NEWS · OCTOBER 30, 2025

React Components: Understanding the Web UI Model (Part 3 of 5)

By Dr. Holger Flick

> **Update (July 2026): the full series recap — with proof**
> This series now has a companion post that condenses all five parts into one page and pairs them with real-world 2026 case studies — including a production Delphi multi-tier app rebuilt on Next.js in about four hours: Delphi to Next.js: The Complete Series, and the Proof It Works.

The Conceptual Shift

Remember the first time you understood how Delphi's component model worked? You could drop a `TButton` on a form, set its properties, write an `OnClick` handler, and suddenly you had a working interface. Components were reusable. You could build custom components and use them everywhere. The VCL made desktop development intuitive.

React is the same idea for web applications. Instead of desktop controls, you have web components. Instead of forms, you have pages. But the fundamental concept—building interfaces from composable, reusable pieces—is identical.

This article isn't about teaching you to write React code. It's about helping you understand the mental model so you can evaluate whether your Delphi applications can translate to this approach.

Understanding Components

In Delphi, you might create a custom panel with a label and button:

```
type
  TCustomerPanel = class(TPanel)
  private
    FCustomerName: string;
    FLabelCaption: TLabel;
    FButtonAction: TButton;
  published
    property CustomerName: string read FCustomerName write SetCustomerName;
  end;
```

In React, a component is just a function that returns what should be displayed:

```
interface CustomerPanelProps {
  customerName: string;
  onActionClick: () => void;
}

function CustomerPanel({ customerName, onActionClick }: CustomerPanelProps) {
  return (
    <div>
      <label>{customerName}</label>
      <button onClick={onActionClick}>Action</button>
    </div>
  );
}
```

The key difference: in Delphi, you're creating an object that exists over time. In React, you're describing what should be rendered given certain inputs. It's a declarative model rather than an imperative one.

Key Concepts (Delphi Translation)

- **Component** = A custom control (like `TCustomerPanel`)
- **Props** = Properties passed in (like `CustomerName` property)
- **State** = Internal fields that can change (like `FSelectedIndex`)
- **Events** = Event handlers (`onActionClick` = `OnButtonClick`)

The biggest mental shift: instead of updating the screen manually (like calling `Label1.Caption := 'New Text'`), you change the data and React automatically updates what's displayed.

State Management - Data That Changes

In Delphi, components have internal fields that change:

```
type
  TCounter = class(TCustomControl)
  private
    FCount: Integer;
  public
    procedure Increment;
  end;

procedure TCounter.Increment;
begin
  Inc(FCount);
  Invalidate; // Trigger repaint
end;
```

React has a similar concept called "state":

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

When `setCount` is called, React automatically re-renders the component with the new value. You don't manually call `Invalidate()`—React handles it.

The key insight: Instead of imperatively updating the UI ("change this label's text"), you declaratively describe what the UI should look like for any given state. When state changes, React figures out what needs to update.

Event Handling - Responding to User Actions

Delphi events:

```
procedure TMyForm.ButtonClick(Sender: TObject);
begin
  ShowMessage('Clicked!');
end;
```

React events are similar:

```
function MyButton() {
  const handleClick = () => {
    alert('Clicked!');
  };

  return <button onClick={handleClick}>Click Me</button>;
}
```

The syntax is different, but the concept is identical: attach a function to an event, and that function runs when the event fires.

Props vs State: The Core Distinction

Props (Properties):

- Data passed from parent to child
- Read-only within the component
- Like Delphi published properties

State (Internal Fields):

- Data managed within the component
- Can be modified by the component
- Like Delphi private fields

In Delphi, you might have:

```
type
  TCustomerEdit = class(TEdit)
  private
    FCustomerId: Integer;  // State (internal)
  published
    property CustomerId: Integer read FCustomerId write FCustomerId; // Can be prop
  end;
```

In React:

```
function CustomerDisplay({ customerId }: { customerId: number }) {
  const [customerName, setCustomerName] = useState('');
  // customerId is a prop (from parent)
  // customerName is state (internal)

  return <div>{customerName}</div>;
}
```

Component Composition - Building Blocks

Just like in Delphi where you compose complex forms from panels, group boxes, and controls, React applications are built by composing components:

```
Page
   Header
      Logo
      Navigation
   Content
      CustomerList
         CustomerCard (repeated)
      Sidebar
   Footer
```

Each piece is a self-contained component that can be reused anywhere.

Lists and Iteration

In Delphi, you might loop through a list to populate a grid:

```
for I := 0 to Customers.Count - 1 do
  AddRow(Customers[I]);
```

In React, you transform arrays directly into UI elements:

```
function CustomerList({ customers }: { customers: Customer[] }) {
  return (
    <div>
      {customers.map(customer => (
        <div key={customer.id}>
          {customer.name}
        </div>
      ))}
    </div>
  );
}
```

The `.map()` function transforms each customer into a component. React handles the rendering.

Conditional Rendering

In Delphi: `Panel1.Visible := IsAdmin;`

In React:

```
{isAdmin && <AdminPanel />}
{hasPermission ? <EditButton /> : <ReadOnlyView />}
```

You describe what should show under what conditions. React figures out when to update the display or which part of it.

What This Means for Your Applications

The React component model maps well to Delphi's approach:

- Both use reusable, composable building blocks
- Both have properties/props and internal state
- Both handle events
- Both encourage separation of concerns

The key difference: React is declarative (describe what should be shown) while Delphi is imperative (tell the computer what to do step by step). This takes some getting used to, but many developers find it leads to cleaner code once they adjust.

Evaluating Your Migration

When looking at your Delphi applications, ask:

- How much of your UI is standard controls that could map to React components?
- How much is custom-drawn or heavily customized?
- How tightly coupled is your UI to your business logic?

Standard CRUD forms translate easily. Complex custom controls might need more thought. The good news: your business logic (calculations, validations, rules) can often be moved almost directly once you understand the TypeScript syntax.

When Professional Help Makes Sense

Understanding React components conceptually is one thing. Architecting a full migration is another. You need to consider:

- Component structure and reusability
- State management across large applications
- Form validation strategies
- Error handling patterns
- Performance optimization
- Testing approaches

These are strategic architectural decisions, not just syntax changes.

I help Delphi teams plan and execute these migrations:

- Analyze your existing forms and data entry screens
- Design a component architecture that fits your domain
- Create reusable component libraries
- Train your team on React patterns and best practices
- Guide the migration of complex business logic
- Establish patterns for your team to follow

The goal is to give your team a solid foundation and clear patterns so they can continue building after the initial migration work.

What's Next

In Part 4, we'll look at the Next.js application structure—how projects are organized, how routing works, and where your business logic lives. You'll see how this compares to Delphi's project structure and how your database operations translate.

We're building toward Part 5, where we'll discuss actual migration strategies: what to migrate first, how to do it incrementally, and when to bring in professional help.

Ready to discuss your specific Delphi applications? I'd be happy to review your forms and screens to give you a realistic assessment of what migration would involve. Even a brief conversation can help you understand the scope and effort required. Contact me!

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —