ДИСЦИПЛИНА	Фронтенд и бэкенд разработка
ИНСТИТУТ	ИПТИП
КАФЕДРА	Индустриального программирования
ВИД УЧЕБНОГО МАТЕРИАЛА	Методические указания к практическим занятиям
ПРЕПОДАВАТЕЛЬ	Астафьев Рустам Уралович
СЕМЕСТР	1 семестр, 2025/2026 уч. год

Ссылка на материал:
https://github.com/astafiev-rustam/frontend-and-backend-development/tree/practice-1-25

Практическое занятие 25: Формы React: валидация, сообщения об ошибках и элементы доступности

В рамках данного занятия будут рассмотрены возможности работы с формами в React и обеспечением доступности. Материалы занятия соответствуют представлениям о формировании ввода и доступности страниц из предыдущих практик и лекций.

Теоретическая часть

Пример 1. Форма с валидацией в реальном времени

Проблема: Нужно создать форму добавления технологии с валидацией полей в реальном времени и понятными сообщениями об ошибках.

Подход к решению: Используем состояние для хранения ошибок, проверяем валидность при каждом изменении и блокируем отправку при ошибках.

Пошаговая реализация:

Шаг 1: Настройка состояний формы и параметра initialData

Компонент `TechnologyForm` принимает props от родительского компонента:

```
function TechnologyForm({ onSave, onCancel, initialData = {} }) {
  // ...
}
```

Пропс initialData:

`initialData` - это объект с данными технологии, который передается родительским компонентом:

- Когда **создаем новую** технологию, родитель вызывает компонент БЕЗ initialData:

```
<TechnologyForm onSave={handleSave} onCancel={handleCancel} />
// initialData будет пустым объектом {} (значение по умолчанию)
```

- Когда **редактируем существующую** технологию, родитель передает данные:

```
<TechnologyForm
  onSave={handleSave}
  onCancel={handleCancel}
  initialData={{
    title: 'React Hooks',
    description: 'Изучение хуков useState и useEffect',
    category: 'frontend',
    difficulty: 'intermediate',
    deadline: '2024-12-31',
    resources: ['https://react.dev', 'https://example.com/tutorial']
  }}
/>
```

Что такое "ресурсы" (resources):

Ресурсы - это массив URL-адресов полезных материалов для изучения технологии. Например:

- Официальная документация: `['https://react.dev']`
- Несколько источников: `['https://nodejs.org/docs', 'https://example.com/tutorial']`

Пользователь может динамически добавлять/удалять поля для ресурсов. Минимум должно быть одно поле.

Создаем три основных состояния:

```
// состояние формы с начальными значениями
const [formData, setFormData] = useState({
  title: initialData.title || '',              // название технологии
  description: initialData.description || '',   // описание
  category: initialData.category || 'frontend', // категория
  difficulty: initialData.difficulty || 'beginner', // сложность
  deadline: initialData.deadline || '',         // дедлайн (необязательно)
  resources: initialData.resources || ['']      // массив URL ресурсов
});

// состояние для хранения ошибок валидации
const [errors, setErrors] = useState({});

// флаг валидности всей формы
const [isFormValid, setIsFormValid] = useState(false);
```

Назначение:

- `formData` хранит все значения полей. Используем `initialData.field || defaultValue` для инициализации
- `errors` - объект с сообщениями об ошибках: `{ title: 'Название обязательно' }`
- `isFormValid` - можно ли отправить форму (true/false)

Почему `resources: initialData.resources || ['']`:

- При редактировании: берем существующий массив ресурсов
- При создании: инициализируем массив с одним пустым полем `['']`
- Гарантирует наличие минимум одного поля для ввода URL

Шаг 2: Функция валидации с обнулением времени

```
const validateForm = () => {
  const newErrors = {};

  // валидация названия технологии
  if (!formData.title.trim()) {
    newErrors.title = 'Название технологии обязательно';
  }

  // валидация дедлайна (не должен быть в прошлом)
  if (formData.deadline) {
    const deadlineDate = new Date(formData.deadline);
    const today = new Date();
    today.setHours(0, 0, 0, 0);

    if (deadlineDate < today) {
      newErrors.deadline = 'Дедлайн не может быть в прошлом';
    }
  }

  setErrors(newErrors);
  setIsFormValid(Object.keys(newErrors).length === 0);
};
```

today.setHours(0, 0, 0, 0):

Эта строка обнуляет время у сегодняшней даты, оставляя только дату.

```
today.setHours(0, 0, 0, 0);
// setHours(часы, минуты, секунды, миллисекунды)
```

Зачем это нужно:

БЕЗ обнуления времени:

```
// сейчас: 24 ноября 2024, 16:45:30
const today = new Date(); // 2024-11-24 16:45:30
const deadline = new Date('2024-11-24'); // 2024-11-24 00:00:00

if (deadline < today) {
  // ошибка! deadline (00:00) < today (16:45)
  // пользователь выбрал сегодняшнюю дату, но получит ошибку валидации
}
```

С обнулением времени:

```
const today = new Date();
today.setHours(0, 0, 0, 0); // 2024-11-24 00:00:00
const deadline = new Date('2024-11-24'); // 2024-11-24 00:00:00

if (deadline < today) {
  // false - даты равны, ошибки не будет
  // пользователь может выбрать сегодняшнюю дату как дедлайн
}
```

Итог: обнуление времени позволяет сравнивать только даты, игнорируя текущее время дня.

Шаг 3: Запуск валидации при изменении данных

```
// запуск валидации при каждом изменении formData
useEffect(() => {
  validateForm();
}, [formData]);
```

Назначение: каждый раз когда пользователь меняет любое поле, автоматически запускается валидация. Это обеспечивает валидацию в реальном времени без задержек.

Шаг 4: Обработчик изменения конкретного ресурса

```
// обработчик изменения конкретного ресурса в массиве
const handleResourceChange = (index, value) => {
  const newResources = [...formData.resources];
  newResources[index] = value;
  setFormData(prev => ({
    ...prev,
    resources: newResources
  }));
};
```

Подробное объяснение работы:

Допустим, у нас есть такое состояние:

```
formData.resources = [
  'https://react.dev',      // индекс 0
  'https://nodejs.org',     // индекс 1
  ''                        // индекс 2 (пустое поле)
]
```

Пользователь вводит текст во второе поле (индекс 1). Вызывается `handleResourceChange(1, 'https://nodejs.org/docs')`:

Шаг 1: Создаем копию массива

```
const newResources = [...formData.resources];
// [...] - spread оператор создает НОВУЮ копию массива
// это важно для иммутабельности - не мутируем исходный массив
```

Шаг 2: Обновляем нужный элемент по индексу

```
newResources[1] = 'https://nodejs.org/docs';
// теперь массив: ['https://react.dev', 'https://nodejs.org/docs', '']
```

Шаг 3: Обновляем состояние

```
setFormData(prev => ({
  ...prev,              // сохраняем все остальные поля (title, description и
т.д.)
  resources: newResources  // заменяем только массив resources
}));
```

Итог: React увидит изменение состояния и перерендерит компонент. Второе поле покажет новое значение.

Шаг 5: Добавление и удаление полей ресурсов

```
// добавление нового пустого поля для ресурса
const addResourceField = () => {
  setFormData(prev => ({
    ...prev,
    resources: [...prev.resources, '']
  }));
```

```
};

// удаление поля ресурса по индексу
const removeResourceField = (index) => {
  if (formData.resources.length > 1) {
    const newResources = formData.resources.filter((_, i) => i !== index);
    setFormData(prev => ({
      ...prev,
      resources: newResources
    }));
  }
};
```

Подробное объяснение addResourceField:

Начальное состояние:

```
formData.resources = ['https://react.dev', 'https://nodejs.org']
```

Пользователь нажимает "+ Добавить ресурс". Выполняется:

```
resources: [...prev.resources, '']
// ...prev.resources - "разворачивает" массив: 'https://react.dev',
'https://nodejs.org'
// '' - добавляем пустую строку в конец
// результат: ['https://react.dev', 'https://nodejs.org', '']
```

removeResourceField и filter((_, i) => i !== index):

В JavaScript _ - это обычное имя переменной. Используется как **соглашение** для обозначения неиспользуемого параметра.

Метод `filter` вызывает функцию для каждого элемента с двумя параметрами:

1. Сам элемент (значение)
2. Индекс элемента

```
// полная запись:
formData.resources.filter((element, index) => index !== 1)

// краткая запись (когда элемент не нужен):
formData.resources.filter((_, index) => index !== 1)
```

Использование _ явно показывает: "первый параметр не используется, но должен быть указан для доступа ко второму".

Пример удаления:

```
// массив: ['https://react.dev', 'https://nodejs.org', 'https://example.com']
// удаляем индекс 1 (второй элемент)

filter((_, i) => i !== 1)

// вызов 1: i=0 → 0 !== 1 → true → оставляем элемент
// вызов 2: i=1 → 1 !== 1 → false → УДАЛЯЕМ элемент
// вызов 3: i=2 → 2 !== 1 → true → оставляем элемент

// результат: ['https://react.dev', 'https://example.com']
```

Проверка `if (formData.resources.length > 1)`: гарантирует что минимум одно поле всегда останется.

Шаг 9: Обработка отправки формы с очисткой пустых ресурсов

```
// обработчик отправки формы
const handleSubmit = (e) => {
  e.preventDefault();

  if (isFormValid) {
    // очищаем пустые ресурсы перед сохранением
    const cleanedData = {
      ...formData,
      resources: formData.resources.filter(resource => resource.trim() !== '')
    };

    onSave(cleanedData);
  }
};
```

Подробное объяснение resources.filter(resource => resource.trim() !== ''):

Эта строка удаляет все пустые или заполненные только пробелами поля ресурсов перед отправкой данных.

Ситуация: Пользователь добавил три поля для ресурсов, но заполнил только два:

```
formData.resources = [
  'https://react.dev',      // заполнено
  'https://nodejs.org',     // заполнено
  ''                        // пустое поле
]
```

Как работает filter:

```
resources.filter(resource => resource.trim() !== '')

// вызов 1: resource='https://react.dev'
// 'https://react.dev'.trim() → 'https://react.dev'
// 'https://react.dev' !== '' → true → ОСТАВЛЯЕМ

// вызов 2: resource='https://nodejs.org'
// 'https://nodejs.org'.trim() → 'https://nodejs.org'
// 'https://nodejs.org' !== '' → true → ОСТАВЛЯЕМ

// вызов 3: resource=''
// ''.trim() → ''
// '' !== '' → false → УДАЛЯЕМ

// результат: ['https://react.dev', 'https://nodejs.org']
```

Зачем нужен trim():

Метод `trim()` удаляет пробелы с начала и конца строки:

```
'  '.trim() === ''           // true - только пробелы стали пустой строкой
'  hello  '.trim() === 'hello' // true - пробелы убраны с краев
'hello'.trim() === 'hello'     // true - строка без изменений
```

Это защищает от "псевдозаполненных" полей, где пользователь случайно ввел только пробелы.

Итоговые данные для отправки:

```
cleanedData = {
  title: 'React Hooks',
  description: 'Изучение хуков',
  category: 'frontend',
  difficulty: 'intermediate',
  deadline: '2024-12-31',
  resources: ['https://react.dev', 'https://nodejs.org'] // только заполненные URL
}
```

Зачем это нужно: на сервер или в родительский компонент попадают только реально заполненные ресурсы, без пустых строк.

Полный исходный код TechnologyForm.jsx

```jsx
import { useState, useEffect } from 'react';
import './TechnologyForm.css';

function TechnologyForm({ onSave, onCancel, initialData = {} }) {
  // состояние формы с начальными значениями или данными для редактирования
  const [formData, setFormData] = useState({
    title: initialData.title || '',
    description: initialData.description || '',
    category: initialData.category || 'frontend',
    difficulty: initialData.difficulty || 'beginner',
    deadline: initialData.deadline || '',
    resources: initialData.resources || ['']
  });

  // состояние для хранения ошибок валидации
  const [errors, setErrors] = useState({});

  // флаг валидности всей формы
  const [isFormValid, setIsFormValid] = useState(false);

  // функция валидации всей формы
  const validateForm = () => {
    const newErrors = {};

    // валидация названия технологии
    if (!formData.title.trim()) {
      newErrors.title = 'Название технологии обязательно';
    } else if (formData.title.trim().length < 2) {
      newErrors.title = 'Название должно содержать минимум 2 символа';
    } else if (formData.title.trim().length > 50) {
      newErrors.title = 'Название не должно превышать 50 символов';
    }

    // валидация описания
    if (!formData.description.trim()) {
      newErrors.description = 'Описание технологии обязательно';
    } else if (formData.description.trim().length < 10) {
      newErrors.description = 'Описание должно содержать минимум 10 символов';
    }

    // валидация дедлайна (не должен быть в прошлом)
    if (formData.deadline) {
      const deadlineDate = new Date(formData.deadline);
      const today = new Date();
      today.setHours(0, 0, 0, 0); // обнуляем время для сравнения только дат

      if (deadlineDate < today) {
        newErrors.deadline = 'Дедлайн не может быть в прошлом';
      }
    }

    // валидация URL-адресов ресурсов
    formData.resources.forEach((resource, index) => {
```

```
      if (resource && !isValidUrl(resource)) {
        newErrors[`resource_${index}`] = 'Введите корректный URL';
      }
    });

    setErrors(newErrors);
    setIsFormValid(Object.keys(newErrors).length === 0);
  };

  // проверка корректности URL
  const isValidUrl = (string) => {
    try {
      new URL(string);
      return true;
    } catch (_) {
      return false;
    }
  };

  // запуск валидации при каждом изменении formData
  useEffect(() => {
    validateForm();
  }, [formData]);

  // обработчик изменения обычных полей (input, select, textarea)
  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData(prev => ({
      ...prev,
      [name]: value
    }));
  };

  // обработчик изменения конкретного ресурса в массиве
  const handleResourceChange = (index, value) => {
    const newResources = [...formData.resources];
    newResources[index] = value;
    setFormData(prev => ({
      ...prev,
      resources: newResources
    }));
  };

  // добавление нового пустого поля для ресурса
  const addResourceField = () => {
    setFormData(prev => ({
      ...prev,
      resources: [...prev.resources, '']
    }));
  };

  // удаление поля ресурса по индексу (минимум одно поле должно остаться)
  const removeResourceField = (index) => {
    if (formData.resources.length > 1) {
```

```
        const newResources = formData.resources.filter((_, i) => i !== index);
        setFormData(prev => ({
          ...prev,
          resources: newResources
        }));
      }
    };

    // обработчик отправки формы
    const handleSubmit = (e) => {
      e.preventDefault();

      if (isFormValid) {
        // очищаем пустые ресурсы перед сохранением
        const cleanedData = {
          ...formData,
          resources: formData.resources.filter(resource => resource.trim() !== '')
        };

        onSave(cleanedData);
      }
    };

    return (
      <form onSubmit={handleSubmit} className="technology-form" noValidate>
        <h2>{initialData.title ? 'Редактирование технологии' : 'Добавление новой
технологии'}</h2>

        {/* поле названия */}
        <div className="form-group">
          <label htmlFor="title" className="required">
            Название технологии
          </label>
          <input
            id="title"
            name="title"
            type="text"
            value={formData.title}
            onChange={handleChange}
            className={errors.title ? 'error' : ''}
            placeholder="Например: React, Node.js, TypeScript"
            aria-describedby={errors.title ? 'title-error' : undefined}
            required
          />
          {errors.title && (
            <span id="title-error" className="error-message" role="alert">
              {errors.title}
            </span>
          )}
        </div>

        {/* поле описания */}
        <div className="form-group">
          <label htmlFor="description" className="required">
```

```
          Описание
        </label>
        <textarea
          id="description"
          name="description"
          value={formData.description}
          onChange={handleChange}
          rows="4"
          className={errors.description ? 'error' : ''}
          placeholder="Опишите, что это за технология и зачем её изучать..."
          aria-describedby={errors.description ? 'description-error' : undefined}
          required
        />
        {errors.description && (
          <span id="description-error" className="error-message" role="alert">
            {errors.description}
          </span>
        )}
      </div>

      {/* выбор категории */}
      <div className="form-group">
        <label htmlFor="category">Категория</label>
        <select
          id="category"
          name="category"
          value={formData.category}
          onChange={handleChange}
        >
          <option value="frontend">Frontend</option>
          <option value="backend">Backend</option>
          <option value="database">База данных</option>
          <option value="devops">DevOps</option>
          <option value="other">Другое</option>
        </select>
      </div>

      {/* выбор сложности */}
      <div className="form-group">
        <label htmlFor="difficulty">Сложность</label>
        <select
          id="difficulty"
          name="difficulty"
          value={formData.difficulty}
          onChange={handleChange}
        >
          <option value="beginner">Начальный</option>
          <option value="intermediate">Средний</option>
          <option value="advanced">Продвинутый</option>
        </select>
      </div>

      {/* дедлайн */}
      <div className="form-group">
```

```
          <label htmlFor="deadline">Дедлайн (необязательно)</label>
          <input
            id="deadline"
            name="deadline"
            type="date"
            value={formData.deadline}
            onChange={handleChange}
            className={errors.deadline ? 'error' : ''}
            aria-describedby={errors.deadline ? 'deadline-error' : undefined}
          />
          {errors.deadline && (
            <span id="deadline-error" className="error-message" role="alert">
              {errors.deadline}
            </span>
          )}
        </div>

        {/* список ресурсов для изучения */}
        <div className="form-group">
          <label>Ресурсы для изучения</label>
          {formData.resources.map((resource, index) => (
            <div key={index} className="resource-field">
              <input
                type="url"
                value={resource}
                onChange={(e) => handleResourceChange(index, e.target.value)}
                placeholder="https://example.com"
                className={errors[`resource_${index}`] ? 'error' : ''}
                aria-describedby={errors[`resource_${index}`] ? `resource-${index}-
error` : undefined}
              />
              {formData.resources.length > 1 && (
                <button
                  type="button"
                  onClick={() => removeResourceField(index)}
                  className="btn-remove"
                  aria-label={`Удалить ресурс ${index + 1}`}
                >
                  Удалить
                </button>
              )}
              {errors[`resource_${index}`] && (
                <span id={`resource-${index}-error`} className="error-message"
role="alert">
                  {errors[`resource_${index}`]}
                </span>
              )}
            </div>
          ))}
          <button
            type="button"
            onClick={addResourceField}
            className="btn-add-resource"
          >
```

```
            + Добавить ресурс
          </button>
        </div>

        {/* кнопки действий */}
        <div className="form-actions">
          <button
            type="submit"
            className="btn-primary"
            disabled={!isFormValid}
          >
            Сохранить
          </button>
          <button
            type="button"
            onClick={onCancel}
            className="btn-secondary"
          >
            Отмена
          </button>
        </div>
      </form>
    );
}

export default TechnologyForm;
```

Пример 2. Доступная форма с ARIA атрибутами

Проблема: Нужно создать форму, которая корректно работает со скринридерами и другими вспомогательными технологиями.

Подход к решению: Используем ARIA атрибуты для обеспечения доступности, правильные связи между элементами и объявление статуса операций.

Пошаговое объяснение реализации:

Шаг 1: Создание области для объявлений скринридерами

Добавляем скрытый элемент, который будет объявлять статус отправки формы пользователям скринридеров:

```
{/* область для скринридера - объявляет статус отправки */}
<div
  role="status"
  aria-live="polite"
  aria-atomic="true"
  className="sr-only"
>
  {isSubmitting && 'Отправка формы...'}
```

```
  {submitSuccess && 'Форма успешно отправлена!'}
</div>
```

Подробное объяснение aria-live="polite":

Атрибут `aria-live` указывает скринридеру отслеживать изменения содержимого элемента и объявлять их пользователю. Есть три возможных значения:

1. **"off"** (по умолчанию) - не объявлять изменения
2. **"polite"** - объявить изменения когда скринридер закончит текущее чтение
3. **"assertive"** - немедленно прервать текущее чтение и объявить изменения

Сравнение polite и assertive:

```
// polite - вежливое объявление
<div aria-live="polite">Форма успешно отправлена!</div>
// скринридер: "...и вы также можете воспользоваться поиском..."
// (заканчивает текущую фразу)
// "Форма успешно отправлена!"

// assertive - агрессивное прерывание
<div aria-live="assertive">Критическая ошибка!</div>
// скринридер: "...и вы также можете восполь..."
// (прерывается немедленно)
// "Критическая ошибка!"
```

Когда использовать:

- **"polite"** для некритичной информации: статус отправки формы, уведомления, загрузка данных
- **"assertive"** для срочной информации: критические ошибки, предупреждения безопасности, таймеры

Для статуса отправки формы используем **"polite"**, чтобы не прерывать пользователя резко.

Подробное объяснение aria-atomic="true":

Атрибут `aria-atomic` контролирует, что именно объявляется при изменении содержимого:

- **"false"** (по умолчанию) - объявить только изменившуюся часть
- **"true"** - объявить весь контент элемента целиком

Примеры работы:

```
// aria-atomic="false" - только изменения
<div aria-live="polite" aria-atomic="false">
  <span>Загружено файлов: </span>
  <span id="count">5</span>
</div>
// при изменении count с 5 на 6:
```

```
// скринридер объявит: "6" (только новое число, непонятно)

// aria-atomic="true" - весь контент
<div aria-live="polite" aria-atomic="true">
  <span>Загружено файлов: </span>
  <span id="count">5</span>
</div>
// при изменении count с 5 на 6:
// скринридер объявит: "Загружено файлов: 6" (весь контент, понятно)
```

Когда использовать:

- **"true"** когда изменение теряет смысл без контекста (счетчики, статусы)
- **"false"** когда изменяющаяся часть самодостаточна (список элементов)

Мы используем **"true"**, чтобы пользователь услышал полное сообщение "Форма успешно отправлена!", а не просто "true".

CSS класс sr-only (screen reader only):

Элемент скрыт визуально, но доступен для скринридеров:

```
.sr-only {
  position: absolute;
  width: 1px;
  height: 1px;
  padding: 0;
  margin: -1px;
  overflow: hidden;
  clip: rect(0, 0, 0, 0);
  white-space: nowrap;
  border-width: 0;
}
```

Шаг 2: Связывание полей с ошибками и двойное отрицание

```
<input
  id="contact-name"
  type="text"
  value={name}
  onChange={(e) => setName(e.target.value)}
  aria-required="true"
  aria-invalid={!!errors.name}
  aria-describedby={errors.name ? 'name-error' : undefined}
  className={errors.name ? 'error' : ''}
/>
{errors.name && (
  <span id="name-error" className="error-text" role="alert">
    {errors.name}
```

```
    </span>
  )}
```

Назначение ARIA атрибутов:

- `aria-required="true"` - сообщает что поле обязательное
- `aria-invalid={!!errors.name}` - указывает валидно ли поле (true/false)
- `aria-describedby="name-error"` - связывает поле с описанием ошибки по ID
- `role="alert"` на ошибке - скринридер немедленно объявит ошибку

!! (двойное отрицание):

Двойное отрицание `!!` преобразует любое значение в строгий boolean (true или false).

Примеры с разными значениями:

```
!!''              // false (пустая строка - falsy)
!!'текст'         // true  (непустая строка - truthy)
!!0               // false (ноль - falsy)
!!42              // true  (число не ноль - truthy)
!!null            // false (null - falsy)
!!undefined       // false (undefined - falsy)
!!{}              // true  (любой объект - truthy)
!![]              // true  (любой массив - truthy)
!!false           // false (boolean false - falsy)
!!true            // true  (boolean true - truthy)
```

Альтернативы двойного отрицания:

```
// вариант 1: двойное отрицание (короткий, идиоматичный)
aria-invalid={!!errors.name}

// вариант 2: явное сравнение (понятный, многословный)
aria-invalid={errors.name !== undefined}

// вариант 3: Boolean конструктор (редко используется)
aria-invalid={Boolean(errors.name)}
```

Мы используем `!!` потому что это короткая и широко принятая идиома в JavaScript для преобразования в boolean.

Полный исходный код WorkingAccessibleForm.jsx

```
import { useState } from 'react';
import './WorkingAccessibleForm.css';
```

```
function WorkingAccessibleForm() {
  // состояния для полей формы
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [message, setMessage] = useState('');

  // состояния для ошибок
  const [errors, setErrors] = useState({});

  // состояние отправки
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [submitSuccess, setSubmitSuccess] = useState(false);

  // валидация формы
  const validateForm = () => {
    const newErrors = {};

    // валидация имени
    if (!name.trim()) {
      newErrors.name = 'Имя обязательно для заполнения';
    } else if (name.trim().length < 2) {
      newErrors.name = 'Имя должно содержать минимум 2 символа';
    }

    // валидация email с помощью регулярного выражения
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!email) {
      newErrors.email = 'Email обязателен для заполнения';
    } else if (!emailRegex.test(email)) {
      newErrors.email = 'Введите корректный email адрес';
    }

    // валидация сообщения
    if (!message.trim()) {
      newErrors.message = 'Сообщение обязательно для заполнения';
    } else if (message.trim().length < 10) {
      newErrors.message = 'Сообщение должно содержать минимум 10 символов';
    }

    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };

  // обработчик отправки формы
  const handleSubmit = async (e) => {
    e.preventDefault();

    if (validateForm()) {
      setIsSubmitting(true);

      // имитация отправки на сервер
      await new Promise(resolve => setTimeout(resolve, 1500));

      setIsSubmitting(false);
```

```
      setSubmitSuccess(true);

      // очистка формы после успешной отправки
      setName('');
      setEmail('');
      setMessage('');

      // скрытие сообщения об успехе через 3 секунды
      setTimeout(() => {
        setSubmitSuccess(false);
      }, 3000);
    }
  };

  return (
    <div className="accessible-form-container">
      <h1>Контактная форма</h1>

      {/* область для скринридера - объявляет статус отправки */}
      <div
        role="status"
        aria-live="polite"
        aria-atomic="true"
        className="sr-only"
      >
        {isSubmitting && 'Отправка формы...'}
        {submitSuccess && 'Форма успешно отправлена!'}
      </div>

      {/* визуальное сообщение об успехе */}
      {submitSuccess && (
        <div className="success-message" role="alert">
          Спасибо! Ваше сообщение успешно отправлено.
        </div>
      )}

      <form onSubmit={handleSubmit} noValidate>
        {/* поле имени */}
        <div className="form-field">
          <label htmlFor="contact-name">
            Ваше имя <span aria-label="обязательное поле">*</span>
          </label>
          <input
            id="contact-name"
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
            aria-required="true"
            aria-invalid={!!errors.name}
            aria-describedby={errors.name ? 'name-error' : undefined}
            className={errors.name ? 'error' : ''}
          />
          {errors.name && (
            <span id="name-error" className="error-text" role="alert">
```

```
            {errors.name}
          </span>
        )}
      </div>

      {/* поле email */}
      <div className="form-field">
        <label htmlFor="contact-email">
          Email <span aria-label="обязательное поле">*</span>
        </label>
        <input
          id="contact-email"
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          aria-required="true"
          aria-invalid={!!errors.email}
          aria-describedby={errors.email ? 'email-error' : undefined}
          className={errors.email ? 'error' : ''}
        />
        {errors.email && (
          <span id="email-error" className="error-text" role="alert">
            {errors.email}
          </span>
        )}
      </div>

      {/* поле сообщения */}
      <div className="form-field">
        <label htmlFor="contact-message">
          Сообщение <span aria-label="обязательное поле">*</span>
        </label>
        <textarea
          id="contact-message"
          value={message}
          onChange={(e) => setMessage(e.target.value)}
          rows="5"
          aria-required="true"
          aria-invalid={!!errors.message}
          aria-describedby={errors.message ? 'message-error' : undefined}
          className={errors.message ? 'error' : ''}
        />
        {errors.message && (
          <span id="message-error" className="error-text" role="alert">
            {errors.message}
          </span>
        )}
      </div>

      {/* кнопка отправки */}
      <button
        type="submit"
        disabled={isSubmitting}
        aria-busy={isSubmitting}
```

```
          >
            {isSubmitting ? 'Отправка...' : 'Отправить'}
          </button>
        </form>
      </div>
    );
  }

  export default WorkingAccessibleForm;
```

Пример 3. Импорт и экспорт данных в JSON

Проблема: Нужно создать функционал для экспорта данных в JSON-файл и импорта данных из файла.

Подход к решению: Используем FileReader API для чтения файлов, Blob для создания файлов, и localStorage для сохранения данных между сессиями.

Пошаговое объяснение реализации:

Шаг 1: Экспорт данных в JSON-файл с подробным разбором

```
// экспорт данных в JSON-файл
const exportToJSON = () => {
  try {
    const dataStr = JSON.stringify(technologies, null, 2);
    const dataBlob = new Blob([dataStr], { type: 'application/json' });

    const url = URL.createObjectURL(dataBlob);
    const link = document.createElement('a');
    link.href = url;
    link.download = `technologies_${new Date().toISOString().split('T')[0]}.json`;

    document.body.appendChild(link);
    link.click();
    document.body.removeChild(link);

    URL.revokeObjectURL(url);

    setStatus('Данные экспортированы в JSON');
  } catch (error) {
    setStatus('Ошибка экспорта данных');
  }
};
```

JSON.stringify(technologies, null, 2):

Метод `JSON.stringify()` преобразует JavaScript объект или массив в JSON-строку. Принимает три параметра:

Параметр 1: value - данные для преобразования (technologies - массив технологий)

Параметр 2: replacer - функция или массив для фильтрации свойств (**null = включить все свойства**)

```
// null - включить все свойства объектов
JSON.stringify(data, null, 2)

// массив - включить только указанные свойства
JSON.stringify(data, ['title', 'category'], 2)
// результат содержит только title и category, без остальных полей
```

Параметр 3: space - отступы для форматирования (**2 = использовать 2 пробела для отступов**)

```
// 2 - красиво отформатированный JSON с отступами в 2 пробела
JSON.stringify(data, null, 2)
/* результат:
[
  {
    "id": 1,
    "title": "React",
    "category": "frontend"
  }
]
*/

// 0 или undefined - компактный JSON в одну строку без форматирования
JSON.stringify(data, null, 0)
// результат: [{"id":1,"title":"React","category":"frontend"}]
```

Зачем используем (technologies, null, 2):

- `technologies` - наши данные для экспорта
- `null` - сохраняем все свойства объектов без фильтрации
- `2` - форматируем с отступом 2 пробела для читаемости человеком

Blob:

Blob (Binary Large Object) - это объект, представляющий неизменяемые сырые данные, похожие на файл. Используется для работы с бинарными или текстовыми данными как с файлом.

Создание Blob:

```
const dataBlob = new Blob([dataStr], { type: 'application/json' });
```

Параметры конструктора:

1. **Массив данных** `[dataStr]` - может содержать строки, ArrayBuffer, другие Blob объекты

2. **Опции** `{ type: 'application/json' }` - MIME-тип данных

Примеры создания разных Blob:

```
// blob из текста
new Blob(['Hello, World!'], { type: 'text/plain' });

// blob из JSON
new Blob([JSON.stringify(data)], { type: 'application/json' });

// blob из HTML
new Blob(['<h1>Title</h1>'], { type: 'text/html' });

// blob из нескольких частей
new Blob(['Hello', ' ', 'World'], { type: 'text/plain' });
```

Схема работы экспорта:

```
JavaScript данные
  → JSON.stringify()
  → JSON-строка
  → new Blob()
  → Blob объект
  → URL.createObjectURL()
  → blob://... URL
  → скачивание файла
```

Зачем нужен Blob: позволяет работать с данными как с файлом - создать URL для скачивания, отправить на сервер через FormData, прочитать через FileReader.

Шаг 2: Импорт данных с FileReader и readAsText

```
// импорт данных из JSON-файла
const importFromJSON = (event) => {
  const file = event.target.files[0];
  if (!file) return;

  const reader = new FileReader();

  reader.onload = (e) => {
    try {
      const imported = JSON.parse(e.target.result);

      if (!Array.isArray(imported)) {
        throw new Error('Неверный формат данных');
      }
```

```
        setTechnologies(imported);
        setStatus(`Импортировано ${imported.length} технологий`);
      } catch (error) {
        setStatus('Ошибка импорта: неверный формат файла');
      }
    };

    reader.readAsText(file);
    event.target.value = '';
  };
```

Зачем вызывается reader.readAsText(file):

Проблема: мы не можем напрямую получить содержимое File объекта:

```
const file = event.target.files[0];
console.log(file); // File { name: "data.json", size: 1234, ... }
console.log(file.content); // undefined! нет прямого доступа к содержимому
```

Решение - FileReader API:

FileReader - это API для асинхронного чтения содержимого файлов, выбранных пользователем.

```
const reader = new FileReader();

reader.onload = (e) => {
  // только ЗДЕСЬ мы получаем содержимое файла
  const content = e.target.result;
  console.log(content); // '{"id": 1, "title": "React"}'

  // теперь можем распарсить JSON
  const data = JSON.parse(content);
};

// запускаем асинхронное чтение файла
reader.readAsText(file);
```

Методы FileReader:

```
reader.readAsText(file);        // для текстовых файлов (.txt, .json, .csv) →
строка
reader.readAsDataURL(file);     // для изображений → base64 строка
reader.readAsArrayBuffer(file); // для любых файлов → бинарные данные
```

Почему чтение асинхронное:

- файл может быть большим (несколько МБ или ГБ)
- синхронное чтение заблокировало бы UI браузера
- асинхронное чтение не замораживает интерфейс

Схема работы импорта:

```
Пользователь выбирает файл
  → event.target.files[0]
  → File объект (только метаданные, без содержимого)
  → reader.readAsText(file)
  → браузер читает файл асинхронно
  → reader.onload срабатывает когда чтение завершено
  → e.target.result содержит текст файла
  → JSON.parse() преобразует строку в объект
  → setTechnologies() обновляет состояние
```

Зачем нужно event.target.value = '':

Проблема: браузер не вызывает событие `onChange` если пользователь выбирает тот же самый файл повторно.

Ситуация без очистки:

```
// шаг 1: пользователь выбирает data.json
// input.value = 'C:\fakepath\data.json'
// onChange срабатывает, файл импортирован

// шаг 2: пользователь исправляет data.json в редакторе
// шаг 3: пользователь снова выбирает data.json
// input.value уже = 'C:\fakepath\data.json' (не изменилось!)
// onChange НЕ СРАБАТЫВАЕТ, потому что значение не изменилось
// файл не импортируется заново
```

Решение - очистка значения:

```
reader.readAsText(file);
event.target.value = ''; // очищаем input после чтения файла
```

`reader.readAsText(file)` запускает асинхронное чтение содержимого файла в виде текстовой строки. Мы вызываем его после определения `reader.onload`, потому что сначала нужно установить обработчик события (ловушку), а только потом запускать асинхронную операцию, которая это событие вызовет — иначе событие может произойти до того, как мы успеем установить обработчик.

Ситуация с очисткой:

```
// шаг 1: пользователь выбирает data.json
// input.value = 'C:\fakepath\data.json'
// onChange срабатывает, файл импортирован
// input.value очищается до ''

// шаг 2: пользователь исправляет data.json
// шаг 3: пользователь снова выбирает data.json
// input.value меняется с '' на 'C:\fakepath\data.json'
// onChange СРАБАТЫВАЕТ, потому что значение изменилось!
// файл импортируется с новыми данными
```

Практический пример:

```
1. Импортировали data.json с 5 технологиями → input.value = ''
2. Добавили в data.json еще 3 технологии в редакторе
3. Импортировали data.json снова → onChange сработает → теперь 8 технологий

Без очистки: повторный импорт того же файла не сработает
С очисткой: можно импортировать один и тот же файл сколько угодно раз
```

Полный исходный код DataImportExport.jsx

```jsx
import { useState, useEffect } from 'react';
import './DataImportExport.css';

function DataImportExport() {
  // состояние для списка технологий
  const [technologies, setTechnologies] = useState([]);

  // состояние для сообщений о статусе операций
  const [status, setStatus] = useState('');

  // состояние для перетаскивания файла
  const [isDragging, setIsDragging] = useState(false);

  // загрузка данных из localStorage при монтировании компонента
  useEffect(() => {
    loadFromLocalStorage();
  }, []);

  // функция загрузки данных из localStorage
  const loadFromLocalStorage = () => {
    try {
      const saved = localStorage.getItem('technologies');
      if (saved) {
        const parsed = JSON.parse(saved);
        setTechnologies(parsed);
```

```
        setStatus('Данные загружены из localStorage');
        setTimeout(() => setStatus(''), 3000);
      }
    } catch (error) {
      setStatus('Ошибка загрузки данных из localStorage');
      console.error('Ошибка загрузки:', error);
    }
  };

  // функция сохранения данных в localStorage
  const saveToLocalStorage = () => {
    try {
      localStorage.setItem('technologies', JSON.stringify(technologies));
      setStatus('Данные сохранены в localStorage');
      setTimeout(() => setStatus(''), 3000);
    } catch (error) {
      setStatus('Ошибка сохранения данных');
      console.error('Ошибка сохранения:', error);
    }
  };

  // экспорт данных в JSON-файл
  const exportToJSON = () => {
    try {
      // преобразуем данные в JSON-строку с форматированием
      const dataStr = JSON.stringify(technologies, null, 2);

      // создаем Blob объект из строки
      const dataBlob = new Blob([dataStr], { type: 'application/json' });

      // создаем временную ссылку для скачивания
      const url = URL.createObjectURL(dataBlob);
      const link = document.createElement('a');
      link.href = url;
      link.download = `technologies_${new Date().toISOString().split('T')
[0]}.json`;

      // программно кликаем по ссылке для начала скачивания
      document.body.appendChild(link);
      link.click();
      document.body.removeChild(link);

      // освобождаем память
      URL.revokeObjectURL(url);

      setStatus('Данные экспортированы в JSON');
      setTimeout(() => setStatus(''), 3000);
    } catch (error) {
      setStatus('Ошибка экспорта данных');
      console.error('Ошибка экспорта:', error);
    }
  };

  // импорт данных из JSON-файла
```

```
  const importFromJSON = (event) => {
    const file = event.target.files[0];
    if (!file) return;

    const reader = new FileReader();

    // обработчик завершения чтения файла
    reader.onload = (e) => {
      try {
        const imported = JSON.parse(e.target.result);

        // проверка что импортированные данные - это массив
        if (!Array.isArray(imported)) {
          throw new Error('Неверный формат данных');
        }

        setTechnologies(imported);
        setStatus(`Импортировано ${imported.length} технологий`);
        setTimeout(() => setStatus(''), 3000);
      } catch (error) {
        setStatus('Ошибка импорта: неверный формат файла');
        console.error('Ошибка импорта:', error);
      }
    };

    // запускаем асинхронное чтение файла как текста
    reader.readAsText(file);

    // сбрасываем значение input для возможности повторного импорта того же файла
    event.target.value = '';
  };

  // обработчики drag-and-drop
  const handleDragOver = (e) => {
    e.preventDefault();
    setIsDragging(true);
  };

  const handleDragLeave = () => {
    setIsDragging(false);
  };

  const handleDrop = (e) => {
    e.preventDefault();
    setIsDragging(false);

    const file = e.dataTransfer.files[0];
    if (file && file.type === 'application/json') {
      // используем ту же логику чтения что и в importFromJSON
      const reader = new FileReader();
      reader.onload = (event) => {
        try {
          const imported = JSON.parse(event.target.result);
          if (Array.isArray(imported)) {
```

```
            setTechnologies(imported);
            setStatus(`Импортировано ${imported.length} технологий`);
            setTimeout(() => setStatus(''), 3000);
          }
        } catch (error) {
          setStatus('Ошибка импорта: неверный формат файла');
        }
      };
      reader.readAsText(file);
    }
  };

  return (
    <div className="data-import-export">
      <h1>Импорт и экспорт данных</h1>

      {/* статусное сообщение */}
      {status && (
        <div className="status-message">
          {status}
        </div>
      )}

      {/* кнопки управления */}
      <div className="controls">
        <button onClick={exportToJSON} disabled={technologies.length === 0}>
          Экспорт в JSON
        </button>

        <label className="file-input-label">
          Импорт из JSON
          <input
            type="file"
            accept=".json"
            onChange={importFromJSON}
            style={{ display: 'none' }}
          />
        </label>

        <button onClick={saveToLocalStorage} disabled={technologies.length === 0}>
          Сохранить в localStorage
        </button>

        <button onClick={loadFromLocalStorage}>
          Загрузить из localStorage
        </button>
      </div>

      {/* область drag-and-drop */}
      <div
        className={`drop-zone ${isDragging ? 'dragging' : ''}`}
        onDragOver={handleDragOver}
        onDragLeave={handleDragLeave}
        onDrop={handleDrop}
```

```
        >
          Перетащите JSON-файл сюда
        </div>

        {/* список импортированных технологий */}
        {technologies.length > 0 && (
          <div className="technologies-list">
            <h2>Импортированные технологии ({technologies.length})</h2>
            <ul>
              {technologies.map((tech, index) => (
                <li key={index}>
                  <strong>{tech.title}</strong> - {tech.category}
                </li>
              ))}
            </ul>
          </div>
        )}
      </div>
    );
  }

  export default DataImportExport;
```

Самостоятельная работа

Задание 1: Создайте форму для установки сроков изучения с валидацией. Форма должна включать валидацию, работающую в реальном времени, с понятными и доступными сообщениями об ошибках. Реализуйте доступность для пользователей с ограниченными возможностями (ARIA-атрибуты, навигация с клавиатуры).

Задание 2: Добавьте компонент для массового редактирования статусов технологий. Компонент должен позволять выбирать несколько технологий одновременно и корректно применять изменения ко всем выбранным элементам. Обеспечьте доступность формы для пользователей с ограниченными возможностями.

Задание 3: Проверьте корректность работы экспорта и импорта данных. Убедитесь, что экспорт создает валидный JSON файл, а импорт правильно обрабатывает ошибки формата и некорректные данные.