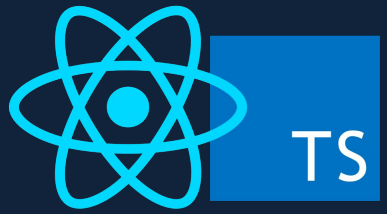


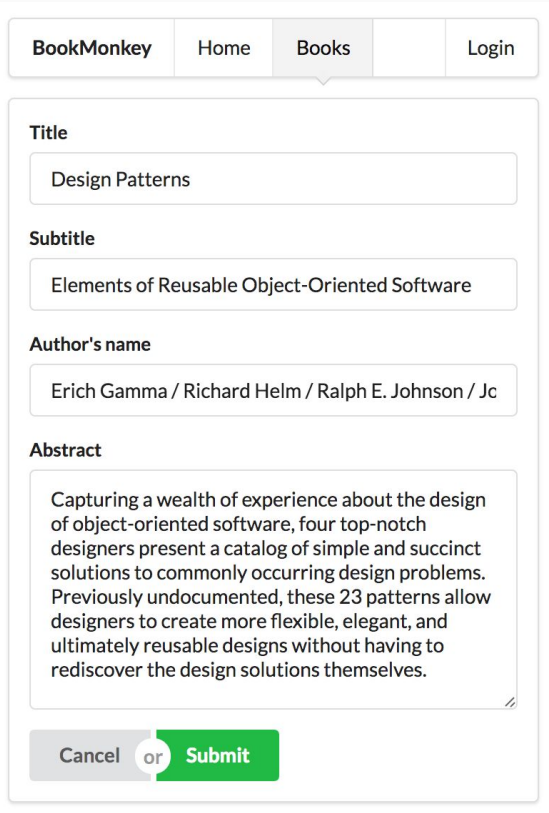
Workshop

React Forms

Task

What do you know about forms?

Handle user input in your application



Why / What you'll learn

- Manage user input to create and edit data
- Create forms with React
- Different types of input validation

Handling user input

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

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**)

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: React.FC<{ onSubmit: (data: string) => void }> = ({
  onSubmit
}) => {
  const inputRef = React.useRef<HtmlInputElement | null>(null);
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(inputRef.current.value);
        inputRef.current.value = "";
      }}
    >
      <input type="text" placeholder="Title" ref={inputRef} />
      <button>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.

```tsx
export const UncontrolledForm: React.FC<{ onSubmit: (data: string) => void }> = ({
  onSubmit
}) => {
  const inputRef = React.useRef<HtmlInputElement | null>(null);
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(inputRef.current.value);
        inputRef.current.value = "";
      }}
    >
      <input type="text" placeholder="Title" ref={inputRef} />
      <button>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.

```tsx
export const UncontrolledForm: React.FC<{ onSubmit: (data: string) => void }> = ({
  onSubmit
}) => {
  const inputRef = React.useRef(null);
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(inputRef.current.value);
        inputRef.current.value = "";
      }}
    >
      <input type="text" placeholder="Title" ref={inputRef} />
      <button>Submit</button>
    </form>
  );
};
```

When submitting the form, we have to read the value from the DOM and reset the value in the DOM!

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 component

<code>

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

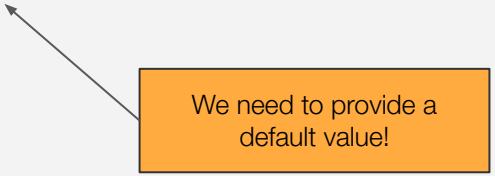
```
export const ControlledForm: React.FC<{ onSubmit: (data: string) => void }> = ({ onSubmit }) => {
  const [title, setTitle] = React.useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(title);
        setTitle("");
      }}
    >
      <input
        type="text"
        placeholder="Title"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button>Submit</button>
    </form>
  );
};
```

Controlled component

<code>

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

```tsx
export const ControlledForm: React.FC<{ onSubmit: (data: string) => void }> = ({ onSubmit }) => {
  const [title, setTitle] = React.useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(title);
        setTitle("");
      }}
    >
      <input
        type="text"
        placeholder="Title"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button>Submit</button>
    </form>
  );
};
```

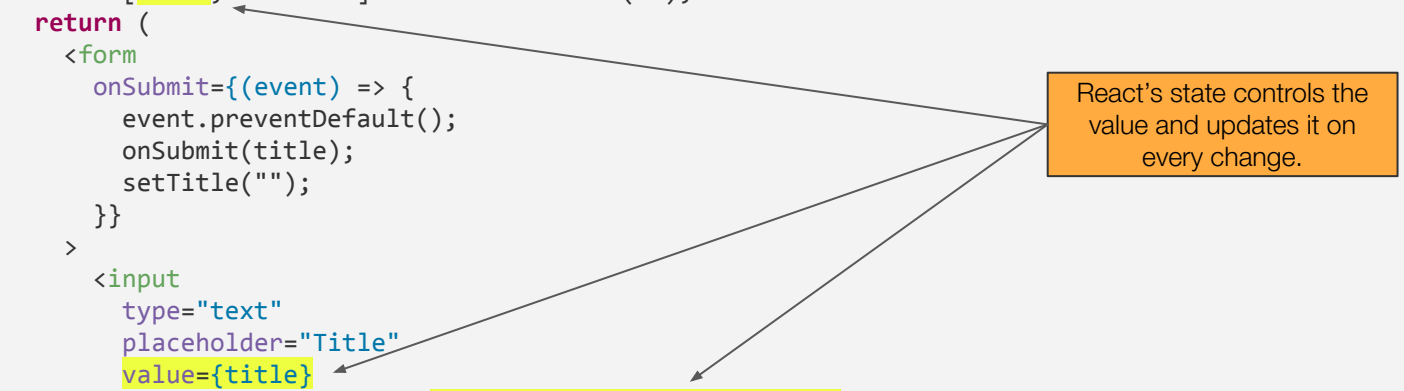


Controlled component

<code>

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

```tsx
export const ControlledForm: React.FC<{ onSubmit: (data: string) => void }> = ({ onSubmit }) => {
  const [title, setTitle] = React.useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(title);
        setTitle("");
      }}
    >
      <input
        type="text"
        placeholder="Title"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button>Submit</button>
    </form>
  );
};
```



Controlled component

<code>

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

```tsx
export const ControlledForm: React.FC<{ onSubmit: (data: string) => void }> = ({ onSubmit }) => {
  const [title, setTitle] = React.useState("");
  return (
    <form
      onSubmit={(event) => {
        event.preventDefault();
        onSubmit(title);
        setTitle("");
      }}
    >
      <input
        type="text"
        placeholder="Title"
        value={title}
        onChange={(event) => setTitle(event.target.value)}
      />
      <button>Submit</button>
    </form>
  );
};
```

When submitting the form, we read and update our local state.

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 an uncontrolled component to a controlled component

Handle Submit in Forms

<code>

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

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

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

Keep in mind...

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

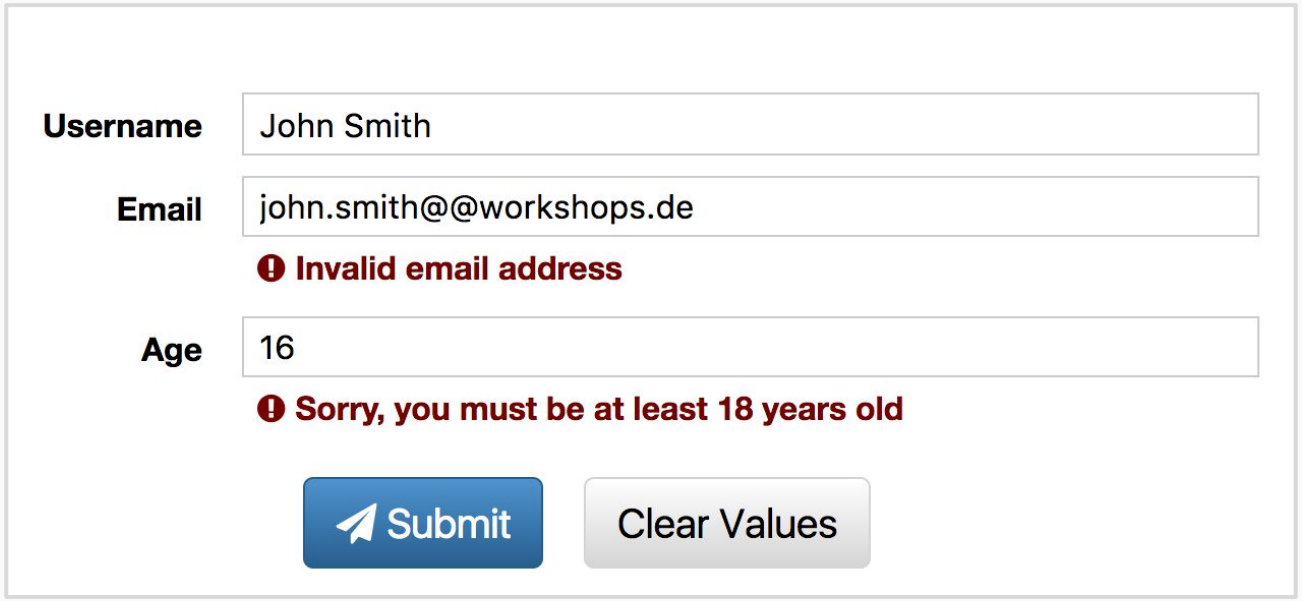
Form Validation

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

Why / What you'll learn

- Sometime 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



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

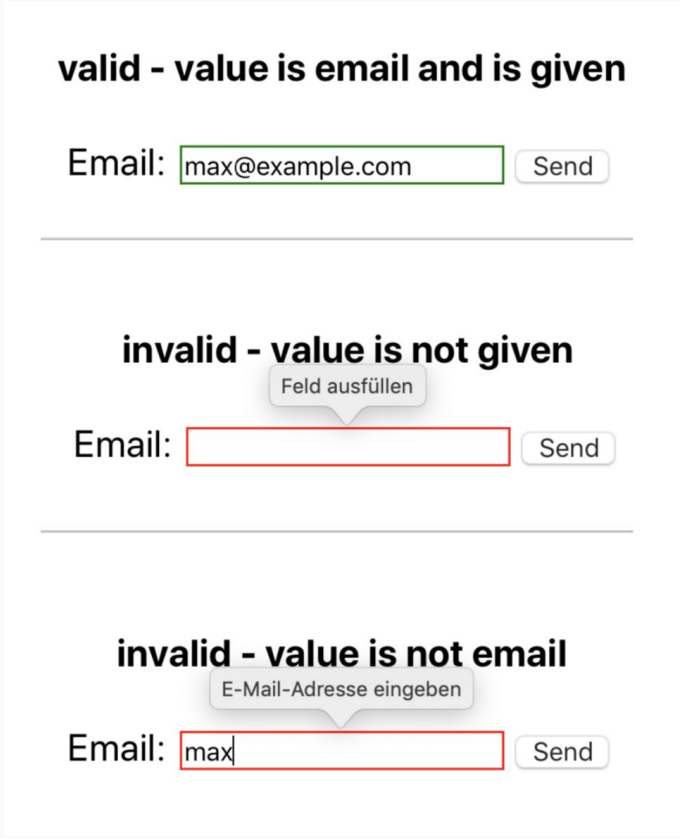
<code>

Example usage of build-in form validations

```
export function SimpleForm() {
    const [email, setEmail] = useState<string>('');

    function handleChange({ target: { value } }: React.ChangeEvent<HTMLInputElement>) {
        setEmail(value);
    }
    return (
        <form onSubmit={sendForm}>
            <label htmlFor="userEmail">Email: </label>
            <input id="userEmail"
                name="userEmail"
                type="email"
                required
                value={email}
                onChange={handleChange} />
            <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;
}
```



Task

Create a form with built-in validation

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

Validate with our own strategy

Let us define an own validation strategy:

- Each input is in the state: **(un-)touched** and **(in-)valid**
- The error message depends on the kind of the error.
- Disable each submit trigger (submit button) if the form is in an invalid state.

Validation Form - noValidate

<code>

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

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

Validate with our own strategy

<code>

Disable built-in validation

```
export function SimpleForm() {
    return (
      <form onSubmit={sendForm} noValidate>
            {emailError && (<p className="errorMessage">{emailError}</p>)}
            {!emailError && email && (<p className="successMessage">Thank you for your email.</p>)}
            {!emailError && !email && (<p className="hintMessage">Please input an email.</p>)}
            <label htmlFor="userEmail">Email: </label>
            <input id="userEmail"
                name="userEmail"
                type="email"
                required
                value={email}
                onChange={e => handleChange(e)} />
            <button type="submit" disabled="disabled">Send</button>
        </form>
   )
}
```

Validate on with our own strategy

<code>

A reference can be bound directly to a DOM Node.

```
import React, { useState, useRef } from 'react';
import './SimpleForm.css';


export function SimpleForm() {
    const [email, setEmail] = useState('');
    const [emailError, setEmailError] = useState('');
    const submitButtonRef = useRef(undefined);
    const formRef = useRef(undefined);

    // ..
```

Validate on with our own strategy

<code>

Bind the reference to the DOM Node

```
export function SimpleForm() {
    return (
      <form ref={formRef} onSubmit={sendForm} novalidate>
            {emailError && (<p className="errorMessage">{emailError}</p>)}
            {!emailError && email && (<p className="successMessage">Thank you for your email.</p>)}
            {!emailError && !email && (<p className="hintMessage">Please input an email.</p>)}
            <label htmlFor="userEmail">Email: </label>
            <input id="userEmail"
                name="userEmail"
                type="email"
                required
                value={email}
                onChange={e => handleChange(e)} />
            <button ref={submitButtonRef} disabled="disabled">Send</button>
        </form>
   )
}
```

Validate with our own strategy

<code>

Define custom error messages for our form

```
export function SimpleForm() {
    return (
      <form onSubmit={sendForm} novalidate>
            {emailError && (<p className="errorMessage">{emailError}</p>)}
            {!emailError && email && (<p className="successMessage">Thank you for your email.</p>)}
            {!emailError && !email && (<p className="hintMessage">Please input an email.</p>)}
            <label htmlFor="userEmail">Email: </label>
            <input id="userEmail"
                name="userEmail"
                type="email"
                required
                value={email}
                onChange={e => handleChange(e)} />
            <button type="submit" disabled="disabled">Send</button>
        </form>
   )
}
```

Validate with our own strategy

<code>

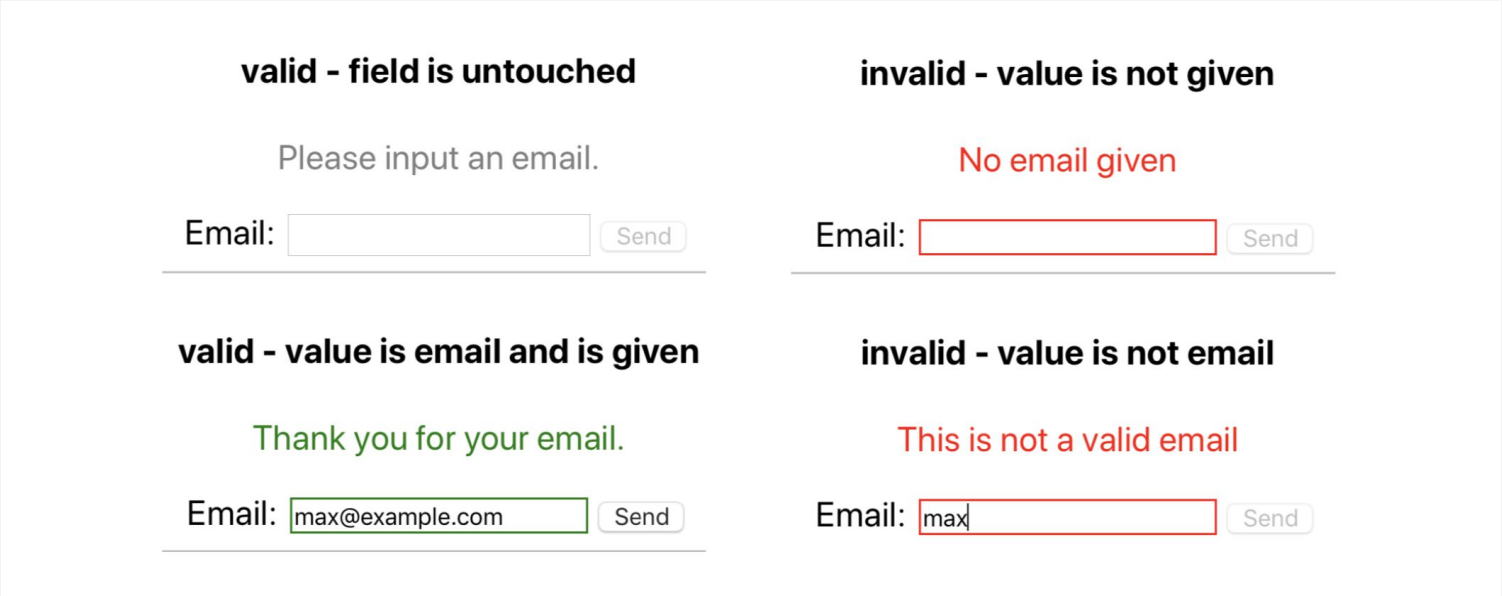
Define our handleChange method

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

    const handleEmailChange = ({ target: { value } }: React.ChangeEvent<HTMLInputElement>) => {
      setEmail(value);
       const error = validateEmail(value);
       if (error) {
         setEmailError(error);
       } else {
        setEmailError(null);
       }
    };

    return (
       ...
   )
}
```

Validate with our own strategy

<code>

See our own strategy in action in different states



Delayed Validation

<code>

Validating on every keystroke is generally not great UX

```
export function SimpleForm() {
    const handleEmailValidation = ({ target: { value } }: React.ChangeEvent<HTMLInputElement>) => {
      const error = validateEmail(value);
      if (error) {
        setEmailError(error);
      } else {
        setEmailError(null);
      }
    };

    return (
      ...
      <input id="userEmail"
        ...
        onBlur={e => handleEmailValidation(e)}
        onChange={e => handleEmailChange(e)} />
   )
}
```

Validation Function

Validation Function

- Get an object of all inputs as values
- Return an object of all errors

```
values = { age: 16 }
```

```
errors = { age: 'Sorry, you must
be at least 18 years old' }
```

Validation Function

<code>

Simple function that accepts form values and returns an error object

```
const validate = values => {
  const errors = {}
  if (Number(values.age) < 18) {
    errors.age = 'Sorry, you must be at least 18 years old'
  }
  return errors;
}
```

Validation Function - Example Required

<code>

If an input is empty it is undefined

```
const validate = values => {
  const errors = {}
  if (!values.age) {
    errors.age = 'Required'
  }
  return errors
}
```

Task

Create a form with our own 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)

We teach.

workshops.de

We teach.

workshops.de