PRACTICAL REACT WITH TYPESCRIPT

Setup

- Install
  - Node LTS - https://nodejs.org/en
  - Git - https://git-scm.com
  - Visual Studio Code - https://code.visualstudio.com

- https://tinyurl.com/practical-react
  - `git clone https://github.com/rudfoss/practical-react-with-typescript.git`

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

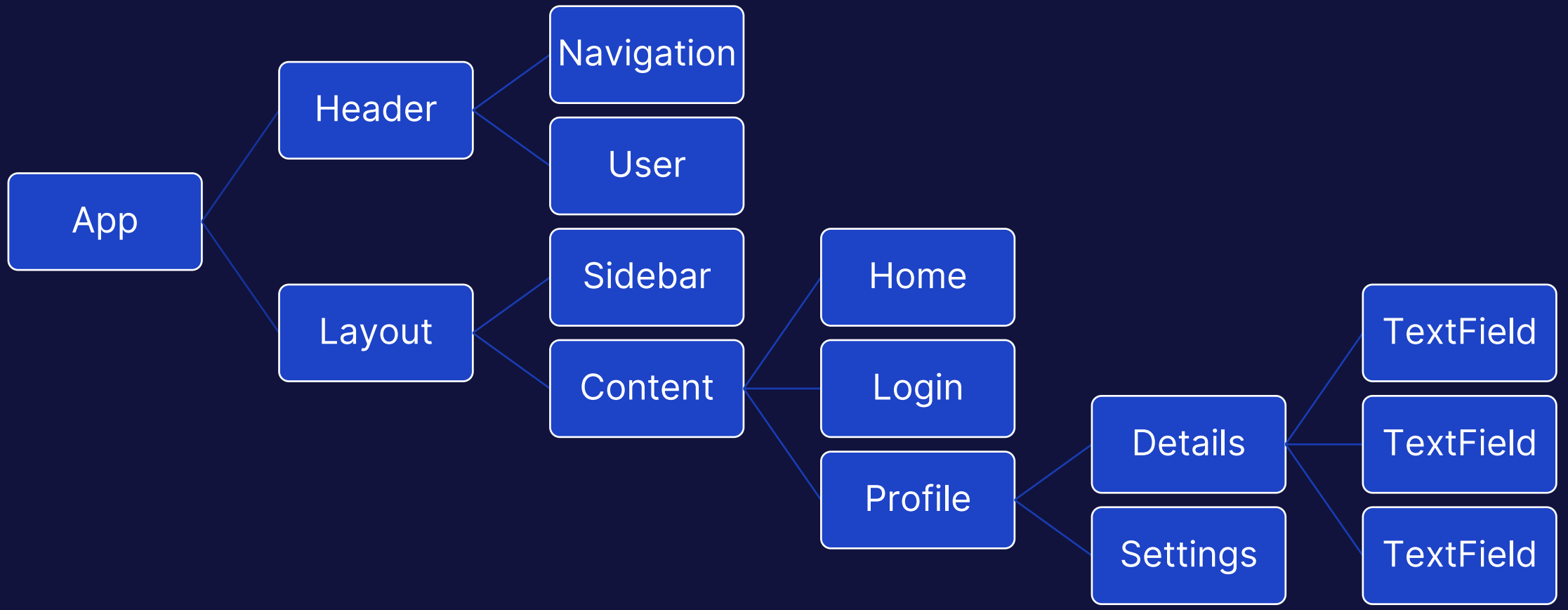
bouvet

React basics

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

bouvet

Anatomy of React



bouvet

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.
- Add a button to clear the text.

bouvet

<> 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

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

Interface describing the components **props**

The **component** function

```tsx
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

```tsx
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.

```tsx
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

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

ex                                    dProps) => {
  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 hooks into Reacts state mechanism allowing storage and retrieval of state.

Anatomy of a component

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

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

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

Anatomy of a component

```tsx
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>
  )
}
```

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.

Anatomy of a component

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

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

  retur
    <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

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

export function 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>
  )
}
```

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

```tsx
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.

bouvet

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

```tsx
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

```tsx
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

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

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

```tsx
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

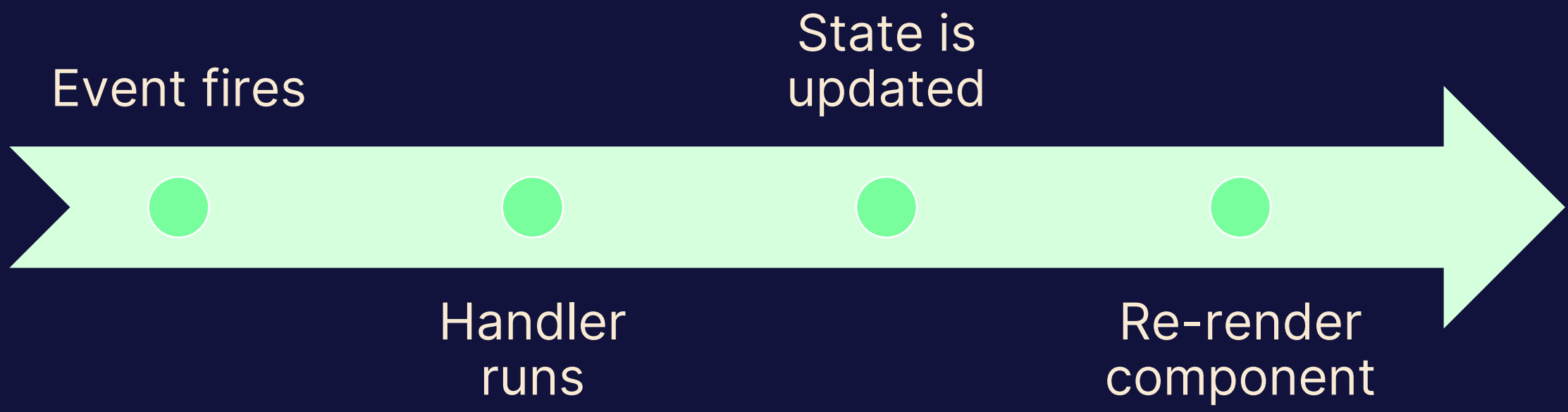
```tsx
export interface TextFieldProps {
  label: string
}

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



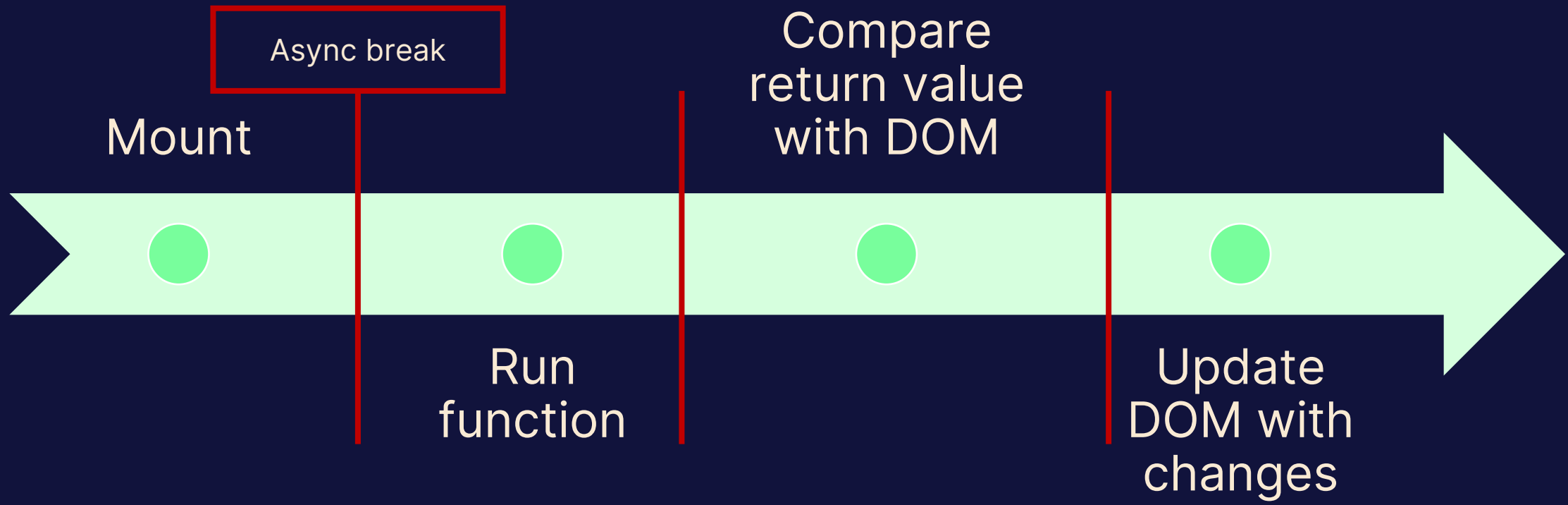
bouvet

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

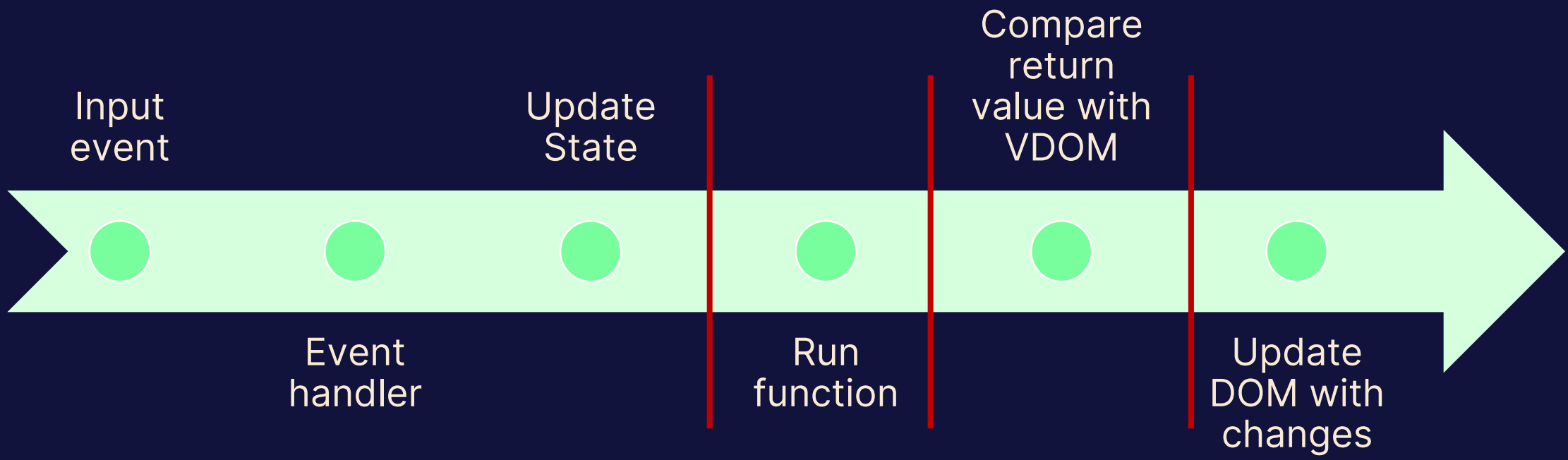
TextField



bouvet

Component lifecycle

TextField



bouvet

<> 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 |max-min| <= 50 and decimals are not allowed use “range” input.

<> ClickUntilForm

- Create a 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.

Immutability

- Data cannot change once created.
- You cannot change data you do not own.

UserDetails

- Copy a single user from the users API for this task.
- Create a component that takes a user as a prop and these details:
  - User name
  - Id
  - First Name
  - Last Name
  - Email
  - Age
- Make firstName and lastName editable using TextField.
- Create a parent component and add multiple instances of UserDetails to that page with the same user object as input.

<> UserDetailsEditable

- Make all fields editable in UserDetails except userName.
- Add a save button that saves values from one UserDetails component to the central user state.

- Bonus: Update all other UserDetails with new user values once any UserDetails component performs a save.

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.

bouvet

<> Style components

- BooleanField
- NumericField
- ClickUntil

bouvet

EoD 1

bouvet

Organizing our repository

- Apps: Deployable elements
  - Bootstrapping
  - Routing
- Libraries: Reusable elements
  - Components
  - Features
  - Utilities
  - Layouts
  - Services
  - ++

Loops

- Repeatable JSX components that represent list data.
- Using «keys» (identifiers) to optimize updates.

ListGroupData component

- Copy groups from API for this task.
- List group names in an ordered list.
- When an item is clicked highlight it and display it above the list, include the items index.
- Allow sorting in ascending or descending order by name.
- Add a delete button to each item that removes it when clicked.

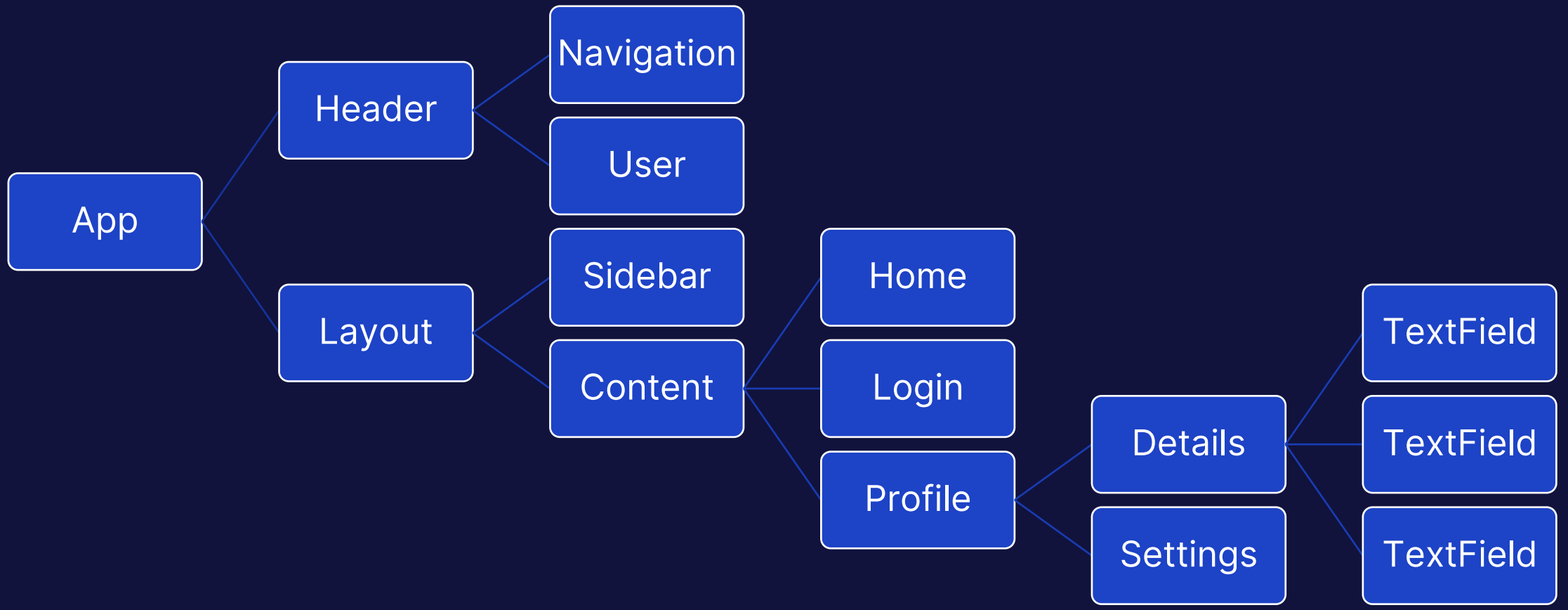
bouvet

<> UsersTable

- Copy users from the API for this task
- Create a component that lists all users in a table with columns: userName, firstName, lastName, email
- Make it possible to sort by each column in ascending or descending order by clicking on them.
- Make the firstName of each user editable.

- Bonus: Create a search field that filters the table based on search text written by the user. Allow searching in multiple properties.

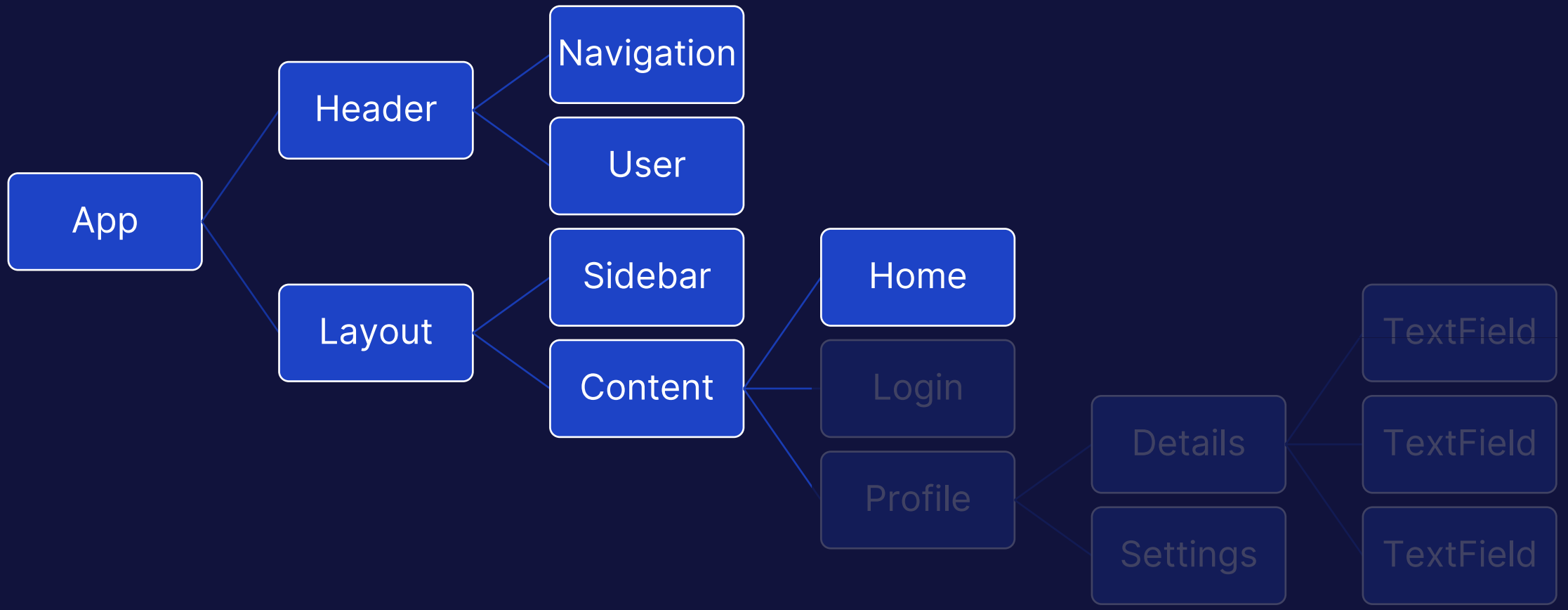
Anatomy of routing



bouvet

Anatomy of routing

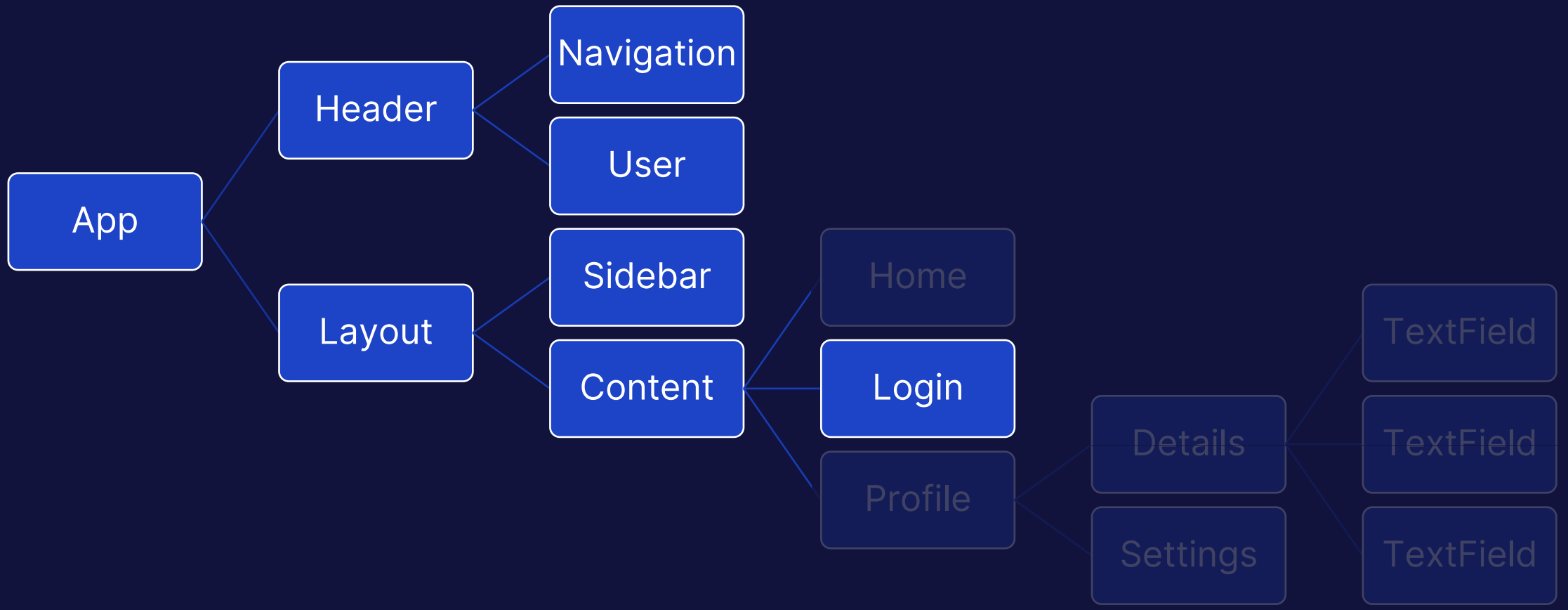
/home



bouvet

Anatomy of routing

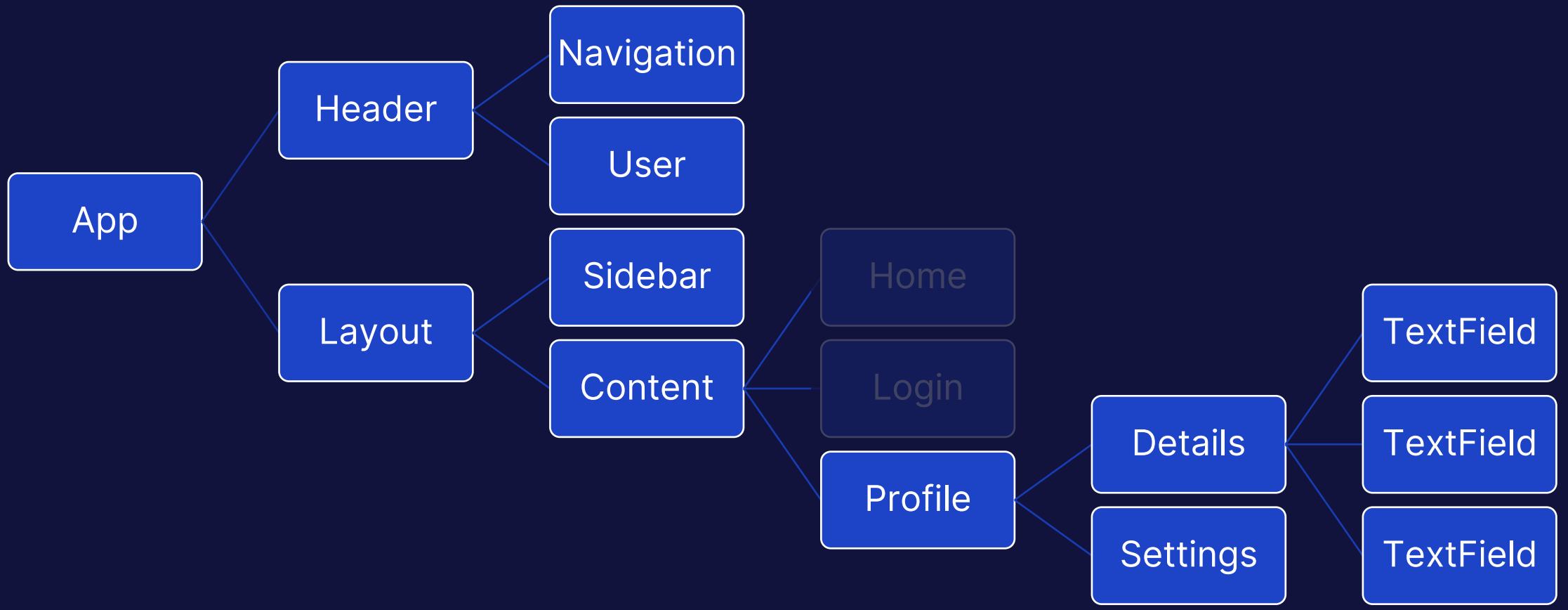
/login



bouvet

Anatomy of routing

/profile



bouvet

HomePage and UserDetailsPage

- Use a static version of all users from the API for this task.
- Create a home page and a user details page.
- Add a link to the home page that navigates to the user details page.
- Display user details for a specific user based on URL parameters.

bouvet

MainLayout

- Create a layout that displays the Navigation component in a sidebar and the Main content next to it.
- Move links to the Navigation component.

EoD 2

bouvet

Hooks

- Small functions that may use other react hooks to perform work.
- Some are built-in, but you can also create your own
  - By convention they are named use[something], e.g.: useRandom, useTitle

Most common react hooks

- **useId:** Create a stable, unique id for accessibility purposes
- **useState:** Store a variable that React can «observe»
- **useMemo:** Cache result of a function until dependencies change
- **useEffect:** Perform some non-react related action based on changes to dependencies
- **useRef:** Store a reference to something that **does not** affect rendering.

useRandom

- Create a custom hook that takes a single value and returns a new random value whenever the given value changes.

bouvet

<> useWindowTitle

- Create a custom hook that changes the «document.title» whenever a provided value changes.
- Use the hook to change the title based on which page you are currently on.

TypeScript Generics

- A «placeholder» for some value with optional restrictions.

TypeScript Generics

```ts
interface ContainerFor<TType> {
  id: string
  value: TType
}
```

```ts
const stringContainer: ContainerFor<string> = {
  id: "123",
  value: "foo"
}

const booleanContainer: ContainerFor<boolean> = {
  id: "123",
  value: true
}

interface Name {
  firstName: string
  lastName: string
}
const objectContainer: ContainerFor<Name> = {
  id: "123123",
  value: {
    firstName: "Test",
    lastName: "Testington"
  }
}
```

TypeScript Generics

```ts
interface ContainerFor<TType extends string | boolean> {
  id: string
  value: TType
}
```

```ts
const stringContainer: ContainerFor<string> = {
  id: "123",
  value: "foo"
}

const booleanContainer: ContainerFor<boolean> = {
  id: "123",
  value: true
}
```

bouvet

TypeScript Generics

```ts
interface GenericInterface<T> {
  id: string
  value: T
}

type GenericType<T> = {
  id: string
  value: T
}

const genericFn = <T, T2>(firstArg: T, secondArg: T2) => {
  //...
}

function genericFn2<T, TReturn>(firstArg: T): TReturn {
  //...
}
```

bouvet

TypeScript Generics with React

```tsx
export interface GenericComponentProps<TType> {
  label: string
  value: TType
}

export const GenericComponent = <TType,>(props: GenericComponentProps<TType>) => {
  // ..
}
```

bouvet

<> ChoiceField

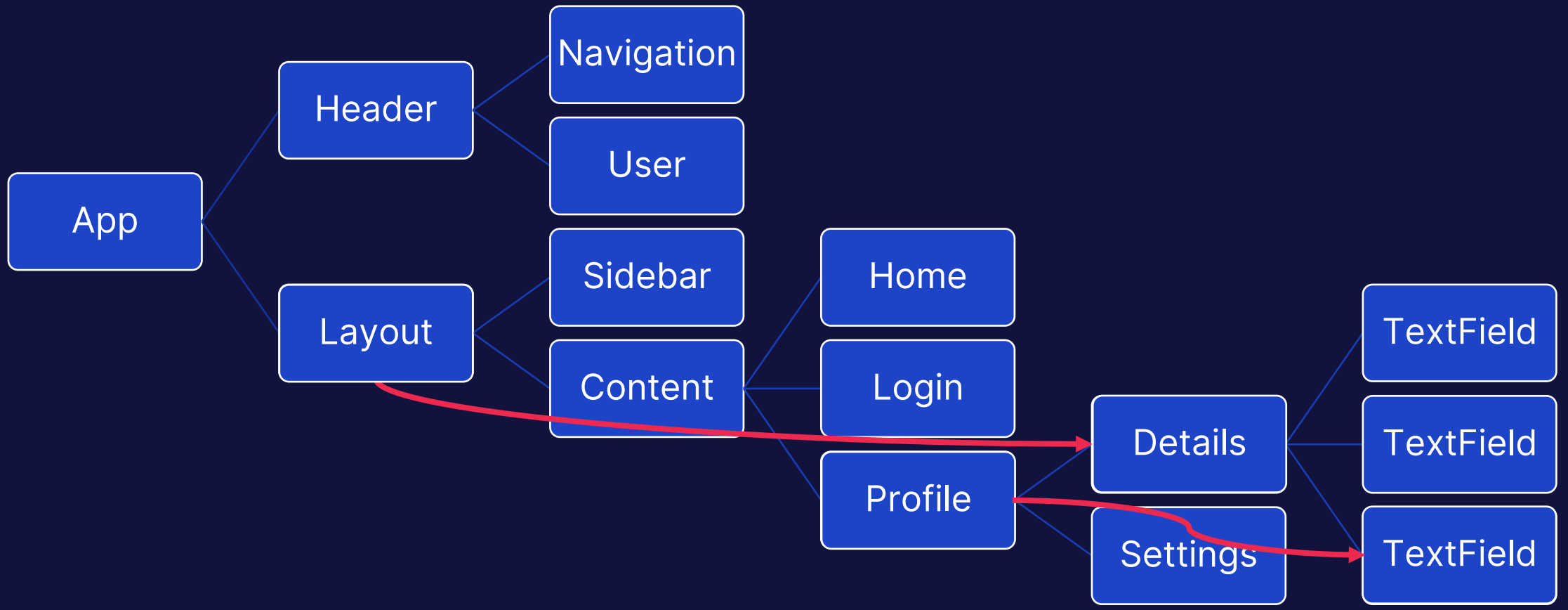
- Create an input field component for picking one choice from a set of choices. Choices are passed as props.
- Specify type for choices with label and value (value must be unique).
- Use generics as type-aids to prevent inconsistent state types.
- Add styling.

- Bonus: Allow choosing between radio buttons and a drop down.

Contexts

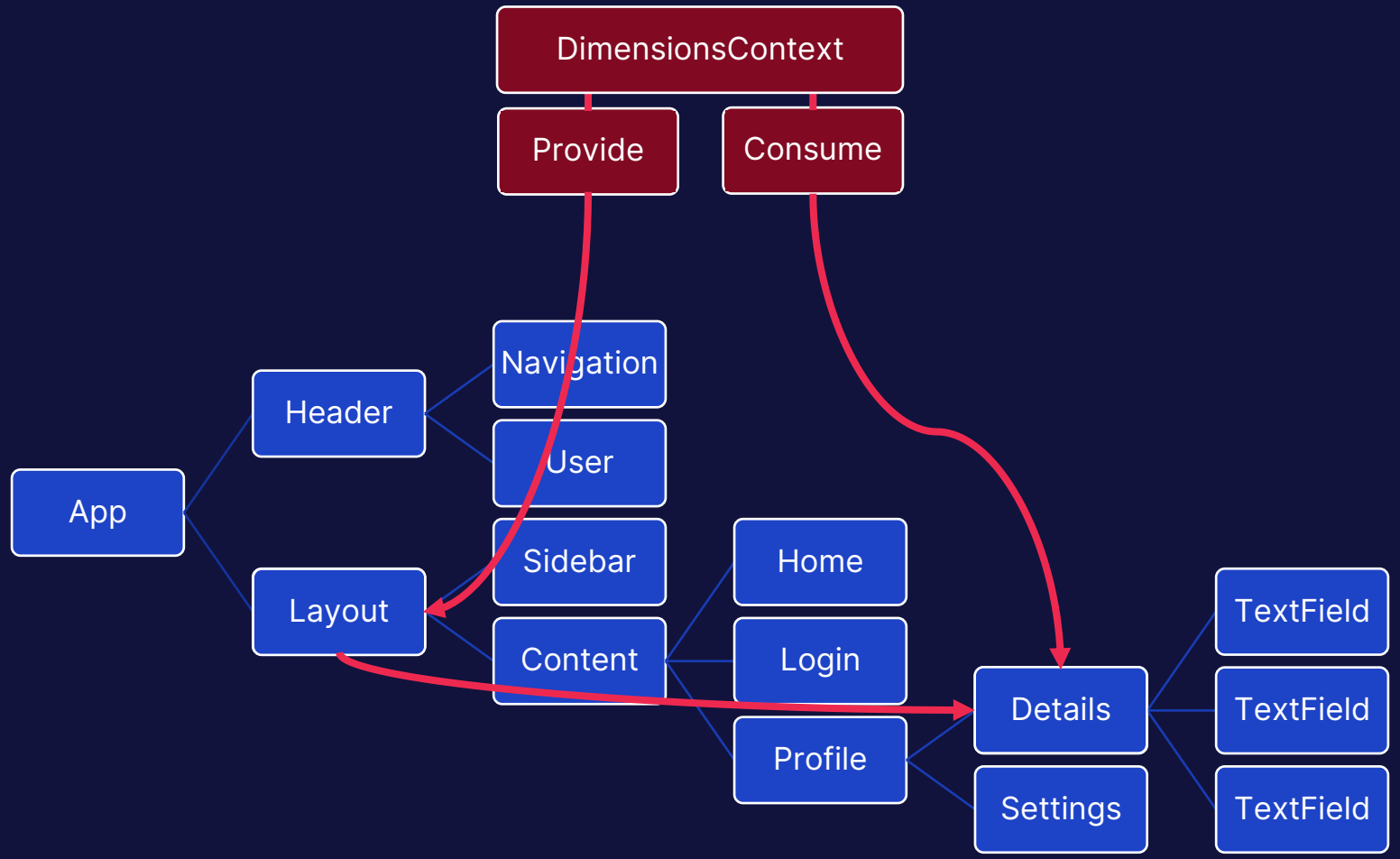
- Passing props from an ancestor component to a descendant without going through the components in-between.
- “Provide” a service to an application.

Anatomy of Context



bouvet

Anatomy of Context



bouvet

NavContext

- Create a context that provides helper functions for navigation to the UsersTableRow.

bouvet

<> DisableFieldsContext

- Create a context that disables every field currently on the page.
- Provide a way to toggle disabled state through the context.
- Add a button that toggles disabled fields.
- Update fields to use (consume) the context.

State management scopes

- Local: useState inside component
- Instanced: useState inside custom hook
- Shared Instanced: useState inside Context
- Global: useState inside single-instance Context

- State libraries: Zustand, Redux, MobX, ++

- Remote: TanStack Query

Communication with a server

- Get data from a server and send updates to it.
- Use a library to aid in state management and caching.
- Tanstack-query.

Generating clients

- APIs with OpenAPI descriptions can be used to generate clients.
- Use NSwag to generate a client from a definition.
- Provide clients through context.

<> ServerSideUserTable

- Load user data from a server.
- Use the UserTable component to display the data.
- Display the text «loading…» while waiting for users to load.
- Use tanstack-query to manage state.

UserDetails

- Create a component that takes a user ID as a prop and displays the users details by loading data from the server.
- Make firstName, lastName, email and age editable.
- Add button to save changes to the server.
- Disable all fields while saving/loading.

<> GroupDetailsPage

- Create a GroupDetailsPage displaying details for a group based on url parameters.
- Allow editing the name and description of the group.
- Add a button to undo changes.
- Highlight fields that have been changed.
- Save changes to the server and update the cache.

<> GroupsTable

- Create a component to load groups from the server.
- Display groups in a table with columns: id, name, role
- Allow sorting by columns in ascending or descending order by clicking on them.
- Display the text «loading...» while waiting for groups to load.
- Add a delete button to each row that removes a group.

- Bonus1: Require confirmation before group is deleted.
- Bonus2: Display the number of members of a group in a separate column.

<> GroupMembers

- Create a component that, given a group ID displays the group name and all members of that group.
- Add UI to add/remove members.

Testing

- Jest/Vitest
- React Testing Library

bouvet

Optimizations

- **Avoid unecessary re-renders**
  - Enable highlighting on changing components
  - Use profiler to identify unecessary render trees
- Avoid duplicate work
  - Memoize (cache) operations that don’t need to be repeated.
- Split code into smaller packages
  - Smaller bundles => less to download and less to parse
- Single-responsibility = faster development
  - Make sure components don’t do “too much”

<> UserSearch

- Add a search field to the UsersTable allowing you to search for users by userName, firstName, lastName or email.

<> CreateUser

- Create a page and component where new users can be created.
- Add appropriate routes.

<> CreateGroup

- Create a page and component where new groups can be created.
- Add appropriate routes.

<> Assign memberships

- Update UserDetails to allow assigning memberships to groups for the specific user.
- Update GroupDetails to allow assigning members to the specific group.

Resources

- https://nx.dev/getting-started
- https://react.dev
- https://developer.mozilla.org
- https://www.mockaroo.com
- https://github.com/pmndrs/zustand
- https://tanstack.com/query
  - https://tkdodo.eu/blog/practical-react-query
- https://github.com/streamich/react-use
- https://emotion.sh
- https://github.com/RicoSuter/NSwag
- https://vitest.dev