Modelling exercise

Let's assume we have to write the frontend of a netbank application in Typescript. Let's also assume we want to hold the whole state of the application in one complex value. The purpose of this exercise is to model, to define the type (which can be composed of many other types), of that value, with all the possible states that the application can be in.
This is something that you might not do in a real app, since the state tends to be spread in different components on different pages, but the goal is to try to see how to model problems and compose those models. You will try to write this big complex type in such a way as to express all the requirements below and minimise or disallow all the states and combinations of states that should not happen. If something should be impossible based on the requirements, the type should ideally reflect that, if it can. If something should be possible then the type should allow for that. Note that you can define many helper types.

The goal is NOT to write any logic for the below requirements, only the type of the state of the app which would allow one to implement the below requirements. For example, if the app can have only one page active at any given time, we would model that with
"`type ActivePage = PageOne | PageTwo | PageThree`".

If some requirement is unclear or you don't know how to express it, don't worry, the purpose of this exercise is to see how you justify certain decisions and to guide a conversation about modelling problems. Your skills won't be primarily judged based on how complete your model is.

The netbank

The user is either logged in or not. If he is logged in, he has an email and a name which are fetched asynchronously, therefore there is a full-screen loading page that we show while the user's data is fetched. If this fails, we show a full-screen error which distinguished between an error on the user's side (network went down) or on our side (server error or request timed out).

The app has 3 pages: accounts, international payments, support.
Each page loads its data asynchronously, and each page has a full-screen loading and an error state.

The loading state shows an extra message if the request takes takes longer than usual to complete.
The error pages shows a message either that it is the app's fault (either a server error or the request timed out) or the user's (ie. The user went into a cave and lost connection) so we need to distinguish between these.

The accounts page:

Loads the user's accounts which can be of different types: regular accounts and pocket accounts. The user has at least one regular account.

A regular account has a name and an IBAN and uses the local currency which is DKK. The amount can be positive or negative.

A pocket account also has a name and an IBAN but can have multiple "pockets" with different amounts in different currencies at the same time. The currencies can be one of: SEK, NOK, USD or EUR. A pocket account has at least one amount in one of these currencies and it cannot have multiple "pockets" with the same currency at the same time.

The International payments page

The **international payments** page allows the user to type an IBAN and a SWIFT code. After the user types in the IBAN, the SWIFT code input becomes disabled while the app tries to fill it asynchronously. If it succeeds, the SWIFT input is enabled again and filled with a SWIFT code based on the IBAN. If it fails the field becomes enabled again but without any pre-filled value.
If instead, the user typed in something in the SWIFT code field before the IBAN field, the above logic should not happen. The SWIFT code should remain as it is and not be pre-filled after the IBAN is typed.

The "from" account is preselected as the first regular account but the user can select either another regular account or a pocket account to pay from. If the user pays from a pocket account which has more than one "pocket" of currency he should also select which currency to pay from.

If the account number is valid, and the amount is valid then the form is valid so the user can do the transfer.

The transfer happens asynchronously and can either succeed, in which case the user sees a receipt with the details of the transfer and the time or fail in which case the user sees an error with a generic message.

The support page

This page only has one button: start chat. After a user presses it, the page tries to connect with an agent. As soon as the user tries to connect with an agent, the chat information is available on all pages. If the user moves away from the support page, a chat window will be floating on all screens.

The chat window can be "**minimized**", "**open**" or "**full-screen**". A chat window can be "minimized" or "open" on any page in the app except the support page. When the user navigates to the support page the chat is always "full-screen". The chat window becomes minimized if the user moves from the support page to any other page.

If after 15 seconds the chat didn't connect with an agent we show a message that the queue is too long and the user's place in the queue (ie. "You are 4th in the queue"). If the place in queue is greater than 10 we also asynchronously load information about which times and days are most and least busy in the week and show that (but with no loading or error state).

Once the chat connects with an agent, the user can type a message and can see the messages thread. Once the user closes the chat, no floating chat window will appear anywhere and if the user visits the support page again he again sees the "start chat" button.