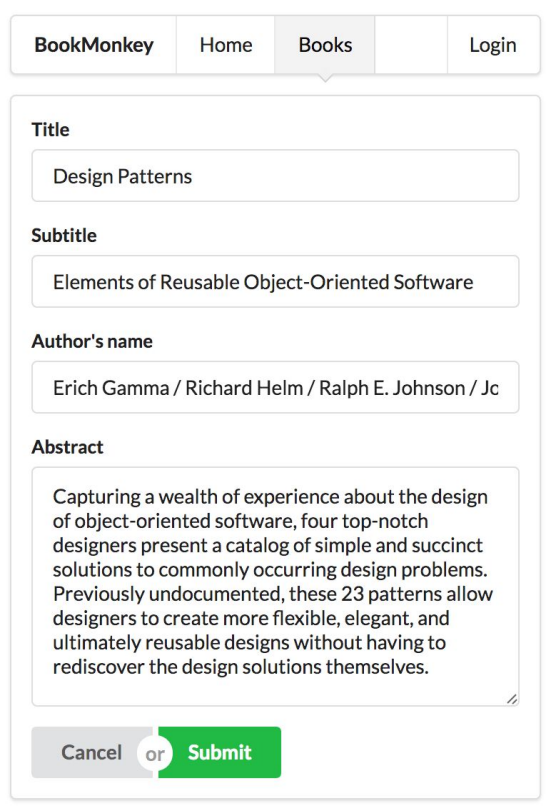
Workshop

React Forms

Why / What you'll learn

- How to access HTMLElements from react with the `useRef` Hook
- Create forms with React
- How to add instant-feedback input validation to a form

Handle user input in your application



Using common HTML elements, we can create complex forms to retrieve user input.

Forms

<code>

A simple HTML5 form

In react, this is called `htmlFor` instead of `for`

```
<form>
  <label for="title">Title:</label>
  <input type="text" id="title" name="title">

  <button type="submit">Save</button>
</form>
```

Who controls the state of the user input?

- With plain HTML:
  - form elements (`input`, `textarea`, ...) control their own state
- With React, we have two options:
  - “just HTML”: form elements control the state (**uncontrolled components**)
  - React controls the state (**controlled components**)

Who controls the state of the user input?

- With plain HTML:
  - form elements (`input`, `textarea`, ...) control their own state
- With React, we have two options:
  - "just HTML": form elements control the state (**uncontrolled components**)
  - React controls the state (**controlled components**)

this is preferred

useRef & uncontrolled Components

useRef

<code>

Adding a ref on a react element allows us access to the underlying HTMLElement

```
import { useRef } from "react";

export const MyComponent = () => {
  const buttonRef = useRef<HTMLButtonElement>(null);

  const onClick = () => {
    console.log(buttonRef.current!.innerHTML);
  };

  return <button ref={buttonRef} onClick={onClick}>Click me!</button>;
};
```

useRef

<code>

Adding a ref on a react element allows us access to the underlying HTMLElement

```
import { useRef } from "react";

export const Button = () => {
  const buttonRef = useRef<HTMLButtonElement>(null);

  const onClick = () => {
    console.log(buttonRef.current!.innerHTML);
  };

  return <button ref={buttonRef} onClick={onClick}>Click me!</button>;
};
```

Create a new ref, that's initially set to null

useRef

<code>

Adding a ref on a react element allows us access to the underlying HTMLElement

```
import { useRef } from "react";

export const Button = () => {
  const buttonRef = useRef<HTMLButtonElement>(null);

  const onClick = () => {
    console.log(buttonRef.current!.innerHTML);
  };

  return <button ref={buttonRef} onClick={onClick}>Click me!</button>;
};
```

Tell react which element the ref should bind to

useRef

<code>

Adding a ref on a react element allows us access to the underlying HTMLElement

```
import { useRef } from "react";

export const Button = () => {
  const buttonRef = useRef<HTMLButtonElement>(null);

  const onClick = () => {
    console.log(buttonRef.current!.innerHTML);
  };

  return <button ref={buttonRef} onClick={onClick}>Click me!</button>;
};
```

The HTMLElement is accessible at `.current`

Uncontrolled component

<code>

In an uncontrolled component, the state of the input field is **managed by the browser itself** (and kept in the DOM) – we have limited control.

```
export const UncontrolledForm = () => {
  const inputRef = useRef<HTMLInputElement>(null);
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        console.log(inputRef.current!.value);
      }}
    >
      <input type="text" placeholder="Title" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
};
```

Uncontrolled component

<code>

In an uncontrolled component, the state of the input field is **managed by the browser itself** (and kept in the DOM) – we have limited control.

```
export const UncontrolledForm = () => {
  const inputRef = useRef<HTMLInputElement>(null);
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        console.log(inputRef.current!.value);
      }}
    >
      <input type="text" placeholder="Title" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
};
```

We need a reference to the input element.

Uncontrolled component

<code>

In an uncontrolled component, the state of the input field is **managed by the browser itself** (and kept in the DOM) – we have limited control.

```
export const UncontrolledForm = () => {
  const inputRef = useRef<HTMLInputElement>(null);
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        console.log(inputRef.current!.value);
      }}
    >
      <input type="text" placeholder="Title" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
};
```

When submitting the form, we have to read the value from the DOM

Submit Event

<code>

Overwrite the default event on submit. Otherwise a request is triggered.

```
<form onSubmit={onSubmit}>
<!-- ... -->
<input type="submit" value="Submit"/>
</form>

onSubmit (event: FormEvent) {
    // do something with the input state
    event.preventDefault();
}
```

Task

Add a BookEditScreen with a simple uncontrolled form

Summary uncontrolled components

- Limited control:
  - data is managed by the browser, we can’t interfere e.g. when user inputs invalid data
  - We can provide a default value setting the `defaultValue`-attribute, a change will not cause any update of the DOM
- Working with `ref`s can lead to “dirty” patterns – avoid it if possible.
- `<input type="file" />` are **always uncontrolled**!

Controlled Components

Controlled component

<code>

In a controlled component, the state of the input field is managed by React's state – we have full control.

```
export const ControlledForm = () => {
  const [title, setTitle] = useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        console.log(title);
      }}
    >
      <input
        type="text"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
};
```

Create a new state, that will be in sync with the input

Controlled component

<code>

In a controlled component, the state of the input field is managed by React's state – we have full control.

```
export const ControlledForm = () => {
  const [title, setTitle] = useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        console.log(title);
      }}
    >
      <input
        type="text"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
};
```

Actively set the value of the input field to match the react state

Controlled component

<code>

In a controlled component, the state of the input field is managed by React's state – we have full control.

```
export const ControlledForm = () => {
  const [title, setTitle] = useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        console.log(title);
      }}
    >
      <input
        type="text"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
};
```

Add an **onChange** event handler that updates the react state to match the DOM state

Controlled component

<code>

In a controlled component, the state of the input field is managed by React's state – we have full control.

```
export const ControlledForm = () => {
  const [title, setTitle] = useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        console.log(title);
      }}
    >
      <input
        type="text"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
};
```

Otherwise use the state just as any other react state

Typing the onChange Event Handler

<code>

Fun fact: onChange actually runs on the HTML `input` event

```
import { ChangeEvent } from "react";

<input
  type="text"
  value={title}
  onChange={(event: ChangeEvent<HTMLInputElement>) =>
    setTitle(event.target.value)
  }
/>;
```

Controlled components

- More code, but more control:
  - data is managed by us, we can interfere when user inputs invalid data
  - we can pass the value to other UI elements or reset it from other event handlers
- A **default value** is required, otherwise the component is uncontrolled at first!
- We *can* also keep the input state higher up in the component tree – this can have performance drawbacks due to more components updating, keeping it local is preferred.

Task

Refactor the edit form into a controlled form and prefill it with data from the api

Keep in mind...

- When dealing with user input, React will not prevent users from entering invalid or even malicious data – you have to take appropriate measures!

Form Validation

Validation shows the users what they are doing wrong and how to fix it as soon as possible!

Why / What you'll learn

- Sometimes a user needs to be guided through a form
- Provide the user of your form a better UX
- Write and use functions for errors and warnings
- Create a reusable input field that renders errors and warnings

Validation - Example

Username John Smith

Email john.smith@@workshops.de

Invalid email address

Age 16

Sorry, you must be at least 18 years old

Submit Clear Values

Form Validation - Strategies

There are multiple ways to achieve Form Validation (Client-side):

- **Built-in form validation**
  - Uses HTML5 form validation features.
- **JavaScript - The constraint validation API**
  - e.g. `required` or `pattern` attributes
  - More and more browsers now support the constraint validation API, and it's becoming reliable.
- **JavaScript - Custom Implementation**
  - Sometimes the constraint validation API is not enough.

Keep in mind...

Client-side validation gives users fast feedback if their input is valid or not, it is not meant to protect your database from malicious input.

In a production application, always **combine client-side validation with server-side validation**.

Built-in form validation

HTML5 Built-in Validators

In HTML5 there are **built-in validators** that can be used with the Built-in form validation and constraint **validation API**

HTML5 Built-in Validators

- **`type`**
  - The type attribute of an input is also a validator.
  - E.g. `email`, `number`, `color`, `date`, `datetime-local`, `month`, `number`, `range`, `password`
- **`required`**
  - A value is required
- **`minlength`, `maxlength`**
  - The minimal or maximal length of the input value
- **`pattern`**
  - The input value has to match the given regular expression

Built-in form validation

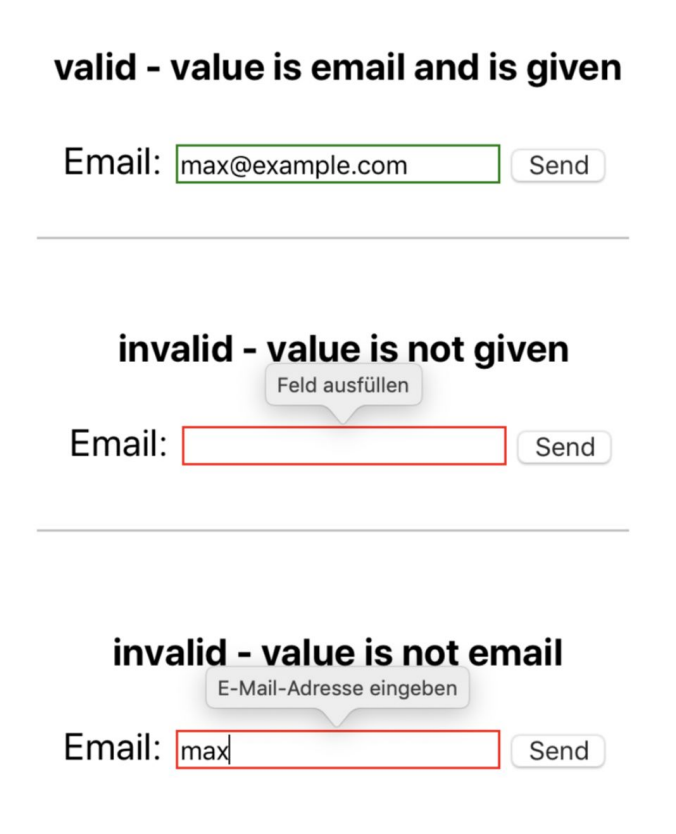
<code>

Example usage of build-in form validations

```
return (
    <form onSubmit={onSubmit}>
      <label htmlFor="userEmail">Email: </label>
      <input
        id="userEmail"
        name="userEmail"
        type="email"
        required
        value={email}
        onChange={onChange}
      />
      <button>Send</button>
    </form>
  );
```

Using CSS-Pseudo classes with validation

```
input {
    outline: none;
}
input:valid {
    border: 1px solid green;
}
input:invalid {
    border: 1px solid red;
}
```



Disadvantages of Built-in Form Validation

- **No immediate user guidance**
  - Error messages are shown on form submit.
  - There is no way to immediately show an input hint, error or success message depending on the inputs validity and / or touched state - while the user is typing.
- **The error messages are pre-styled and pre-defined**
  - We’d like to show a custom message with our custom design and behavior
- **No cross-field validation**
  - Validation happens on input element basis.

Validate with our own strategy

Validation Form - noValidate

<code>

Disable built-in browser-specific HTML5 validation for a form

```
<!-- "noValidate" with a capital 'V'. -->
<form onSubmit={handleSubmit} noValidate></form>
```

Validate with our own strategy

<code>

Simplest setup: compute validation on each render with a simple variable

```
export const SimpleForm = () => {
  const [email, setEmail] = useState<string>("");
  const emailError = email.includes("@") ? "" : "Email must contain @";

  return (
    <form onSubmit={onSubmit} noValidate>
      <input
        value={email}
        onChange={(event) => setEmail(event.target.value)}
      />
      {emailError && <p>{emailError}</p>}
      <button type="submit">Send</button>
    </form>
  );
};
```

emailError will be empty if valid, otherwise it will be an error message

Validate with our own strategy

<code>

Simplest setup: compute validation on each render with a simple variable

```
export const SimpleForm = () => {
  const [email, setEmail] = useState<string>("");
  const emailError = email.includes("@") ? "" : "Email must contain @";

  return (
    <form onSubmit={onSubmit} noValidate>
      <input
        value={email}
        onChange={(event) => setEmail(event.target.value)}
      />
      {emailError && <p>{emailError}</p>}
      <button type="submit">Send</button>
    </form>
  );
};
```

Display the error message if there is one

Validate with our own strategy

<code>

Alternative: manage error as separate state

```
export const SimpleForm = () => {
  const [email, setEmail] = useState('');
  const [emailError, setEmailError] = useState('');

  const handleEmailChange = ({event: ChangeEvent<HTMLInputElement>) => {
    setEmail(value);
    setEmailError(value.includes('@') ? '' : 'Email must contain @');
  };

  return (
    <form onSubmit={onSubmit} noValidate>
      <input
        value={email}
        onChange={handleEmailChange}
      />
      {emailError && <p>{emailError}</p>}
      <button>Send</button>
    </form>
  );
}
```

emailError is now a state

Validate with our own strategy

<code>

Alternative: manage error as separate state

```
export const SimpleForm = () => {
  const [email, setEmail] = useState('');
  const [emailError, setEmailError] = useState('');

  const handleEmailChange = ({event: ChangeEvent<HTMLInputElement>) => {
    setEmail(value);
    setEmailError(value.includes('@') ? '' : 'Email must contain @');
  };

  return (
    <form onSubmit={onSubmit} noValidate>
      <input
        value={email}
        onChange={handleEmailChange}
      />
      {emailError && <p>{emailError}</p>}
      <button>Send</button>
    </form>
  );
}
```

We revalidate onChange

Task

Add form validation

Forms can be tricky

Libraries can…

- …support setting up and working with forms:
  - Formik ("plain" React, takes care of the complicated pieces)
- …writing validation logic:
  - Yup (validates plain JavaScript objects based on a schema)

And now putting it all together:

Task

Save the changes to the edited books

We teach.

workshops.de