PRACTICAL REACT WITH TYPESCRIPT

bouvet

Setup

- Install
  - Node LTS - https://nodejs.org/en
    - Verify: `node –v`
    - Verify npm: `npm -v`
  - Git - https://git-scm.com
    - Verify: `git -v`
  - Visual Studio Code - https://code.visualstudio.com
    - Verify: `code -v`
  - Browser Extension: React Developer Tools

- 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
  - Tooling

Agenda

Day 1

- Components and JSX
- Props and State
- Events
- Lifecycle

Day 2

- Loops
- Styling
- Component composition
- Routing

Day 3

- Organizing repository
- Server communication
- Testing
- Code-splitting

Set up our workspace

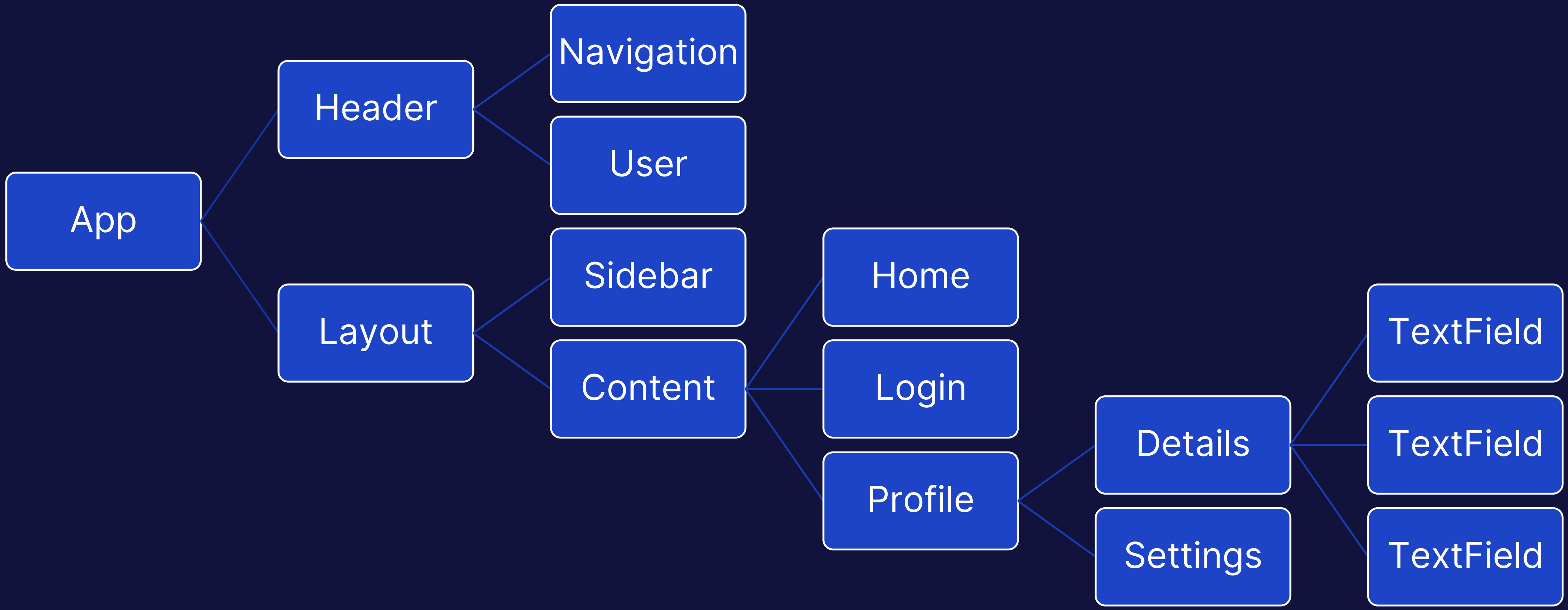
- https://nx.dev/
- `npx create-nx-workspace@latest`

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.
- Allow customizing the label.

<> 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.
- Allow customizing the label.

<> NumericField

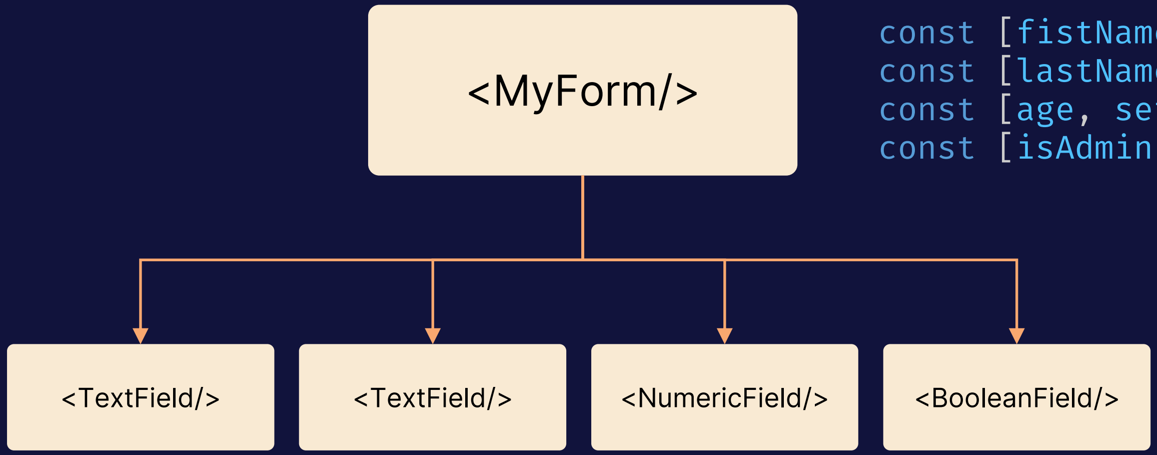
- Create an input field component for inputting numeric values.
- It should have an input field and a label like the TextField.
- Label should be configurable.
- The following parameters should be configurable as props.
  - A minimum value (default 0)
  - A maximum value (default 100)
  - Whether or not decimals are allowed (default false)
- If $|max-min| <= 50$ and decimals are not allowed use “range” input.
- If range input is used display the value right after the range selector.

<> 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.

Hoisting state

- Move state “up” to the component where it makes logical sense.
- Pass state down through props to modify



```
const [fistName, setFirstName] = useState("")
const [lastName, setLastName] = useState("")
const [age, setAge] = useState(0)
const [isAdmin, setIsAdmin] = useState(0)
```

Hoist state out of TextField

- Modify the TextField so that the live value and the setter are provided from props and not internal state.

<> ClickUntilForm

- Create a component ClickUntilForm that will allow the user to control properties of the ClickUntil component.
- Modify NumericField and BooleanField so that they «hoist» their state.
- Use fields and state so the user can control:
  - The message that appears once the clicks reach the limit.
  - The limit.
  - A checkbox that, when checked, allows clicking past the limit while the message is still displayed.

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

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

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

```
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

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

```
export interface TextFieldProps {
  label: string
}

ex                                             dProps) => {
  const id
  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) => {
```

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("")

  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

```
export interface TextFieldProps {
  label: string
}

exp
  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

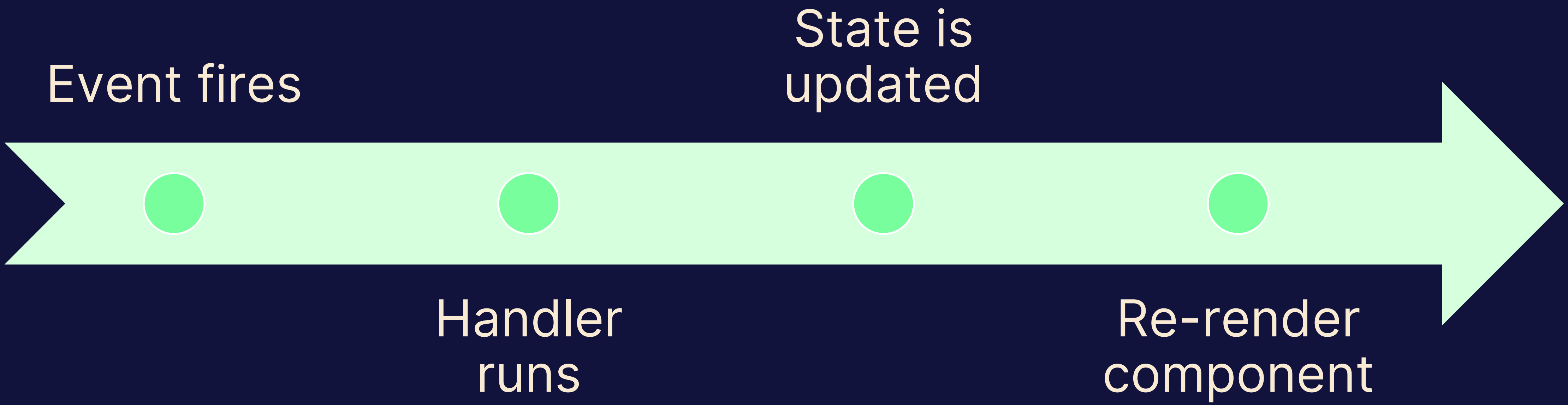
```
export interface TextFieldProps {
  label: string
}

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

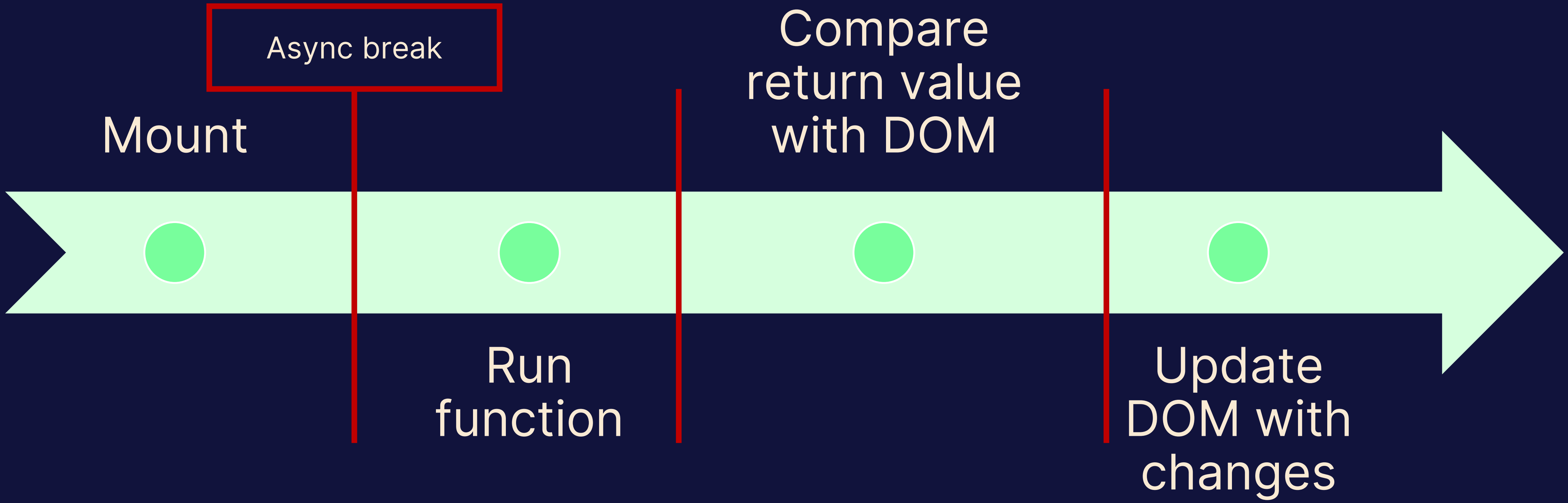


Component lifecycle

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

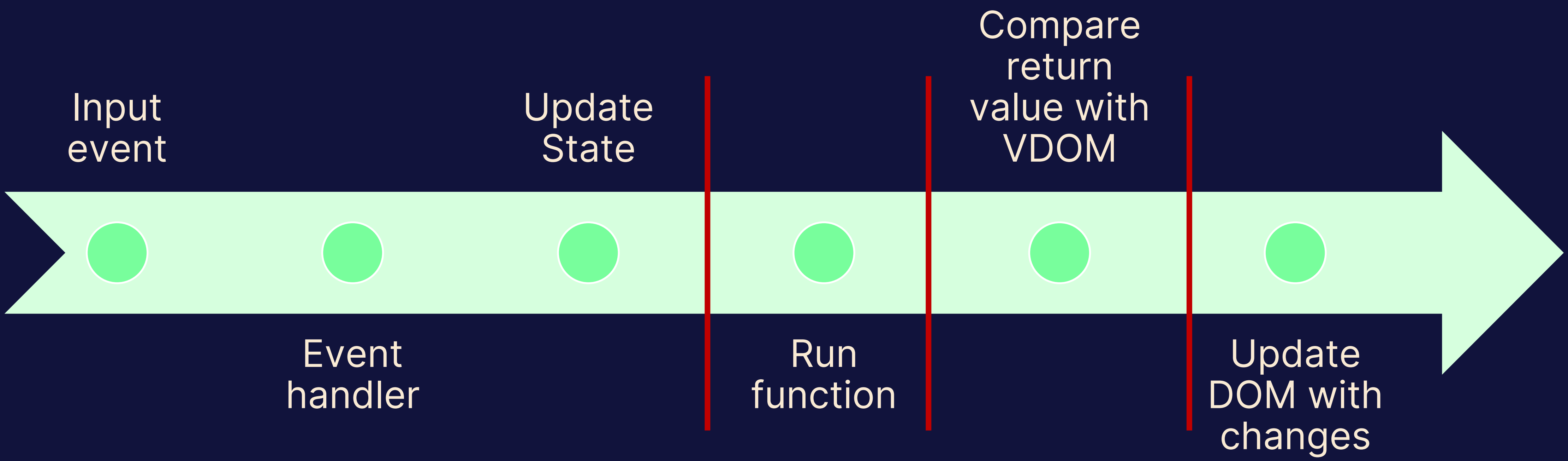
Component lifecycle

TextField



Component lifecycle

TextField



Loops

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

Immutability

- React works on the assumption that objects are immutable.
- An immutable object cannot change it can only be replaced.
- Optimize for performance.

EoD 1

ListProductNames

- Copy 20 products from the API for this task.
- List product names in an unordered list.
- When an item is clicked highlight it and display it above the list, include the items index.
- Add a button that sorts the names in ascending or descending order.
- Add a delete button to each item that removes it when clicked.

<> ProductsTable

- Use products from API for this task.
- List products in a table with the following columns:
  - Title
  - Category
  - Price
  - Rating and number of ratings
- Add ability to delete a product.

<> ProductsTable2

- Add ability to click a column and sort by that column.
- Clicking the column again should reverse the sorting direction.
- Show which column is currently being sorted with direction.

Styling

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

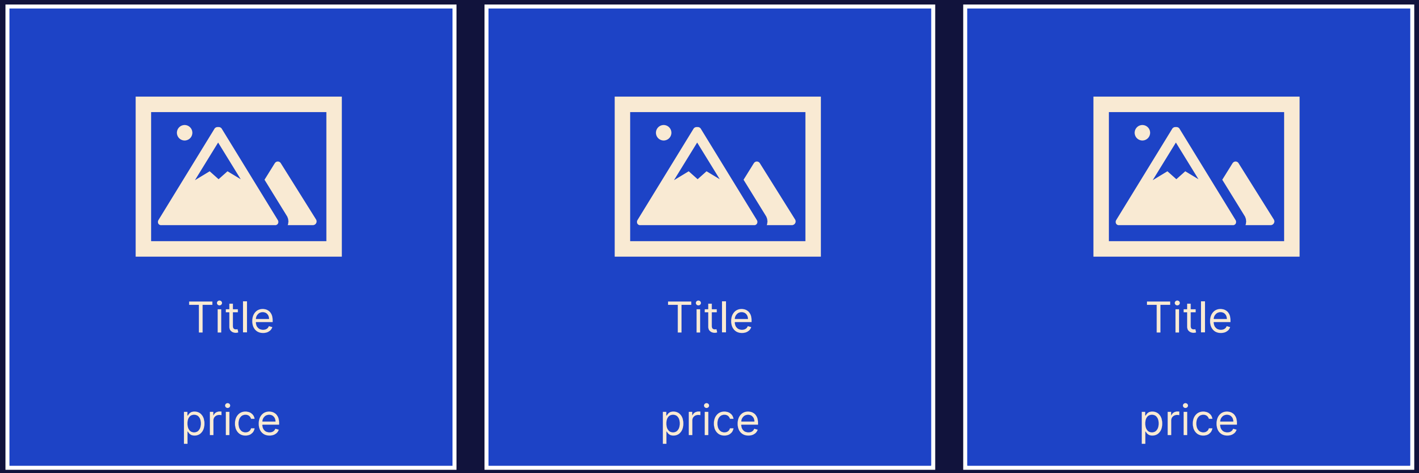
<> Style Fields

- TextField
  - Display label above input.
  - Create a container div and add some padding.
  - The input field takes up all available width.
- BooleanField
  - Display label after the checkbox with some spacing between them.
  - Add container div.
- NumericField
  - Display label above input.
  - Create a container div and add some padding.
  - The input field takes up all available width.
  - Bonus: Style the value so that it reserves space for the maximum possible number of digits.

<> PromotedProducts

bouvet

- Create a component that displays products in a set of boxes next to each other.
- You can use https://picsum.photos/ to get pictures for each product.



Component composition

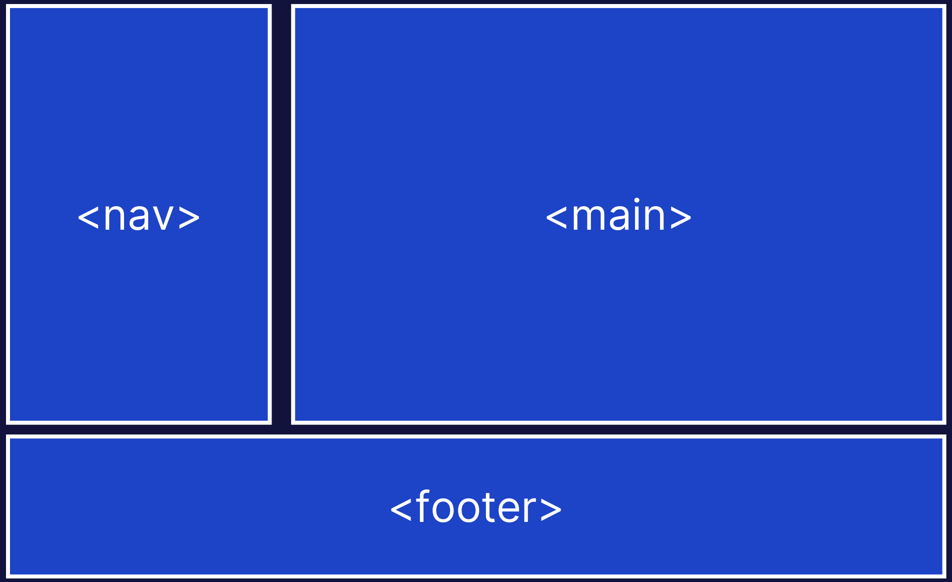
- Pass components as props to other components.
- Replaces most use cases for the old “higher-order components” pattern.

EmphasizeComponent

- Create a component that “emphasizes” another component using styles.
- Emphasized components should be obvious to the user.

<> MainLayout

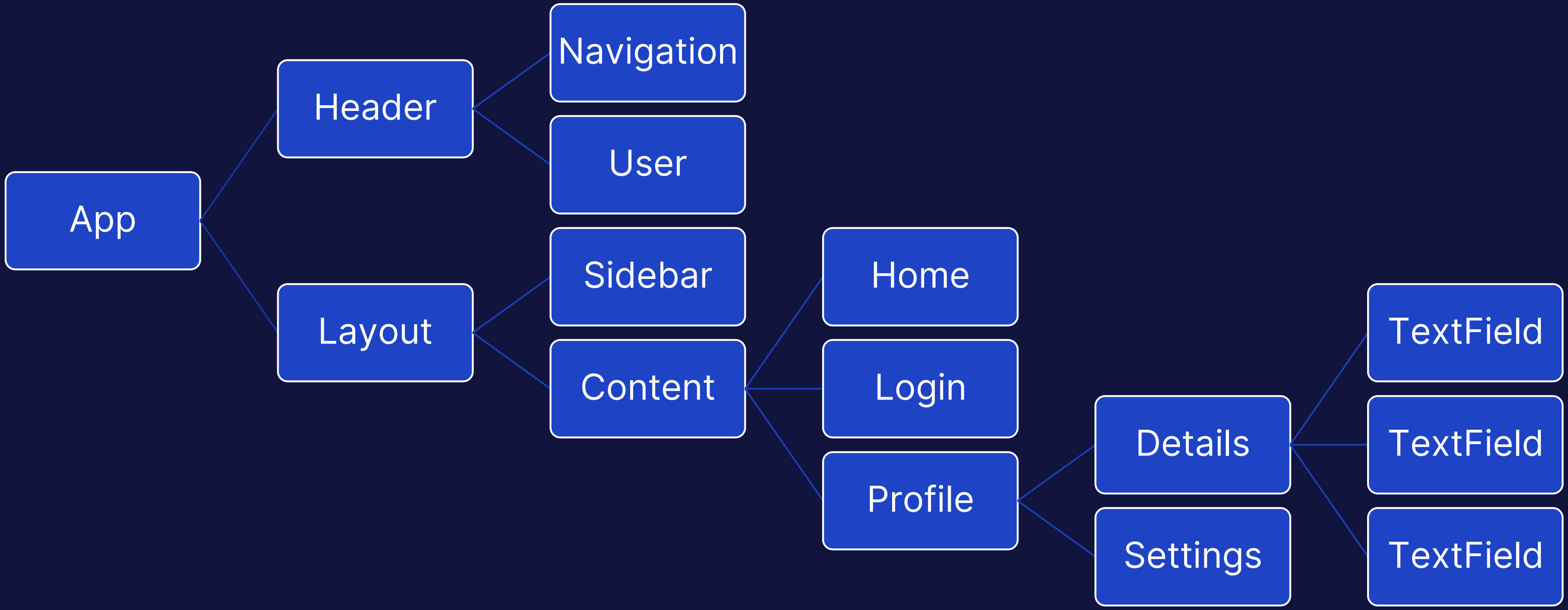
- Create a MainLayout component for the layout illustrated below.
- Allow composing the navigation, main content and footer.



Routing

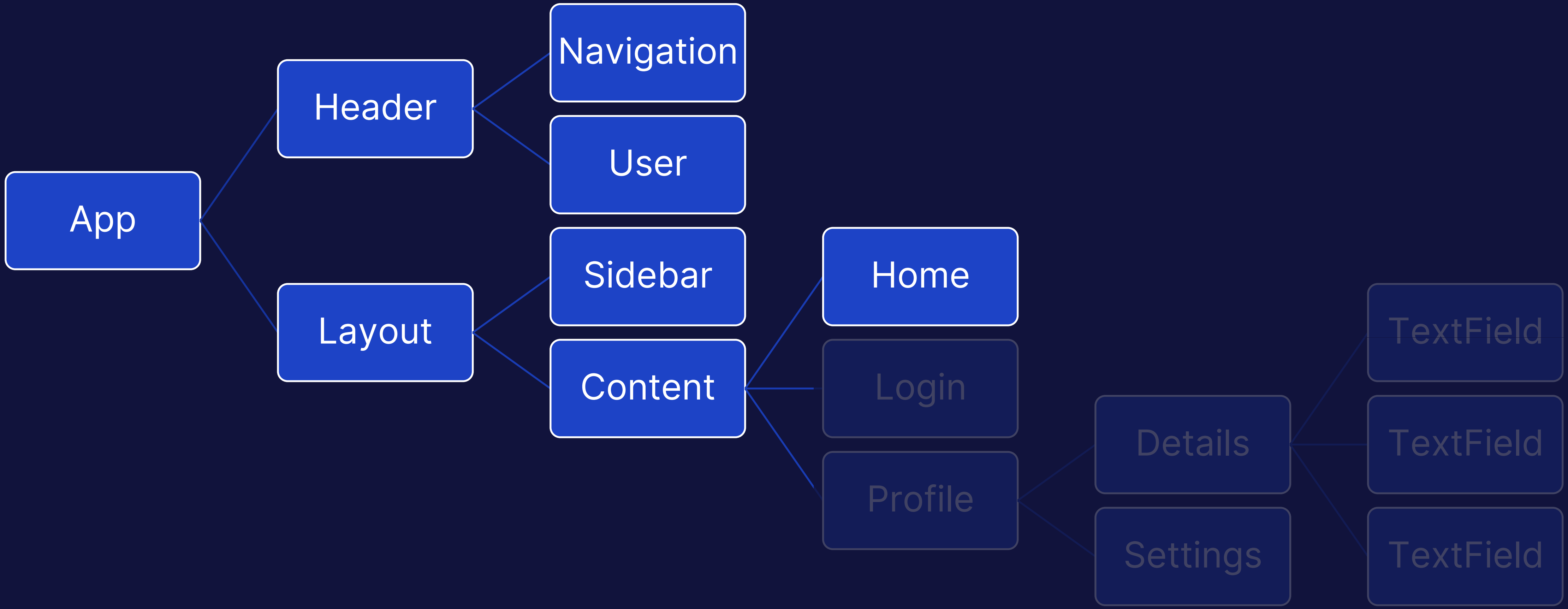
- Use location as state.
- Filter component tree based on location.

Anatomy of routing



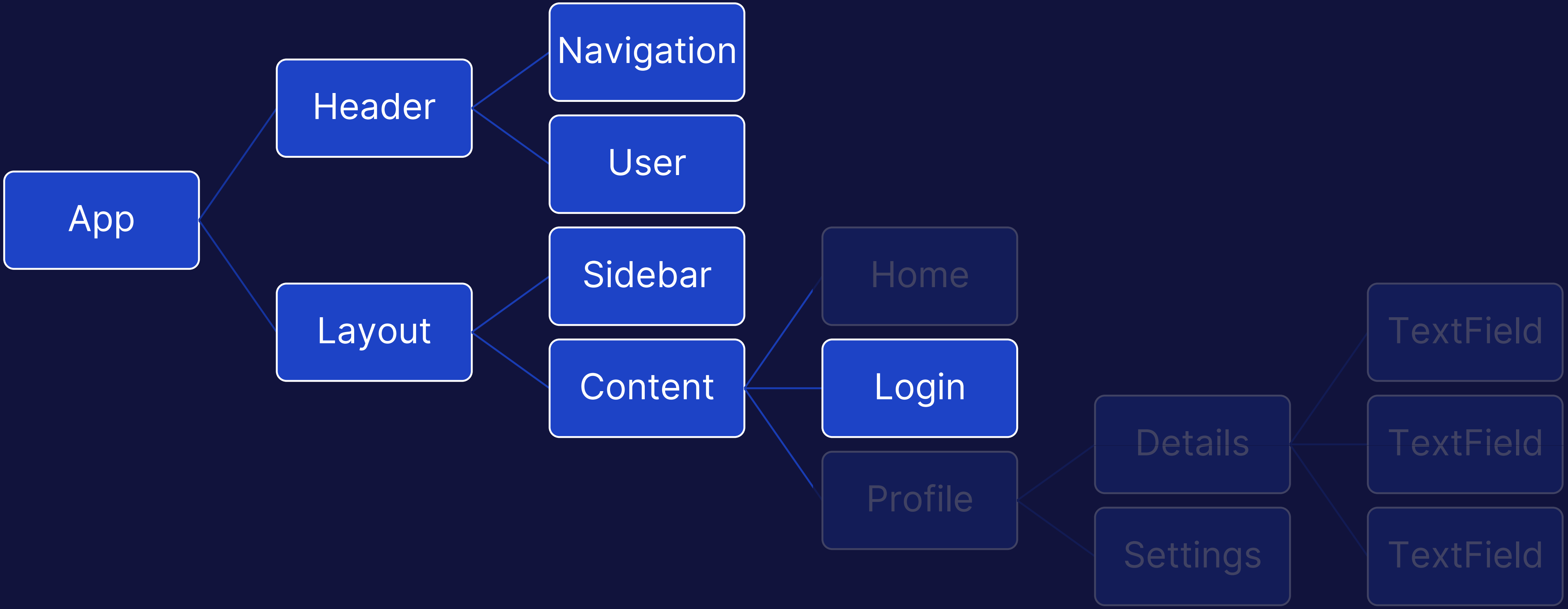
Anatomy of routing

/home



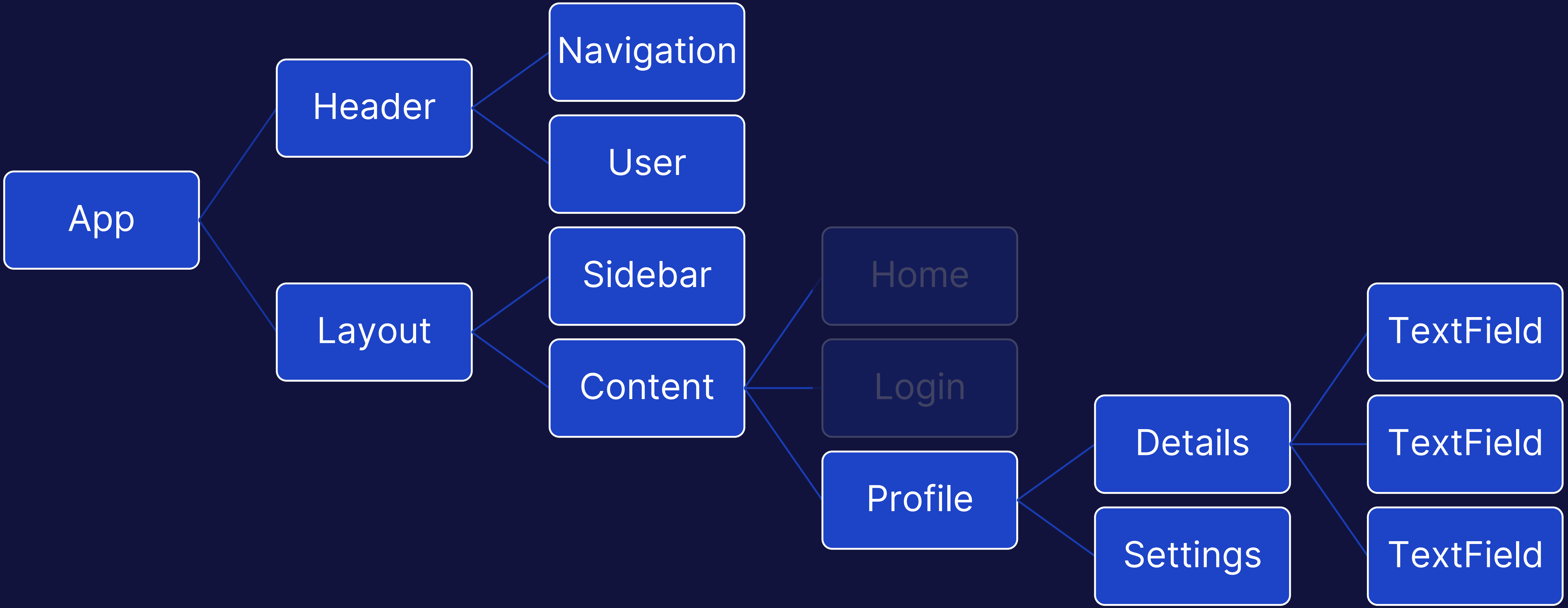
Anatomy of routing

/login



Anatomy of routing

/profile



Routing parameters

- Parameters can be extracted from routes and used as input in components.
- Paths can contain many parameters.
- Examples:
  - users/:userId
  - products/:productId/details
  - books/:bookId/pages/:pageId/word/:wordId

ProductDetailsPage

- Create a page that displays product information based on a product id specified as a route parameter.

<> HomePage and Nav

- Create a page that displays 3 promoted products.
- Use it as the root page for the app.

<> ProductsPage

- Create a products page that displays all products as a table.
- Add links for each product in the table that points to the product details page for that product.

Code-splitting

- Load pieces of the app when needed.
- Reduce initial bundle size.

EoD 2

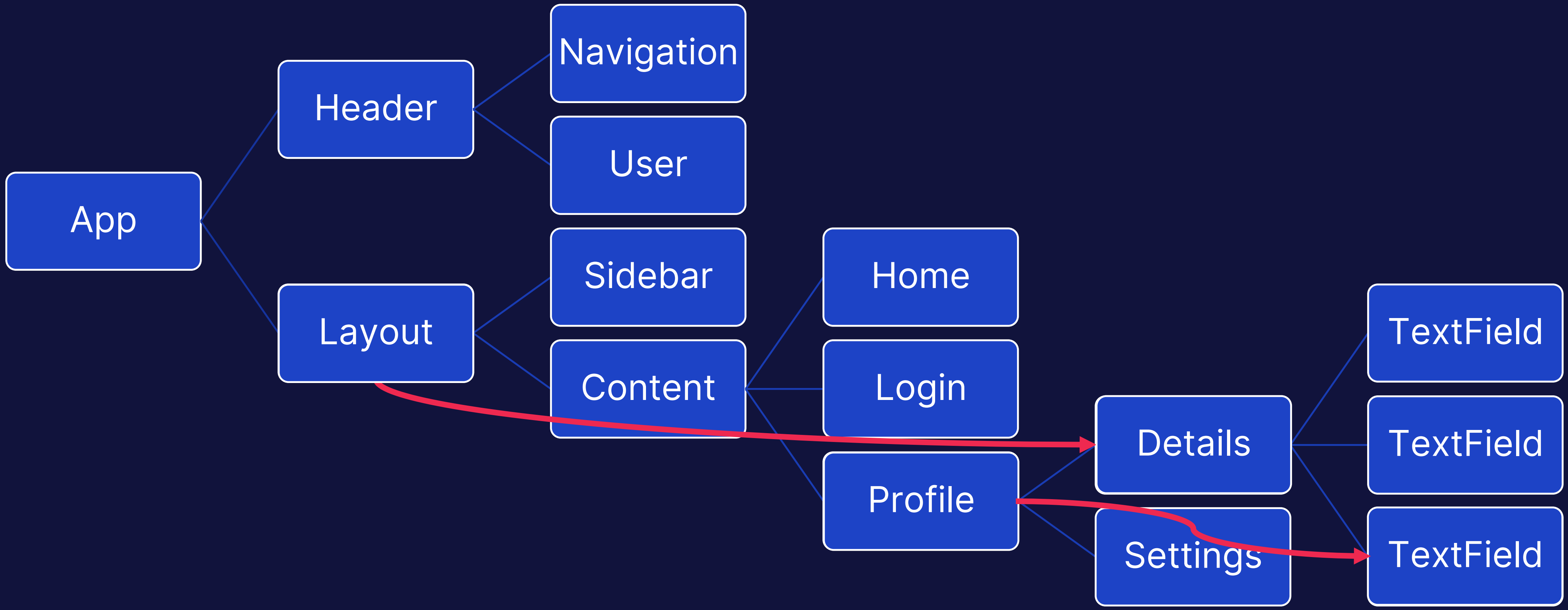
Organizing our repository

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

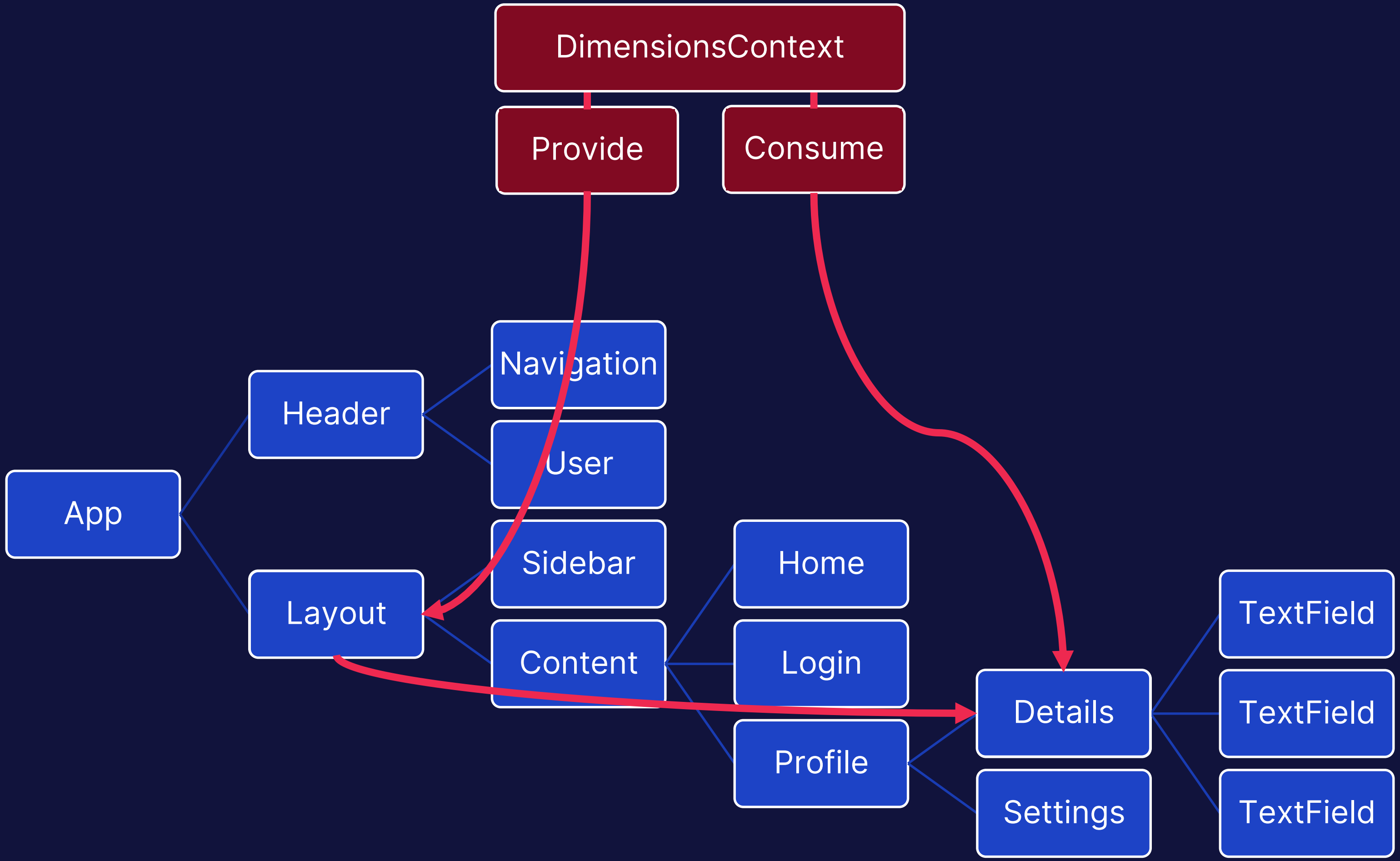
Contexts

- 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



Anatomy of 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

AuthContext

- Create a context for handling authentication. It should contain:
  - Information about the current state: is authenticated, username and role
  - A login function
  - A logout function
- Create a basic login form and page that logs a user in and updates the context.

<> DisableFieldsContext

- Create a context that disables every field.
- Provide a way to toggle disabled state through the context.
- Update fields to use (consume) the context.

<> FieldsService

- Create a service that makes it possible to disable every field below it.
- Provide a way to toggle disabled state through the service.
- Update fields to use use it.

Communication with a server

- Get data from a server and send updates back.
- Use a library to aid in state management and caching.
- Tanstack Query (formerly React 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 service.

ListProducts from server

- Update the ListProducts component to get products from the server using React Query.

<> ProductDetails from server

- Update the ProductDetails component to get product details from the server using React Query.
- Display inventory for each product on the details page.

<> ProductsSearch

- Create a component that allows the user to search for products.
- Add filters and parameters as needed.
- Display inventory for each product.

Mutating state on the server

- Handled through Tanstack Query “Mutations”

<> LogInComponent

- Create a component that can log a user in.
- Update the UserSessionContext to contain information about the logged in user.
- Add a logout button in the footer that also clears the UserSessionContext.

<> ProductDetails inventory

- Update the ProductDetails component to allow warehouse admins to change the inventory for a product.
- Persist the changes on the server.

Effective Query Management

- Move “data functions” to their own files and create custom hooks.
- Put parameters of the query in query keys.
  - Use Query Key factories.

- Read: Practical React Query | TkDodo's blog

Error boundaries

- “Catch” errors thrown by components.
- Can catch errors from React Query as well.

Resources

- https://react.dev - React library home page
- https://immerjs.github.io/immer/ - Helps with immutable objects
- https://nx.dev/getting-started - Monorepo utilityhttps://developer.mozilla.org
- https://www.mockaroo.com - Tool for generating test data
- https://picsum.photos/ - Generate dummy images
- https://github.com/pmndrs/zustand - Simple global state manager
- https://tanstack.com/query - Server-state cache and orchestration library
  - https://tkdodo.eu/blog/practical-react-query
- https://github.com/streamich/react-use - A bunch of useful hooks
- https://emotion.sh - CSS-in-JS styling library
- https://prettier.io - An opinionated code formatter
- https://eslint.org - Linting tool for enforcing coding standards.
- https://github.com/RicoSuter/NSwag - Tool for generating TypeScript (and other) clients from OpenApi
- npm audit (BlackDuck, SonarCloud, Snyk) - Security utilities