JSONForms

Introduction

JSONForms.io is the official website and community hub for the JSONForms framework. It serves as a one-stop shop for everything related to building web forms with JSON Schema:

Key Features:

- Documentation: Comprehensive documentation covering the core JSONForms library, integrations with various UI frameworks like React, Angular, and Vue, tutorials, and API references.
- Demos: Interactive demos showcasing the capabilities of JSONForms for building different types of forms.
- Playground: An online playground where you can experiment with JSON Schema and UI Schemas to see how your forms would look and behave.
- Community: A forum and discussion board for developers to ask questions, share their experiences, and contribute to the development of the framework.
- Learning Resources: Blog posts, articles, and video tutorials covering various aspects of JSONForms development.
- Packages and Integrations: Links to external libraries and tools that work with JSONForms, expanding its functionality and applicability.

Essentially, JSONForms.io acts as a central hub for:

- Learning about JSONForms: Whether you're a beginner or experienced developer, the website provides resources to get you up to speed on the framework's core concepts and functionalities.
- Building forms with JSONForms: You can find everything you need to create efficient and flexible forms, from understanding JSON Schema to customizing the UI.

- Connecting with the community: Ask questions, share your projects, and get help from other JSONForms users and developers.

Dependent Dropdowns

To implement dependent dropdowns in React using JSON Forms, you would typically follow these steps:

1. Define Your Schema: First, define your JSON schema. This schema specifies the structure of your data, including the properties for each dropdown.

2. Create UI Schema: The UI schema defines how your form should be rendered. Here, you can specify which fields are dropdowns and how they are related.

3. Implement Dependency Logic: You need to handle the logic for dependent dropdowns. This usually involves updating the state of a dropdown based on the selection in another.

4. Handle State and Data Binding: Manage the state of your form, ensuring that the dropdown values are correctly set and updated in response to user interactions.

5. Rendering with JSON Forms: Use JSON Forms to render the form based on your schemas. JSON Forms will take care of generating the necessary UI elements.

6. Data Validation: Optionally, you can add validation rules to ensure that the user's input is correct.

Here is a basic example to illustrate these steps:

```javascript
import React, { useState } from 'react';
import { JsonForms } from '@jsonforms/react';
import { materialRenderers, materialCells } from '@jsonforms/material-renderers';

const schema = {
  type: 'object',
  properties: {
    country: {
      type: 'string',
      enum: ['USA', 'Canada', 'Mexico']
    },
    city: {
      type: 'string',
      enum: []
    }
```

```
  }
};

const uischema = {
  type: 'VerticalLayout',
  elements: [
    {
      type: 'Control',
      scope: '#/properties/country'
    },
    {
      type: 'Control',
      scope: '#/properties/city'
    }
  ]
};

const cities = {
  USA: ['New York', 'Los Angeles', 'Chicago'],
  Canada: ['Toronto', 'Vancouver', 'Montreal'],
  Mexico: ['Mexico City', 'Guadalajara', 'Monterrey']
};

const App = () => {
  const [data, setData] = useState({});

  const handleChange = (event) => {
    const newData = { ...data, [event.data.property]: event.data.value };

    if (event.data.property === 'country') {
      // Update cities based on the selected country
      schema.properties.city.enum = cities[event.data.value] || [];
      newData.city = ''; // Reset city when country changes
    }

    setData(newData);
  };

  return (
    <JsonForms
      schema={schema}
      uischema={uischema}
      data={data}
      renderers={materialRenderers}
```

```
      cells={materialCells}
      onChange={handleChange}
    />
  );
};

export default App;
```

In this example, when a user selects a country, the `city` dropdown updates its options based on the selected country. This is achieved by updating the `enum` property for the `city` field in the schema. The `handleChange` function manages this logic and updates the state accordingly.

Custom Error Validations

To implement form validation with custom error messages in React using JSON Forms, you will need to follow these steps:

1. Define JSON Schema: Start with your JSON schema which describes the data structure, including the fields you want to validate.

2. Add Validation Keywords: In your JSON schema, use validation keywords like `minLength`, `maxLength`, `pattern`, etc., for standard validations.

3. Custom Error Messages: JSON Forms uses AJV for validation, which supports custom error messages. You can define a custom AJV instance with your own error messages.

4. Pass AJV Instance to JSON Forms: Inject your custom AJV instance into JSON Forms so that it uses this for validation.

5. Handle Form State and Display Errors: Manage the state of your form and display the error messages as needed.

Here's an example to illustrate these steps:

Step 1: Define JSON Schema

```javascript
const schema = {
  type: 'object',
  properties: {
    username: {
```

```
      type: 'string',
      minLength: 3
    },
    age: {
      type: 'number',
      minimum: 18
    }
  },
  required: ['username', 'age']
};
```

Step 2: Customize AJV with Custom Error Messages

First, install AJV and AJV-errors:

```bash
npm install ajv ajv-errors
```

Then, create a custom AJV instance:

```javascript
import Ajv from 'ajv';
import addAjvErrors from 'ajv-errors';

const ajv = new Ajv({ allErrors: true, jsonPointers: true });
addAjvErrors(ajv);

// Custom error messages
ajv.addKeyword('errorMessage', {
  modifying: true,
  compile: (errors, parentSchema) => {
    return (data, dataPath, parentData, parentDataProperty) => {
      parentData[parentDataProperty] = errors[parentSchema.type];
      return false;
    };
  },
  errors: 'full'
});

// Update schema with custom error messages
schema.properties.username.errorMessage = {
  minLength: 'Username must be at least 3 characters long.'
```

```
};
schema.properties.age.errorMessage = {
  minimum: 'You must be at least 18 years old.'
};
```

 Step 3: Pass Custom AJV to JSON Forms

When rendering your form with JSON Forms, pass the custom AJV instance:

```javascript
import { JsonForms } from '@jsonforms/react';
import { materialRenderers, materialCells } from '@jsonforms/material-renderers';

const App = () => {
  const [data, setData] = useState({});

  return (
    <JsonForms
      schema={schema}
      data={data}
      renderers={materialRenderers}
      cells={materialCells}
      onChange={({ errors }) => {
        // Handle form data and display errors
      }}
      ajv={ajv} // Pass the custom AJV instance
    />
  );
};

export default App;
```

In this setup, when a validation error occurs, AJV uses the custom error messages defined in the schema. The `onChange` handler of JSON Forms receives these errors, which you can then use to display custom error messages on your form.