Formative React:
Optimizing Web App Development Through Programmatic Division of Labor

Jacob Polatty
Brown University

January 6, 2020

Abstract

The **Formative React** framework provides an interface that automatically generates an HTML form with controlled input components and built-in API loading and saving functionality using two simple JavaScript objects: a data schema complying with the JSON Schema draft-07 standards and a UI schema object that provides powerful display options including the ability to swap in an entirely different input component with a single line of code. The construction of the form also endows each input, label, button, and group of inputs on the page with a unique CSS id property, allowing for total visual customization of the layout and appearance of the form.

This dynamic method of form generation not only removes a significant complexity from the front-end React developer's workflow, but also facilitates a much more compartmentalized and balanced distribution of labor to the appropriate members of a development team. The central principle of the library is to separate the three central concerns of full stack web development: *back-end data management, the user experience, and the appearance of the user interface*; such that each can be handled independently and efficiently in contrast to the convoluted development process of dealing directly with HTML inputs in React. The core form generation engine is also designed to be completely modular to allow for new input components to be contributed to the repository and grow the library of available inputs over time, giving developers the opportunity to integrate their existing specialized form inputs into the framework and enabling them to immediately begin using these components within their forms.

Contents

1 Background 3
1.1 Problem Statement 3
1.2 Existing React Form Implementations 3
1.2.1 React Bootstrap 3
1.2.2 Formik 5
1.2.3 react-jsonschema-form 6

2 Formative Philosophy 9
2.1 Schema-Based Design 9
2.1.1 Data and UI Separation 9
2.1.2 Inclusion List 9
2.2 Input Component Extensibility 10
2.2.1 Open Source Development 10
2.2.2 Locally Defined Inputs 10

3 Formative React Design 11
3.1 Schema Parsing Overview 11
3.2 Form Generation Overview 12
3.3 State Management Overview 13
3.3.1 Update State Tracking 13
3.4 API Interface Overview 14
3.5 Input Design Overview 14
3.5.1 Testing and Static Type Checking 14
3.5.2 Autocomplete Input Example 15

4 Programmatic Work Division 15
4.1 Interface Structure Defined by Back-end 15
4.2 Extensibility Driven by Front-end 16
4.3 Full Visual Customizability for Designer 16

1 Background

1.1 Problem Statement

One of the most common challenges in React web app development is the management of HTML forms and form data, with libraries such as React Bootstrap only providing a simple wrapper on HTML input fields and the developer still forced to lay out these components individually and handle data population and server requests on submit [1]. As a result, React forms are frequently hard-coded in place in order to comply directly with the back-end requirements, resulting in an inflexible tight coupling between the server and client application that can limit the potential for updates. Additionally, this coding pattern often demands a crossover in responsibilities on the development team, with a back-end developer having to dictate the presentation of front-end content and the React developer forced to either design the UI of the form through component props or to construct their own CSS id format in order to facilitate a designers work on the project.

1.2 Existing React Form Implementations

While efforts have been made to streamline the generation of forms in React through open source libraries such as `Formik` and Mozilla Services' `react-jsonschema-form`, these existing implementations have fundamental constraints that can still demand extensive setup work to use with simple forms [2,3]. Evaluations of the limitations of these libraries and the React Bootstrap `Form` component are presented in the following pages:

1.2.1 React Bootstrap

The React Bootstrap framework implements all of the features of the Bootstrap web design framework as standalone React components, including providing React wrappers for HTML form inputs with Bootstrap styling. Under React Bootstrap v1.0, all form inputs and fields are defined within the `Form` class, with the `Form.Control` and `Form.Check` components taking in user inputs and the `Form.Label`, `Form.Text`, and `Form.Group` components being used for form display and organization [1].

The `Form.Control` component acts as a wrapper on the HTML `<input/>` and takes in a `type` prop that defines the HTML 5 input type that will be rendered with Bootstrap styling. These form controls typically are written explicitly within a `Form` component wrapper that provides default HTML form submission behavior when a `Button` of `type="submit"` is included in the Form. An example login form from the React Bootstrap documentation is displayed below:

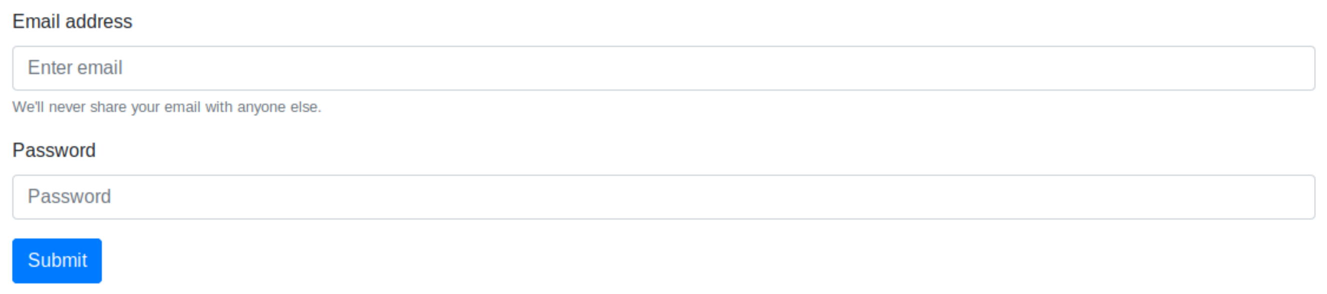


Figure 1: The sample two-field login form rendered with default React Bootstrap styling.

```
<Form>
  <Form.Group controlId="formBasicEmail">
    <Form.Label>Email address</Form.Label>
    <Form.Control type="email" placeholder="Enter email" />
    <Form.Text className="text-muted">
      We'll never share your email with anyone else.
    </Form.Text>
  </Form.Group>

  <Form.Group controlId="formBasicPassword">
    <Form.Label>Password</Form.Label>
    <Form.Control type="password" placeholder="Password" />
  </Form.Group>
  <Button variant="primary" type="submit">
    Submit
  </Button>
</Form>
```

As seen in this example, even one of the simplest possible form implementations requires a fair degree of setup and knowledge of the details of the React Bootstrap framework in order to structure the form input groups and labels for the fields. This complexity increases even further when dealing with more diverse form inputs or when including additional information or actions on a field with the `InputGroup` component [4]. Furthermore, the usage of the `Form` wrapper component locks the form data submission into a standard HTTP `GET` or `POST` request when the submit button is clicked, inhibiting any customized data parsing or API interfaces.

If more control over these server calls is desired, the `Form.Control` components can be used without an enclosing `Form`, yet in this case the developer must implement their own data storage layer in order to track the changes in each input in order to submit the updated data to the back-end. Both of these workflows for using the React Bootstrap `Form` components demand extensive manual setup to execute common form tasks, implying the need for a more robust React form implementation to handle these design challenges.

1.2.2 Formik

The Formik library aims to standardize these recurring form patterns by addressing three main issues: ”keeping track of values/errors/visited fields, orchestrating validation, and handling submission” [2]. The primary export from the library is the `<Formik/>` component, which exposes props for custom submit functionality, passing in initial field values, and providing a schema to validate the values entered into the form. The library relies upon the React render props [5] coding style to pass a standard HTML form into the `Formik` component, which exposes the submission action, initial values, and error handling options to this inner `<form/>` and its inputs. An example Formik implementation is included below:

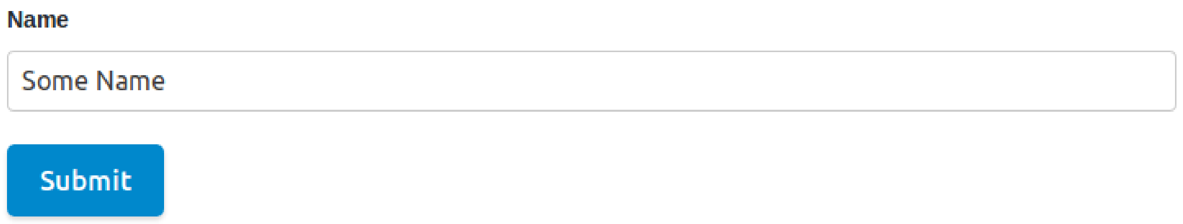


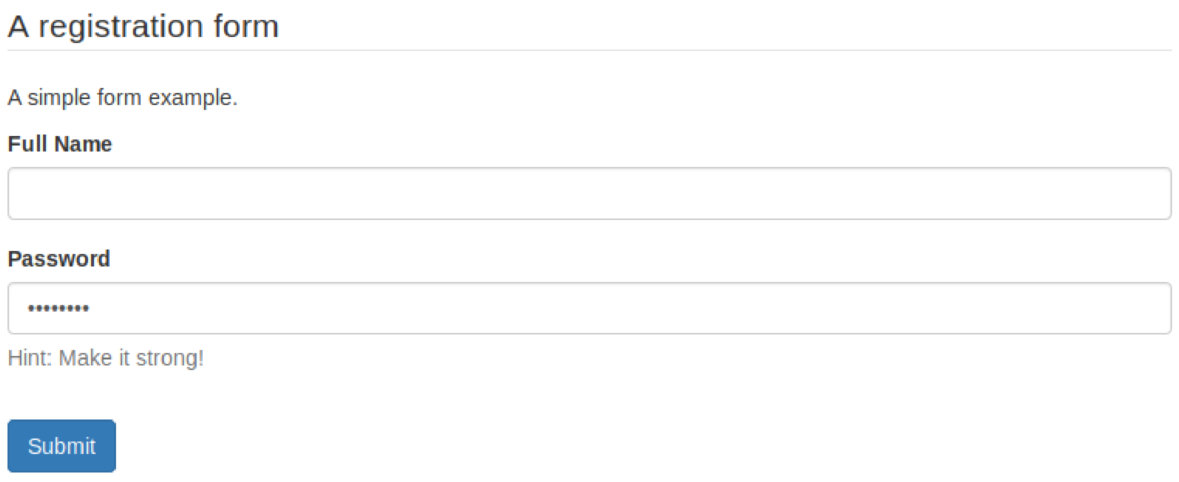
Figure 2: The sample one field form rendered using the `Formik` component.

```
<Formik
  initialValues={{ name: 'Some Name' }}
  onSubmit={(values, actions) => {
    setTimeout(() => {
      alert(JSON.stringify(values, null, 2));
      actions.setSubmitting(false);
    }, 1000);
  }}
  render={props => (
    <form onSubmit={props.handleSubmit}>
      <label htmlFor="name" style={{ display: 'block' }}>
          Name
        </label>
      <input
        type="text"
        onChange={props.handleChange}
        onBlur={props.handleBlur}
        value={props.values.name}
        name="name"
      />
      {props.errors.name && <div id="feedback">{props.errors.name}</div>}
      <button type="submit">Submit</button>
    </form>
  )}
/>
```

While Formik provides far more potential for form customization than a standard component library such as React Bootstrap, it is apparent from the above example that the required setup to leverage this functionality places a significant burden on the developer. The usage of the render props pattern establishes a harsh learning curve for library usage by necessitating the developer to learn all of the props that are passed down into the nested `<form/>`. Additionally, Formik still requires each input and label within the form to be written explicitly in the order that they will be rendered to the page, leading to a large degree of repetitive code and lengthy component definitions when dealing with longer form implementations.

1.2.3 react-jsonschema-form

`react-jsonschema-form` aims to provide a more dynamic approach to React form development by "automatically generat[ing] a React form based on a JSON schema" [3]. The library employs the JSON Schema draft-07 specification [6] as the standard input that is parsed into the form to construct field components. Overall, the project presents a major improvement over solutions such as Formik by removing the need to write each form input as a React component or HTML field and instead populating the form based upon a simple JSON object that can quickly be edited or updated. The `Form` component also takes in a `formData` prop that is expected to match the schema object format and will populate the rendered fields with any intitial values. A simple example form is presented below:


A registration form

A simple form example.

Full Name

Password

••••••••

Hint: Make it strong!

Figure 3: The sample registration form rendered with `react-jsonschema-form`.

```
const schema = {
  title: "A registration form",
  description: "A simple form example.",
  type: "object",
  properties: {
    name: {
      type: "string",
      title: "Full Name"
    },
    password: {
      type: "string",
      title: "Password",
      minLength: 3
    }
  }
};

const uiSchema = {
  password: {
    "ui:widget": "password",
    "ui:help": "Hint: Make it strong!"
  }
};

const formData = {
  password: "passw0rd"
};

render(
    <Form schema={schema} formData={formData} uiSchema/>
);
```

This schema-based form generation presents a significantly more streamlined and extensible React form development style, and is provided with additional customization options through the use of the UI schema object, which exposes the `ui:widget` flag to change the HTML 5 input type for a generated field from the default text input, number input, or checkbox for boolean schema values. The above example demonstrates this functionality for a password field and shows how the `ui:help` flag can be used to render help text beneath the input.

In spite of the many benefits provided by `react-jsonschema-form`, the library has several key limitations that prevent it from functioning in many common use cases. The first main restriction is that each `<Form/>` component exists as a self-contained entity that is required to render a standard HTML `type="submit"` button at the bottom of each form, undercutting the potential for custom server interfaces as seen in Formik. The component directly `POST`s a simple JSON object containing the form data keyed by field id to the specified endpoint, preventing any customized parsing code for either individual inputs or the entire form.

The other significant constraint within the library is the difficulty in adding any customized field components to provide a more dynamic user experience with more complex input interfaces. By default, the framework is only capable of rendering the standardized HTML 5 `<input/>` fields, and any additional input components must be passed in as alternative widget options [7]. An example of this custom widget registration is provided below:

```
const MyCustomWidget = (props) => {
  return (
    <input type="text"
      className="custom"
      value={props.value}
      required={props.required}
      onChange={(event) => props.onChange(event.target.value)} />
  );
};

const widgets = {
  myCustomWidget: MyCustomWidget
};

const uiSchema = {
  "ui:widget": "myCustomWidget"
}

render((
  <Form
    schema={schema}
    uiSchema={uiSchema}
    widgets={widgets} />
), document.getElementById("app"));
```

While this pattern does add some potential for input diversity to `react-jsonschema-form`, the lack of support for custom form data parsing greatly inhibits more advanced uses of this capability. Even a simple geolocation input with separate latitude and longitude fields that function together to produce a single output string demands a complex input component definition to handle the parsing internally, and the sample implementation of this component (provided on `Custom` tab of the library's live playground site [8]) has significant performance issues and often causes the browser to lag when an input is first provided. Overall, these deficiencies highlight the status of `react-jsonschema-form` as a promising but limited approach to the problem of simplifying the development of forms in React, leaving open the potential for a more robust schema-driven form library.

2 Formative Philosophy

When setting out to design the Formative React library, the primary goal was to build upon the best features of earlier React form frameworks while providing a much more extensible and flexible approach to avoid the key limitations described in detail above in Section 1.2. A critical part of the Formative design philosophy was completely decoupling the JSON schema parsing and form generation from the definitions of the form input components such that new inputs could be added incrementally without any modifications to the core library, facilitating open source contribution to continually grow the library and integrate more advanced input types.

2.1 Schema-Based Design

2.1.1 Data and UI Separation

One of the central design objectives for the Formative library was to improve upon the definitions of the data schema and UI schema from `react-jsonschema-form`, defining a clearer separation of concerns between the two JSON objects. Within Formative, the JSON schema serves to define the set of all possible form fields and input types that *can be* displayed in the form, while the UI schema describes the input type used for each field as well as options that directly affect user interactions. In contrast to the `react-jsonschema-form` strategy of also employing the UI schema to pass in basic styling options for the inputs, the form generation layer of the library provides each component on the page with a standardized CSS identifier that can be used to provide much more powerful and extensive options for visual customization.

2.1.2 Inclusion List

Another major design decision for the project was the definition of an `includeFields` list such that only the specified fields are rendered to the page. This feature diverts significantly from the `react-jsonschema-form` model of automatically displaying every field within the JSON schema to the page, yet provides far greater flexibility for usage with existing schema definitions. In particular, many existing JSON schema definitions such as the Project Open Data metadata schemas [9] feature dozens of fields, including some internal fields that are not intended to be edited by a user and instead should only be visible within the back-end. In `react-jsonschema-form`, constructing a form that used a subset of fields from one of these complex schemas would require the developer to define their own JSON schema including just the fields that will be displayed within the form; this demands a far greater setup effort and may even cause difficulties in interfacing with existing server code that uses the complete schema for validation. The `includeFields` list allows for the full original schema to be used directly with no additional setup, and only the desired fields will be rendered as editable form inputs.

2.2 Input Component Extensibility

2.2.1 Open Source Development

The other main tenet of the Formative design philosophy was the ability for the framework to leverage the full potential of collaborative, open source development to provide an expanding library of input components. A critical aspect of this design pattern is ensuring that new inputs can be contributed to the repository without making any changes to the schema parsing or form generation code, which could break existing form implementations. In order to achieve this objective, all references to available input components were contained within the `inputTypeMap` and `reactInputMap` files, which provide a generic interface such that the schema parser has no knowledge of any inputs except for the 4 data type defaults (`TextInput` for strings, `NumberInput` for numbers, `CheckboxInput` for booleans, and `SelectInput` for enumerated values). The `inputTypeMap` is used by the schema parser to determine which input types are capable of processing each JSON schema data type, adding a layer of protection against UI schema errors. The `reactInputMap` is the primary file that exposes all input components within the repository to the form generation layer such that they can be referenced from the UI schema and rendered in the page. In order to contribute a new component to the repository, a user must first add the component to these two map files such that it can be exposed to the form generator in future `Form` components.

2.2.2 Locally Defined Inputs

While the open source model provides a significant degree of flexibility and extensibility, it still has the inherent limitation that a developer seeking to use a custom component within their form must make a pull request in order for it to be included within the `reactInputMap` file. In some instances this might not be desirable or even possible, including in the cases where the input components may be proprietary or the use case calls for a different component framework such as Material UI instead of React Bootstrap. To account for these scenarios, an additional `customInputMap` prop was included within the `Form` and `FormPage` components that allows for the user to define unique key-to-component mappings that will be merged with the `reactInputMap`. This object is passed into the form locally, so it can reference input components from within the current application context without any integration with the main repository. The `customInputMap` also provides the capability for the developer to overwrite the definitions of the main library components if they wish to customize the underlying React definitions. This fully genericizes the library to not rely explicitly upon React Bootstrap and allow for other UI frameworks to be used for the base components, while still preserving the core form generation functionality so that it can be employed in these alternative use cases.

3 Formative React Design

In order to achieve the goal of a fully generic form generation engine, the library was structured to rely upon discrete schema parsing and form rendering layers that interfaced with the JSON schema and UI schema through authoritative `inputMaps` that exposed all usable input components.

3.1 Schema Parsing Overview

The `SchemaParser` functions by taking in a JSON schema and UI schema, validating the JSON schema against the draft-07 standards, and building a nested object describing all of the fields that can be used on the page, the types of input components they should be rendered, and any properties specific to those inputs. Another central goal in the design of the `SchemaParser` was ensuring that it was robust and would not crash when provided with a malformed JSON schema; this demanded extensive unit testing to cover a wide variety of failing schema formats, as well as establishing an error callback pattern to return relevant information when a schema fails to adhere to the draft-07 specification. Additionally, the `SchemaParser` was designed to protect against a developer requesting an input type in the UI schema that is incompatible with that field's data type (for example, attempting to use a boolean `CheckboxInput` on numerical data), providing default inputs for each JSON Schema data type that are used when there is no component listed in the UI schema or the chosen input does not comply with the data model.

Another significant challenge within the schema parsing design was handling the `anyOf` and `oneOf` keys that are used to combine multiple schemas. Within `react-jsonschema-form`, these complex options were both handled by providing a dropdown for the user to select between the different schemas and the relevant fields were then populated below the dropdown. While this approach can be effective for simple schemas, it quickly becomes inefficient and overcomplicated for larger schemas such as the Project Open Data dataset schema which contains a total of 19 `anyOf` fields [10]. Furthermore, although the ability to select between schemas can be user-friendly if the options are provided with informative titles, many industry-standard schemas use `anyOf` and `oneOf` to specify data formatting options that the user should not be expected to understand or have to choose between.

In line with the Formative philosophy that the JSON schema outlines the extent of possibilities that the front-end developer can display in the form, the `anyOf` and `oneOf` layers were reinterpreted as a set of options that can be selected in the UI schema using the `ui:format` flag. As a result, the `anyOf` and `oneOf` parsers will each return a single field chosen within the UI schema. This selection can be performed by either matching on the string `title` attribute for named subschemas or by matching the entire schema object to account for anonymous subschemas that merely include a text pattern or parameterized information. Additionally, these parsers automatically handle any nullable `anyOf` or `oneOf` fields that contain one standard subschema and one null subschema by directly rendering the non-null field. Due to the nature of form inputs this data value can be made null simply by clearing the input field, satisfying the schema fully while streamlining the development process for these nullable fields.

3.2 Form Generation Overview

The form generation process was divided into several steps such that each serves a distinct function and each phase could be developed independently without interfering with the others. The first stage passes the JSON schema and UI schema into the `SchemaParser` to return a nested object containing the input types and any of the visual props for each field. The second phase sends this parsed schema to an instance of the `FormGenerator` class that has been initialized with any default form data, the `includes` list, and optionally a `customInputMap` if any input components from outside the library will be used. Rather than existing as a React component, the decision to structure `FormGenerator` as a plain JavaScript class with methods that return React inputs interrupts the standard render cycle and ensures that updates can occur within the input fields without causing the entire form to be regenerated. The main entry point for this class is the `generateForm()` method, which takes in a parsed schema object, recursively traverses the groups of inputs with the `generateInputList()` and `generateInputFormat()` methods, and retrieves the React component for each field using the `reactInputMap`. The `generateForm()` method also accounts for any custom inputs that are passed in with the `customInputMap` argument, overwriting any keys that are defined in both maps so that the developer can provide their own local definitions of base inputs.

Each field returned from the `FormGenerator` is wrapped in an `InputFormField` React component which handles the setup of the update and save handlers on each input instance. The main complexity in this section is introduced when handling updates for array type fields, which must retain a reference to their index within the array of inputs so that the appropriate value within the form data array can be changed. This process is facilitated by passing the `setFormData` method defined in the top level `Form` component into each of the `InputFormFields`, then using the `setFormData(prevData => )` callback syntax to edit the data at the correct location in the array.

The final step of form generation occurs in the `Input` component, which lays out the input component, field save button, as well as optional label, description, and hint hover text. The input component itself is created using the React feature that allows component types to be stored within variables and rendered to the DOM by wrapping the variable in angle brackets. The `Input` component takes in a prop `Type` which must be a React component that takes in the three required input field props of `id`, `initialValue`, and an `onUpdate` handler, and all input-specific props are passed in with a {`...rest`} spread operator.

```
<Type
  id={id}
  name={id}
  initialValue={initialValue}
  onUpdate={onUpdate !== undefined ? (newValue) =>
    handleUpdate(newValue) : undefined}
  {...rest}
/>
```

3.3 State Management Overview

A fundamental tension when working with HTML forms through React is that HTML inputs are uncontrolled and by default merely store the value typed into them. By contrast, React development typically supports a design pattern of using controlled components that interface with some state-containing parent through value props and update callbacks. In the case of form inputs, this implies that a successful form implementation must be able to pre-populate all of its child inputs with initial values and pass update handlers to each of the children in order to update the main data store each time an input value is changed. A controlled component would pass in the initial value as a variable into a value prop (`value={initialValue}`) and would ensure that the value of the variable updates each time an input action is fired such that the field maintains the correct value. By contrast, uncontrolled inputs merely require a constant initial value to be passed in (`initialValue={initialValue}`) and the form field is left to store its own value while still firing update handlers to maintain the accurate state in the parent data store. While both methods of component design have the same result at the `Form` component level in terms of returning the current state in all of the fields, the uncontrolled approach is actually more efficient since it requires less rerenders of the input components. By default React components rerender when any of their props or state variables change, so controlled components will be required to rerender on every keypress or input event since their value prop will have to update to maintain the correct visual state. This also forces rerenders of the components in the parent tree that the new value prop passes through, which can be incredibly inefficient for larger forms with many inputs. As a result, the uncontrolled input design is enforced by the static `initialValue` prop required in all input components.

3.3.1 Update State Tracking

One of the most important form states to track beyond the field data is the information about which inputs have been edited by the user since the last save or load event, which determines the fields and sections that should have active save buttons. This is best implemented as an object containing the keys in the form, with a `true` value for each of the fields that have been updated. Under this model, any input with a `true` value in the update dictionary should have its single-field save button enabled, and if any of the fields within the form have a `true` value then the form's save button should be enabled as well. The first challenge with tracking updates arises due to the implementation of the `FormGenerator` as a pure JS class rather than a React component, since changes to the updated dictionary would not propagate through to the inputs by default. In order to pull the state outside of the typical props flow, the updated dictionary and a dispatch method that fired a reducer were both passed into a React context provider node and each input component was able to access the latest value of the dictionary with the `useContext()` hook. However, due to the nature of storing this data in an object, any update to any of the inputs will cause a change in the updated object and as a result would trigger a rerender of every input due to the new prop value. These unnecessary render cycles were avoided by wrapping each input group in a `useMemo()` hook, which caches the inner components and only causes the children to rerender if a specified value changes (in this case the `updated` flag for that input field). These optimizations combined to make the form far more efficient while providing reliable update state management.

3.4 API Interface Overview

The API layer for the library was implemented in a wrapper on top of the base `Form` component, such that `Form` included all of the core library functionality of schema parsing and form generation and the default API implementation could be optionally adopted through the `FormPage` component. This default API structure is intended to provide a standard interface for constructing a simple full stack application that will work with the Formative library. The main features required for such a back-end are the existence of static `schema.json` files that can be retrieved from an endpoint, the ability to make an HTTP `GET` request to the `/schema/` endpoint to retrieve an object containing all of the form data, and the ability to `POST` a form data object to one of these schema endpoints and recieve a confirmation that the save was successful. Any networking errors that arise during these operations will be relayed back to the parent component through the `onError` callback, which is left up to the developer to implement for their use case. A sample implementation could pass the error message from the callback into a modal dialogue to alert the user of the issue, although many more advanced possibilities are feasible as well. In the case where the application back-end requires a more complex data transfer protocol or the developer is using an alternative API format, it is recommended to use the `APISchemaForms` as a guide to construct a more advanced API wrapper for the `Form` component which can then be used locally in the project.

3.5 Input Design Overview

In order to make the input components completely modular, a consistent prop structure had to be established for all inputs included within the library so that the critical functionality could be automatically provided by the form generator. Using Flow static type checking [11], all input components are expected to take in a unique string `id` for the CSS generation, an `initialValue` to populate the field with when the form first loads, and an `onUpdate` handler that sends back the new value entered into the field such that the main form data state can remain in sync with the inputs. In order to standardize the formatting of all input fields and simplify the creation of new input components, *the label, description text, hint text, and field save button* are all automatically provided within the `Input` wrapper component and the developer only needs to implement the data entry field.

3.5.1 Testing and Static Type Checking

In an effort to ensure the consistent quality of the inputs that are included within the main Formative repository, all new inputs components are expected to be covered by extensive unit testing and employ the Flow static type checking library to support better type safety. Unit testing for all components in the project is powered by Jest and Enzyme, with input tests relying upon the Enzyme shallow renderer to access component props and test event handlers through simulated events [12, 13]. All inputs are expected to have unit tests proving that the component can render without crashing, that it passes through the `id` and `initialValue` props correctly, and that new values are sent back on update events in the nested children. The usage of Flow provides an additional layer of uniformity to the library, as all new inputs are expected to take in one of the standard `propType` definitions provided in the `inputFlowTypes` file.

3.5.2 Autocomplete Input Example

While many of the base library inputs such as the `TextInput` are merely wrappers on React Bootstrap `Form.Control` components, the real power of Formative is demonstrated through more complex fields such as the `AsyncAutocompleteInput`. This field leverages the modular nature of the library and the ability to pass in specialized props through the UI schema to provide an interface that can generate asynchronous autocompletion suggestions from a remote API simply by specifying the API url and the name of the key to filter on. At the level of the form this component is no different from a standard `TextInput` and still sends back updates when the field value is changed, yet the component is able to have this powerful internal functionality to handle search debouncing and cache results for better performance.

4 Programmatic Work Division

In addition to the immediate benefits of the React Formative framework in front-end development, the library also encourages a much more balanced distribution of labor between each member or group in a development team. Specifically, the use of Formative enables the separation of the back-end, front-end functionality, and visual design aspects of development such that each can be worked on by a dedicated developer or team. Even in the case where the entire full stack application is being coded by a single developer, this separation of concerns helps streamline the overall development process and mitigate error-prone tight connections that can limit extensibility and hamper future updates. Overall, the Formative framework outlines a development pattern for decoupling these core application components and facilitating modular, data-driven development with the potential for robust maintenance and update cycles.

4.1 Interface Structure Defined by Back-end

One central issue in web development is the creation of sufficient front-end functionality to satisfy all of the needs of the server and populate all of the relevant data. In a typical workflow, this will require a back-end developer to pass database requirements or a sample API call to the front-end team to inform the fields that should be displayed in a page or define the data transfer format. In addition to forcing the front-end to be manually defined in terms of this specification, this development pattern can also be highly fallible since any mistakes in the front-end will cause malformed data to be returned that will either be rejected by the back-end or cause further problems on the server side. The Formative framework enforces adherence to these back-end specifications by only allowing the generation of fields that are defined within the JSON data schema and standardizing the data intake and return formats to prevent the introduction of developer errors that are difficult to catch and debug. By re-contextualizing the role of the API as defining the set of all possible input fields that the front-end developer can choose to display, Formative ensures that the data returned from the form is compatible with the back-end structure. Furthermore, this prevents the back-end developer from being forced to interface directly with the front-end in any way beyond providing their existing JSON schema as input to the `Form` component, forestalling the need for a back-end developer to take on front-end responsibilities in order to establish conformance with their server model.

4.2 Extensibility Driven by Front-end

While the creation of front-end functionality driven by the back-end is a powerful tool on its own, there must be a sufficient degree of customizability for the front-end developer or else the use cases for the framework may be greatly restricted. The core of the Formative philosophy was built around extending the data-driven form creation originally seen in the `react-jsonschema-form` project and providing the potential for a far wider set of inputs that the React developer can select between with a single line of code. React actually promotes a modular component structure that directly enables the swapping of components that share standardized prop formats, facilitating the introduction of an extensive range of input options to satisfy a single data type. The primary goal of the React developer should be constructing the functionality of an application and specifying the methods of user interaction with the fields on the page, and Formative allows them to focus solely on these tasks and avoid dealing with the intricacies of the data model or data parsing operations to correctly populate the various fields within the app.

4.3 Full Visual Customizability for Designer

The final area of web development efforts that is often overlooked within React frameworks is support for designers to style the components on the page with CSS rather than through style properties contained within the JavaScript code. The primary dilemma with the latter approach is that it typically forces the styling work to be done by a front-end developer rather than a designer, who would have limited experience dealing with inline styles and may struggle to apply the appropriate React props. While component libraries such as React Bootstrap provide some CSS customization options through the default Bootstrap CSS hooks, this does not provide any mechanism for a designer to access the children of a higher order component and modify their position or visual appearance. The Formative library introduces a set of standardized CSS ids that are procedurally generated for every element within the `Form` component and use a suffix containing a hyphen and capitalized field type to avoid conflicts with existing ids in an application. Since these ids are generated based upon the unique keys for each field within the form, the designer is able to access the ids for every element simply by viewing the keys in the form's `includes` list. As a result, the designer does not need to edit any of the React code and can produce their designs completely independently through style sheets, providing better separation of concerns for the functionality and visual appearance of the application.

References

[1] React Bootstrap: Forms. https://react-bootstrap.github.io/components/forms/. Accessed: 2019-08-12.

[2] Formik. https://jaredpalmer.com/formik/. Accessed: 2019-08-12.

[3] react-jsonschema-form. https://react-jsonschema-form.readthedocs.io/en/latest/. Accessed: 2019-08-12.

[4] React Bootstrap: Input Group. https://react-bootstrap.github.io/components/input-group/. Accessed: 2019-08-12.

[5] React: Render Props. https://reactjs.org/docs/render-props.html. Accessed: 2019-08-12.

[6] JSON Schema. http://json-schema.org/. Accessed: 2019-08-12.

[7] react-jsonschema-form: Custom widgets and fields. https://react-jsonschema-form.readthedocs.io/en/latest/advanced-customization/#custom-widgets-and-fields. Accessed: 2019-08-12.

[8] react-jsonschema-form: Live Playground. https://mozilla-services.github.io/react-jsonschema-form/. Accessed: 2019-08-12.

[9] Project Open Data: Metadata Schemas. https://project-open-data.cio.gov/v1.1/schema/. Accessed: 2019-08-14.

[10] Project Open Data: Dataset Schema. https://project-open-data.cio.gov/v1.1/schema/dataset.json. Accessed: 2019-08-15.

[11] Flow Type Checker. https://flow.org/en/. Accessed: 2019-08-15.

[12] Jest. https://jestjs.io/en/. Accessed: 2019-08-15.

[13] Enzyme. https://airbnb.io/enzyme/. Accessed: 2019-08-15.