KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – lecture 6

Web, API and React Query in React Native

Mateusz Pawełkiewicz

1.10.2025

1. Basics of Working with APIs in React Native

In React Native applications, communication with external APIs occurs similarly to web applications. Standard methods such as the **Fetch API** and popular libraries like **Axios** are available. In this section, we will discuss both approaches, including their configuration options (headers, interceptors, timeouts), request cancellation, and compare their advantages and disadvantages.

1.1 Fetch API – Built-in Network Requests

The **Fetch API** is a built-in JavaScript method for making HTTP requests. In the React Native environment, fetch is available globally (as a polyfill). Its syntax is similar to that in the browser:

JavaScript

```
// Example GET request using fetch:
try {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) {
    // Checking the status code (fetch does not throw an error for HTTP errors)
    throw new Error(`HTTP Error: ${response.status}`);
  }
  const data = await response.json();  // Manual JSON parsing
  console.log('Received data:', data);
} catch (error) {
  console.error('Network or request error:', error);
}
```

Characteristics of Fetch API:

- **No global configuration** – There is no built-in mechanism for setting a base URL or default headers for all requests. It is necessary to provide options every time or wrap fetch in your own helper function.
- **Response handling** – Fetch does not throw an exception for HTTP error responses (e.g., 404, 500). You must manually check response.ok or response.status. Fetch will only throw an exception in case of a network error (no connection, DNS issues, etc.), but an incorrect HTTP code is not automatically treated as an exception.
- **JSON Parsing** – Fetch does not automatically process the response content. To obtain data, we usually call the response.json() method (or response.text(), response.blob() depending on the format).
- **No built-in interceptors** – Fetch does not offer a native mechanism to intercept requests/responses. However, a similar effect can be achieved by writing your own wrapper functions or using other libraries for logging or request modification.
- **Timeouts and cancellation** – Fetch does not have a direct timeout option. To limit waiting time or cancel a request, we use AbortController (discussed below).

Example of setting a timeout and cancelling a fetch request:

JavaScript

```
const controller = new AbortController();
const id = setTimeout(() => controller.abort(), 5000); // abort after 5s

try {
  const response = await fetch(url, { signal: controller.signal });
  clearTimeout(id);
  // ... processing response
} catch (err) {
  if (err.name === 'AbortError') {
    console.log('Request was aborted (timeout or cancellation)');
  } else {
    console.log('Other request error:', err);
  }
}
```

In the code above, we use AbortController to generate an abort signal. The controller.abort() method causes the promise returned by fetch to be rejected with an error named "AbortError". This is a universal way to cancel requests (e.g., when unmounting a screen to avoid updating a non-existent component).

Debugging: In React Native, it is worth using the **Flipper** tool to monitor network requests. Flipper has a **Network** plugin that displays a list of all HTTP requests made by the app (both via fetch and XHR/axios). Ensure the app is running in development mode and connected to Flipper – then you can inspect the details of every request (URL, headers, body, response, duration), which greatly facilitates diagnosing API problems. Additionally, you can add your own logs in the code (e.g., in .then().catch() for fetch or in axios interceptors) to log requests and responses during development.

1.2 Axios – HTTP Library with Extra Features

Axios is a popular library for making HTTP requests, used in browsers, Node.js, and React Native. It brings many conveniences compared to the pure Fetch API:

- **Simpler syntax and automatic processing** – Axios automatically serializes data to JSON when sending (a JavaScript object passed as data will be sent as JSON) and deserializes the JSON response into a JavaScript object (the response.data field already contains parsed data). We don't have to manually call .json() on the response.
- **Global configuration and instances** – We can create an axios instance with a defined baseURL, default headers, or parameters. This simplifies managing common request elements throughout the application:

JavaScript

```
import axios from 'axios';
const api = axios.create({
  baseURL: 'https://api.example.com/v1',
  timeout: 10000, // maximum wait time 10s
  headers: { 'Content-Type': 'application/json' }
});
```

Such an api instance will be used for all requests, eliminating the repetition of the base address and headers.

- **Interceptors** – One of the biggest advantages of axios is request and response interceptors. They allow you to intercept every request or response to modify them or handle errors globally:

JavaScript

```
// Intercepting every request – e.g., attaching an auth token:
api.interceptors.request.use(config => {
  const token = authStorage.getToken();  // pseudocode getting a token
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Intercepting the response – e.g., handling 401 errors globally:
api.interceptors.response.use(
  response => response,
  error => {
    if (error.response?.status === 401) {
      // e.g., automatic attempt to refresh token or logout
      console.log('Error 401 - unauthorized, refreshing token...');
      // ... (implementation of refresh logic or redirection to login)
    }
    return Promise.reject(error);
  }
);
```

Interceptors make it easier to add common logic – e.g., logging all requests in debug mode, setting Authorization headers, handling authentication errors, etc. In the case of fetch, you would have to manually wrap functions, while axios offers this out of the box.

- **HTTP error handling** – Unlike fetch, axios automatically rejects the Promise for response codes outside the 2xx range. This means a response with code 404 or 500 will go straight to the .catch() block with a convenient error object (containing, among others, error.response with the response data). This makes it easier to write unified error handling in one place.
- **Timeouts and cancellation** – Axios has a timeout option in its configuration (as shown in axios.create above). If a response does not arrive within the specified time, the request will be automatically aborted with an error. Furthermore, axios supports request cancellation via AbortController (from version 0.22) and the historical (now deprecated) CancelToken. The modern approach is to use signal with AbortController – very similar to fetch:

JavaScript

```
const controller = new AbortController();
try {
  await api.get('/tasks', { signal: controller.signal });
} catch(e) {
  if (axios.isCancel(e)) {
```

```
   // Checking if the error results from cancellation
   console.log('Request cancelled.');
  }
}
controller.abort(); // the request can be aborted at any time
```

From version v0.22, axios recommends using AbortController instead of CancelToken. The mechanism is the same as in fetch – aborting the signal will cause the promise to be rejected. In axios, the axios.isCancel(error) method allows you to recognize if the error resulted from cancellation.

- **Additional features** – Axios allows, for example, tracking upload/download progress (onUploadProgress, onDownloadProgress), which is useful for sending files or downloading larger resources (though in RN, dedicated native libraries are often used for large files). It also allows for easy parallel requests (using axios.all or Promise.all with an axios instance).

Example of a POST request with axios (for comparison with fetch):

JavaScript
```
const newTask = { title: 'New task', completed: false };
try {
  const response = await api.post('/tasks', newTask);
  // Axios will serialize newTask to JSON and set the Content-Type header automatically
  console.log('Task created, data:', response.data);
} catch (error) {
  if (error.response) {
    console.log('Request error:', error.response.status, error.response.data);
  } else {
    console.log('Network or other error:', error.message);
  }
}
```

In the code above, we see there is no need to perform JSON.stringify on the data or response.json() on the response – axios does it automatically. Error handling is also simpler: the error.response object exists only when the server returns an error response (4xx/5xx code).

Tip: In the React Native environment, you don't need to install a separate polyfill for fetch – it is available globally. However, if you decide on axios, you should install it via npm/yarn (npm install axios) and remember that it relies on the native XHR module, which RN supports. You can use fetch and axios simultaneously in a project, but it is usually better to choose one approach for consistency.

1.3 Interceptors, Headers, and Global Configuration

As mentioned, axios stands out for its global configuration capability and interceptors:

- **Global Headers:** We can define default headers at the axios instance level. E.g., `api.defaults.headers.common['Accept-Language'] = 'pl-PL';` will set the default language. Similarly, `api.defaults.headers.common.Authorization = 'Bearer TOKEN'` can globally add a token (though it's better to do this dynamically in an interceptor to always get the current token).
- **Request Interceptors:** The ideal place for attaching an authentication token before sending. In the interceptor, we have access to the `config` object – we can add or modify headers, change the address, etc. It is also worth handling token refresh cases here (more on this in the authorization section).
- **Response Interceptors:** They allow for global error interception. For example, we can check for a 500 code and log the event to an external service, or perform an automatic token refresh for 401. However, care must be taken to avoid infinite loops (e.g., if the token refresh also returns 401). A queue mechanism is often used – new requests are paused during token refresh and continued after.

Fetch does not have such a mechanism – how to solve this? We can write our own functions, e.g., `customFetch(url, options)`, which will fill in the headers (e.g., add a token from memory) before calling and check the status after receiving the response (e.g., if 401 – refresh the token and retry the request). This is, however, more work and has more room for errors. Therefore, many RN apps use axios for convenience, though pure fetch is also fine for simple needs.

Comparison Fetch vs Axios: Fetch is native and requires no dependencies, has a simple syntax, and is supported by browsers and RN out-of-the-box. Axios is an additional library but offers a richer API: interceptors, automatic JSON processing, better error handling, and cancellation. From a performance standpoint, differences are negligible in typical use cases (axios uses a similar mechanism to fetch/XHR under the hood). If the app requires many requests and extensive network handling (e.g., authorization, global logging, special headers), axios provides greater convenience. If you need minimalism or want to limit dependencies, fetch is perfectly sufficient – most axios functionality can be implemented manually with a bit of effort. The choice depends on project preferences and needs.

In summary: Both solutions are good for React Native. In the following sections (e.g., with React Query), we will see that higher-level tools can partially take over request-related tasks (e.g., retry mechanisms, cache, etc.), reducing the difference between fetch and axios because certain things will be managed by the library.

---

2. React Query – Advanced API Data State Management

As application complexity grows, manually managing the state of data coming from the API (e.g., item lists, object details, loading states, errors, data refreshing) becomes difficult. **React Query** (part of TanStack Query) is a library that simplifies working with asynchronous data: fetching, caching, refreshing, and mutations. In the context of React Native, it works the same way as in React on the web – with a few additions for the mobile environment. Currently, **React Query v5** (TanStack Query v5) is available, which introduces improvements

over previous versions (including smaller size, unified API). We will focus on the current version and modern hooks.

2.1 Basics and Configuration of React Query

To use React Query, you must install the @tanstack/react-query package (the newer name of the library – formerly react-query). In an Expo/RN project, installation proceeds standardly via npm/yarn. Next, we configure the query client (QueryClient) and a Provider at the top level of the app, usually in the App.tsx file:

JavaScript
```
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      retry: 2,             // retry failed requests 2 times by default
      staleTime: 5 * 60 * 1000, // time (ms) after which data is considered "stale"
    },
  },
});

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <NavigationContainer>{/* rest of the app */}</NavigationContainer>
    </QueryClientProvider>
  );
}
```

Above, we create a QueryClient instance with default options. We can specify global settings there, such as how many times a retry should occur, what the default staleTime is, whether the query should refresh upon re-entering the screen, etc. Placing the entire application inside QueryClientProvider provides access to React Query functionality in components.

Note: React Query v5 simplified the API – all configurations are passed as a single object instead of multiple positional arguments, as was often the case before. For example, what was formerly useQuery('tasks', fetchTasks) is now written as useQuery({ queryKey: ['tasks'], queryFn: fetchTasks }). This allows TypeScript to better support this call and keeps the entire configuration in one object.

2.2 Queries – the useQuery hook

The useQuery hook is used for fetching data (GET-type requests). Basic usage consists of a unique query key and a function that returns a promise (Promise) with data:

JavaScript
```
import { useQuery } from '@tanstack/react-query';
import api from './api'; // assume this is an axios instance

function TasksList() {
```

```
 const {
  data: tasks,
  error,
  isLoading,
  isFetching,
  refetch
 } = useQuery({
  queryKey: ['tasks'],          // unique cache key
  queryFn: async () => {
   const res = await api.get('/tasks');
   return res.data;           // return data (list of tasks)
  },
  staleTime: 1000 * 30,        // data valid for 30s (optional)
  refetchOnMount: false        // optional: don't refresh on mount if data is in cache
 });

 if (isLoading) {
  return <Text>Loading tasks...</Text>;
 }
 if (error) {
  return <Text>Error: failed to fetch the task list.</Text>;
 }

 return (
  <FlatList
   data={tasks}
   /* ... rendering the task list ... */
   refreshing={isFetching}      // isFetching can be used for the refresh indicator
   onRefresh={refetch}          // pull to refresh
  />
 );
}
```

Several things happen automatically above:

- The first execution of the request occurs immediately after mounting the component (unless we set the `enabled: false` option – then execution is manual via `refetch`).
- The `useQuery` hook returns an object with many useful properties:
  - **data** – the result of the query (cached). In our example, `tasks` is likely an array of tasks or an object fetched from the API.
  - **error** – the error object (if the request failed). When there was no error, it will be `undefined`.
  - **Loading state:** `isLoading` is true during the first execution of the query (until the response arrives). `isFetching` is true every time a data fetch is in progress for this query – e.g., also during a refresh (`refetch`). In React Query v5, the new name `isPending` was introduced instead of `isLoading` to unify state in queries and mutations. In code, you may encounter both names depending on the version – here we use `isLoading` for clarity.
  - **refetch** – a function whose call will trigger a re-execution of the query (ignoring any potential cache). Useful, for example, for handling a pull-to-refresh gesture of a list in RN.
  - **Others:** `status` (string 'loading' | 'error' | 'success'), `isSuccess`, `isError`, etc., for convenience.

- **Cache and staleTime:** React Query will store the result in the cache under the ['tasks'] key. As long as the data is considered fresh (via staleTime or the default 0 ms, which means immediately stale), re-entering the screen will not trigger a re-fetch – data from the cache will be shown immediately. After staleTime expires, the next mount will trigger a background refresh. We can customize this behavior:
    - Setting a larger staleTime (e.g., a few minutes) means that during this time data will be considered up-to-date and will not be automatically refetched on re-entry.
    - Setting refetchOnMount (default true when data is stale) allows, for example, to completely disable refreshing on mount (refetchOnMount: false), or to always force a refresh (refetchOnMount: 'always').
    - There is also a refetchInterval option for polling at certain intervals.

Integration with TypeScript: useQuery works great with TS. You can specify the result data type: useQuery<MyDataType>({ queryKey: [...], queryFn: ... }), but if we use a queryFn that itself has a well-defined return type (e.g., a function calling axios with a typed result), TS usually deduces the data type automatically. In case of an error, the default type is unknown, but you can use your own error classes or cast as needed.

Retry (Repeating): By default, React Query retries a failed query 3 times (with a short delay) before marking it as failed. This can be changed globally or per query (retry: number or retry: false to disable). Only network errors or response errors (e.g., 5xx code) are retried. This behavior is useful with an unstable connection – the user won't see an error immediately if, for example, the first request didn't go through. Of course, care should be taken not to repeat operations that shouldn't be idempotent (though by default React Query does not retry mutations, only GET-type queries). **Example:** If the /tasks endpoint occasionally returns a 500 error randomly, React Query will automatically try to fetch it up to 3 times before passing the error to the interface.

2.3 Mutations – the useMutation hook

The second important pillar is useMutation, used for operations that change data (POST/PUT/DELETE, etc.). Mutations can cause changes in server state, so we often want to update the cache or refresh GET queries after they are executed. Basic usage of useMutation:

JavaScript
```
import { useMutation, useQueryClient } from '@tanstack/react-query';

function NewTaskForm() {
  const queryClient = useQueryClient();

  const { mutate: addTask, isLoading: isAdding } = useMutation({
    mutationFn: async (newTaskData) => {
      const res = await api.post('/tasks', newTaskData);
      return res.data; // assume the API returns the created task object
    },
    onSuccess: (createdTask) => {
```

```
    // After successfully adding a task, refresh the task list:
    queryClient.invalidateQueries({ queryKey: ['tasks'] });
    // Alternatively: you could update the cache immediately instead of refetching (discussed below)
   },
   onError: (error) => {
    console.error('Failed to add task:', error);
   }
  });

  // ... UI form with a button:
  const handleSubmit = () => {
   addTask({ title: 'New task', completed: false });
  };

  return (
   <Button onPress={handleSubmit} disabled={isAdding} title="Add task" />
  );
}
```

In the code above, we define a mutation for adding a task:

- **mutationFn** – the function performing the request (here POST /tasks with the new task data). It can return a result (e.g., the created object), which is then available in onSuccess.
- **onSuccess** – an optional callback triggered when the mutation succeeds. The ideal place to invalidate the cache of queries whose data might have changed. We use queryClient.invalidateQueries with the key – in this case, the ['tasks'] list should be fetched again to include the new task. Instead of invalidating, you can also update the data in the cache manually.
- **onError** – error handling (e.g., showing a message). You can also use the error object in the component via mutation.error, similar to a query.

The useMutation hook returns, among others:

- **mutate** (or mutateAsync) – the function to call the mutation by passing data.
- **States:** isLoading (in v5 also called isPending), isSuccess, isError – analogous to queries, they inform about the state of a given modifying request.
- **Additionally data** – the result returned from the mutation (e.g., createdTask in onSuccess), **error** – the error object if one occurred.

Optimistic updates – optimistic UI updates An optimistic update consists of assuming the operation succeeds and immediately modifying the UI before receiving a response from the server. If it later turns out the operation failed, we must roll back these changes or signal an error to the user. React Query supports optimistic updates in two ways:

1. **Through UI and mutation state (simpler method in React Query v5):** We can leverage the fact that useMutation returns the parameters of the last mutation (variables) and the isPending state. This allows us, for example, to temporarily add an element to the task list without messing with the cache:

JavaScript

```
const addTaskMutation = useMutation({
  mutationFn: addTaskApiCall,
  onSettled: () => queryClient.invalidateQueries({ queryKey: ['tasks'] })
});
const { mutate: addTask, variables: newTaskVars, isPending: isAdding } = addTaskMutation;

// ... in the task list component:
return (
  <View>
    {tasks.map(task => <TaskItem key={task.id} {...task} />)}
    {isAdding && (
      <TaskItem task={{ ...newTaskVars, id: 'temp-id' }} style={{ opacity: 0.5 }} />
    )}
  </View>
);
```

In the scheme above, when we call addTask(newTaskData), the isPending state will be true, and variables will store newTaskData. So we can render a temporary element with the new task data (e.g., with a semi-transparent style to distinguish it) into the list. When the mutation succeeds, isPending returns to false, and the refetch in onSettled refreshes the list with the actual new element from the database. If the mutation fails, the temporary element also disappears (though we can leave it and, for example, add a "Try again" button – still utilizing variables even after an error, as React Query does not remove them automatically on error). This method is simpler because we don't touch the cache manually – we simply conditionally render an extra element based on the mutation state. React Query v5 also added the useMutationState hook, which allows tracking changes even in another part of the app if the mutation and temporary data display are not in the same component.

2. **Through cache modification (traditional method):** A more advanced approach consists of making a direct change to the cache in the mutation's onMutate, and restoring the previous state in onError in case of failure. Example for adding a task:

JavaScript

```
const queryClient = useQueryClient();
useMutation({
  mutationFn: addTaskApiCall,
  // before performing the mutation:
  onMutate: async (newTask) => {
    await queryClient.cancelQueries({ queryKey: ['tasks'] });
    const previousTasks = queryClient.getQueryData(['tasks']);
    // Optimistically set the new task in the cache:
    queryClient.setQueryData(['tasks'], old => [...(old || []), { id: '__temp__', ...newTask }]);
    return { previousTasks }; // return a snapshot
  },
  onError: (err, newTask, context) => {
    // on error, restore the previous task list
    queryClient.setQueryData(['tasks'], context.previousTasks);
  },
  onSettled: () => {
    // regardless of the result, get current data from the server
    queryClient.invalidateQueries({ queryKey: ['tasks'] });
  }
});
```

- onMutate stops any current refreshes (cancelQueries), gets the current task list from the cache (previousTasks), and then adds a new object to the list using setQueryData. We use a temporary id: '__temp__' or something unique to distinguish the optimistic object.
- We return a value from onMutate (here an object with the previous list), which will be passed to onError if the mutation fails.
- In onError, upon error, we restore the previous state of the task list from context.previousTasks.
- onSettled (called after both success and error) is used to ensure we finally have consistency with the server – here through invalidation and refetching of the list.

This approach provides full control and an immediate reaction in the UI. It is slightly more complicated (you have to remember to restore state), but it may be necessary in some scenarios, e.g., when a change is harder to reflect with conditional rendering.

The choice of method depends on the case. For simply adding an element to a list, the first method (UI) is sufficient and less error-prone – in case of failure, the element will simply disappear or we can offer a retry. The second method provides full flexibility (you can, for example, optimistically update many different queries in the cache). React Query v5 tries to facilitate this process, which is why simplifications and the useMutationState hook were introduced.

Parallel mutations: It is worth mentioning that React Query allows performing several mutations at once and managing their state globally. Using useIsMutating(), you can check if any mutation is in progress (e.g., to block buttons globally), and useMutationState allows monitoring active mutations by key. For example, you can call useMutation({ mutationKey: ['deleteTask'], … }) for deleting a task in one component, and in another listen for any pending deleteTask mutation and display, for example, a spinner next to the item.

2.4 Cache, Refetching, and Data Invalidation

One of React Query's greatest advantages is the key-based data cache. Understanding a few concepts will help you use it effectively:

- **Query Key (queryKey):** an array or string uniquely identifying data. E.g., ['tasks'] for a list of all tasks, but ['task', id] for the details of a single task. The array structure can contain query parameters (e.g., ['tasks', { page: 2 }] if the key should be page 2). The key determines what is treated as the same data in the cache.
- **Cache Time vs Stale Time:**
    - **Stale Time** – data freshness time (configured, e.g., 0 by default means data becomes "stale" immediately after fetching). As long as data is fresh, React Query will not initiate a re-fetch during re-renders or application focus.
    - **Cache Time** – storage time for old data in memory after its "expiration". Default is 5 minutes. After this time, if there is no component using a given query, the data will be removed from cache memory (so-called garbage collection). In v5, this was better named gcTime (garbage collection time). If we

refer to this query again during this time, even stale data can be shown immediately (with a flag that it is stale) and a refetch will occur in parallel.

- **Invalidation (`invalidateQueries`):** programmatically marking data as out-of-date. We use this, for example, after mutations in `onSuccess` to notify React Query: "hey, the data under key X has changed, fetch it again at the next opportunity." Invalidation sets the state to `stale` and, if a given query is currently used in the UI, it will trigger an automatic refetch. If it's not in use, the next time it appears (mount), it will fetch newer data.
- **Refetching on focus/connection:** By default (in a browser), React Query refreshes queries upon returning to the window (window focus) and after regaining connection (network reconnect). In React Native, we can also have these mechanisms, but we must integrate them manually:
    - **App Focus:** using `focusManager` and the `AppState` module (React Query provides `focusManager`).
    - **Online Status:** using `onlineManager` and, for example, the `NetInfo` library to detect network loss/regain. After configuration, React Query will know, for example, that it was offline and is now online – it can refresh pending queries.
- **Manual Refresh (`refetch`):** as shown, the `useQuery` hook provides a `refetch` method. You can also globally refresh certain groups of queries: `queryClient.refetchQueries({ queryKey: ['tasks'] })` to refresh all queries starting with `tasks` (which is useful when, for example, many different variants of task data are to be refreshed, although most often `invalidate` is sufficient).

Integration with React Native – focus and online: In pure RN, React Query does not have access to window focus events (because there is no browser window) or network status, but this can be easily added:

JavaScript

```
// Somewhere in the app initialization code (e.g., right after creating QueryClient):
import NetInfo from '@react-native-community/netinfo';
import { onlineManager, focusManager } from '@tanstack/react-query';
import { AppState } from 'react-native';

// Connection: listen for network status changes and inform React Query
onlineManager.setEventListener(setOnline => {
  return NetInfo.addEventListener(state => {
    setOnline(!!state.isConnected);
  });
});

// App focus: listen for whether the app is active (foreground)
AppState.addEventListener('change', state => {
  focusManager.setFocused(state === 'active');
});
```

The code above will cause, for example, if the device loses internet, React Query to pause automatic queries and mark offline mode (queries may enter an error state that can be handled appropriately). When the network returns, `onlineManager` will notify the library – by default, this will cause a refetch of all previously failed queries immediately (which is desired behavior in most cases). Similarly with focus: when the user minimizes and restores the app,

calling focusManager.setFocused(true) can trigger a data refresh (provided some query was stale). We can also customize this behavior with the refetchOnReconnect, refetchOnFocus parameters (just like on the web, only here it depends on this integration).

Error Handling in React Query: By default, query errors are not thrown as exceptions to the component but are passed in error and change isError. However, you can enable a mode where an exception will be thrown upon error and caught by the nearest Error Boundary in the tree. The useErrorBoundary: true option (globally or for an individual query) is used for this. This is useful in larger apps where we want, for example, one global error boundary component instead of checking error in every place.

2.5 Handling Loading States and Errors (UI)

In the interface, we must anticipate three main states for queries: **loading**, **error**, and **data loaded**. Above in the TasksList code, we saw simple handling via if (isLoading) ... else if (error) ... else .... In practice, this can be expanded:

- **Loading placeholders:** Instead of plain "Loading..." text, placeholder components or spinners are often used. E.g., in React Native, ActivityIndicator as a loading indicator, or prepared UI "shells" (so-called **skeleton screens** – e.g., gray bars mimicking a list). Example:

  JavaScript

  ```
  {isLoading && <ActivityIndicator size="large" color="#0000ff" />}
  ```

  Alternatively, libraries like react-native-paper or NativeBase offer ready-made placeholder components.

- **Error State:** In case of an error, it's worth displaying a user-friendly message. The error object itself from React Query can vary (for HTTP errors it might be an axios error with status info, for network errors – it might be a TypeError from fetch). Therefore, we often prepare a helper function that maps various errors to a message:

JavaScript

```
function getErrorMessage(error) {
  if (!error?.response) {
    return 'No internet connection.';
  }
  const status = error.response.status;
  if (status === 404) return 'Resource not found.';
  if (status === 500) return 'Server error, try again later.';
  return 'An error occurred. Code: ' + status;
}
// ...
{error && <Text style={{color: 'red'}}>{getErrorMessage(error)}</Text>}
```

Such consistent error handling ensures the user gets a readable message instead of, for example, a raw TypeError: Network request failed. You can also consider sending error logs to an external system (Sentry, LogRocket, etc.), but that is beyond our lecture.

- **Refresh State:** We often want to signal when data is being refreshed in the background (e.g., after scrolling down a list, or automatically every so often). In React Query, `isFetching` (for a query) or `isFetchingNextPage` (for an infinite query, more on that in a moment) is used for this. We can, for example, show a small "Refreshing..." indicator at the top of the list or change the pull-to-refresh title. In RN, `FlatList` has a `refreshing` and `onRefresh` prop – connecting `isFetching` and `refetch` there as above will automatically show a spinner upon pull-to-refresh gestures.

React Query also takes care of **keeping previous data** during a refetch (so-called `keepPreviousData` in v4, currently in v5 this is combined with `placeholderData`) – this allows, for example, showing the old page during pagination until the new one loads, instead of an empty screen. By default, however, when refreshing data, `data` gets overwritten. If we want, we can set `keepPreviousData: true` to keep old data during a fetch (in v5 this option was merged with `placeholderData` and it might need to be set differently, but the idea is similar).

In summary: loading and error states should be clearly handled in the UI. Don't leave the user with a frozen app – better to show a spinner. And don't hide errors in the console – display a message or try to automatically retry (if it makes sense). React Query facilitates a lot because it provides these states right away without the need to manually create `isLoading` type variables.

---

3. Pagination and "Infinite" Queries

Many APIs provide result pagination – e.g., item lists are divided into pages of 10, 20 items or provide a cursor-based pagination mechanism. React Query supports these scenarios using two approaches:

- **useQuery with a page parameter** – traditional approach: each page is a separate query, e.g., `useQuery(['tasks', page], queryFn)`. You can then have "Next Page" / "Previous Page" buttons in the UI, or dynamically increase the page number while scrolling and perform subsequent queries.
- **useInfiniteQuery** – a special React Query hook that supports fetching subsequent "batches" of data and joining them automatically. Ideal for lists with **infinite scroll** (scrolling down triggers the fetch of the next set of results).

3.1 Offset/Limit Pagination vs Cursors

Before we move to the code, let's explain the two ways of paging we may encounter in an API:

- **Offset/Limit (or page/size):** The most common approach – e.g., we have an endpoint `/tasks?page=2&limit=10` or `/tasks?offset=20&limit=10`. The server then returns, for example, `tasks` and information about the total number of items or addresses for the next/previous page. Offset/page requires knowing how many items to skip. A disadvantage is that with dynamically changing data, offset pagination may skip or

duplicate items if some were added/removed in the meantime (e.g., inserting a new item at the beginning of the list will change offsets).

- **Cursor (or continuation tokens):** An increasingly popular approach – the server returns a marker for the next page, e.g., `nextCursor` or `nextPageToken`. The query then looks like `/tasks?cursor=abc123` where `abc123` indicates where to continue. Most often, the API also returns information on whether there is a next page (e.g., `hasNextPage: true/false`). Cursors are more resistant to data changes – they point to a specific point in time/order.

In React Query, we will handle both scenarios using `useInfiniteQuery`, but the `getNextPageParam` logic will be slightly different.

3.2 The useInfiniteQuery Hook

`useInfiniteQuery` works similarly to `useQuery`, but assumes we will be fetching multiple pages of data. Its `queryFn` should accept a `pageParam` parameter, and in the options, we must provide a `getNextPageParam` function that tells the library which `pageParam` to use for the next page.
Example usage (for an offset/page API):

JavaScript

```
const {
  data,
  fetchNextPage,
  hasNextPage,
  isFetchingNextPage,
  isLoading
} = useInfiniteQuery({
  queryKey: ['tasks'],
  queryFn: ({ pageParam = 1 }) => api.get(`/tasks?page=${pageParam}`).then(res => res.data),
  getNextPageParam: (lastPage, pages) => {
    // Assume the lastPage response contains a nextPage field (next page number) or null if it's the end
    return lastPage.nextPage ?? false;
    // if we return false/undefined, React Query will consider that there is no next page
  }
});
```

Several explanations:

- **pageParam** – page parameters; upon the first call, it equals the default value (here 1, as we set it). Later, React Query will insert subsequent values returned from `getNextPageParam`.
- **getNextPageParam(lastPage, allPages)** – the function receives the last fetched page (data returned by `queryFn`) and an array of all pages so far. It must return the `pageParam` value for the next page or `false/undefined` if it's the end. In the pseudo-code above, we assume that `lastPage` (data from the server) has a `nextPage` field. Alternatively, if the API returns, e.g., `currentPage` and `totalPages`, we could write:

  JavaScript

  ```
  getNextPageParam: (lastPage) => {
  ```

```
  return lastPage.currentPage < lastPage.totalPages
    ? lastPage.currentPage + 1
    : undefined;
}
```

- **hasNextPage** – a boolean value that React Query determines automatically based on getNextPageParam. If getNextPageParam returns a value (e.g., page number or cursor), hasNextPage will be true. If it returns undefined or false, hasNextPage will be false (end of data).
- **fetchNextPage** – function to fetch the next page. It will internally call queryFn with the next pageParam (the one getNextPageParam returned previously).
- **data** – here the structure is different than with useQuery. data contains an object with the fields:
    - **data.pages** – an array where each element is the result of one page (exactly what queryFn returns).
    - **data.pageParams** – an array with parameters used for those pages (can usually be ignored, unless debugging).

To present the data in a component, we usually join all pages into one list. E.g.:

JavaScript
```
<FlatList
  data={ data ? data.pages.flatMap(page => page.results) : [] }
  renderItem={...}
  onEndReached={() => { if (hasNextPage) fetchNextPage(); }}
  ListFooterComponent={ isFetchingNextPage ? <ActivityIndicator /> : null }
/>
```

In the example above:

- We assume that each page (page) has a results field with an array of items (like how the API returns a list of tasks). We use flatMap (or map(...).flat()) to create one flattened array of all results. Thanks to this, FlatList treats it as one continuous list.
- **onEndReached** – FlatList event triggered when the user scrolls near the bottom of the list. Inside, we call fetchNextPage() if there is a next page. Additionally, it's worth giving an onEndReachedThreshold, e.g., 0.3 (30%), to trigger the fetch a bit before the very end of the scroll, which will ensure smoother loading.
- **ListFooterComponent** – a component displayed at the end of the list. We use this to show a spinner (ActivityIndicator) at the moment when isFetchingNextPage is true (i.e., the next page is fetching). This way, the user sees a spinning wheel at the bottom of the list while more data is loading.

Note: FlatList also requires keyExtractor and renderItem props – we implement these standardly. Remember that the keys of list items should be globally unique. If we attach items from subsequent pages, the best key will be, for example, the unique id of each task. If we use list indices as key, when loading pages it may cause problems with element refreshing – better to avoid index as key.

Alternative – pagination with a "Load more" button: Sometimes, instead of infinite scroll, a classic "Load more" button is used. In React Query, this can be implemented based on useInfiniteQuery or regular useQuery. E.g., with useInfiniteQuery you can omit onEndReached entirely and place below the list:

JavaScript

```
{ hasNextPage && !isFetchingNextPage && (
   <Button title="Show more" onPress={() => fetchNextPage()} />
 )
}
```

This will display the button as long as there is a next page, and upon clicking, fetch the next one. After fetching (when hasNextPage changes to false, e.g., the end of the list is reached), the button will disappear. This approach can be clearer for the user when pages are distinct.

Comparison useInfiniteQuery vs useQuery for pagination: You can of course do pagination using multiple useQuery calls and page state in the component (e.g., keep const [page, setPage] = useState(1), then a button increases page, and useQuery reacts to page as part of queryKey). However, useInfiniteQuery simplifies this in that:

- It manages the array of pages and their joining itself.
- It keeps information on whether there is a next page (hasNextPage).
- It provides a unique isFetchingNextPage distinguishing fetching from initial isLoading. Therefore, in the case of **infinite scroll**, React Query clearly recommends using useInfiniteQuery because the code is cleaner and less error-prone.

3.3 List Scrolling and Dynamic Data Fetching (Infinite Scroll)

Combining the above, the infinite scroll implementation in RN looks as follows:

1. We use useInfiniteQuery with appropriate getNextPageParam.
2. We utilize the FlatList component to display data:
    - We set data as a flattened list of all items from the pages.
    - The onEndReached prop calls fetchNextPage (protected by hasNextPage).
    - Optionally onEndReachedThreshold e.g., 0.5 (or another value) to control the fetch moment. Default is 0.5, which means when we scroll to half before the end of the list, onEndReached will trigger.
    - The ListFooterComponent prop displays a spinner when isFetchingNextPage is true.
    - You can also add refreshing and onRefresh to handle pull-to-refresh, e.g.:

      JavaScript

      ```
      refreshing={ isLoading && data?.pages?.length > 0 }
      onRefresh={ refetch }
      ```

      The above assumes that if we already have some data and isLoading is true, it means we are performing a refresh (e.g., manual).

3. **Handling unmount/leaving the screen:** React Query can automatically cancel queries that are in progress when the component unmounts (e.g., the user leaves the list before the next page finishes loading). This mechanism is based on AbortController underneath – React Query passes a signal to queryFn. However, queryFn must use fetch or axios with signal handling. In practice, using axios in RN, we can also pass signal (React Query in queryFn arguments provides them in context.signal in v5, or abort itself in v4). You must ensure that our fetching function takes this into account if we want to benefit from automatic cancellation of unnecessary queries (e.g., scrolling quickly down and up so that old queries don't land in the meantime). Details of this mechanism are described in the TanStack Query documentation, but in many cases, we don't have to do anything special – just use axios (≥0.22) or fetch and React Query itself will cancel the signal upon query deactivation.

Performance tip: With very long lists, it's worth limiting the number of cached pages. React Query v5 allows setting the maximum number of pages stored in memory (maxPages option in infiniteQuery configuration). If a list can have hundreds of pages, keeping them all might consume memory – you can, for example, always discard the oldest pages from the cache. However, typical lists (several dozen items) do not require such optimization at the start. FlatList also has optimization mechanisms (item recycling, removeClippedSubviews, etc.) that prevent performance problems with long lists.

---

4. Error Boundaries and Unified Fallback UI

Despite best efforts, errors in apps are inevitable – it could be a programming error (e.g., exception in component render) or an environmental error (e.g., no network). **Error Boundaries** are a React mechanism that allows catching JavaScript errors during component rendering and displaying a backup interface instead of crashing the entire app. In React (from version 16), ready-made implementations can be used. In the context of React Native, libraries such as react-native-error-boundary are often used, providing an easy-to-use <ErrorBoundary> component. **Example usage of ErrorBoundary:**

JavaScript

```
import ErrorBoundary from "react-native-error-boundary";

const FallbackUI = ({ error, resetError }) => (
  <View style={{ padding: 20 }}>
    <Text>An unexpected error occurred.</Text>
    <Text>{error.toString()}</Text>
    <Button title="Try again" onPress={resetError} />
  </View>
);

export default function App() {
  return (
    <ErrorBoundary FallbackComponent={FallbackUI}>
      {/* All rest of app / navigation */}
      <MainNavigator />
    </ErrorBoundary>
```

```
  );
}
```

In the code above, we wrap the main part of the app in an ErrorBoundary. If any of the child components throw an error during rendering, it will be caught, and FallbackUI will be displayed instead. This helps prevent the "white screen of death" and gives the user an option like retrying (resetError button resets the state in ErrorBoundary, allowing components to re-render as if nothing happened – this can be used, for example, to reset an error after navigation or data refresh).

Granularity: We can have one global ErrorBoundary for the whole app (displaying, e.g., a "Something went wrong. Restart the app" message) or place them more locally, e.g., around individual sensitive screens, so that the failure of one screen doesn't disable the whole app. For instance, if we have a separate ErrorBoundary around the task list component and that component hits an exception, it will show fallback UI only instead of the list, while the rest of the app (e.g., header, menu) will function.

It's worth highlighting: ErrorBoundary protects against rendering errors and component lifecycle methods. It will **not** catch asynchronous errors in event handlers or inside a promise (there you must use try/catch in the code or .catch in the Promise). Network query errors (e.g., error in React Query) are not runtime exceptions themselves (unless we enable useErrorBoundary mode). Therefore, an error boundary will mainly be useful for unforeseen exceptions (e.g., a TypeError when referring to an undefined field).

4.1 Global Network Error Handling and Offline Messages

Beyond ErrorBoundary, in a mobile app, it's worth thinking about consistent handling of connection loss and API errors at the UX level:

- **No Network:** Using the aforementioned NetInfo module, we can listen for connection state changes. When isConnected changes to false, we can, for example, display a global banner at the top of the app with "No internet connection" info. This can be done via some global component (e.g., in NavigationContainer in the header conditionally insert something), or even a Popup/Toast. Also, ensure that actions requiring internet are blocked or queued so the user doesn't click buttons in vain.
- **Unified Error Messages:** As already discussed, it's good to translate errors into human language. This can be centralized. E.g., have middleware in an axios interceptor that calls a showErrorToast(message) function for every error. Or in React Query, we can utilize onError globally:

JavaScript
```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      onError: error => {
        showErrorToast(getErrorMessage(error));
      }
    },
    mutations: {
```

```
    onError: error => {
      showErrorToast(getErrorMessage(error));
    }
   }
  }
});
```

Then every query/mutation error will trigger our function showing, for example, a Toast with a message. Libraries like react-native-toast-message or react-native-flash-message enable easily displaying a good-looking error notification on the screen. Such a global strategy ensures that errors won't be overlooked – the user will always get some message.

- **Retry / User Action Repeating:** If an operation failed due to the network, a retry mechanism can be considered. At the UI level – e.g., after a task list error, a "Try again" button can be shown (which will trigger refetch()). For mutations (e.g., adding a task), in the optimistic example we even showed a "Retry" button next to the element if it failed. It's important the user doesn't feel blocked – if something didn't work, give them an opportunity to react.

Capturing uncaught errors: In JavaScript, you can register a global handler for unhandled promise rejections (UnhandledPromiseRejection) and for unhandled errors (ErrorUtils in RN or globalThis.onerror). However, in practice, it's better to use dedicated tools like Sentry for this – this is beyond the scope of the lecture, but in large apps, integration with, e.g., Sentry allows for automatic reporting of every JS exception along with a stack trace.

In summary, consistent error handling means:

- reacting to no internet (e.g., "You are offline" info),
- showing understandable messages instead of leaving the user without info,
- logging errors for developers (to be able to fix them),
- and not crashing the whole app in case of an exception (Error Boundary).

---

5. Authorization and Authentication in Requests

In mobile apps, we often need to communicate with APIs that require an authentication token (e.g., a JWT token passed in the Authorization header as Bearer). In this section, we will discuss:

- Storing tokens after logging in,
- Automatically attaching them to requests (e.g., in an axios interceptor),
- Token refreshing (refresh token flow),
- Secure storage of credentials (SecureStore, EncryptedStorage).

5.1 Token Storage – Where and How?

The simplest approach is keeping the token in state memory (e.g., in React context or some global store) during app runtime, and also saving it to permanent memory so that re-logging is not required upon restarting the app. **AsyncStorage vs Secure Storage:** Although AsyncStorage (asynchronous key-value memory in RN) can be used to store a token, it is not recommended for sensitive data because AsyncStorage stores data in plain text on the disk. Better solutions are secure stores:

- **Expo SecureStore:** if using Expo, we have easy access to the SecureStore module. It allows saving encrypted data in the Keychain (iOS) or Keystore (Android). Example:

  JavaScript

  ```
  import * as SecureStore from 'expo-secure-store';
  await SecureStore.setItemAsync('token', accessToken);
  // ... then to fetch:
  const token = await SecureStore.getItemAsync('token');
  ```

  SecureStore is part of the Expo SDK and is used for encrypted storage of key-value pairs. It is ideal for tokens, passwords, etc. (max value size is approx. ~2048 bytes, which is perfectly sufficient for tokens).

- **EncryptedStorage (react-native-encrypted-storage):** in RN apps outside of Expo, the react-native-encrypted-storage library can be used. It works similarly – under the hood it uses Keychain on iOS and EncryptedSharedPreferences on Android – and abstractly gives a simple JS API:

  JavaScript

  ```
  import EncryptedStorage from 'react-native-encrypted-storage';
  await EncryptedStorage.setItem('token', accessToken);
  const token = await EncryptedStorage.getItem('token');
  ```

  EncryptedStorage is a secured alternative to AsyncStorage – it uses system mechanisms for secure storage, so data is encrypted and tied to the app (no one will directly read it from files). It requires adding a native module to the project (linking or autolinking).

- **Keychain on iOS / Keystore on Android directly:** Libraries also exist for directly using iOS Keychain, e.g., react-native-keychain. SecureStore or EncryptedStorage basically facilitate this, but you can also use react-native-keychain without Expo to save passwords/tokens with some extra options (e.g., "only after unlocking device" type access control).

It is important **not to store tokens in ordinary, unsecured places** (AsyncStorage, file, redux, plaintext, etc.) because if someone gains access to the phone's memory (e.g., via a malicious app on a rooted/jailbroken device), it's easier for them to extract such data. SecureStore/Keychain makes an attack difficult – data is encrypted per app and the system takes care of its protection.

Information: SecureStore and EncryptedStorage store data permanently – i.e., it remains even after shutting down the app, and even after uninstalling (in the case of iOS Keychain). If we

don't want tokens to survive uninstallation, the EncryptedStorage documentation describes a trick for clearing the Keychain on the first run after reinstallation. Usually, however, this is not a problem – the token will expire by then anyway.

5.2 Attaching the Token to Requests (Bearer)

The best place for this is our communication layer, e.g., an axios interceptor. As shown previously:

JavaScript
```
api.interceptors.request.use(config => {
  const token = authToken; // e.g., global variable or from some auth module
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
```

Such a solution requires having access to the token at the time of setting up the interceptor. You can:

- Load the token from SecureStore into a variable upon app startup (e.g., in AuthContext state). When the user logs in, also save the variable.
- Alternatively, if the token can expire and be refreshed, the interceptor could always use the current value from some central place (context, module).
- Remember that SecureStore.getItemAsync is asynchronous. An axios interceptor can return a promise – meaning async/await can be used in it. Alternatively, it's simpler to keep the token in memory (since we have it after logging in anyway) and sync it with secure storage.

In the case of using Fetch: there are no interceptors, so the header must be added every time:

JavaScript
```
const token = await SecureStore.getItemAsync('token');
fetch(url, {
  headers: { Authorization: `Bearer ${token}`, ... }
});
```

Which of course is tedious. So you can wrap fetch:

JavaScript
```
async function authorizedFetch(endpoint, options = {}) {
  const token = await SecureStore.getItemAsync('token');
  const authHeaders = token ? { Authorization: `Bearer ${token}` } : {};
  const finalOptions = { ...options, headers: { ...options.headers, ...authHeaders } };
  return fetch(baseURL + endpoint, finalOptions);
}
```

And use authorizedFetch throughout the app instead of naked fetch. Such a pattern ensures centralization of authentication logic.

5.3 Token Refreshing (Refresh Token Flow)

Many authentication systems issue two tokens: an **Access Token** (short-lived, e.g., 15 min) for request authorization and a **Refresh Token** (longer-lived, e.g., 7 days or more), which can be used to obtain a new Access Token when the old one expires. The scenario is:

1. The user logs in – we get AccessToken + RefreshToken.
2. We use AccessToken in headers for every protected action.
3. When the server responds that the AccessToken is invalid/expired (e.g., 401 code with "Token expired" info), we must call the refresh endpoint with the RefreshToken.
4. After receiving a new AccessToken (and sometimes a new RefreshToken), we save them and retry the original request that failed.
5. If refreshing fails (e.g., RefreshToken also invalid), then we must re-log the user (e.g., eject them to the login screen, clear state).

Implementation in an axios interceptor:

JavaScript

```
let isRefreshing = false;
let pendingRequests = []; // queue of requests waiting for refresh

api.interceptors.response.use(
  res => res,
  async error => {
    const { response } = error;
    if (response?.status === 401) {
      // If the 401 error concerns token expiration:
      if (response.data?.message === 'TokenExpired') {
        if (!isRefreshing) {
          isRefreshing = true;
          try {
            const refreshToken = await SecureStore.getItemAsync('refreshToken');
            const refreshRes = await api.post('/auth/refresh', { token: refreshToken });
            const newAccessToken = refreshRes.data.accessToken;
            // Save the new token
            await SecureStore.setItemAsync('token', newAccessToken);
            authToken = newAccessToken; // update variable in memory
            isRefreshing = false;
            // Carry out pending requests with the new token
            pendingRequests.forEach(cb => cb(newAccessToken));
            pendingRequests = [];
          } catch (err) {
            isRefreshing = false;
            pendingRequests = [];
            // Refresh failed – log out user
            navigateToLogin();
            return Promise.reject(err);
          }
        }
        // We return a new Promise, which will cause a delay in executing the original request until refresh:
```

```
      return new Promise((resolve, reject) => {
        pendingRequests.push((token) => {
          // Retry the original request with the new token
          error.config.headers.Authorization = 'Bearer ' + token;
          resolve( axios(error.config) );
        });
      });
    }
  }
  return Promise.reject(error);
 }
);
```

The code above is somewhat complex, but the key points are:

- isRefreshing – a flag so that only one refresh is performed at a time, even if many requests received a 401 simultaneously.
- pendingRequests – a queue of functions to be executed after refresh (i.e., they will retry original requests).
- When the first 401 with the reason "TokenExpired" arrives, we enter the if. If no one is refreshing (!isRefreshing), we start the refresh process:
    - Extract refreshToken from storage,
    - Call /auth/refresh (assuming it returns a new accessToken, potentially also a new refreshToken – this should also be handled).
    - Upon success, save the new token (in memory and secure storage), set isRefreshing = false, and execute all pending requests with the new token (calling callbacks from the queue).
    - If refreshing fails, meaning the session is lost – clear tokens, direct to login.
- If a subsequent request hits a 401 while isRefreshing is true (i.e., refresh already in progress), we don't refresh again but return a new Promise and add its resolver to the queue. This Promise is what the interceptor will return for that second request – so the original code waiting for a response will be suspended. When refreshing finishes, we will call the callback from the queue, which will:
    - Set the new Authorization header,
    - Call the original request again (axios(error.config)),
    - The external Promise will resolve, so the original request will receive its response (it won't hit the 401 interceptor a second time because the token is new).

Such a pattern ensures that upon token expiration, we don't log the user out immediately but smoothly renew the token in the background. The user might not even notice, besides perhaps a minimal delay. Details should of course be refined (e.g., what if the refresh endpoint itself returns 401 – then logout as well). If we use React Query, most of this happens at the axios level and is transparent to React Query – the request just takes longer because it waits for a refresh, but finally receives data. One could potentially react to a global 401 error differently, but the above is a quite comprehensive solution.

Note on Refresh Token: We store two tokens – access and refresh. We keep access, e.g., in memory/context (and secure store for reopening the app), and refresh in secure store. The

refresh token is even more sensitive – if someone gets it, they can extort subsequent access tokens. Therefore:

- The refresh token's lifespan should be limited (e.g., forcing login every 2 weeks).
- Consider keeping the refresh token only in SecureStorage and never keeping it in memory explicitly (only fetching it when needed for refresh).
- After logging out, remember to remove both tokens from storage.

5.4 Secure Storage – Summary

- **expo-secure-store** – simple way to securely store tokens in Expo. Uses system mechanisms (Keychain/Keystore), therefore it is recommended for confidential data.
- **react-native-encrypted-storage** – analogous solution outside of Expo, also uses Keychain/EncryptedSharedPrefs. Simple interface of four methods: setItem, getItem, removeItem, clear.
- **Do not use AsyncStorage for tokens** – lack of encryption, data in a JS file. Ultimately, if the app doesn't store anything critical and we're not worried about it, AsyncStorage can be used.

---

6. Demo: "Tasks" App – Practical Combination of All Elements

Finally, let's combine theory in a mini example. Imagine a Tasks app (a list of things to do) with a REST backend. We have the following functionalities:

- User login (obtaining a token – assume they are already logged in and we have the token).
- Fetching the task list – GET /tasks endpoint, returning a list of tasks (paginated).
- Adding a new task – POST /tasks endpoint (returns the created task).
- Editing a task – PUT /tasks/:id endpoint (returns the updated task).
- Pagination – assume the task list can be long, so the API paginates results (e.g., ?page= parameters).
- Authorization – all the above require a Bearer token in the header.

Let's try to build a simplified architecture: **Step 1: API and React Query Configuration**

JavaScript
```
// api.ts - Axios and tokens configuration
import axios from 'axios';
import * as SecureStore from 'expo-secure-store';

const API_URL = 'https://example.com/api';

export const api = axios.create({
  baseURL: API_URL,
  timeout: 5000,
```

```
});

// Insert token into every request (if exists)
api.interceptors.request.use(async config => {
  const token = await SecureStore.getItemAsync('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// (Optional) Response interceptor for token refresh / 401 handling:
api.interceptors.response.use(
  response => response,
  async error => {
    const originalRequest = error.config;
    if (error.response?.status === 401 && !originalRequest._retry) {
      originalRequest._retry = true;
      // Assume the existence of a refreshToken() function that fetches and saves a new token
      try {
        const newToken = await refreshToken();
        originalRequest.headers.Authorization = `Bearer ${newToken}`;
        return api(originalRequest); // retry original request with new token
      } catch (e) {
        // Refresh failed - user could be logged out
        // navigateToLoginScreen();
        return Promise.reject(e);
      }
    }
    return Promise.reject(error);
  }
);
```

Above:

- We create axios with baseURL.
- Request interceptor attaches the token (fetched from SecureStore with every request – this might be slightly inefficient, it would be better to keep the token in a variable, but for certainty we use storage).
- Response interceptor checks for 401. We use simplified logic here: checking the _retry flag to avoid infinite loops. We call refreshToken() – this should be a function performing, e.g., api.post('/auth/refresh') (we'd have to be careful not to hit the interceptor again – another axios instance without interceptors could be used for this). After obtaining a new token, we save it (here we assume refreshToken() already does this and returns accessToken) and retry the original request. If it fails, we reject the error and likely catch it in the app to log out.

Step 2: Hooks Using React Query for Requests: To organize, we'll make our own hooks for task handling:

JavaScript

```
// tasksApi.ts - API functions for tasks
export const fetchTasksPage = async ({ pageParam = 1 }) => {
```

```
  const res = await api.get(`/tasks?page=${pageParam}`);
  return res.data;
};
export const addTaskRequest = async (newTask) => {
  const res = await api.post('/tasks', newTask);
  return res.data;
};
export const updateTaskRequest = async ({ id, updates }) => {
  const res = await api.get(`/tasks/${id}`, updates);
  return res.data;
};

// tasksHooks.ts - hooks using above functions with React Query
import { useInfiniteQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { fetchTasksPage, addTaskRequest, updateTaskRequest } from './tasksApi';

export function useTasksList() {
  return useInfiniteQuery({
    queryKey: ['tasks'],
    queryFn: fetchTasksPage,
    getNextPageParam: (lastPage) => lastPage.nextPage ?? undefined
  });
}

export function useAddTask() {
  const queryClient = useQueryClient();
  return useMutation({
    mutationFn: addTaskRequest,
    onMutate: async (newTask) => {
      // Optimistically add new task to cache:
      await queryClient.cancelQueries({ queryKey: ['tasks'] });
      const prevData = queryClient.getQueryData(['tasks']);
      if (prevData) {
        queryClient.setQueryData(['tasks'], (oldData) => {
          const newTaskObj = { ...newTask, id: Math.random().toString(36) }; // temporary ID
          // Insert new task at the beginning of the first page (assume)
          return {
            ...oldData,
            pages: [
              [ newTaskObj, ...oldData.pages[0] ],
              ...oldData.pages.slice(1)
            ]
          };
        });
      }
      return { prevData };
    },
    onError: (err, newTask, context) => {
      // roll back changes on error
      if (context?.prevData) {
        queryClient.setQueryData(['tasks'], context.prevData);
      }
    },
    onSettled: () => {
      // regardless of result - refresh task list from server
      queryClient.invalidateQueries({ queryKey: ['tasks'] });
    }
```

```
  });
}

export function useUpdateTask() {
  const queryClient = useQueryClient();
  return useMutation({
    mutationFn: updateTaskRequest,
    onMutate: async ({ id, updates }) => {
      await queryClient.cancelQueries({ queryKey: ['tasks'] });
      const prevData = queryClient.getQueryData(['tasks']);
      if (prevData) {
        queryClient.setQueryData(['tasks'], (oldData) => {
          // Update task locally
          return {
            ...oldData,
            pages: oldData.pages.map(page =>
              page.map(task => task.id === id ? { ...task, ...updates } : task)
            )
          };
        });
      }
      return { prevData };
    },
    onError: (err, vars, context) => {
      if (context?.prevData) {
        queryClient.setQueryData(['tasks'], context.prevData);
      }
    },
    onSettled: () => {
      queryClient.invalidateQueries({ queryKey: ['tasks'] });
    }
  });
}
```

What's happening here:

- We have `useTasksList()` using `useInfiniteQuery` for the list. It uses `fetchTasksPage` (which does GET `/tasks?page=`). We assume the server returns, e.g., `{ tasks: [...], nextPage: 2 }` or `nextPage: null`. `getNextPageParam` function gets `lastPage.nextPage`.
- Mutation `useAddTask`:
  - `onMutate`: We cancel current `tasks` queries, fetch previous data. Then, if there was some data (cache exists), we optimistically insert the new task. Here for simplicity we generate a temporary id (random string) and add the new object to the start of the first page of the list (assuming we want it at the start). The cache structure in `infiniteQuery` is an object with `pages`, so we insert into `oldData.pages[0]`. We return `previousData` to be able to roll back.
  - `onError`: Upon error, we restore `prevData` if it existed.
  - `onSettled`: After everything (success or error), we invalidate `tasks` – server state should be the ultimate truth.
- Mutation `useUpdateTask`:
  - `onMutate`: similarly we cancel, fetch `prevData`. Then we iterate through all pages (`oldData.pages.map`), and map each list of tasks in a page – if `task.id === updatedId`, we create a new object with `updates` applied (e.g., `{ completed: true }`). This will

immediately change the UI – the user will see, e.g., that a task marked itself as completed. We return prevData.

- onError: On error we restore.
- onSettled: invalidation. This approach causes minimal perceived delay for the user – most actions seem instantaneous.

Step 3: UI Components Using These Hooks: We assume the login mechanism is behind us and we have a token (otherwise, a login screen and logic for saving the token in SecureStore after successful login etc. should be added). For the task list with infinite scroll:

JavaScript

```
// TasksListScreen.tsx
import { useTasksList } from './tasksHooks';

export function TasksListScreen({ navigation }) {
 const {
  data,
  isLoading,
  isError,
  error,
  fetchNextPage,
  hasNextPage,
  isFetchingNextPage,
  refetch
 } = useTasksList();

 if (isLoading) {
  return <ActivityIndicator size="large" color="#000" style={{ flex: 1, justifyContent: 'center' }} />;
 }
 if (isError) {
  return (
   <View style={{ padding: 20 }}>
    <Text style={{ color: 'red' }}>Error loading task list: {error.message}</Text>
    <Button title="Try again" onPress={refetch} />
   </View>
  );
 }

 const tasks = data.pages.flatMap(pageData => pageData.tasks); // assume API returns {tasks: [], nextPage: ...}

 return (
  <FlatList
   data={tasks}
   keyExtractor={task => task.id.toString()}
   renderItem={({ item }) => (
    <TaskListItem task={item} onPress={() => navigation.navigate('TaskDetails', { id: item.id })} />
   )}
   onEndReached={() => { if (hasNextPage) fetchNextPage(); }}
   onEndReachedThreshold={0.5}
   ListFooterComponent={ isFetchingNextPage ? <ActivityIndicator /> : null }
   refreshing={ isLoading }   {/* potentially pull-to-refresh handling */}
   onRefresh={ refetch }
  />
 );
}
```

Here:

- We use useTasksList(). When loading the first page (isLoading true), we show a large centrally located spinner.
- If error (isError), we show a message and a button that triggers refetch (re-fetching).
- When data exists, we join all pages into one tasks array.
- FlatList displays each task through the TaskListItem component (this can be, e.g., a list item with a completion checkbox, etc. – styling details omitted).
- onEndReached logic as discussed.
- refreshing/onRefresh also enables refreshing via gesture – we used isLoading which here upon returning to the screen could be false if data is in the cache. Alternatively, better: refreshing={isFetching && data} – i.e., if a fetch is in progress after initial load.
- (We assumed the API returns an object with tasks and nextPage keys. In our tasksApi functions we should potentially adapt this, but that's an implementation detail.)

Task addition/edit component: Assume we have a screen for adding a new task and editing an existing one (e.g., changing the title or marking as complete). We can solve this differently – e.g., a "TaskDetails" screen with an edit option. For simplicity, we'll just show what the usage of useAddTask and useUpdateTask hooks would look like:

JavaScript

```
// NewTaskScreen.tsx
export function NewTaskScreen({ navigation }) {
 const { mutate: addTask, isLoading: isAdding } = useAddTask();
 const [title, setTitle] = useState('');

 const handleSave = () => {
  addTask(
   { title, completed: false },
   {
    onSuccess: () => {
     navigation.goBack(); // after adding, go back to list
    }
   }
  );
 };

 return (
  <View style={{ padding: 20 }}>
   <Text>New task:</Text>
   <TextInput value={title} onChangeText={setTitle} placeholder="Task title" />
   {isAdding && <ActivityIndicator />}
   <Button title="Add" onPress={handleSave} disabled={isAdding || !title} />
  </View>
 );
}
```

Here useAddTask is used. When the user clicks "Add":

- We call addTask({ title, completed: false }, { onSuccess: ... }). You can pass a callback directly in the mutate call if we want to perform an action upon success (alternative to defining

onSuccess in the hook itself, which we did for invalidation anyway). Here after success we return to the previous screen.

- isAdding allows showing a spinner or blocking the button to not add multiple times.
- Thanks to optimistic list updating in onMutate, the user returning to the list will immediately see a new entry (semi-transparent to signal it's still saving – this would need a conditional style added in TaskListItem, e.g., if id is temporary, or one could base it on useAddTask state as in the UI optimistic example earlier). In our code above we gave ActivityIndicator for isAdding element – that's a simplified example, in practice we'd prefer, e.g., passing some flag or using variables from useMutation.

Edit screen (e.g., mark as completed):

JavaScript

```
// TaskDetailsScreen.tsx
export function TaskDetailsScreen({ route, navigation }) {
 const { id } = route.params;
 const { mutate: updateTask, isLoading: isUpdating } = useUpdateTask();
 const task = // ... assume we passed the task object in navigation or have a useQuery for single task

 const toggleComplete = () => {
  updateTask({ id, updates: { completed: !task.completed } }, {
   onSuccess: () => {
    // can navigate back or notify
   }
  });
 };

 return (
  <View>
   <Text>{task.title}</Text>
   <Text>Status: {task.completed ? 'completed' : 'to do'}</Text>
   {isUpdating && <ActivityIndicator />}
   <Button title={task.completed ? "Mark as incomplete" : "Mark as completed"}
        onPress={toggleComplete} disabled={isUpdating} />
  </View>
 );
}
```

Here useUpdateTask will work optimistically – it will immediately change the status on the list (if the list is displayed somewhere, e.g., previous screen), because onMutate will update the cache. When we return to the list (if using react-navigation stack, the list has likely already been updated in the background thanks to this), the user sees the change right away.

Token handling: thanks to the interceptor, our api.get, api.post requests automatically attach the token. If the token was outdated, the refresh interceptor will try to fix it. **Global error:** if refreshing fails and the interceptor throws a 401 error, we can catch it, e.g., with a global error boundary or detect in navigation and move the user to the login screen. You can also check in the mutation's onError if the error has a 401 status and, for example, display a "Session expired" modal. **No network:** It's worth adding NetInfo listening and e.g., when isConnected === false, then:

- Block addition/edit buttons (or give a "Offline - cannot synchronize changes" message).
- Show an offline banner as mentioned earlier. React Query along with onlineManager can also cause that when the network returns, it automatically refreshes data or executes pending mutations (although offline mutations by default return an error immediately – TanStack Query has separate plugins for persisting offline mutations, but that's an advanced topic).

Debugging: While creating such an app:

- Use logs (console.log) inside onError to check what went wrong.
- Check in Flipper whether requests are actually going out, what URLs and statuses are returning.
- Use debug mode in React Query – e.g., enableDevTools() in RN is not automatic, but you can use the Flipper plugin for React Query or Reactotron plugin. The mentioned Flipper plugin (react-query-native-devtools) allows inspecting active queries, cache, etc., which can be immensely helpful in understanding what's happening in React Query. In case of cache problems, you can also call queryClient.getQueryData(['tasks']) in the debugger console to see what's in memory.

Summary: Utilizing React Query in React Native can significantly simplify working with the network. We get automatic handling of query states, data cache, refreshing, as well as advanced mechanisms like optimistic updates or infinite scroll, which in a pure approach would require a lot of code. Combining this with the axios library (or native fetch) allows covering most needs: global headers, token handling via interceptors, and secure storage of credentials. Remember to handle errors – both those from the API (messages for the user) and JS exceptions (Error Boundaries). Thanks to this, the app will react in a clear way even in problematic situations, improving the user experience.

Literature:

1. https://reactnavigation.org/docs/getting-started/ (Access Date: 1.10.2025) - Official React Navigation documentation (main page).
2. https://reactnavigation.org/docs/typescript/ (Access Date: 1.10.2025) - Official guide for integrating React Navigation with TypeScript.
3. https://reactnavigation.org/docs/auth-flow/ (Access Date: 1.10.2025) - Key documentation describing the recommended authentication flow pattern.
4. https://reactnavigation.org/docs/deep-linking/ (Access Date: 1.10.2025) - Official guide for configuring Deep Links.
5. https://reactnavigation.org/docs/navigating/ (Access Date: 1.10.2025) - Documentation for basic operations (navigate, push, goBack).
6. https://reactnavigation.org/docs/params/ (Access Date: 1.10.2025) - Documentation on passing and receiving parameters (route.params).
7. https://reactnavigation.org/docs/hooks/ (Access Date: 1.10.2025) - Documentation for useNavigation and useRoute hooks.
8. https://reactnavigation.org/docs/native-stack-navigator/ (Access Date: 1.10.2025) - Documentation for Native Stack Navigator (recommended for performance).
9. https://reactnavigation.org/docs/bottom-tab-navigator/ (Access Date: 1.10.2025) - Documentation for Bottom Tab Navigator.
10. https://docs.expo.dev/routing/linking/ (Access Date: 1.10.2025) - Expo guide regarding linking configuration (including expo-linking).