Players_Nexus

Route: `/`

Implement `Infinite Scrolling`:

1. The **fetchMoreData** function is called when the user scrolls to the bottom of the page.

2. Inside the **useEffect** hook with the dependency on **isLoading**, there's a scroll event listener attached to the window. When triggered, it checks if the user has scrolled to the bottom of the page and if **isLoading** is false.

3. If conditions are met, **fetchMoreData** is called, which in turn calls **fetchGamesByPage** with the next page number.

4. **fetchGamesByPage** fetches additional data based on the current page number and updates the state with the new game list, incrementing the page count.

This process repeats as long as there are more pages to load (**currentPage < totalPages**) and the user continues scrolling to the bottom of the page. Therefore, it achieves an infinite loading effect by dynamically loading more data as needed.

Route: `/get-started`

For Avatar `Choosing, Proview & Store at Firebase Storage`:

1. **Image Preview**:

    - The **avatarPreview** state holds the preview of the selected image file. It's displayed using the **Image** component from Next.js, which shows the image preview as a rounded avatar.

    - When a user selects an image file using the file input field, the selected file's data is read using **FileReader**, and its result (base64 encoded string) is set to the **avatarPreview** state. This allows users to see a preview of the image they've selected before uploading it.

2. **Choosing Image**:

    - The input field with **type="file"** allows users to choose an image file from their device.

    - The **handleAvatarChange** function is triggered when the file input field's value changes (i.e., when a user selects an image file).

    - It updates the **avatar** state with the selected image file and sets the **avatarPreview** state with a base64 encoded representation of the selected image for preview purposes.

3. **Storing Image in Firebase Storage**:

    - Upon clicking the "Finish" button, the **handleFinishRegistration** function is called.

    - It first checks if an avatar file has been selected. If not, it logs an error message.

- If an avatar file has been selected, it uploads the file to Firebase Storage using **uploadBytesResumable**.
- The upload progress and completion are monitored using event listeners on the **uploadTask**.
- Upon completion of the upload, the download URL of the uploaded avatar is retrieved using **getDownloadURL**.
- The download URL is then stored in Firestore under the user's document along with other information like the update timestamp (**updatedAt**).

Overall, this functionality allows users to select an image file, preview it before uploading, and store it in Firebase Storage, making it accessible via a download URL for use in their profile.

Route: `/profile`

Display DP, Discord Profile Link & Username. Including Edit Profile Button. Here's breakdown of these segments:

1. **Profile.tsx**:
    - This component is responsible for displaying a user's profile information, including their avatar, username, bio, and various categories such as ratings, reviews, and custom lists.
    - It retrieves user data using the **useGetUser** hook and displays it once loaded. While waiting for the user data, it displays a loading spinner.
    - Users can switch between different categories (ratings, reviews, and custom lists) using tabs.
    - It fetches user-specific lists from Firestore and displays them as additional categories.
    - Each category's content (ratings, reviews, or custom lists) is rendered based on the selected category.
2. **EditProfile.tsx**:
    - This component allows users to edit their profile information, including their avatar, username, and bio.
    - It retrieves the current user's information using the **useGetUser** hook and initializes state variables accordingly.
    - Users can upload a new avatar image file, change their username, and update their bio.
    - Upon uploading a new avatar, the component uploads the image to Firebase Storage and updates the user's document in Firestore with the new avatar URL.
    - The component provides validation for the username to ensure it's unique. If the chosen username already exists, it displays a warning message.

    - After editing profile information, users can click the "Proceed" button to save changes, which triggers the update of the user's document in Firestore.

In summary, these components handle the display and editing of user profile information, including avatar, username, bio, and additional user-specific data like ratings, reviews, and custom lists. They interact with Firebase Storage and Firestore for storing and retrieving user data.

- **Props**:
    - **title**: A string representing the title to be displayed.
    - **onClick**: A function to be called when the heading is clicked.
    - **isActive**: A boolean indicating whether the heading is currently active or not.
- **Component Behavior**:
    - The component renders an **<h2>** element with the specified **title**.
    - It applies styles and behavior based on the provided props:
        - Sets the cursor to 'pointer' to indicate interactivity.
        - Adjusts font weight and text decoration based on whether the heading is active or not.
        - Defines transitions for smoother visual changes.
        - Sets padding, border radius, and background color for styling.
    - The **className** attribute is used to apply additional tailwind CSS classes for hover effects.
- **Event Handling**:
    - When the heading is clicked, the **onClick** function provided via props is called.
- **Rendering**:
    - The component renders the **title** within the **<h2>** element.

Overall, the **NewsHeading** component provides a reusable and customizable way to render interactive headings with dynamic styles and behavior based on their active state.

The **UserRatings** component is responsible for displaying user ratings for games. Here's a breakdown of its functionality:

- **Props**:
    - **email**: A string or null representing the user's email.
- **State**:
    - **listRatings**: An array of objects representing the user's ratings for different games.

    - **filterListRatings**: An array of objects representing filtered user ratings based on selected criteria.
    - **showModal**: A boolean state to control the visibility of the rating modification modal.
    - **rating**: An object representing the current selected rating for modification.
- **Functions**:
    - **fetchData**: Fetches user ratings data from Firestore based on the provided email.
    - **navigateToGame**: Navigates to the game page when a game is clicked.
    - **deleteListItem**: Deletes a rating item from Firestore.
    - Handlers for changing sorting criteria (**handleOverallChange**, **handleGameplayChange**, etc.).
- **Rendering**:
    - Dropdowns for selecting sorting criteria (Overall, Story, Gameplay, Graphics, Audio, Multiplayer).
    - Conditional rendering based on the presence of ratings data:
        - If no ratings are found, it displays a message "No Ratings Found!".
        - If ratings are found, it displays them in a grid format.
    - Each rating item includes:
        - Game image (or a placeholder if not available).
        - Game name.
        - Star ratings for different aspects (Overall, Story, Gameplay, Graphics, Audio, Multiplayer).
        - Option to delete the rating (if not on the friends page).
        - Clicking on a rating opens a modal for modifying the rating.
- **Modal**:
    - A modal (**ModifyRatings**) is displayed when a rating is clicked for modification.
    - The modal allows the user to modify the individual ratings for different aspects of the game.

Overall, the **UserRatings** component provides a user-friendly interface for viewing and modifying game ratings, with options for sorting and deleting ratings.

The **UserReviews** component is responsible for displaying user reviews for games. Here's a breakdown of its functionality:

- **Props**:
    - **email**: A string or null representing the user's email.
- **State**:
    - **listReviews**: An array of objects representing the user's reviews for different games.
    - **showModal**: A boolean state to control the visibility of the review modification modal.
    - **review**: An object representing the current selected review for modification.
    - **sortOption**: A string representing the selected sorting option for reviews.
- **Functions**:
    - **fetchData**: Fetches user reviews data from Firestore based on the provided email.
    - **navigateToGame**: Navigates to the game page when a game is clicked.
    - **handleSortChange**: Handles the change in sorting option for reviews.
    - **reviewStats**: Analyzes the sentiment of a review text and returns an emoji representing the sentiment.
    - **deleteListItem**: Deletes a review item from Firestore.
- **Rendering**:
    - Dropdown for selecting sorting criteria (Newest, Oldest).
    - Conditional rendering based on the presence of reviews data:
        - If no reviews are found, it displays a message "No Reviews Found!".
        - If reviews are found, it displays them in a grid format.
    - Each review item includes:
        - Game image (or a placeholder if not available).
        - Game name.
        - Star rating.
        - Review title and text.
        - Sentiment analysis emoji.
        - Option to delete the review (if not on the friends page).
        - Clicking on a review opens a modal for modifying the review.
- **Modal**:
    - A modal (**ModifyReviews**) is displayed when a review is clicked for modification.

    - The modal allows the user to modify the review title and text.

Overall, the **UserReviews** component provides a user-friendly interface for viewing and modifying game reviews, with options for sorting and deleting reviews. It also includes sentiment analysis to give users a quick overview of the review's sentiment.

The **ListCards** component is responsible for displaying a grid of game cards based on the user's selected category. Here's an overview of its functionality:

- **Props**:
    - **currentCategory**: A string representing the current category selected by the user.
    - **email**: A string or null representing the user's email.
- **State**:
    - **lists**: An array of objects representing the lists of games associated with the user.
- **Functions**:
    - **fetchData**: Fetches the lists of games associated with the user from Firestore.
    - **navigateToGame**: Navigates to the game page when a game card is clicked.
    - **deleteListItem**: Deletes a game from the list and updates the state accordingly.
- **Rendering**:
    - Conditional rendering based on the presence of game lists:
        - If no games are found in the selected category, it displays a message "No Games Found!".
        - If games are found, it displays them in a grid layout.
    - Each game card includes:
        - Game image (or a placeholder if not available).
        - Game name.
        - Option to delete the game from the list (if not on the friends page).
- **Interaction**:
    - Clicking on a game card navigates the user to the corresponding game page.
    - Clicking the delete button removes the game from the list and triggers a refresh of the displayed games.

Overall, the **ListCards** component provides a visually appealing and interactive interface for viewing and managing lists of games based on the user's selected category. It integrates with Firestore to fetch and update game lists dynamically.

Route: `/news`

Fix display images in News cards:

In the **News** component, the image functionality involves displaying images associated with news articles fetched from the GNews API. Let's break down how the image is shown and how the display image functionality works:

1. **Image Display in NewsCard Component**:
    - Each news article is represented by a **NewsCard** component.
    - The **NewsCard** component receives an **article** prop, which contains information about the news article, including its **title**, **url**, and **image**.
    - The **image** property in the **article** object represents the URL of the image associated with the news article.
    - To display the image, the **NewsCard** component uses the **Image** component from Next.js.
2. The Image component is passed the following props:
    - src: The URL of the image to be displayed.
    - alt: The alternative text for the image.
    - layout: Defines how the image is laid out. In this case, it's set to "fill", which allows the image to fill its container.
    - objectFit: Defines how the image should be resized to fit its container. In this case, it's set to "cover", which maintains the aspect ratio while covering the container's dimensions.
3. Image Functionality:
    - When the News component fetches news articles from the GNews API, each article object includes a URL pointing to an image associated with that article.
    - This URL is stored in the image property of the article object.
    - When rendering each NewsCard component, the Image component from Next.js is used to display the image associated with the news article.
    - The Image component handles loading the image efficiently, optimizing performance by lazy-loading and providing automatic resizing and optimization based on the device's screen size.
    - Overall, the image functionality in the News component effectively displays images associated with news articles using the Next.js Image component, providing a visually appealing user experience.

How the `Search Query` works?

1. **Input Field**:
    - The search engine starts with an input field where users can type their search queries.
    - This input field is implemented using an HTML **<input>** element of type **search**.
    - The value entered by the user is stored in the **searchQuery** state variable using the **useState** hook.
2. **Search Button**:
    - Next to the input field, there is a search button that users can click to initiate the search.
    - When the search button is clicked, it triggers the **handleSearch** function.
3. **Search Functionality** (**handleSearch** function):
    - The **handleSearch** function is responsible for initiating the search process.
    - When called, this function first checks if the **searchQuery** is empty. If it is, the function returns early to prevent searching with an empty query.
    - If the **searchQuery** is not empty, the function sets the **isSearchActive** state to **true**. This state variable is used to indicate whether a search is currently active.
    - The function then calls the **fetchGamingNews** function with the **searchQuery** as the query parameter. This function is responsible for fetching news articles based on the search query.
    - The **fetchGamingNews** function is an asynchronous function that makes a request to the GNews API to fetch news articles matching the specified query.
    - Once the API responds with the search results, the function updates the **articles** state variable with the retrieved articles using the **setArticles** function.
    - If there is an error during the API request, the function logs the error to the console.
4. **Displaying Search Results**:
    - After the search is initiated, the **articles** state variable is updated with the search results.
    - The **News** component re-renders, and the **articles** are mapped over to display each news article using the **NewsCard** component.
    - The **NewsCard** component displays the title, image, and other information about each news article.
    - Users can then view the search results and click on individual articles to read more about them.

In summary, the search engine functionality allows users to enter search queries, initiates a search request to the GNews API, retrieves search results, and displays them to the user in the form of news articles matching the search query.

Route: `/friends`

Describe How Friends & Activities of Friends work and their workflows:

1. **Friends.tsx**:
   - This component represents the main page for managing friends and activities.
   - It consists of two tabs: "Friends" and "Activities".
   - Users can switch between tabs to view their friends or activities.
   - The **useSession** hook from NextAuth is used to check if the user is authenticated. If not, it redirects the user to the sign-in page.
   - State variable **tab** is used to manage the active tab.
   - Depending on the active tab, either the **MyFriends** or **Activities** component is rendered.
2. **MyFriends.tsx**:
   - This component displays a list of users (potential friends) that the current user can add as friends.
   - It allows users to search for friends by their usernames.
   - Users can add friends by clicking the "Add" button next to each user.
   - Once friends are added, they are displayed in another section along with options to view their profile or delete them as friends.
   - State variables are used to manage the list of friends, search query, and user's existing friends.
   - Firebase Firestore queries are used to fetch and manage user data.
3. **Activities.tsx**:
   - This component displays activities such as game reviews and feedback provided by the user's friends.
   - It fetches data related to feedback and reviews from Firebase Firestore.
   - The user's friends' information is retrieved, and based on that, feedback and reviews are displayed.
   - Each activity item includes details such as the friend's avatar, game information, ratings, review, and timestamp.

- The **sentiment** library is used to analyze the sentiment of the review text and display an appropriate emoji indicating the sentiment.
- Functions are defined to format timestamps and calculate the time difference between the current time and the time when the activity was created.

Overall, these components together provide functionality for managing friends, adding new friends, and viewing activities such as game reviews and feedback provided by friends. They interact with Firebase Firestore to fetch and update data related to users, friends, and activities.

Route: `friends/{{email}}`

Identify friend's activities as a whole:

1. **Initialization**:
    - The component imports necessary dependencies like React, Next.js router, Firebase Firestore utilities, and session management functionalities.
2. **State Initialization**:
    - State variables are initialized to manage user data, current category (whether it's ratings, reviews, or custom lists), search state, and user-specific lists fetched from Firestore.
3. **Data Fetching**:
    - Two **useEffect** hooks are used to fetch user data and user-specific lists from Firestore based on the provided user email (**uemail**).
    - The first **useEffect** fetches the user's details from the Firestore collection based on the provided email.
    - The second **useEffect** fetches the user's custom lists from the Firestore collection based on the provided email.
4. **Loading State**:
    - While the user data is being fetched, a loading spinner (**LoadingCircle**) is displayed to indicate that data is loading.
5. **Category Selection**:
    - Users can switch between different categories (Ratings, Reviews, and Custom Lists) using tabs.
    - The **handleCategoryChange** function is called when a category tab is clicked to update the current category state.
6. **Rendering**:
    - The component renders the user's avatar, username, and bio (if available) at the top.

- It renders tabs for categories (Ratings, Reviews) and custom lists.
- Depending on the selected category, it renders different components:
    - For "Ratings" category, it renders the **UserRatings** component passing the user's email.
    - For "Reviews" category, it renders the **UserReviews** component passing the user's email.
    - For custom lists, it renders the **ListCards** component passing the user's email and the current category.

7. **External Link**:
    - If the user has a bio, it renders a link to the bio page. Clicking on it will redirect the user to the external bio page.

Overall, **UEmail.tsx** provides a user-specific view where users can see their ratings, reviews, and custom lists based on their email. It leverages Firebase Firestore for data storage and retrieval and Next.js router for navigation.

Responsive `Navbar`

That used to display on various devices:

A responsive navbar is a navigation menu that adjusts its layout and appearance based on the screen size of the device being used. Here's a brief explanation of the components and behavior of a responsive navbar:

1. **Navigation Links**: These are the menu items or links that users can click on to navigate through different sections of a website. In a responsive navbar, these links may be displayed horizontally or vertically depending on the screen size.
2. **Hamburger Menu Icon**: On smaller screens, such as mobile devices, the navigation links may be hidden behind a hamburger menu icon. Clicking on this icon reveals the navigation links in a dropdown menu, allowing users to access them easily without cluttering the screen.
3. **Media Queries**: CSS media queries are used to define different styles for different screen sizes. For example, the navbar might be displayed horizontally on larger screens and vertically (as a hamburger menu) on smaller screens.
4. **JavaScript Functionality**: JavaScript is often used to add interactivity to the responsive navbar. For instance, a JavaScript function may be used to toggle the visibility of the navigation links when the hamburger menu icon is clicked.

Overall, a responsive navbar ensures that website navigation remains accessible and user-friendly across various devices and screen sizes, providing a seamless user experience regardless of the device being used.