Big 5 Personality Quiz

Demo Video:
https://www.loom.com/share/c5e0975ed3a044be84b09565e4b72fff?sid=a8953f1c-95ab-4fba-81a1-374bbeae3966

Tech Stack:
Languages & Frameworks: *TypeScript, React*
Backend: *Prisma, Next.js, SQLiteDB*
Libraries: *Material UI, Emotion*

Getting Started

Run the following command in your terminal to install the required dependencies for this project.

```
npm install
npm ci
```

Start the development server.

```
npm run dev
```

If you would like to add seed data, add the following line to the `package.json` file:

```
"type": "module",
```

Then, run the following command in your terminal to run the seed script.

```
npm run seed
```

Overview

File Structure

The application uses Next.js for routing.

Personality Quiz Components

src/pages/index.tsx

- top level file that renders the `Survey` Component as well as the `AppMenu` in the top right hand corner for navigation

src/pages/Survey.tsx

- handles the routing to the results page when the form is submitted
- displays the progress bar
- renders the `QuestionSet` component

src/pages/QuestionSet.tsx

- Renders the entire list of `SurveyQuestion` components and handles logic to display one question at a time and store the values of the form data that is selected
- conditionally renders submit when the last question is completed
- handles the logic for getting the next and previous question and controls timing for the transition animations

src/pages/SurveyQuestion.tsx

- Renders the question itself and the radio group buttons for that question (lowest level component in this tree)
- The animation logic for the buttons and transitions is being applied here

Results Page Components

src/pages/results/index.tsx

- receives form data from the `Survey` component and calculates the score for each personality trait and displays it on the page
- once the scores are calculated, it sends a post request to make a `User` object and a post request to make a `PersonalityResult` object in the database
- renders the `PersonalityChart` component which displays the bar chart breakdown of your score
- also renders a particle animation for the background as a fun touch!

Prisma Files

- holds the migrations, the SQLite db `dev.db` and database schema
- There are currently two models in the database:
  - `User`
    - only required field is id (which is auto-set by prisma)
    - intended to store demographic data, but the frontend is currently not asking for that data, so all users are created with only the `id` field when the personality quiz is submitted
  - `PersonalityResult`
    - all fields are required
    - has a mapping to the user that submitted the quiz
    - each score is stored as a number out of 100

```
model User {
  id                Int                @id @default(autoincrement())
  firstName         String?            @default("")
  lastName          String?            @default("")
  age               Int?               @default(0)
  zipCode           String?            @default("")
  race          String?            @default("")
  PersonalityResult PersonalityResult[]
}

model PersonalityResult {
  id                    Int  @id @default(autoincrement())
  user                  User @relation(fields: [userId], references: [id])
  userId                Int
  OpennessScore         Int
  ConscientiousnessScore Int
  ExtraversionScore     Int
  AgreeablenessScore    Int
  NeuroticismScore      Int
}
```

Data Files

src/data/questions.json

- list of 50 question objects, each structured in the following way:

```
{
    "id": 0,
    "text": "I am the life of the party",
    "sign": "plus",
    "trait": "E"
  },
```

- `sign` represents whether or not the selected value for this question needs to be added or subtracted to the final score

src/data/personalityDescriptions.json

- list of mappings of trait to descriptions for the corresponding low, moderate, and high scores

example entry for 'O'

```
"O": {
    "low": "Individuals with low openness tend to prefer routine and familiarity. They may be more conventional and traditional in their thinking and behavior. They might be less interested in new ideas, art, or unconventional experiences.",
    "moderate": "People with moderate scores are somewhat open to new experiences but also value routine. They strike a balance between tradition and novelty. They can appreciate creativity and originality but may not actively seek out highly unconventional experiences.",
    "high": "Individuals with high openness scores are very open to new experiences and ideas. They are often curious, imaginative, and willing to explore the unknown. They may have a strong interest in the arts, nature, and intellectual pursuits. They tend to be more creative and open-minded."
  },
```

Scripts

src/scripts/seed.ts

- generates fake data using `Faker` and makes post requests to the database
- generates data for both `User` and `PersonalityResult` objects

Design Decisions

Choosing the tech stack

Languages & Frameworks: *TypeScript, React*
Backend: *Prisma, Next.js, SQLiteDB*
Libraries: *Material UI, Emotion*

When it came to choosing the tech stack, there were two primary things I considered:

- How familiar am I already with it? (for quick iteration)
  - If it's something I'm less familiar with, will it help more than it takes to learn? (given time limit)
- How does it fit into this use case?
  - Even if it's better for the long term, is it worth implementing in this size project?

React & Next.js- Choosing React was straightforward because I am quite familiar with it, and it is industry-standard for web development. Next.js is the natural backend framework that goes along with React for routing, SSR, SSG, etc.

Typescript vs. Javascript - Even though Typescript is a bit more tedious to write because you have to explicitly type every variable, I think it was worth choosing because it caught my errors in passing wrong data types before I could have myself, and saved me time debugging.

Prisma vs. SQL - I wanted an ORM so I could create models and API requests more easily, allowing me to focus more on application development rather than writing complex SQL queries. It has automatic query generation and an integration with Typescript for type safety to ensure more robust and error-free code. To clarify, Prisma still sits on top of a SQL database.

Material UI vs Tailwind - Tailwind makes it easy to fit styles within the same HTML tag, which makes it straightforward to write and debug. It also makes it easier to build custom components. However, for this project I didn't require any fancy custom components, so I wanted to use Material UI for their pre-built components that allow you to pass in more props and functionality than a normal CSS component does.

Major Trade-offs I considered

Airtable as a CMS vs storing JSON in-memory
When it came to thinking about how to store the question data itself, my first thought was to use a content management system like Airtable. Airtable has a nice UI that looks like Google Sheets and Excel, that allows you to make edits to the data in real time. This would allow me or the admin to easily make changes to the question set without directly editing any code. For smaller applications, Airtable can also serve as a database that sits on top of SQL as well. It has the same Google Sheets-esque interface, that makes it easy to edit data.

However, after thinking about it some more, I realized that for an external application, it might be best to use an internal data store to avoid any possible security concerns. Using a JSON file is also easier to implement, and given the short and fixed amount of questions, it's ok to store the data in memory. If we wanted to scale the application in the future to serve hundreds of thousands of requests, Airtable would not suffice as a database either.

Deciding what tables to store
At the current moment, the database is only being read when the admin dashboard graphics are being displayed. Because there are no hard requirements on what type of analytics we want from the responses, there are varying degrees of information we could possibly show.

The most basic table we need is the `PersonalityResults` table which stores each response as the 5 scores they received. We technically don't need to map it to a

specific user if we are not running any demographic analysis on them, but I wanted to add the User foreign key in case we want to add that functionality.

Another table I considered storing is a `Responses` table that stores all the value mappings of the actual question itself to the value it received by a user. This would allow us to run more complex analysis on what responses tend to receive the same answer no matter the overall result, allowing us to pick out potentially unhelpful questions. For this first iteration of the project, I decided to not store this data, as it wasn't necessary for doing basic analysis on the results. Storing this would mean storing 50 fields for each response, and that could get large very quickly. This could affect performance of the website and potentially require us to spend more scaling our databases in the future.

Best UI for displaying questions

There were 3 ways the questions could be displayed:

- showing whole list of questions in one page
  - I didn't want a static website because that would make it less interesting to the user, and with 50 questions, they might lose interest and not finish.
- break down list into subsets and do pagination
  - Pagination makes it more interesting than showing all the questions at once, so it was a close tie between this and displaying one question at a time
- showing one question at a time
  - What sold me was looking at the really clean interface of Typeform's forms. They're one of the biggest form providers out there, and I looked through many of their templates and it looks like all of them display one question at a time. Seeing only one question at a time, allows the user to focus on the question at hand instead of rushing through the form or thinking about the other questions, which is why I ultimately went with this choice.
    - **Sub-decision:** save answers to database as we go or submit at the end?
      - would improve performance if we didn't have to make an API request for every question
      - decided not to save the values for each question in a separate answers table (explained above)
      - one pro for submitting the response for every question is that if there was a server failure or any failure at all, the user could go back to their

place, and we could still have partial responses
    - we could still save the user's place without sending updates to the database though

Process Breakdown

Research

- What questions are generally asked? How are questions scored? What do scores mean?
    - I realized that there is no consistent set across the board and different quizzes ask varying number of questions as well. Because of this it made it difficult figuring out the exact subset of questions I would need for an accurate score
    - I ended up finding a manual paper version of the quiz, that allows users to score themselves, which helped me figure out how the scores are calculated
        - https://openpsychometrics.org/printable/big-five-personality-test.pdf
        - data on what is considered a low, moderate, and high score, and what each of them mean: https://warwick.ac.uk/services/dc/phdlife/wellbeing/potentialadvantage/personality_and_wellbeing_results/

System Design

- planning out what tables to store
- SQL vs NoSQL
    - Because we are running analytics on all the responses, we need a relationship between various elements of the data
        - e.g. User age & their personality results, User zip code & their results, Change to personality result for a specific user over time, etc.
    - There are also a fixed number of questions, so there is no need to use NoSQL to handle unstructured data
    - NoSQL scales better because we don't have to worry about query optimization, but our application requires relationships between different

parts of the data

Process Planning

- What features are required & what are stretch goals? The following are required:
  - displaying questions
  - displaying results
  - admin dashboard for analytics on results
  - making API endpoints for creating users and results and getting all results

I wanted to paste my notes here I made when I first looked at the prompt and was trying to break down the approach into steps:

1. Json file with hardcoded questions & do mapping or airtable & console log to make sure it works
2. Implement react components for basic form (don't do saving to db stuff yet) only get frontend to work
   a. Full page (loads data from Airtable and passes down as props, what's current question) & navigation buttons (own component), has callback, passing to child, list mapping
      i. Inside on submit callback, submit data to db, change page
   b. Rotating card with question (only functional component, take in question and current value and renders) (technically only display only) (can make dumb form if rotating card hard)
      i. When want to save answer to db, calls the callback
      ii. Scrollbar in here, with boundaries (optional)
   c. Landing page (optional)
   d. Finish page (could add animation here)
3. Define db tables
   a. make migrations
4. Update frontend to submit data
   a. build up list of responses, and then submit to database once completed
   b. Fetch data from data base
   c. Render data in react component
   d. add fancy visualizations like plotly
5. Navbar component

6. Admin dashboard
7. Add unit tests (if have extra time)
8. Separate page for admin password (optional)

Future Improvements

If I had more time or were scaling the app out, I'd love to implement the following features:

- unit tests
  - test routing, and animation and rendering logic on frontend
  - make sure api requests return back right information
- more visualizations for admin dashboard
- saving a separate `Answers` table to store all answer value responses
- user authentication
  - users would be able to save their progress and come back to it later, or retake the quiz over time and see analytics on changes over time
  - admin dashboard should only be available to admins
- "Are you sure you want to leave" prompt?
- saving where you left off in the survey
- landing page
- add data validation to api requests before making the request (using joi or similar)
- survey asking for demographic information that we can store to make better analytics
- spend more time on UI enhancements
  - button to advance to latest question
- hosting on separate server