FUNCTIONAL PROGRAMMING AND STATE MANAGMENT IN REACT

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

FUNCTIONAL PROGRAMMING

FUNCTIONAL PROGRAMMING
WHAT IS FUNCTIONAL PROGRAMMING

“Applications developed in a functional style use side-effect free functions as their main building blocks. (Made up definition by myself)”

FUNCTIONAL PROGRAMMING
FP VS. OOP

“Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts. (Michael Feathers)”

FUNCTIONAL PROGRAMMING
WHY FUNCTIONAL PROGRAMMING

- More testable
  - pure functions simplify testing
- Declarative APIs which are easier to reason about
- Easy concurrency because of statelessness and immutability
  - State is pushed out of the application core to the boundaries

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
UNFREEZE AN OBJECT

```
const immutableObject = Object.freeze({ test: 1 })
const unfrozenCopy = { ...immutableObject }
```

FUNCTIONAL PROGRAMMING
OBJECT.FREEZE IS MUTABLE

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

FUNCTIONAL PROGRAMMING
WHY IMMUTABILITY

» race conditions impossible

» state of the application is easier to reason about

» easier to test

FUNCTIONAL PROGRAMMING
MUTABLE BUG

```
const users = []
const loadUsers = async () => {
  const result = await fetchUsers('/users')
  users.push(...result)
  return users
}

loadUsers()
loadUsers()
```

FUNCTIONAL PROGRAMMING
IMMUTABLE VERSION

```
const loadUsers = () => {
  return fetchUsers('/users');
}

const result1 = await loadUsers();
const result2 = await loadUsers();
```

FUNCTIONAL PROGRAMMING
HIGHER ORDER FUNCTIONS

"A higher order function is a function that returns a function."

FUNCTIONAL PROGRAMMING
HIGHER ORDER FUNCTIONS

```
const buildCreateUser = (dbAdapter) => {
  return (user) => {
    if (!isValid(user)) { throw new Error('User Invalid') }
    return dbAdapter.create(user)
  }
}
const createUserInPG = buildCreateUser(postgresAdapter)
const createUserInMemory = buildCreateUser(inMemoryAdapter)
```

GLOBAL STATE MANAGEMENT

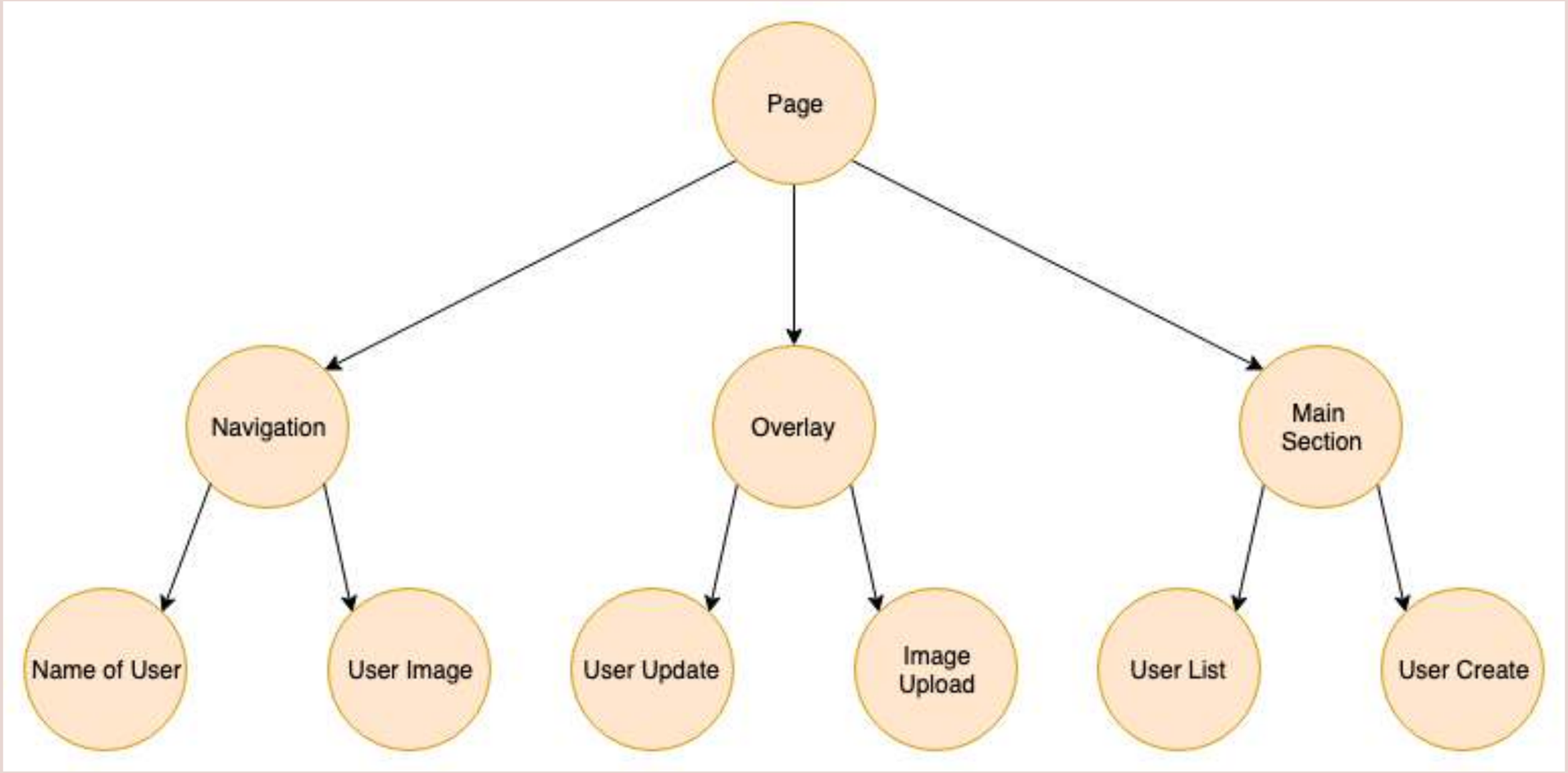
APPLICATION STATE
WHAT IS APPLICATION STATE

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

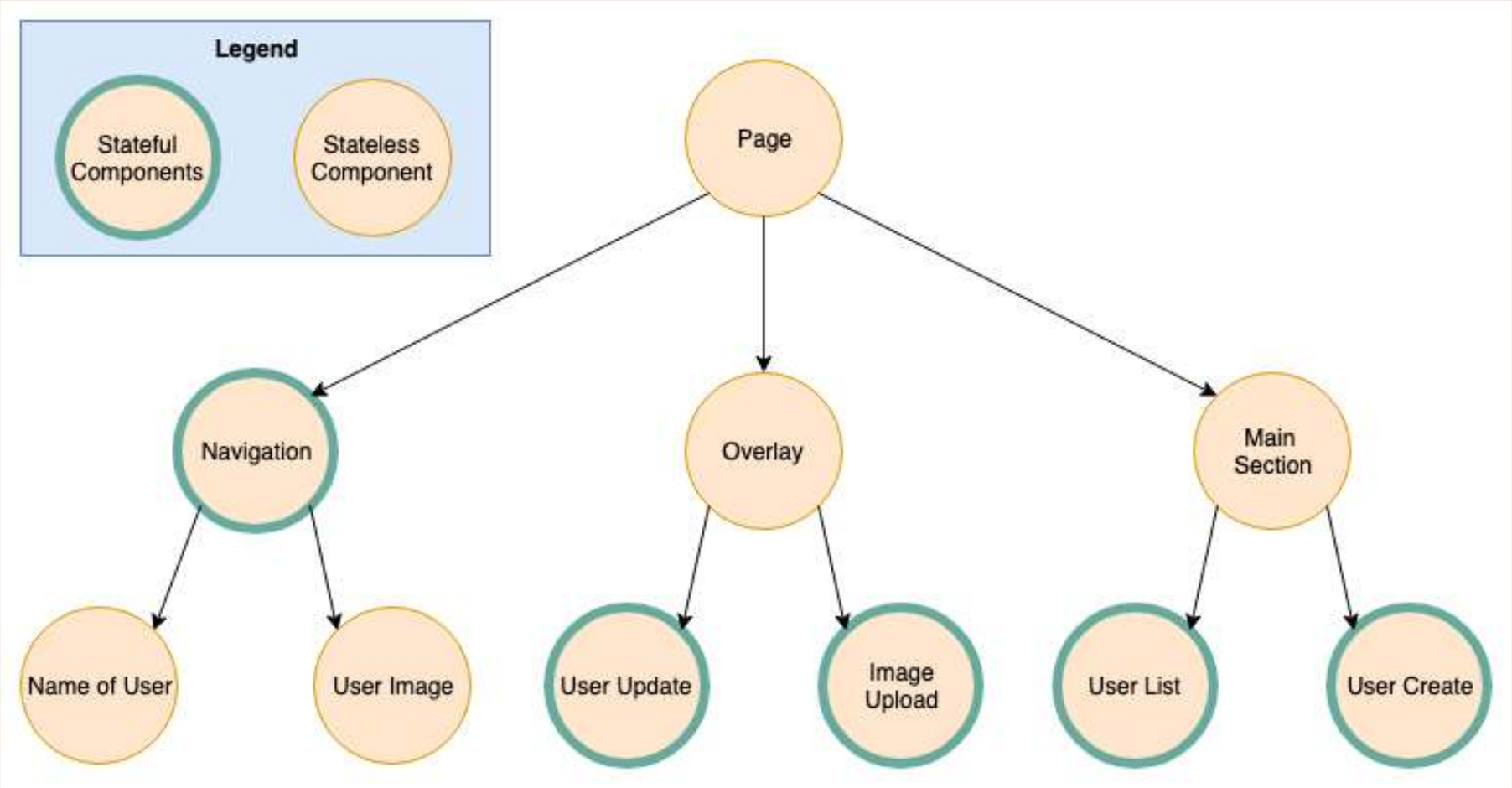
APPLICATION STATE
STATE IN REDUX TERMS

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

APPLICATION STATE
REACT COMPONENT TREE



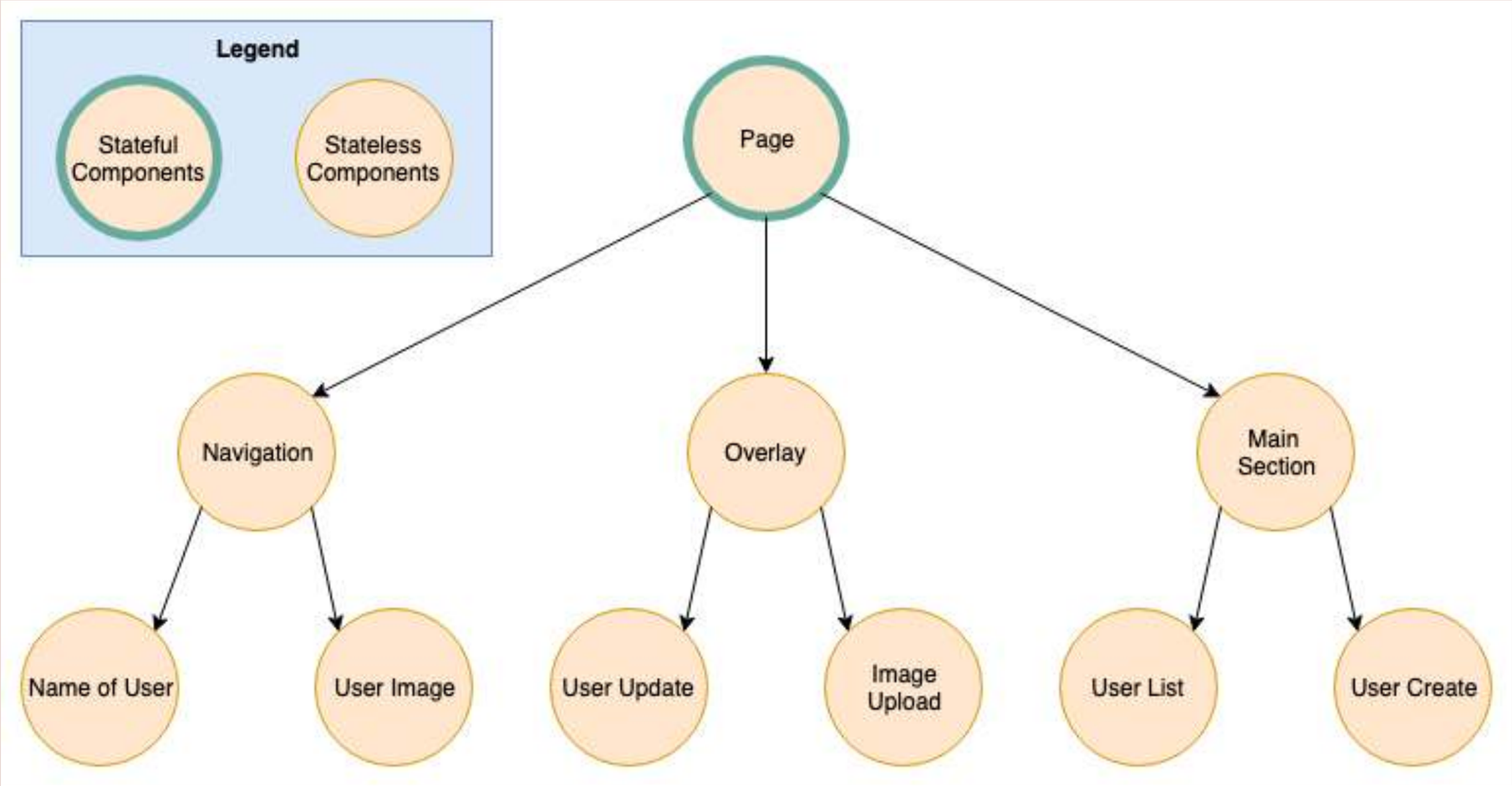
APPLICATION STATE
STORING STATE IN COMPONENTS



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

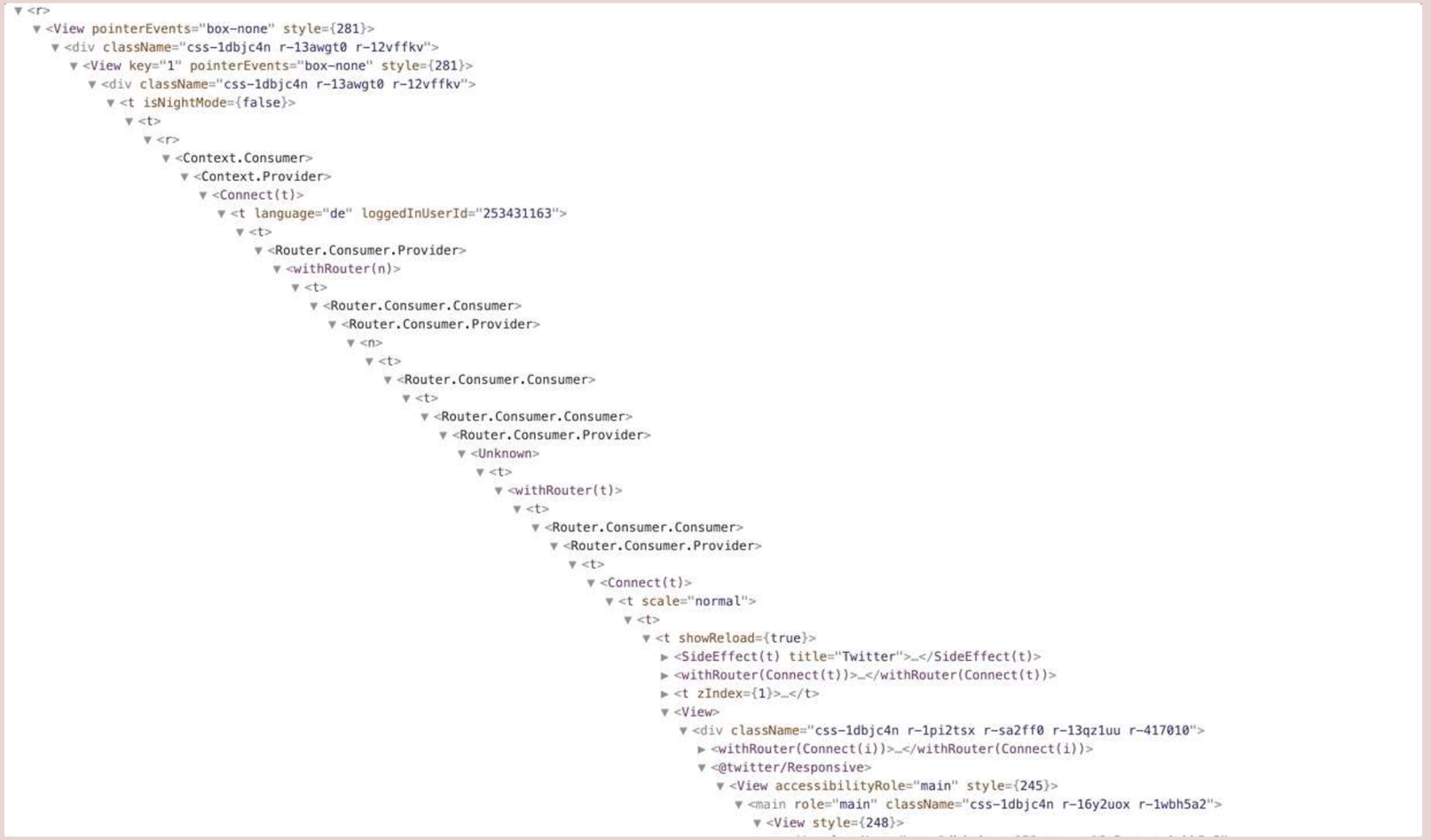
APPLICATION STATE
STORING STATE IN THE ROOT COMPONENT



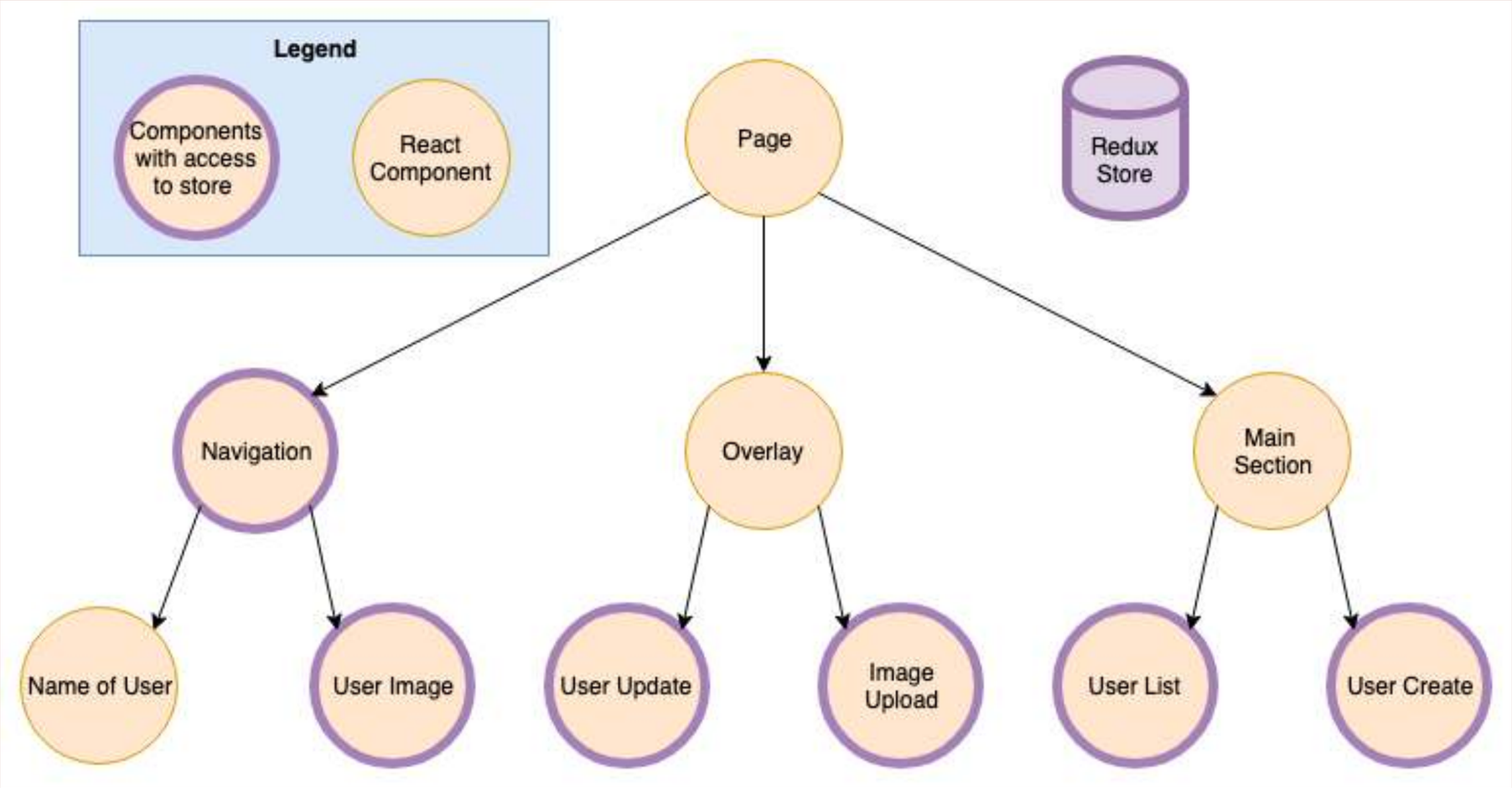
APPLICATION STATE
STORING STATE IN THE ROOT COMPONENT

- 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

CUSTOM STATE MANAGEMENT

FRAMEWORK AGNOSTIC STATE MGMT
CREATING A LIBRARY AGNOSTIC STORE

» Create a library agnostic store to be used by any framework

```
// Interface
type CreateStore = <T>(stateFactory: () => T) => {
  set: (updateFn: (state: T) => T) => unknown
  get: () => T
}
```

FRAMEWORK AGNOSTIC STATE MGMT
TASK

» Create an implementation for CreateStore type

» Add unit tests to your implementation

» Usage:

```
const store = createStore(() => { someValue: 1 })

store.get() // 1
store.set((current) => ({ someValue: current.someValue + 1}))
store.get() // 2
```

REACT STATE MGMT ADAPTER

» Interface for wrapping the agnostic state management

```
type UseReactStore<T> = () => [
  T,
  (updateFn: (state: T) => T) => unknown
]

type CreateReactStore = <T>(stateFactory: () => T) => UseReactStore<T>
//                                                    ^^^^^^^^^^^^^^^^^
// higher order function which returns a hook
```

REACT STATE MGMT ADAPTER TASK

» create an implementation for CreateReactStore

» Usage:

```
const useMyStore = createReactStore(() => ({ someValue: 1 }))
//    ^^^^^^^^^^
// useMyStore is defined globally/outside of the react context

const MyComponet = () => {
   const [myState, setMyState] = useMyStore()
   // ...
}
```

BUILD ADAPTER FOR REACT
ISSUE

» Did you encounter any issues?

BUILD ADAPTER FOR REACT
ISSUE

» state is not updated in components

BUILD ADAPTER FOR REACT
USESYNCEXTERNALSTORE

"useSyncExternalStore is a React Hook that lets you subscribe to an external store."

```
const todos = useSyncExternalStore(store.subscribe, store.getSnapshot);
// 1)                                ^^^^^^^^^^^^^^^^
// 2)                                                  ^^^^^^^^^^^^^^^^^

// 1) callback function which adds the current component to the list of components
//    to be updated in case the state changes. (similar to observer pattern)
// 2) callback function which returns an immutable snapshot of the current state.
//    This function is used to get the state into the react component.
```

BUILD ADAPTER FOR REACT
SUBSCRIBE FUNCTION

» callback function which adds/removes components to be notified on state changes

```
type Listener = () => unknown
const createMyStore = () => {
  // ...
  const listeners: Listener[] = []

  return () => {
    const subscribe = (listener: Listener) => {
      listeners.push(listener)
// 1)             ^^^^^^^^^^^^^^
      return () => listeners.splice(listeners.indexOf(listener), 1)
// 2)         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
      }
  }
}
// 1) adds listener to be called when the store changes
// 2) removes listener when component gets unmounted
```

BUILD ADAPTER FOR REACT
NOTIFY COMPONENTS ABOUT CHANGES

» notify react about state changes

```
const createMyStore = <T>() => {
  const listeners: Listener[] = []
  const emitChanges = () => listeners.forEach((listener) => listener())
//                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// notify components about state changes

  return () => {
    const set = (updateFn: (state: T) => T) => {
        store.set(updateFn)
        emitChanges()
    }
    // ...
  }
}
```

BUILD ADAPTER FOR REACT
GETSNAPSHOT FUNCTION

» callback function which adds/removes components to be notified on state changes

```
const createMyStore = () => {
// ...

  const getSnapshot = () => store.get()
//                     ^^^^^^^^^^^^^^^^^^
// function which returns the current snapshot of the store
}
```

BUILD ADAPTER FOR REACT
COMBINE CALLBACK FUNCTIONS

```
const createMyStore = () => {
// ... listeners, emitChanges
  return () => {
    // ...
    const state = useSyncExternalStore(subscribe, getSnapshot);

    return [
      state,
      set
    ]
  }
}
```

BUILD ADAPTER FOR REACT
TASK

» Add useSyncExternalStore to your state management solution

» Verify that connected components are updated

FUNCTIONAL PROGRAMMING
PURE FUNCTIONS

- A function is considered pure when:
  - for the same input it always returns the same output
  - it has no side effects
  - no mutation of non-local state,

```
const add = (a, b) => a + b
```

FUNCTIONAL PROGRAMMING
ATTRIBUTES OF PURE FUNCTIONS

- They are idempotent
- They offer referential transparency
  - calls to this function can be replaced by the value without changing the programs behaviour
- They can be memoized (or cached)
- They can be lazy
- They can be tested more easy

FUNCTIONAL PROGRAMMING
PURE OR INPURE? 1/3

```
const array = [1, 2, 3, 4, 5, 6]
const fn1 = (array) => array.slice(0, 3)
const fn2 = (array) => array.splice(0, 3)
const fn3 = (array) => array.shift()
const fn4 = (array) => array.pop()
const fn5 = (array) => array.sort((a, b) => a - b)
const fn6 = (array) => [...array].sort((a, b) => a - b)
const fn7 = (array) => array.map((item) => item * 2)
const fn8 = (array) => array.forEach((item) => console.log(item))
```

FUNCTIONAL PROGRAMMING
PURE OR INPURE? 2/3

```
const config = { minimumAge: 18 }
const isAllowedToDrink = (age) => age >= config.minimumAge

const config = { minimumAge: 18 }
const isAllowedToDrink = (age) => age >= config.minimumAge
```

PURE OR INPURE? 3/3

```
const isIndexPage = () => window.location.pathname === '/';
const isIndexPage = (pathname) => pathname === '/';
```

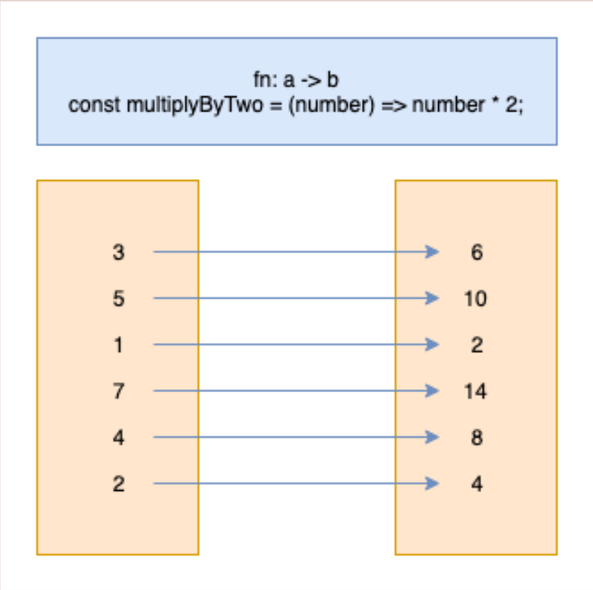
FUNCTIONAL PROGRAMMING
MEMOIZATION

“`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



FUNCTIONAL PROGRAMMING
TASK

» Memoize the fibonacci sequence

» Compare results with and without memoize

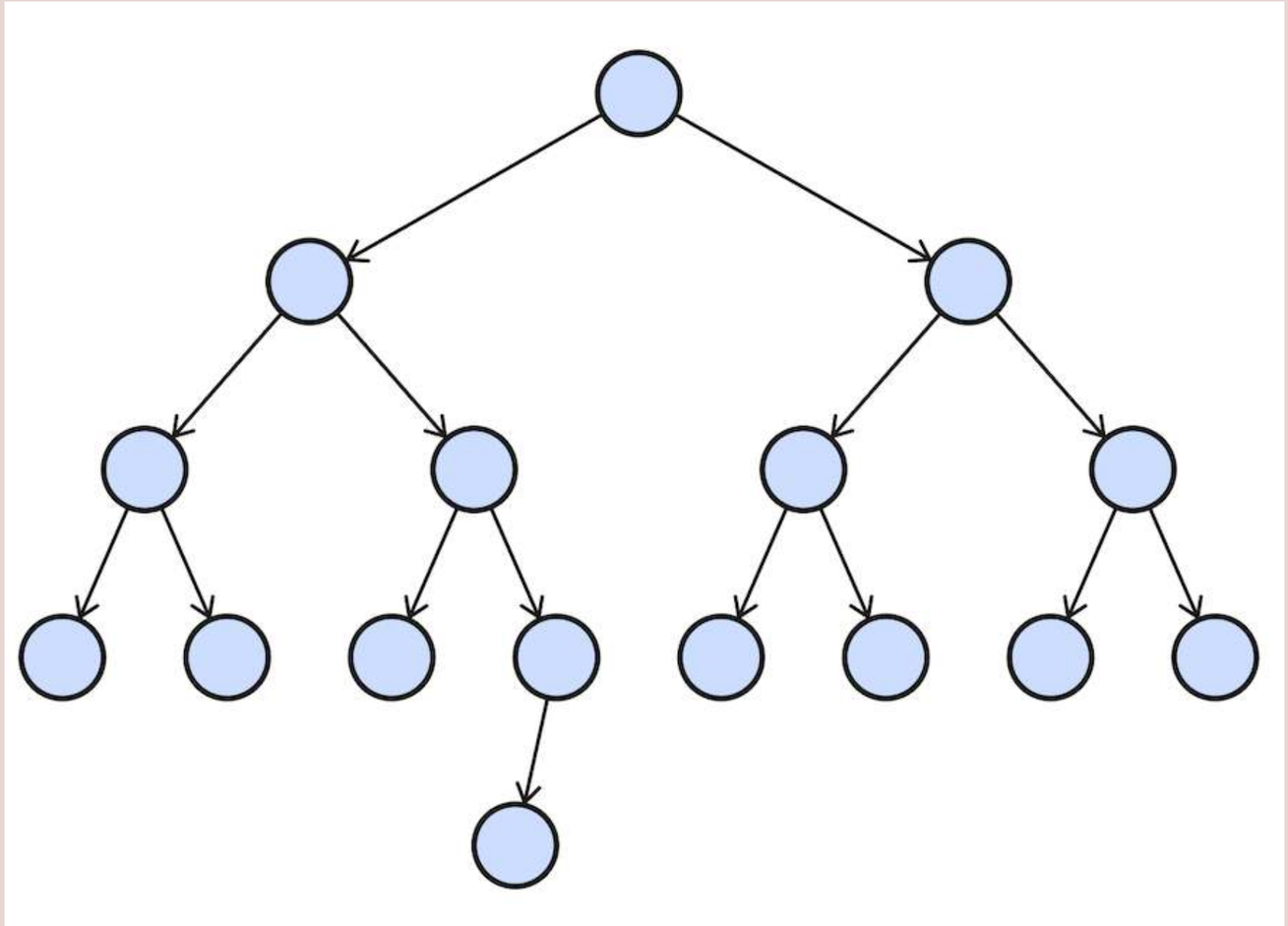
```
const memoize = () => {} // TODO: implement me
const fibonacci = memoize((num) => {
  if (num <= 1) return 1
  return fibonacci(num - 1) + fibonacci(num - 2)
})
```

» helper to measure time https://bit.ly/2UOFgAE

FUNCTIONAL PROGRAMMING
POSSIBLE IMPLEMENTATION

» Will be added after the lecture

FUNCTIONAL PROGRAMMING
REACT AND MEMOIZATION



FUNCTIONAL PROGRAMMING
WHERE IS MEMOIZATION USED?

```
import React, {useMemo} from 'react'

const MyComponent = ({ users }) => {
  const activeUsers = useMemo(
//                    ^^^^^^^^
// add the useMemo hook
    () => users.map((user) => !user.inactive),
//  ^^
// define a callback so react can decide when to filter
    [users])
//  ^^^^^^
// define the "dependencies"
// = the filter function should be executed
//   when users array changes

  return (
    <Users users={activeUsers} />
  )
}
```

FUNCTIONAL PROGRAMMING IN REACT
HOW TO USE useMemo IN GLOBAL STATE MGMT

» Extracting/Transforming state from our state management

» new value won't be recalculated on every rerender

```
type UseReactStore<T> = <O>(transform: (state: T) => O) => [
//                          ^^^^^^^^^^^^^^^^^^^
// add possibility to add a transform function
  O, // Instead of T the transformed type O is returned
  (updateFn: (state: T) => T) => unknown
]

type CreateReactStore = <T>(stateFactory: () => T) => UseReactStore<T>
```

FUNCTIONAL PROGRAMMING
TASK

- Add transformer to your state mgmt solution
  - make sure the value is not recalculated on every rerender (use Reacts useMemo)
  - value should only be recalculated when state changes
- Throw/log error when state update is not immutable in generic store
  - previousState !== updatedState

FEEDBACK

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

» https://s.surveyplanet.com/x1ibwm85