Lumino: A UI-Independent React Application Framework

"Build enterprise React applications with any design system"

Technical Whitepaper v1.0

Abstract

Lumino is a comprehensive, declarative, UI-independent framework for building enterprise React applications. It provides a complete solution for forms, pages, routing, API integration, state management, and event handling—all while maintaining independence from any specific UI component library. By introducing an abstraction layer between application logic and presentation, Lumino enables organizations to build maintainable, scalable applications that can adapt to any design system without code changes.

This whitepaper presents the architectural decisions, design patterns, complete feature set, and benefits of adopting Lumino for enterprise React applications.

Table of Contents

1. Introduction

Modern enterprise applications require sophisticated capabilities across multiple domains: form handling, page construction, routing, API integration, state management, and event coordination. Traditional approaches to React development often result in fragmented solutions using multiple libraries, each with its own patterns and learning curves.

Lumino addresses these challenges through a unified, declarative framework that treats all application components as first-class entities with their own lifecycle, configuration, and rendering logic—all independent of the underlying UI component library.

Key Features

- **UI Independence**: Write once, render with any design system (Salt DS, Material UI, custom)
- **Declarative Syntax**: Define what your application does, not how it renders
- **Type Safety**: Full TypeScript support with compile-time validation
- **Unified Architecture**: Single framework for forms, pages, routing, API, and state
- **Extensible Adapter System**: Plugin-based adapters for any UI library
- **Comprehensive Validation**: Built-in validators with async and cross-field support
- **Integrated State Management**: No need for Redux, MobX, or other external libraries
- **Event-Driven Architecture**: Robust event system for component communication
- **Production Ready**: Battle-tested in enterprise applications

2. Problem Statement

2.1 The Challenge

In enterprise environments, UI framework decisions change frequently due to corporate standardization initiatives, vendor relationships, technology evolution, and business unit requirements. Our organization has experienced multiple mandatory framework transitions:

Framework	Reason for Change
Tuxedo React	Corporate standard at the time
Salt Design System	New enterprise design standard
LOB-Specific Frameworks	Business units requiring customization

Each transition required **6-12 months** of development effort, involving:

- **Complete rewrite** of all UI components
- **Re-implementation** of business logic embedded in components
- **Extensive regression testing** across hundreds of forms
- **Team retraining** on new component APIs and patterns
- **Risk of introducing bugs** during migration

2.2 The Root Cause

The core problem is **tight coupling**. In traditional React development, three concerns are mixed together in every component:

1. **Business Logic** - Validation rules, calculations, workflows
2. **State Management** - Form values, errors, loading states
3. **UI Markup** - Framework-specific components and styling

```
// Tuxedo React version
function EmployeeForm() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [errors, setErrors] = useState({});

  // Business logic embedded in component
  const validate = () => {
    const newErrors = {};
    if (!name || name.length < 2) newErrors.name = "Name required (min 2 chars)";
    if (!email || !email.includes("@")) newErrors.email = "Valid email required";
    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };

  return (
    <TuxedoForm>
      <TuxedoTextField        // ← Tuxedo-specific
        label="Name"
        value={name}
        onChange={(e) => setName(e.target.value)}
        error={errors.name}
      />
      <TuxedoTextField        // ← Tuxedo-specific
        label="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        error={errors.email}
      />
      <TuxedoButton onClick={validate}>Submit</TuxedoButton>
    </TuxedoForm>
  );
}
```

When migrating to Salt Design System, the **same business logic** must be completely rewritten:

```
// Salt version - identical logic, complete rewrite
function EmployeeForm() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [errors, setErrors] = useState({});

  // Same validation logic - duplicated
  const validate = () => {
    const newErrors = {};
    if (!name || name.length < 2) newErrors.name = "Name required (min 2 chars)";
    if (!email || !email.includes("@")) newErrors.email = "Valid email required";
    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };

  return (
    <FormField validationStatus={errors.name ? "error" : undefined}>
      <FormLabel>Name</FormLabel>
      <Input                         // ← Salt-specific (different API)
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      {errors.name && <FormHelperText>{errors.name}</FormHelperText>}
    </FormField>
    // ... repeat for every field
  );
}
```

The cost multiplies across the organization:

- 500+ forms × complete rewrite = massive project
- Business logic scattered across hundreds of files
- Validation rules duplicated and often inconsistent
- No guarantee of feature parity after migration
- Developer time spent on rewrites instead of new features

2.3 The Cost

Impact Area	Consequence
Development Time	6-12 months per migration
Team Resources	Multiple developers dedicated to migration
Business Logic	Scattered, duplicated, inconsistent across forms
Quality Risk	Regression bugs introduced in every migration
Team Productivity	Months spent on rewrites instead of delivering features
Technical Debt	Each migration adds complexity and inconsistency
Onboarding	New developers must learn framework-specific patterns

3. The Lumino Solution

3.1 Our Approach

Lumino solves this problem by introducing an **abstraction layer** that separates **what your application does** from **how it looks**:

```
┌──────────────────────────────────────┐
│     Business Logic (NEVER changes)   │
│  Form definitions, validation rules  │
│  Workflows, calculations, API calls  │
├──────────────────────────────────────┤
│          Lumino Framework            │
│    Adapter Pattern / Abstraction     │
├──────────────────────────────────────┤
│       UI Library (Swappable)         │
│   Tuxedo → Salt → Future Framework   │
└──────────────────────────────────────┘
```

Key Insight: Your business logic—validation rules, form structure, field dependencies, calculations—should never need to change when the UI framework changes.

With Lumino, you define your form **once** using abstract, UI-independent components:

```
class EmployeeForm extends Form<Employee> {
  configure() {
    this.addSection("Employee Information")
      .addRow()
        .addField("name")
          .component(LuminoTextInput)  // Abstract - not tied to any UI library
          .label("Name")
          .placeholder("Enter employee name")
          .rules(
            Validators.required({ message: "Name is required" }),
            Validators.minLength(2, { message: "Minimum 2 characters" })
          )
        .endField()
        .addField("email")
          .component(LuminoTextInput)
          .label("Email")
          .rules(
            Validators.required(),
            Validators.email({ message: "Valid email required" })
          )
        .endField()
      .endRow()
    .endSection();
  }
}
```

This form definition:

- Contains all business logic (validation rules)
- Defines the form structure (sections, rows, fields)
- Uses abstract component references (LuminoTextInput)
- **Never changes** regardless of UI framework

3.2 The Result

Switching from Tuxedo to Salt? Change one line in your application:

```
// Before - using Tuxedo
<LuminoProvider adapter={TuxedoAdapter}>
  <App />
</LuminoProvider>

// After - using Salt
<LuminoProvider adapter={SaltAdapter}>
  <App />
</LuminoProvider>
```

All 500+ forms instantly use the new design system. Zero rewrites. Zero regression risk.

3.3 Comparison

Metric	Traditional Approach	With Lumino
Framework migration	6-12 months	1-2 days (adapter swap)
Code to rewrite	100% of components	0% (business logic unchanged)
Business logic changes	Required for every migration	None
Regression risk	High - bugs in every migration	Minimal - logic unchanged
Developer training	Full framework retraining	Adapter interface only
Future migrations	Same pain every time	Same simple adapter swap

3.4 Additional Benefits

- **Consistency**: All forms follow the same patterns and validation rules
- **Testability**: Business logic can be tested independently of UI
- **Maintainability**: Changes to validation or structure happen in one place
- **Reusability**: Form components can be shared across applications
- **Type Safety**: Full TypeScript support with compile-time validation

---

4. Architecture Overview

Lumino employs a layered architecture separating concerns across four primary layers:

```
┌─────────────────────────────────────────────────────────┐
│                    Application Layer                     │
│  (App, Pages, Forms, Business Logic)                    │
├─────────────────────────────────────────────────────────┤
│                     Lumino Core                          │
│  ┌─────────┬─────────┬─────────┬─────────┬─────────┐    │
│  │  Form   │  Page   │ Routing │   API   │  State  │    │
│  │ Builder │ Builder │         │ Layer   │ Manager │    │
│  └─────────┴─────────┴─────────┴─────────┴─────────┘    │
│  ┌─────────┬─────────┬─────────┬─────────────────────┐  │
│  │Validation│ Events  │ Context │     Containers     │  │
│  │ Runner  │ Emitter │ System  │ (Tabs/Dialog/Table) │  │
│  └─────────┴─────────┴─────────┴─────────────────────┘  │
├─────────────────────────────────────────────────────────┤
│                    Adapter Layer                         │
│  ┌─────────────┬─────────────┬─────────────────────┐    │
│  │  Salt DS    │ Material UI │   Custom Adapters   │    │
│  │  Adapter    │   Adapter   │                     │    │
│  └─────────────┴─────────────┴─────────────────────┘    │
├─────────────────────────────────────────────────────────┤
│                      React DOM                           │
└─────────────────────────────────────────────────────────┘
```

4.1 Design Principles

Separation of Concerns: Definitions describe structure and behavior; adapters handle rendering.

Composition Over Inheritance: Applications are composed of reusable components.

Convention Over Configuration: Sensible defaults reduce boilerplate while allowing customization.

Progressive Enhancement: Start simple, add complexity only when needed.

Type Safety First: Full TypeScript integration with compile-time checks.

5. Core Concepts

5.1 The Lumino Namespace

All Lumino functionality is accessed through a unified namespace:

```
import { Lumino } from "lumino/core";

// Configure the framework
Lumino.configure({
  adapter: SaltAdapter,
  defaultLocale: "en-US",
  validation: {
    validateOnBlur: true,
    validateOnChange: false,
  }
});

// Access framework services
Lumino.api.register(employeeApi);
Lumino.state.create("employees", initialState);
Lumino.events.on("form:submit", handleSubmit);
```

5.2 LuminoProvider

The provider wraps your application and provides context:

```
import { LuminoProvider } from "lumino/react";
import { SaltAdapter } from "lumino/adapters/salt";

function App() {
  return (
    <LuminoProvider adapter={SaltAdapter}>
      <YourApplication />
    </LuminoProvider>
  );
}
```

5.3 Abstract Components

Lumino provides abstract component references that resolve to UI implementations at runtime:

```
// These are UI-independent references
import {
  LuminoTextInput,
  LuminoSelect,
  LuminoCheckbox,
  LuminoButton,
  LuminoCard,
  LuminoTable,
} from "lumino/react";

// Used in form definitions - resolved by adapter
.addField("email")
  .component(LuminoTextInput)
  .label("Email")
  .props({ type: "email" })
.endField()
```

---

6. The Adapter Pattern

6.1 How Adapters Work

Adapters translate abstract Lumino components to concrete UI implementations:

```
Abstract Component      Adapter Resolution         Concrete UI
─────────────────      ──────────────────        ────────────
LuminoTextInput    ──→  Salt Adapter       ──→   <Input />
                   ──→  Material Adapter   ──→   <TextField />
                   ──→  Custom Adapter     ──→   <CustomInput />
```

6.2 Adapter Interface

Each adapter implements a comprehensive interface:

```
interface UIAdapter {
  // Field Components
  TextInput: React.ComponentType<TextInputProps>;
  NumberInput: React.ComponentType<NumberInputProps>;
  TextArea: React.ComponentType<TextAreaProps>;
  Select: React.ComponentType<SelectProps>;
  MultiSelect: React.ComponentType<MultiSelectProps>;
  Autocomplete: React.ComponentType<AutocompleteProps>;
  Checkbox: React.ComponentType<CheckboxProps>;
  CheckboxGroup: React.ComponentType<CheckboxGroupProps>;
  Switch: React.ComponentType<SwitchProps>;
  RadioGroup: React.ComponentType<RadioGroupProps>;
  DatePicker: React.ComponentType<DatePickerProps>;
  TimePicker: React.ComponentType<TimePickerProps>;

  // Container Components
  Card: React.ComponentType<CardProps>;
  Table: React.ComponentType<TableProps>;
  Tabs: React.ComponentType<TabsProps>;
  Dialog: React.ComponentType<DialogProps>;
  Accordion: React.ComponentType<AccordionProps>;

  // Action Components
  Button: React.ComponentType<ButtonProps>;
  IconButton: React.ComponentType<IconButtonProps>;
  Link: React.ComponentType<LinkProps>;

  // Layout Components
  FormLayout: React.ComponentType<FormLayoutProps>;
  Section: React.ComponentType<SectionProps>;
  Row: React.ComponentType<RowProps>;
  Stack: React.ComponentType<StackProps>;
  Flex: React.ComponentType<FlexProps>;

  // Feedback Components
  Toast: React.ComponentType<ToastProps>;
  Progress: React.ComponentType<ProgressProps>;
  Spinner: React.ComponentType<SpinnerProps>;
  Badge: React.ComponentType<BadgeProps>;
  Skeleton: React.ComponentType<SkeletonProps>;

  // Navigation Components
  Navbar: React.ComponentType<NavbarProps>;
  Sidebar: React.ComponentType<SidebarProps>;
  Menu: React.ComponentType<MenuProps>;
  Breadcrumbs: React.ComponentType<BreadcrumbsProps>;
  Pagination: React.ComponentType<PaginationProps>;
  Stepper: React.ComponentType<StepperProps>;

  // Typography Components
  Heading: React.ComponentType<HeadingProps>;
  Text: React.ComponentType<TextProps>;
  Icon: React.ComponentType<IconProps>;
  Avatar: React.ComponentType<AvatarProps>;

  // Error Components
  ErrorBoundary: React.ComponentType<ErrorBoundaryProps>;
  Alert: React.ComponentType<AlertProps>;
  ErrorPage: React.ComponentType<ErrorPageProps>;
}
```

6.3 Creating Custom Adapters

Organizations can create adapters for their design systems:

```
const CustomAdapter: UIAdapter = {
  TextInput: ({ value, onChange, error, label, placeholder }) => (
    <div className="custom-field">
      <label className="custom-label">{label}</label>
      <input
        className={`custom-input ${error ? "error" : ""}`}
        value={value}
        onChange={onChange}
        placeholder={placeholder}
      />
      {error && <span className="custom-error">{error}</span>}
    </div>
  ),
  // ... implement all required components
};
```

6.4 Benefits

1. **Design System Flexibility**: Switch UI libraries without changing application code
2. **Gradual Migration**: Adopt new design systems incrementally
3. **Consistent Theming**: Apply styling across all components
4. **Simplified Testing**: Test with lightweight mock adapters
5. **Future-Proofing**: Adapt to new UI trends without rewrites

---

7. Lumino Components

7.1 Field Components

Lumino provides abstract field components for all common input types:

Component	Description	Features
`LuminoTextInput`	Single-line text input	Type variants, masks, icons
`LuminoNumberInput`	Numeric input	Min/max, step, formatting
`LuminoTextArea`	Multi-line text	Auto-resize, character count
`LuminoSelect`	Dropdown selection	Search, groups, async loading
`LuminoMultiSelect`	Multiple selection	Tags, search, limits
`LuminoAutocomplete`	Type-ahead search	Async data, custom rendering
`LuminoCheckbox`	Boolean toggle	Indeterminate state
`LuminoCheckboxGroup`	Multiple checkboxes	Group validation
`LuminoSwitch`	Toggle switch	Labels, sizes
`LuminoRadioGroup`	Single selection	Horizontal/vertical
`LuminoDatePicker`	Date selection	Range, disabled dates
`LuminoTimePicker`	Time selection	12/24 hour, intervals

7.2 Container Components

Component	Description	Features
`LuminoCard`	Content container	Header, body, footer, actions
`LuminoTable`	Data table	Sorting, filtering, pagination
`LuminoTabs`	Tabbed interface	Icons, badges, closable
`LuminoDialog`	Modal dialog	Sizes, forms, confirmations
`LuminoAccordion`	Collapsible sections	Multiple expand, icons

7.3 Action Components

Component	Description	Features
`LuminoButton`	Action button	Variants, loading, icons
`LuminoIconButton`	Icon-only button	Tooltips, sizes
`LuminoLink`	Navigation link	Internal/external, styling

7.4 Feedback Components

Component	Description	Features
`LuminoToast`	Notification toast	Types, auto-dismiss, actions
`LuminoProgress`	Progress indicator	Linear, circular, indeterminate
`LuminoSpinner`	Loading spinner	Sizes, overlay
`LuminoBadge`	Status badge	Colors, counts, dots
`LuminoSkeleton`	Loading placeholder	Shapes, animation

7.5 Navigation Components

Component	Description	Features
`LuminoNavbar`	Top navigation	Responsive, dropdowns
`LuminoSidebar`	Side navigation	Collapsible, nested items
`LuminoMenu`	Context menu	Nested, icons, shortcuts
`LuminoBreadcrumbs`	Path navigation	Truncation, custom separators
`LuminoPagination`	Page navigation	Sizes, boundaries
`LuminoStepper`	Step indicator	Vertical/horizontal, clickable

7.6 Typography Components

Component	Description	Features
`LuminoH1-H6`	Headings	Responsive sizes
`LuminoText`	Body text	Variants, truncation
`LuminoIcon`	Icons	Library integration, sizes
`LuminoAvatar`	User avatar	Images, initials, groups

8. Layout System

8.1 Layout Components

Lumino provides flexible layout primitives:

```
import {
  LuminoStackLayout,
  LuminoFlowLayout,
  LuminoFlexLayout,
  LuminoGridLayout,
  LuminoContainer,
} from "lumino/react";
```

8.2 Stack Layout

Vertical arrangement with consistent spacing:

```
<LuminoStackLayout gap={2}>
  <LuminoText>First item</LuminoText>
  <LuminoText>Second item</LuminoText>
  <LuminoText>Third item</LuminoText>
</LuminoStackLayout>
```

8.3 Flex Layout

Flexible box layout with full control:

```
<LuminoFlexLayout
  direction="row"
  justify="space-between"
  align="center"
  wrap="wrap"
  gap={2}
>
  <LuminoButton>Left</LuminoButton>
  <LuminoButton>Right</LuminoButton>
</LuminoFlexLayout>
```

8.4 Grid Layout

CSS Grid-based layout:

```
<LuminoGridLayout
  columns={3}
  gap={4}
  responsive={{
    sm: { columns: 1 },
    md: { columns: 2 },
    lg: { columns: 3 },
  }}
>
  <LuminoCard>Card 1</LuminoCard>
  <LuminoCard>Card 2</LuminoCard>
  <LuminoCard>Card 3</LuminoCard>
</LuminoGridLayout>
```

9. App Builder

9.1 App Class

The App class defines application-level configuration:

```
import { App } from "lumino/core";

class MyApp extends App {
  configure() {
    // Set application metadata
    this.name("My Enterprise App");
    this.version("1.0.0");

    // Configure authentication
    this.auth({
      provider: "oauth2",
      loginPath: "/login",
      logoutPath: "/logout",
    });

    // Register global APIs
    this.api(EmployeeApi);
    this.api(DepartmentApi);

    // Configure global state
    this.state("user", { role: "guest" });

    // Set default layout
    this.layout(MainLayout);

    // Register pages
    this.page(DashboardPage);
    this.page(EmployeesPage);
    this.page(SettingsPage);
  }
}
```

9.2 LuminoApp

Render the application:

```
import { LuminoApp } from "lumino/react";

const app = new MyApp();

function Root() {
  return <LuminoApp app={app} />;
}
```

9.3 App Configuration Options

Method	Description
`.name()`	Application name
`.version()`	Version string
`.auth()`	Authentication config
`.api()`	Register API class
`.state()`	Initialize global state
`.layout()`	Default layout component
`.page()`	Register page
`.errorBoundary()`	Custom error handling
`.loading()`	Global loading config

10. Page Builder

9.1 Page Class

Pages define routes and content:

```
import { Page } from "lumino/core";

class EmployeesPage extends Page {
  configure() {
    // Route configuration
    this.route("/employees");
    this.route("/employees/:id", "detail");

    // Page metadata
    this.title("Employees");
    this.icon("users");

    // Access control
    this.guard((ctx) => ctx.user.hasPermission("employees:read"));

    // Page content
    this.section("Employee List")
      .component(EmployeeTable)
    .endSection();

    // Page actions
    this.action("add")
      .label("Add Employee")
      .icon("plus")
      .onClick((ctx) => ctx.navigate("/employees/new"))
    .end();
  }
}
```

9.2 Page with Tabs

```
class SettingsPage extends Page {
  configure() {
    this.route("/settings");
    this.title("Settings");

    // Add tabs to a page
    this.addTabs(new SettingsTabs("settings-tabs"));
  }
}

class SettingsTabs extends Tabs {
  configure() {
    this.tab("profile")
      .label("Profile")
      .form(ProfileForm)
    .end();

    this.tab("preferences")
      .label("Preferences")
      .form(PreferencesForm)
    .end();

    this.tab("security")
      .label("Security")
      .form(SecurityForm)
    .end();
  }
}
```

9.3 PageRenderer

Render pages with the PageRenderer:

```
import { PageRenderer } from "lumino/react";

const employeesPage = new EmployeesPage();

function EmployeesRoute() {
  return <PageRenderer page={employeesPage} />;
}
```

11. Form Builder

10.1 Form Class

The Form class is the central abstraction for form handling:

```typescript
import { Form, Validators } from "lumino/core";
import { LuminoTextInput, LuminoSelect } from "lumino/react";

interface Employee {
  firstName: string;
  lastName: string;
  email: string;
  department: string;
  startDate: Date;
}

class EmployeeForm extends Form<Employee> {
  configure() {
    this.addSection("Personal Information")
      .addRow()
        .addField("firstName")
          .component(LuminoTextInput)
          .label("First Name")
          .placeholder("Enter first name")
          .rules(Validators.required({ message: "First name is required" }))
        .endField()
        .addField("lastName")
          .component(LuminoTextInput)
          .label("Last Name")
          .rules(Validators.required())
        .endField()
      .endRow()
      .addRow()
        .addField("email")
          .component(LuminoTextInput)
          .label("Email")
          .props({ type: "email" })
          .rules(
            Validators.required(),
            Validators.email({ message: "Invalid email format" })
          )
        .endField()
      .endRow()
    .endSection();

    this.addSection("Employment Details")
      .addRow()
        .addField("department")
          .component(LuminoSelect)
          .label("Department")
          .props({
            options: [
              { value: "engineering", label: "Engineering" },
              { value: "sales", label: "Sales" },
              { value: "marketing", label: "Marketing" },
            ]
          })
          .rules(Validators.required())
        .endField()
        .addField("startDate")
          .component(LuminoDatePicker)
          .label("Start Date")
        .endField()
      .endRow()
    .endSection();

    // Form actions
    this.addAction("save")
      .label("Save Employee")
      .variant("primary")
      .onClick(async (ctx) => {
        const isValid = await ctx.validate();
        if (isValid) {
          await this.onSubmit(ctx.getFormData());
        }
      })
    .end();
  }
}
```

10.2 Field Configuration Options

Method	Description
`.component()`	UI component to render
`.label()`	Field label
`.placeholder()`	Placeholder text
`.defaultValue()`	Initial value
`.rules()`	Validation rules
`.props()`	Component-specific props
`.visibleByCondition()`	Conditional visibility
`.disabledByCondition()`	Conditional disabled state
`.computedValue()`	Derived value calculation
`.help()`	Help text
`.tooltip()`	Tooltip content

10.3 Conditional Visibility

Show/hide fields based on other values:

```
.addField("billingAddress")
  .component(LuminoTextInput)
  .label("Billing Address")
  .visibleByCondition((ctx) => !ctx.getValue("sameAsShipping"))
.endField()
```

10.4 Computed Values

Fields that derive values from other fields:

```
.addField("total")
  .component(LuminoTextInput)
  .label("Total")
  .readonly()
  .computedValue((ctx) => {
    const price = ctx.getValue("price") || 0;
    const quantity = ctx.getValue("quantity") || 0;
    const tax = ctx.getValue("taxRate") || 0;
    return (price * quantity * (1 + tax)).toFixed(2);
  })
.endField()
```

10.5 Nested Objects

Handle complex nested data structures:

```
interface Employee {
  name: string;
  address: {
    street: string;
    city: string;
    state: string;
    zip: string;
  };
}

class EmployeeForm extends Form<Employee> {
  configure() {
    this.addSection("Basic Info")
      .addRow()
        .addField("name").component(LuminoTextInput).label("Name").endField()
      .endRow()
    .endSection();

    // Nested object section
    this.addSection("Address")
      .addRow()
        .addField("address.street").component(LuminoTextInput).label("Street").en
      .endRow()
      .addRow()
        .addField("address.city").component(LuminoTextInput).label("City").endFie
        .addField("address.state").component(LuminoTextInput).label("State").endF
        .addField("address.zip").component(LuminoTextInput).label("ZIP").endField
      .endRow()
    .endSection();
  }
}
```

10.6 Array Fields (Lists)

Handle arrays of items:

```
interface Employee {
  name: string;
  addresses: Address[];
}

class EmployeeForm extends Form<Employee> {
  configure() {
    this.addSection("Name")
      .addRow()
        .addField("name").component(LuminoTextInput).label("Name").endField()
      .endRow()
    .endSection();

    // Array field - renders as list with add/remove
    this.addList<Address>("addresses")
      .label("Addresses")
      .include(AddressFields)  // Reusable component
      .addable()               // Allow adding items
      .removable()             // Allow removing items
      .minItems(1)             // Minimum 1 address
      .maxItems(5)             // Maximum 5 addresses
    .end();
  }
}
```

10.7 Tabs Inside Forms

Organize complex forms with tabs:

```
class EmployeeForm extends Form<Employee> {
  configure() {
    this.addSection("Header")
      .addRow()
        .addField("name").component(LuminoTextInput).label("Name").endField()
      .endRow()
    .endSection();

    // Tabs within the form
    this.addTabs("details")
      .tab("personal")
        .label("Personal")
        .form(PersonalInfoForm)
      .endTab()
      .tab("employment")
        .label("Employment")
        .form(EmploymentForm)
      .endTab()
      .tab("benefits")
        .label("Benefits")
        .form(BenefitsForm)
      .endTab()
    .endTabs();
  }
}
```

10.8 Reusable Form Components

Create reusable form sections:

```
// Reusable address fields component
class AddressFields extends Component<Address> {
  configure() {
    this.addRow()
      .addField("street").component(LuminoTextInput).label("Street").endField()
    .endRow()
    .addRow()
      .addField("city").component(LuminoTextInput).label("City").endField()
      .addField("state").component(LuminoTextInput).label("State").endField()
      .addField("zip").component(LuminoTextInput).label("ZIP").endField()
    .endRow();
  }
}

// Use in multiple forms
class ShippingForm extends Form<Order> {
  configure() {
    this.addSection("Shipping Address")
      .include(AddressFields, "shippingAddress")
    .endSection();
  }
}

class BillingForm extends Form<Order> {
  configure() {
    this.addSection("Billing Address")
      .include(AddressFields, "billingAddress")
    .endSection();
  }
}
```

10.9 FormRenderer

Render forms with the FormRenderer:

```
import { FormRenderer } from "lumino/react";

const employeeForm = new EmployeeForm();

function EmployeeFormPage() {
  return (
    <FormRenderer
      form={employeeForm}
      initialValues={{
        firstName: "",
        lastName: "",
        email: "",
      }}
      onSubmit={async (values) => {
        await saveEmployee(values);
      }}
      validateOnBlur={true}
      validateOnChange={false}
    />
  );
}
```

10.10 FormContext API

Access form state and operations within form definitions:

```typescript
interface FormContext<T> {
  // Values
  getValue<K extends keyof T>(field: K): T[K];
  setValue<K extends keyof T>(field: K, value: T[K]): void;
  getFormData(): T;
  setValues(values: Partial<T>): void;

  // Validation
  validate(): Promise<boolean>;
  validateField(field: string): Promise<boolean>;
  getErrors(): Record<string, string[]>;
  getFieldError(field: string): string | null;
  clearErrors(): void;

  // State
  isDirty(): boolean;
  isTouched(field: string): boolean;
  isValid(): boolean;
  isSubmitting(): boolean;

  // Actions
  submit(): Promise<void>;
  reset(): void;

  // Notifications
  notify(message: string, type: "success" | "error" | "warning" | "info"): void;

  // Navigation
  navigate(path: string): void;

  // Dialogs
  openDialog(dialog: Dialog, options?: DialogOptions): void;
}
```

12. Validation System

11.1 Built-in Validators

Lumino provides comprehensive validators:

```
import { Validators } from "lumino/core";

// Basic validators
Validators.required({ message: "Field is required" })
Validators.email({ message: "Invalid email format" })
Validators.minLength(5, { message: "Minimum 5 characters" })
Validators.maxLength(100, { message: "Maximum 100 characters" })
Validators.min(0, { message: "Must be positive" })
Validators.max(1000, { message: "Maximum is 1000" })
Validators.pattern(/^[A-Z]{2}[0-9]{6}$/, { message: "Invalid format" })

// Type validators
Validators.number({ message: "Must be a number" })
Validators.integer({ message: "Must be a whole number" })
Validators.url({ message: "Invalid URL" })
Validators.phone({ message: "Invalid phone number" })
```

11.2 Custom Validators

Create custom validation logic:

```
Validators.custom({
  validate: (value, ctx) => {
    // Custom logic
    return value.startsWith("ABC");
  },
  message: "Must start with ABC"
})
```

11.3 Cross-Field Validation

Validate based on other field values:

```
// Password confirmation
.addField("confirmPassword")
  .component(LuminoTextInput)
  .label("Confirm Password")
  .rules(
    Validators.custom({
      validate: (value, ctx) => value === ctx.getValue("password"),
      message: "Passwords do not match"
    })
  )
.endField()

// Date range validation
.addField("endDate")
  .component(LuminoDatePicker)
  .label("End Date")
  .rules(
    Validators.custom({
      validate: (value, ctx) => {
        const startDate = ctx.getValue("startDate");
        if (!startDate || !value) return true;
        return new Date(value) > new Date(startDate);
      },
      message: "End date must be after start date"
    })
  )
.endField()
```

11.4 Conditional Validation

Validate only when conditions are met:

```
.addField("companyName")
  .component(LuminoTextInput)
  .label("Company Name")
  .visibleByCondition((ctx) => ctx.getValue("employmentType") === "employed")
  .rules(
    Validators.custom({
      validate: (value, ctx) => {
        if (ctx.getValue("employmentType") !== "employed") return true;
        return !!value && value.length > 0;
      },
      message: "Company name is required for employed"
    })
  )
.endField()
```

11.5 Async Validation

Server-side validation with automatic debouncing:

```
.addField("username")
  .component(LuminoTextInput)
  .label("Username")
  .rules(
    Validators.required(),
    Validators.custom({
      validate: async (value) => {
        const response = await fetch(`/api/check-username?name=${value}`);
        const { available } = await response.json();
        return available;
      },
      message: "Username already taken"
    })
  )
.endField()
```

11.6 Validation Modes

Configure when validation runs:

```
<FormRenderer
  form={form}
  validateOnBlur={true}     // Validate when field loses focus (default)
  validateOnChange={false}   // Validate on every keystroke
  validateOnSubmit={true}    // Validate on form submission
/>
```

11.7 Action-Based Validation

Skip or require validation for specific actions:

```
Validators.required({
  message: "Required",
  skipOn: ["draft"],      // Skip when action is "draft"
  validateOn: ["submit"]  // Only validate on "submit"
})
```

13. Routing

12.1 LuminoRouter

Integrated routing solution:

```
import { LuminoRouter, Route } from "lumino/react";

function App() {
  return (
    <LuminoRouter>
      <Route path="/" component={HomePage} />
      <Route path="/employees" component={EmployeesPage} />
      <Route path="/employees/:id" component={EmployeeDetailPage} />
      <Route path="/settings/*" component={SettingsPage} />
      <Route path="*" component={NotFoundPage} />
    </LuminoRouter>
  );
}
```

12.2 Navigation

Programmatic navigation:

```
// In form/page context
ctx.navigate("/employees");
ctx.navigate("/employees/123");
ctx.navigate(-1); // Go back

// Using hook
const { navigate } = useNavigation();
navigate("/employees");
```

12.3 Route Guards

Protect routes with guards:

```
class AdminPage extends Page {
  configure() {
    this.route("/admin");

    // Route guard
    this.guard((ctx) => {
      if (!ctx.user.isAdmin) {
        ctx.navigate("/unauthorized");
        return false;
      }
      return true;
    });
  }
}
```

12.4 Router Hooks

```
import { useRouter, useParams, useSearchParams } from "lumino/react";

function EmployeeDetail() {
  const { id } = useParams();
  const [searchParams, setSearchParams] = useSearchParams();
  const { navigate, location } = useRouter();

  return <div>Employee ID: {id}</div>;
}
```

14. API & Data Layer

13.1 API Builder

Define APIs declaratively:

```
import { Api } from "lumino/core";

class EmployeeApi extends Api {
  configure() {
    this.baseUrl("/api/employees");

    this.endpoint("list")
      .method("GET")
      .path("/")
      .cache({ ttl: 60000 })
    .end();

    this.endpoint("get")
      .method("GET")
      .path("/:id")
    .end();

    this.endpoint("create")
      .method("POST")
      .path("/")
    .end();

    this.endpoint("update")
      .method("PUT")
      .path("/:id")
    .end();

    this.endpoint("delete")
      .method("DELETE")
      .path("/:id")
    .end();
  }
}
```

13.2 CrudApi

Shorthand for CRUD operations:

```
import { CrudApi } from "lumino/core";

class EmployeeApi extends CrudApi<Employee> {
  constructor() {
    super({
      baseUrl: "/api/employees",
      // All CRUD endpoints auto-configured
    });
  }
}

// Usage
const api = new EmployeeApi();
const employees = await api.list();
const employee = await api.get(123);
await api.create(newEmployee);
await api.update(123, updatedEmployee);
await api.delete(123);
```

13.3 LookupApi

For reference data and dropdowns:

```
import { LookupApi } from "lumino/core";

class DepartmentLookup extends LookupApi<Department> {
  constructor() {
    super({
      baseUrl: "/api/departments",
      labelField: "name",
      valueField: "id",
      cache: { ttl: 300000 }, // Cache for 5 minutes
    });
  }
}

// Use in form
.addField("department")
  .component(LuminoSelect)
  .lookup(DepartmentLookup)
  .label("Department")
.endField()
```

13.4 Mapper (DTO/Entity)

Transform between API DTOs and domain entities:

```
import { Mapper } from "lumino/core";

class EmployeeMapper extends Mapper<EmployeeDTO, Employee> {
  toEntity(dto: EmployeeDTO): Employee {
    return {
      id: dto.employee_id,
      fullName: `${dto.first_name} ${dto.last_name}`,
      email: dto.email_address,
      startDate: new Date(dto.start_date),
    };
  }

  toDTO(entity: Employee): EmployeeDTO {
    const [firstName, lastName] = entity.fullName.split(" ");
    return {
      employee_id: entity.id,
      first_name: firstName,
      last_name: lastName,
      email_address: entity.email,
      start_date: entity.startDate.toISOString(),
    };
  }
}
```

13.5 ApiRegistry

Centralized API management:

```
import { Lumino } from "lumino/core";

// Register APIs
Lumino.api.register("employees", EmployeeApi);
Lumino.api.register("departments", DepartmentApi);

// Access APIs
const employeeApi = Lumino.api.get("employees");
```

13.6 CacheManager

Automatic response caching:

```
import { CacheManager } from "lumino/core";

// Configure caching
CacheManager.configure({
  defaultTTL: 60000,      // 1 minute default
  maxSize: 100,           // Max 100 cached responses
  storage: "memory",      // or "localStorage"
});

// Manual cache operations
CacheManager.invalidate("employees");
CacheManager.clear();
```

15. State Management

14.1 StateManager

Integrated state management without external libraries:

```
import { Lumino } from "lumino/core";

// Create state
Lumino.state.create("employees", {
  list: [],
  selected: null,
  loading: false,
});

// Update state
Lumino.state.set("employees", {
  list: employees,
  loading: false,
});

// Get state
const state = Lumino.state.get("employees");
```

14.2 EntityStore

Single entity state management:

```
import { EntityStore } from "lumino/core";

const userStore = new EntityStore<User>("currentUser");

// Set entity
userStore.set(user);

// Get entity
const user = userStore.get();

// Update entity
userStore.update({ name: "New Name" });

// Clear entity
userStore.clear();
```

14.3 CollectionStore

Collection state management:

```
import { CollectionStore } from "lumino/core";

const employeesStore = new CollectionStore<Employee>("employees");

// Set all items
employeesStore.setAll(employees);

// Add item
employeesStore.add(newEmployee);

// Update item
employeesStore.update(123, { name: "Updated" });

// Remove item
employeesStore.remove(123);

// Filter/sort
const filtered = employeesStore.filter((e) => e.active);
const sorted = employeesStore.sort((a, b) => a.name.localeCompare(b.name));
```

14.4 State Hooks

React hooks for state access:

```
import { useEntity, useCollection } from "lumino/react";

function UserProfile() {
  const { data: user, loading, update } = useEntity<User>("currentUser");

  return <div>{user?.name}</div>;
}

function EmployeeList() {
  const { items, loading, add, remove } = useCollection<Employee>("employees");

  return (
    <ul>
      {items.map(emp => <li key={emp.id}>{emp.name}</li>)}
    </ul>
  );
}
```

16. Event System

15.1 EventEmitter

Robust event system for component communication:

```
import { Lumino } from "lumino/core";

// Subscribe to events
Lumino.events.on("employee:created", (employee) => {
  console.log("New employee:", employee);
});

// Emit events
Lumino.events.emit("employee:created", newEmployee);

// One-time subscription
Lumino.events.once("app:ready", () => {
  console.log("App is ready");
});

// Unsubscribe
const unsubscribe = Lumino.events.on("event", handler);
unsubscribe();
```

15.2 Form Events

```
// Available form events
Lumino.events.on("form:init", ({ formId }) => {});
Lumino.events.on("form:change", ({ formId, field, value }) => {});
Lumino.events.on("form:validate", ({ formId, errors }) => {});
Lumino.events.on("form:submit", ({ formId, values }) => {});
Lumino.events.on("form:reset", ({ formId }) => {});
Lumino.events.on("form:error", ({ formId, error }) => {});
```

15.3 Page Events

```
Lumino.events.on("page:enter", ({ pageId, params }) => {});
Lumino.events.on("page:leave", ({ pageId }) => {});
Lumino.events.on("page:load", ({ pageId, data }) => {});
```

15.4 API Events

```
Lumino.events.on("api:request", ({ url, method }) => {});
Lumino.events.on("api:response", ({ url, status, data }) => {});
Lumino.events.on("api:error", ({ url, error }) => {});
```

15.5 App Events

```
Lumino.events.on("app:init", () => {});
Lumino.events.on("app:ready", () => {});
Lumino.events.on("app:error", ({ error }) => {});
Lumino.events.on("auth:login", ({ user }) => {});
Lumino.events.on("auth:logout", () => {});
```

15.6 Custom Events

```
// Define custom events
Lumino.events.on("cart:itemAdded", ({ item, quantity }) => {
  updateCartBadge();
});

Lumino.events.on("notification:received", ({ message }) => {
  showToast(message);
});

// Emit from anywhere
Lumino.events.emit("cart:itemAdded", { item: product, quantity: 1 });
```

15.7 Event Hook

Use events in React components:

```
import { useEvents } from "lumino/react";

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

  useEvents("notification:received", () => {
    setCount(c => c + 1);
  });

  return <Badge count={count}><BellIcon /></Badge>;
}
```

17. React Hooks

16.1 Overview

Lumino provides a comprehensive set of React hooks:

Hook	Description
`useLumino()`	Access Lumino context
`useForm()`	Form instance and state
`useFormData()`	Form data subscription
`usePage()`	Page instance and state
`useApi()`	API calls with loading state
`useMutation()`	Mutation operations
`useDialog()`	Dialog management
`useNavigation()`	Navigation helpers
`useEvents()`	Event subscriptions
`useEntity()`	Entity store access
`useCollection()`	Collection store access

16.2 useLumino

Access the Lumino context:

```
import { useLumino } from "lumino/react";

function MyComponent() {
  const { adapter, state, events, api } = useLumino();

  return <div>Using: {adapter.name}</div>;
}
```

16.3 useForm

Form state and operations:

```
import { useForm } from "lumino/react";

function MyFormComponent() {
  const {
    values,
    errors,
    isValid,
    isDirty,
    isSubmitting,
    getValue,
    setValue,
    validate,
    submit,
    reset,
  } = useForm(myForm);

  return (
    <div>
      <span>Valid: {isValid ? "Yes" : "No"}</span>
      <button onClick={submit} disabled={!isValid || isSubmitting}>
        Submit
      </button>
    </div>
  );
}
```

16.4 useApi

API calls with automatic state management:

```
import { useApi, useMutation } from "lumino/react";

function EmployeeList() {
  // Query
  const { data, loading, error, refetch } = useApi(
    () => employeeApi.list(),
    { autoFetch: true }
  );

  // Mutation
  const { mutate: deleteEmployee, loading: deleting } = useMutation(
    (id) => employeeApi.delete(id),
    { onSuccess: () => refetch() }
  );

  if (loading) return <Spinner />;
  if (error) return <Error message={error} />;

  return (
    <ul>
      {data.map(emp => (
        <li key={emp.id}>
          {emp.name}
          <button onClick={() => deleteEmployee(emp.id)}>Delete</button>
        </li>
      ))}
    </ul>
  );
}
```

16.5 useDialog

Dialog management:

```
import { useDialog } from "lumino/react";

function MyComponent() {
  const { open, close, isOpen } = useDialog();

  const handleClick = () => {
    open(ConfirmDialog, {
      title: "Confirm Delete",
      onConfirm: async () => {
        await deleteItem();
        close();
      }
    });
  };

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

18. Error Handling

17.1 Error Boundary

Catch and handle React errors:

```
import { LuminoErrorBoundary } from "lumino/react";

function App() {
  return (
    <LuminoErrorBoundary
      fallback={<ErrorPage />}
      onError={(error, errorInfo) => {
        logError(error, errorInfo);
      }}
    >
      <MyApplication />
    </LuminoErrorBoundary>
  );
}
```

17.2 Error Pages

Pre-built error pages:

```
import {
  LuminoErrorPage,
  LuminoNotFoundPage,
  LuminoUnauthorizedPage,
  LuminoForbiddenPage,
  LuminoServerErrorPage,
} from "lumino/react";

// 404 Page
<Route path="*" component={LuminoNotFoundPage} />

// Custom error page
<LuminoErrorPage
  code={500}
  title="Something went wrong"
  message="Please try again later"
  action={{ label: "Go Home", path: "/" }}
/>
```

17.3 Form Error Handling

```
class MyForm extends Form<Data> {
  configure() {
    // ... fields

    this.addAction("save")
      .onClick(async (ctx) => {
        try {
          await saveData(ctx.getFormData());
          ctx.notify("Saved successfully", "success");
        } catch (error) {
          ctx.notify("Failed to save", "error");
          // Or set field errors
          ctx.setFieldError("email", "Email already exists");
        }
      })
    .end();
  }
}
```

17.4 API Error Handling

```
// Global API error handler
Lumino.events.on("api:error", ({ error, url }) => {
  if (error.status === 401) {
    Lumino.events.emit("auth:logout");
  } else if (error.status === 403) {
    navigate("/forbidden");
  } else {
    showToast("An error occurred", "error");
  }
});
```

19. Performance Considerations

18.1 Rendering Optimization

Lumino employs several optimization strategies:

1. **Selective Re-rendering**: Only affected fields re-render on value changes
2. **Memoization**: Computed values cached until dependencies change
3. **Lazy Validation**: Async validators automatically debounced
4. **Virtual Rendering**: Support for virtualized lists in large forms

18.2 Bundle Size

Lumino is designed to be lightweight:

Package	Size (gzipped)
lumino/core	~18KB
lumino/react	~10KB
Salt Adapter	~6KB
Total	**~34KB**

18.3 Code Splitting

Lumino supports lazy loading:

```
// Lazy load pages
const EmployeesPage = lazy(() => import("./pages/EmployeesPage"));

// Lazy load forms
const ComplexForm = lazy(() => import("./forms/ComplexForm"));
```

18.4 Memory Management

- Forms automatically clean up subscriptions when unmounted

- Event listeners properly disposed
- State cleared when components unmount
- No memory leaks in long-running applications

---

20. Migration Strategy

19.1 Incremental Adoption

Lumino can be adopted incrementally:

Phase 1: Install and configure

```
npm install lumino
```

```
<LuminoProvider adapter={SaltAdapter}>
  <ExistingApp />
</LuminoProvider>
```

Phase 2: Convert new features to Lumino

Phase 3: Gradually migrate existing forms/pages

Phase 4: Extract reusable components

19.2 Coexistence

Lumino works alongside existing code:

```
function App() {
  return (
    <LuminoProvider adapter={SaltAdapter}>
      {/* New Lumino pages */}
      <Route path="/new/*" component={LuminoPages} />

      {/* Existing pages */}
      <Route path="/legacy/*" component={LegacyPages} />
    </LuminoProvider>
  );
}
```

19.3 Adapter Compatibility

Create adapters that wrap existing components:

```
// Wrap existing design system
const LegacyAdapter: UIAdapter = {
  TextInput: (props) => <LegacyInput {...mapProps(props)} />,
  // ... wrap all components
};
```

---

21. Conclusion

Lumino represents a comprehensive solution for React application development. By providing:

1. **UI Independence**: Write once, render with any design system
2. **Unified Architecture**: Single framework for all application concerns
3. **Type Safety**: Full TypeScript integration
4. **Comprehensive Features**: Forms, pages, routing, API, state, events
5. **Extensibility**: Adapter system for any UI library
6. **Developer Experience**: Declarative, readable code

Organizations can:

- **Reduce Technical Debt**: Applications remain maintainable as requirements evolve
- **Improve Consistency**: Centralized patterns across the codebase
- **Accelerate Development**: Reusable components and patterns
- **Enable Flexibility**: Switch UI libraries without rewriting logic
- **Enhance Testing**: Test logic independently of UI

Getting Started

```
npm install lumino
```

```tsx
import { Form, Validators } from "lumino/core";
import { LuminoProvider, FormRenderer, LuminoTextInput } from "lumino/react";
import { SaltAdapter } from "lumino/adapters/salt";

class MyForm extends Form<{ name: string }> {
  configure() {
    this.addSection("Info")
      .addRow()
        .addField("name")
          .component(LuminoTextInput)
          .label("Name")
          .rules(Validators.required())
        .endField()
      .endRow()
    .endSection();
  }
}

function App() {
  return (
    <LuminoProvider adapter={SaltAdapter}>
      <FormRenderer form={new MyForm()} />
    </LuminoProvider>
  );
}
```

Appendix A: Complete API Reference

See the Lumino documentation site for complete API reference including:

- Lumino namespace API
- App class methods
- Page class methods
- Form class methods
- All validator signatures
- All hook APIs
- Event type definitions
- Context interfaces

Appendix B: Comparison with Alternatives

Feature	Lumino	React Hook Form	Formik	Final Form
UI Independent	**Yes**	No	No	No
Declarative Forms	**Yes**	Partial	Partial	Partial
Page Builder	**Yes**	No	No	No
App Builder	**Yes**	No	No	No
Integrated Routing	**Yes**	No	No	No
State Management	**Yes**	No	No	No
Event System	**Yes**	No	No	No
API Layer	**Yes**	No	No	No
Built-in Adapters	**Yes**	No	No	No
TypeScript First	**Yes**	Yes	Partial	Partial
Cross-field Validation	**Yes**	Yes	Yes	Yes
Async Validation	**Yes**	Yes	Yes	Yes
Computed Fields	**Yes**	No	No	Yes
Nested Objects	**Yes**	Yes	Yes	Yes
Array Fields	**Yes**	Yes	Yes	Yes
Form Composition	**Yes**	Partial	Partial	Partial

Appendix C: Glossary

- **Adapter**: Module mapping Lumino abstract components to concrete UI implementations
- **App**: Top-level application class defining configuration and structure
- **Component**: Reusable form section or field group
- **CrudApi**: API class with standard CRUD operations
- **EntityStore**: State store for single entities
- **CollectionStore**: State store for collections
- **EventEmitter**: Pub/sub system for component communication
- **Form Class**: TypeScript class extending Form defining form structure
- **FormContext**: Runtime interface for form state and operations
- **LookupApi**: API for reference data and dropdowns
- **Mapper**: Transforms between DTOs and domain entities
- **Page**: Route and content definition class
- **Section**: Logical grouping of form fields
- **Tabs**: Tabbed interface container
- **Validator**: Function validating field values

Document Version: 1.0 **Last Updated**: December 2024 **Authors**: Lumino Development Team

---

Lumino - A UI-Independent React Application Framework

"Build enterprise React applications with any design system"