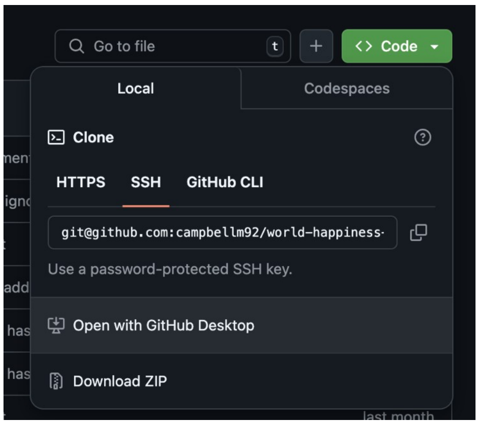
## User guide

This guide will take you through how to get the World Happiness Rankings application running on a local machine and how you can interact with the data. The guide assumes you have the following software installed and have a GitHub account:

- Visual Studio Code
- Node and Node Package Manager (npm)
- Git
- GitHub

### Step 1: Clone the repository

Visit the repository on GitHub. Once there, click on the **Code** button and copy the value in the SSH field. Open VS Code, then go to **File** > **Open Folder** and open the directory where you want to clone the project. After that, open a new terminal by navigating to **Terminal** > **New Terminal**. Go to the terminal and type git clone, followed by the SSH value you copied:



```
git@github.com:campbellm92/world-happiness-
```

Use a password-protected SSH key.

```
git clone git@github.com:campbellm92/world-happiness-rankings.git
```

Once you've done this, navigate to the project:

```
cd world-happiness-rankings
```

### Step 2: Install dependencies and run the local server

Again in the terminal, install the dependencies necessary to run the app:

```
npm install
```

Then run `npm start` to get the server up and running.

You can now use the app by visiting http://localhost:5173/ in your browser.

## Exploring the app

The app is divided into three sections: rankings (the first section), trends (the second section) and factors (the third section). Data can be explored in the first and second sections without an account, but for the third section, you'll need to sign up. Since this is not a real-life application, feel free to just make an account with a fake email address, like example99@email.com. Once logged in, you can explore the factors data as well.

## References

- Gayan, K. (November 28, 2023). *Beyond 100vh: Simplifying Responsive Design with CSS Viewport Units.* Medium. https://medium.com/@kanushka/beyond-100vh-simplifying-responsive-design-with-css-viewport-units-6b9f00c8e444
- W3 Schools. (n.d.). *CSS Media Queries - Examples.* W3 Schools. https://www.w3schools.com/css/css3_mediaqueries_ex.asp
- Adobe. (2024). *Free logo maker: Design custom logos*. Adobe Express. https://www.adobe.com/express/create/logo
- Geco colorpick. (n.d.). https://chromewebstore.google.com/detail/color-picker-eyedropper-g/eokjikchkppnkdipbiggnmlkahcdkikp?pli=1
- Google Fonts. (n.d.) https://fonts.google.com/
- Senadheera, P. (December 18, 2023). *Using !important in CSS is generally considered bad practice*. LinkedIn. https://www.linkedin.com/pulse/using-important-css-generally-considered-bad-practice-senadheera-4rpbc/
- Olanrewaju, T. (August 29, 2024). *Design-first vs Logic-first Approach – How Should You Start Your Front-end Projects?* freeCodeCamp. https://www.freecodecamp.org/news/design-first-vs-logic-first-approach/#:~:text=The%20logic%2Dfirst%20approach%20focuses,data%20as%20regards%20front%2Dend.
- srinam. (February 22, 2024). *Don't repeat yourself(DRY) in Software Development.* Geeks for Geeks https://www.geeksforgeeks.org/dont-repeat-yourselfdry-in-software-development/
- Dhanani, H. (January 5, 2024). *A Definitive Guide to React Architecture Patterns.* TatvaSoft. https://www.etatvasoft.com/blog/react-architecture-patterns/
- React. (n.d.). *Context*. Meta Platforms. https://legacy.reactjs.org/docs/context.html
- Arya, S. (May 28, 2023). *JWT Authentication in React with react-router.* DEV. https://dev.to/sanjayttg/jwt-authentication-in-react-with-react-router-1d03?utm_source=reactdigest&utm_medium=&utm_campaign=1655
- Joy, C. (March 31, 2023). *Building a Custom Fetch Hook in React.* Open Replay. https://blog.openreplay.com/building-a-custom-fetch-hook-in-react/#:~:text=Building%20the%20custom%20Fetch%20Hook&text=You%20can%20name%20the%20hook,are%20fetching%20the%20data%20from.
- Adegbuyi, A. (July 13, 2020). *How To Create A Custom React Hook To Fetch And Cache Data.* Smashing Magazine. https://www.smashingmagazine.com/2020/07/custom-react-hook-fetch-cache-data/
- Chamochumbi, J. (July 16, 2021). *Using React to understand Abort Controllers.* Medium. https://medium.com/@icjoseph/using-react-to-understand-abort-controllers-eb10654485df
- Labaran, A. (April 24, 2024). *Cancel Asynchronous React App Requests with AbortController.* The New Stack. https://thenewstack.io/cancel-asynchronous-react-app-requests-with-abortcontroller/
- W3 Schools. (n.d.). *React ES6 Destructuring.* W3 Schools. https://www.w3schools.com/react/react_es6_destructuring.asp
- Odyntsov, Y. (April 26, 2024). *Developer's Guide on JavaScript Email Validation: Client & Server-Side Methods Covered.* Mailtrap. https://mailtrap.io/blog/javascript-email-validation/
- Ibrahim, R. (20 June, 2022). *Write Regex Pattern for Password Validation Like a Pro.* DEV. https://dev.to/rasaf_ibrahim/write-regex-password-validation-like-a-pro-5175
- Fulness, O. (October 29, 2023). *How To Create a Custom Hook to Handle Inputs in React.* Medium. https://medium.com/@ojebiyifulness/how-to-create-a-custom-hook-to-handle-inputs-in-react-cf2ab177f70d
- AG Grid. (n.d.). *Quick Start.* AG Grid. https://www.ag-grid.com/react-data-grid/getting-started/
- AG Grid. (n.d.). *Theme Builder*. AG Grid. https://www.ag-grid.com/theme-builder/
- Recharts. (n.d.). *AreaChart*. Recharts. https://recharts.org/en-US/api

- Ibaba, T. (May 25, 2022). *How to Reverse an Array without Modifying in JavaScript.* Medium https://javascript.plainenglish.io/javascript-reverse-array-without-modifying-dac063623490
- Recharts. (n.d.). *SimpleRadarChart*. Recharts. https://recharts.org/en-US/examples/SimpleRadarChart
- Grafana Labs. (n.d.). *URLs with query parameters.* Grafana Labs. https://grafana.com/docs/k6/latest/examples/url-query-parameters/
- React. (n.d.). *Conditional Rendering*. React. https://react.dev/learn/conditional-rendering