

# STATE MANAGEMENT WITH REDUX

## (MMT-B2019)

# ROADMAP

- » Questions to homework
- » Functional Programming 101
- » Side Effects with React
- » State Management with Redux

# QUESTIONS TO HOMEWORK

# FUNCTIONAL PROGRAMMING 101

# FUNCTIONAL PROGRAMMING 101

- » Immutability
- » Pure functions
- » Side Effect

# FUNCTIONAL PROGRAMMING 101

## 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 101

## 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 - IMMUTABILITY

## IMMUTABLE OBJECTS IN JS

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

# FUNCTIONAL PROGRAMMING - IMMUTABILITY

## 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 - IMMUTABILITY

## UNFREEZE AN OBJECT

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

# FUNCTIONAL PROGRAMMING - IMMUTABILITY

## WHY IMMUTABILITY

- » race conditions impossible
- » state of the application is easier to reason about
- » easier to test

# FUNCTIONAL PROGRAMMING - SIDE EFFECTS

“A side effect is a change of system state or observable interaction with the outside world that occurs during the calculation of a result. (Chris Barbour)”

# FUNCTIONAL PROGRAMMING - SIDE EFFECTS

## SOME SIDE EFFECTS

- » DB/HTTP calls
- » changing the file system
- » querying the DOM
- » printing/logging
- » accessing system state (eg. Clock, Geolocation, ...)

# FUNCTIONAL PROGRAMMING - SIDE EFFECTS

## WHERE TO DEAL WITH SIDE EFFECTS

- » Moved to the boundaries of the system
- » Business logic stays pure functional



# 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 - PURE FUNCTIONS

## PURE OR IN-PURE

```
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 FUNCTIONS

## PURE OR IN-PURE

```
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 FUNCTIONS

## PURE OR IN-PURE

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

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

# FUNCTIONAL PROGRAMMING - PURE FUNCTIONS

## PURE OR IN-PURE

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

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

```
// both are not pure. const saves the pointer. config is still mutable
isAllowedToDrink(18) // true
config.minimumAge = 19
isAllowedToDrink(18) // false
```

# FUNCTIONAL PROGRAMMING - PURE FUNCTIONS

## PURE OR INPURE? 2/2

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

// freezing the config makes the function pure
isAllowedToDrink(18) // true
config.minimumAge = 19
isAllowedToDrink(18) // true
```

# FUNCTIONAL PROGRAMMING - SUMMARY

- » Immutability
  - » Object can't be changed after its creation
- » Side-Effects
  - » Communication with the outside world (eg. db, http, ...)
- » Pure-Functions
  - » returns the same output for the same input

# SIDE EFFECTS WITH REACT

# SIDE EFFECTS WITH REACT

## RECAP USEEFFECT

“The Effect Hook lets you perform side effects in function components”

# SIDE EFFECTS WITH REACT

## RECAP USEEFFECT

```
// Executed on every rerender
useEffect(() => {})

// Executed when component rendered initially
useEffect(() => {}, [])

// Executed when component rendered initially
// and when variable changes.
useEffect(() => {}, [variable])

// Cleanup when component unmounts (eg. eventHandlers, setInterval/setTimeout)
useEffect(() => {
  // do something fancy
  return () => { console.log('cleanup') }
}, [variable])
```

# SIDE EFFECTS WITH REACT

## RECAP USEEFFECT

```
export const MoneyTransactions = () => {
  const [moneyTransactions, setMoneyTransactions] = useState([])
  useEffect(() => {                                     // 1)
    fetch("http://localhost:3001/money-transaction") // 2)
      .then((response) => response.json())           // 3)
      .then((json) => setMoneyTransactions(json))     // 4)
  }, [])
}

// 1) define the useEffect hook
// 2) make the HTTP request to the backend
// 3) get the JSON from the response
// 4) set the response as state
// 5) call the effect when the component is mounted

// ... remaining component
}
```

# SIDE EFFECTS WITH REACT EXTRACT INTO CUSTOM HOOK

```
const useHTTPEffect = (endpoint) => {
  const [response, setResponse] = useState([])
  useEffect(() => {
    fetch(endpoint)
      .then((response) => response.json())
      .then((json) => setResponse(json))
  }, [])
  return response;
}

export const MoneyTransactions = () => {
  const moneyTransactions = useHttpEffect("http://localhost:3001/money-transaction")
  const users = useHttpEffect("http://localhost:3001/user")

  // ...
}
```

# **TASK (30 MIN)**

## **FETCH AND DISPLAY MONEY-TRANSACTIONS**

- » npm i json-server -D
- » add npm script
  - » "start:server": "json-server --watch db.json --port 3001",
- » add db.json
- » fetch/display money-transactions and users

# STATE MANAGEMENT WITH REDUX



# **REDUX - STATE**

## **WHAT IS APPLICATION STATE**

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

# **REDUX - STATE**

## **STATE IN REDUX TERMS**

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

# REDUX - STATE

## WHAT INFORMATION DO WE NEED TO RENDER THIS PAGE

App

Button

I owe somebody      Somebody owes me

User                  Amount

Select                  Create

---

A user	10,40\$	Paid
A user	10,40\$	
A user	10,40\$	
A user	-10,40\$	Paid
A user	10,40\$	

# REDUX - STATE

## WHAT INFORMATION DO WE NEED TO RENDER THIS PAGE

### QUESTION?

### STATE NAME

Is the user authenticated?	authenticationStatus
Is a form already filled with values?	formValues
Is the input field hovered/focused/ filled?	inputStatus
Am I owning money to somebody?	moneyTransactions
Is somebody owning me some money?	moneyTransactions
Which users can I owe some money?	users
Which components should be rendered?	url

# REDUX - STATE

## CATEGORISING DIFFERENT TYPES

- » Relevant for other parts of the application?
  - » add to global state
- » Irrelevant for other parts of the application?
  - » use component state (`useState` or `setState`)
  - » also known as UI State

# REDUX - STATE GLOBAL/LOCAL/URL

STATE NAME	STATE TYPE
authenticationStatus	
moneyTransactions	
users	
formValues	
inputStatus	
url	

# REDUX - STATE GLOBAL/LOCAL/URL

STATE NAME	STATE TYPE
authenticationStatus	global
moneyTransactions	global
users	global
formValues	local
inputStatus	local
url	url

# REDUX - STATE

## GLOBAL STATE

- » relevant for other components
- » could be seen as a client side database
  - » or a cached version of the server data
- » domain object should be stored here
  - » eg. users, money transactions, authentication token

# **REDUX - STATE**

## **UI STATE**

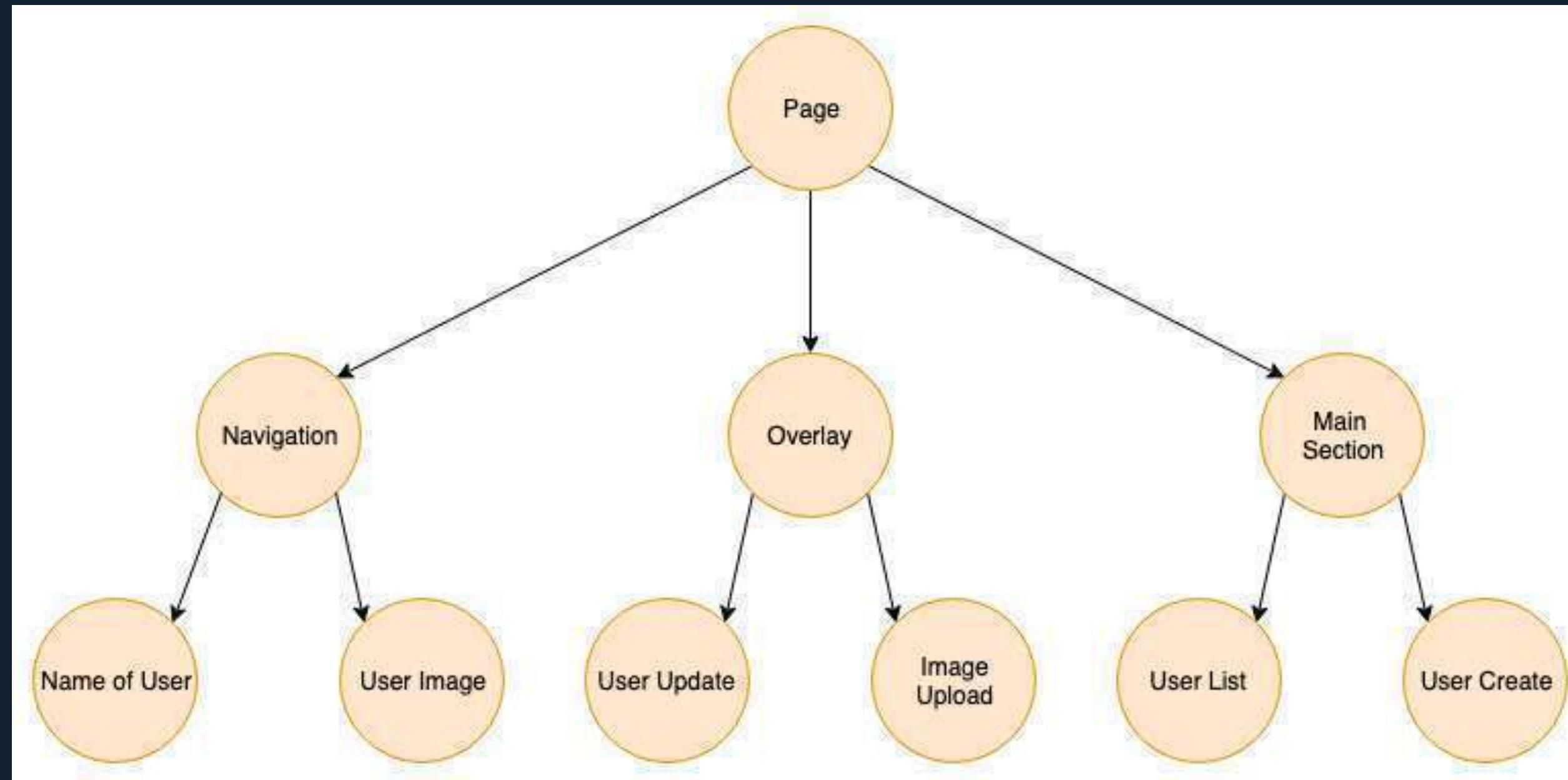
- » irrelevant for other parts of the application
- » or state which shouldn't be shared with others
- » What to store in UI state?
- » Form states
- » visual enhancements

# REDUX - STATE

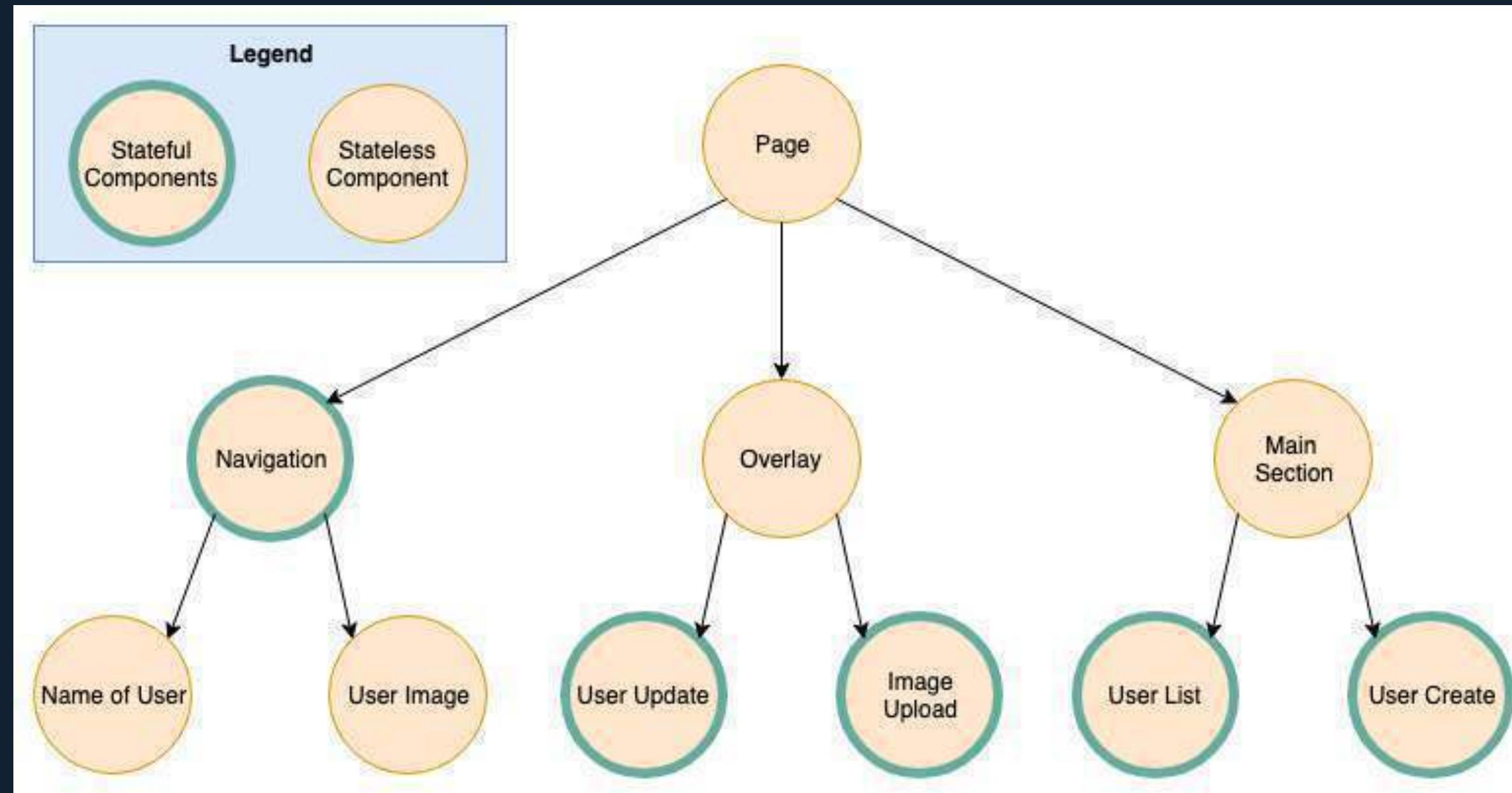
## URL STATE

- » defines which set of components should be rendered
- » persists on page reloads
- » What to store in URL state?
  - » the current route
  - » the current page of a paginated list

# REDUX - STATE REACT COMPONENT TREE



# REDUX - STATE STORING STATE IN COMPONENTS

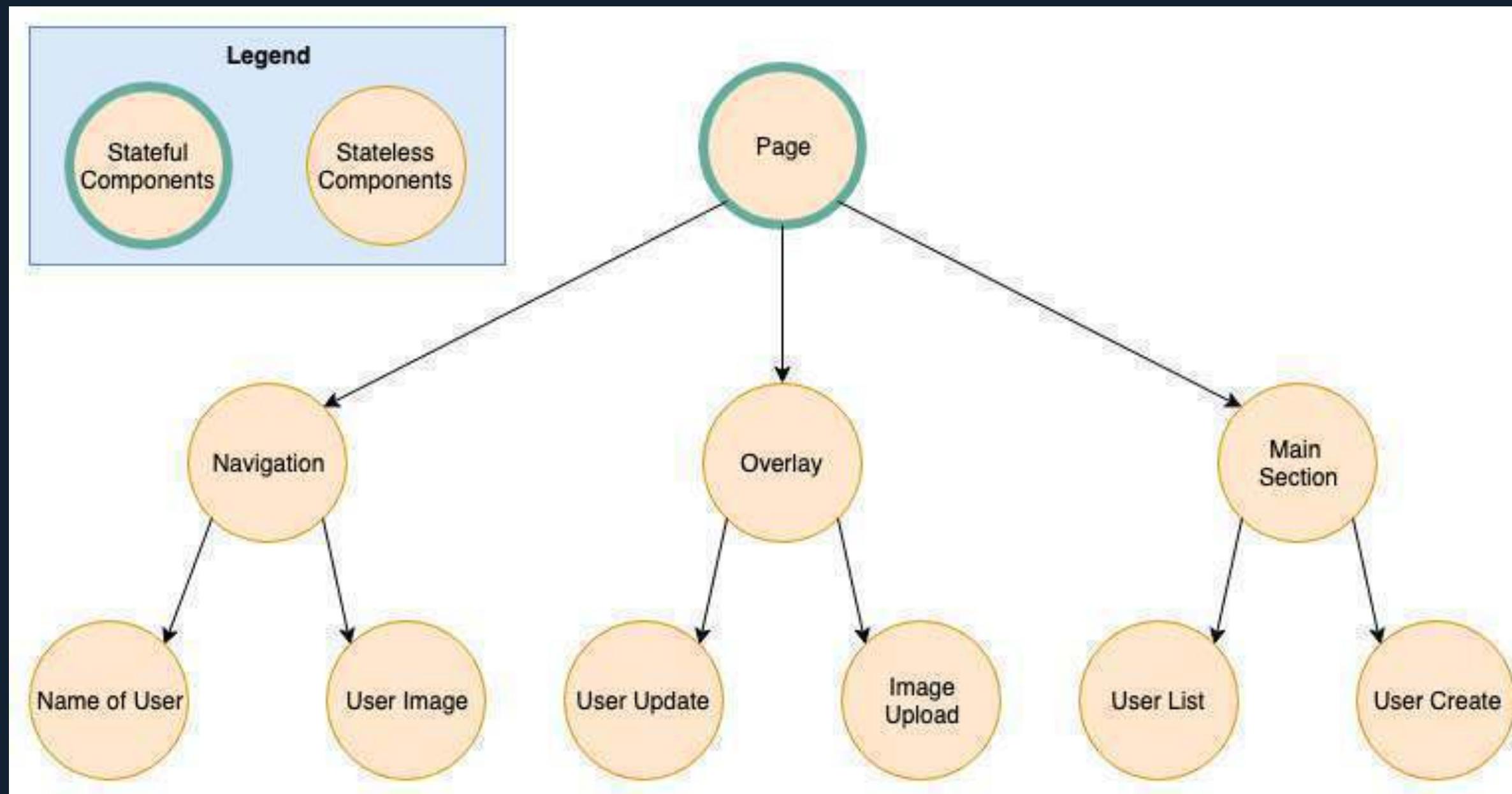


# REDUX - STATE

## 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

# REDUX - STATE STORING STATE IN THE ROOT COMPONENT



# **REDUX - STATE**

## **STORING STATE IN THE ROOT COMPONENT**

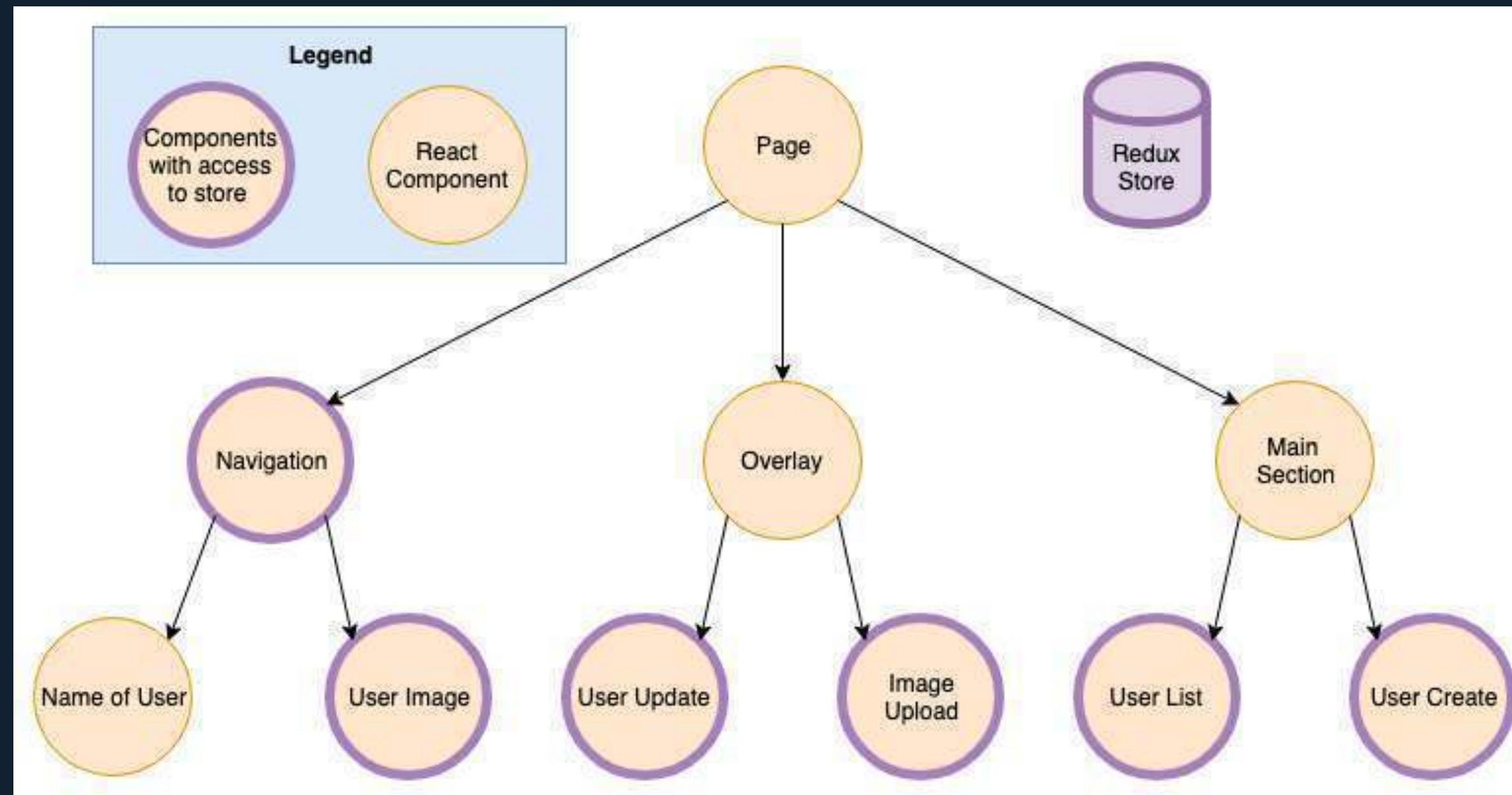
- » Pros
  - » User data could be fetched only once
  - » If UserUpdate component changes name of user
  - » navigation component is automatically updated
- » Cons
  - » State needs to be passed down to every component
  - » (Root component contains all state logic)

# REDUX - STATE STORING STATE IN THE ROOT COMPONENT

```
▼ <r>
  ▼ <View pointerEvents="box-none" style={281}>
    ▼ <div className="css-1dbjc4n r-13awgt0 r-12vffkv">
      ▼ <View key="1" pointerEvents="box-none" style={281}>
        ▼ <div className="css-1dbjc4n r-13awgt0 r-12vffkv">
          ▼ <t isNightMode={false}>
            ▼ <t>
              ▼ <r>
                ▼ <Context.Consumer>
                  ▼ <Context.Provider>
                    ▼ <Connect(t)>
                      ▼ <t language="de" loggedInUserId="253431163">
                        ▼ <t>
                          ▼ <Router.Consumer.Provider>
                            ▼ <withRouter(n)>
                              ▼ <t>
                                ▼ <Router.Consumer.Consumer>
                                  ▼ <Router.Consumer.Provider>
                                    ▼ <n>
                                      ▼ <t>
                                        ▼ <Router.Consumer.Consumer>
                                          ▼ <t>
                                            ▼ <Router.Consumer.Consumer>
                                              ▼ <Router.Consumer.Provider>
                                                ▼ <Unknown>
                                                  ▼ <t>
                                                    ▼ <withRouter(t)>
                                                      ▼ <t>
                                                        ▼ <Router.Consumer.Consumer>
                                                          ▼ <Router.Consumer.Provider>
                                                            ▼ <t>
                                                              ▼ <Connect(t)>
                                                                ▼ <t scale="normal">
                                                                  ▼ <t>
                                                                    ▼ <t showReload={true}>
                                                                      ▶ <SideEffect(t) title="Twitter">...</SideEffect(t)>
                                                                      ▶ <withRouter(Connect(t))>...</withRouter(Connect(t))>
                                                                      ▶ <t zIndex={1}>...</t>
                                                                ▼ <View>
                                                                  ▼ <div className="css-1dbjc4n r-1pi2tsx r-sa2ff0 r-13qz1uu r-417010">
                                                                    ▶ <withRouter(Connect(i))>...</withRouter(Connect(i))>
                                                                  ▼ <@twitter/Responsive>
                                                                    ▼ <View accessibilityRole="main" style={245}>
                                                                      ▼ <main role="main" className="css-1dbjc4n r-16y2uox r-1wbh5a2">
                                                                        ▼ <View style={248}>
```

# REDUX - STATE

## STORING STATE IN REDUX

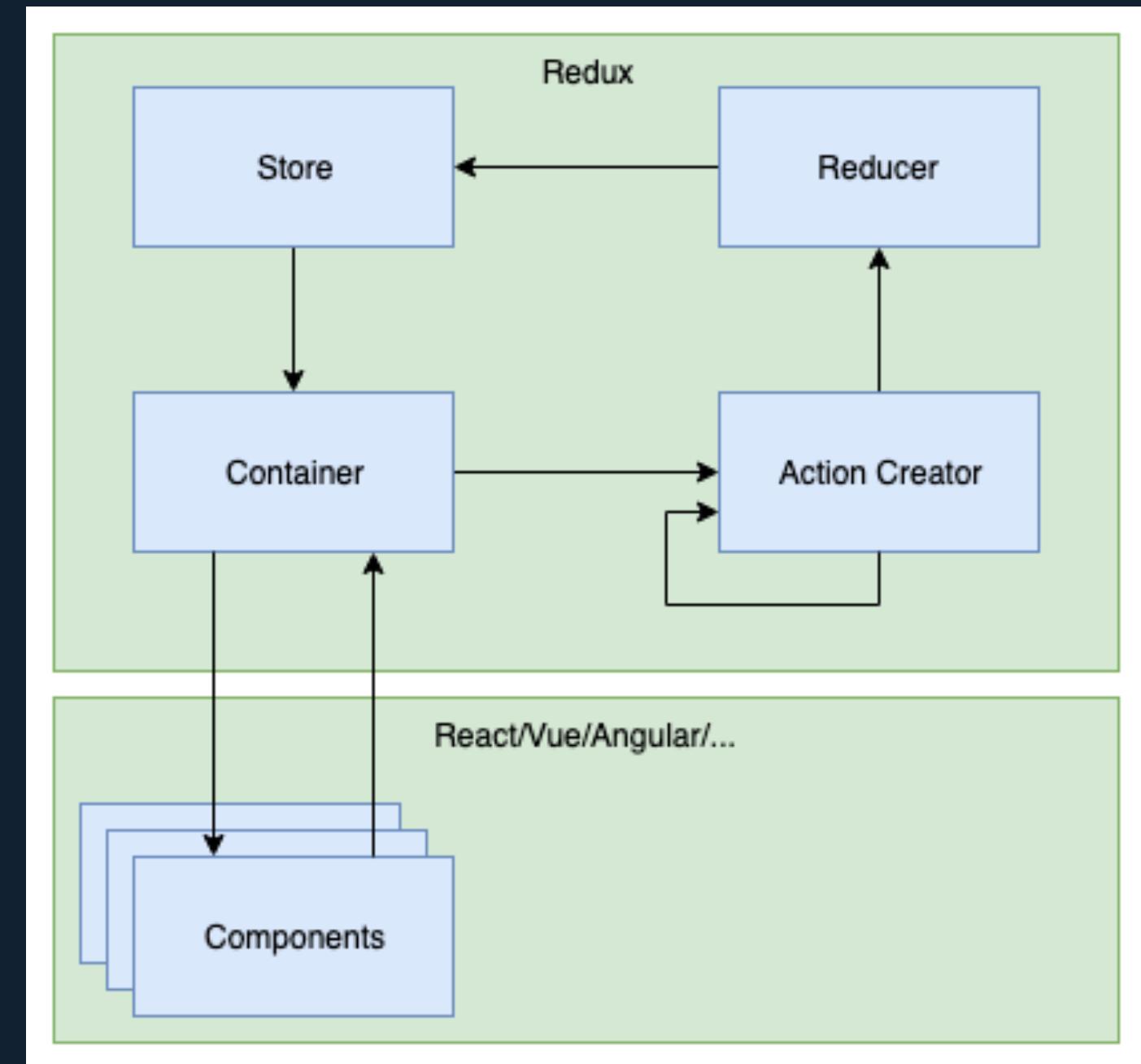


# **REDUX - STATE**

## **STORING STATE IN REDUX**

- » 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

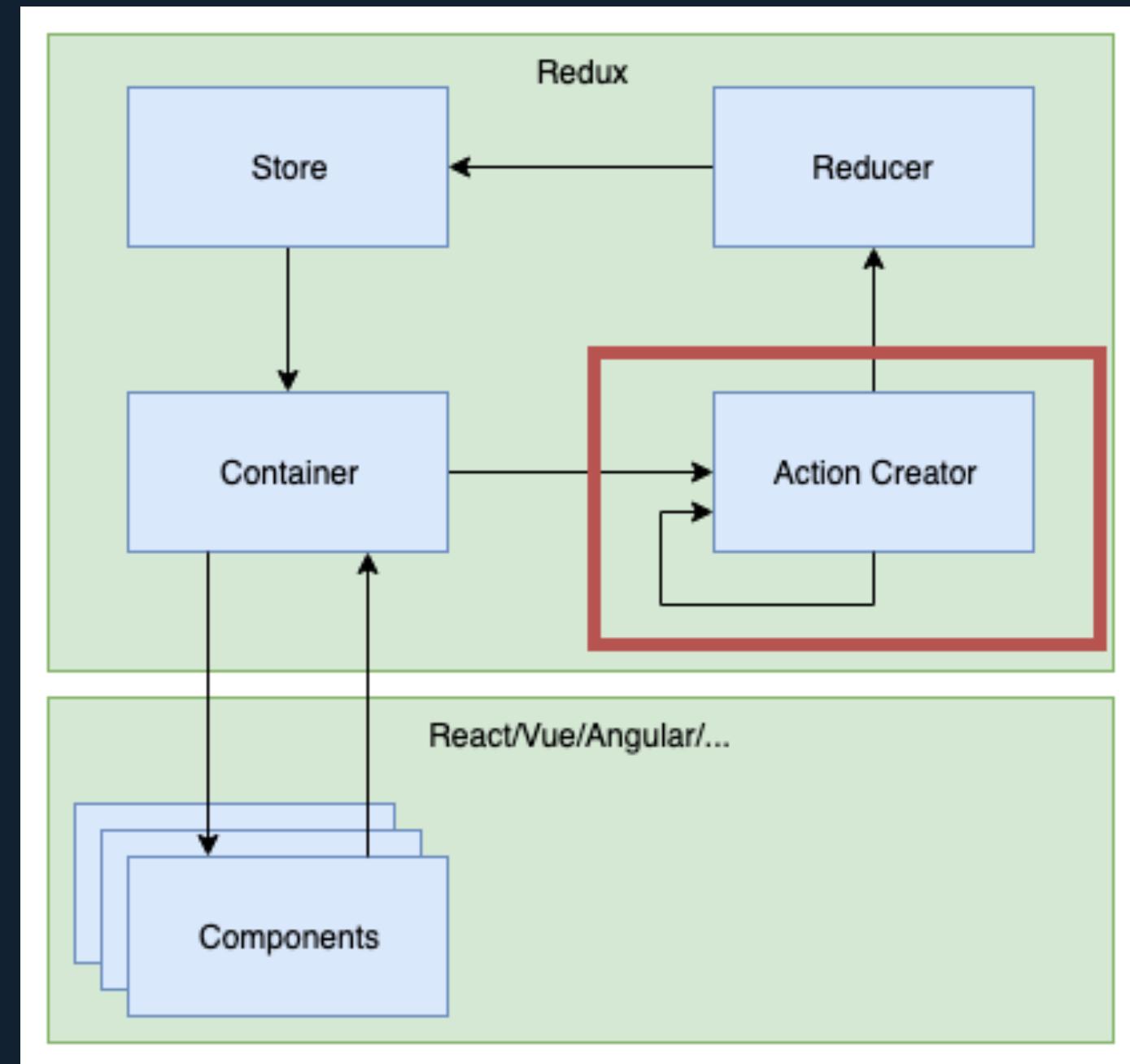
# REDUX



# WHY REDUX

- » Managing state in react can be challenging
  - » How to synchronize state between distant UI parts
- » Redux provides a predictable way to manage state
- » State can only be changed by dispatching an action
- » Each action might change the previous state to a new updated state
- » Works with react, vue, angular, ...

# REDUX - ACTIONS



# REDUX - ACTIONS

“Something happened in the app which might be interesting.”

# REDUX - ACTIONS

- » An action is data from the application which might be relevant for the store
- » Information is sent to the store via `store.dispatch`

```
const signInAction = {
  type: 'signIn',
  payload: {
    username: 'peter',
    password: 'the clam'
  }
}

store.dispatch(signInAction)
```

# REDUX - ACTION CREATORS

- » A functions which creates actions
- » With redux-thunk action creators can dispatch itself
- » This is where side effects are handled

```
const actionCreator = () => (dispatch) => {  
  dispatch({ type: 'action1', payload: {} })  
  dispatch({ type: 'action2', payload: { something: 'random' } })  
  dispatch({ type: 'action3', payload: { something: 'random' } })  
  // ...  
}  
  
store.dispatch(actionCreator())
```

# REDUX - ACTION CREATORS

## ASYNC ACTION CREATORS

```
const actionCreatorWithData = ({ username, password }) => (dispatch) => {
  dispatch({ type: 'action1', payload: { username, password } })
  dispatch({ type: 'action2/success', payload: { something: 'random' } })
}

store.dispatch(signInAction({ username: 'Mike', password: '1234' }))
```

# REDUX - ACTION CREATORS

## ASYNC ACTION CREATORS

```
const createMoneyTransaction = ({ creditorId, debtorId, amount }) =>
  async (dispatch) => {
    dispatch({ type: 'createMoneyTransaction/initiated', payload: {} })
    try {
      const moneyTransaction = await fetch('/money-transaction/' , {
        creditorId,
        debtorId,
        amount
      })
      dispatch({
        type: 'createMoneyTransaction/success',
        payload: moneyTransaction
      })
    } catch (e) {
      dispatch({ type: 'createMoneyTransaction/error', payload: e })
    }
  }

store.dispatch(createMoneyTransaction({ creditorId: 1, debtorId: 2, amount: 10.3 }))
```

# TASK 1 - REDUX SETUP

- » Download
  - » React dev tools
  - » Redux dev tools
- » in src/store.js
  - » add `window.store = store;`

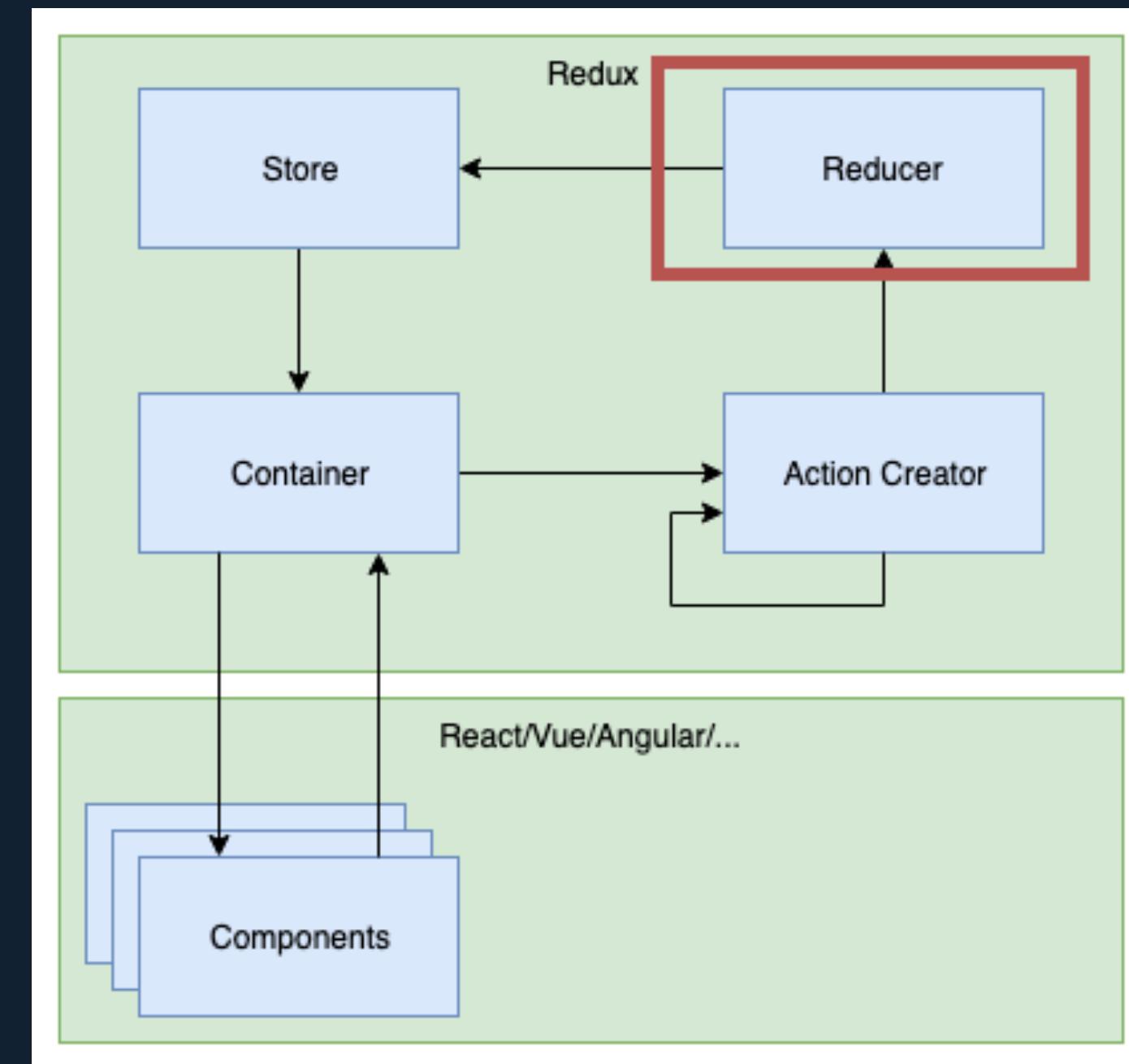
# TASK 2 - ACTIONS

```
» npm run start:app  
» go to localhost:3000  
» open dev tools/redux tab  
» store.dispatch({ type: 'signIn', payload: { name: 'name', password: 'password' } })
```

# TASK 3 - ASYNC ACTION CREATOR

- » Create an action creator which calls /user and /money-transaction
- » dispatch result of http call

# REDUCERS



# REDUCERS

“Reducers specify how the application's state changes in response to actions sent to the store. (Source)”

# REDUCERS

- » Specify how state changes in response to actions.
- » Pure function
  - » input appState and action
  - » output next application state

```
const initialState = {}  
const reducer = (previousState = initialState, action) => {  
  // do something with the state  
  return nextState  
}
```

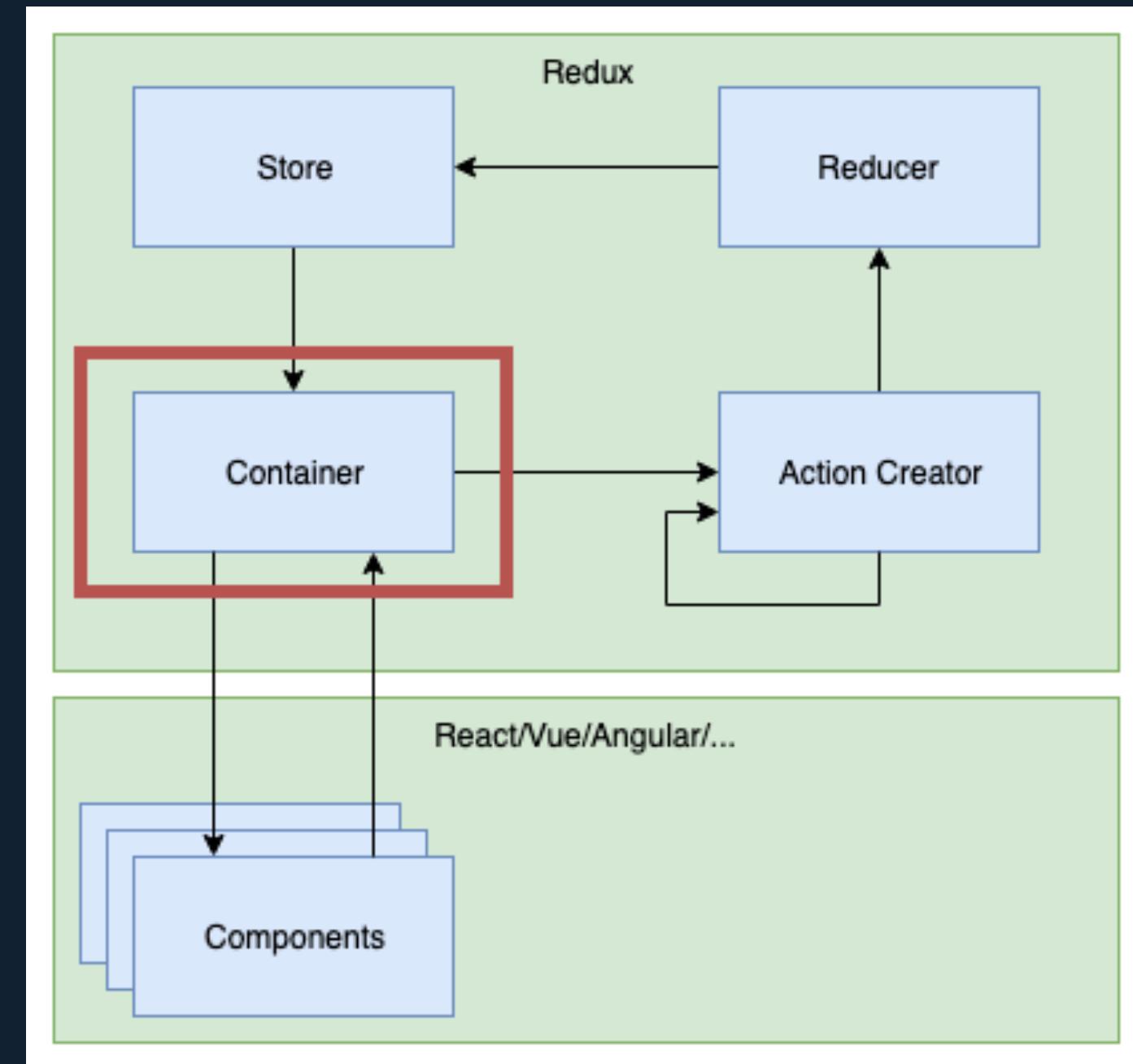
# REDUCERS

```
const initialState = []
const moneyTransactionReducer = (previousState = initialState, action) => {
  switch (action.type) {
    case 'createMoneyTransaction/success':
      return [...previousState, action.payload]
    case 'reset':
      return initialState
    default:
      return previousState
  }
}
```

# TASK 1 (REDUCER)

- » Add a `userReducer` reducer
- » entry point: `src/reducer/index.js`
- » listen to '`fetchUser/success`'
- » try to populate the redux store with a new user

# CONTAINER COMPONENTS



# CONTAINER COMPONENTS

- » Glue between react and redux
- » Provides data from the global store to the components
- » Provides "callbacks" to trigger actions

# CONTAINER COMPONENTS

```
import { createMoneyTransaction } from './action-creators/money-transactions'
const mapStateToProps = (state, props) => {
  return {
    moneyTransactions: state.moneyTransactions
  }
}

const mapDispatchToProps = (dispatch, props) => {
  return {
    createMoneyTransaction: (payload) =>
      dispatch(createMoneyTransaction(payload))
  }
}

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(MoneyTransactionList)
```

# MAPSTATETOPROPS

- » extract data from the store and provides it to a component
- » data filtering can be done here
- » function `mapStateToProps(state, ownProps?)`
  - » state -> the entire application state
  - » ownProps -> properties which are passed from other components
- » Docs

# MAPDISPATCHTOPROPS

- » binds actions with the store and provides those actions to a component
- » function mapStateToProps(dispatch, ownProps?)
  - » dispatch -> the stores dispatch function
  - » ownProps -> properties which are passed from other components
- » Docs

# TASK 1 (CONTAINER)

- » Try to connect your moneyTransactionCreate dropdown with users from the store

# TASK 2 (CONNECT TO BACKEND)

» Try to connect fetchUsers action creator

# FURTHER LINKS

- » Redux Tutorial
- » Mostly adequate guide to FP
- » Hands-On Functional Programming with TypeScript
- » Immutable Data Structures

# FEEDBACK

- » Questions: [tmayrhofer.lba@fh-salzburg.ac.at](mailto:tmayrhofer.lba@fh-salzburg.ac.at)
- » <https://de.surveymonkey.com/r/8TW92LL>