React Workshop

Michael Merrill and **Carlos Paelinck**

Morning Session

Render Props

Forms

Context

Afternoon Session

GraphQL

Routing

Animations

Morning Session

Render Props

Forms

Context

Afternoon Session

GraphQL

Routing

Animations

State in **React**

React has fantastic built-in APIs for state management: stateful components and Context.

Key Points about **State**

Key Points about **State**

State flows down in the React component tree.

Key Points about **State**

State flows down in the React component tree.

State is encapsulated. Parents and children have no idea if a component is stateful or not.

Key Points about **State**

State flows down in the React component tree.

State is encapsulated. Parents and children have no idea if a component is stateful or not.

A component can pass its state down to a child as a prop.

Key Points about **State**

State flows down in the React component tree.

State is encapsulated. Parents and children have no idea if a component is stateful or not.

A component can pass its state down to a child as a prop.

Updating state only affects any UI that is derived from that state or child components that use it below in the tree.

Render Props

A Render Prop is a component whose prop takes a function that renders a component.

It’s a good way to encapsulate behavior and provide values to rendered components.

Using a **Render Prop**

Using a **Render Prop**

```
<List
  data={dogs}
```

Using a **Render Prop**

```
<List
  data={dogs}
  renderTitle={() => <h1>Dogs</h1>}
```

Using a **Render Prop**

```
<List
  data={dogs}
  renderTitle={() => <h1>Dogs</h1>}
  renderItems={dog => (
    <strong>{ dog.name }</strong>
  )}
/>
```

```
<MousePosition
  render={(x, y) => (
    <p>Mouse at { x }, { y }</p>
  )}
/>
```

Creating a **Render Prop**

```
class MousePosition extends React.Component {
  state = { x: 0, y: 0 };
  handleMouseMove = event => this.setState({
    x: event.clientX, y: event.clientY
  });
```

```
class MousePosition extends React.Component {
  state = { x: 0, y: 0 };
  handleMouseMove = event => this.setState({
    x: event.clientX, y: event.clientY
  });

  render() {
    return (
      <div onMouseMove={this.handleMouseMove}>
```

```
class MousePosition extends React.Component {
  state = { x: 0, y: 0 };
  handleMouseMove = event => this.setState({
    x: event.clientX, y: event.clientY
  });

  render() {
    return (
      <div onMouseMove={this.handleMouseMove}>
        {this.props.render({
          this.state.x, this.state.y })}
      </div>
    );
  }
}
```

Forms

Formik

Most apps have forms which require managing transient state, validation, and submission.

Formik helps bring clarity and structure to forms in React and uses the Render Prop pattern.

Formik also provides functions that help improve the UX of your form.

<Formik

```
<Formik
  initialValues={{ name: "", password: "" }}
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
  render={({ values, handleChange, handleSubmit }) => (
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
  render={({ values, handleChange, handleSubmit }) => (
    <form onSubmit={handleSubmit}>
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
  render={({ values, handleChange, handleSubmit }) => (
    <form onSubmit={handleSubmit}>
      <input
        name=“name”
        onChange={handleChange}
        value={values.name}
      />
      ...
    </form>
  )}
/>
```

Formik supports validation schemas with Yup.

```
<Formik
 onSubmit={async (values, props) => {
```

```
<Formik
 onSubmit={async (values, props) => {
   props.setSubmitting(true);
```

```
<Formik
 onSubmit={async (values, props) => {
   props.setSubmitting(true);
   try {
     await postFormToAPI(values);
```

```
<Formik
 onSubmit={async (values, props) => {
   props.setSubmitting(true);
   try {
     await postFormToAPI(values);
   catch (err) {
     props.setErrors({ name: err.name.message });
```

```
<Formik
 onSubmit={async (values, props) => {
   props.setSubmitting(true);
   try {
     await postFormToAPI(values);
   catch (err) {
     props.setErrors({ name: err.name.message });
   }
   props.setSubmitting(false);
}}
```

```
<Formik
 onSubmit={async (values, props) => {
   props.setSubmitting(true);
   try {
     await postFormToAPI(values);
   catch (err) {
     props.setErrors({ name: err.name.message });
   }
   props.setSubmitting(false);
}}
render={({ isSubmitting, errors }) => (
```

```
<Formik
 onSubmit={async (values, props) => {
   props.setSubmitting(true);
   try {
     await postFormToAPI(values);
   catch (err) {
     props.setErrors({ name: err.name.message });
   }
   props.setSubmitting(false);
}}
render={({ isSubmitting, errors }) => (
   ...
   <button disabled={isSubmitting}>
```

```
import * as yup from “yup”;
```

```
import * as yup from “yup”;

const validator = yup
```

```
import * as yup from “yup”;

const validator = yup
  .string()
```

```
import * as yup from “yup”;

const validator = yup
  .string()
  .email()
```

```
import * as yup from “yup”;

const validator = yup
  .string()
  .email()
  .isRequired(“A valid email address is required”);
```

```
import * as yup from “yup”;

const validator = yup
  .string()
  .email()
  .isRequired(“A valid email address is required”);
```

```
import * as yup from “yup”;

const validator = yup
  .string()
  .email()
  .isRequired(“A valid email address is required”);

async function validateEmail(email) {
   try {
     const valid = await validator(email);
```

```
import * as yup from “yup”;

const validator = yup
  .string()
  .email()
  .isRequired(“A valid email address is required”);

async function validateEmail(email) {
    try {
      const valid = await validator(email);
    } catch (err) {
      return err;
    }
}
```

```
const validator = yup.object().shape({
```

```
const validator = yup.object().shape({
  password: yup
    .string()
    .isRequired(“A password is required”),
```

```
const validator = yup.object().shape({
  password: yup
    .string()
    .isRequired(“A password is required”),
  confirmPassword: yup
    .string()
```

```
const validator = yup.object().shape({
  password: yup
    .string()
    .isRequired(“A password is required”),
  confirmPassword: yup
    .string()
    .oneOf([yup.ref(“password”)],
      “Passwords do not match”)
```

```
const validator = yup.object().shape({
  password: yup
    .string()
    .isRequired("A password is required"),
  confirmPassword: yup
    .string()
    .oneOf([yup.ref("password")],
      "Passwords do not match")
    .isRequired("A password confirmation is
      required")
});
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
  render={({ values, handleChange, handleSubmit }) => (
    <form onSubmit={handleSubmit}>
      <input
        name=“name”
        onChange={handleChange}
        value={values.name}
      />
      ...
    </form>
  )}
/>
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
  validationSchema={validator}
  render={({ values, handleChange, handleSubmit }) => (
    <form onSubmit={handleSubmit}>
      <input
        name=“name”
        onChange={handleChange}
        value={values.name}
      />
      ...
    </form>
  )}
/>
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
  validationSchema={validator}
  isInitialValid={false}
  render={({ values, handleChange, handleSubmit }) => (
    <form onSubmit={handleSubmit}>
      <input
        name=“name”
        onChange={handleChange}
        value={values.name}
      />
      ...
    </form>
  )}
/>
```

```
<Formik
  initialValues={{ name: “”, password: “” }}
  onSubmit={values => this.submitForm(values)}
  validationSchema={validator}
  render={({ errors }) => (
    ...
    <div>
     { errors.name }
     { errors.password }
    </div>
    ...
  )}
/>
```

Context

Context lets you pass data and actions throughout your component tree without manually having to pass as props at every level.

This makes it easier to share a unified state and actions with many components that may not be direct descendants.

Prior to React's new Context APIs many apps used Redux for centralized state management.

Creating Context

```
import React from “react”;

const AnimalContext = React.createContext();
```

Creating Context

```
import React from “react”;

const AnimalContext = React.createContext();


const AnimalContext = React.createContext({
  name: “Milo”
});
```

Provider Components wrap your App's component subtree and are the source of truth for state and provide actions.

Using a **Provider**

Using a **Provider**

```
const AnimalContext = React.createContext();
```

Using a **Provider**

```
const AnimalContext = React.createContext();

function WrappedApp() {
  return (
```

Using a **Provider**

```
const AnimalContext = React.createContext();

function WrappedApp() {
  return (
<AnimalContext.Provider value={{
  name: “Milo”
}}>
```

Using a **Provider**

```
const AnimalContext = React.createContext();

function WrappedApp() {
  return (
    <AnimalContext.Provider value={{
      name: “Milo”
    }}>
      <App />
```

Using a **Provider**

```
const AnimalContext = React.createContext();

function WrappedApp() {
  return (
    <AnimalContext.Provider value={{
      name: “Milo”
    }}>
      <App />
    </AnimalContext.Provider>
  );
}
```

```
class WrappedApp extends React.Component {
  state = { name: “Milo” };
  handleChangeName = name =>
    this.setState({ name });
```

```
class WrappedApp extends React.Component {
  state = { name: “Milo” };
  handleChangeName = name =>
    this.setState({ name });

  render() {
    return (
      <AnimalContext.Provider value={{
  name: this.state.name,
  changeName: this.handleChangeName
}}>
```

```
class WrappedApp extends React.Component {
  state = { name: “Milo” };
  handleChangeName = name =>
    this.setState({ name });

  render() {
    return (
      <AnimalContext.Provider value={{
        name: this.state.name,
        changeName: this.handleChangeName
      }}>
        <App />
```

```
class WrappedApp extends React.Component {
  state = { name: “Milo” };
  handleChangeName = name =>
    this.setState({ name });

  render() {
    return (
      <AnimalContext.Provider value={{
        name: this.state.name,
        changeName: this.handleChangeName
      }}>
        <App />
      </AnimalContext.Provider>
    );
  }
}
```

Consumer Components wrap any descendants in your App's tree and let you use any state or actions you've provided as a value.

Using a **Consumer**

Using a **Consumer**

```
import { AnimalContext } from “./contexts”;
```

Using a **Consumer**

```
import { AnimalContext } from “./contexts”;

function DogCard() {
  return (
```

Using a **Consumer**

```
import { AnimalContext } from “./contexts”;

function DogCard() {
  return (
    <AnimalContext.Consumer>
```

Using a **Consumer**

```
import { AnimalContext } from "./contexts";

function DogCard() {
  return (
    <AnimalContext.Consumer>
      {({ name }) => (
```

Using a **Consumer**

```
import { AnimalContext } from “./contexts”;

function DogCard() {
  return (
    <AnimalContext.Consumer>
      {({ name }) => (
        <h1>{ name }</h1>
      });
```

Using a **Consumer**

```
import { AnimalContext } from “./contexts”;

function DogCard() {
  return (
    <AnimalContext.Consumer>
      {({ name }) => (
        <h1>{ name }</h1>
      });
    </AnimalContext.Consumer>
  );
}
```

```
function DogCard() {
  return (
    <AnimalContext.Consumer>
      {({ changeName }) => (
        <button
          onClick={() => changeName(“Beau”)}
        >
          Change Name
        </button>
      });
    </AnimalContext.Consumer>
  );
}
```

GraphQL

URQL

What is **GraphQL?**

GraphQL is a query language for your data.

A query is a way to get data from your GraphQL service.

Queries

A query is a way to get data from your GraphQL service.

```
query {
  dogs {
    name
    type
  }
}
```

```
query {
  dogs {
    name
    type
  }
}
```

```
[
  {
    name: “Milo”,
    type: “Chihuahua”,
  },
  {
    name: “Beau”,
    type: “Chocolate Lab”,
  },
]
```

Mutations

A mutation is how you create or edit data.

```
mutation($name: String!, $type: String!) {
  createDog(name: $name, type: $type) {
    name
    type
  }
}
```

```
mutation($name: String!, $type: String!) {
  createDog(name: $name, type: $type) {
    name
    type
  }
}
```

Always return the entity you are mutating!

URQL provides everything you need to use GraphQL with your React apps with a simple API.

Creating a client and connecting your app

```
import { Client } from “urql”;
```

```
import { Client } from “urql”;
const client = new Client({
```

```
import { Client } from “urql”;

const client = new Client({
  url: “https://www.myapi.com/graphql"
});
```

```
import { Client } from “urql”;

const client = new Client({
  url: “https://www.myapi.com/graphql",
  fetchOptions: () => ({
    “Content-Type" : “application/json",
    “Authorization” : getMyAuthToken()
  })
});
```

```
import { Client, Provider } from “urql”;
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });

function WrappedApp() {
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });

function WrappedApp() {
  return (
    <Provider client={client}>
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });

function WrappedApp() {
  return (
    <Provider client={client}>
      <App />
    </Provider>
  );
};
```

```
import { Client, Provider } from “urql”;
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });

function WrappedApp() {
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });

function WrappedApp() {
  return (
    <Provider client={client}>
```

```
import { Client, Provider } from “urql”;

const client = new Client({ ... });

function WrappedApp() {
  return (
    <Provider client={client}>
      <App />
    </Provider>
  );
};
```

Creating queries and mutations for your app

```
const dogsQuery = `
  query {
    dogs {
      name
        type
    }
  }
`;
```

```
const createDogMutation = `
  mutation($name: String!, $type: String!) {
    createDog(name: $name, type: $type) {
      name
      type
    }
  }
`;
```

Using queries and mutations in your components

```
import { Connect, query } from “urql”;
```

```
import { Connect, query } from “urql”;

function DogsView() {
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      query={query(dogQuery)}
    >
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      query={query(dogQuery)}
    >
      ({ data, fetching, loaded }) => (
        ...
      ))
    </Connect>
  );
}
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      query={
        [query(dogQuery), query(catQuery)]
      }
    >
      ({ data, fetching, loaded }) => (
        ...
      ))
    </Connect>
  );
}
```

```
import { Connect, query } from “urql”;
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      mutations={{
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      mutations={{
        createDog: mutation(createDogMutation)
      }}
    >
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      mutations={{
        createDog: mutation(createDogMutation)
      }}
    >
      ({ createDog }) => (
        ...
      ))
    </Connect>
  );
}
```

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      mutations={{
        createDog: mutation(createDogMutation),
        feedDog: mutation(feedDogMutation),
      }}
    >
      ({ createDog, feedDog }) => (
        ...
      ))
    </Connect>
  );
}
```

URQL will automatically update the data from queries when mutations are called.

```
import { Connect, query } from “urql”;

function DogsView() {
  return (
    <Connect
      queries={query(getDogsQuery)}
      mutations={{
        createDog: mutation(createDogMutation)
      }}
    >
      ({ dogs, createDog }) => (
        ...
      ))
    </Connect>
  );
}
```

Routing

Reach Router

Reach Router is a simple way to add routing to a React app.

```
import { Router } from “@reach/router”;
```

```
import { Router } from “@reach/router”;

const IndexView = () => <div />;
const DetailView = () => <div />;
```

```
import { Router } from “@reach/router”;

const IndexView = () => <div />;
const DetailView = () => <div />;

function App() {
  return (
    <Router>
```

```
import { Router } from “@reach/router”;

const IndexView = () => <div />;
const DetailView = () => <div />;

function App() {
  return (
    <Router>
      <IndexView path=“/“ />
      <DetailView path=“detail” />
    </Router>
  );
}
```

```
import { Router } from “@reach/router”;

const IndexView = () => <div />;
const DetailView = () => <div />;

function App() {
  return (
    <Router>
      <IndexView path=“/“ />
      <DetailView path=“detail/:id” />
    </Router>
  );
}
```

```
import { Link } from “@reach/router”;

function NavigationBar() {
  return (
    <div>
      <Link to=“/“>Home</Link>
      <Link to=“detail”>Detail</Link>
    </div>
  );
}
```

```
import { Link } from “@reach/router”;

function NavigationBar() {
  return (
    <div>
      <Link to=“/“>Home</Link>
      <Link to=“detail/abc-123”>Detail</Link>
    </div>
  );
}
```

```
function DetailView(props) {
  return (
    <div>
      {props.id}
    </div>
  );
}
```

Since params are part of the URL they must be serializable strings.

Do not stringify objects and use that as a param.

Use app-based state management for instead.

```
import { Router } from “@reach/router”;

function App() {
  return (
    <Router>
      <IndexView path=“/“ />
      <Dashboard path=“dashboard”>
        <About path=“about”>
        <Contact path=“contact”>
      </Dashboard>
    </Router>
  );
}
```

```
    return (
      <Router>
        <IndexView path="/" />
        <Dashboard path="dashboard">
          <About path="about">
          <Contact path="contact">
        </Dashboard>
      </Router>
    );
  }
```

Routing to "`dashboard/about`" will render both Dashboard and About. It is important that your Dashboard component can correctly render the child component.

You can also navigate to routes programmatically.

```
import { navigate } from “@reach/router”;

function NavigationBar(props) {
  return (
    <button onPress={() =>
        navigate(`details/${props.id}`)
      }
    />
      Go To Detail View
    </button>
  );
}
```

```
import { navigate } from “@reach/router”;

function NavigationBar(props) {
  return (
    <button onPress={async () => {
        await navigate(`details/${props.id}`);
        await fetchData(props.id);
      }}
    />
      Go To Detail View
    </button>
  );
}
```

Animations

Pose

Pose lets you create animatable reusable components.

Think of it as the animated version of Styled Components.

Pose can animate any standard DOM elements—including SVG.

```
import posed from “pose”;
```

```
import posed from “pose”;

const Box = posed.div({
```

```
import posed from “pose”;

const Box = posed.div({
  visible: { opacity: 1.0 },
```

```
import posed from “pose”;

const Box = posed.div({
  visible: { opacity: 1.0 },
  hidden: { opacity: 0.0 },
});
```

```
import posed from “pose”;

const Box = posed.div({
  visible: { opacity: 1.0 },
  hidden: { opacity: 0.0 },
});

function View(props) {
  return (
    <Box
```

```
import posed from “pose”;

const Box = posed.div({
  visible: { opacity: 1.0 },
  hidden: { opacity: 0.0 },
});

function View(props) {
  return (
    <Box
     pose={props.show ? “visible” : “hidden”}
    >
     ...
    </Box>
  );
}
```

You can customize Pose animatable components with delays, timing, easing functions, and springs.

```
import posed from “pose”;

const Box = posed.div({
  visible: { opacity: 1.0 },
  hidden: { opacity: 0.0 },
});
```

```
import posed from “pose”;

const Box = posed.div({
  visible: {
    opacity: 1.0,
    transition: { timing: 500 },
  },
  hidden: { opacity: 0.0 },
});
```

```
import posed from “pose”;

const Box = posed.div({
  visible: {
    opacity: 1.0,
    transition: {
      timing: 500, ease: “linear”
    },
  },
  hidden: { opacity: 0.0 },
});
```

```
import posed from “pose”;

const Box = posed.div({
  visible: {
    opacity: 1.0,
    transition: {
      timing: 500, ease: “linear”, delay: 250
    },
  },
  hidden: { opacity: 0.0 },
});
```

```
import posed from “pose”;

const Box = posed.div({
  open: {
    width: “100%”,
    transition: {
      type: "spring", stiffness: 70
    },
  },
  closed: { width: 0 },
});
```

Pose can even animate CSS transitions.

```
import posed from “pose”;
```

```
import posed from “pose”;

const Box = posed.rect({
  visible: {
    opacity: 1.0,
```

```
import posed from “pose”;

const Box = posed.rect({
  visible: {
    opacity: 1.0,
    transform: “translate(100, 100)”,
  },
```

```
import posed from “pose”;

const Box = posed.rect({
  visible: {
    opacity: 1.0,
    transform: “translate(100, 100)”,
  },
  hidden: {
    opacity: 0.0,
    transform: “translate(0, 0)”,
  },
});
```