Custom hooks in React are functions that allow you to reuse stateful logic across different components. They are particularly useful for managing complex state logic in a clean and modular way, promoting code reusability and maintainability. Custom hooks are a powerful tool for abstracting and sharing logic without the need for higher-order components or render props.

Here are some reasons why custom hooks are useful:

1. **Reusability:** Custom hooks encapsulate logic that can be reused across different components. This promotes a DRY (Don't Repeat Yourself) coding philosophy, as you can write the logic once and use it in multiple places.
2. **Separation of Concerns:** Custom hooks help in separating the logic from the presentation. This makes your components more focused on rendering and UI concerns, while the custom hooks handle the underlying logic.
3. **Testability:** Since custom hooks are just functions, they can be easily tested in isolation. This makes it easier to write unit tests for your application logic.

Now, let's look at examples of three custom hooks:

1. useFetch

```
import { useState, useEffect } from 'react';

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        const result = await response.json();
        setData(result);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
};

// Usage
// const { data, loading, error } = useFetch('https://api.example.com/data');

import React from 'react';
import { useFetch } from './useFetch'; // Assuming the file is in the same directory

const DataFetchingComponent = () => {
  const { data, loading, error } =
useFetch('https://jsonplaceholder.typicode.com/todos/1');

  return (
    <div>
      <h1>Data Fetching Example</h1>
      {loading && <p>Loading...</p>}
      {error && <p>Error: {error.message}</p>}
      {data && (
        <div>
          <h2>Data:</h2>
          <pre>{JSON.stringify(data, null, 2)}</pre>
```

```
        </div>
      )}
    </div>
  );
};

export default DataFetchingComponent;
```

2. useForm

```
import { useState } from 'react';

const useForm = (initialValues) => {
  const [values, setValues] = useState(initialValues);

  const handleChange = (e) => {
    const { name, value } = e.target;
    setValues((prevValues) => ({
      ...prevValues,
      [name]: value,
    }));
  };

  const resetForm = () => {
    setValues(initialValues);
  };

  return { values, handleChange, resetForm };
};

// Usage
// const { values, handleChange, resetForm } = useForm({ username: '', password: '' });

// Example Component
import React from 'react';

const MyFormComponent = () => {
  const { values, handleChange, resetForm } = useForm({ username: '', password: '' });

  const handleSubmit = (e) => {
    e.preventDefault();
    // Handle form submission logic with the values object
    console.log('Form submitted with values:', values);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Username:
        <input type="text" name="username" value={values.username} onChange={handleChange}
/>
      </label>
      <br />
      <label>
        Password:
```

```
        <input type="password" name="password" value={values.password} onChange=
{handleChange} />
      </label>
      <br />
      <button type="submit">Submit</button>
      <button type="button" onClick={resetForm}>
        Reset
      </button>
    </form>
  );
};

export default MyFormComponent;
```

3. useToggle

Certainly! Here's an example of a `useToggle` custom hook that manages a boolean toggle state:

```
import { useState } from 'react';

const useToggle = (initialValue = false) => {
  const [value, setValue] = useState(initialValue);

  const toggle = () => {
    setValue((prevValue) => !prevValue);
  };

  return [value, toggle];
};
```

In this example, the `useToggle` hook returns an array with two elements: the boolean state (`isToggled`) and a function (`toggle`) to toggle the state between `true` and `false` . The initial value can be specified as an argument (default is `false`).

Usage example in a component:

```jsx
import React from 'react';
import { useToggle } from './useToggle'; // Assuming the file is in the same directory

const ToggleComponent = () => {
  const [isToggled, toggle] = useToggle(false);

  return (
    <div>
      <p>Toggle State: {isToggled ? 'On' : 'Off'}</p>
      <button onClick={toggle}>Toggle</button>
    </div>
  );
};

export default ToggleComponent;
```

These examples demonstrate custom hooks for managing fetching data, and handling form state and toggling. You can tailor these hooks to suit the specific needs of your application and reuse them across different components.