PRACTICAL REACT WITH TYPESCRIPT

bouvet

Agenda

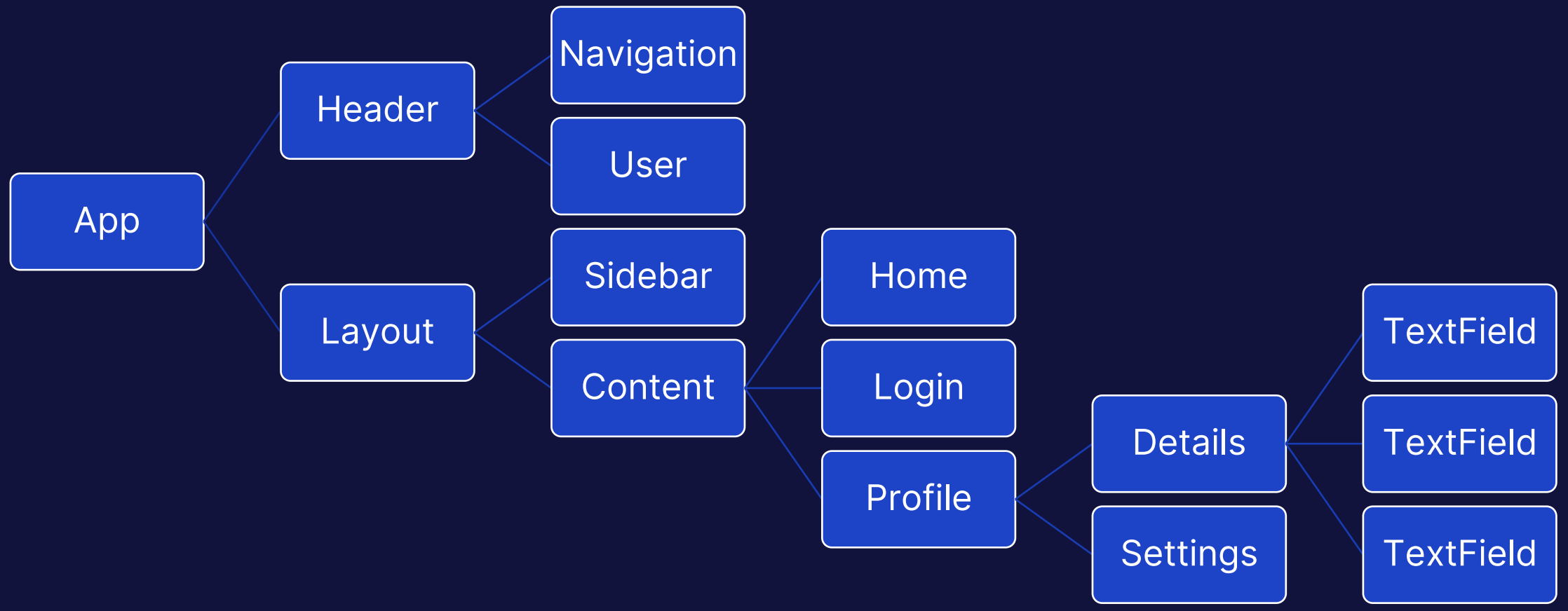
- React basics
  - Components and JSX
  - Props and state
  - Events
  - Lifecycle
- Structure and patterns
  - Hoisting
  - Composition
  - Contexts
  - Type-definitions with Typescript
  - File and folder structure
- Building applications
  - Styling
  - Routing
  - Immutability
  - Optimization
  - Code-splitting
  - Testing
  - Server communication

bouvet

React basics

> A JavaScript library for building user interfaces
>
> -reactjs.org

Anatomy of React



TextField

- Create a component that renders a text field with a label.
- Clicking the label should put focus in the text field.
- Print the text from the text input under it.

<> BooleanField

- Create an input field component for inputting boolean (true/false) values.
- It should have an input field and a label like the TextField except the label should be placed after the checkbox.

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}
```

Interface describing the components **props**

The **component** function

```
export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

Arguments to a React component are usually called **props**

Anatomy of a component

use* functions are called **hooks** and usually «hook» into the React engine.

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}

ex                                    dProps) => {
  const id = uuid()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

useState hooks into Reacts state mechanism allowing storage and retrieval of state.

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
```

A React component must return something that React can render.
Here a nested **jsx** object is returned.

```
  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")
```

JSX works like a template, you can run arbitrary JavaScript inside { }.
Here we set the value of the **htmlFor** prop of **label** to the value of the **id** variable.

```
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

The value between an opening and closing tag is called the **children**.
Here we set the **children** prop of the **label** to the value of the **label** prop of **TextField**

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}

expo
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

useState returns a tuple with a current value and a setter to update it.
We can **destructure** this into two variables for use in our component.

Anatomy of a component

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

We set the **value** prop of the **input** component to the current value state.

And set the **onChange** prop to a **function** that will update the state based on the value of the input.

Anatomy of a component

- **Component:** A JavaScript function that returns something react can render.
- **Props**: Arguments to the component.
- **Hooks**: use* functions inside the component.
- **State**: persisted «variable» with a current value and a setter.
- **Children**: Value between opening and closing tag (just another prop)
- **JSX**: Template language that looks like html
- **{ }**: Where you put JavaScript in **JSX**.

Anatomy of an event

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

When an **input*** event occurs run my function

* For historical reasons binding to the **input** event is called **onChange** in React. The underlying HTML event is **input**.

Anatomy of an event

```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

The **event handler** updates the state value using the **setter**.

Anatomy of an event

```
export interface TextFieldProps {
  label: string
}
```

State change triggers React to **re-render** the component with **updated data**.

```
export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

Anatomy of an event

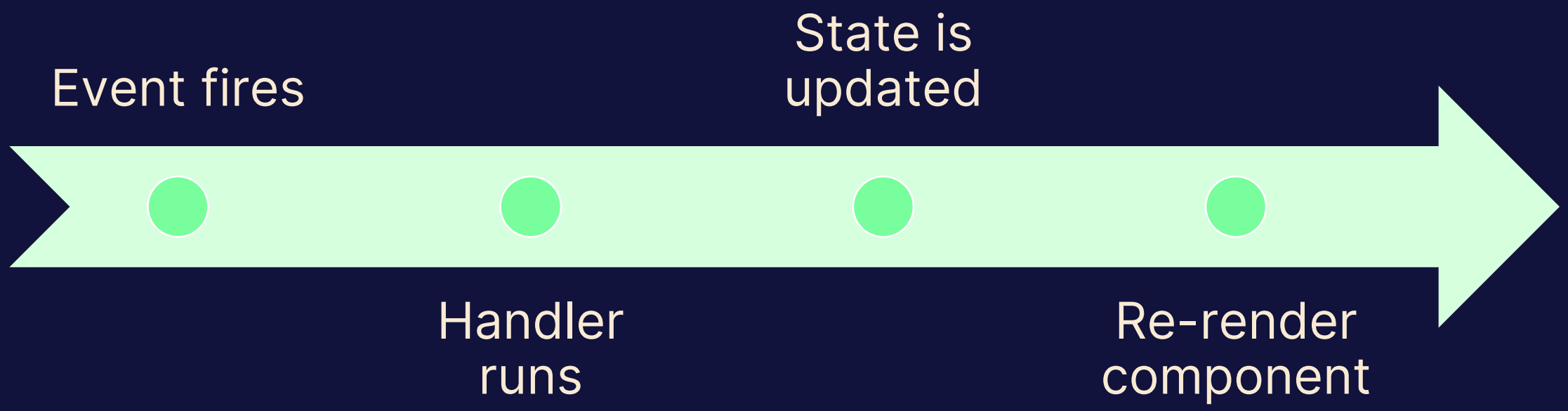
```
export interface TextFieldProps {
  label: string
}

export const TextField = ({ label }: TextFieldProps) => {
  const id = useId()
  const [value, setValue] = useState("")

  return (
    <div>
      <label htmlFor={id}>{label}</label>
      <input id={id} type="text" value={value} onChange={(evt) => setValue(evt.target.value)} />
      <p>{value}</p>
    </div>
  )
}
```

Updated value is passed to the **value** prop and **updating the UI**.

Anatomy of an event

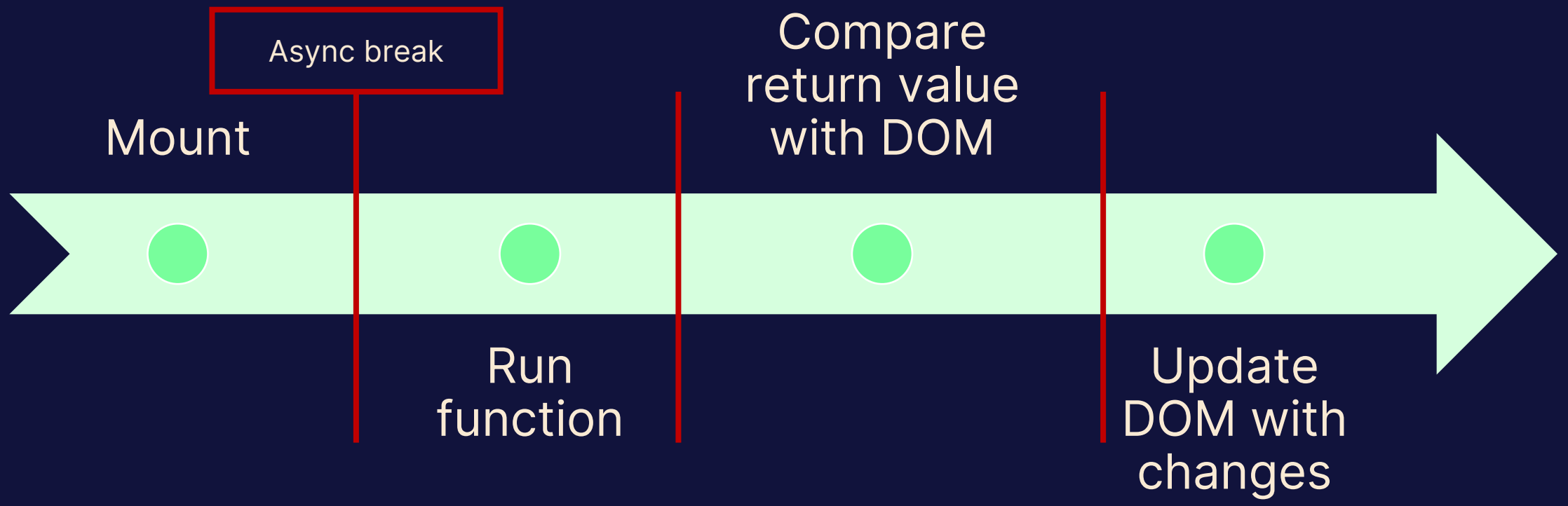


Component lifecycle

- Mount
  - Component is added to the screen.
- Update
  - Any props or state is updated.
- Unmount
  - Component is removed from the screen.

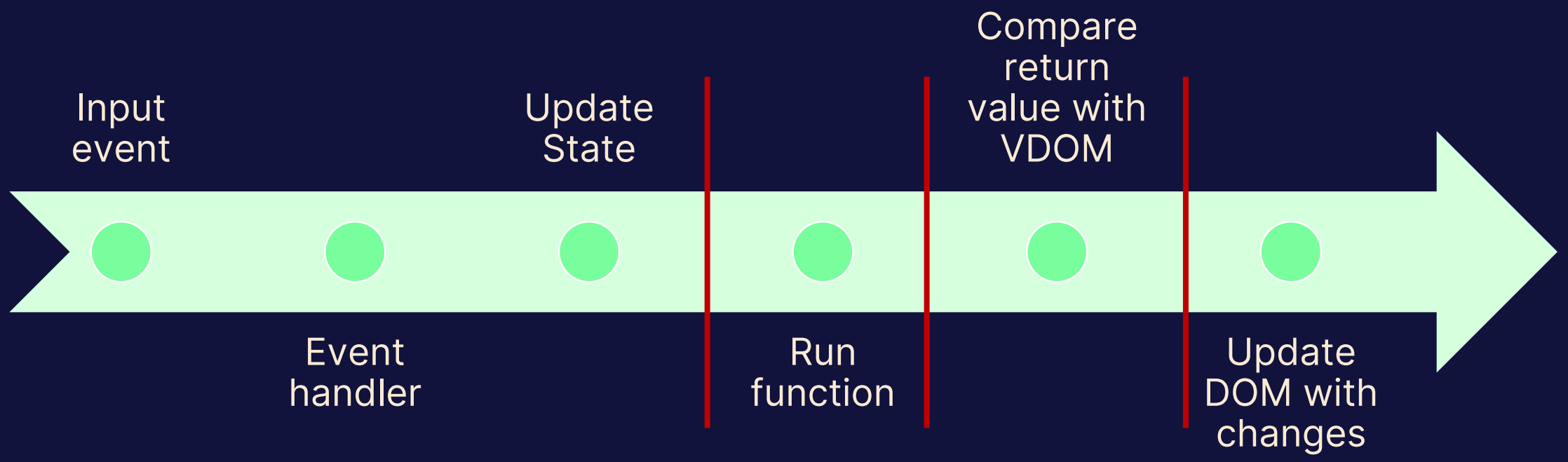
Component lifecycle

TextField



Component lifecycle

TextField



<> ClickUntil

- Create a component with a button and a paragraph.
- Count the number of times the button is clicked and show the count in the paragraph.
- When the limit is reached disable the button and show a “limit reached” message instead of the paragraph.
- Add another button that resets the count.
- The limit and message should be configurable.

<> NumericField

- Create an input field component for inputting numeric values.
- It should have an input field and a label like the TextField.
- The following parameters should be configurable as props.
  - A minimum value (default 0)
  - A maximum value (default 100)
  - Whether or not decimals are allowed (default false)
- If the distance between the minimum and maximum values are <= 50 and decimals are not allowed use “range” input.

<> ClickUntilForm

- Create a parent component ClickUntilForm with states for the «limit» and «limit message» of ClickUntil.
- Provide fields for the user to edit these states and apply them to a ClickUntil component.

Styling

- Many different styling techniques
  - CSS/CSS modules
  - Style-props
  - CSS-in-js: Styled-components/Emotion++

Style TextField

- Create some basic styling for the TextField label.

<> Style components

- BooleanField
- NumericField
- ClickUntil