GLOBAL STATE MANAGEMENT

FHS (TMAYRHOFER.LBA@FH-SALZBURG.AC.AT)

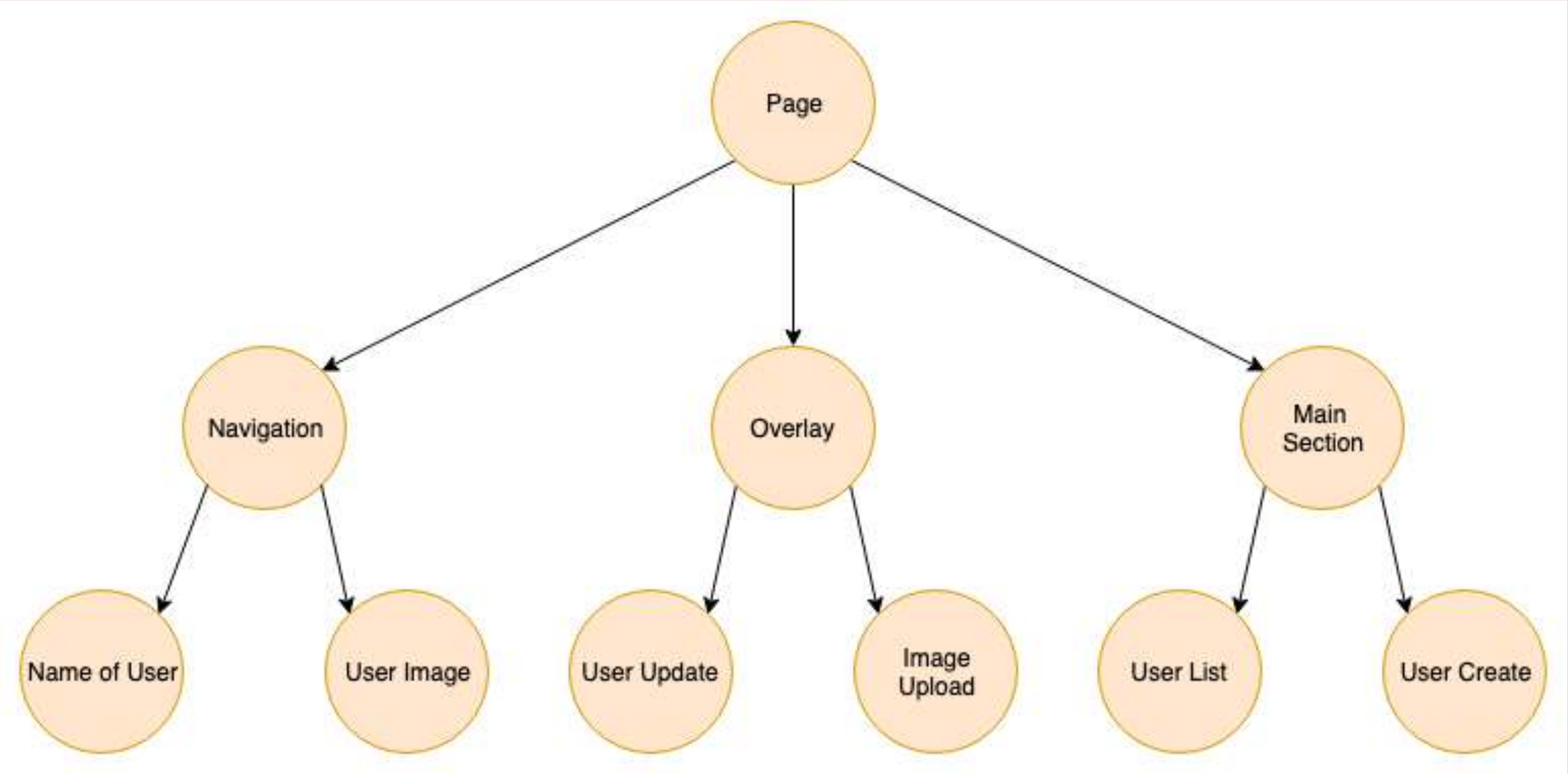
APPLICATION STATE
WHAT IS APPLICATION STATE

“An application's state is roughly the entire contents of its memory. (sarnold)”

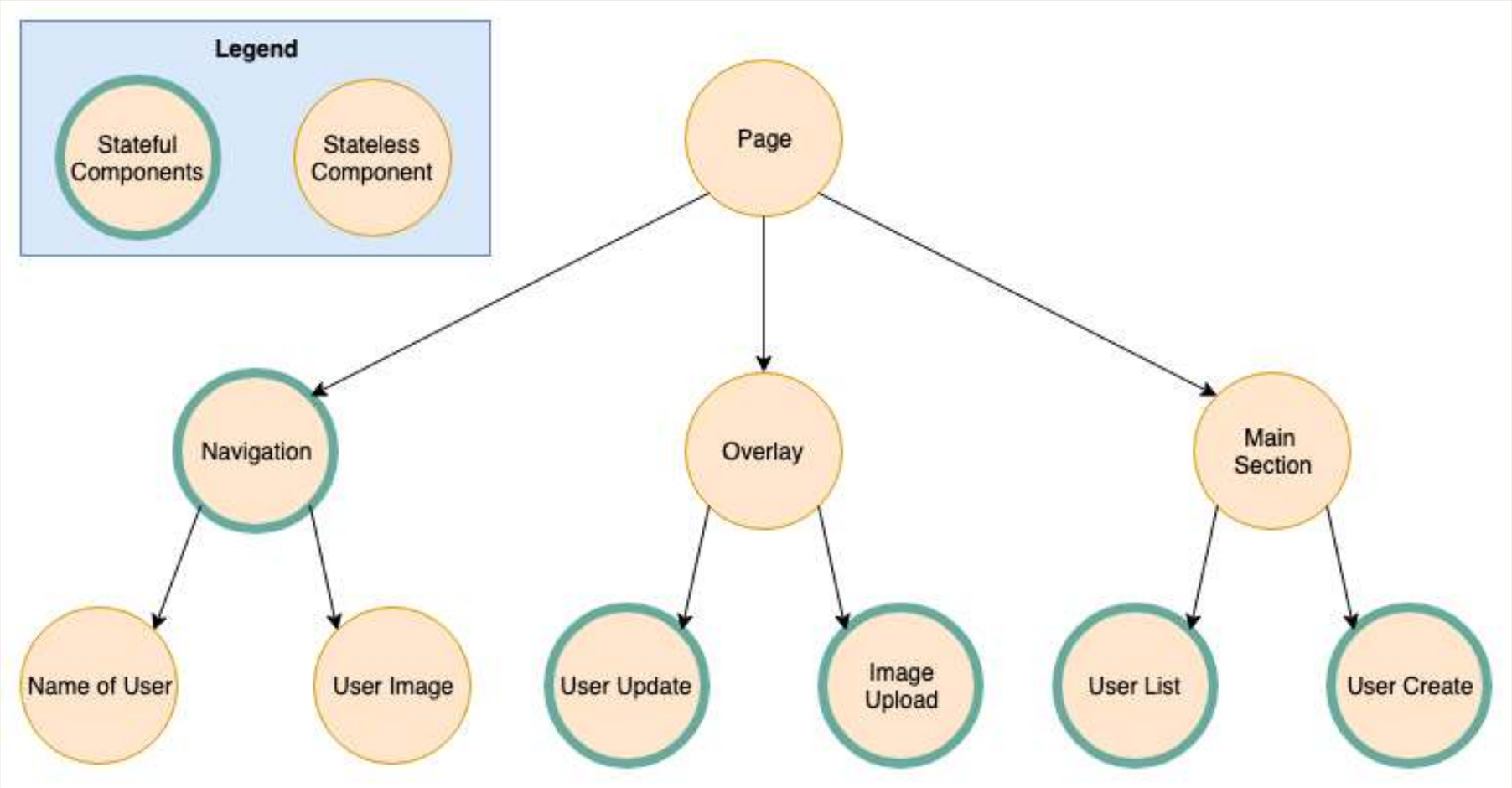
APPLICATION STATE
STATE IN REDUX TERMS

“Every bit of information the application needs in order to render.”

APPLICATION STATE
REACT COMPONENT TREE



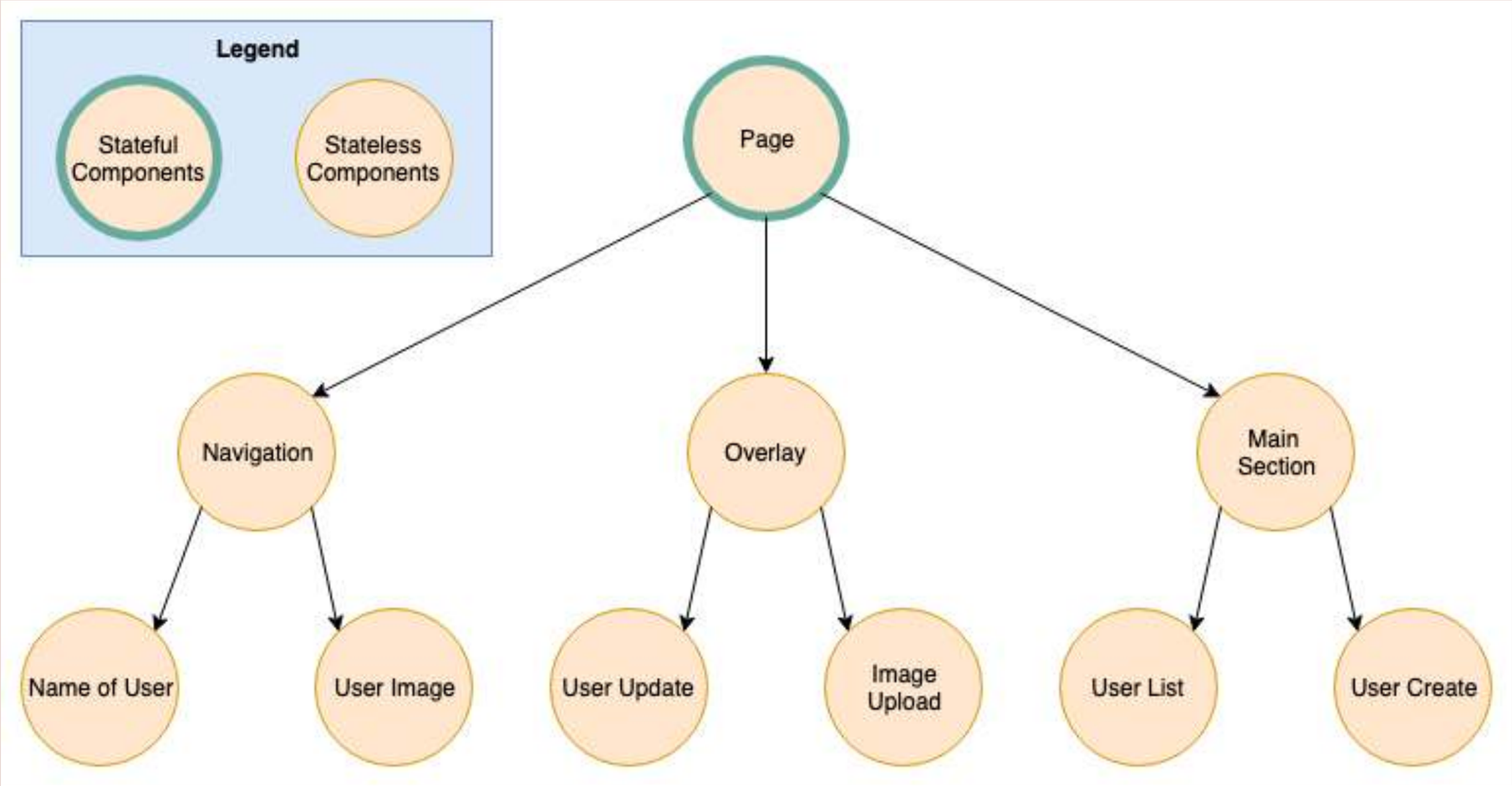
APPLICATION STATE
STORING STATE IN COMPONENTS



APPLICATION STATE
STORING STATE IN COMPONENTS

- Pros
  - Components are independent
  - eg. "Navigation" doesn't know about "User Update"
- Cons
  - User data needs to be fetched multiple times
  - If UserUpdate component changes name of user

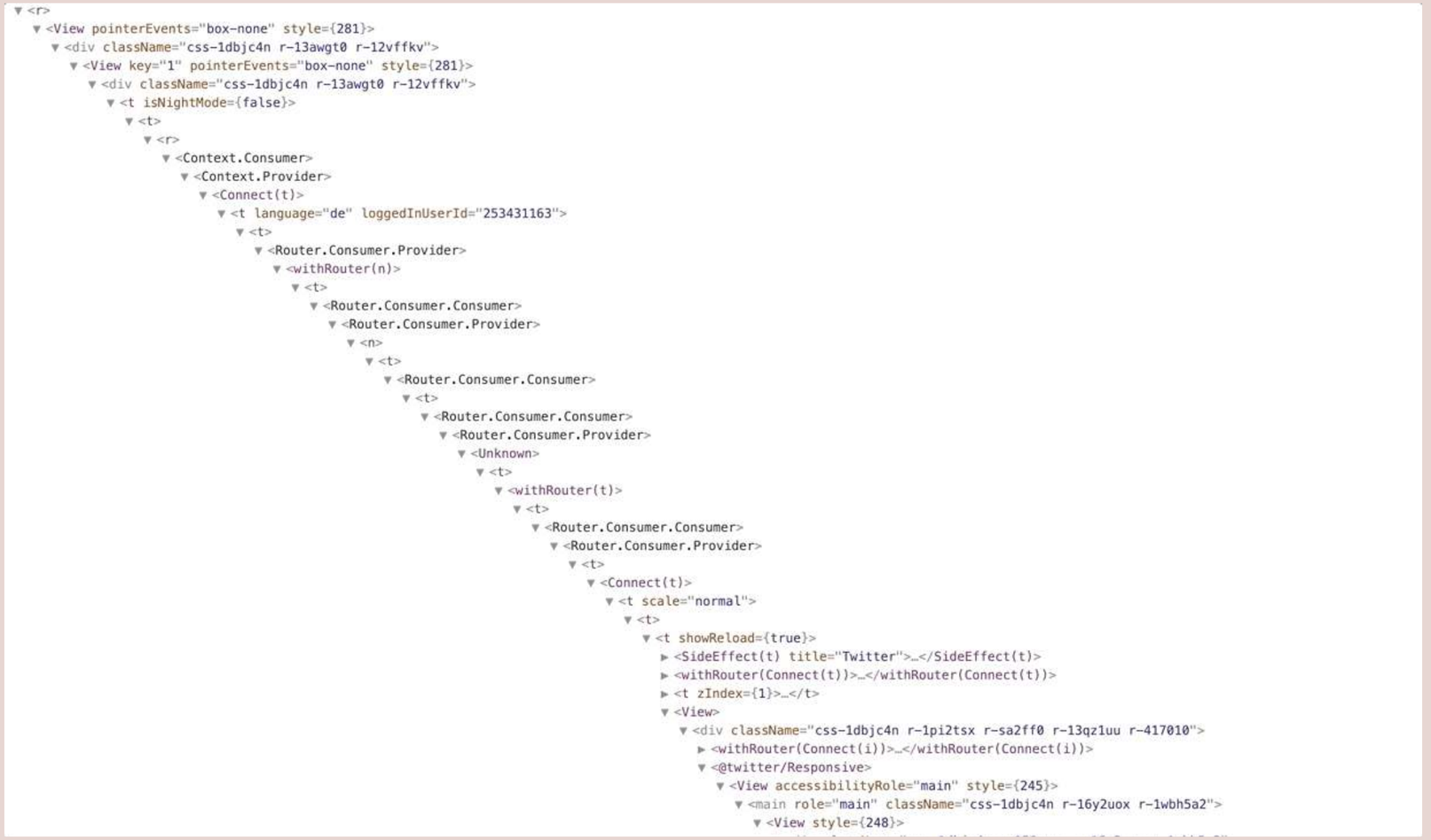
APPLICATION STATE
STORING STATE IN THE ROOT COMPONENT



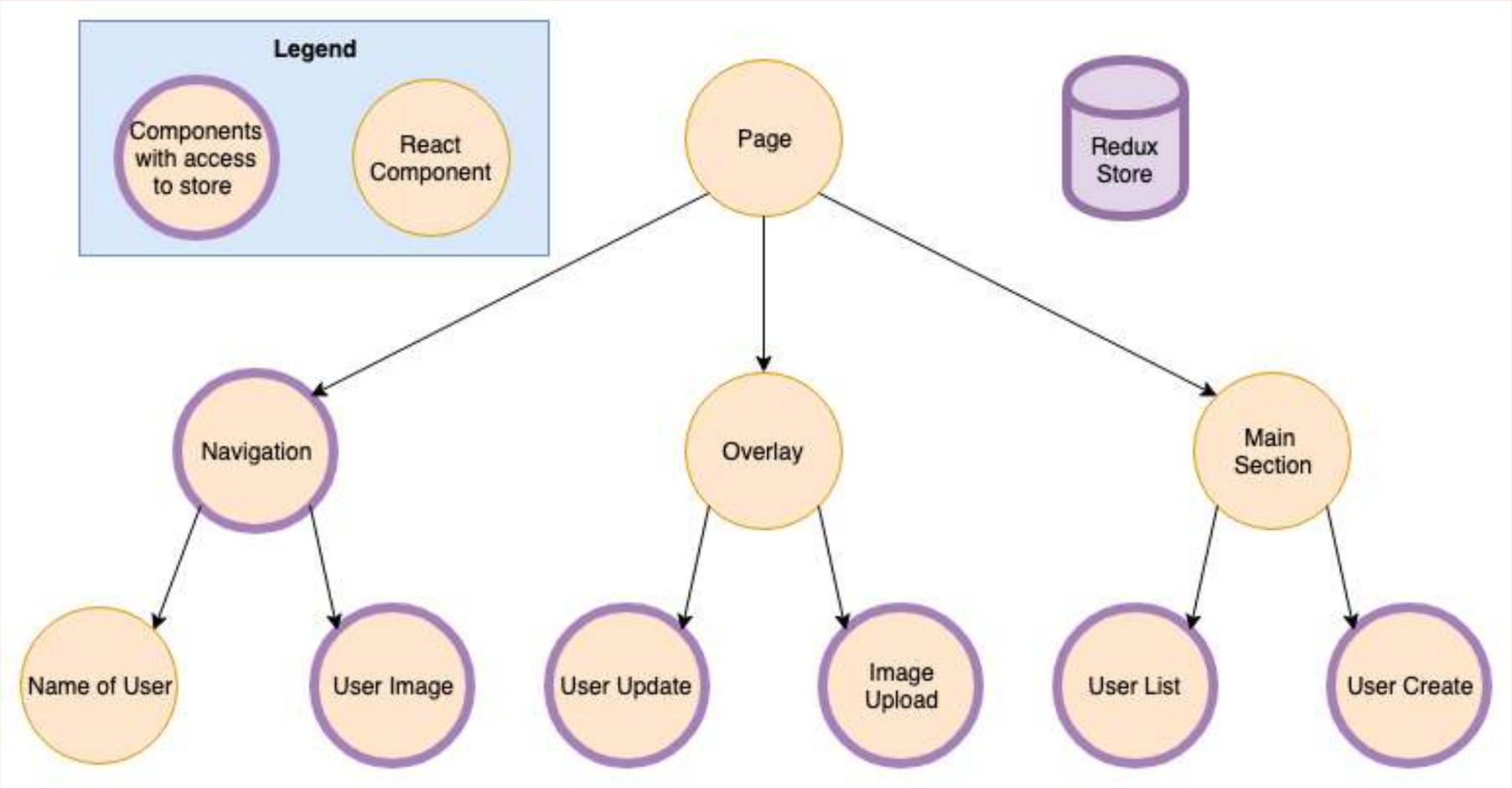
APPLICATION STATE
STORING STATE IN THE ROOT COMPONENT

- Pros
  - User data could be fetched only once
  - If UserUpdate component changes name of user
  - navigation component is automatically updated
- Cons
  - State needs to be passed down to every component
  - (Root component contains all state logic)

APPLICATION STATE
STORING STATE IN THE ROOT COMPONENT



APPLICATION STATE
STORING STATE GLOBALLY



APPLICATION STATE
STORING STATE GLOBALLY

- Global state which acts like local state
- Pros:
  - Components are independent
  - eg. Navigation doesn't know about UserUpdate
  - State changes are synchronised with the whole app
  - State doesn't need to be passed down the tree

ZUSTAND

» Small Statemanagement Solution for react

» react hooks based

» easy to get started

» no context providers needed

ZUSTAND
INSTALLATION

```
npm install zustand
```

ZUSTAND
CREATE GLOBAL STATE

```
import { create } from 'zustand'

const useUsers = create((set) => ({
//                      ^^^^^^
// create some new state/zustand

  users: [],
  addUser: () => set({ users: [{ id: 1, name: "Sepp" }] }),
}))
```

ZUSTAND
CREATE USE STATE

```
const UserList = () => {
  const users = useUsers((state) => state.users)
  //                                 ^^^^^^^^^^^
  // extract users from our zustand

  const addUser = useUsers((state) => state.addUser)
  //                                   ^^^^^^^^^^^^
  // extract mutation from our zustand

  // ....
}
```

ZUSTAND
TRIGGER SIDE EFFECTS

```
import { create } from 'zustand'

const useUsers = create((set) => ({
  users: [],
  fetchUsers: async (state) => {
// 1)                  ^^^^^^
//
    const users = await fetch('./users')

    set({ users: uniqueBy('id', [...state.users, ...users]) })
//           3)  ^^^^^^^^   2) ^^^^^^^^^^^^^^^^^^^^^^^^^^
  },
}))
// 1) current state can be accessed via the argument
// 2) merging the existing users with the already existing users
// 3) remove duplicate users (use lodash)
```

FUNCTIONAL PROGRAMMING
IMMUTABILITY

“An immutable data structure is an object that doesn't allow us to change its value. (Remo H. Jansen)”

FUNCTIONAL PROGRAMMING
IMMUTABLE OBJECTS IN JS

```
const immutableObject = Object.freeze({ test: 1 })
immutableObject.test = 10
console.log(immutableObject) // => { test: 1 }
```

FUNCTIONAL PROGRAMMING
CHANGING AN IMMUTABLE VALUE

```
const immutableObject = Object.freeze({ a: 1, b: 2 })
const updatedObject = Object.freeze({ ...immutableObject, a: 2 })
console.log(updatedObject) // => { a: 2, b: 2 }
```

FUNCTIONAL PROGRAMMING
CHANGING AN IMMUTABLE VALUE

“Working with deeply nested objects is tough”

FUNCTIONAL PROGRAMMING
WORKING WITH NESTED OBJECTS

```
const nextState = baseState.slice() // shallow clone the array
nextState[1] = {
    // replace element 1...
    ...nextState[1], // with a shallow clone of element 1
    done: true // ...combined with the desired update
}
nextState.push({title: "Tweet about it"})
```

FUNCTIONAL PROGRAMMING
WORKING WITH NESTED OBJECTS

“Working with deeply nested objects is tough”

FUNCTIONAL PROGRAMMING
IMMER TO THE RESCUE

```
import produce from "immer"

const baseState = [
    {
        title: "Learn TypeScript",
        done: true
    },
    {
        title: "Try Immer",
        done: false
    }
]

const nextState = produce(baseState, draft => {
//                          ^^^^^^^^^
// object to be mutated
    draft[1].done = true
//  ^^^^^^
// Instead of mutating the state we're mutating the draft
    draft.push({title: "Tweet about it"})
})

// baseState !== nextState
```

FUNCTIONAL PROGRAMMING
IMMER TO THE RESCUE

» immutability without new api

» strongly typed

» object freezing out of the box

» deep updates are easy

» small 3kb in size

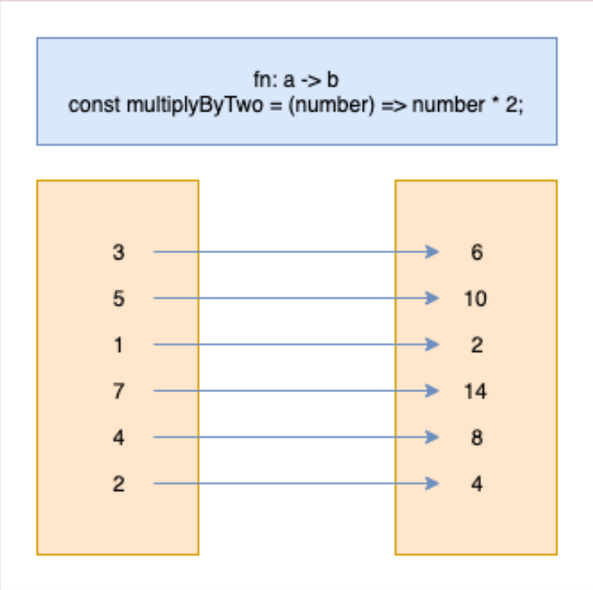
FUNCTIONAL PROGRAMMING
MEMOIZATION

“`Memoizing' a function makes it faster by trading space for time. It does this by caching the return values of the function in a table. (https://metacpan.org/pod/Memoize)”

PURE FUNCTIONS RECAP

» A pure function returns for the same input the same output

» simple mapping from value a to value b



MEMOIZING
MEMOIZEE

```
» $ npm install memoizee
```

```
import memoize from "memoizee";

const fibonacci = memoize((num) => {
  if (num <= 1) return 1
  return fibonacci(num - 1) + fibonacci(num - 2)
})
```

MEMOIZING
MEMOIZEE

» clearing the cache

```
const fibonacci = memoize((num) => {
  if (num <= 1) return 1
  return fibonacci(num - 1) + fibonacci(num - 2)
})

fibonacci() // will execute
fibonacci() // cache hit
fibonacci.clear();
fibonacci() // will execute
```

FEEDBACK

» Questions: tmayrhofer.lba@fh-salzburg.ac.at

» https://s.surveyplanet.com/x1ibwm85