React Training

Understand 100%

by

Many Small Examples

&

Many Small Tasks

jan.schulz@cileria.com

About this course

- Not just a video tutorial - you need to solve many tasks
- Therefore, it is rather a **training program**
- It requires you have knowledge in
  - Basic JavaScript: Variables, Functions, Loops, Arrays, Objects, Conditional Statements
  - Basic HTML/CSS: Selectors, Div, Block VS Inline Elements, Box-Model (margin, padding, border), Flexbox, Tables
- I promise you: **Right after you finished this very intensive training program, you will be ready to be productive in React.**

Agenda

1. Advanced JavaScript
   - EcmaScript6-Requirements
2. React
   - Class- and Functional Components
3. React Hooks
   - Functional Components that use Hooks

Agenda – Part 1
Advanced JavaScript

1. Const, Let and Var
2. Function Constructors & Classes
3. Arrow Functions
4. Function References
5. Promises / Async / Await
6. Destructuring
7. Spread Operator
8. Key Interpolation

1.1. const, let and var

- Var
  - Defines a variable that can be re-assigned and which is visible outside of an anonymous block
- Let
  - Defines a variable that can be re-assigned and which is not visible outside of an anonymous block
- Const
  - Defines a variable that cannot be re-assigned and which is not visible outside of an anonymous block

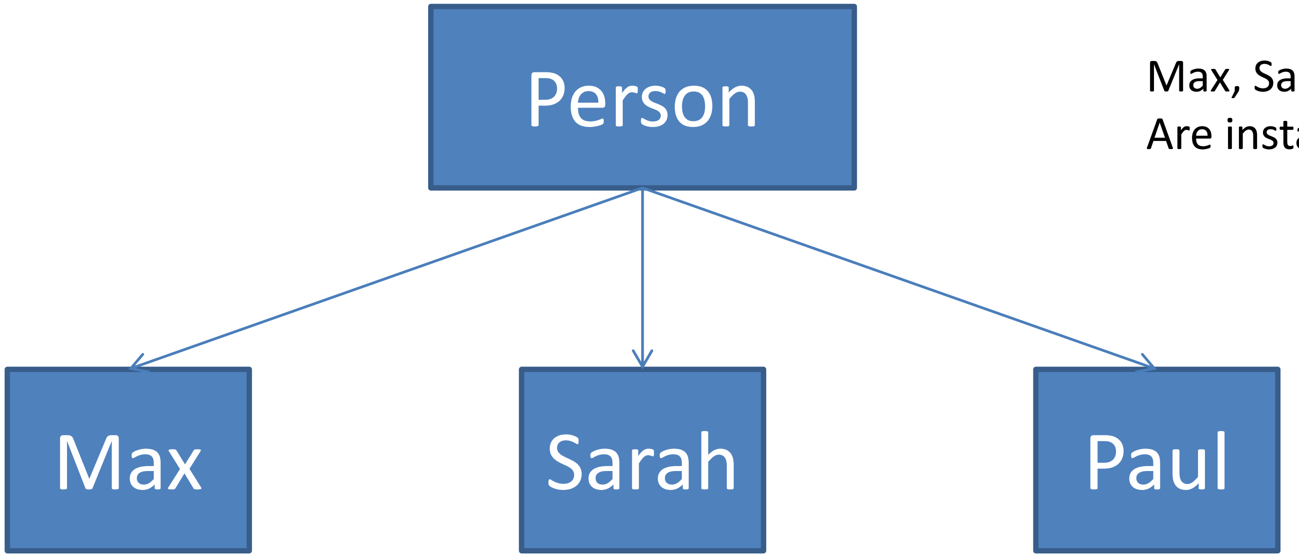
1.1. Task

- Please take a look

  01_es6-refresher/01_varletconst/main.js

1.2. Function Constructors and Classes

- Function Constructors create objects
  - Like a blueprint for objects



Max, Sarah and Paul
Are instances of Person

1.2. Function Constructors and Classes

- Function constructors are **blueprints for objects**

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
}

Let max = new Person(‘Max‘, 35, ‘coder‘);
Let sarah = new Person(‘Sarah‘, 30, ‘designer‘);
Let paul = new Person(‘Max‘, 45, ‘taxi-driver‘);
```

1.2. Function Constructors , Classes and Inheritance

```
let john = {
                Name: ‘John‘,
                yearOfBirth: 1990,
                isMarried: false
}
```

```
let jane = {
                Name: ‘Jane‘,
                yearOfBirth: 1991,
                isMarried: true
}
```

```
let mark = {
                Name: ‘Mark‘,
                yearOfBirth: 1948,
                isMarried: true
}
```

1.2. Function Constructors , Classes and Inheritance

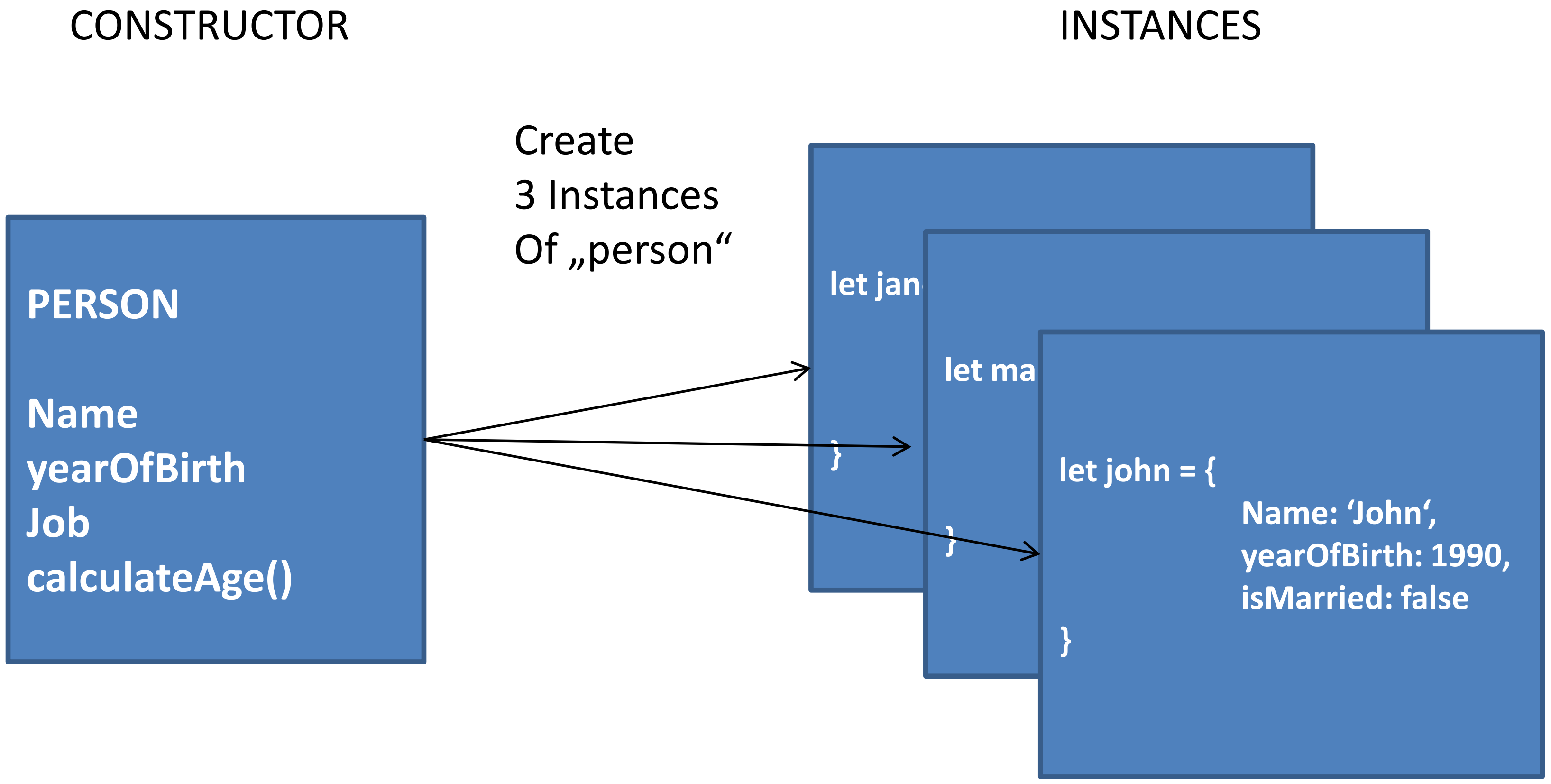
```
let john = {
                Name: ‘John‘,
                yearOfBirth: 1990,
                isMarried: false
}
```

```
let jane = {
                Name: ‘Jane‘,
                yearOfBirth: 1991,
                isMarried: true
}
```

```
let mark = {
                Name: ‘Mark‘,
                yearOfBirth: 1948,
                isMarried: true
}
```

3 Objects = A lot of typing

1.2. Function Constructors , Classes and Inheritance



1.2. Function Constructors , Classes and Inheritance

PERSON

Name
yearOfBirth
Job
calculateAge()

1.2. Function Constructors , Classes and Inheritance

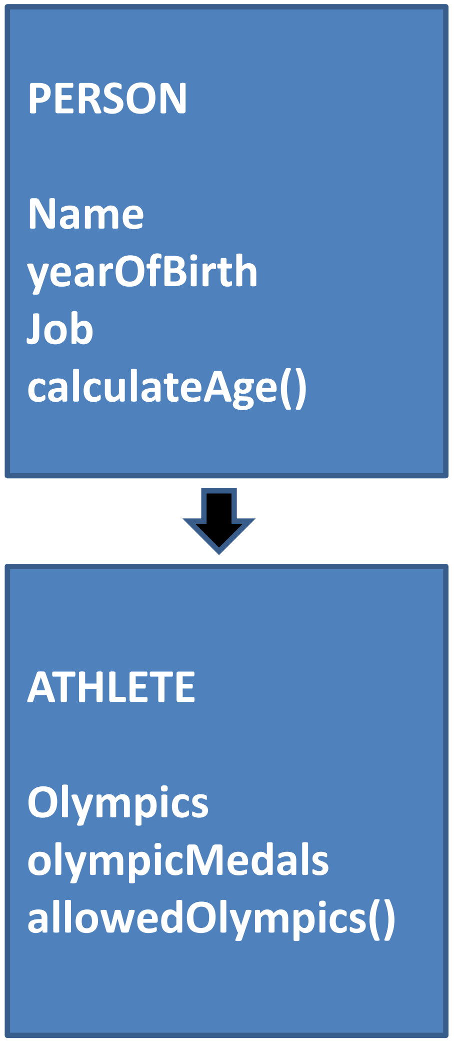
PERSON

Name
yearOfBirth
Job
calculateAge()

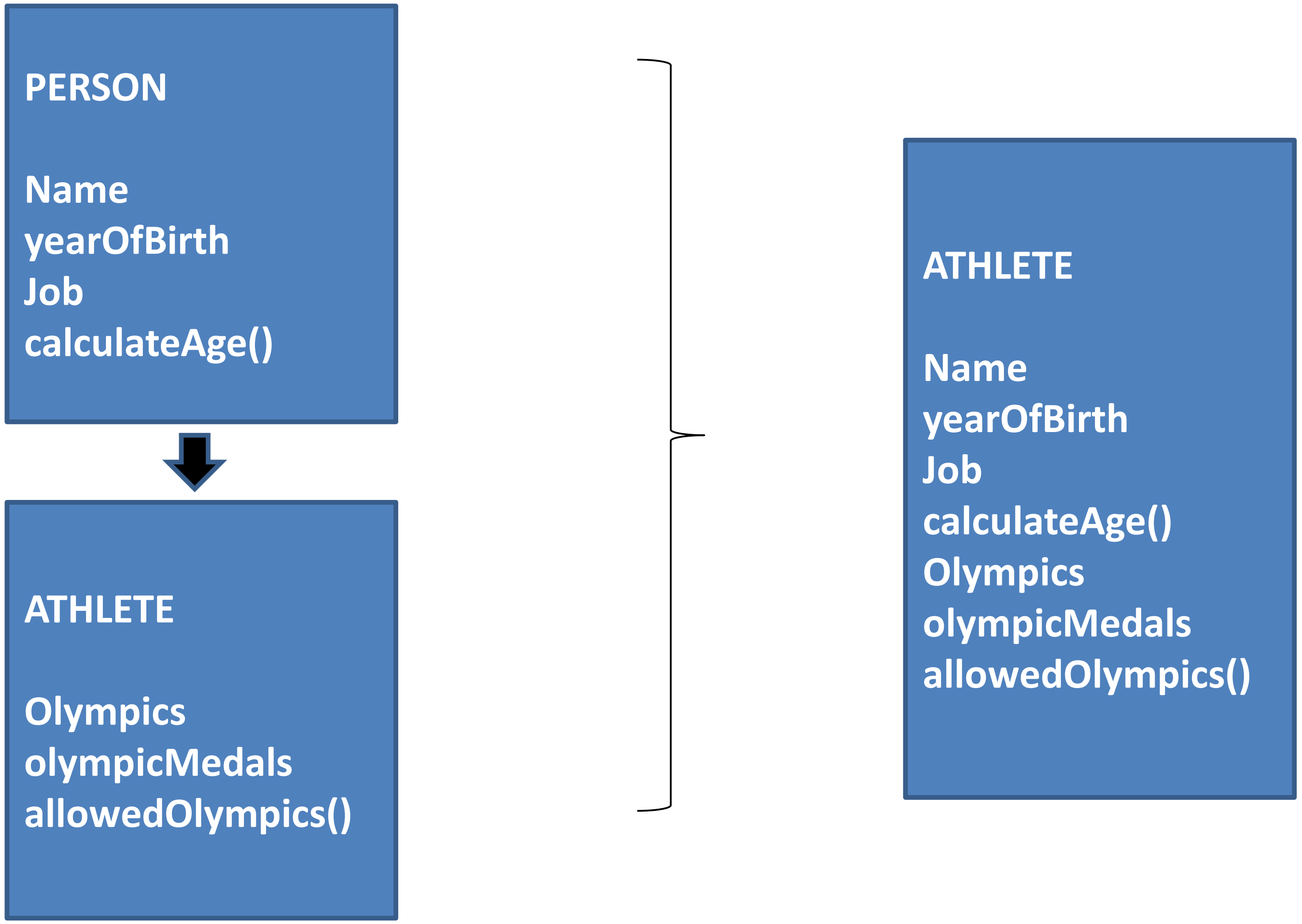
ATHLETE

Olympics
olympicMedals
allowedOlympics()

1.2. Function Constructors , Classes and Inheritance



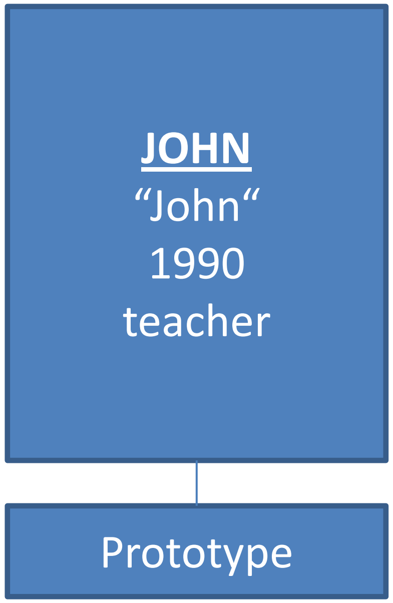
1.2. Function Constructors , Classes and Inheritance



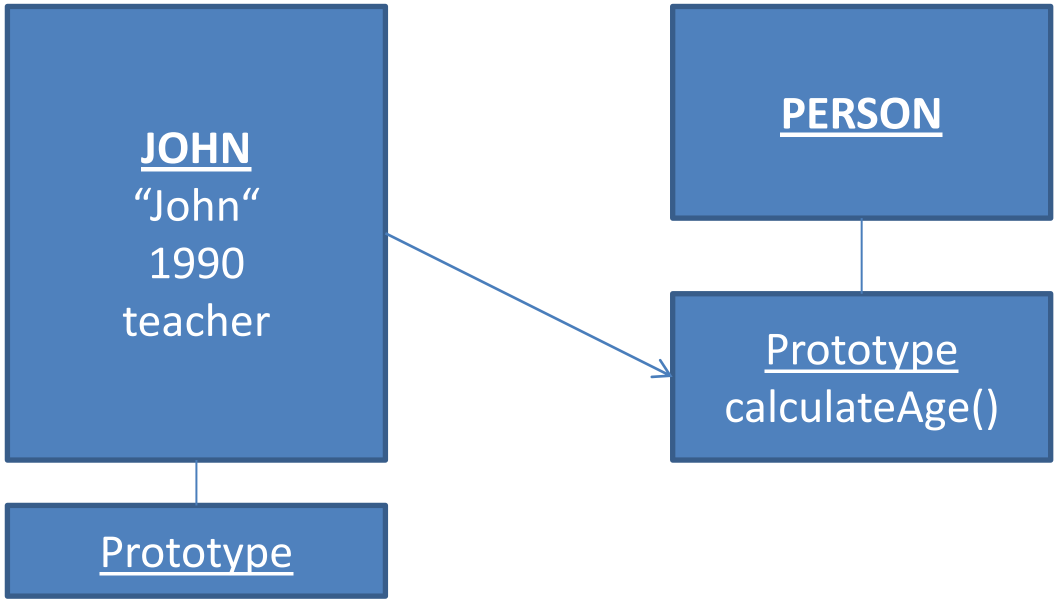
1.2. Function Constructors , Classes and Inheritance

- Prototypes:
  – Every object in JavaScript has an attribute called **prototype**
  – Each prototype has an attribute, which itself a prototype
  – This goes on, **until prototype is null**

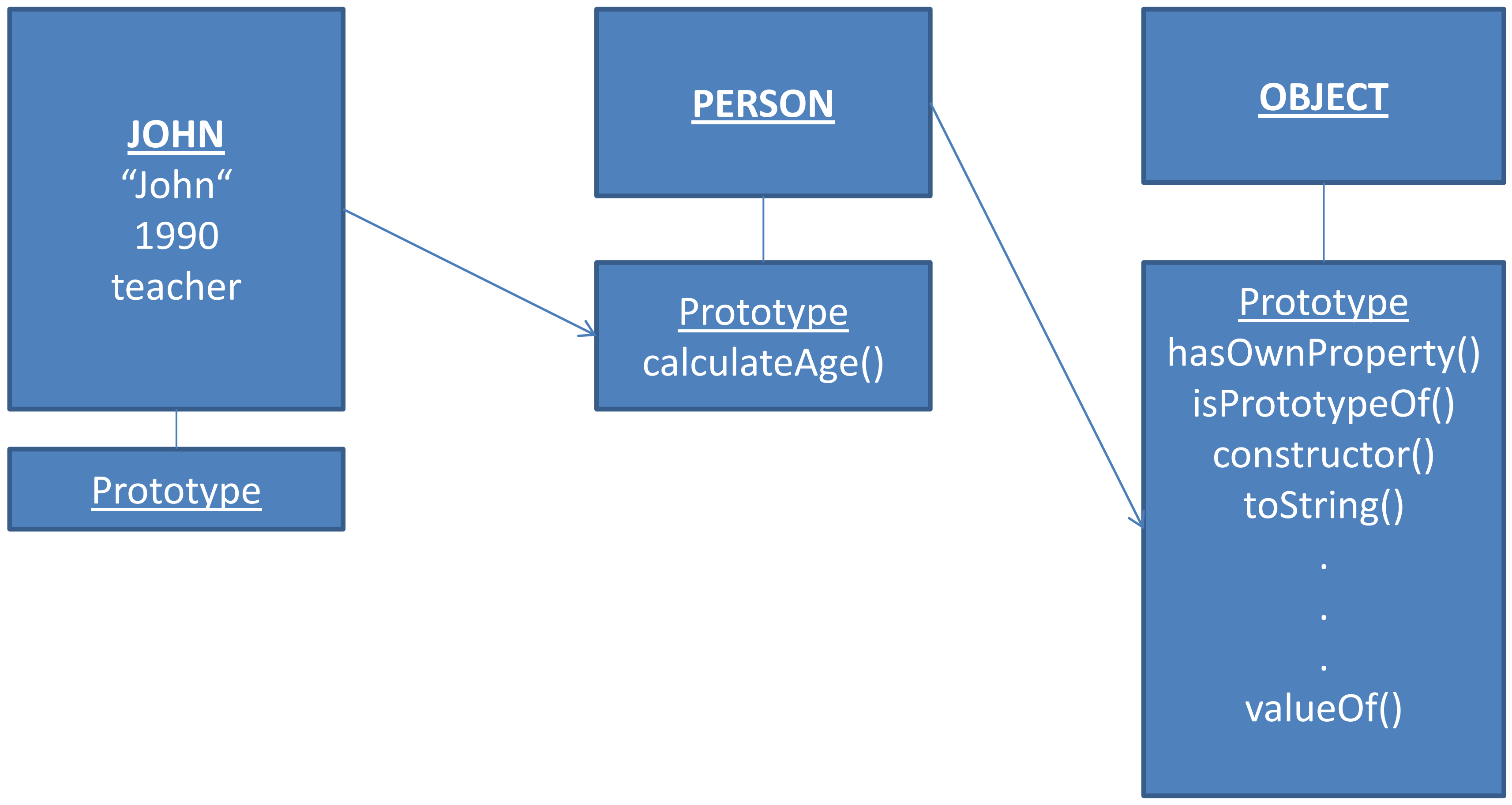
1.2. Function Constructors , Classes and Inheritance



1.2. Function Constructors , Classes and Inheritance

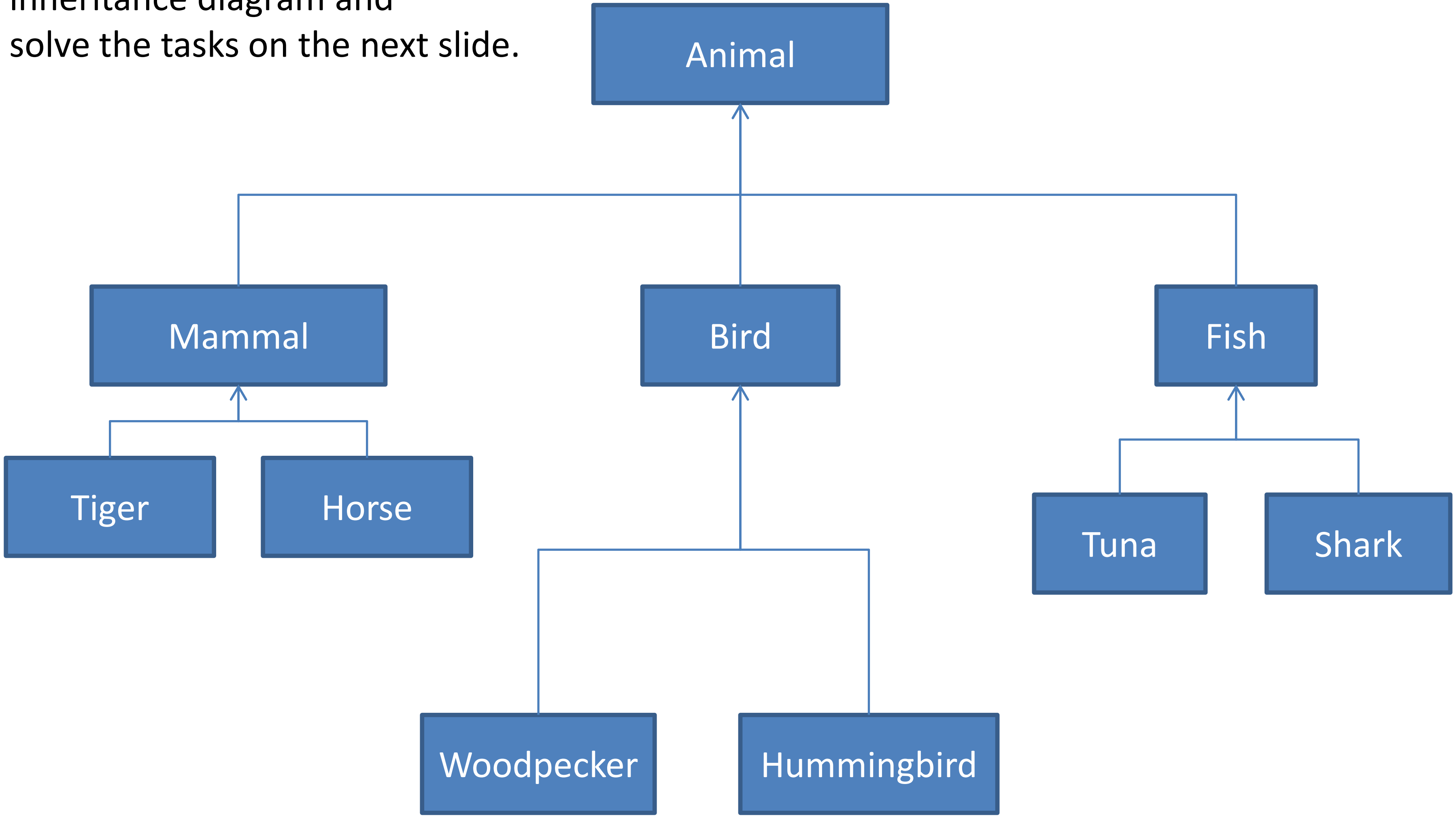


1.2. Function Constructors , Classes and Inheritance



1.2. Task

Please analyze the following inheritance diagram and solve the tasks on the next slide.



1.2. Function Constructors , Classes and Inheritance

- The class keyword is **syntactic sugar** for defining prototypes

```
class Person {
    constructor(name, age, job) {
            this.name = name;
            this.age = age;
            this.job = job;
    }

    calculateAge() {
            return 2018 – this.age;
    }
}
```

1.2. Task

1. With your knowledge about Inheritance, please create the classes (or optionally function constructors) according to the Animal diagram and consider the following rules:
    1. Each animal has a **name** that is set when it is constructed.
    2. All animals can **sleep**, **eat** and **die** (use functions for this, e.g. **sleep()**)
    3. Mammals and birds can **breathe**.
    4. Fishes can **swim**.
    5. Birds can **fly**.
    6. Tigers and Sharks can kill, whereas **kill()** expects one parameter **otherAnimal.** Kill() calls the die() function o**f otherAnimal**.
2. Create one tiger with name „Vitaly“, one Shark with name „Nemo“, one horse with name „Fury“.
3. Nemo is hungry and kills Fury and Vitaly. Then Nemo eats.
4. Nemo dies.

1.3. Arrow Functions

- An Arrow-Function is a function that ...
  1. Is syntactically defined by an arrow „=>“
  2. Is syntactically defined by an expression instead of a declaration
  3. Does not have a sticky this-Keyword

```
const sayHalloWorld = () => {
	console.log(`Hallo World`);
}
```

1.3. Arrow Functions

- What is the difference between an expression and a statement?

1.3. Task

- Please take a look

  01_es6-refresher/03_arrowfunctions/main.js

1.4. Function References

- A function reference is a variable, that leads to the body of a function with calling the actual function.

```
const sayHalloWorld = () => {
    console.log(`Hallo World`);
}

sayHalloWorld(); // a function call
sayHalloWorld; // a function reference
```

1.4. Task

- Please take a look

  01_es6-refresher/04_functionreferences/main.js

1.5. Promises and Async/Await

- There are three ways to deal with asynchronous functions in JavaScript
  1. Callbacks
  2. Promises
  3. Async/Await
     - Is is actually just Promises in a nicer syntax

1.5. Task

- Please take a look

  01_es6-refresher/05_promisesasyncawait/02_task/main.js

1.6. Destructuring

- Destructuring is unpacking
  - Values from arrays
  - Properties from objects

... into distinct variables

1.6. Task

- Please take a look

  01_es6-refresher/06_destructuring/main.js

1.7. Spread Operator

- Allows an iterable such as an array or object to be expanded in places where zero or more
  - arguments -> function calls
  - elements -> for array literals

... are expected.

1.7. Task

- Please take a look

  01_es6-refresher/07_spread/main.js

1.8. Rest Operator

- The rest operator allows to represent arguments as an array

1.8. Task

- Please take a look

  01_es6-refresher/08_rest/main.js

1.9. Key Interpolation

- Key Interpolation is addressing a key-value pair of an object by a string

```
const obj = { x: 1, y: 2 };
console.log(obj.x); // no key-interpolation
console.log(obj[‘x‘]); // key interpolation
```

1.9. Task

- Please take a look

  01_es6-refresher/09_keyinterpolation/main.js

Agenda – Part 2.1. Introduction to React

1. Installation
   - Windows
   - MacOS
   - Ubuntu Linux
2. HalloWorld
3. JSX
4. CSS
5. Sub-Components

2.1.1. Installation

- Windows

2.1.1. Installation

- MacOS

2.1.1. Installation

- Ubuntu Linux

2.1.2. HalloWorld

- The classic HalloWorld

2.1.2. Task

- To the code of 2.2. HalloWorld, please Add a personal greeting – e.g. in an h2-Tag

  If the time is between 05:00 – 11:59 -> „Good Morning!“
  If the time is between 12:00 – 16:59 -> „Good Day!“
  If the time is between 17:00 – 23:59 -> „Good Evening!“
  If the time is between 00:00 – 04:59 -> „Good Night!“

2.1.3. JSX

- XML = Extensible Markup Language
  - <beginningTag>Hallo World</closingTag>
  - <div>Hallo World</div>
- JSX = JavaScript + XML
  - The interpreter understands

2.1.3. Task 1

Please add the solution to the example of 2.1.3.

The following date is given:

```
const users = [
    { money: 150, name: 'paul', country: 'germany', born: 1995 },
    { money: -50, name: 'sarah', country: 'uk', born: 1990 },
    { money: 20, name: 'bob', country: 'spain', born: 1988 },
    { money: 550, name: 'hans', country: 'germany', born: 1982 },
    { money: 5, name: 'julia', country: 'germany', born: 1972 },
    { money: 1040, name: 'carl', country: 'denmark', born: 1999 },
    { money: -500, name: 'peter', country: 'germany', born: 1991 },
    { money: 0, name: 'julia', country: 'ireland', born: 1980 }
];
```

1. Display the users in a table
2. Only show the German users.

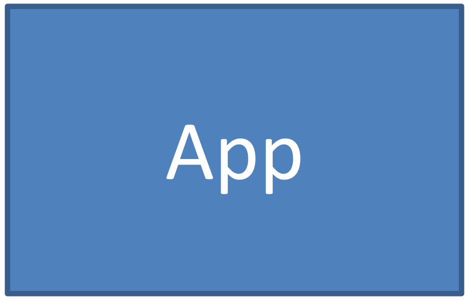
2.1.4. CSS

- In React, CSS can be integrated using
  - Inline CSS
  - CSS Files

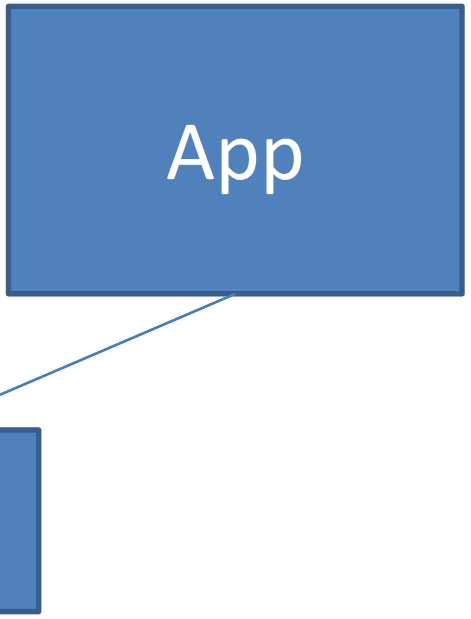
2.1.5. Sub Components

- A sub component is a component, that is instantiated inside another component which is the sub component‘s parent component

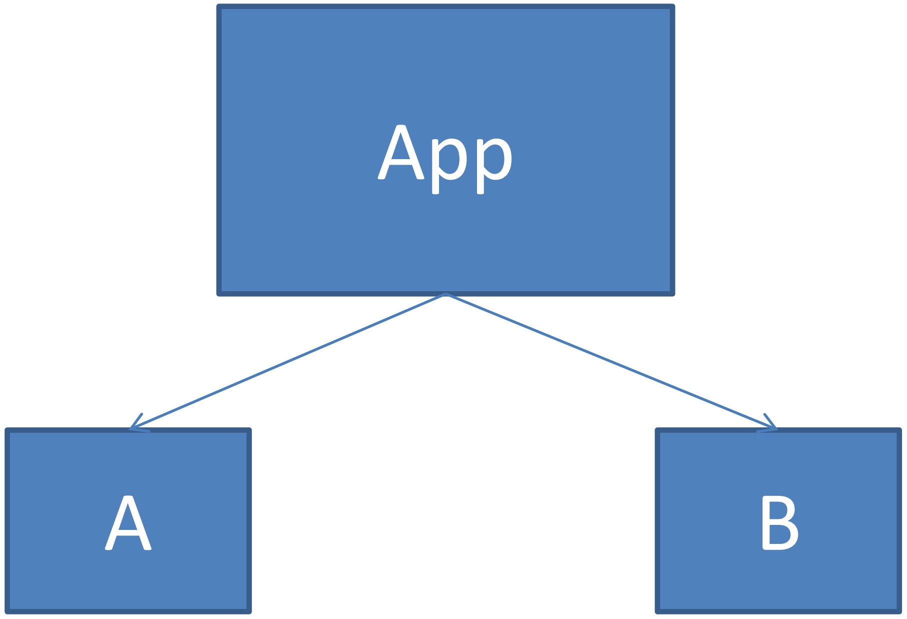
2.1.5. Sub-Components



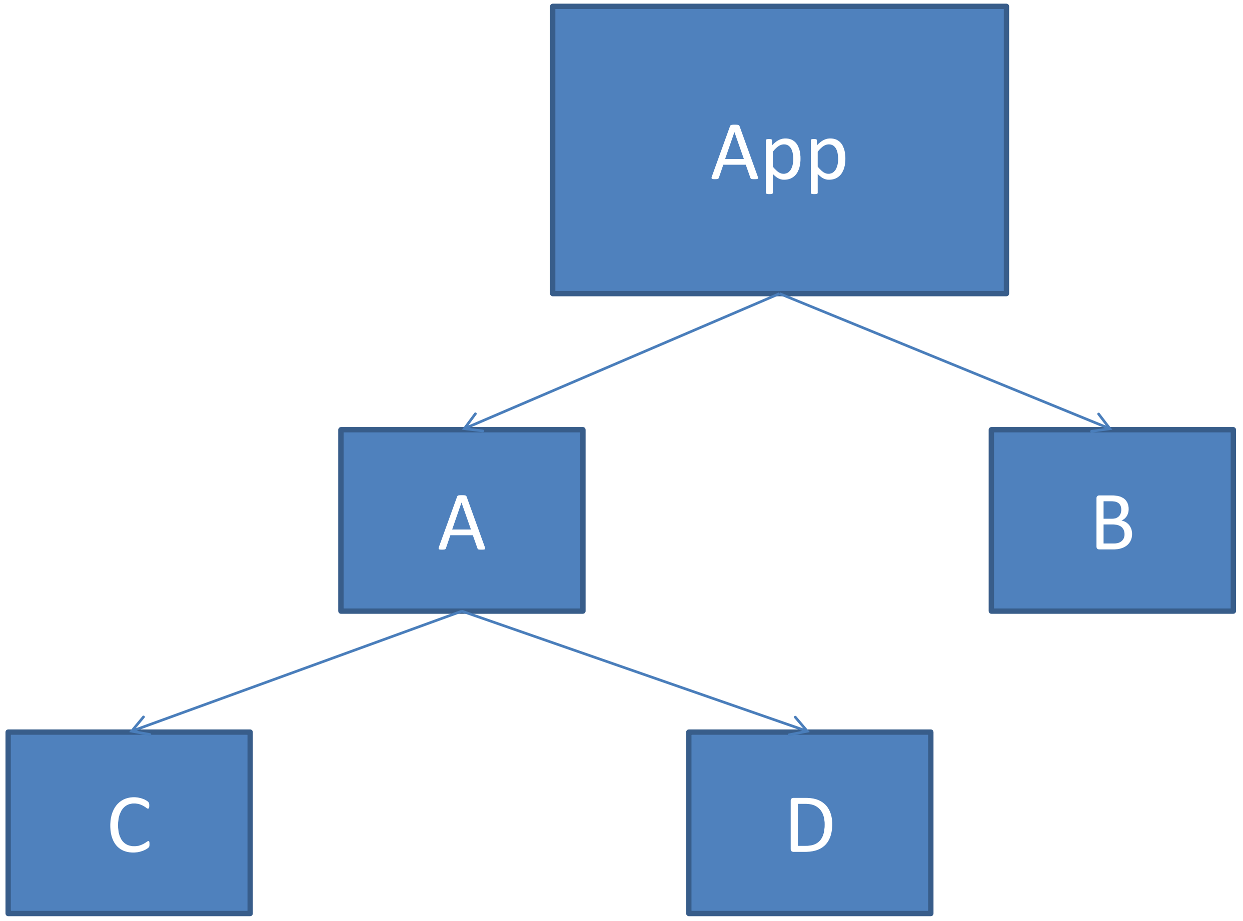
2.1.5. Sub-Components



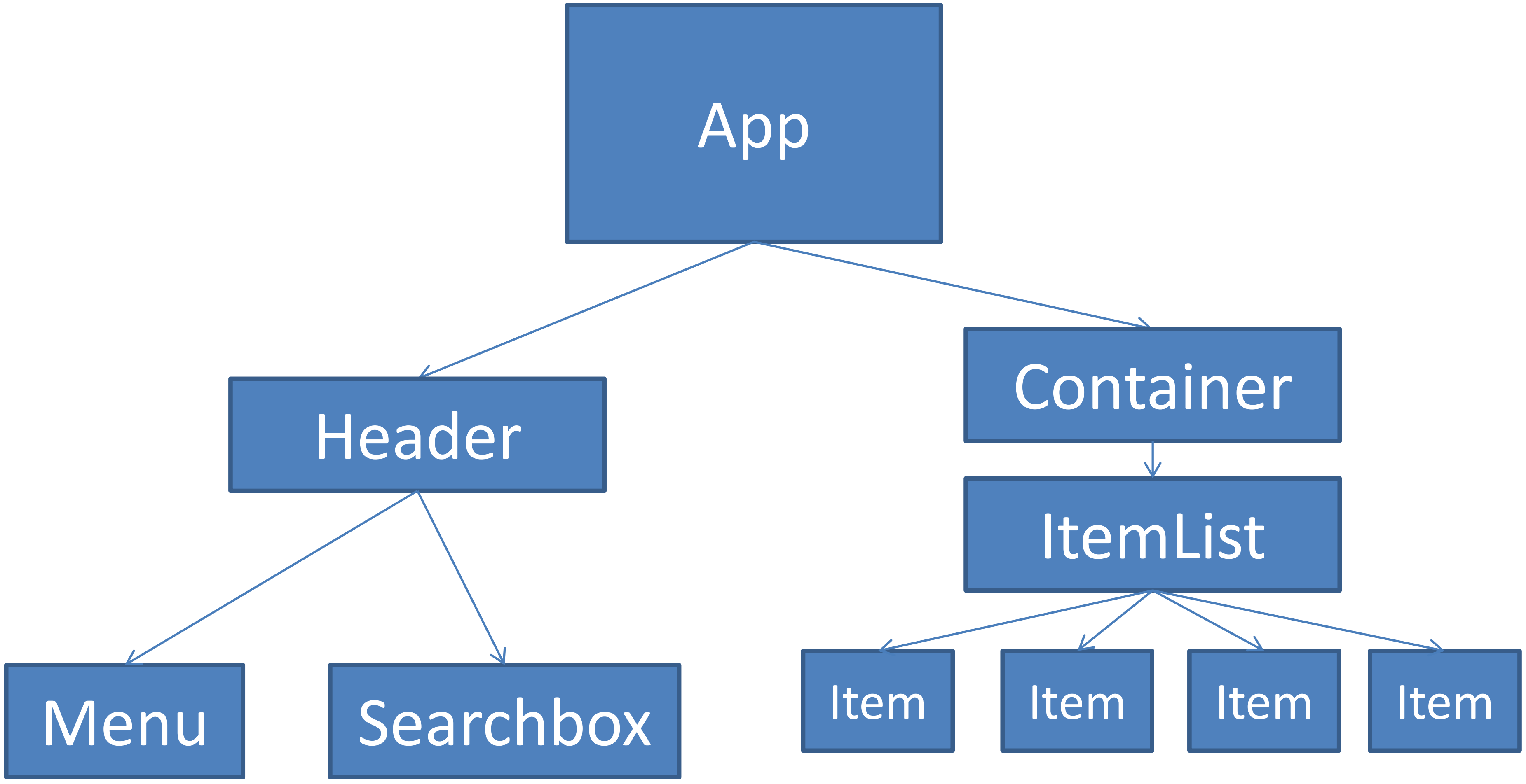
2.1.5. Sub-Components



2.1.5. Sub-Components

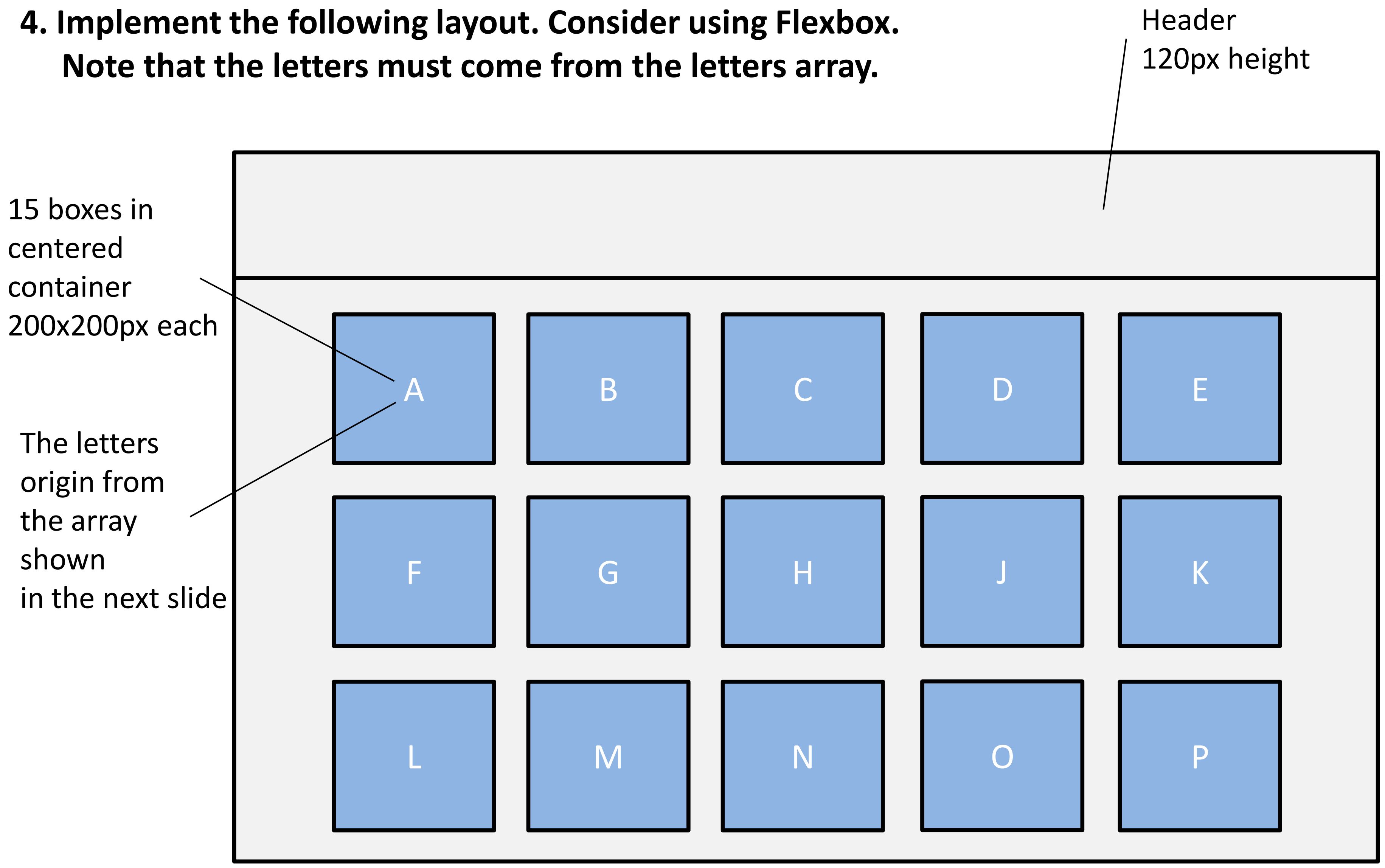


2.1.5. Sub-Components



2.1.5. Task

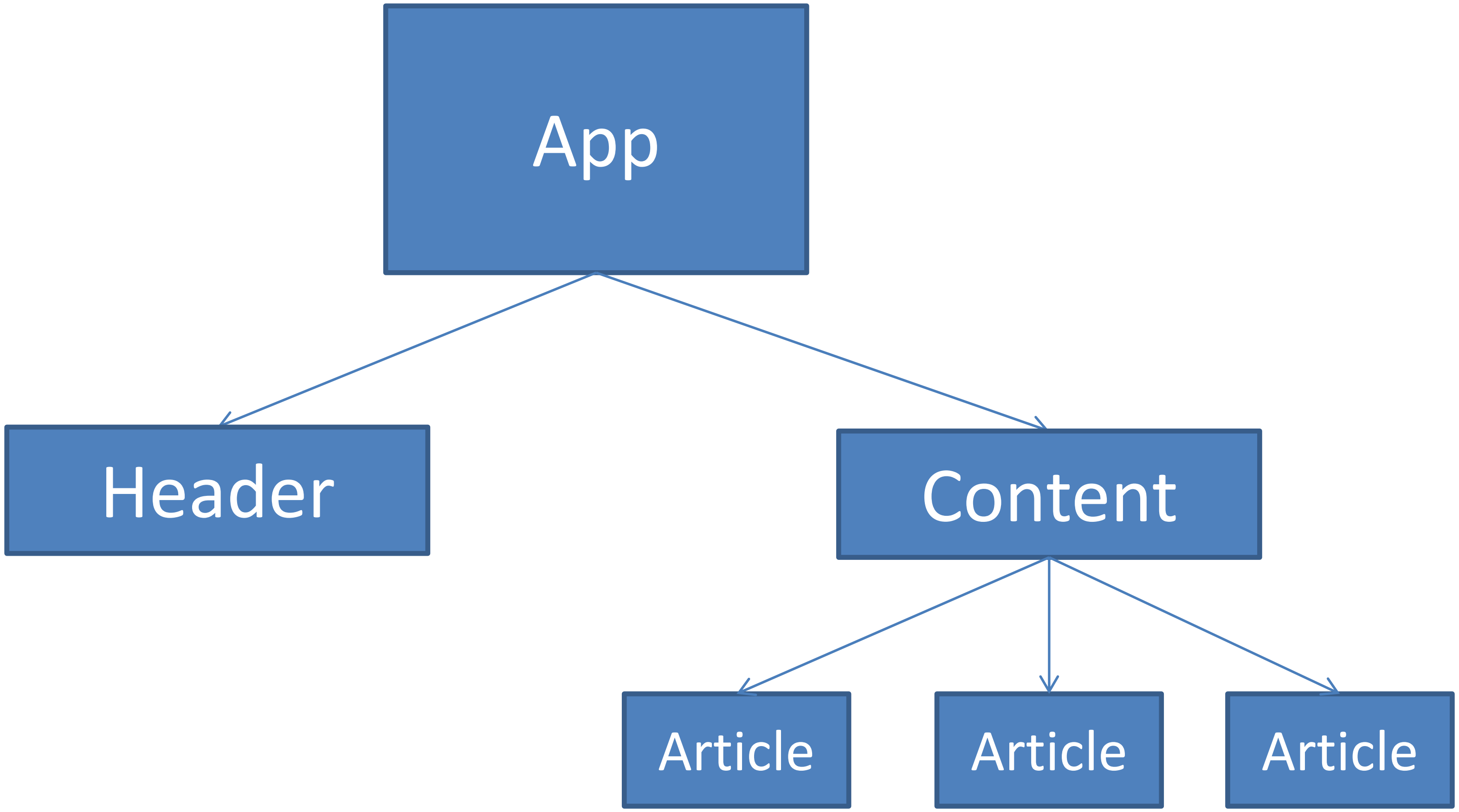
4. Implement the following layout. Consider using Flexbox.
Note that the letters must come from the letters array.



2.1.5. Task

1. Create a new React web app „task-2-1-5“
2. Create two components „Header“ and „Content“. Put them in the App component.
3. In the Header component, create an h1-Element containing the words „Hallo World“. Give Header the background-color „cornflowerblue“ and a height of 80px.
4. Create the component „Article“. It contains a div with background-color a bit brighter than cornflowerblue. Inside the div, write a random text of 25 characters.
5. Put three instances of Article inside the „Content“-component.

2.1.5. Task



Agenda – Part 2.2. States & Events

1. Changing State
2. Events
3. Events that change the state

2.2.1. What is a state?

- **A component‘s state is an object named „state“ inside the component**
    1. It is accessible via this.state
    2. If the state changes, the render() method will be called again
    3. The state can only be changed using this.setState() – it cannot be changed directly

2.2.1. Task

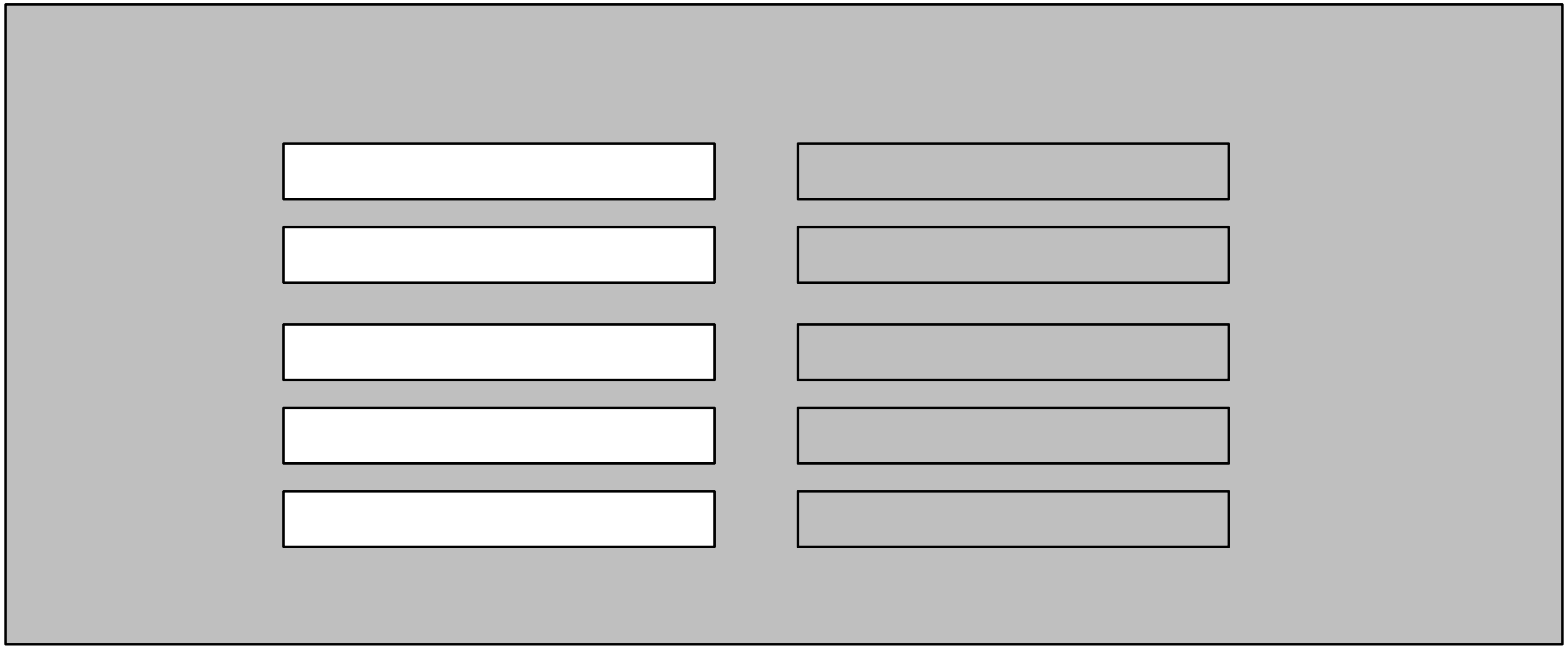
1. Create a new web app „task-2-2-1“
2. Create a new state in the App component and add a new empty array „randomstrings“ to it.
3. „randomstring“ is an array of strings. Create a function randomstring(n) inside the App component that returns a random string of length n. I.e.
    1. randomstring(3) could return ‘ghj‘.
    2. randomstring(7) could return ‘fkdllbx‘
4. Add an interval that adds a new randomstring to randomstrings array every second.
5. Each time a new randomstring is added, append a new div-Element which contains the new randomstring to the App component.

2.2.2. Events

- **What is an event?**
  - Something that happens.
  - The user clicks the mouse, types text, presses a button on the keyboard, the window is being resized, ... etc.
- In React, we attach events to elements
- Overview of events:
  - https://reactjs.org/docs/events.html

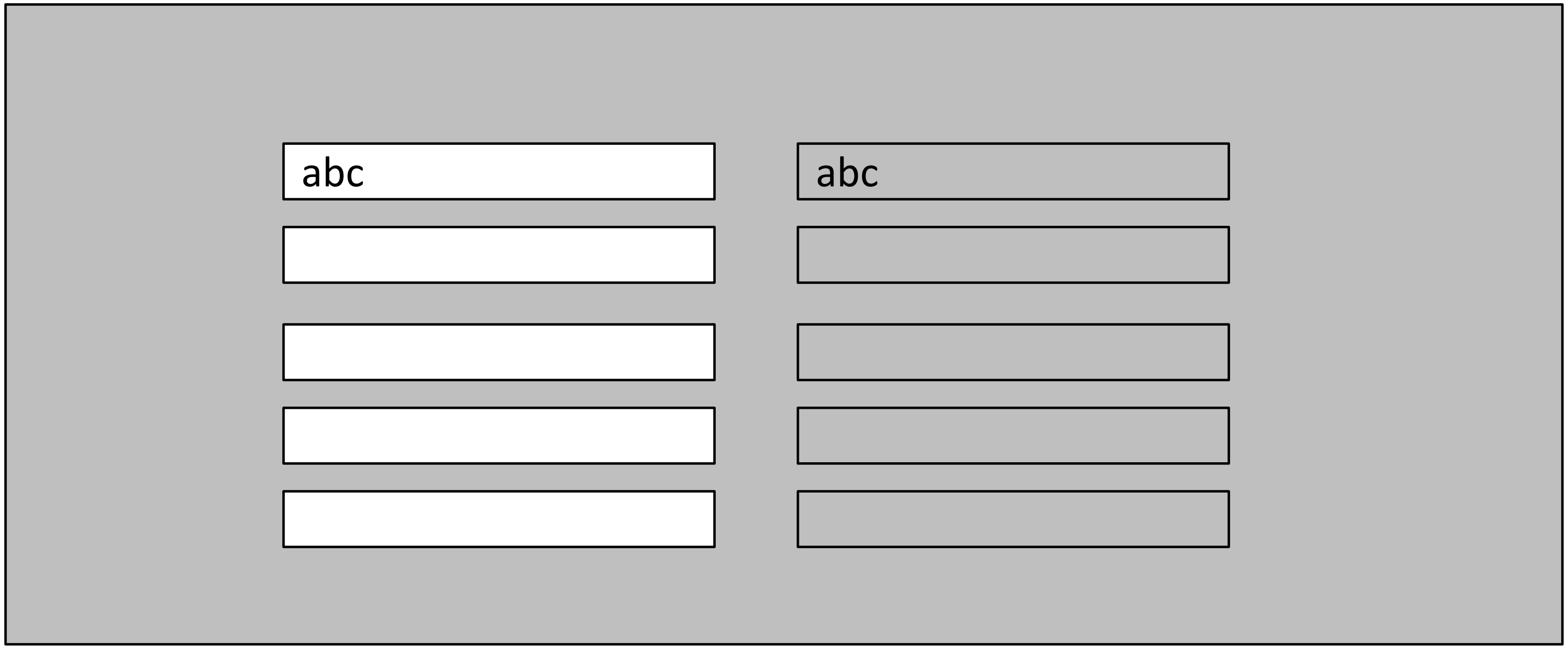
2.2.2. Task

1. Create a new web app „task-2-2-2“
2. Inside the app component, create 5 text boxes and next to them a span each. Give each textbox an unique name attribute, e.g. <input type=“text“ name=“txtBox1“ />



2.2.2. Task

3. When in text box 1 the user enters text, the text should appear in the span next to it, like this:



2.2.2. Task

4. Do the same for the textboxes 2 to 5 by creating 4 more handler functions.
5. Now you have 5 more or less identlical handler functions. Go into one of these handler functions (does not matter which one) and take a closer look at event.target.name - What do you notice?
6. Try to replace the 5 handler functions with one single handler function. Consider using Key Interpolation. (also covered in Chapter 1.8. of this course). In case you forgot about Key Interpolation, it is accessing an object‘s key by a string, e.g.:

   const key = ‘foo‘;
   const obj = {x: 1, foo: ‘hallo‘}
   obj[key] = ‘hi‘; // changes foo to ‘hi‘

2.2.3. Events that change the state

- State: A component‘s state is an object named „state“ inside the component
- Event: Something that happens

-> **Something that happens changes an object named state inside the component**

= **Events that change the state**

2.2.3. Task

1. Make a copy of the “task-2-2-2“ example and save it as “task-2-2-3“.
2. Change the text of the button „show index“ to „remove“.
3. Implement the functionality of the remove – button. Try to use the index to access the array in the state for the removal of the element, consider using splice() – or any other method that would do the same job.
4. Add another button for each fruit with the the text „new color“.
5. Implement the functionality of the „new color“ – button.

Agenda – Part 2.3. Communication between two components

1. Stateful VS Stateless Components
2. Downward Communication via Passing Props
3. Upward Communcation via Passing Function References

3b. Radio Buttons

4. Sub Components with an Ending-Tag

2.3.1. Stateful VS Stateless Components

- A stateful component is a component
  - That has the state-Object inside
- A stateless component is a component
  - That **DOES NOT** have the state-Object inside

- Why?

2.3.1. Stateful VS Stateless Components

- A stateful component is a component
  - That has the state-Object inside
- A stateless component is a component
  - That **DOES NOT** have the state-Object inside

- Why?
  - Stateless Components (also known as **Functional Components**) are supposed to have only little logic inside and their purpose is to represent layout parts of the app

2.3.1. Stateful VS Stateless Components

- A stateful component is a component
  - That has the state-Object inside
- A stateless component is a component
  - That **DOES NOT** have the state-Object inside

- What does this imply?

2.3.1. Stateful VS Stateless Components

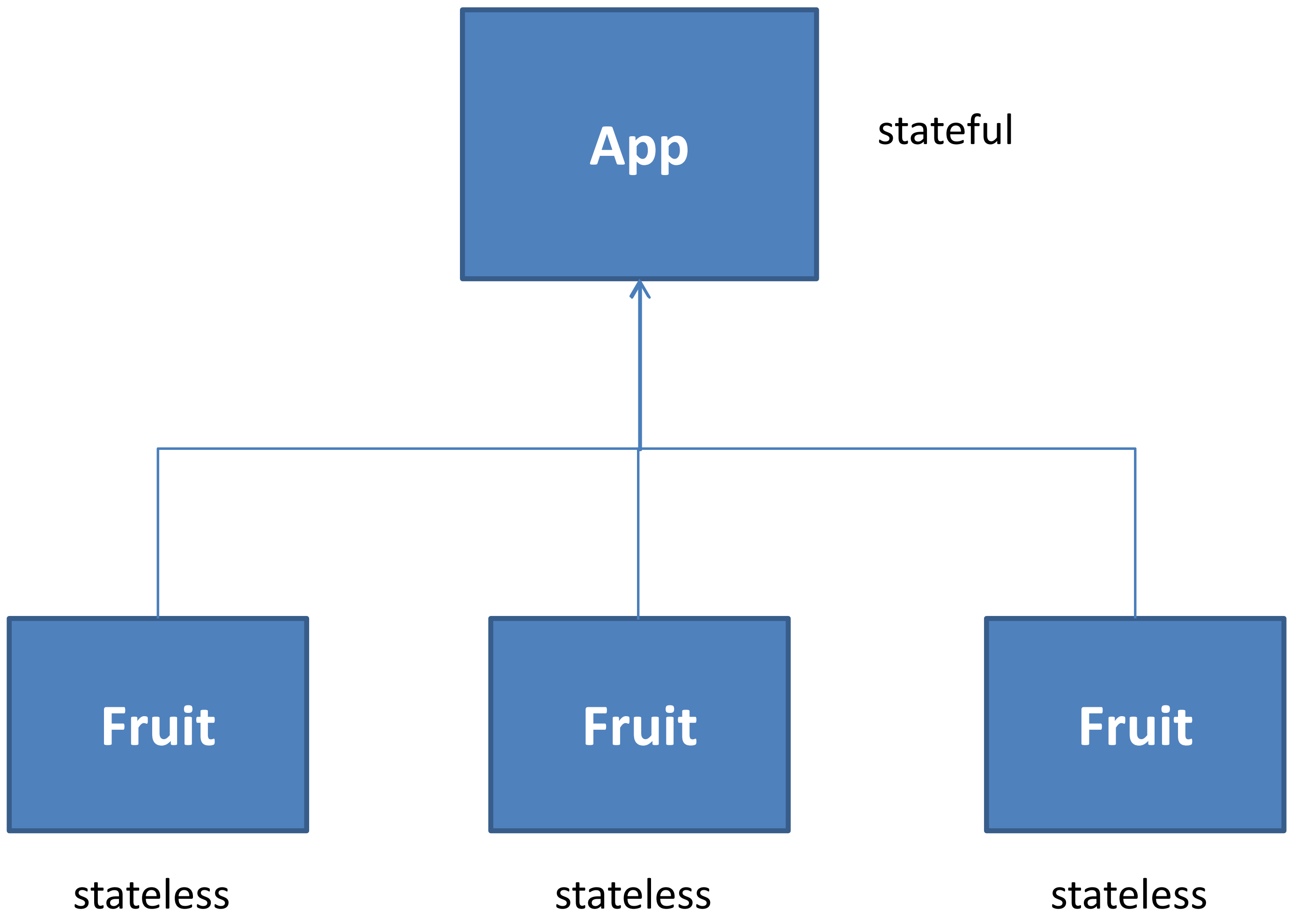
- A stateful component is a component
  - That has the state-Object inside
- A stateless component is a component
  - That **DOES NOT** have the state-Object inside

- What does this imply?
  - Stateful components have the setState() method to re-render themselves
  - Stateless components **DO NOT** have the setState() method to re-render themselves

2.3.1. Stateful VS Stateless Components

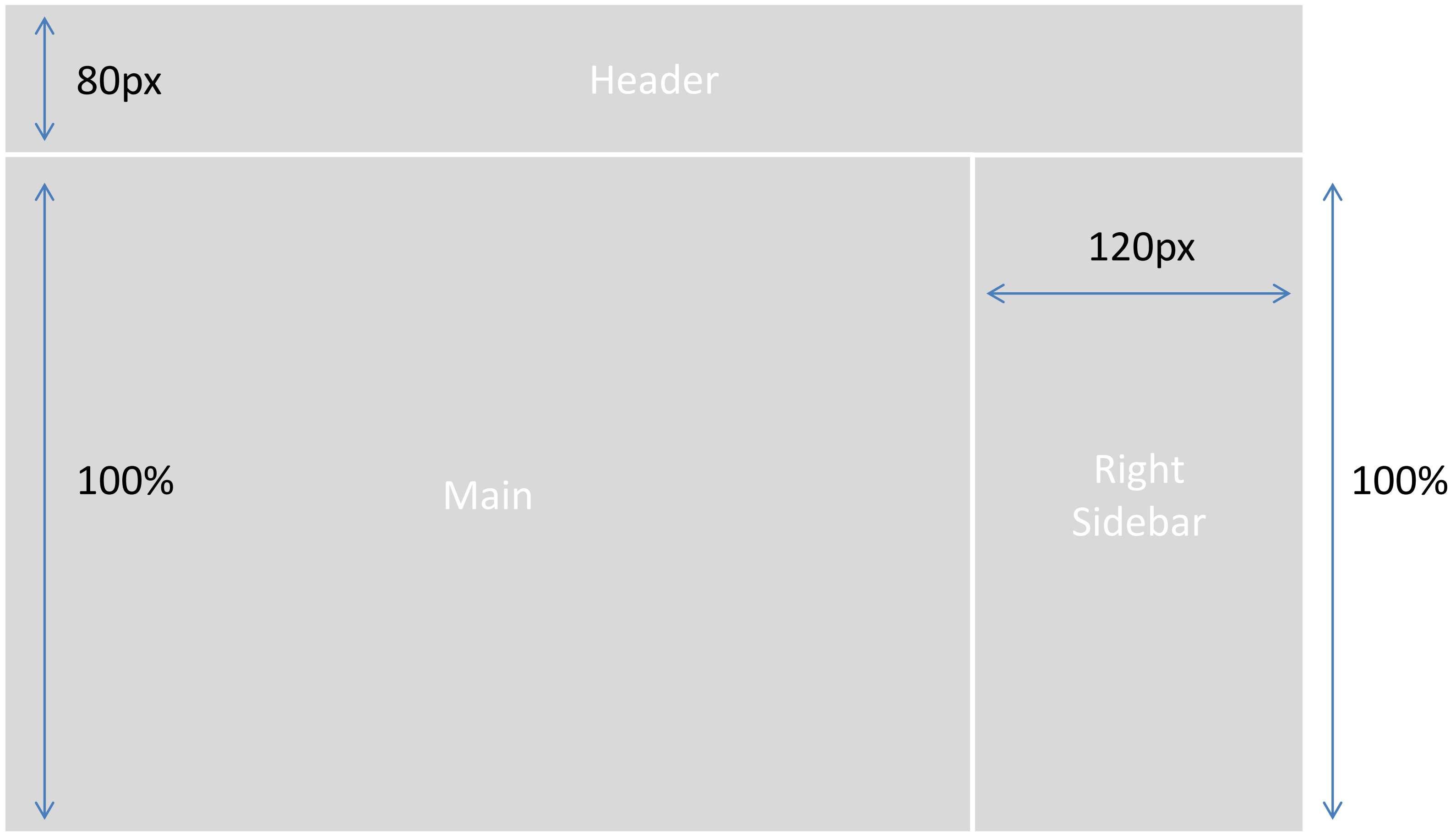
- How do Stateless Components re-render themselves?
  - **<u>They do not</u>** – The rendering is done when the first Stateful Parent Component re-renders, meaning it calls setState()
  - The App-Component can only be a Stateful Component – so when the App-Component calls setState(), all immediate Stateless Components will be re-rendered

2.3.1. Stateful VS Stateless Components



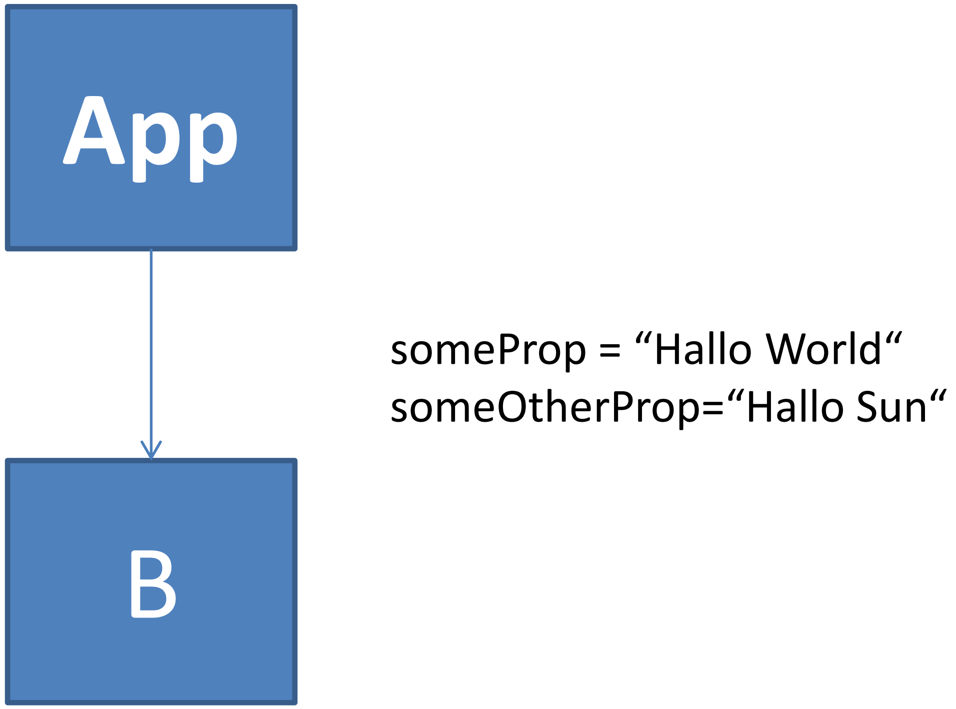
2.3.1. Task

1. Create a new web-app “task-2-3-1“.
2. Implement the following layout and create three components Header, Main and RightSidebar. Make Header and RightSidebar a stateless component and Main stateful.
3. The background-color of the Main component should change every 2 seconds to a random color.



2.3.2. Downward Communication With Props

- Communication from a parent component A down to its child component B can be done using **props**

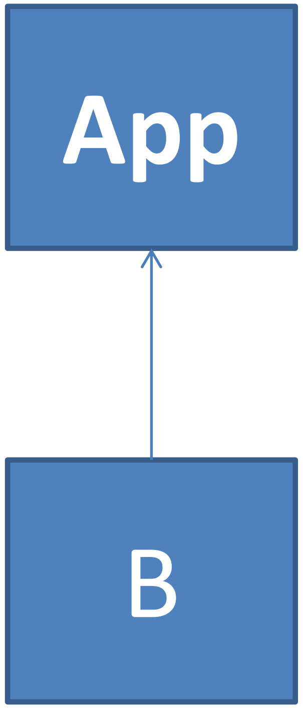


2.3.2. Task

1. Create a new web-app “task-2-3-2“.
2. Create a new stateless component “Fruit“.
3. Inside fruit, there is only one div with the background color red and text inside „Apple“.
4. Create 3 instances of Fruit in your App component.
5. Add one prop the Fruit-component: color
6. Now, give each Fruit component it‘s color via the color prop.
7. Add a button to the App-Component: „Randomize Apples“.
8. Add the following functionality: If the user clicks the „Randomize Apples“ button, each of Fruit components gets a new background color. Possible colors are red, blue, green, black, yellow, pink, fuchsia and grey.

2.3.3. Upward Communication With Function References

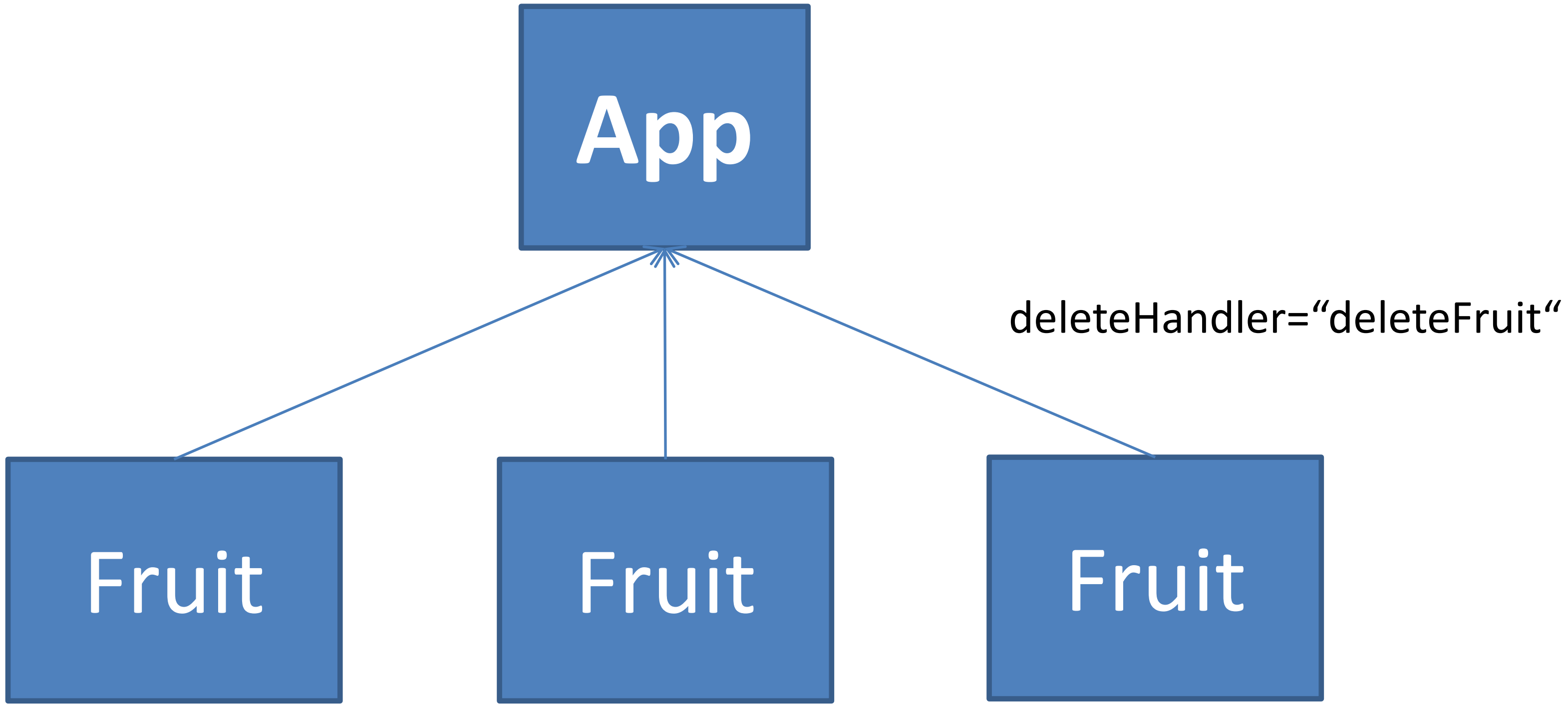
- Communication from child component B upward to parent component A can be done using <u>**function references**</u>



callMe="someFunction"

2.3.3. Upward Communication With Function References

- A function being called in the parent component can change its state

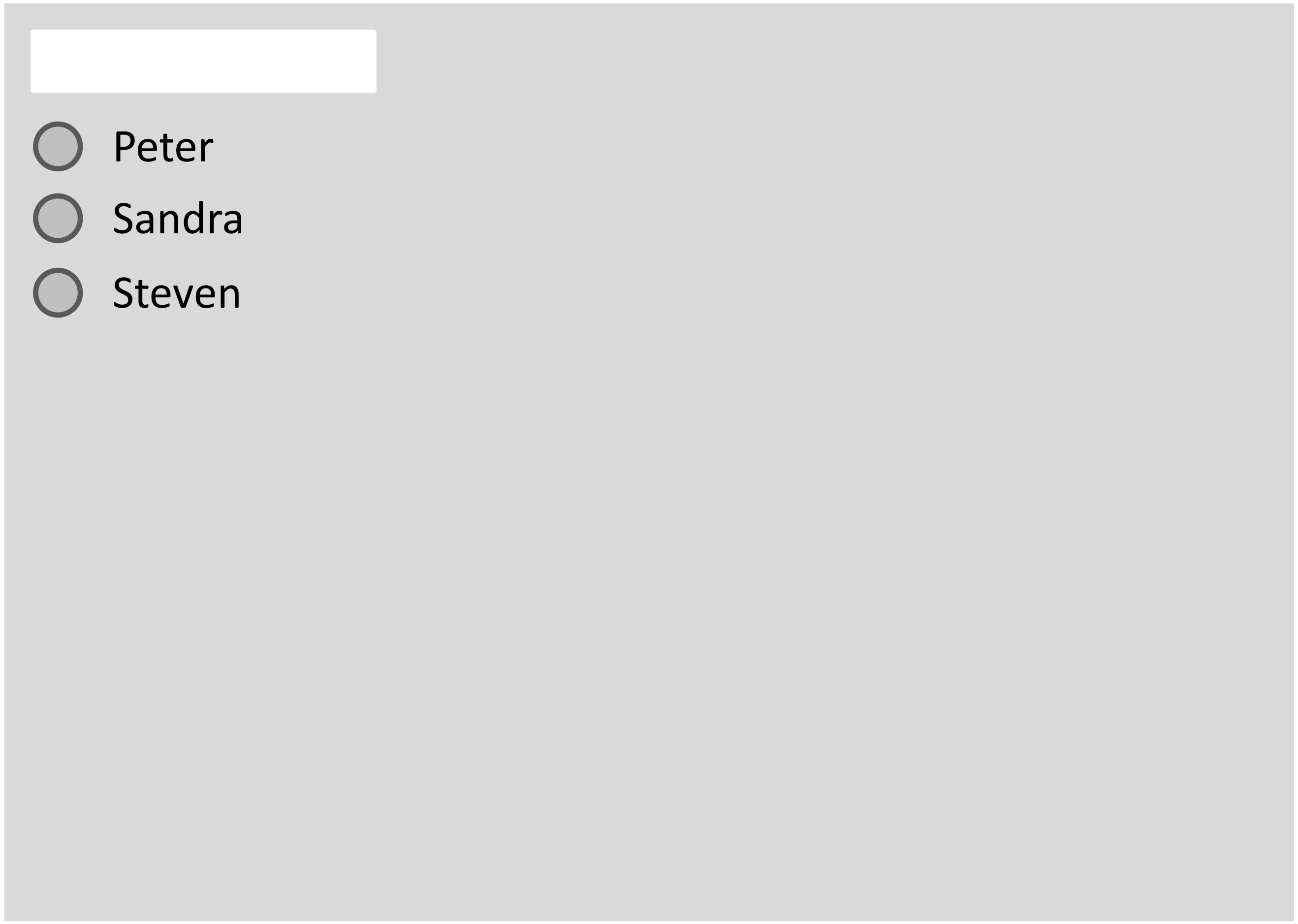


2.3.3. Task

1. Create a new web-app “task-2-3-3“.
2. In the state object, create an array “users“ which consists of 3 example user objects whereas each has a name and a distinct id. I.e. take the users Peter with id = 1, Sandra with id = 2 and Steven with id = 3
3. Create a textbox with the name “username“.
4. Create a component User.
5. Inside User, create a radio button and next to it, an empty span element.
6. Give the User component two props “username“ and “id“. The value of username is supposed to be shown inside the span element. The value of id is supposed to be the value of radio buttons.
7. Based on the users array in the App component‘s state, create instances of the User object.

Your layout now may look like this ... (next slide)

2.3.3. Task



2.3.4. Sub Components with an Ending Tag

- Until now, all of our components are self-closing components:

```
<A />
<Fruit />
<User />
```

- Self-Closing components cannot contain children, there we need Subcomponents with an Ending-Tag, i.e.

```
<B></B>
```

2.3.4. Task

1. Create a new web-app “task-2-3-4“.
2. Create a new stateless component “MyButton“.
3. Inside MyButton, there is a div and inside that div, there is a button. All children of MyButton should be inside the button-Element.
4. Each MyButton has the background color of „cornflowerblue“, the font size of 20 pixels and a padding of 4 pixels.
5. When the button inside of button is clicked, the function reference onClick of MyButton will be called.
6. Create two MyButtons, one that contains the text “Hallo World“ and another one that contains a nice picture of a beautiful beach.
    1. When the first MyButton is clicked, an alert box shall appear saying “HalloWorld“
    2. When the second MyButton is clicked, an alert box shall appear saying “Beach Life! Me gusta!“

Agenda – Part 2.4. Component Lifecycle

1. Overview
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase
5. Task 1
6. Task 2 – Difficult

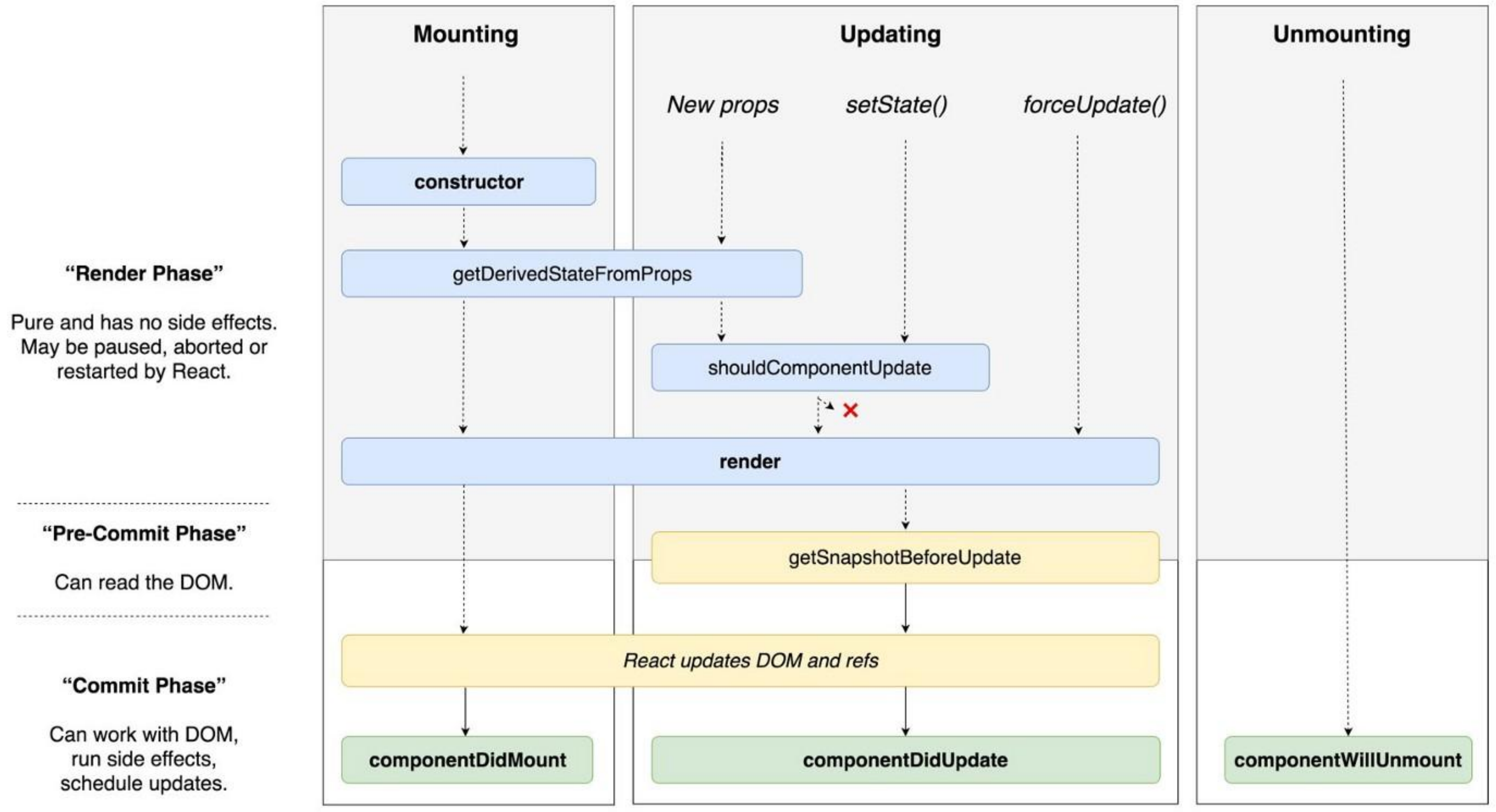
2.4.1. Overview

- Each Stateful Component
  - Is born = The component **mounts**
  - Lives = The component **updates**
  - Eventually Dies = The component **unmounts**
- In each of these lifecycle phases, React calls certain methods of the component
- Each method is initially empty!
- the developer writes code do define the component‘s behaviour when a lifecycle phase occurs

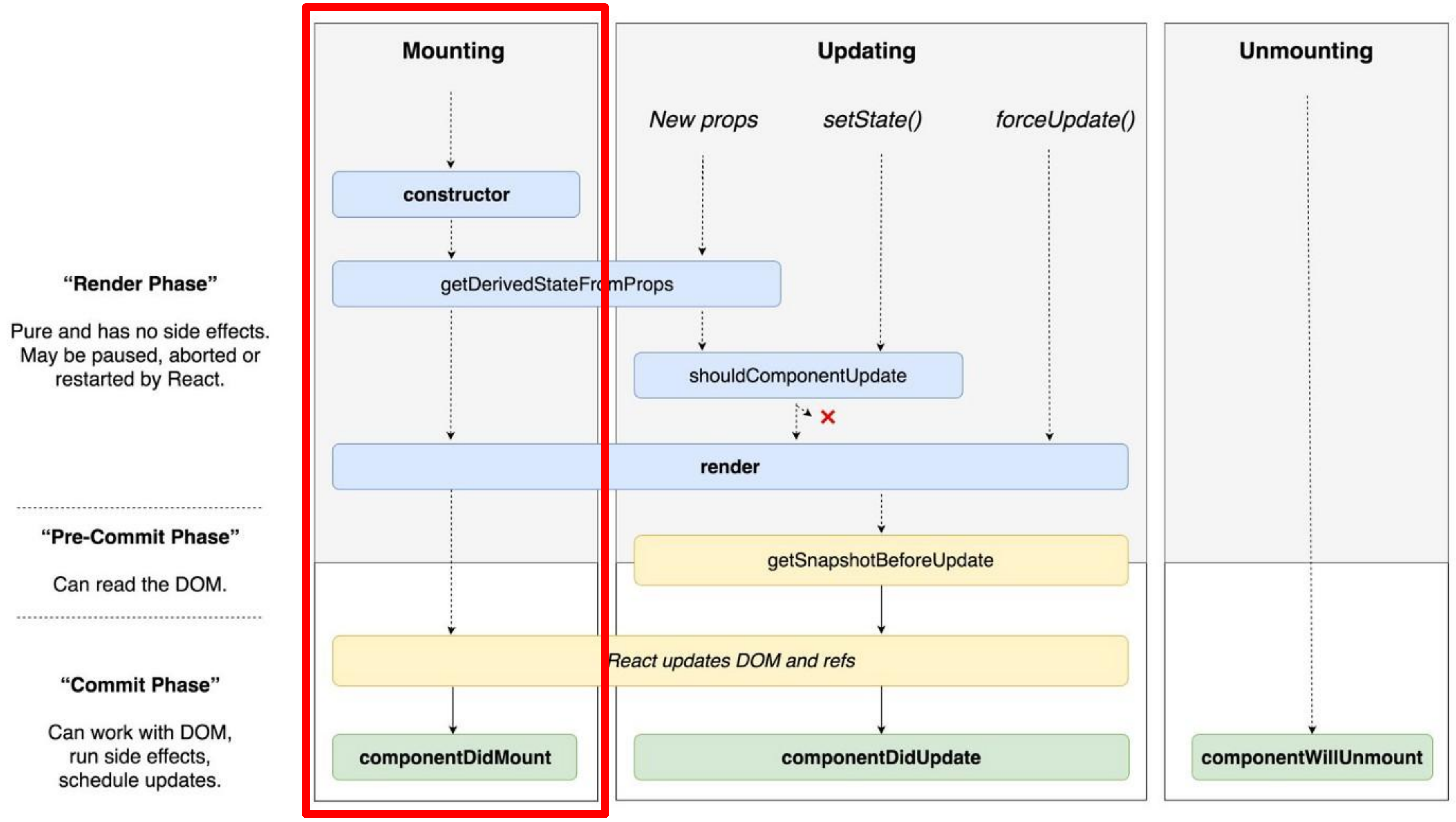
2.4.1. Overview

- Three lifecycle phases
  - Mounting Phase
    - React creates an instance of a component and inserts it into the DOM
  - Updating Phase
    - When props or state of a component are changed
  - Unmounting Phase
    - When the component is removed from the DOM

2.4.1. Overview



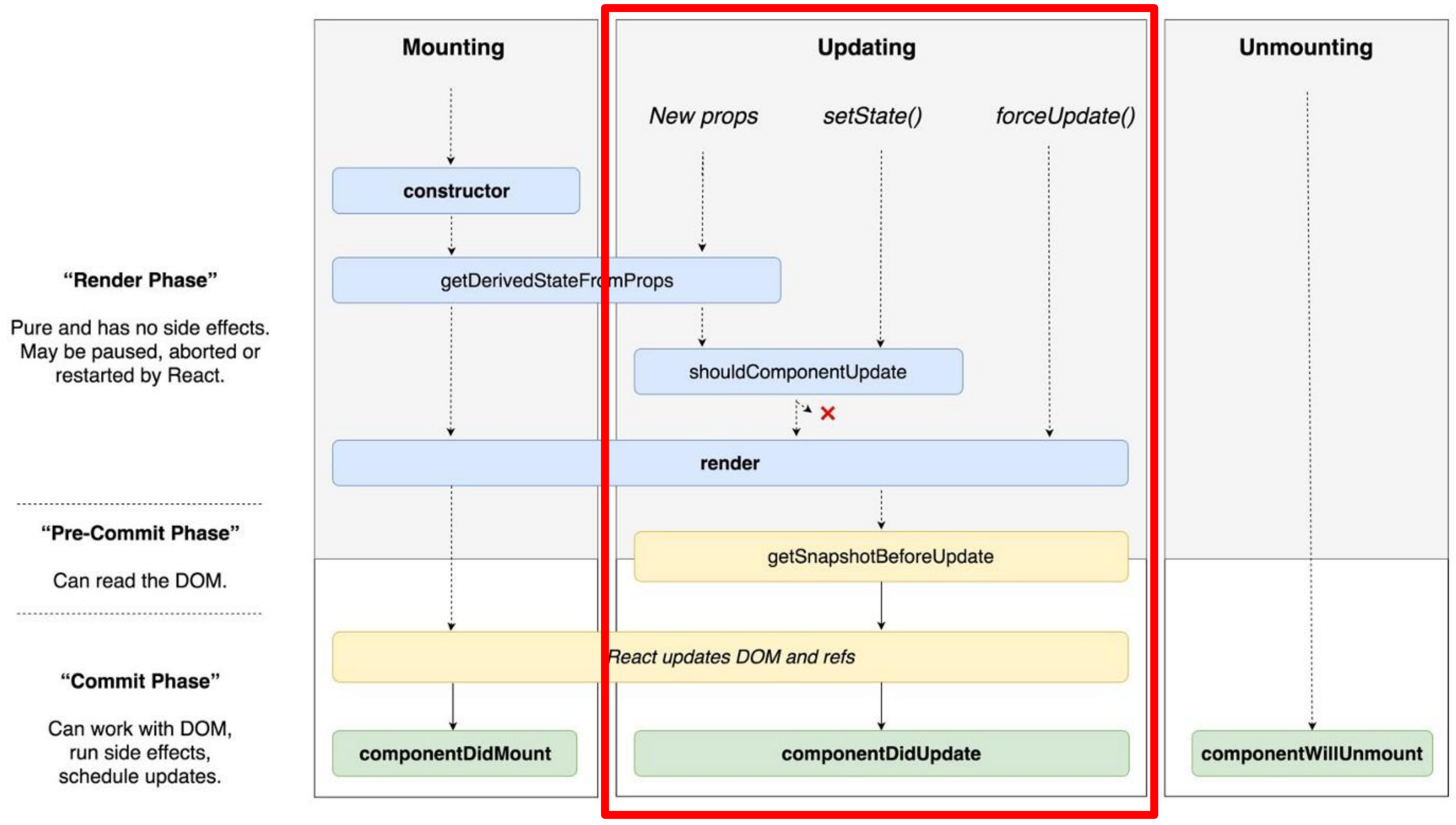
2.4.1. Overview



2.4.2. Methods of Mounting Phase

1. constructor()
   – An instance of the component class is created
2. static getDerivedStateFromProps()
   – Called on every render() – the first render happens in the mounting phase
3. render()
   – Converts VDOM into DOM
4. componentDidMount()
   – After the render method is called

2.4.3. Methods of Updating Phase



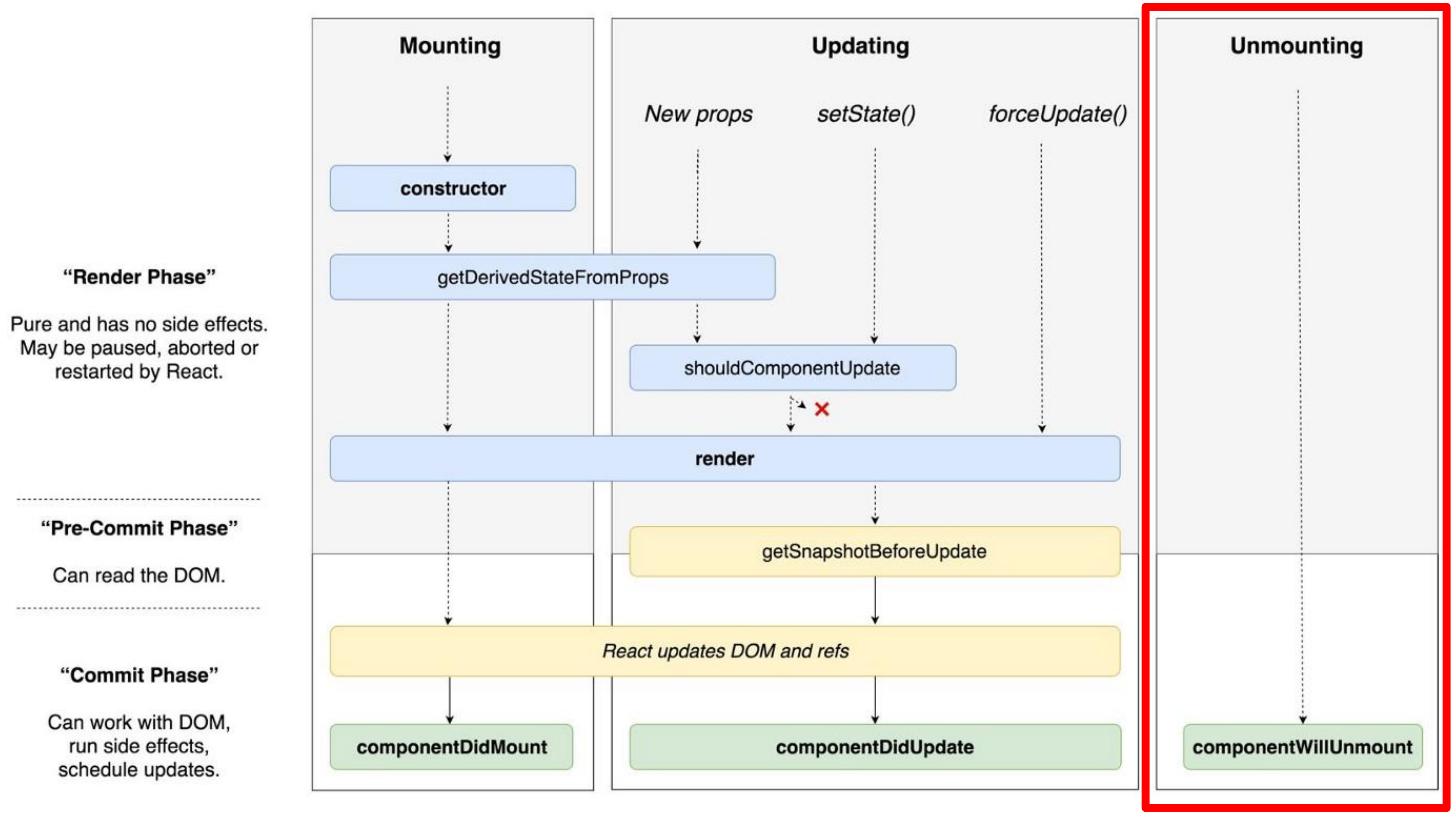
2.4.3. Methods of Updating Phase

1. static getDerivedStateFromProps()
2. shouldComponentUpdate()
3. getSnapshotBeforeUpdate()
4. render()
5. componentDidUpdate()

2.4.4. Methods of Unmounting Phase

1. componentWillUnmount()

2.4.4. Methods of Unmounting Phase



2.4. Task

1. Create a new react-app “task-2-4“
2. In the App component‘s state, create an array randomStringLengths consisting of three random numbers. Each number can either be 4, 5 or 6.
3. Add a button labeled as „Generate New Random String Lengths“
4. When the button of 3) is clicked, new values for randomStringLengths will be generated.
5. Create a new stateful subcomponent „RandomStringGenerator“ that only has one div.
6. The state of RandomStringGenerator solely consists of one variable randomString.
7. Inside RandomStringGenerator‘s div, the current value of this.state.randomString must be shown.
8. In RandomStringGenerator, create the method generateRandomString(n) that returns a random string of length n.
9. Add a new prop “stringLength“ for the RandomStringGenerator component.
10. Create three instances of RandomStringGenerator inside the App component, based on the randomStringLengths array. For each instance, pass down the number as „stringLength“-prop.
11. Implement the following behaviour with your knowledge of component lifecycles:

    a) Whenever RandomStringGenerator receives a new value for stringLength, it generates a new value of this.state.randomString.

    b) Whenever it receives the same value again, RandomStringGenerator does not render.

Hint: Using the Chrome-Debugger might be helpful.

Agenda – 2.5. Routing

1. Definition of a Route

2.5.1. Definition of a Route

- **A route is an address of a resource**
- Resources are exposed by webservers to the Internet
  - https://www.google.com -> Resource / of google.com
  - https://www.linkbox.io/jan -> Resource /jan of Linkbox.io
  - https://www.linkbox.io/jan/music -> Resource /jan/music of linkbox.io
- **In React, a resource is the address of a component**
  - /home -> Renders component Home in App
  - /home -> Renders can also render component Imprint in App
  - / -> Renders App component

2.5.1. Task

1. Create a new web-app “task-2-5“.
2. Create a subcomponent Start that solely contains one div which says „This is the Startpage“.
3. Create another component Users.
4. Open the link: http://jsonplaceholder.typicode.com/users and copy everything into the clipboard. Afterwards, paste it into your Users and assign it as new constant. I.e.

   const users = [{ “id“: 1, “name“: “Leanne Graham“, ...]

5. Create another component UserDetail.
6. Do exactly the same as in 4) for the UserDetail Component.
7. In the App Component, create a Router and a navigation that links the following Routing to the respective Components

   / -> Start
   /users -> Users
   /userdetail/:id -> UserDetail

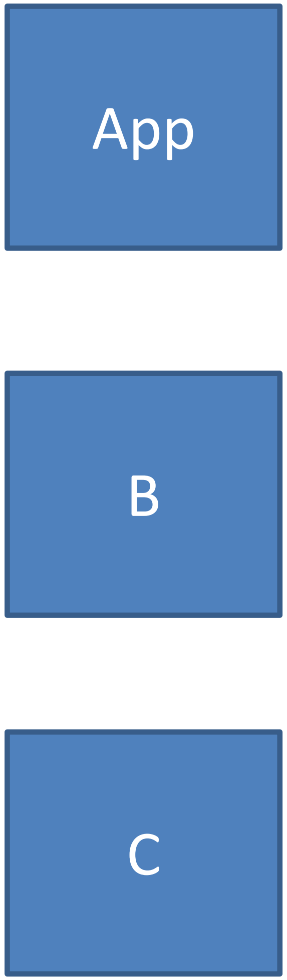
   Please note, that the last route expects an parameter.
8. In the Users component, show the name, the email and the phone of each user in a tabe.
9. In the table of 8), add another column with a Link “Details“ that links the UserDetail component with the id of the respective user.
10. In the component UserDetail, show only one div that consists of the name of the user with id passed as parameter.

Agenda – Part 2.6. Communication between three or more layers of Components

1. Props Drilling
2. Context API
3. Redux
4. Redux Thunk

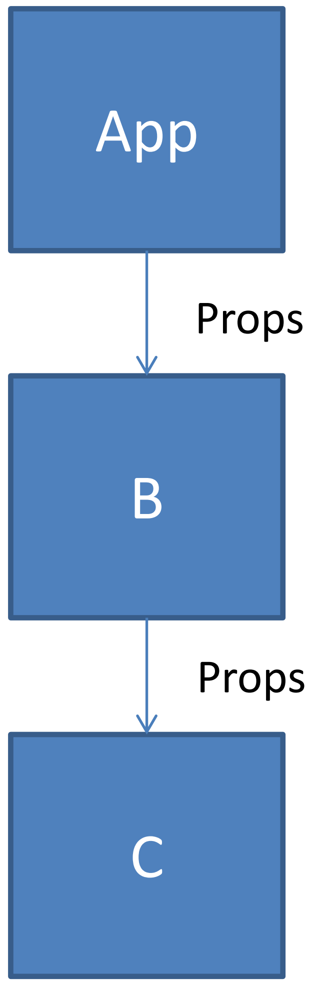
2.6.1. Props Drilling

Props Drilling =



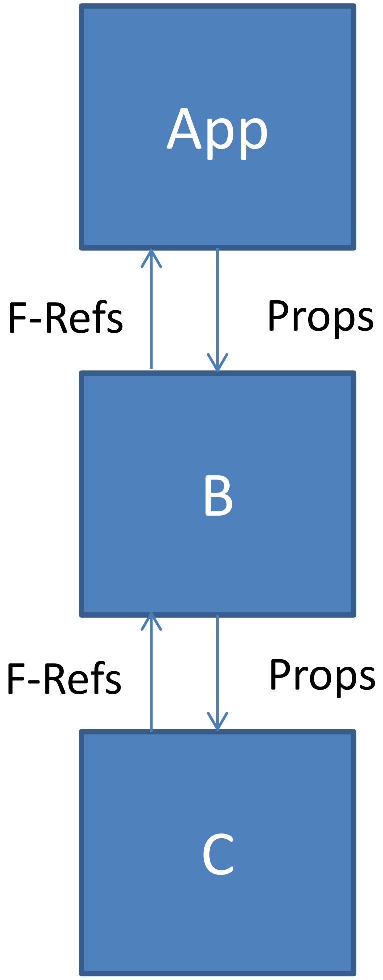
2.6.1. Props Drilling

Props Drilling = Passing props down child by child



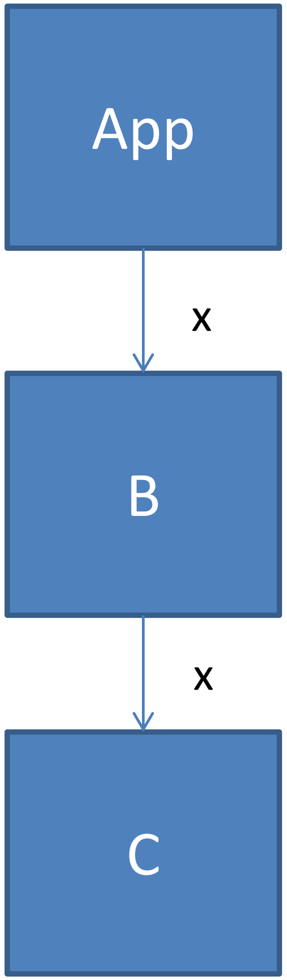
2.6.1. Props Drilling

Props Drilling = Passing props down child by child
Passing function references up parent by parent



2.6.1. Props Drilling

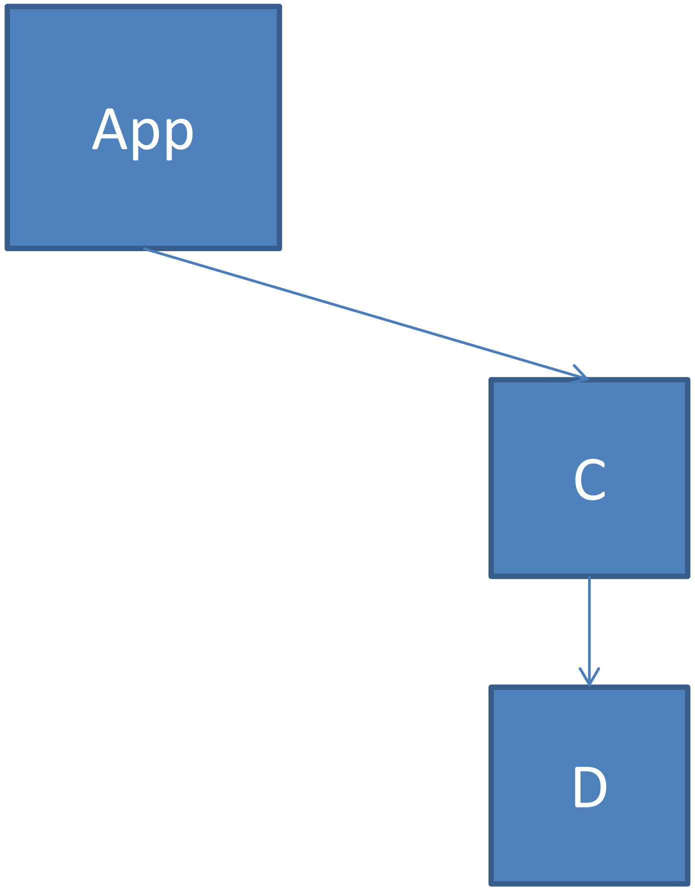
<u>Props Drilling</u> = Passing props down child by child
Passing function references up parent by parent



2.6.1. Task

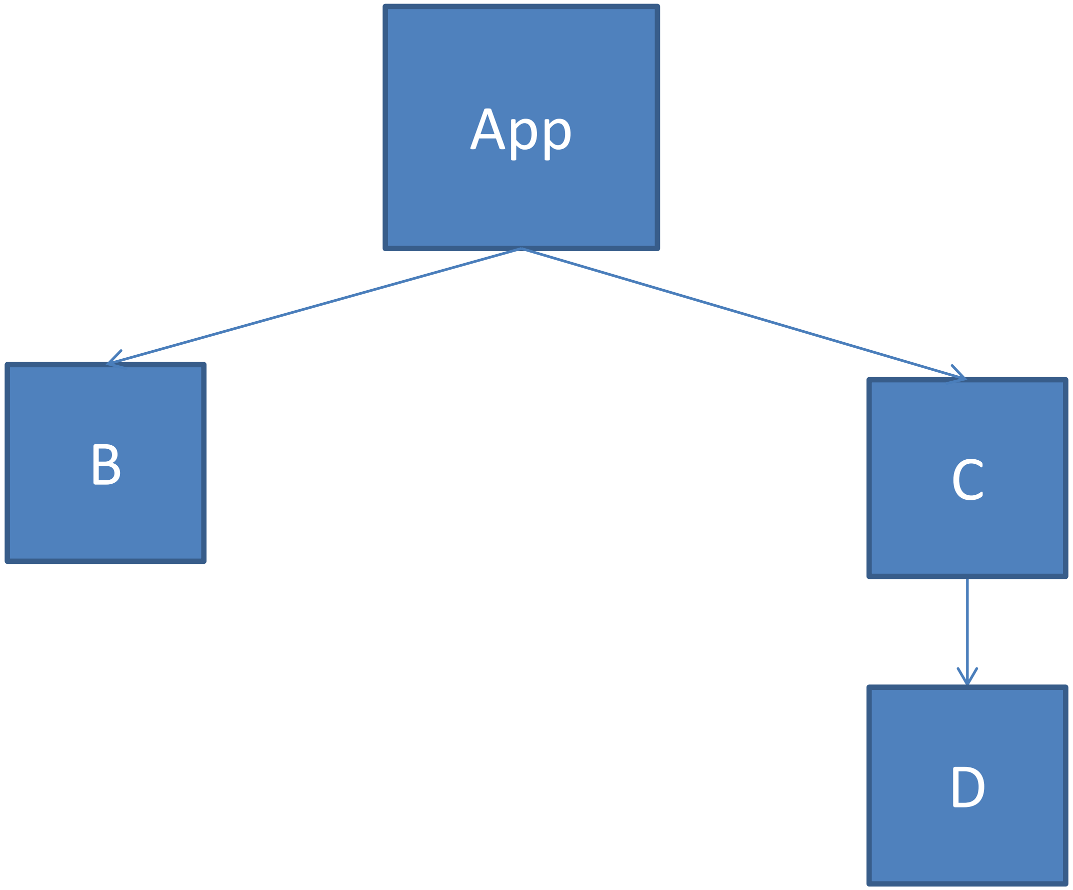
1. Create a new web-app “task-2-6-1“.
2. Inside the App component, create a header that says “App Component“ and underneath a button with the value „Generate Random Number“.
3. When the button of 2) is clicked, a new random number between 0 and 9 shall be generated and saved in App‘s state as randomNumberOfApp.

2.6.1. Task



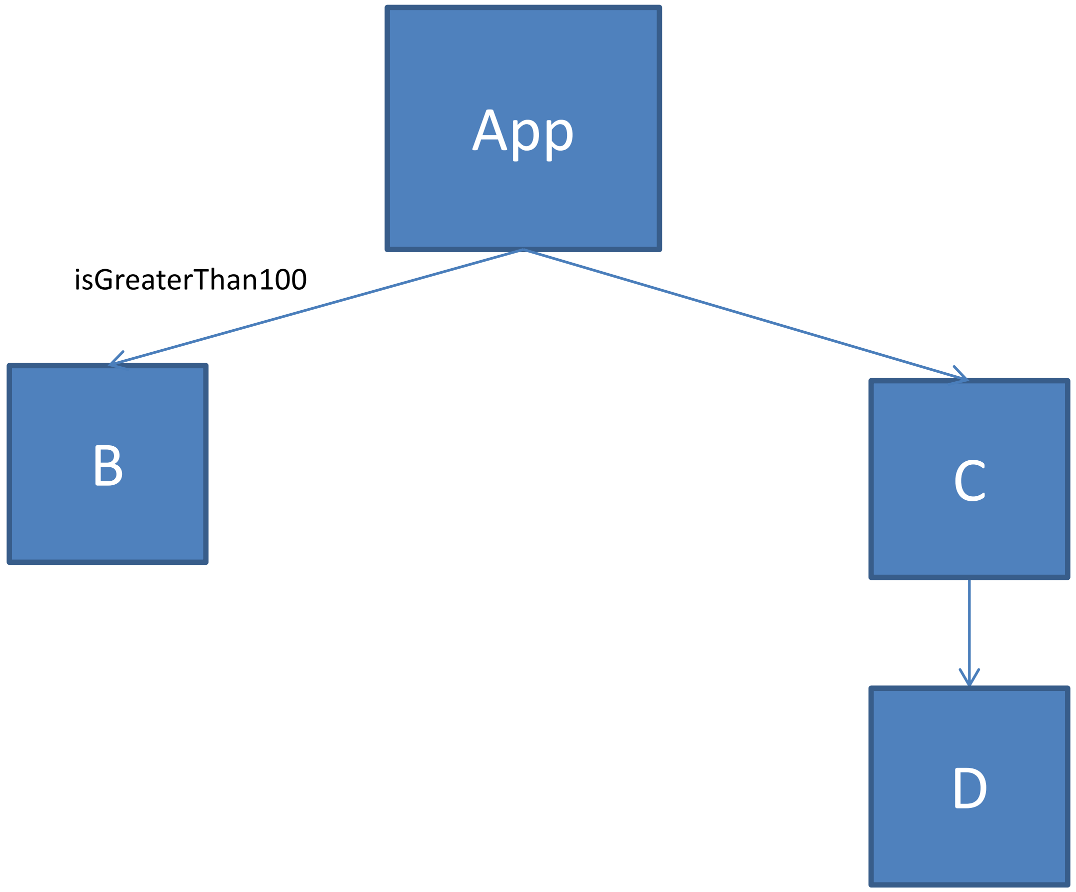
4. Create two stateful Components C and D.
5. Instantiate C inside of App
6. Instantiate D inside of C.

2.6.1. Task



6. Create a stateless component B and instantiate it in the App component. In B, create a header saying “Component B“.

2.6.1. Task



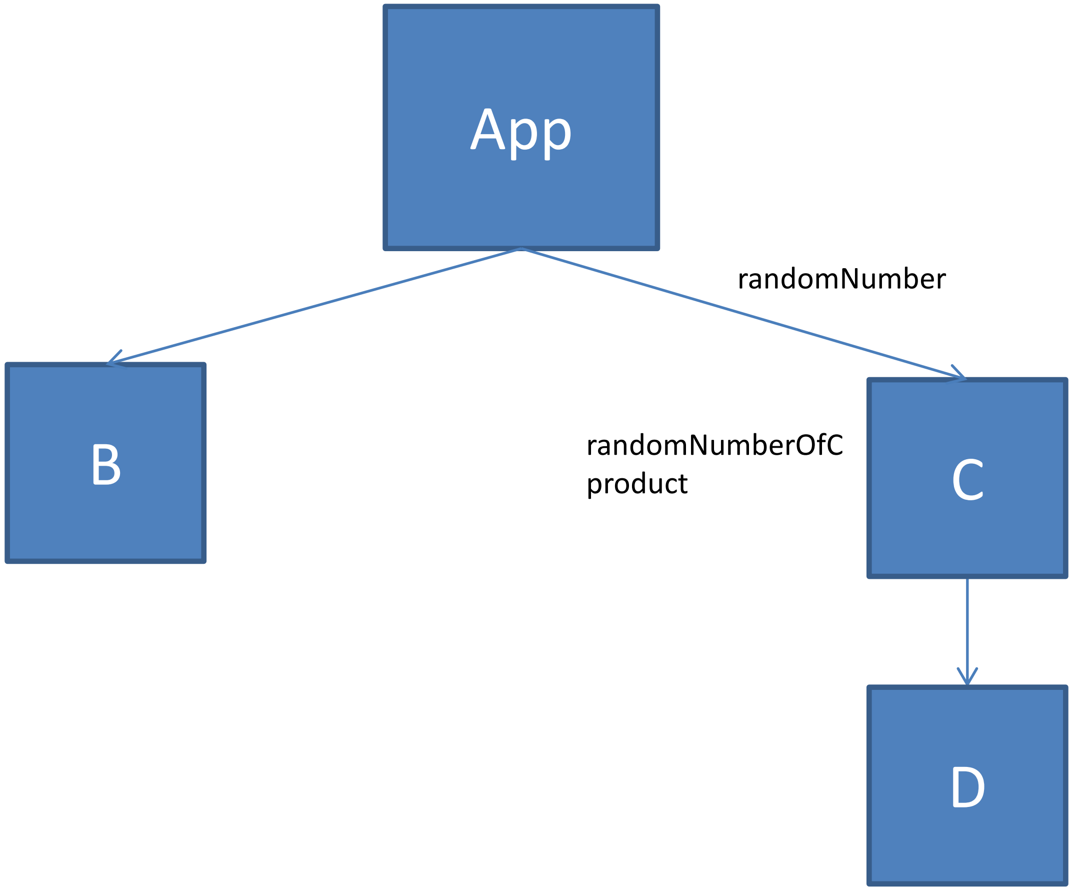
7. Inside App, create the method “greaterThan100“ with one boolean parameter isIt.

If isIt is true, B (<u>not App</u>) shall show the text *“The product of the three random numbers is greater than 100“*

If isIt is false, B shall show the text *“The product of the three random numbers is less or equal than 100.“*

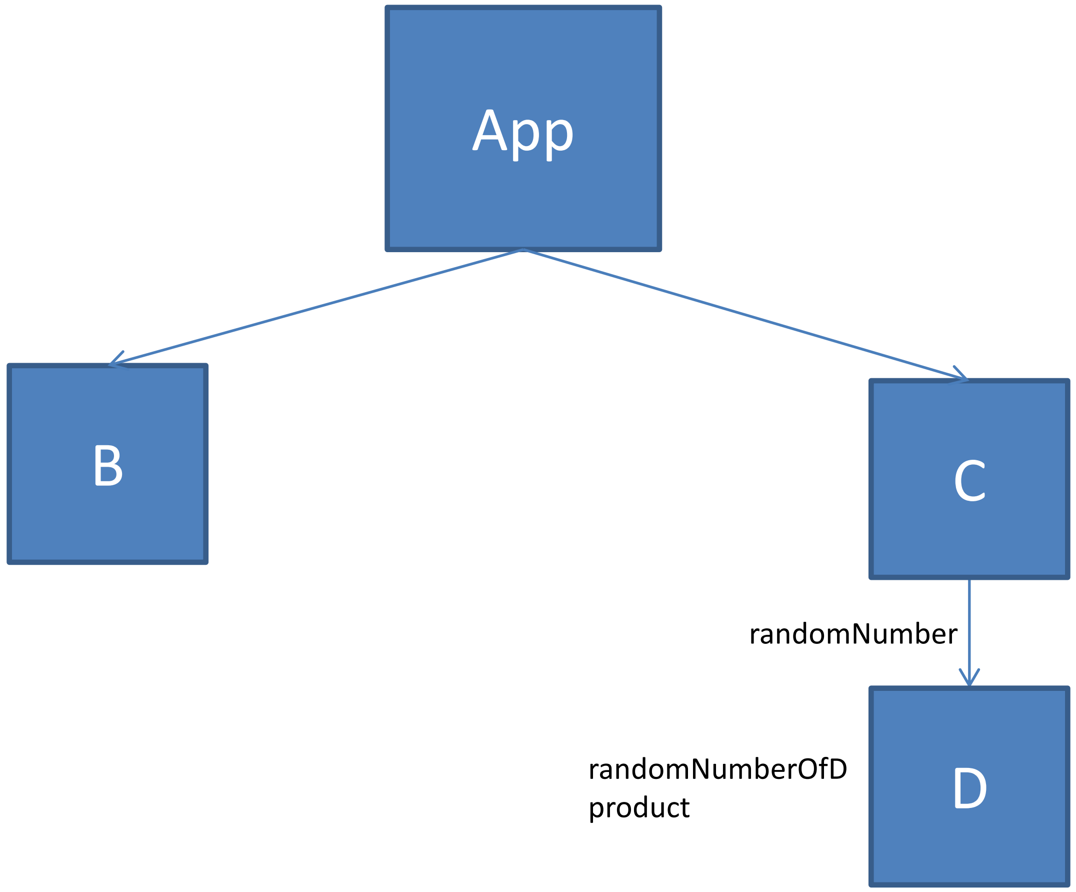
-> use a prop “isGreaterThan100“ for this

2.6.1. Task



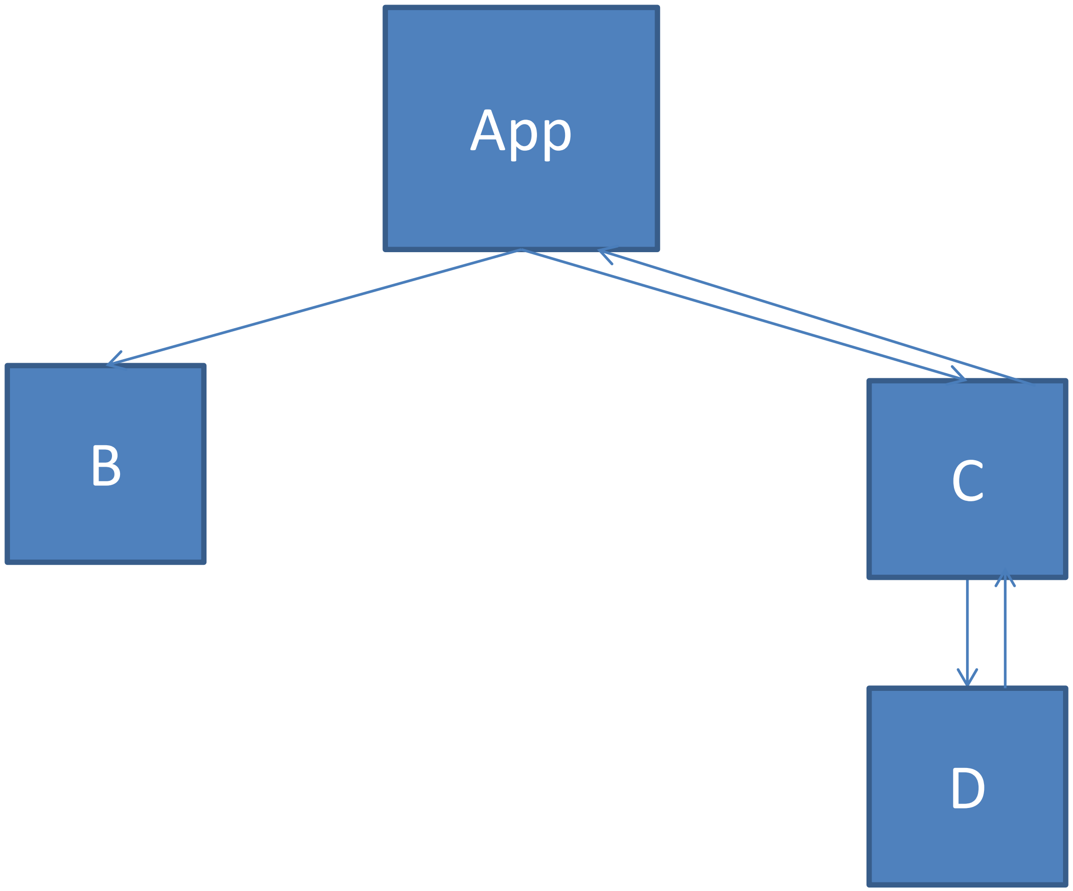
8. Pass App‘s randomNumberOfApp down to C as prop “randomNumber“
9. After C has received App‘s randomNumber, C generates an Internal random number between 0 and 9 and saves it as randomNumberOfC. Then, C calculcates the product of randomNumberOfC and the randomNumber received by App and saves that internally as “product“.

2.6.1. Task



10. Pass C‘s randomNumberOfC down to D as prop “randomNumber“
11. After D has received C‘s randomNumber, D generates an Internal random number between 0 and 9 and saves it as randomNumberOfD. Then, D calculcates the product of randomNumberOfD and the randomNumber received by C and saves that internally as “product“.

2.6.1. Task

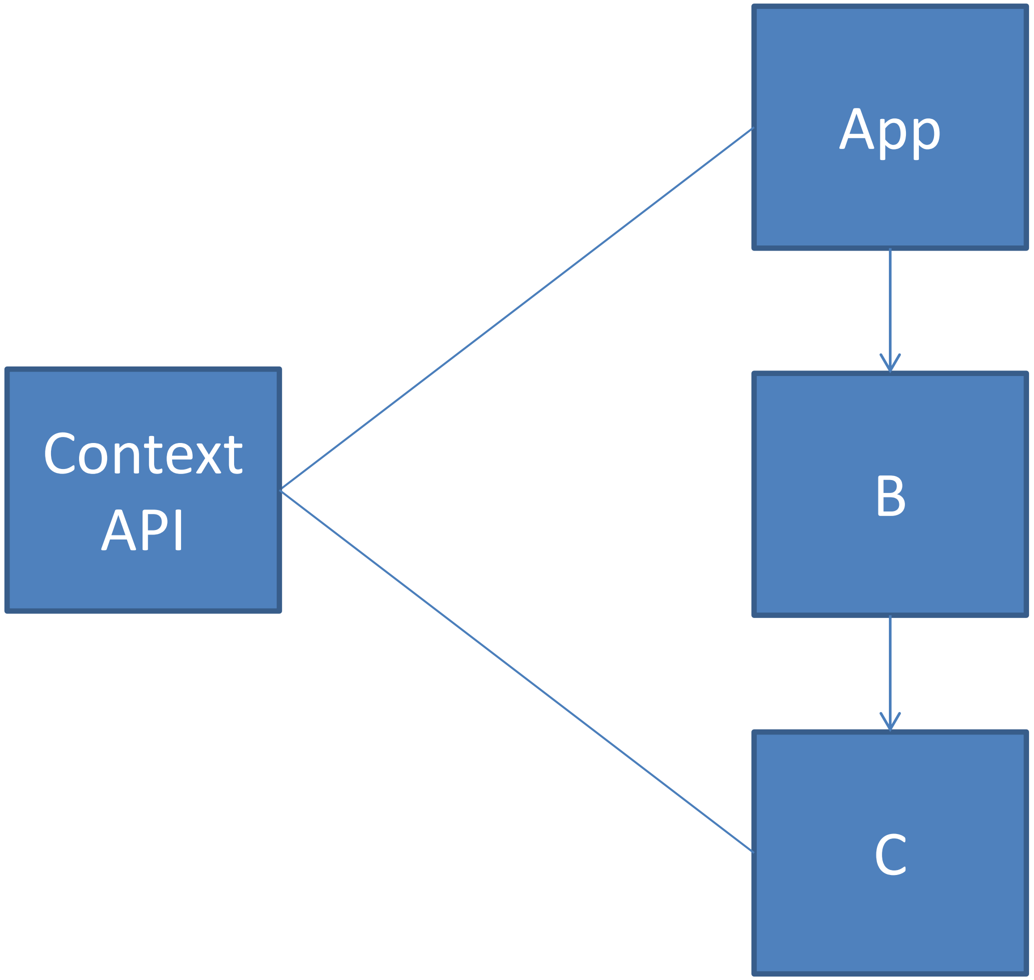


12. If D's product is greater than 100, D calls App's method greaterThan100(true). If D's product is smaller or equal 100, D calls greaterThan100(false). Therefore, implement an upward communication from D to App.

13. Make sure that the render methods of C and D are not called too many times by React.

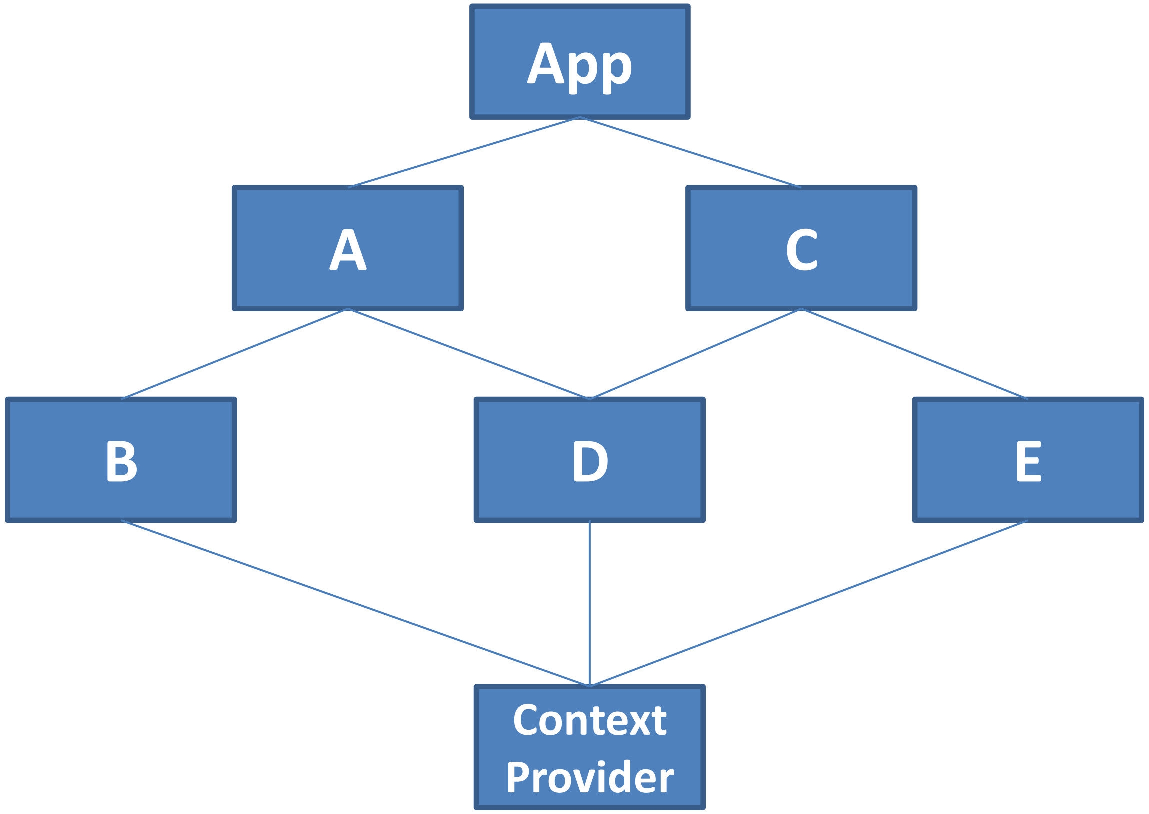
2.6.2. Context API

Context API = A component,
that shares its state with
other components.



2.6.2. Task

1. Implement the following component architecture using the Context API.



2.6.2. Task

2. The context has two variables, x and y. Both are initially set to 0. Furthermore, the context has two functions incrementX and decrementY. incrementX sets x to x + 1 and decrementY sets y to y – 1. Implement the context!
3. Component B has a button that calls incrementX
4. Component D shows the current value of x and y.
5. Component E has a button that calls decrementY

2.6.3. Context API with Reducer

Context API functions can be indirectly called via a **Reducer** which maps messages to function calls.

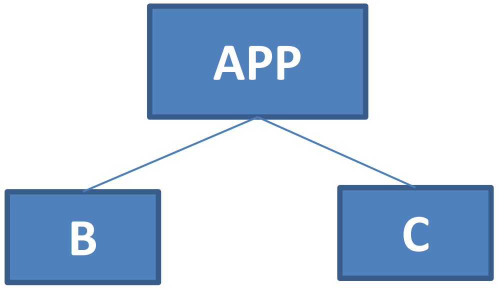
„INCREMENT_X“ -> incrementX()
„DECREMENT_Y“ -> decremenY()

2.6.3. Task

1. Create a new web-app “task-2-6-3“.
2. Refactor your solution of task 2.6.2. to use a Reducer.

2.6.4. Redux

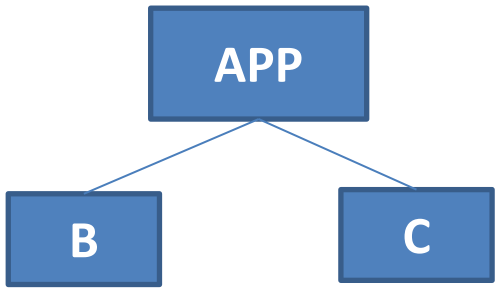
3. Redux = One or multiple shared states managed by a store

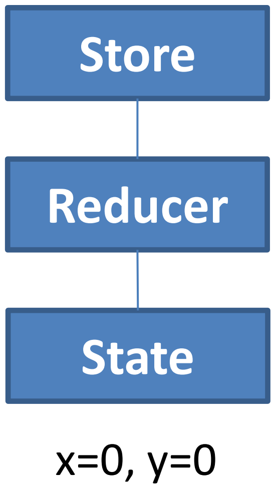


1. Create your App and it‘s child components

2.6.4. Redux

3. Redux = One or multiple shared states managed by a store

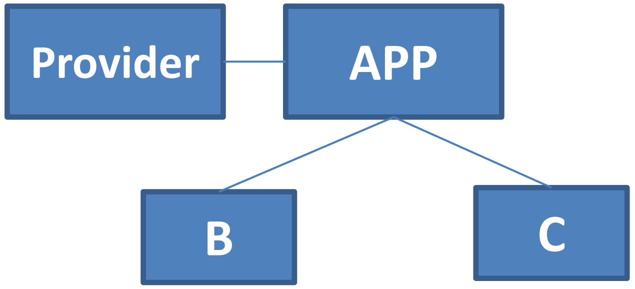


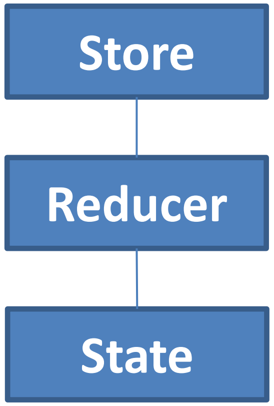


1. Create your App and it‘s child components
2. Create your Store, a Reducer and an initial State

2.6.4. Redux

3. Redux = One or multiple shared states managed by a store



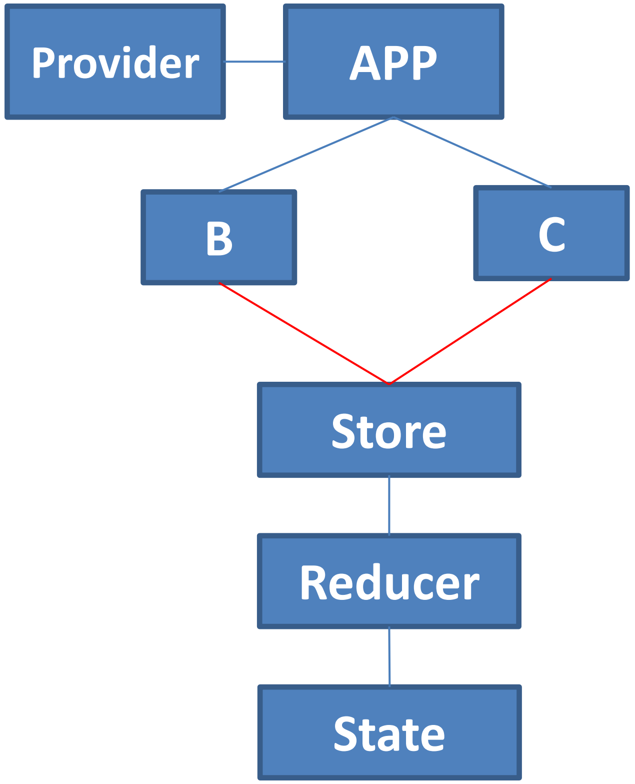


x=0, y=0

1. Create your App and it's child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component

2.6.4. Redux

3. Redux = One or multiple shared states managed by a store

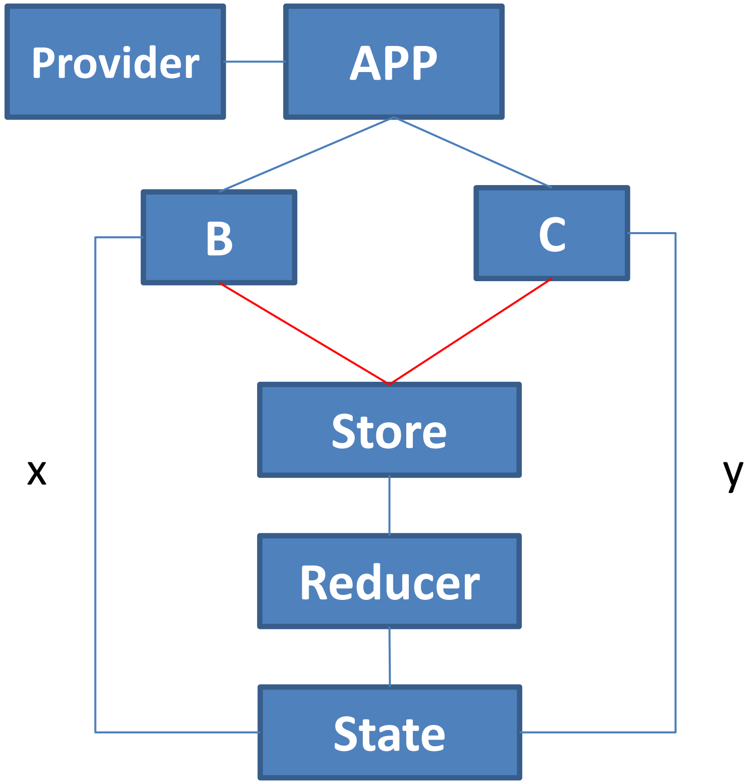


x=0, y=0

1. Create your App and it's child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store

2.6.4. Redux

3. Redux = One or multiple shared states managed by a store

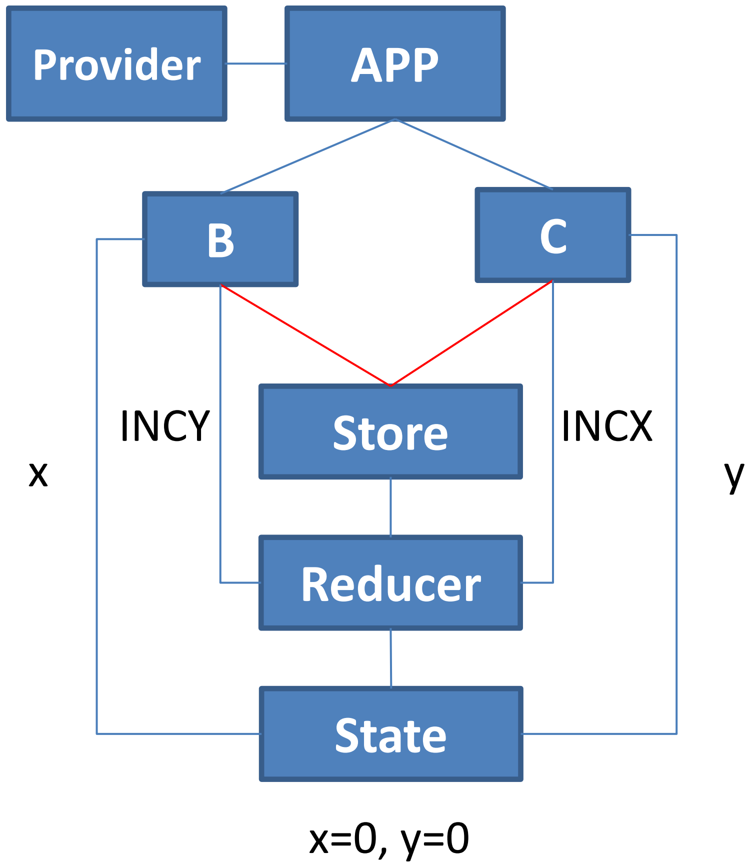


x=0, y=0

1. Create your App and it‘s child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store
5. Map the Reducer‘s state variables to the props B and C

2.6.4. Redux

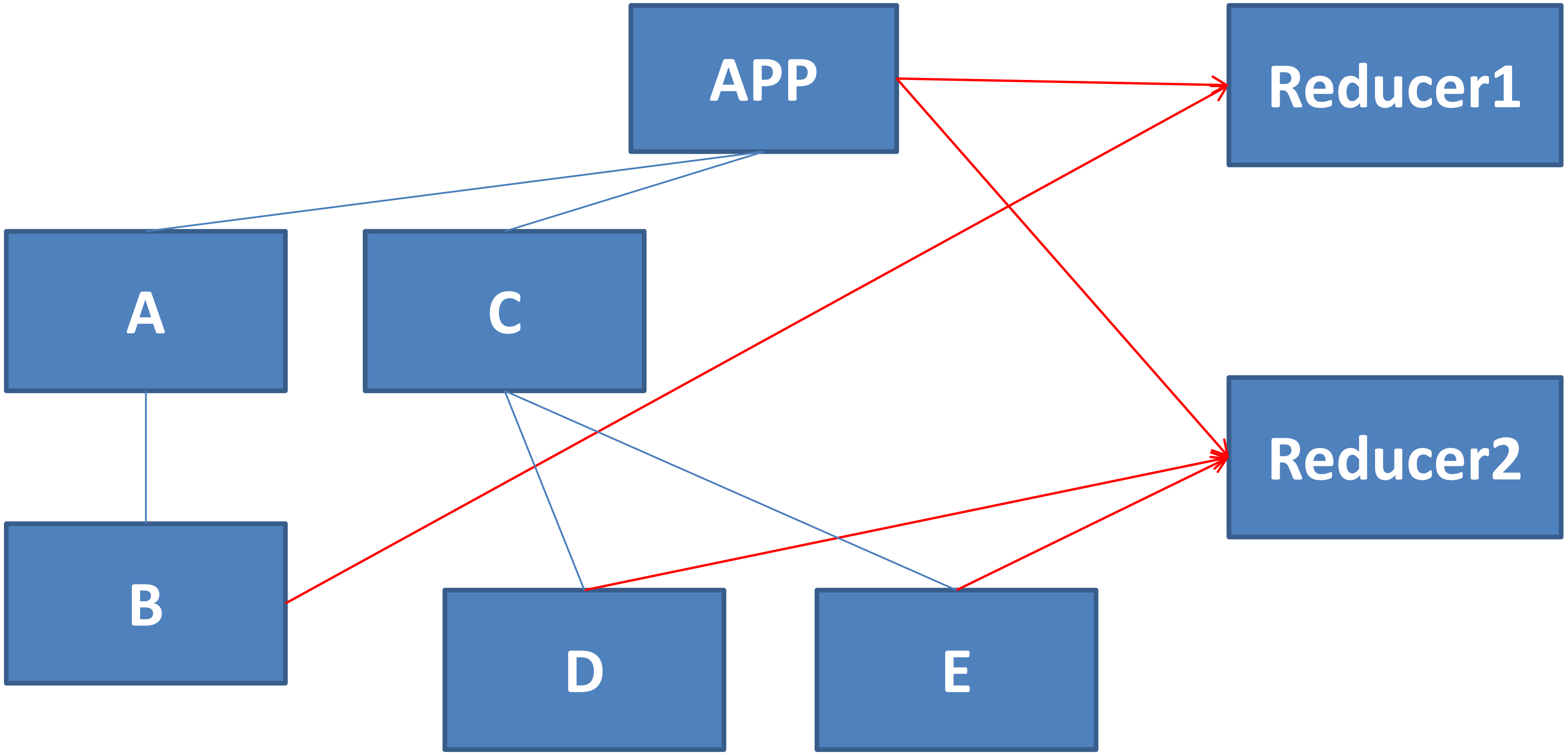
3. Redux = One or multiple shared states managed by a store



1. Create your App and it's child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store
5. Map the Reducer's state variables to the props B and C
6. Map the Reducer to props of B and C

2.6.4. Redux

1. Create a new web-app “task-2-6-4“
1. Implement the following component architecture using Redux.



2.6.4. Task

2. Reducer1 has one variable a. Reducer2 has two variables b = 0 and c = 1.
3. Component B has a button that generates a new randomstring with length 10 and saves it as Reducer1‘s a.
4. Component D has a button that sets Reducer2‘s b to b + 2.
5. Component E has a button that sets Reducer2‘s c to c + 2.
6. The App component shows all variables a, b and c.
7. The App component decides, that whenever b > 10 or c > 11, b will be reset to 0 and c to 1.

2.6.5. Redux Thunk

- Redux Thunk
  - The dispatchers waits for an AXIOS call to finish, then dispatches the messages to the reducer

-> This topic will be covered in 2.7.2.

Agenda – Part 2.7. AJAX

1. AXIOS
2. Redux Thunk
3. Localhost as Backend

2.7.1. AJAX with AXIOS

- AXIOS is a library that implements the functionality of AJAX using promises
- NodeJS Servers can be integrated into React Apps using a Proxy feature

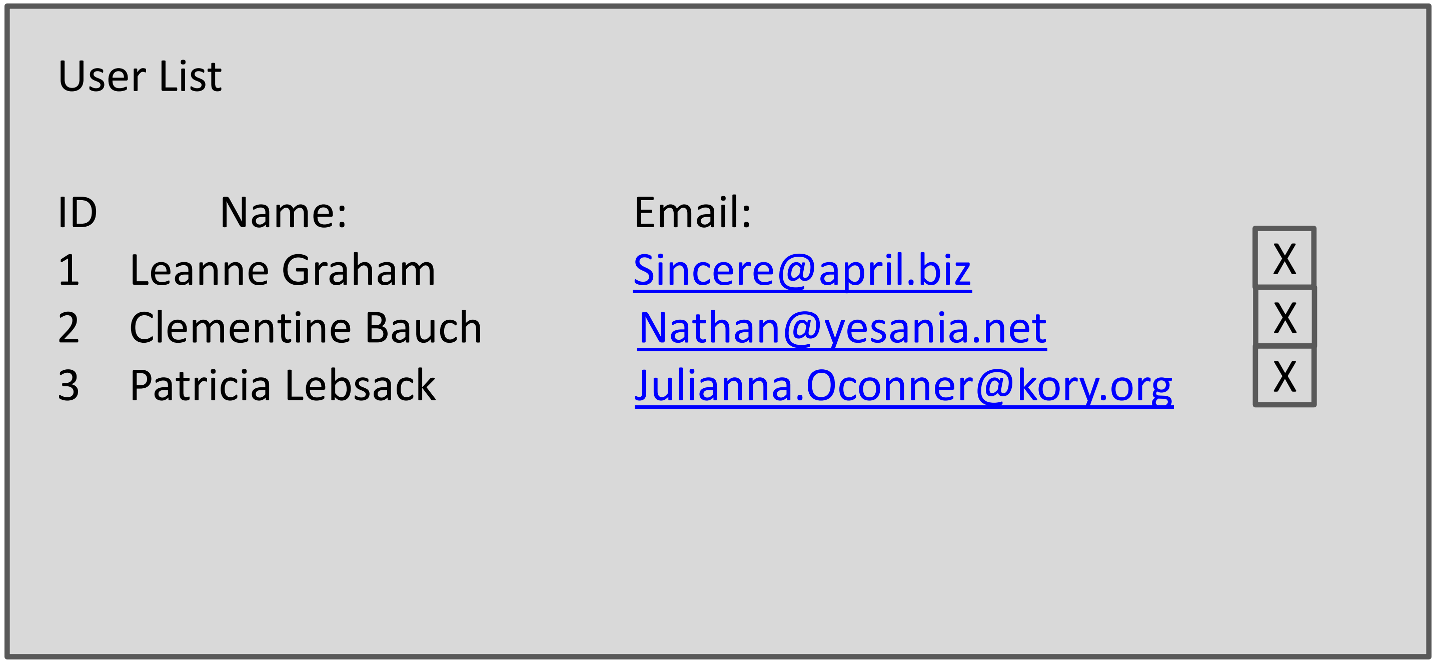
2.7.1. Task 1

1. Create a new React App “task-2-7-1“
2. This app shall load all user data from the following URL:

https://jsonplaceholder.typicode.com/users

The app should shows the id, the name and the email in a table. By clicking on the X next to the user, the user will be removed from the internal state.

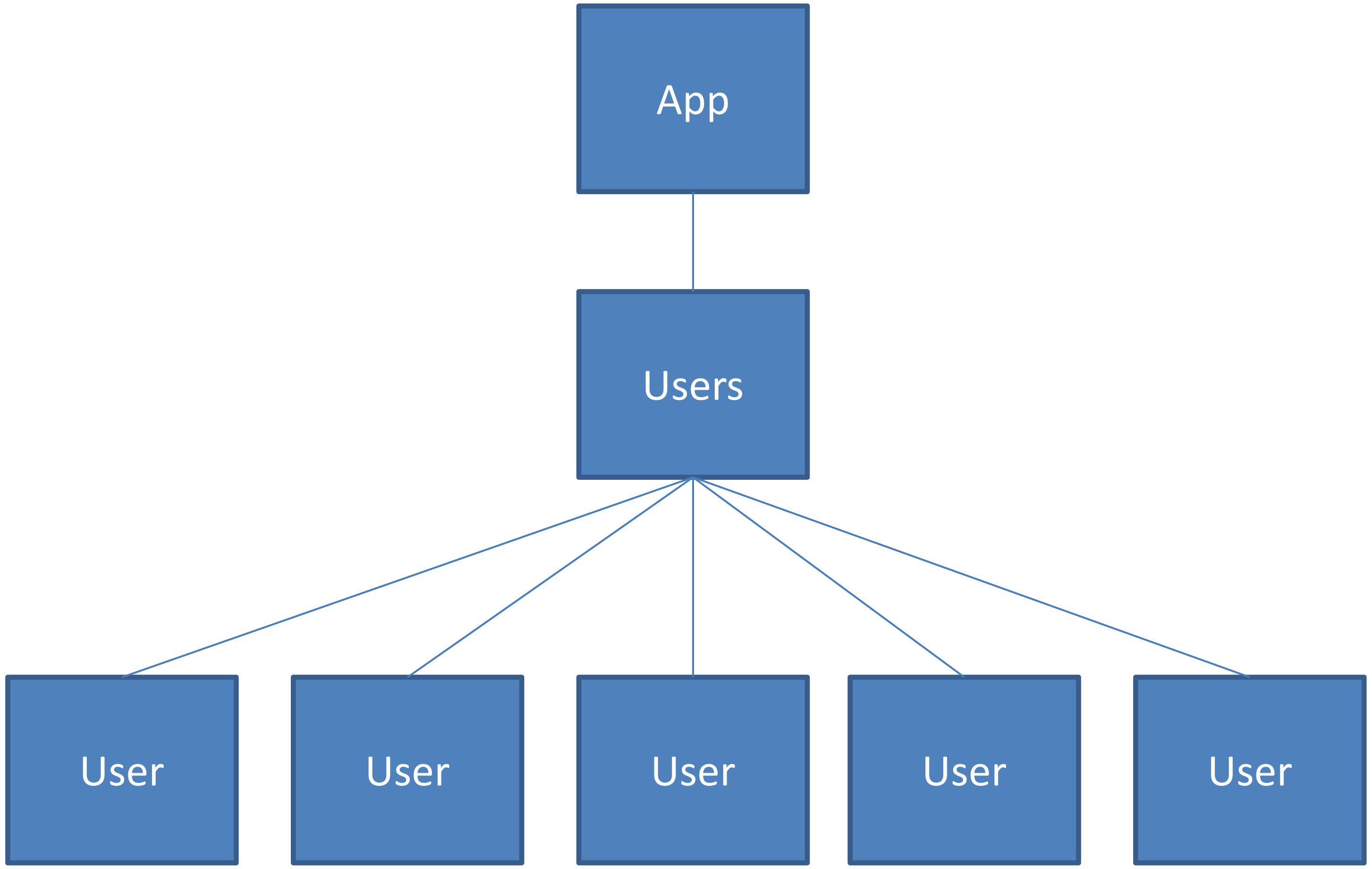
2.7.1. Task 1



User List

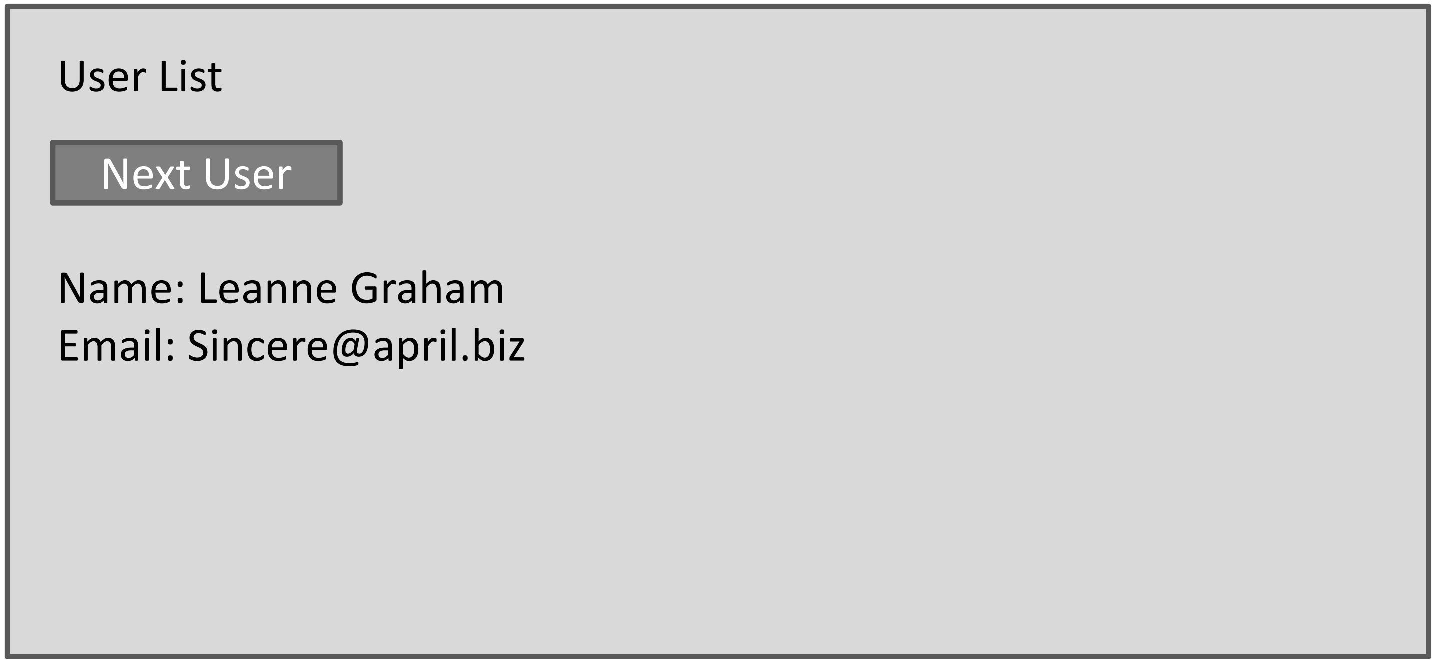
ID	Name:	Email:	
1	Leanne Graham	Sincere@april.biz	X
2	Clementine Bauch	Nathan@yesania.net	X
3	Patricia Lebsack	Julianna.Oconner@kory.org	X

2.7.1. Task 1



2.7.1. Task B (Difficult)

1. Create a React App task-2-7-1B
2. Inside the App component, create a button labeled “Next User“.
3. Create another component User and instantiate it in the App component underneath the “Next User“ button.


User List

Next User

Name: Leanne Graham
Email: Sincere@april.biz

2.7.1. Task 2 (Difficult)

4. Implement the following behaviour:

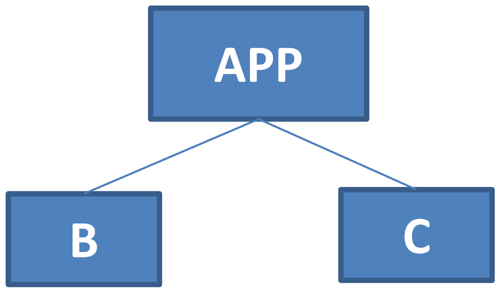
- When the App component initializes
  1) the App component sends the User component the id of the first user (id = 1) via props.
  2) The User component reads the id and loads the name and email via Axios from https://jsonplaceholder.typicode.com/users/1
  3) The User component shows the name and email of the first user (id=1).

2.7.1. Task 2 (Difficult)

5. Furthermore, implement the following behaviour:
   1) When the App‘s “Next User“ button is clicked, the App component internally updates its id from 1 to 2 (or in general from id to id + 1) and sends it down to the User component via props.
   2) The User component reads the id and loads the name and email via Axios from https://jsonplaceholder.typicode.com/users/{id}
   3) The User component shows the name and email of the next user
   4) Afther the 10th user, the id shall start with 1 again.

2.7.2. Redux-Thunk

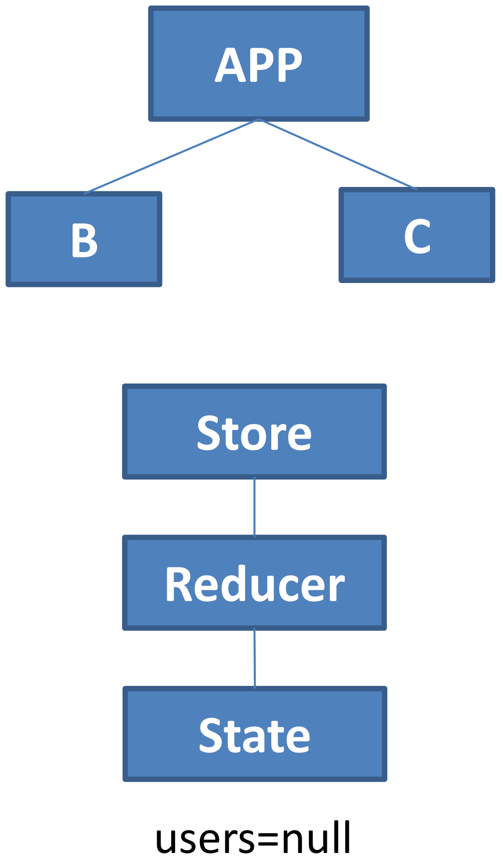
3. Redux = One or multiple shared states managed by a store



1. Create your App and it‘s child components

2.7.2. Redux-Thunk

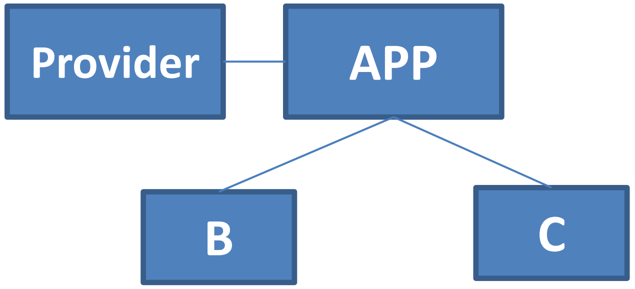
3. Redux = One or multiple shared states managed by a store

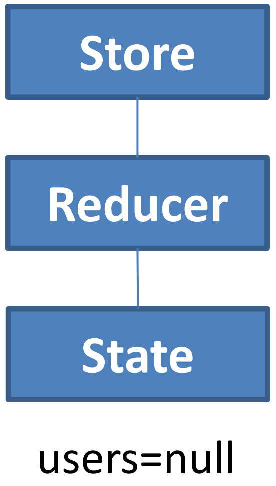


1. Create your App and it‘s child components
2. Create your Store, a Reducer and an initial State

2.7.2. Redux-Thunk

3. Redux = One or multiple shared states managed by a store

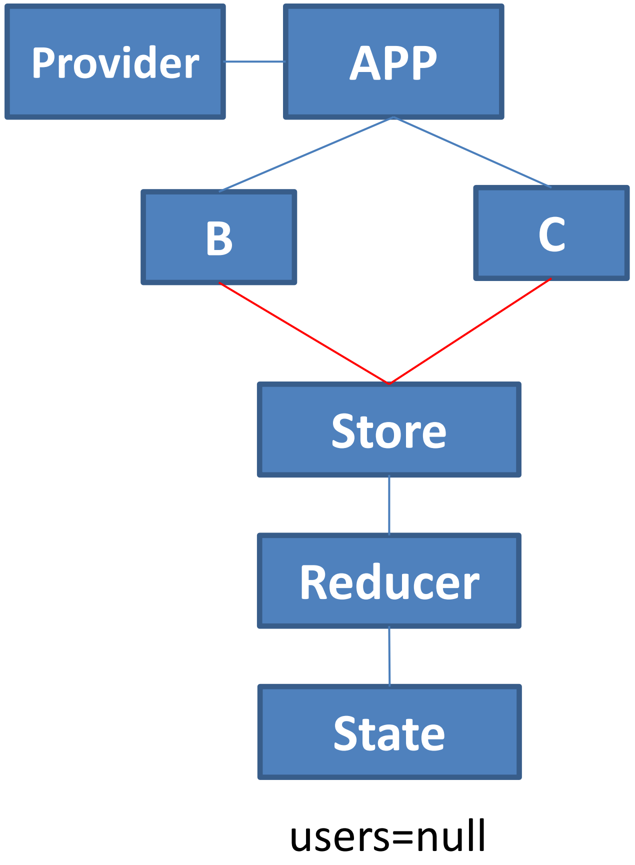




1. Create your App and it‘s child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component

2.7.2. Redux-Thunk

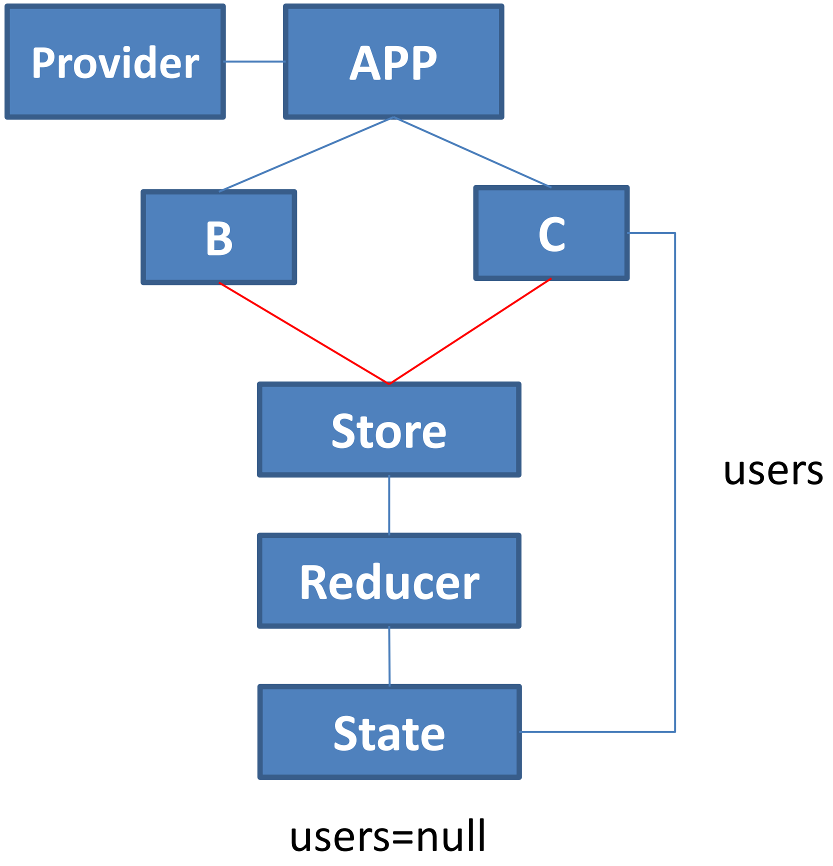
3. Redux = One or multiple shared states managed by a store



1. Create your App and it's child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store

2.7.2. Redux-Thunk

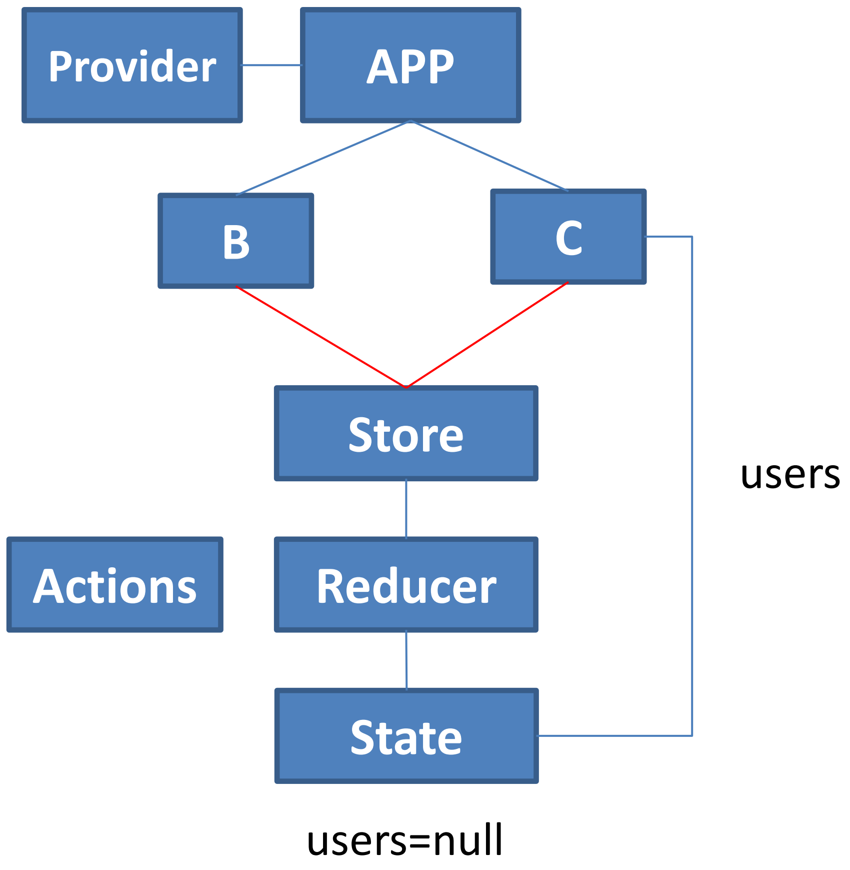
3. Redux = One or multiple shared states managed by a store



1. Create your App and it‘s child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store
5. Map the Reducer‘s state variables to the props of C

2.7.2. Redux-Thunk

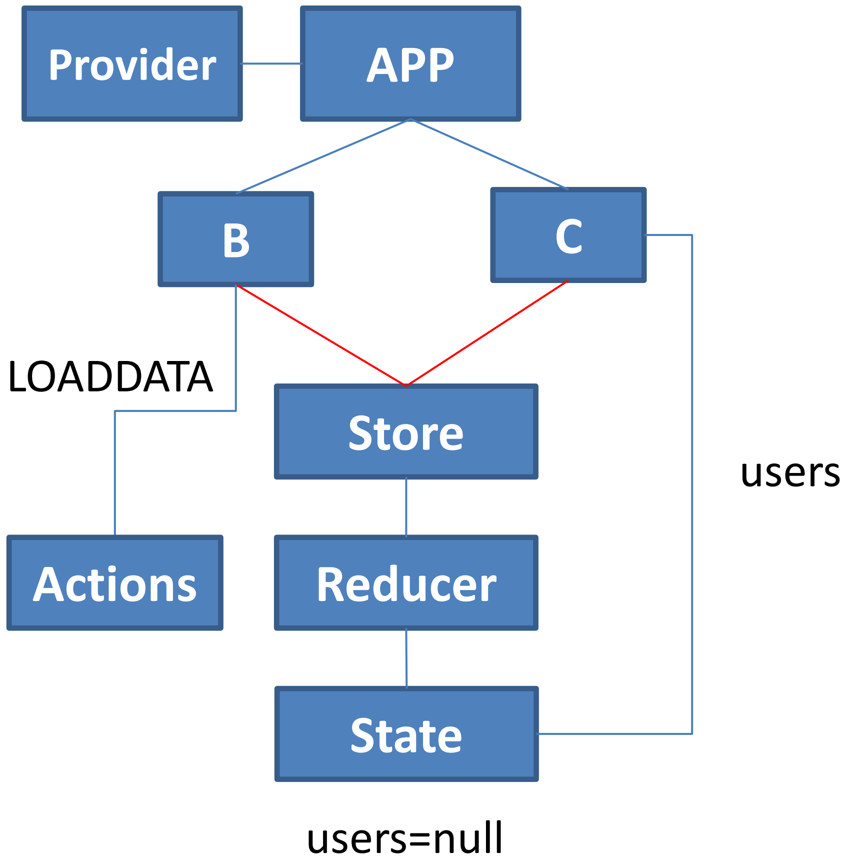
3. Redux = One or multiple shared states managed by a store



1. Create your App and it's child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store
5. Map the Reducer's state variables to the props of C
6. Create an asynchronous function that calls dispatch after it is done.

2.7.2. Redux-Thunk

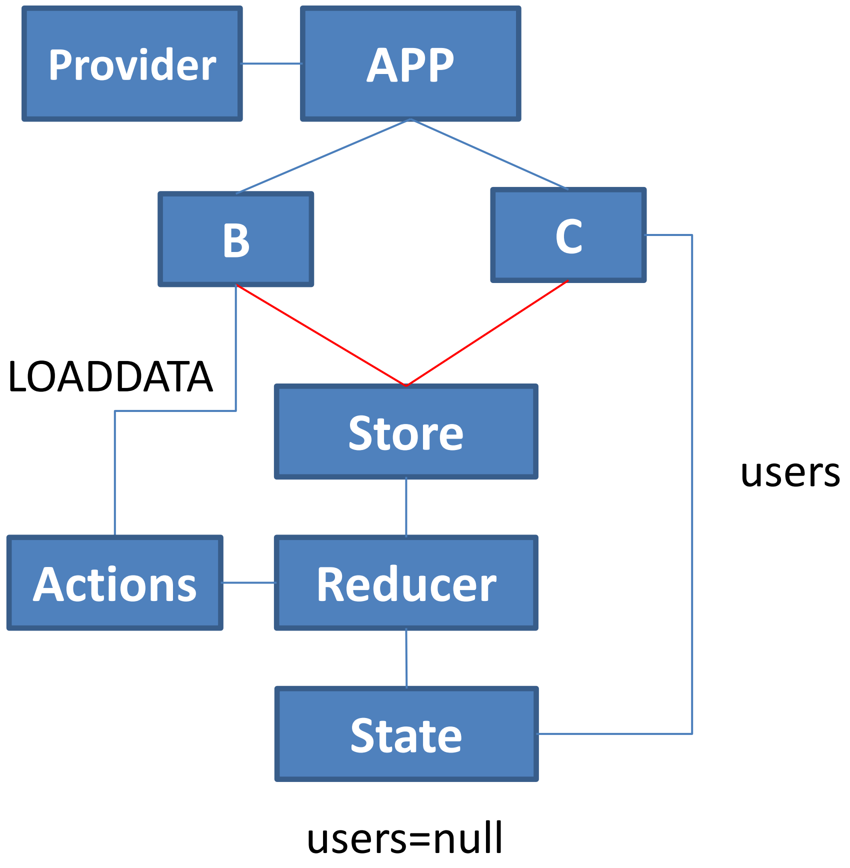
3. Redux = One or multiple shared states managed by a store



1. Create your App and it‘s child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store
5. Map the Reducer‘s state variables to the props of C
6. Create an asynchronous function that calls dispatch after it is done.
7. Import the 6.) into B

2.7.2. Redux-Thunk

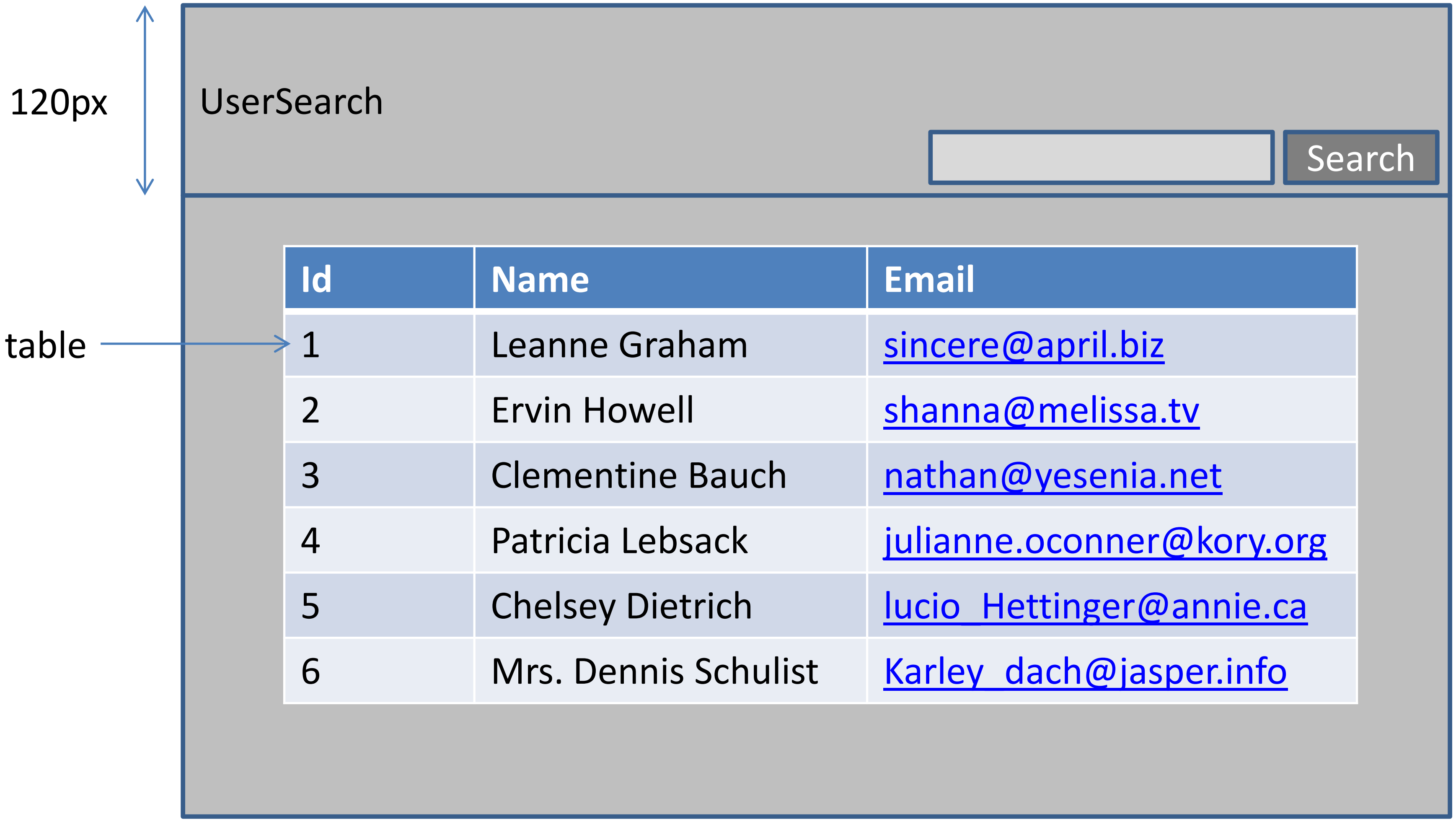
3. Redux = One or multiple shared states managed by a store



1. Create your App and it‘s child components
2. Create your Store, a Reducer and an initial State
3. Wrap your Provider around the App component
4. Connect B and C the store
5. Map the Reducer‘s state variables to the props of C
6. Create an asynchronous function that calls dispatch after it is done.
7. Import the 6.) into B
8. Connect B and the asynchronous functions with the Reducer

2.7.2. Task

1. Create a new web-app “task-2-7-2“.
2. Implement the following layout in the App-component.

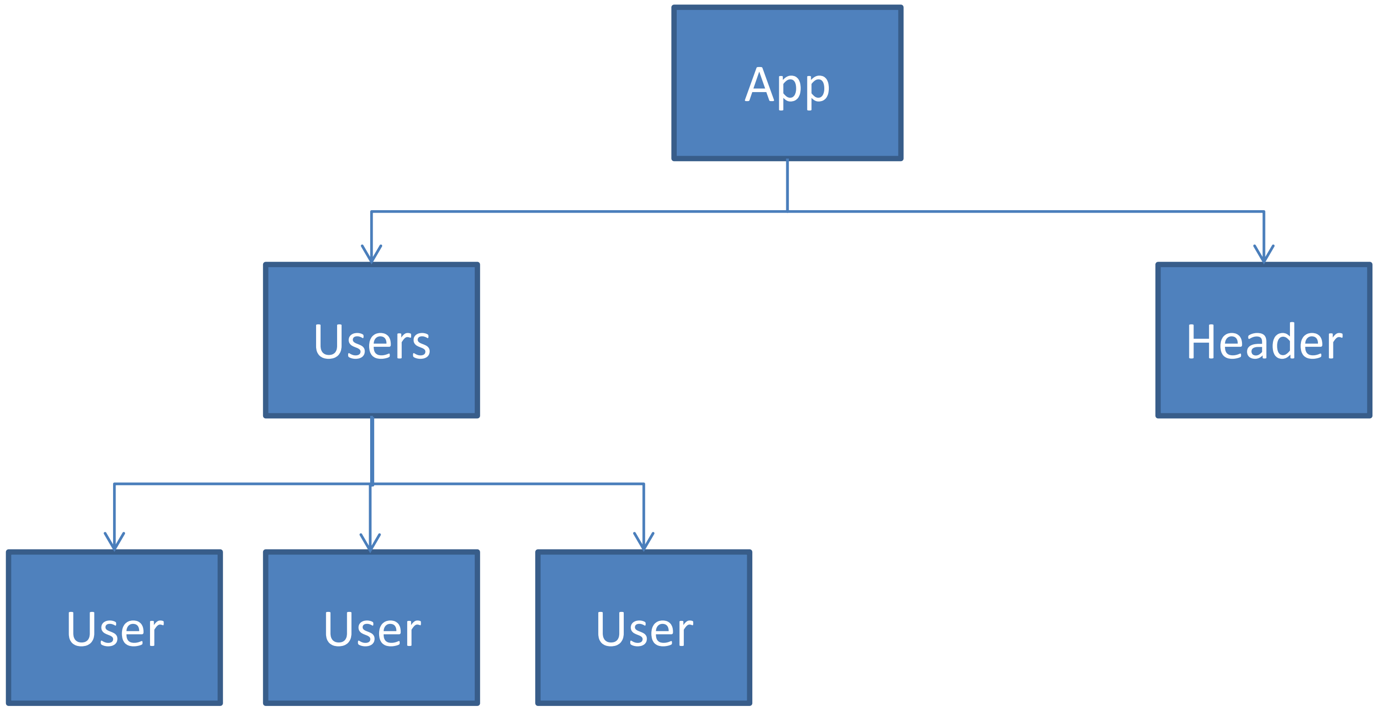


Id	Name	Email
1	Leanne Graham	sincere@april.biz
2	Ervin Howell	shanna@melissa.tv
3	Clementine Bauch	nathan@yesenia.net
4	Patricia Lebsack	julianne.oconner@kory.org
5	Chelsey Dietrich	lucio_Hettinger@annie.ca
6	Mrs. Dennis Schulist	Karley_dach@jasper.info

2.7.2. Task

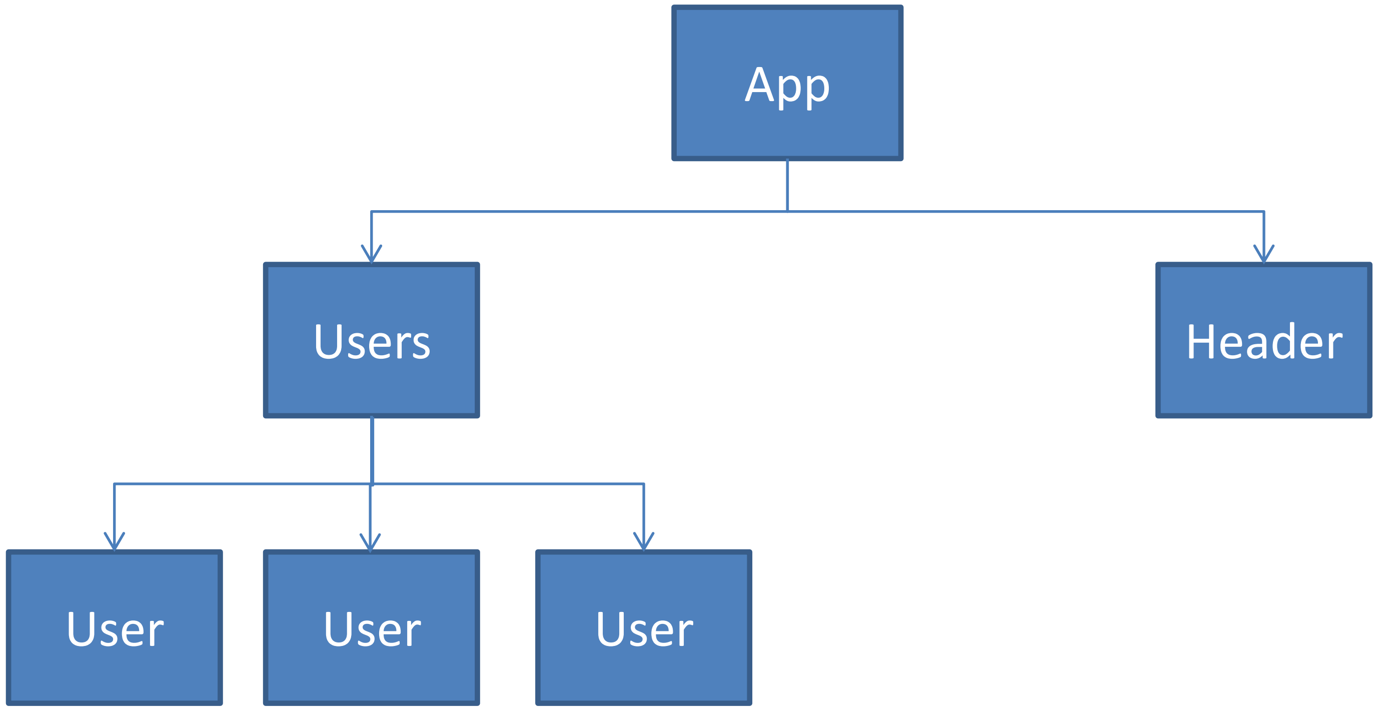
3. Source out the header-part in its own **stateful** Header-component.
4. Source out the users-part in its own **stateful** Users-component.
5. In the Users-component‘s state, create an array of users. For each user in that array, create a row in the table. Make each row be its own User-component.

Now, your component-architecture should look like this:



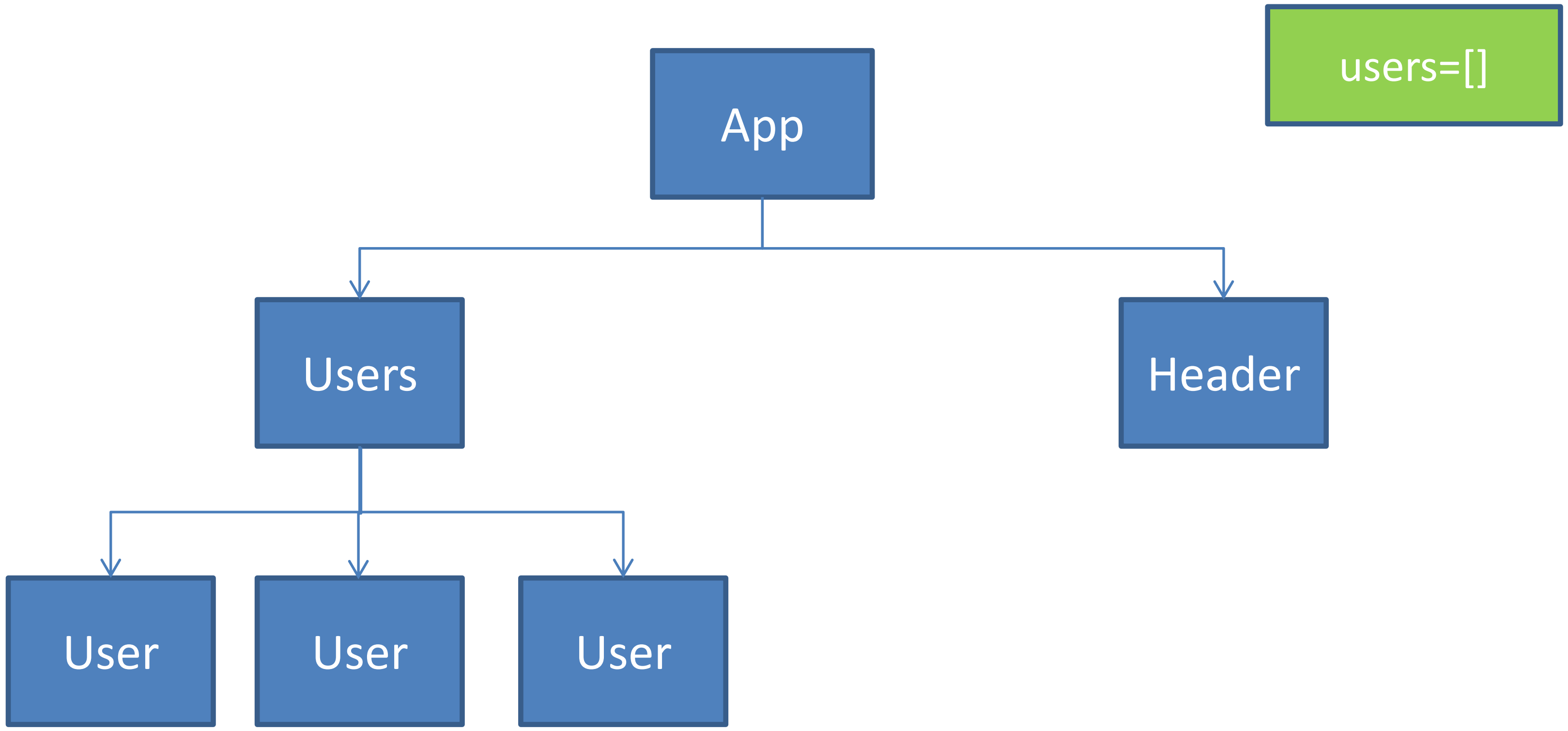
2.7.2. Task

6. When the Users-component mounts, use the fetch-API to load the user-data from https://jsonplaceholder.typicode.com/users and save them in the Users‘ state. (i.e. as an array)
7. Based on the user-Array, display the User-components.



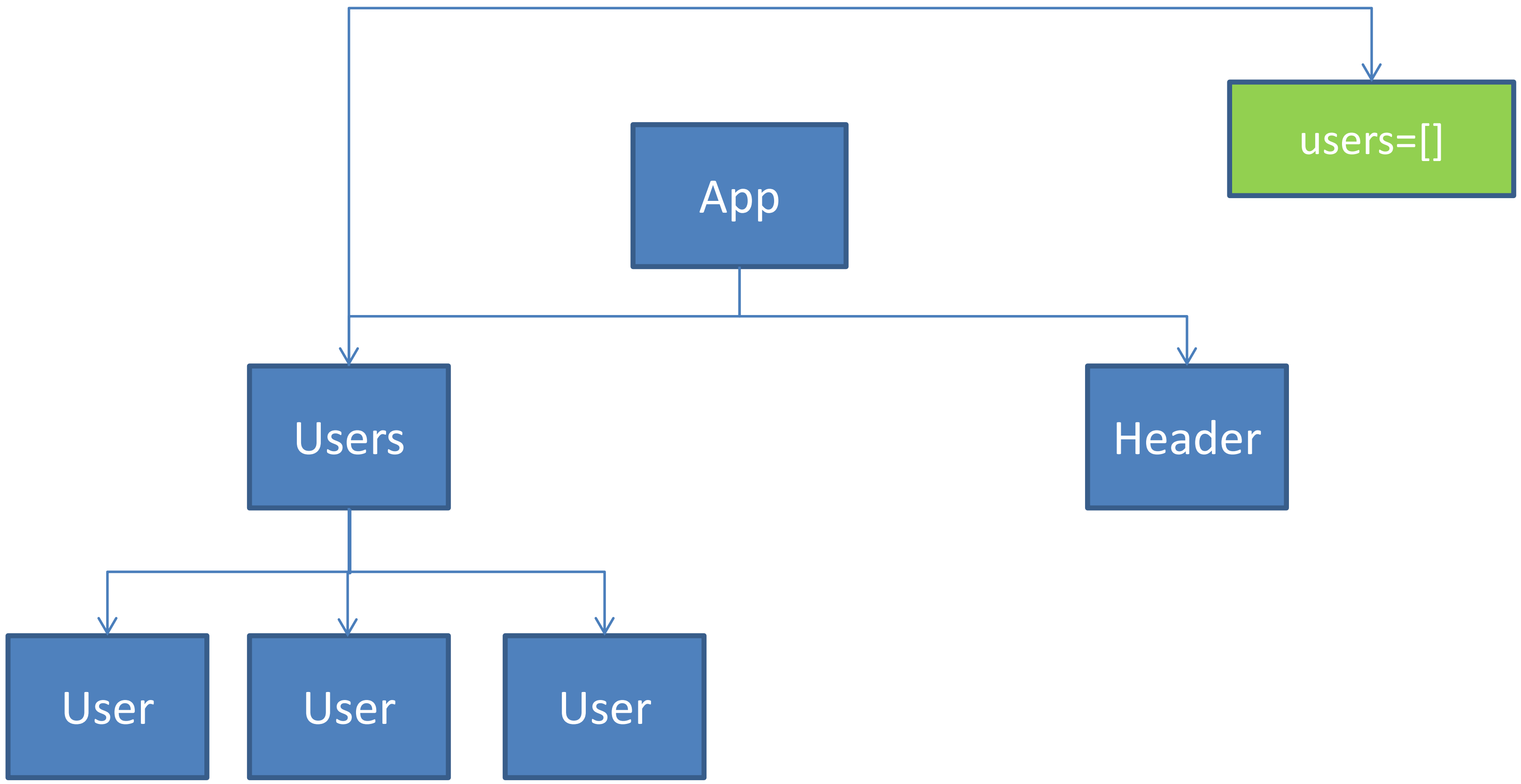
2.7.2. Task

8. Integrate Redux-Thunk in your web-app with a global state that initially consists of an empty array “users“.
9. Add one reducer that receives one message “SEARCHUSER“. When the receiver receives this message, it will override the global users-Array with the array in the payload.



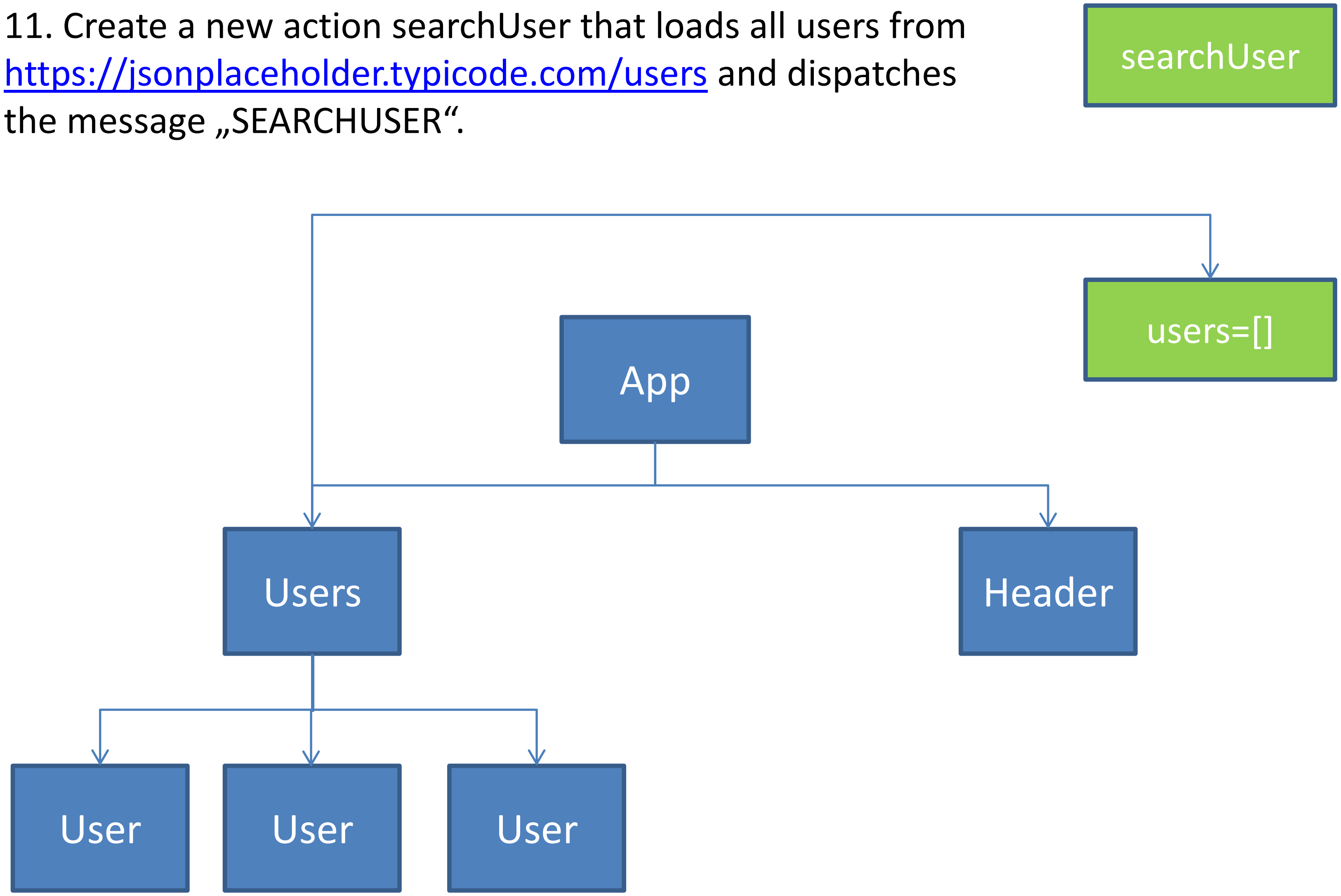
2.7.2. Task

10. Connect the Users-component to the store and replace the Users‘ local users array with the global users array.



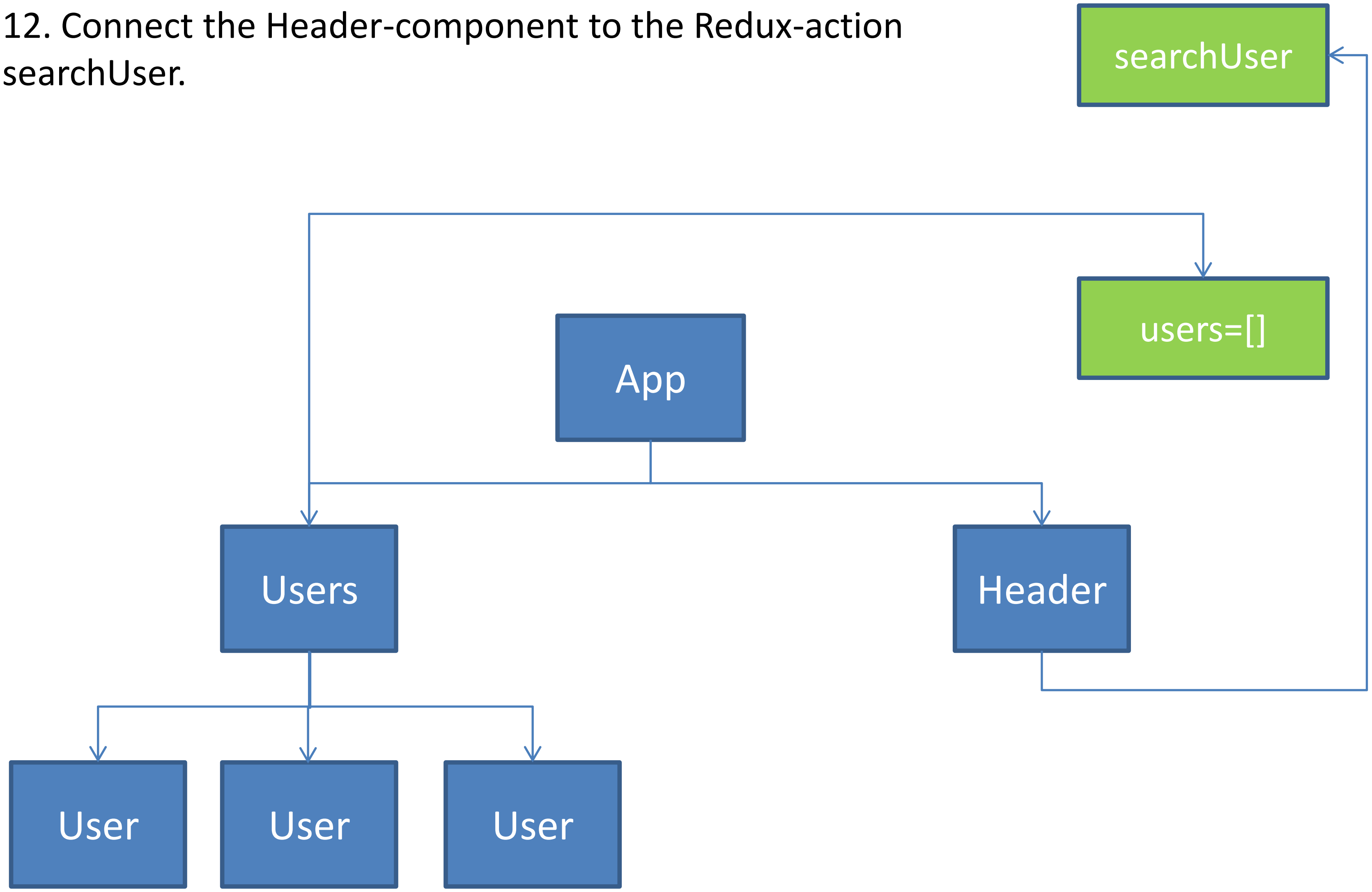
2.7.2. Task

11. Create a new action searchUser that loads all users from https://jsonplaceholder.typicode.com/users and dispatches the message „SEARCHUSER“.



2.7.2. Task

12. Connect the Header-component to the Redux-action searchUser.



2.7.2. Task

12. Implement the following behaviour: When the Header-component mounts, all users will be loaded.

13. Lastly, implement a search-function. I.e. If the search term is „ervin“, only one user will be found:

ID	**Name**	**Email**
1	Ervin Howell	Shanna@melissa.tv

Note: Also conside filtering the email-addresses for the search-term.

2.7.3. Localhost as Backend

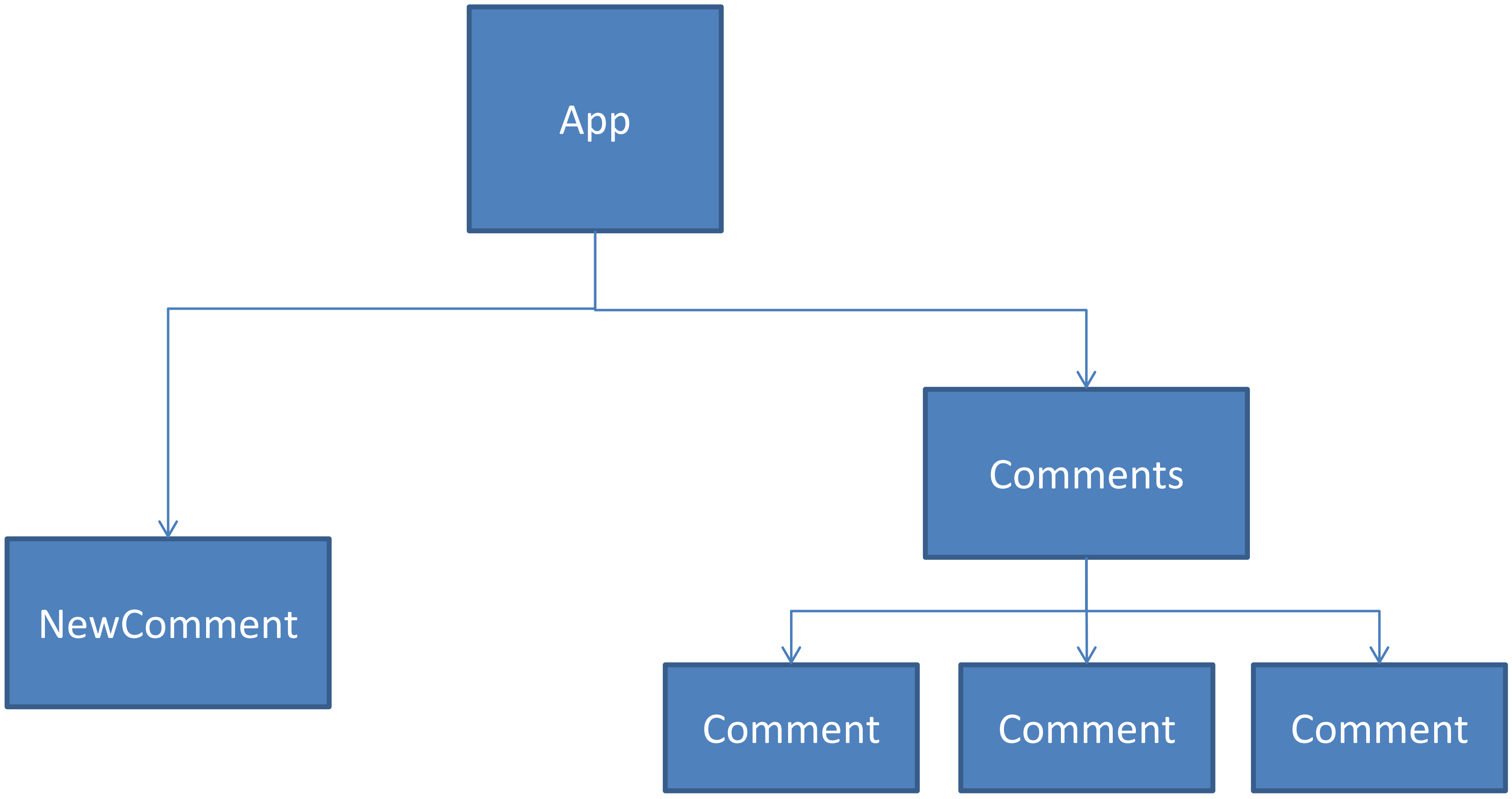
- “npm start“ starts a webserver that exposes the „/public“ – folder to HTTP clients
  - Chrome
  - Firefox
- The routes of that particular webserver are limited to
  - / - root
  - routes defined by our React Router
- The routes can be joined with the routes of a localhost backend server coded in
  - NodeJS
  - Java
  - PHP
  - ...
- Why?
  - Login
  - Signup
  - Protected Routes
  - ...

2.7.3. Task

1. Create a new react app “task-2-7-3“.
2. Create the two stateful components “NewComment“ and “Comments“ and instantiate them in the App component.
3. The NewComment component must contain of two input fields for name and text and a button labeled as „Create comment“.
4. Create a stateless component “Comment“. Each comment should be represented by its own Comment-component. Therefore, from the Comments-component, pass the props “name“ and “text“ down to each Comment-component. Create 3 instances of the Comment-component with the names “John“, “Bob“, “Mary“ and three texts “Hi whatsup“, “How are you?“ and “Good weather today!“
   – The Comment component must consist of three Divs for the name and the text. Inside the third Div, add a button with an “X“ inside.

2.7.3. Task

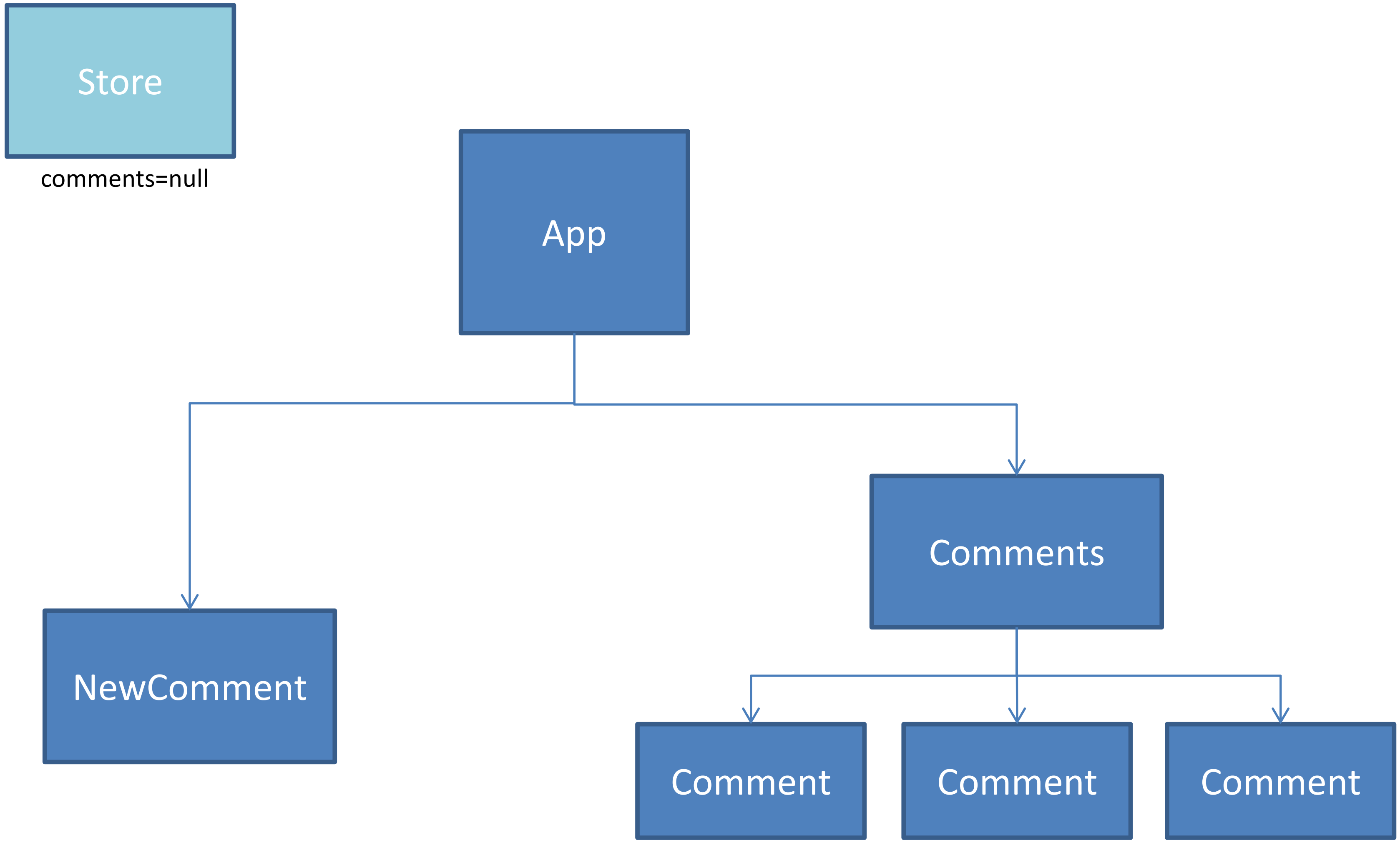
Your component structure should like this:



2.7.3. Task

5. Create a Redux store with one reducer and one state. The state contains of one variable “comments“ which is initially set to null.
6. Integrate the Redux-Thunk middleware. Do not define any actions yet. Just make Thunk work that the App can restart without any errors.

2.7.3. Task


Store
comments=null
App
NewComment
Comments
Comment
Comment
Comment

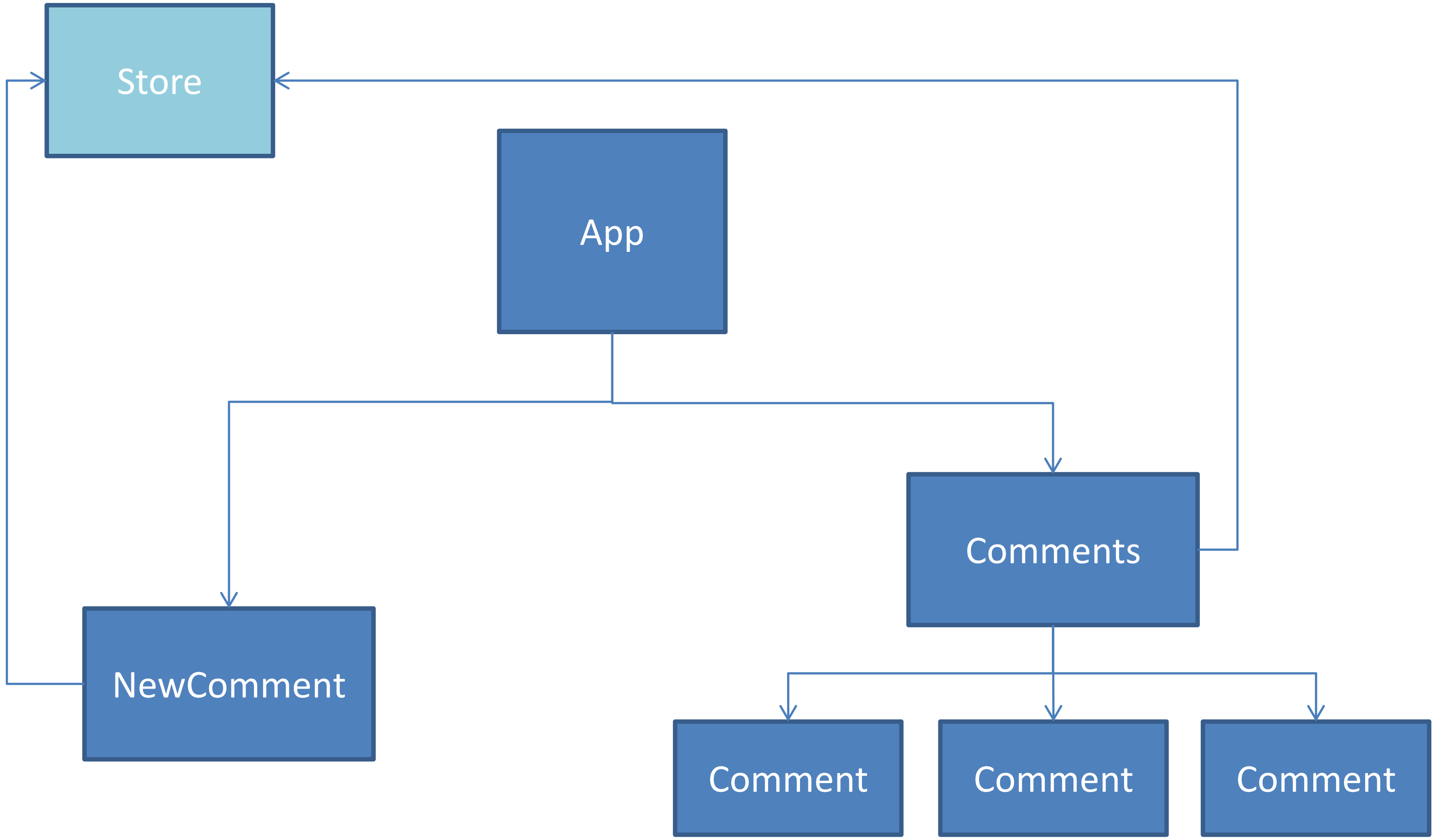
2.7.3. Task

7. For Thunk, create three asynchronous actions getComments, postComment and deleteComment.
   - getComments: sends an Axios GET request to http://localhost:3001/comments and updates the state‘s comments variable with the response
   - postComments: expect two parameters name and text and sends Axios POST request to http://localhost:3001/comments
     - The POST-body must consist of the name and the text, i.e. { name: “Paul“, comment: “Great weather!“ }
     - The response will be the sent POST-body plus an additional id generated by the server. This id must also be saved for each comment saved in the state‘s comments variable
   - deleteComment: sends an Axios DELETE request to http://localhost/comments/:id whereas :id is the id of the respective comment. If the comment was successfully deleted, the response will be { errorId: 0 }

2.7.3. Task

8. Connect the NewComment component‘s dispatcher with the Redux store. Make sure you have all of the three actions getComments, postComment and deleteComment available as action.
9. Connect the Comments component‘s props with the Redux store.

2.7.3. Task



2.7.3. Task

10. Implement the following behaviour:
    - When the Comments component mounts, all comments will be loaded via getComments
    - When the “Create comment“ button is clicked, a new comment based on the two input boxes will created
    - When the “X“-button is clicked, the comment will be removed.

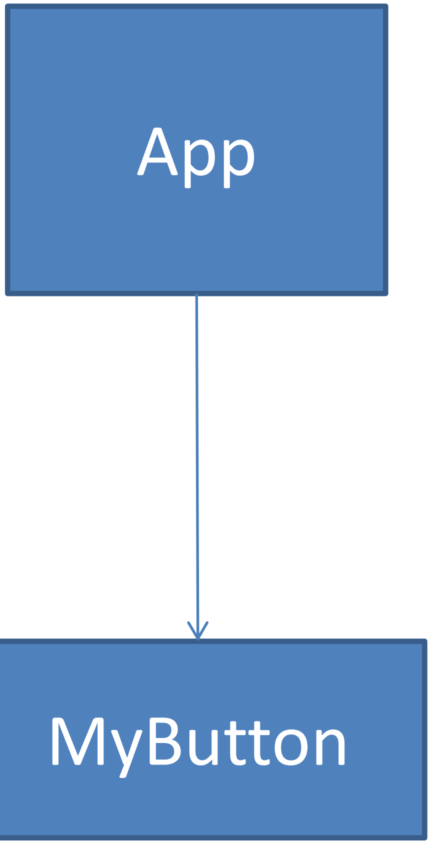
Agenda Part 2.8.
Higher Order Components

1. Definition
2. HalloWorld

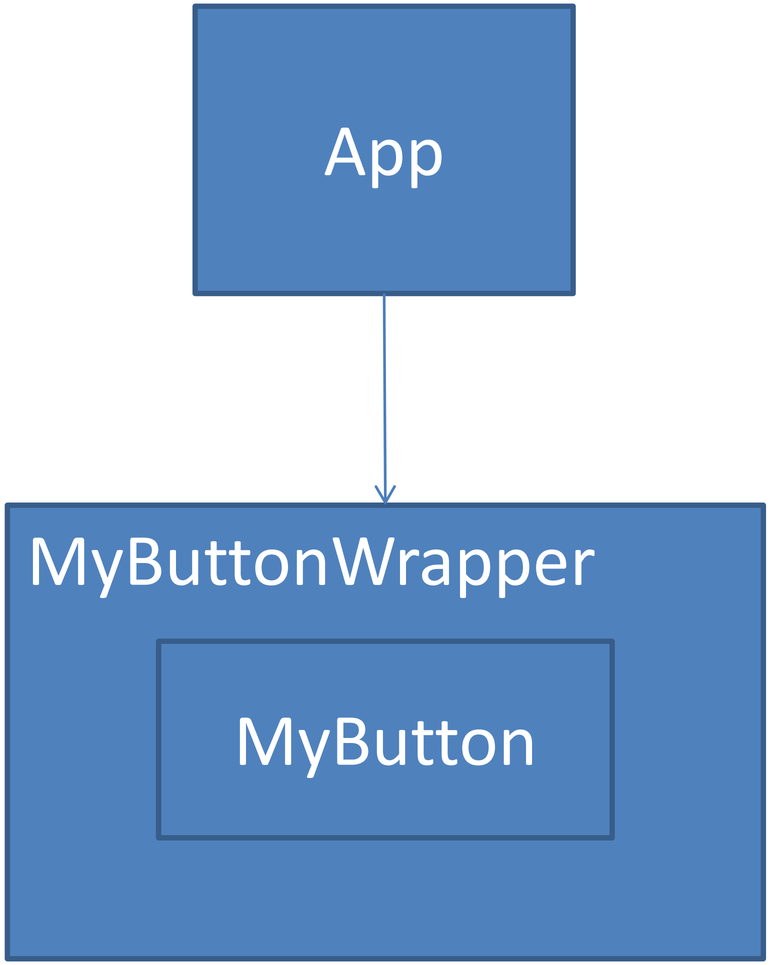
2.8.1. Definition

- Higher Order Components
  - Used for re-using component logic for different props
- Syntactially, they are the same as higher order functions

2.8.2. HalloWorld

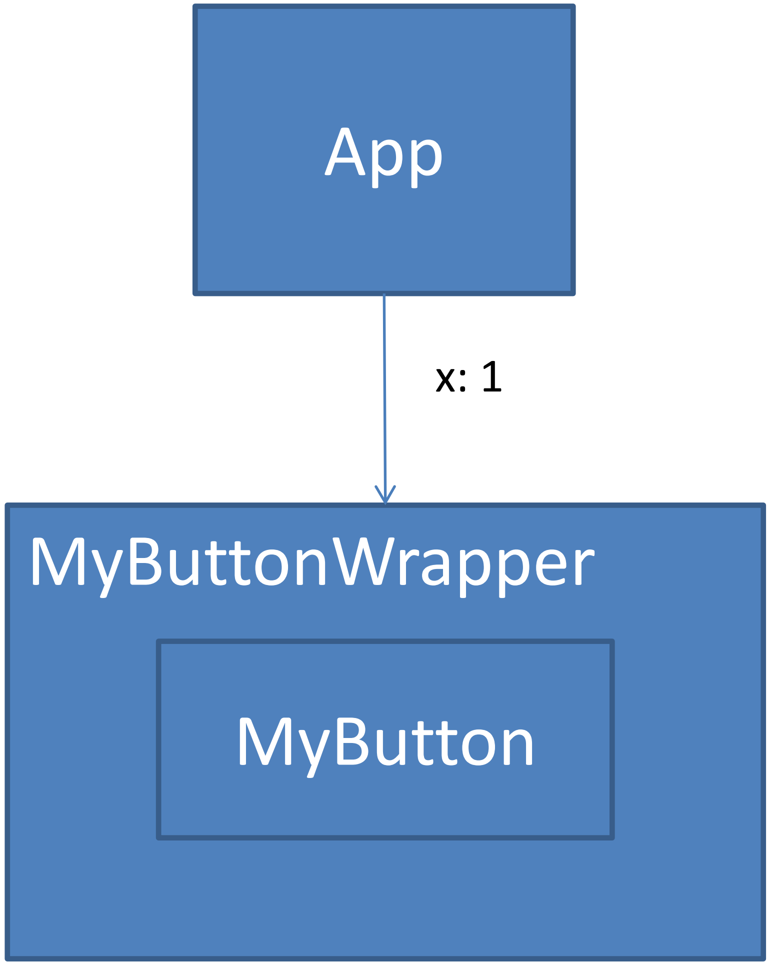


2.8.2. HalloWorld



MyButton will
be returned inside
MyButtonWrapper

2.8.2. HalloWorld



MyButton will
be returned **<u>inside</u>**
MyButtonWrapper,
which passes x:2 to
MyButton

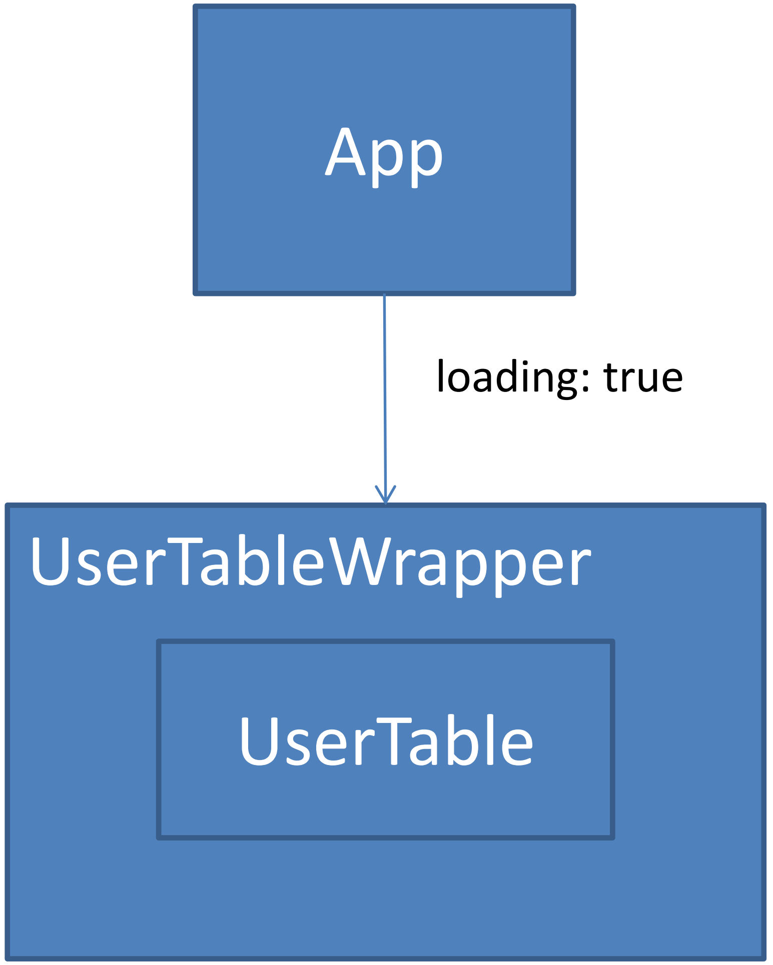
2.8.2. Task

1. Create a new React web app “task-2-8-2“
2. In the App‘s state, add two variables:
   1. “loading“ which is initally set to true
   2. “users“ which is an array and consists of the following user information
      - Id:1, username: ‘Bob‘, email: ‘bob@gmail.com‘
      - Id:2, username: ‘Pete‘, email: ‘pete@gmail.com‘
      - Id:3, username: ‘Sara, email: ‘sara@gmail.com‘
3. Implement the following behaviour: 3 seconds after the App-component has mounted, loading must be set to false.

2.8.2. Task

4. Create a new stateless component UserTable.
5. From App, pass state.users down to UserTable via props.
6. In UserTable, create a table that display the user information.
7. From App, also pass state.loading down to UserTable.
8. Create an Higher Order Component “UserTableWrapper“ that checks if loading is true or false. If true, then a div must be returned that say “Loading...“. Otherwise, show UserTable.

2.8.2. Task



Agenda Part 2.9.
Refs

1. Definition
2. HalloWorld

2.9.1.Definition

- Refs provide direct access to DOM elements created in the render method

2.9.2. HalloWorld

- When the web-app mounts, the focus will be set to an input box

2.9.2. Task

1. Create a new web-app “task-2-9-2“.
2. Go to YouTube and find a music video that you like. Click on „Share“ and copy the embed-code.
3. Paste the Embed-code in your App-component and create a variable “iframeRef“ that gives direct access to the iframe-DOM element.
4. Underneath the iframe, create two buttons “Play Song A“ and “Play Song B“.
5. Go to YouTube again and find another music video that you like. This time, only copy the URL-part of the embed-code.
6. Now, when “Play Song A“ is pressed, load first video, when “Play Song B“ is pressed, play the second video. Only use the useRef-hook for the implementation.

Agenda Part 3
Hooks

1. Motivation – Why React Hooks?
2. Definition
3. useState
4. useEffect
5. useContext
6. useReducer
7. useContext + useReducer
8. useMemo
9. useCallback
10. useRef
11. Custom Hooks

3.1. Motivation – Why React Hooks?

- What is not so great using Class Components
  - Reusing Logic between multiple Components
    - HOC and Render-Props lead to „Wrapper-Hell“
    - Each component returns JSX. Visual elements and logic are always linked to each other.
  - Giant Components
    - 1000 LOC components have code split accross multiple lifecycle methods
    - Component Lifecycle sprinkles throughout the application: if component A renders, its child B renders as well.
  - Confusing classes
    - Hard to minify -> Performance
    - Hot-Reloading not performant

3.2. Definition

- A **React Hook** is a function that enables using **React Features** in a functional component
- A **React Feature** might be:
  - State
  - Component Lifecycle
  - Context API
  - Shareable Non-JSX logic

3.2. Definition

- How does React know which useState-variable corresponds to which useState-call?
  - React relies on the order of these calls
  - It cannot be called inside a condition, it has to be on the top-level of the component

3.3. useState

- The useState-Hook adds a **state** to functional components
- In classes the state is always an object
- useState can be used with
  - Primitives
  - Objects
  - Arrays
- useState hook returns an array with 2 elements
  - The first element is the current value of the state
  - The second element is a state setter function

Primitives

3.3.1. useState

The setter method returned by useState, in this case **setCount** asynchronously enqueues updates of the state between each re-render.

Iteration **1**: **You** say: „Update count. Set it to count + 1“
React says: „Okay, count is 0. Therefore, we set it to 0 + 1 = 1“

setCount(count + 1)

Primitives

3.3. useState

The setter method returned by useState, in this case **setCount** asynchronously enqueues updates of the state between each re-render.

Iteration **2**: **You** say: „Update count. Set it to count + 1“
React says: „Okay, count is 0. Therefore, we set it to 0 + 1 = 1“

```
setCount(count + 1)
```

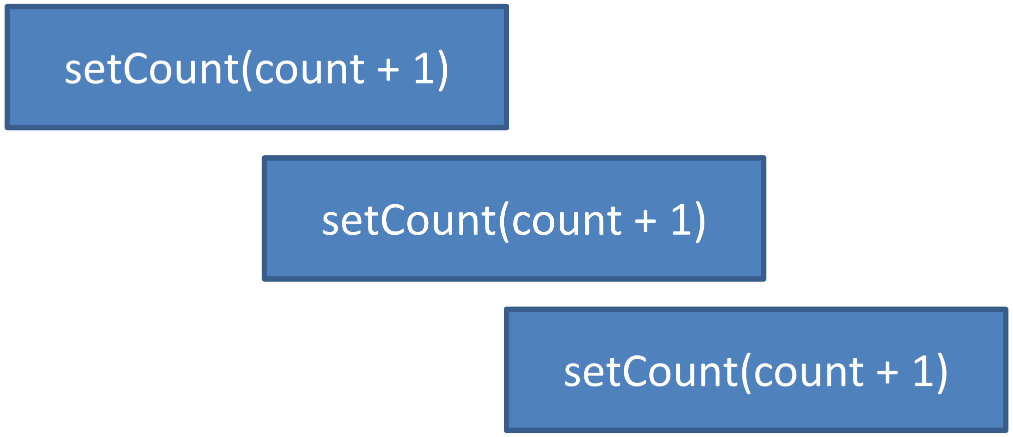
```
setCount(count + 1)
```

Primitives

3.3. useState

The setter method returned by useState, in this case **setCount** asynchronously enqueues updates of the state between each re-render.

Iteration **3**: **You** say: „Update count. Set it to count + 1“
React says: „Okay, count is 0. Therefore, we set it to 0 + 1 = 1“



```
setCount(count + 1)
setCount(count + 1)
setCount(count + 1)
```

Primitives

3.3. useState

Each setter saw the same value for count, which was 0.
0 + 1 = 1

setCount(count + 1)

setCount(count + 1)

setCount(count + 1)

Primitives

3.3. useState

If we pass the setter-Method a function, it will be called by React with the previous value of that setter's variable **after** each rendering is done.

```
setCount( prevCount => prevCount + 1)
```

React calls the passed function f(0) and sets count to 0 + 1

Primitives

3.3. useState

If we pass the setter-Method a function, it will be
called by React with the previous value of that setter‘s variable
after each rendering is done.

```
setCount( prevCount => prevCount + 1)
```

React calls the passed function f(0) and sets count to 0 + 1

```
setCount( prevCount => prevCount + 1)
```

React calls the passed function f(1) and sets count to 1 + 1

Primitives

3.3. useState

If we pass the setter-Method a function, it will be
called by React with the previous value of that setter‘s variable
after each rendering is done.

`setCount( prevCount => prevCount + 1)`	React calls the passed function f(0) and sets count to 0 + 1
`setCount( prevCount => prevCount + 1)`	React calls the passed function f(1) and sets count to 1 + 1
`setCount( prevCount => prevCount + 1)`	React calls the passed function f(2) and sets count to 2 + 1

<u>Objects</u>

3.3. useState

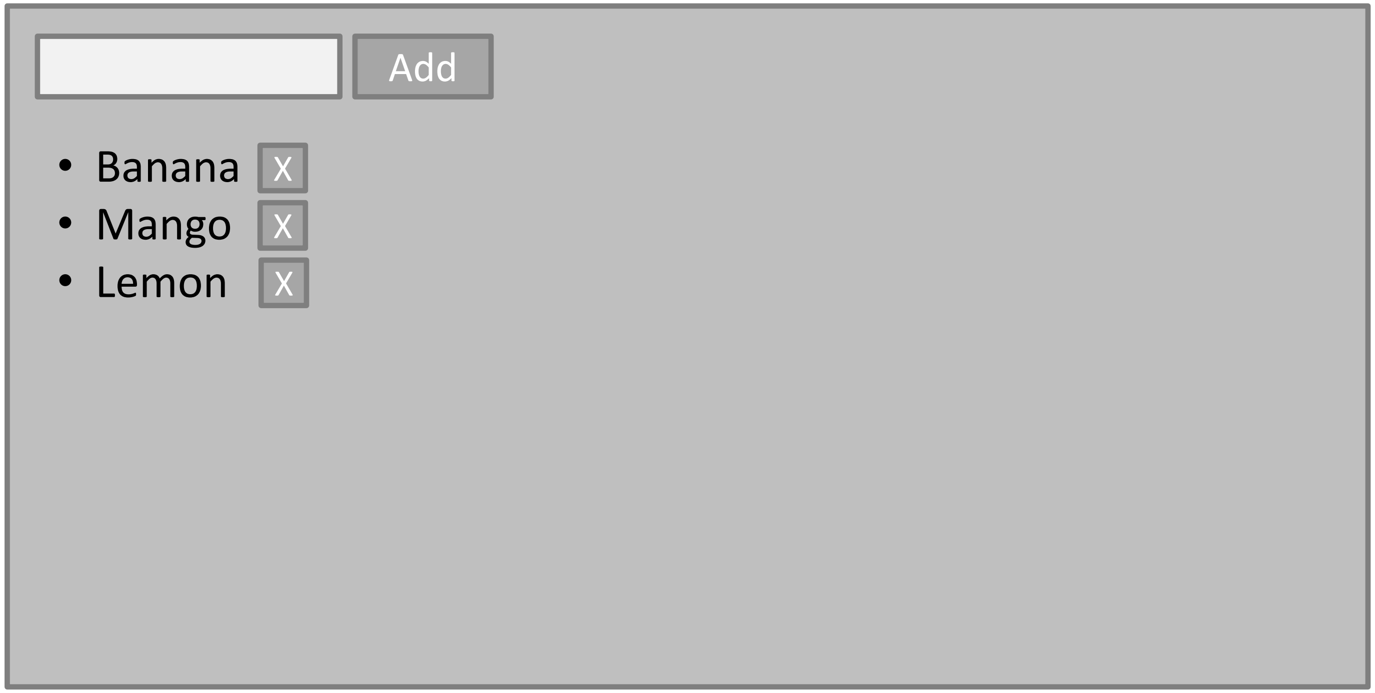
- useState can also save **<u>objects</u>**
- With the setter-methods, the saved object will be entirely overwritten
  – If only one **<u>key/value-pair</u>** is changed and the other **<u>key/value-pairs</u>** of the object need to remain, the spread-operator is very useful

3.3. useState

- useState can also save **arrays**
- With the setter-methods, the saved array will be entirely overwritten
  - If only one **index/value-pair** is changed and the other **index/value-pair** of the array need to remain, the spread-operator is very useful

3.3. Task

1. Create a new web-app „task-3-3“.
2. Create a layout like this: with an input field, a button next to it and underneath it, a list of 3 fruits: Banana, Mango & Lemon. Furthermore, Add an “X“-button to each list-item.



3.3. Task

3. When the user clicks on „Add“, the new fruit must be added to the fruit list, consider using an id for each fruit with a random string length of 5.
4. Now, when the User clicks on an „X“, the selected fruit will be removed.
5. If the list is empty, a header must be shown „No fruits here, yet“.

3.4. useEffect

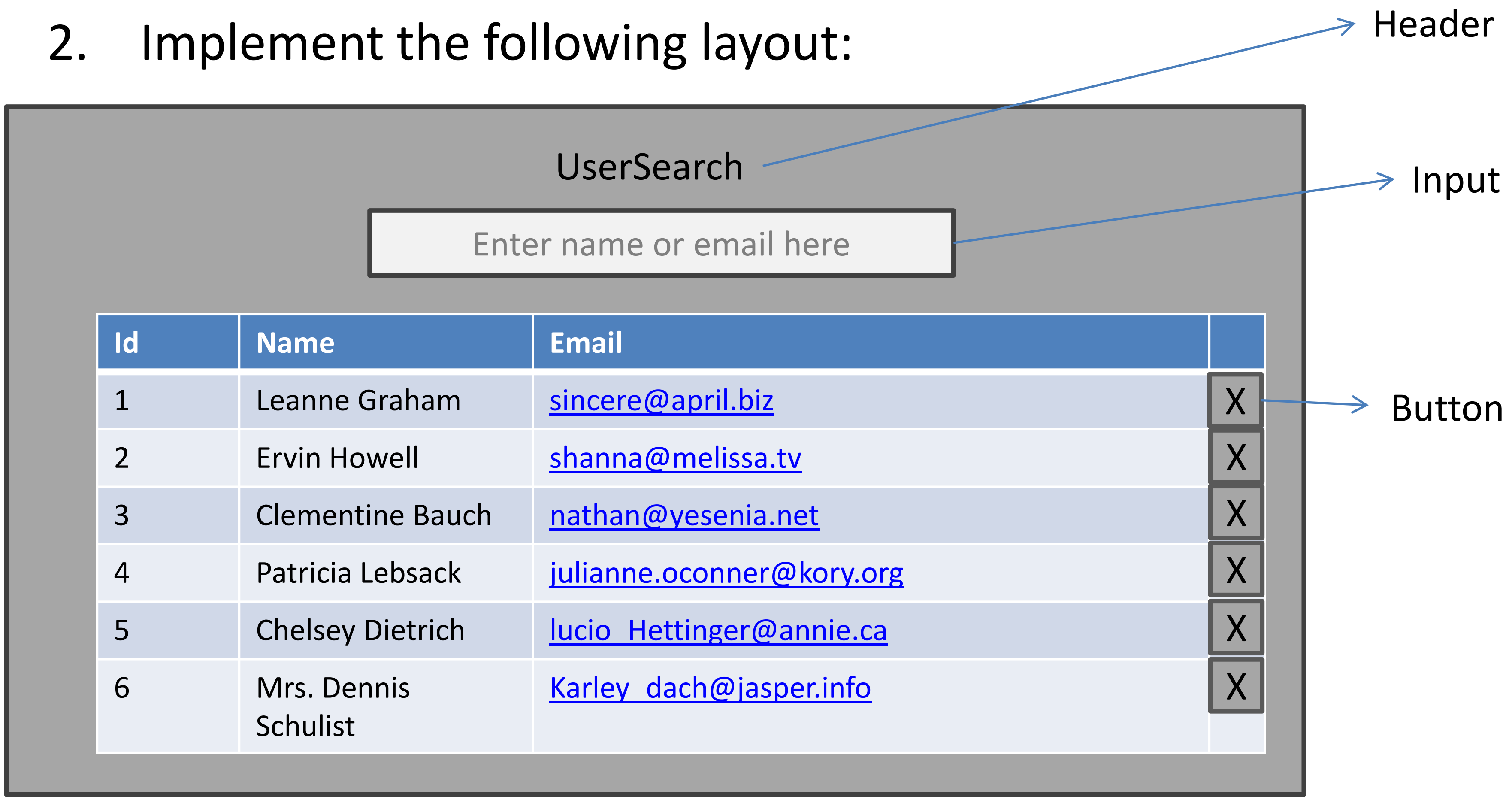
- The useEffect-Hook lets you call functions when
  1. Any state-variable changes
  2. One or multiple state-variable change
  3. After the component has mounted
  4. Before the component will unmount

3.4. useEffect

- Similar to Class Components,
  in Hooks, data-loading with fetch is done when ...
  1. the component mounts -> data loads initially
  2. a state-variable changes -> data re-loads again

3.4. Task

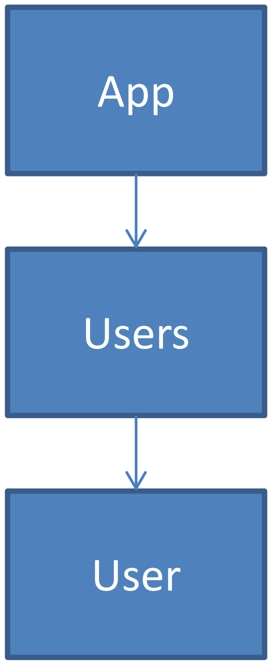
1. Create a new web-app „task-3-4“ with one functional component App.
2. Implement the following layout:



Id	Name	Email	
1	Leanne Graham	sincere@april.biz	X
2	Ervin Howell	shanna@melissa.tv	X
3	Clementine Bauch	nathan@yesenia.net	X
4	Patricia Lebsack	julianne.oconner@kory.org	X
5	Chelsey Dietrich	lucio_Hettinger@annie.ca	X
6	Mrs. Dennis Schulist	Karley_dach@jasper.info	X

3.4. Task

3. Source the User-Table out in its own Users-component and each row of the User-Table in its own User-component. Your component-structure should look like this:



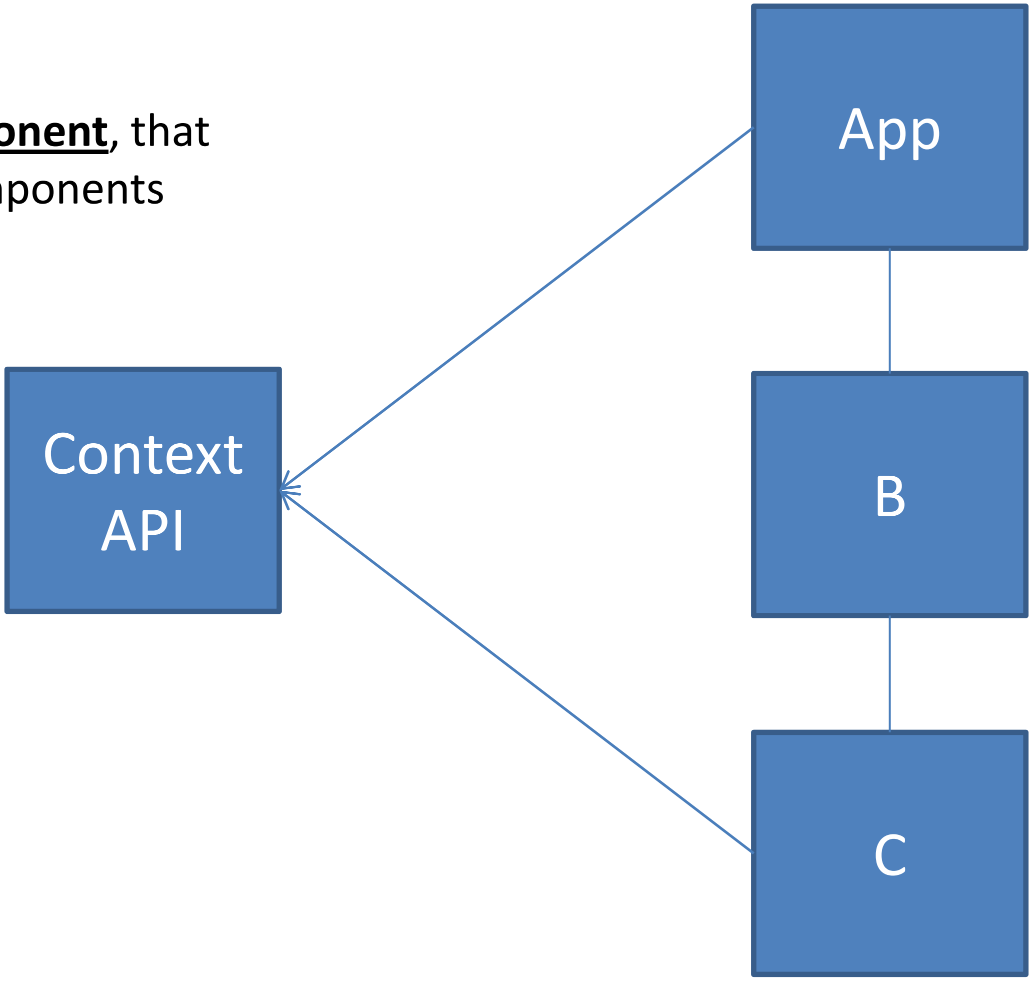
3.4. Task

4. In the App-component, define a new state variable “searchPhrase“.
5. Whenever the textbox in the App changes, the searchPhrase needs to be updated.
6. Pass the searchPhrase as prop down to the Users-component.
7. In the Users-component, create a new state-variable users that initially is an empty array.
8. When the Users-component mounts, load the users-data from https://jsonplaceholder.typicode.com/users and show the data in the table.
9. Lastly, filter the searchResults by the searchPhrase in the App-component.

3.5. useContext

Repetition of

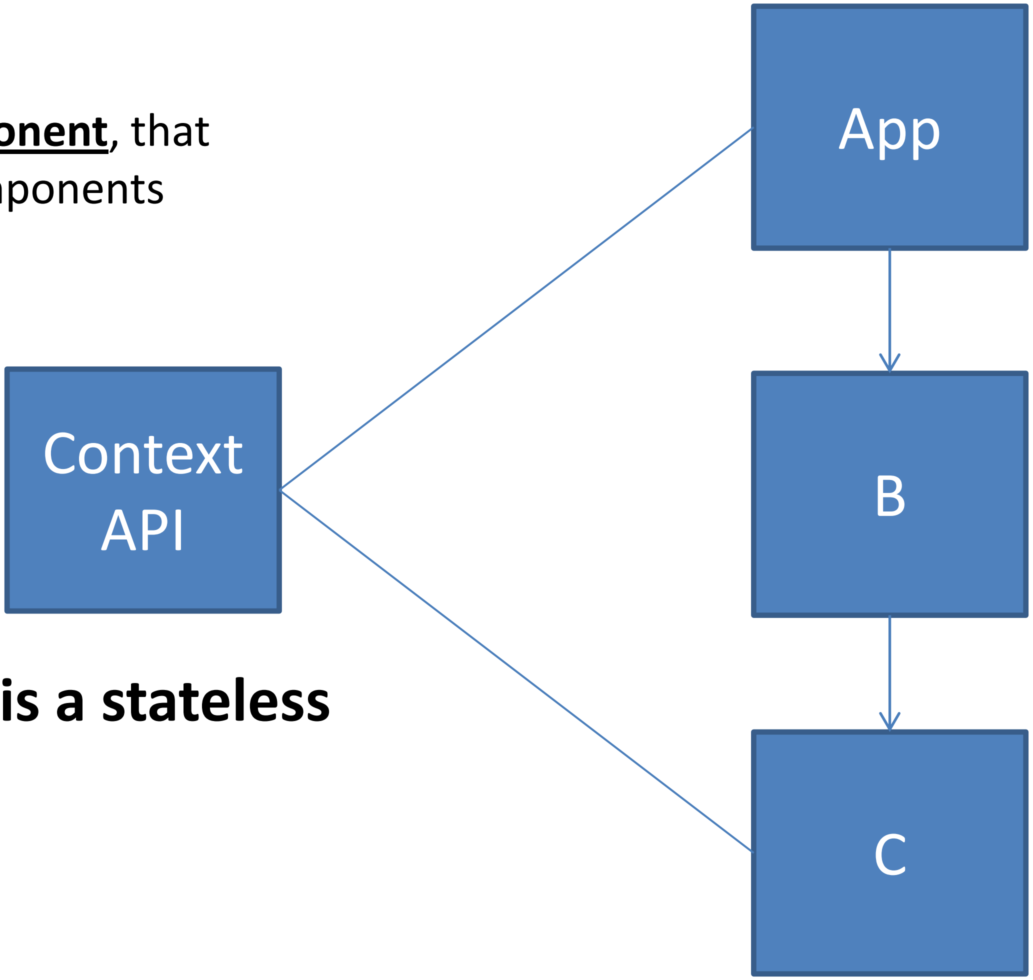
Context API is a **stateful component**, that shares its state with other components



3.5. useContext

Repetition of

Context API is a **<u>stateful component</u>**, that shares its state with other components

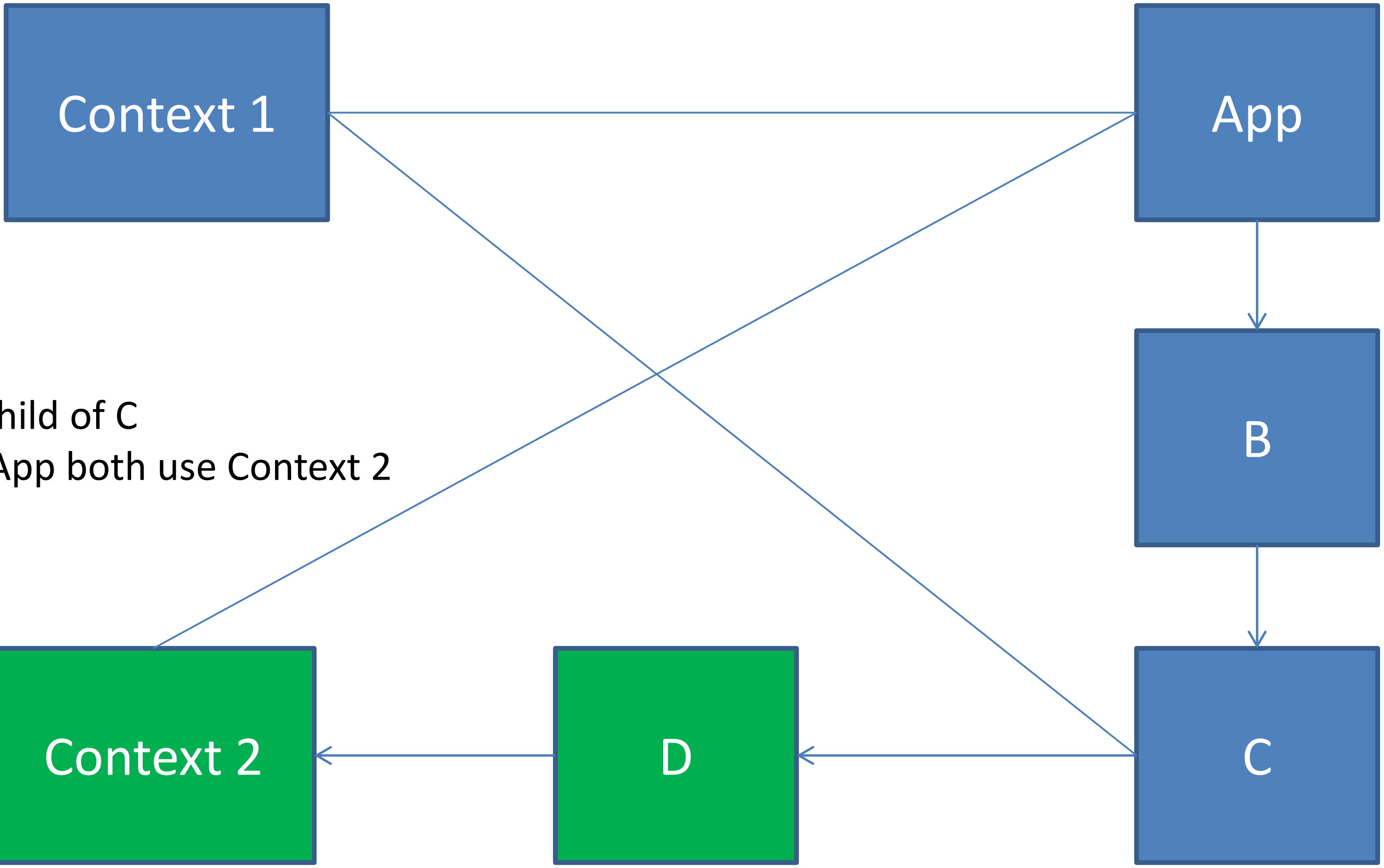


The useContext-Hook is a stateless function.

3.5. useContext

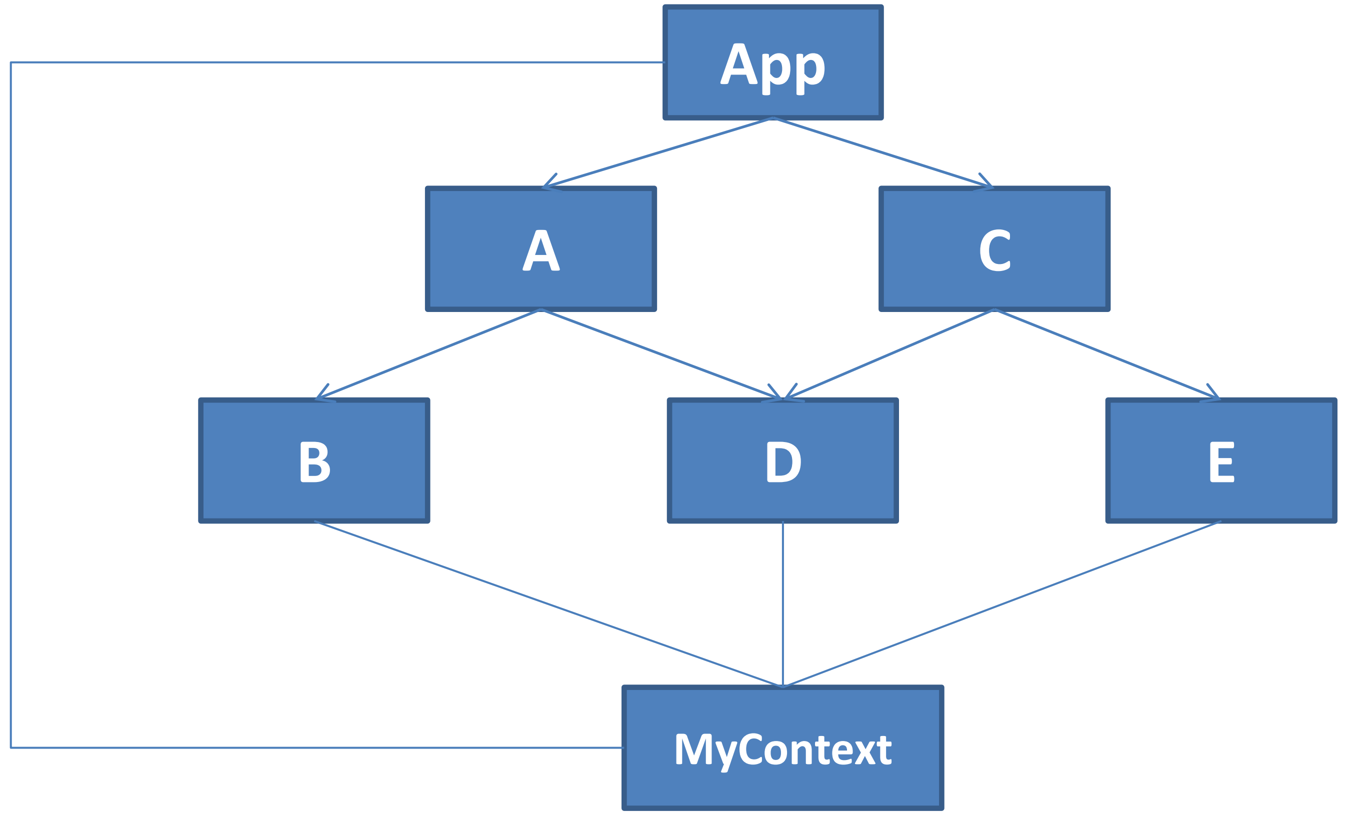
Multiple Context-APIs

D is a child of C
D and App both use Context 2



3.5. Task

1. Create a new web-app „task-3-5“ and implement the following component structure with functional components only. Also create a context MyContext that will be exported from App.js

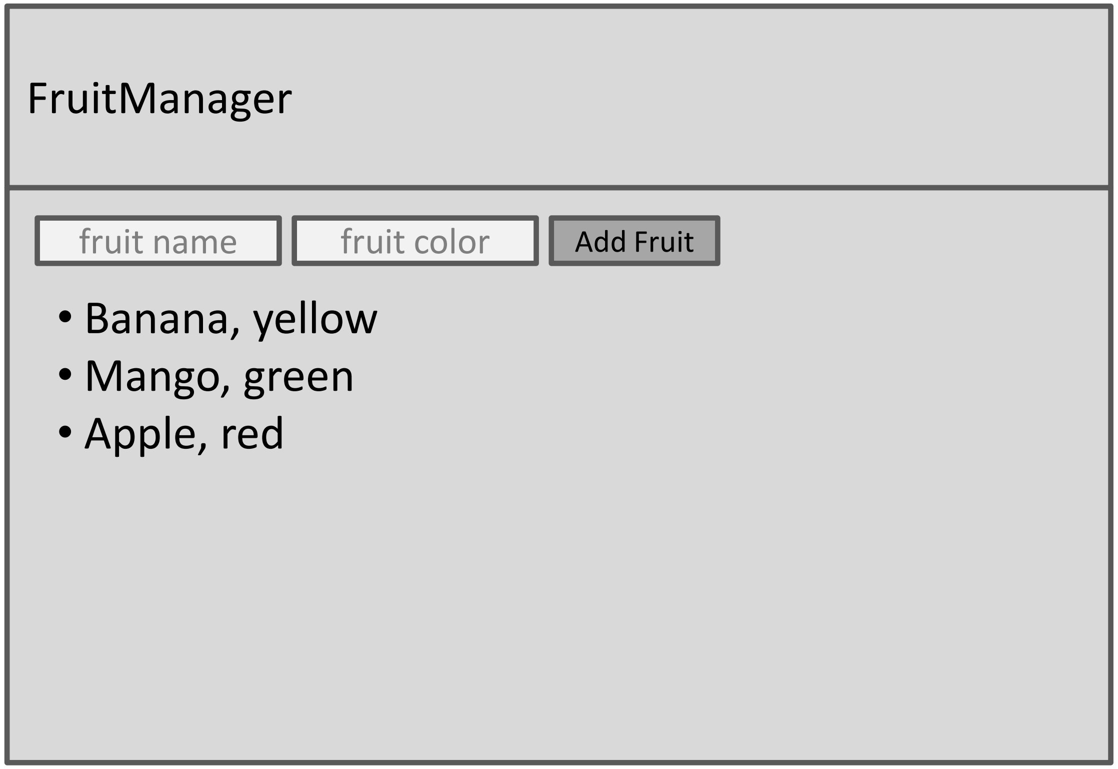


3.6. useReducer

- It does the same as useState, but uses a Reducer in front of it
  - useState is implemented via useReducer

3.6. Task

1. Create a new web-app „task-3-6“.
2. Implement this layout in the App-component!



3.6. Task

3. Add a new initital state with an array of objects of fruits. Each object has a name and a color. Put the three fruits from the layout in the initial state (yellow banana, green mango, red apple).
4. Implement the following behaviour: when the user types in the textboxes, the name and the color will be read out of the textboxes and saved in two variables newFruit and newColor which are managed by useState.
5. When the user clicks on „Add Fruit“, newFruit and newColor will be taken to create a new fruit in the array of fruits objects.

3.7. Task

1. Refactor your solution of task 3.5 to use useReducer instead of useState. Save the new solution as „task-3-7“.

   Tip: Use two reducers.

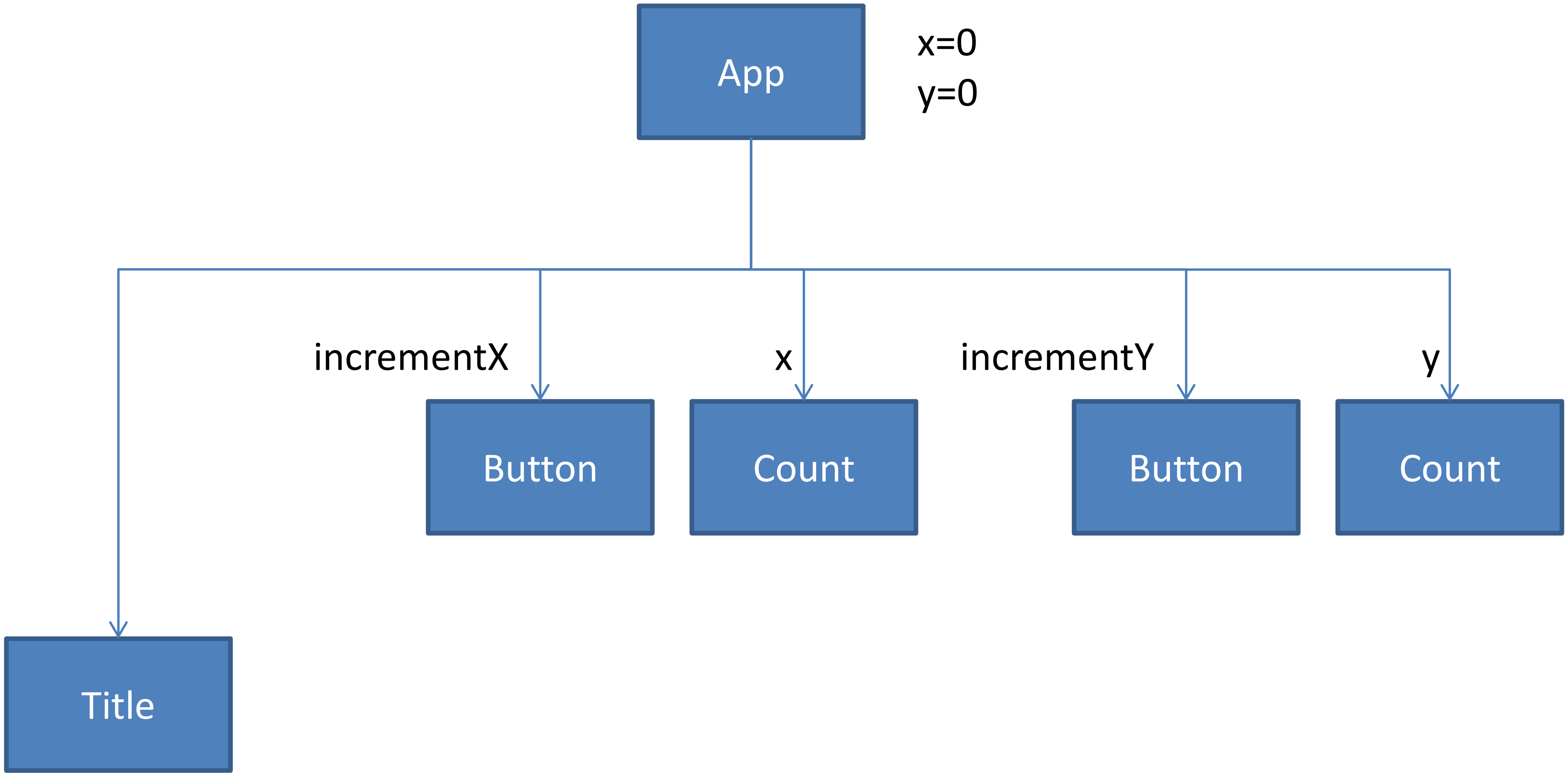
3.8. useCallback

- The hook useCallback saves the value of a function reference
- When does the value of a function reference change?

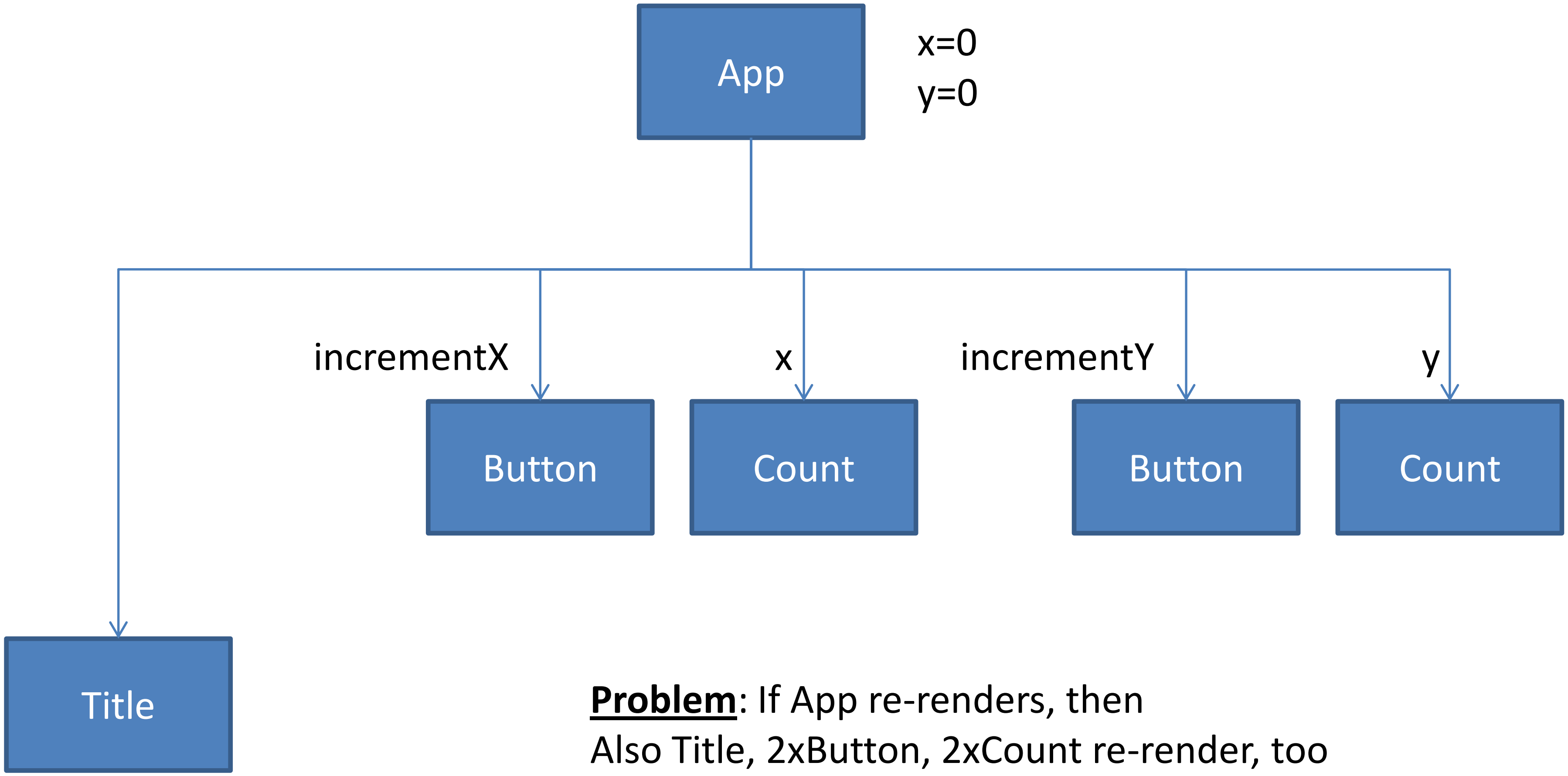
3.8. useCallback

- The hook useCallback saves the value of a function reference
- When does the value of a function reference change?
  - When the body of the function changes

3.8. useCallback



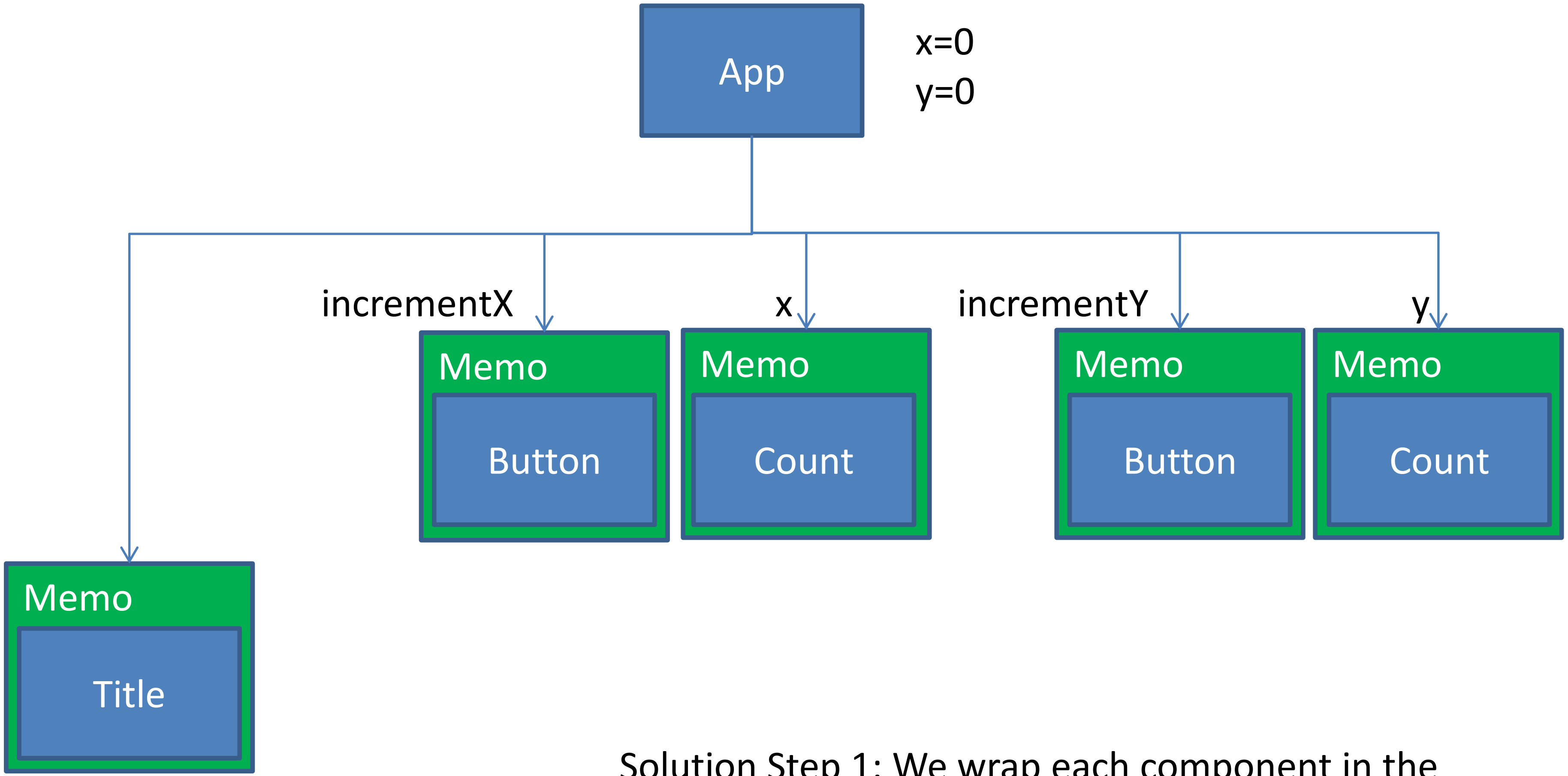
3.8. useCallback



<u>Problem</u>: If App re-renders, then
Also Title, 2xButton, 2xCount re-render, too

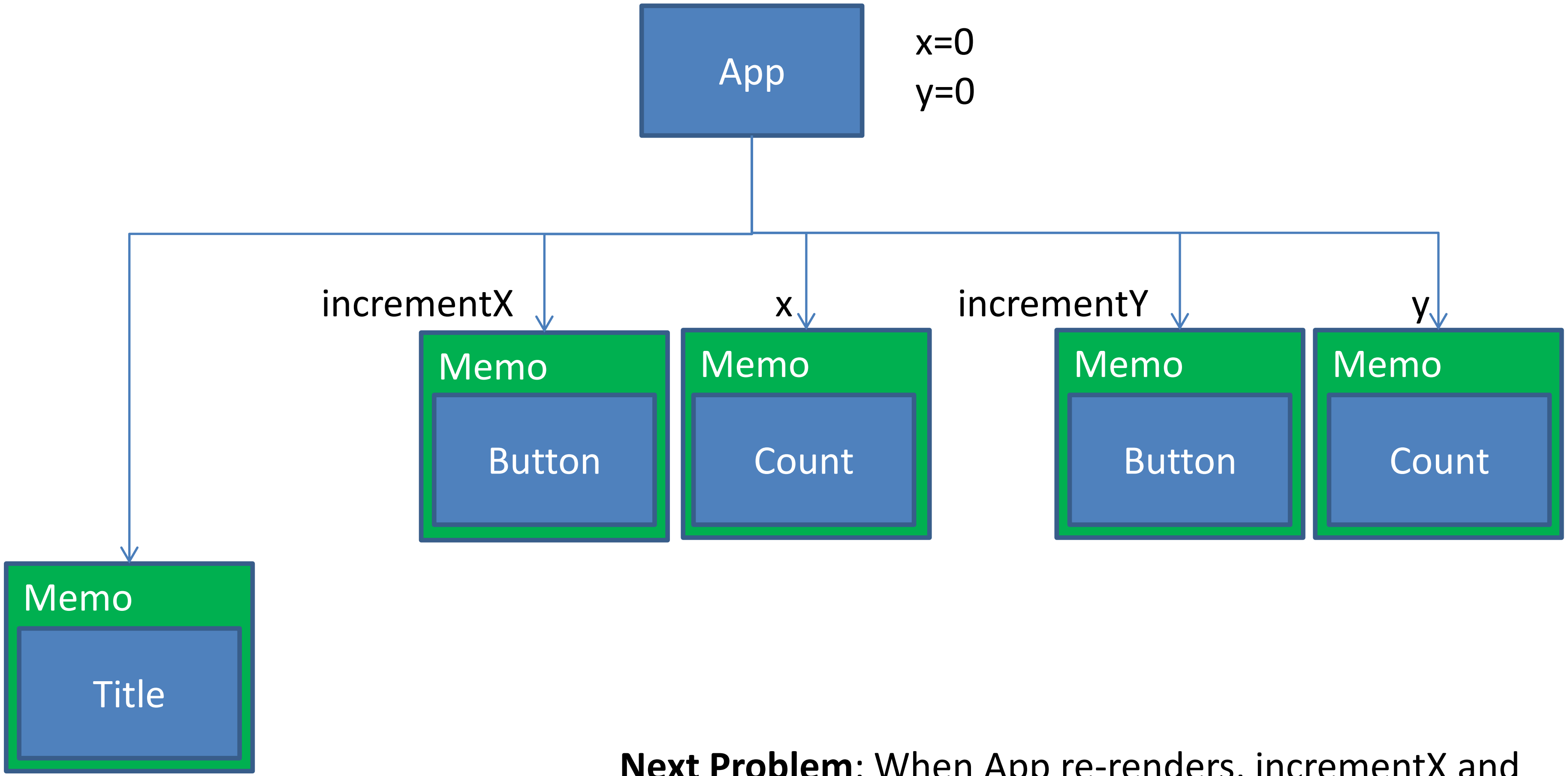
-> Too many renders! Very inefficient!

3.8. useCallback



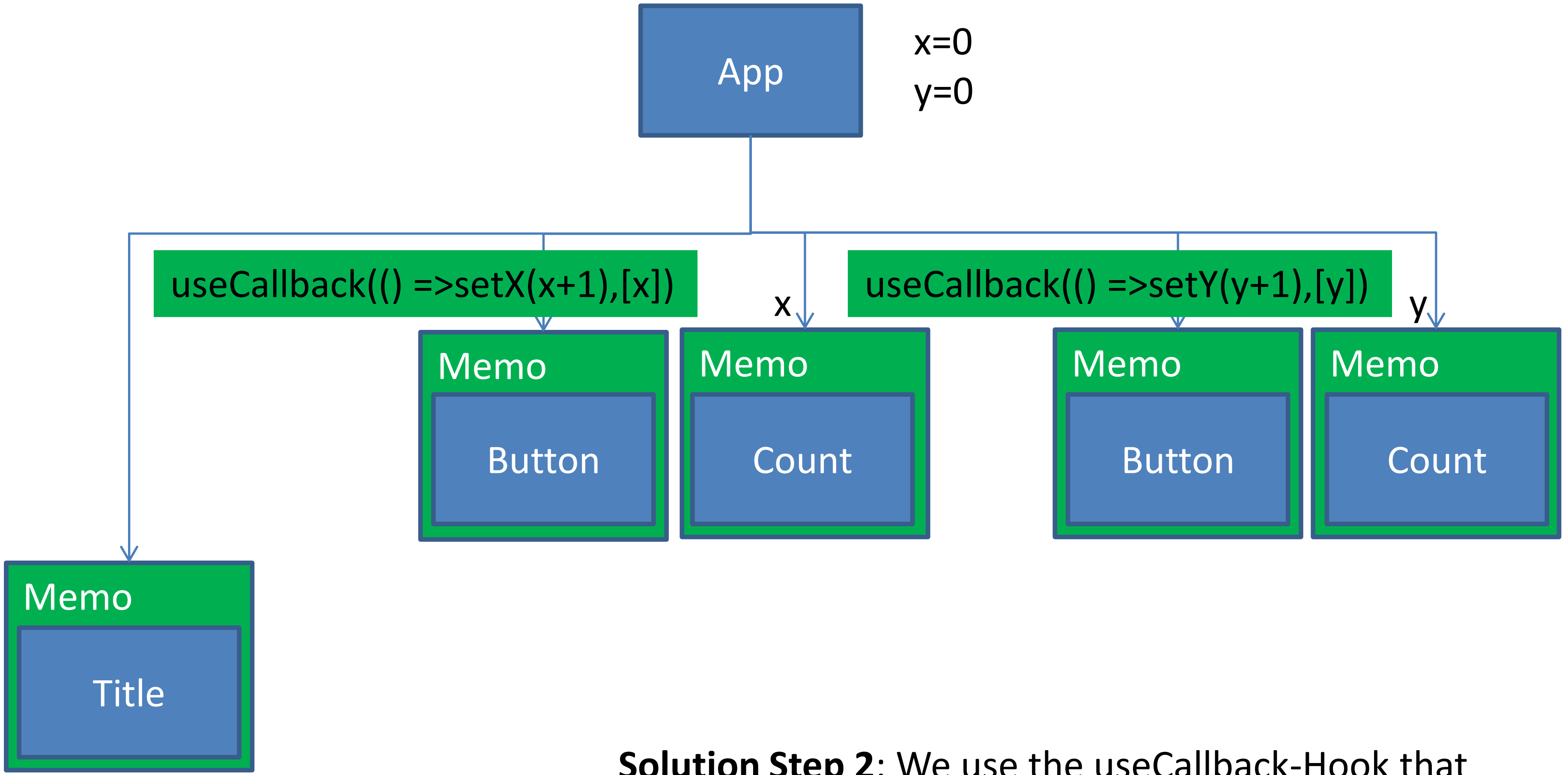
Solution Step 1: We wrap each component in the Memo-HOC which checks first if the incoming props differ from the past incoming props

3.8. useCallback



<u>Next Problem</u>: When App re-renders, incrementX and incrementY will be re-defined and therefore, their function references change

3.8. useCallback


App
x=0
y=0
useCallback(() =>setX(x+1),[x])
x
useCallback(() =>setY(y+1),[y])
y
Memo
Button
Memo
Count
Memo
Button
Memo
Count
Memo
Title

<u>Solution Step 2</u>: We use the useCallback-Hook that caches our function reference values which will still be available in the next render of App

3.8. Task

1. Create a new web-app “task-3-8“.
2. In the App-component, create two constant arrays. They do not change at all! One that contains five colors: yellow, red, green, orange and blue. The second contains four types of fruits: banana, mango, apple and lemon.
3. In the App-component‘s state, create an array of fruits. Each fruit has a type and a color. Inititally, the fruit array consists of three fruits: yellow apple, red mango and blue banana. Give each fruit an id (i.e. 1, 2 or 3).
4. Display the fruits in an unordered list.
5. Create a button above the list “Add Random Fruit“. When this button is pressed, a new fruit must be generated as random combination of color and type. Also generate a new id for the new fruit! Eventually, add the new fruit to the unordered list.
6. Create a new functional component Fruit and replace each li-item of the unordered list with its own Fruit-component. Therefore, inside the Fruit-component, receive the fruit object as prop.
7. In the Fruit-component, add a console.log that says “Fruit component rendered“. Afterwards, take a look at the output on your console when loading the app and when adding a new fruit. How many re-renderings are there? How can you minimize the number of re-renderings.
8. In the App-component, add a function “sayHallo“ that expects a fruit-object as parameter and alerts out “Hallo, I am a TYPE and my color is COLOR“. So, if the object is the orange banana, the alert would be “Hallo, I am a banana and my color is orange“.
9. Pass the reference of the sayHallo-function down to the Fruit-component as prop.
10. In the Fruit-component, add a button “Hallo !“ inside each li-element. Add an onClick-event that calls the down-passed function reference to sayHallo with the fruit-object als parameter.
11. Again, take a look at the number of renderings? How can you minimize them?

3.9. useMemo

- The useMemo-Hook saves the value of a function
- When does the value of a function change?

3.9. useMemo

- The useMemo-Hook saves the value of a function
- When does the value of a function change?
  - When the body of a function changes
  - When the parameters change

3.9. Task

1. Create a new web-app “task-3-9“.
2. Create a textbox with a placeholder „Enter ID“.
3. Create a button labeled “Load Post“.
4. Underneath, create two divs.
5. When the button is pressed, the value of the textbox needs to be taken to load JSON-data from the following URL: https://jsonplaceholder.typicode.com/comments/ID
   Where as ID will be replaced by the ID entered in the textbox. Please use fetch for that. Take the “name“ and “body“ of the resulting object and save them in the two divs respectively.
6. What happens if the user loads the data twice or more for the same id? How many fetch-requests will be sent to the server? Please minimize the number of fetch-requests. I.e. The user has loaded the post with ID = 1 already, if the user clicks again on the “Load Post“-button, another fetch-request should be avoided.

3.10. useRef

- Refs provide direct access to DOM elements created in the render method
  - The same as React.createRef()

3.10. Task

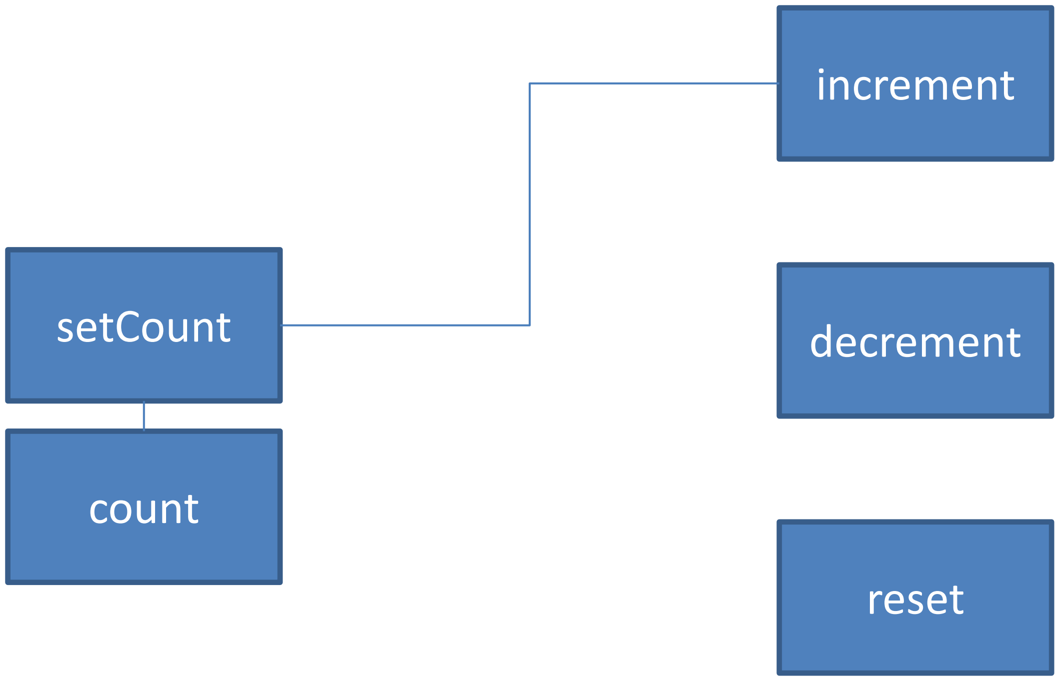
1. Create a new web-app “task-3-10“.
2. Go to YouTube and find a music video that you like. Click on „Share“ and copy the embed-code.
3. Paste the Embed-code in your App-component and create a variable “iframeRef“ that gives direct access to the iframe-DOM element.
4. Underneath the iframe, create two buttons “Play Song A“ and “Play Song B“.
5. Go to YouTube again and find another music video that you like. This time, only copy the URL-part of the embed-code.
6. Now, when “Play Song A“ is pressed, load first video, when “Play Song B“ is pressed, play the second video. Only use the useRef-hook for the implementation.

3.11. Custom Hooks

- Custom Hooks are functions, that contain React Hooks to implement a certain logic

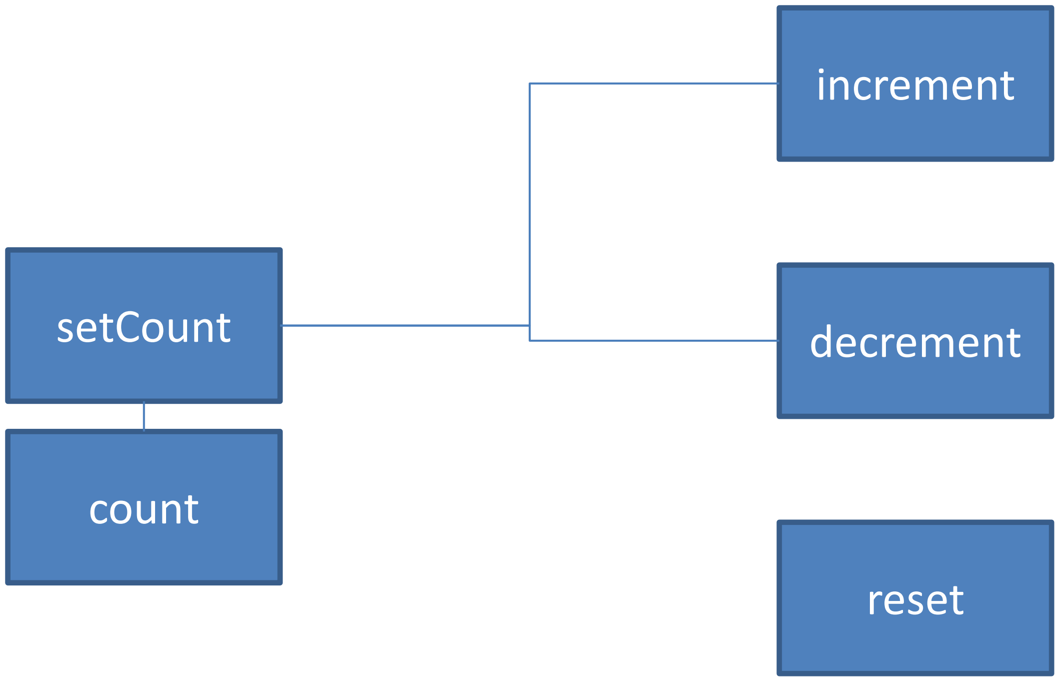
3.11. Custom Hooks

Number of calls setCount = 1



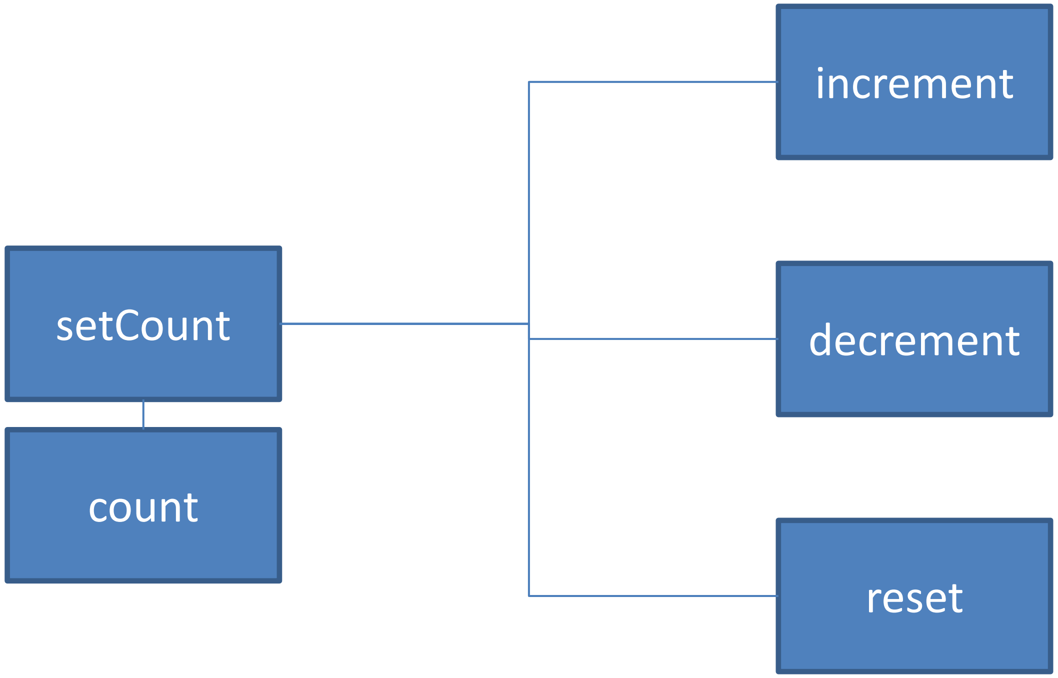
3.11. Custom Hooks

Number of calls setCount = 2



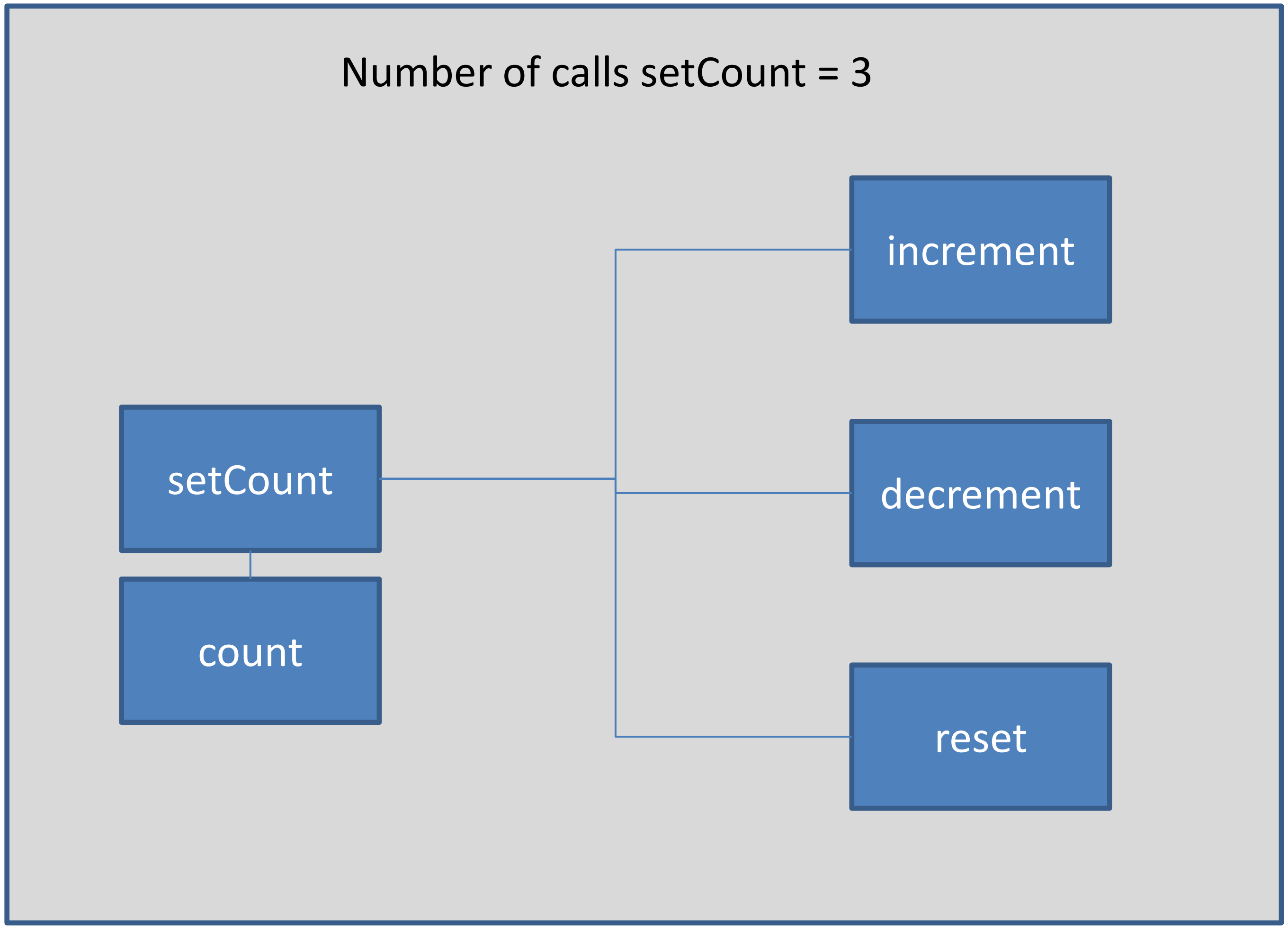
3.11. Custom Hooks

Number of calls setCount = 3

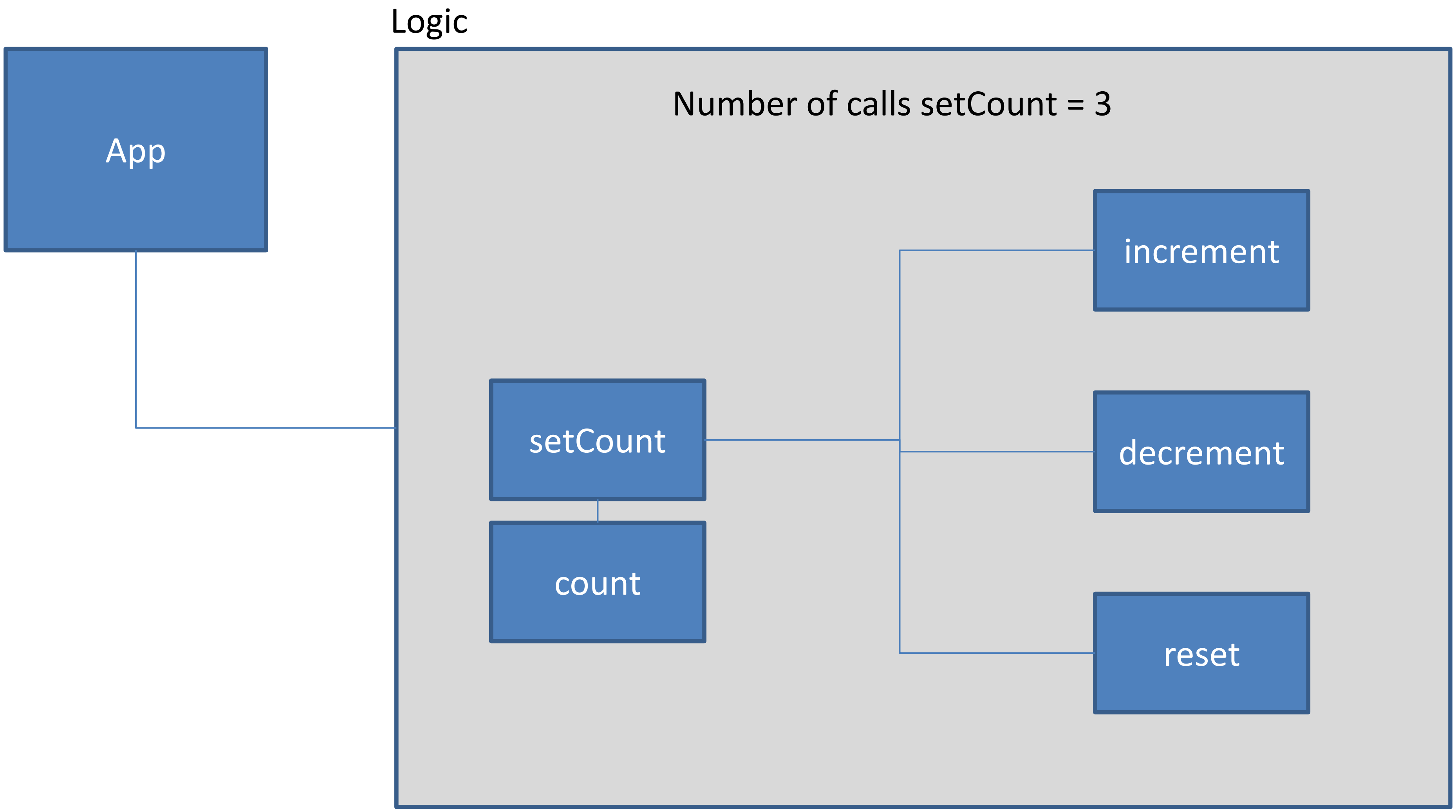


3.11. Custom Hooks

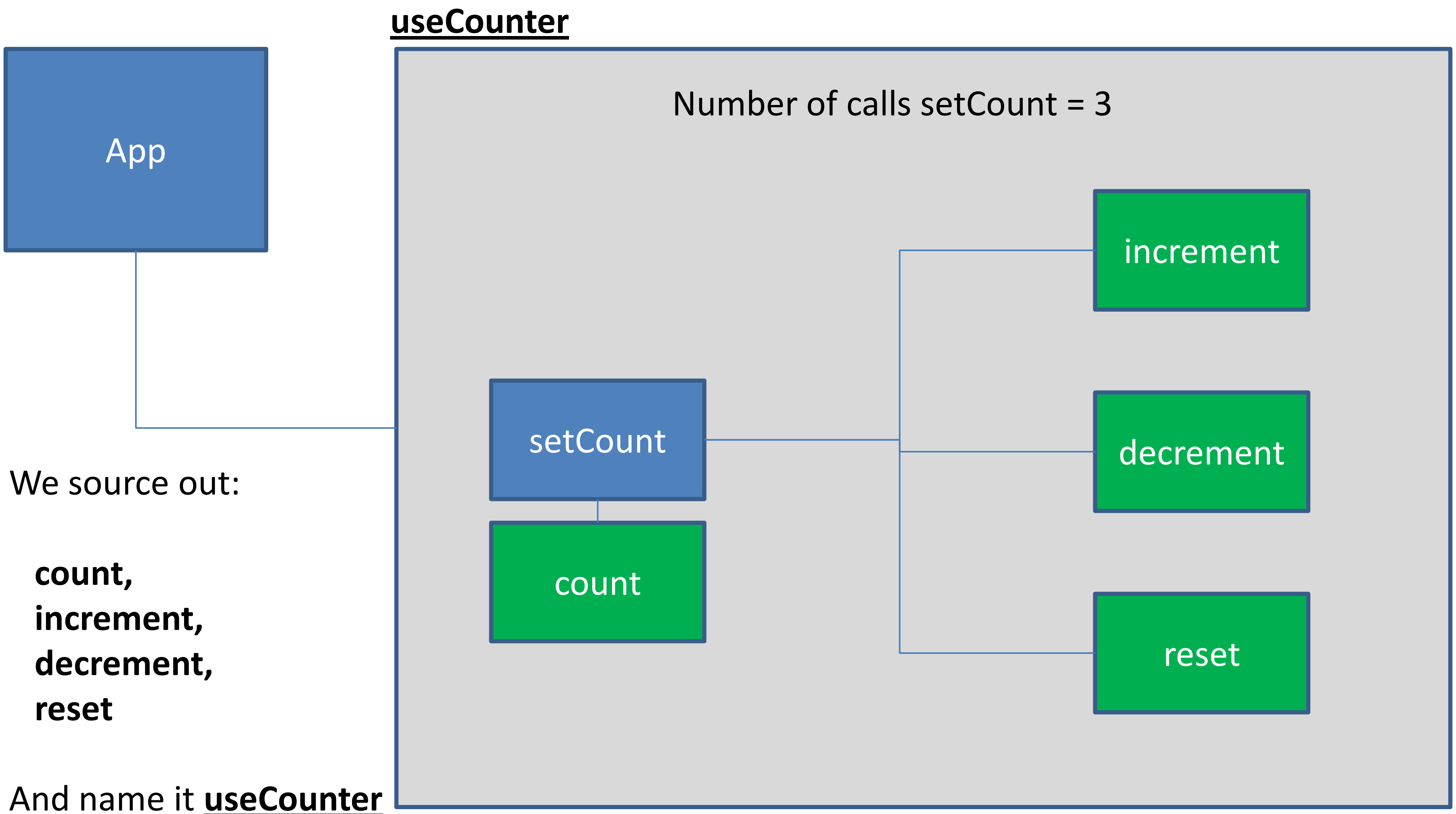
Logic



3.11. Custom Hooks



3.11. Custom Hooks



We source out:

count,
increment,
decrement,
reset

And name it **useCounter**

3.11. Task

1. Create a new web-app “task-3-11“.
2. Inside the App-component, create a new state-variable x which is initially set to 0 and its setter function setX.
3. Create a button which is labeled as “incrementX“. Whenever “incrementX“ is pressed, increment x by 1 and show the value x underneath the button.
4. Create a custom hook “useDocumentTitle“ which updates the document‘s title to the current value of x.