React.Context

Передача данных через props

Привет!

В этом материале рассмотрим интересную проблему, которая может возникнуть при разработке на React.

Добавим приложению для онлайн-бронирования, над которым мы работаем, возможность запоминать пользователя и использовать его данные в нужных местах. Например, чтобы после заполнения заявки на бронирование контактные данные сохранялись в **localStorage**и затем доставались оттуда при необходимости. Эта информация может понадобиться в любом месте приложения, поэтому её стоит хранить как можно выше в иерархии компонентов, например в корневом компоненте.

Добавлять эти данные будем в компонент, отвечающий за вывод формы подтверждения бронирования, а также в Greetings, приветствующий пользователя при входе в приложение. Передадим компонентам данные, используя props.

const queryClient = new QueryClient()  
  
function App() {  
  
    const [user] = useState<{ name: string; phone: string }>({  
  
        name: localStorage.getItem('username') || '',  
  
        phone: localStorage.getItem('phone') || '',  
  
    })  
  
    return (  
  
        <QueryClientProvider client={queryClient}>  
  
            <BaseLayout user={user} />  
  
        </QueryClientProvider>  
  
    )  
  
}  
  
export default App

export const BaseLayout = ({ user }: { user: { name: string; phone: string } }) => {  
  
    return (  
  
        <>  
  
            <header>  
  
                <a href="#">Главная</a>  
  
            </header>  
  
            <main>  
  
                <Greetings user={user} />  
  
                <ReservationPage user={user} />  
  
            </main>  
  
            <footer>2024 ©</footer>  
  
        </>  
  
    )  
  
}

export const ReservationPage = ({ user }: { user: { name: string; phone: string }}) => {  
  
    const [chosenRoom, chooseRoom] = useState<Room | null>(null);  
  
    const { data, isLoading } = useRoomList();  
  
    const handleOnRoomClick = (roomId: Room['id']) => {  
  
        const chosenRoom = data.find(room => room.id === roomId);  
  
        chosenRoom && chooseRoom(chosenRoom);      
  
    }  
  
    const getBackToFirstStep = () => chooseRoom(null);  
  
    if (isLoading) {  
  
        return <div>  
  
            <p>Пожалуйста, подождите</p>  
  
            <p>Номера, доступные для бронирования, скоро появятся здесь</p>  
  
        </div>  
  
    }  
  
    return (  
  
        <>  
  
            <div className='steps'>  
  
                <div className={`step ${!chosenRoom ? 'active' : ''}`}>  
  
                    <div className='step\_count'>1</div>  
  
                    <span>Выбор номера</span>  
  
                </div>  
  
                <div className='divider'></div>  
  
                <div className={`step ${chosenRoom ? 'active' : ''}`}>  
  
                    <div className='step\_count'>2</div>  
  
                    <span>Подтверждение бронирования</span>  
  
                </div>  
  
            </div>  
  
            {!chosenRoom ? (  
  
                <div className='card\_wrapper'>  
  
                    <AvailableRooms onChooseRoom={handleOnRoomClick} rooms={data || []} />  
  
                </div>  
  
            ) : (  
  
                <ReservationForm room={chosenRoom} user={user} onBack={getBackToFirstStep} />  
  
            )}  
  
        </>  
  
    )  
  
}

type TProps = {  
  
    room: Room;  
  
    onBack: () => void;  
  
    user: { name: string; phone: string; }  
  
}  
  
export const ReservationForm = ({ room, onBack, user }: TProps) => {  
  
    const [name, setName] = useState(user.name);  
  
    const [phone, setPhone] = useState(user.phone);  
  
    const { isSuccess, mutate, isPending } = useMutation({  
  
        mutationFn: getPostRequestForRoomReservation,  
  
    })  
  
    const submitForm = () => {  
  
        if (!name || !validateByPhone(phone)) {  
  
            alert('Введите имя или правильный номер телефона')  
  
        } else {  
  
            const response = mutate({ name, phone });  
  
        }  
  
    }  
  
    useEffect(() => {  
  
        if (isSuccess) {  
  
            alert('Спасибо! Наши менеджеры скоро свяжутся с Вами.')  
  
        }  
  
    }, [isSuccess]);  
  
  
  
    if (isPending) {  
  
        <div>Заявка отправляется...</div>  
  
    }  
  
    return (  
  
        <div className='form\_wrapper'>  
  
            <div>Вы бронируете {room.title}.  
  
            <br/> Оставьте ваши контактные данные для связи с оператором</div>  
  
            <form>  
  
                <TextField  
  
                     placeholder='Ваше имя'  
  
                     value={name}  
  
                     onChange={(e: React.ChangeEvent<HTMLInputElement>) => setName(e?.target?.value)}  
  
                />  
  
                <TextField  
  
                     placeholder='Ваш номер телефона'  
  
                     value={phone}  
  
                     onChange={(e: React.ChangeEvent<HTMLInputElement>) => setPhone(e?.target?.value)}  
  
                />  
  
            </form>  
  
            <div className='button\_wrapper'>  
  
                <Button type='outline' onClick={onBack}>Назад</Button>  
  
                <Button  
  
                    disabled={!name || !phone}  
  
                    onClick={submitForm}  
  
                >  
  
                    Отправить заявку на бронирование  
  
                </Button>  
  
            </div>  
  
        </div>  
  
    );  
  
}

Проследить путь объекта user к цели (компонентам Greetings и ReservationForm) может быть непросто, особенно если вы не знакомы с React.

В этом примере мы столкнулись с проблемой: при переносе данных **user**из компонента **App**в компонент **ReservationForm**их также нужно передавать в два компонента-посредника. У каждого из них появляется новый **prop,** за передачу которого в компонент ответственны разработчики.  
С ростом приложения количество таких **props** будет увеличиваться, как и количество уровней компонентов, через которые они будут передаваться. В итоге большинство **props** могут оказаться не нужны компонентам, но необходимы их потомкам. Из-за этого компоненты будут накрепко связаны с той структурой компонентов, в которой они определены.

Prop drilling

В предыдущем блоке мы рассматривали передачу данных между компонентами с помощью props.

Но это неподходящий способ в ситуациях, когда компонент с данными находится на несколько уровней выше компонента — потребителя этих данных. В этом случае проброс данных дублируется дереву компонентов сверху вниз. Эта ошибка называется prop drilling.

Prop drilling — это антипаттерн разработки на React, или подход, которого стоит избегать, так как он приводит к следующим проблемам:

* **Увеличивающаяся сложность с каждым новым уровнем проброса**.  
  С ростом приложения становится трудно следить за потоком данных, проходящим через множество различных компонентов.
* **Проблемы с повторным использованием компонентов**.  
  Каждый компонент-посредник в таком дереве жёстко привязан к своему месту в структуре, в которой он используется. Перенести его в другое место может быть трудно.
* **Сложность поддержки.**Нужно постоянно отслеживать, что происходит с потоком данных в компонентах, обеспечивающих его, а также корректность передаваемых компонентом вниз данных. В результате даже небольшие изменения в передаваемых данных могут потребовать изменений в каждом компоненте всего дерева.

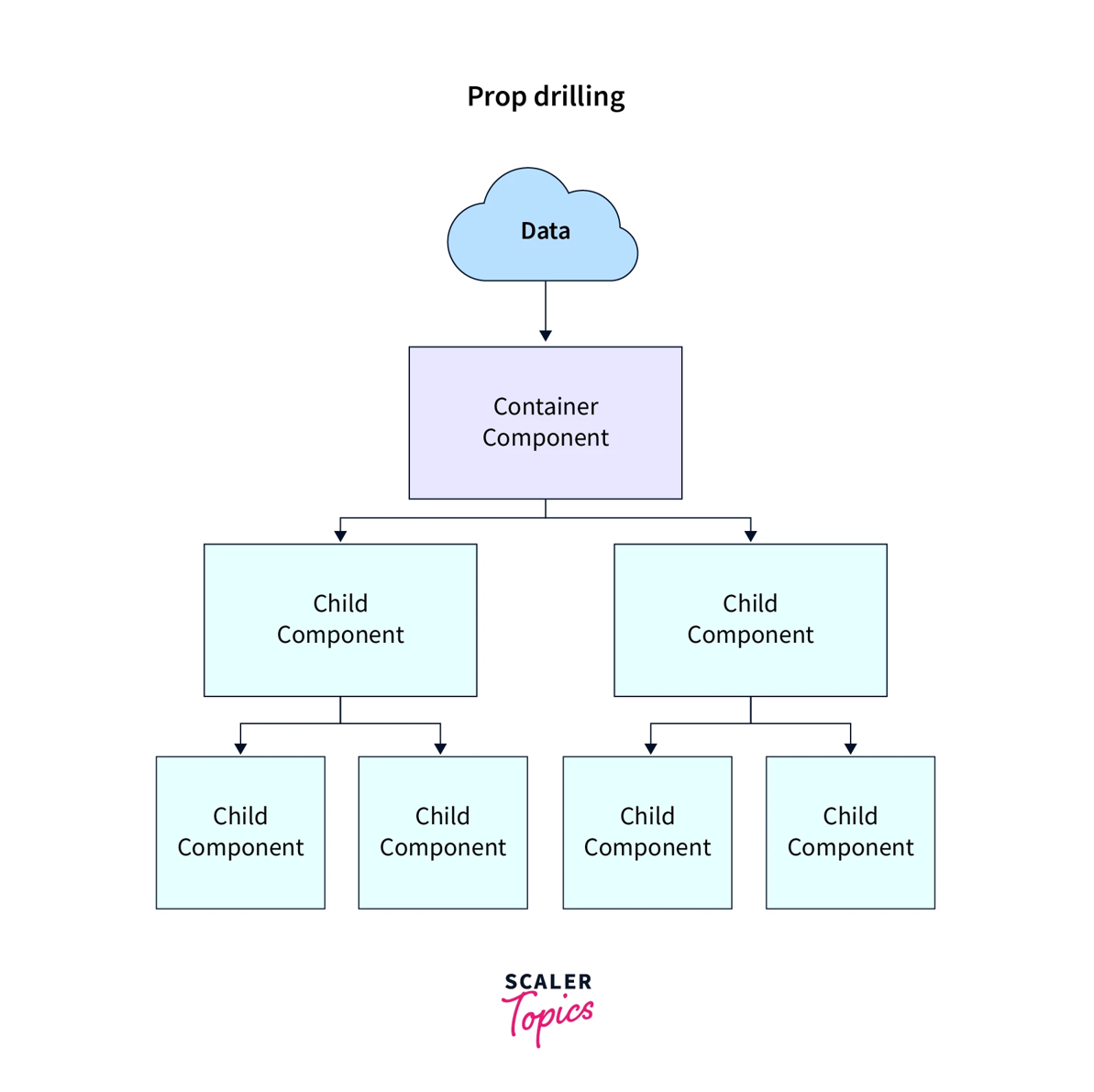


Схема: Eshaan Gupta / [InterviewBit Technologies](https://www.scaler.com/topics/react/contex-api/" \t "_blank)

React.Context

Решение подобной проблемы есть в React — Context (Контекст). Контекст — это встроенный в программу инструмент, с помощью которого можно передавать данные от родителя сразу к любому из его дочерних компонентов, не прибегая к компонентам-посредникам. Данные появляются в компоненте-потребителе как будто по его вызову.

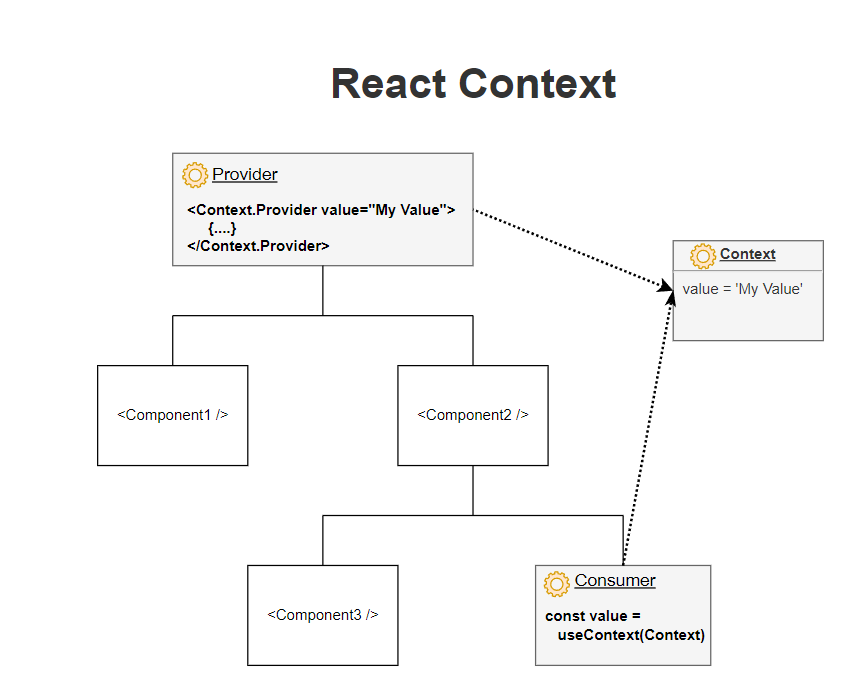


Схема: Dmitri Pavlutin / [Dmitri Pavlutin](https://dmitripavlutin.com/react-context-and-usecontext/" \t "_blank)

Его синтаксис довольно простой. Давайте изучим его на классическом кейсе использования React.Context — на темизации приложения.

Темизация — это кастомизация внешнего вида сайта по заранее определённым темам. Например, переключение тёмной темы сайта на светлую.

1. Объявляем контекст в отдельном файле. Укажем значение по умолчанию: его будут использовать компоненты-потребители, если контекст не будет обозначен в качестве одного из родителей.

import { createContext } from 'react';  
  
export const ThemeContext = createContext('light');

2. Затем создадим контекст для нашего дерева.   
Импортируем экземпляр контекста в файл приложения и вставим его в дерево компонентов таким образом, чтобы все требующие данных компоненты стали дочерними компонентами. Также в ThemeContext.Provider необходимо передать значение, которое он будет отдавать по запросу своим потомкам.

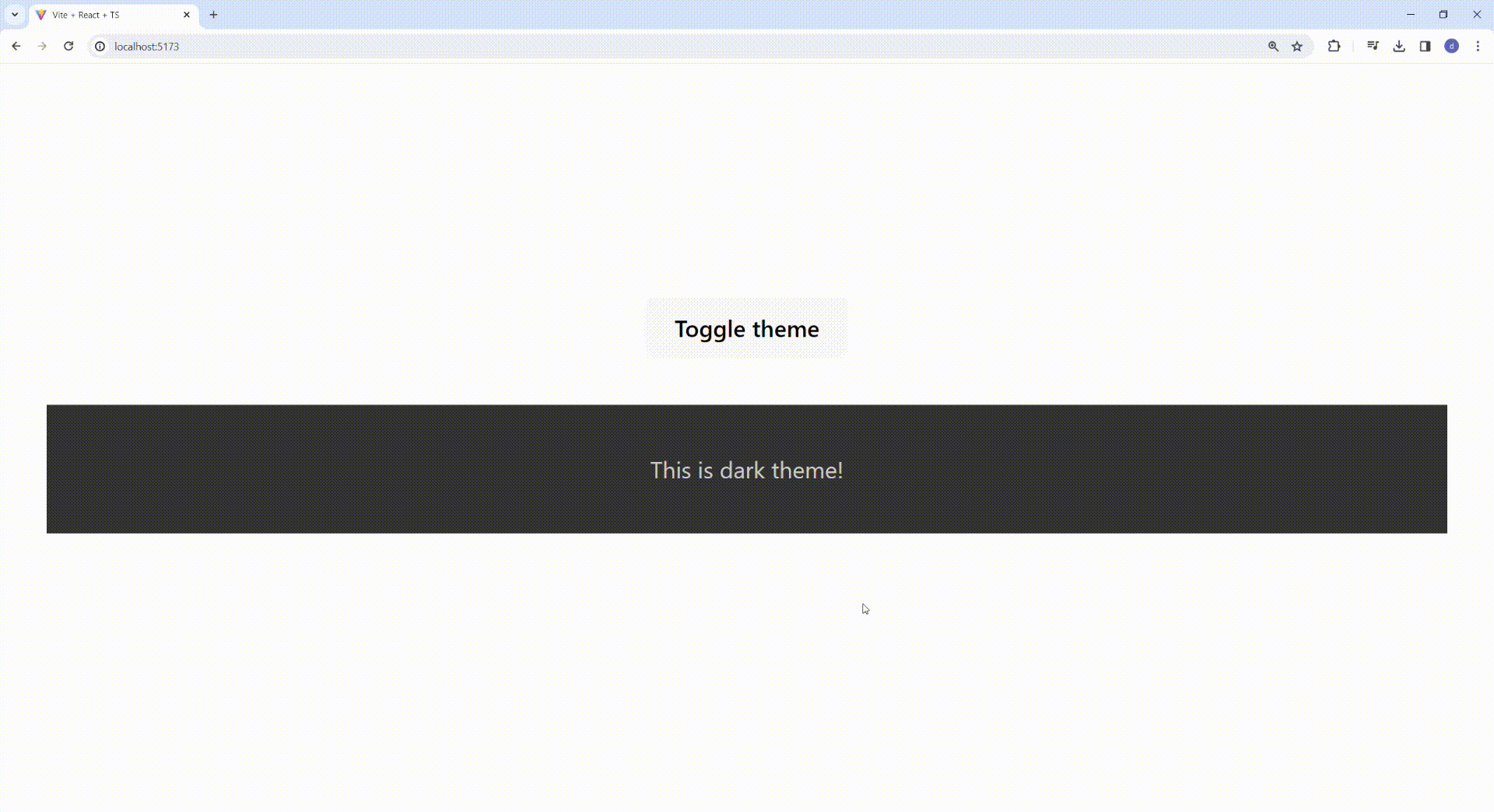
import { ChildComponent } from './ChildComponent';  
  
import { ThemeContext } from './ThemeContext';  
  
import { useState } from 'react';  
  
  
  
function App() {  
  
  const [theme, setTheme] = useState('light');  
  
  
  
  const changeTheme = () => setTheme(theme === 'light' ? 'dark' : 'light');  
  
  
  
  return (  
  
    <div className='App'>  
  
      <button onClick={() => changeTheme()}>Toggle theme</button>  
  
      <ThemeContext.Provider value={theme}>  
  
        <ChildComponent />  
  
      </ThemeContext.Provider>  
  
    </div>  
  
  );  
  
}  
  
  
  
export default App;

3. После этого вызовем в Child Component хук useContext, через который получим переданное в ThemeContext.Provider значение.   
После вывода значения текущей темы определяем и применяем стили.

import { useContext } from 'react';  
  
import { ThemeContext } from './ThemeContext';  
  
  
  
export const ChildComponent = () => {  
  
  const theme = useContext(ThemeContext)  
  
  
  
  const themeStyles = {  
  
    backgroundColor: theme === 'dark' ? '#333' : '#ccc',  
  
    color: theme === 'dark' ? '#ccc' : '#333',  
  
    padding: '2rem',  
  
    margin: '2rem'  
  
  }  
  
  
  
  return (  
  
    <div style={themeStyles}>  
  
        This is {theme} theme!  
  
    </div>  
  
  )  
  
}

Готово! Теперь дочерний компонент видит значение, которое определяется и управляется через родительский компонент, не применяя при этом props. Благодаря этому мы можем удобно переключать темизацию приложения.

Реализация темизации, основанная на Context, во многом похожа на реализацию на чистом CSS, когда CSS-переменные меняют значение в зависимости от положения в каскаде стилей.



Скринкаст: Александр Гайдаржи

Темизация — это только один из способов применения Контекста. Помимо этого, его можно использовать для сохранения в контекст данных о текущем авторизованном пользователе. Этот кейс мы и разберём дальше.

Добавление Context в приложение

Давайте изменим приложение по бронированию отелей, которое мы разбирали в прошлых темах. После того как пользователь отправит заявку, наше приложение сможет сохранить его имя и номер телефона в localStorage, подставлять эти данные при следующих заполнениях формы и выводить приветствие. Добавим эту возможность с помощью Context, чтобы любой компонент узнавал данные пользователя, не передавая объект от родителя наверху дерева через пропсы.

1. Сначала создадим контекст. В качестве значений он будет принимать имя и номер телефона пользователя.

import { createContext } from 'react';  
  
type ContextParams = { name: string; phone: string; };  
  
export const UserContext = createContext<ContextParams>(  
  
    {  
  
        name: '',  
        phone: '',  
  
    }  
  
);

2. Затем в корневом компоненте App поместим UserContext.Provider на самый верхний уровень. Таким образом, у каждого компонента приложения будет доступ к значениям данного контекста.

function App() {  
  
    return (  
  
        <UserContext.Provider value={{ '', phone: '' }}>  
            <QueryClientProvider client={queryClient}>  
                <BaseLayout />  
            </QueryClientProvider>  
        </UserContext.Provider>  
    )  
}  
  
export default App

3. При инициализации приложения проверим в localStorage наличие сохранённых **name**и**phone**. Если они есть, сохраняем их в state. Затем передаём всё это в контекст.

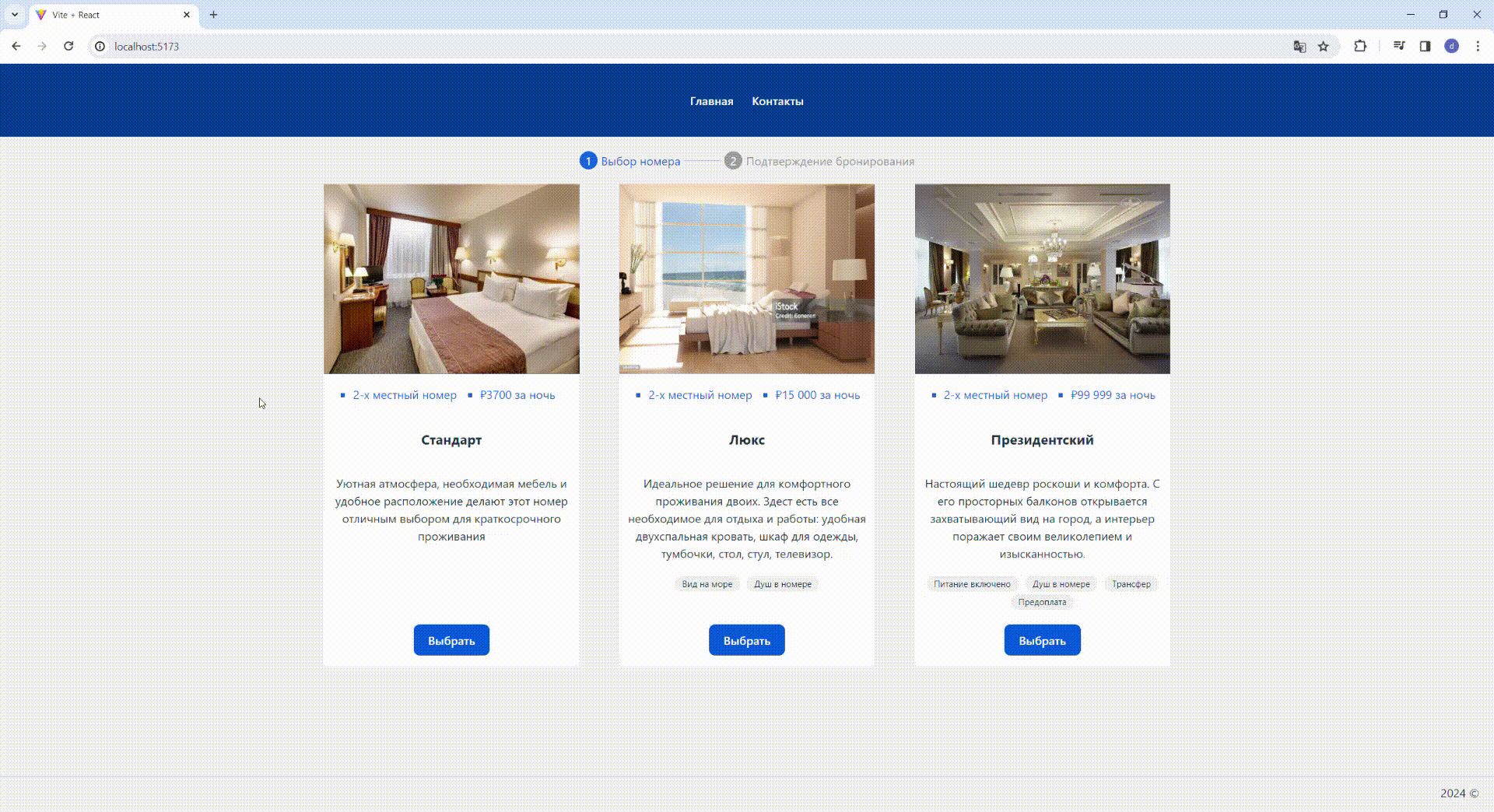
function App() {  
  
    const [user] = useState<{ name: string; phone: string }>({  
  
        name: localStorage.getItem('username') || '',  
  
        phone: localStorage.getItem('phone') || '',  
  
    })  
  
  
  
    return (  
  
        <UserContext.Provider value={{ name: user.name, phone: user.phone }}>  
  
            <QueryClientProvider client={queryClient}>  
  
                <BaseLayout />  
  
            </QueryClientProvider>  
  
        </UserContext.Provider>  
  
    )  
  
}  
  
  
export default App

4. Перейдём к компонентам-потребителям. Используя хук useContext в форме отправки заявки, получаем имя и телефон пользователя и подставляем их в нужные области.

Помните: чтобы получить данные, их нужно для начала сохранить. Сохраним данные пользователя в localStorage после успешной валидации и перед отправкой.

export const ReservationForm = ({ room, onBack }: TProps) => {  
  
    const { name: savedName, phone: savedPhone } = useContext(UserContext);  
  
    const [name, setName] = useState(savedName || '');  
  
    const [phone, setPhone] = useState(savedPhone || '');  
  
  
  
    const { isSuccess, mutate, isPending } = useMutation({  
  
        mutationFn: getPostRequestForRoomReservation,  
  
    })  
  
  
  
    const submitForm = () => {  
  
        if (!name || !validateByPhone(phone)) {  
  
            alert('Введите имя или правильный номер телефона')  
  
        } else {  
  
            localStorage.setItem('username', name);  
  
            localStorage.setItem('phone', phone);  
  
  
  
            const response = mutate({ name, phone });  
  
        }  
  
    }

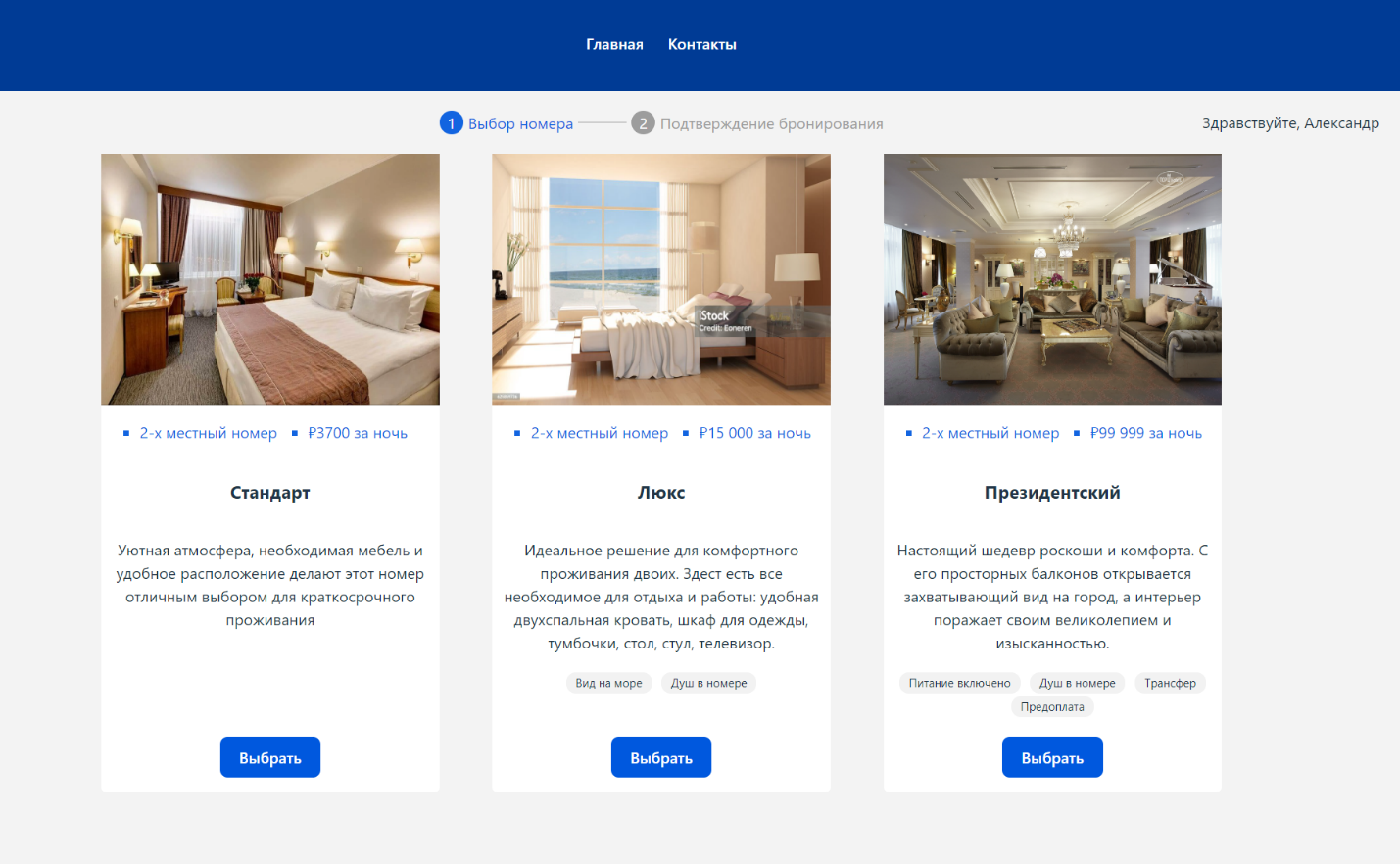
Вот и всё. Теперь приложение запоминает данные пользователя из заполненной им формы и подставляет эти значения через контекст.



Скринкаст: Александр Гайдаржи

В конце добавим компонент Greetings, который также будет считывать данные пользователя из контекста (если есть) и выводить их в приветствии.

const Greetings = () => {  
  
    const { name, phone } = useContext(UserContext);  
  
  
  
    if (name) {  
  
        return <div style={{ position: 'absolute', right: '30px' }}>Здравствуйте, {name}</div>  
  
    }  
  
    return null;  
  
}



Изображение: Александр Гайдаржи

Выводы

В этом материале вы разобрали, что такое Context в React-приложениях, на распространённых случаях его применения.

React.Context — это полезный инструмент разработки, который должен быть в арсенале у каждого React-разработчика. Его можно использовать для упрощения и оптимизации процесса создания программы, особенно в случаях, когда необходимо распределять общие данные между различными компонентами.

Использование Context не ограничивается вышеупомянутыми задачами. Его также используют в разных библиотеках на React для работы с разными аспектами приложения — для управления состоянием, обработки запросов, приходящих на сервер, или перехода между страницами.