ANIAM WOLFF

TECHNICAL DESIGN

MAIN FEATURES

LIST OF PLAYERS

Our list uses data from the external API provided and allows us to display the relevant data to the user.
The list should Include pagination and allow users to switch pages, and search players by name.

The users can select players as “favorite” from found results.

Goals

- Implement Search and pagination.
- Allow favorite selection.
- Responsive and intuative.
- Optimise API calls.

LIST OF FAVORITE PLAYERS

List that displays all players the user selected as favorite,

the users can change each players background color and remove him from the list.

Goals

Implement Background color picking.

- Allow favorite toggle.
- Optimise component rendering.
- Save data between reRenders

API INTERACTION

In order to achieve the desired functionality of our features we implement usage of our given API to retrieve desired players data in our backend.

API Players

Implemented usage of our given API to retrieve desired players data in our backend.

Build: /api/v1/players

Query Params:

1. page - the page we want to receive data about, default is 1 as first page.
2. search - The name(or part of it) we want to search in API.
3. per_page - includes the player amount we want to show on each search, default 25.

Received interface:

```
interface Response
 data: Player[];
 meta: Meta;
```

```
interface Meta
 total_pages:
number;
 current_page:
number;
 next_page: number;
 per_page: number;
 total_count:
```

```
interface Player
 id: number;
 first_name: string;
 height_feet?: number;
 height_inches?: number;
 last_name: string;
 position: string;
 team: Team;
```

```
interface Team {
 id: number;
 abbreviation: string;
 city: string;
 conference: string;
 division: string;
 full_name: string;
 name: string;
```

Get Players

To receive only the relevant data, and optimize requests we use additional steps to change data received from the external API

Build: /api/players

Query Params:

1. page - the page we want to receive data about, default is 1 as first page.
2. search - The name(or part of it) we want to search in API.

Received interface:

```
interface Metadata
 totalPages: number;
 currentPage: number;
 nextPage: number;
 perPage: number;
 total_count:number;
 number;
```

```
interface Player
 id: string;
 fullname: string;
 height: number | string;
 position: string;
 team: string;
```

```
interface PlayerResponse
 data: Record<string, Player>;
 metadata: Metadata;
```

Flow:

1. Client side:
   When user changes pagination page, or search input changes (and valid) a custom hook that triggers a request to backend.
2. Backend:
   We check if our query data is cached using Redis and prevent unnecessary API reqs.
   If data not in Redis we proceed the API request and reformat the result to achieve desired output.
   After formatting data we cache the data in Redis and return data to client.
3. Client - receives data and updates accordingly.

STATE MANAGEMENT

To keep a clean and predictable and scalable application I decided to use recoil for managing different states. Complex/common state logic is in a fitting custom hook.

Player Atoms:

playersState:
Holds the current players page data.

favoritePlayersState:
Holds the current favorite players data.
Uses local storage to keep data up to date between refreshes.

App State Atoms:

loaderState:
if a request has been made we set it to true in order to display loading state for user.

pageState:
Includes the wanted pagination page and used to trigger
Uses local storage to keep data up to date between refreshes.