Politechnika Świętokrzyska w Kielcach

Wydział Elektrotechniki, Automatyki i Informatyki

Politechnika Świętokrzyska
Kielce University of Technology

Zaawansowane aplikacje frontendowe

Projekt

Aplikacja do zarządzania siłownią

Skład zespołu:
Rafał Grot
Filip Stępień
Bartłomiej Karkoszka
Mateusz Karbowniczak

Kierunek/specjalność:	Informatyka Systemy informacyjne
Studia:	stacjonarne
Numer grupy:	3ID11B

Kielce, 18 czerwca 2025

1 Wprowadzenie

1.1 Krótki opis aplikacji

Celem projektu było stworzenie aplikacji służącej do zarządzanie siłownią. System ten został zaprojektowany z myślą o czterech różnych typach użytkowników: kliencie, pracowniku, trenerze oraz menadżerze.

Aplikacja umożliwia m.in.:

- śledzenie postępów treningowych przez użytkownika,
- przegląd i edycję zaplanowanych sesji treningowych,
- zapisywanie przebiegu własnych treningów,
- odnawianie karnetu,
- wgląd w listę klientów i pracowników (dla kadry zarządzającej),
- zarządzanie salami treningowymi,
- tworzenie i edytowanie kont pracowników oraz klientów.

1.2 Wykorzystane technologie oraz narzędzia

Do stworzenia aplikacji wykorzystano następujące technologie i narzędzia:

- **TypeScript** – nadzbiór języka JavaScript umożliwiający statyczne typowanie,
- **Vite** – bundler i środowisko uruchomieniowe do aplikacji frontendowych,
- **Vitest** – framework testowy, zintegrowany z ekosystemem Vite,
- **React** – biblioteka JavaScript służąca do budowy interfejsów użytkownika,
- **React Router** – biblioteka do zarządzania nawigacją i routingiem w aplikacjach React,
- **Axios** – biblioteka do obsługi zapytań HTTP, wykorzystywana do komunikacji z backendem przez REST API,
- **OpenAPI** – standard do opisywania interfejsów REST API,
- **Orval** – narzędzie generujące klienta API na podstawie specyfikacji OpenAPI, ułatwiające komunikację z backendem,
- **Figma** – narzędzie do projektowania interfejsów użytkownika i prototypowania,
- **Task** – narzędzie do automatyzacji zadań deweloperskich,
- **Git** – system kontroli wersji wspierający zarządzanie kodem źródłowym oraz pracę zespołową,
- **GitHub** – serwis internetowy umożliwiający przechowywanie i zarządzanie repozytoriami Git,
- **GitHub Actions** – platforma CI/CD umożliwiająca automatyzację procesów związanych z budowaniem, testowaniem i wdrażaniem aplikacji,
- **Renovate** – bot do automatycznego aktualizowania zależności projektu,

- **Nix** – narzędzie do zarządzania środowiskiem programistycznym w systemie NixOS,
- **AntDesign** – biblioteka komponentów UI dla React,
- **Chart.js** – biblioteka do tworzenia dynamicznych wykresów,
- **Keycloak JS** – biblioteka służąca do integracji aplikacji frontendowej z systemem *Keycloak* do autoryzacji użytkowników,
- **Tailwind CSS** – biblioteka CSS do szybkiego stylowania interfejsów użytkownika,
- **Day.js** – biblioteka do obsługi dat i czasu.

2 Implementacja

2.1 Opis głównych funkcjonalności aplikacji

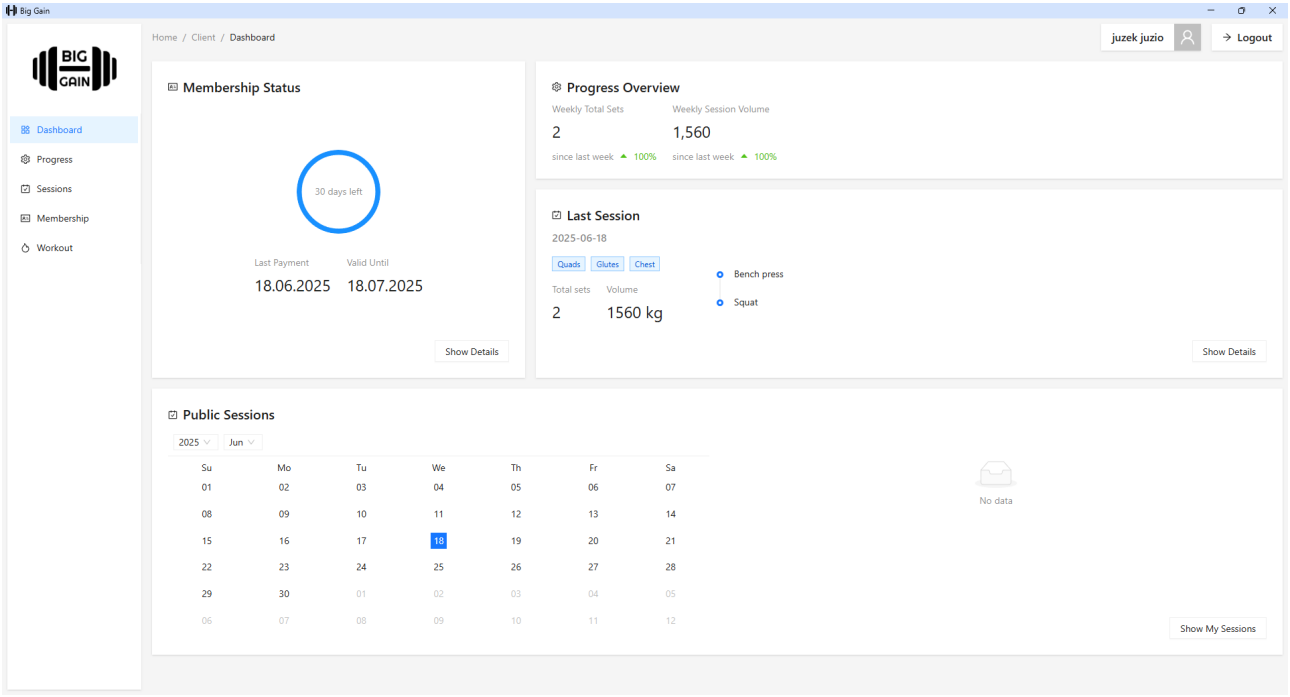
Aplikacja do zarządzania siłownią została zaprojektowana z myślą o różnych rolach użytkowników: klientach, pracownikach, trenerach oraz menadżerach. W zależności od przypisanej roli dostępne są różne funkcjonalności.

- **Klient** może:
  - Przeglądać status swojego karnetu.
  - Śledzić swój postęp treningowy.
  - Przeglądać i zapisywać się na dostępne sesje treningowe.
  - Przeglądać historię treningów.
- **Menadżer** ma możliwość:
  - Tworzenia nowych pracowników poprzez formularz.
  - Przeglądania listy klientów i pracowników.
  - Zarządzania salami treningowymi oraz zadaniami serwisowymi.
- **Pracownik** ma możliwość:
  - Przeglądania listy klientów.
  - Sprawdzania ważności karnetów.
  - Zgłaszania usterek i tworzenia zadań serwisowych.
- **Trener** może:
  - Przeglądać przypisanych podopiecznych.
  - Tworzyć i edytować plany treningowe.
  - Zarządzać kalendarzem sesji treningowych.

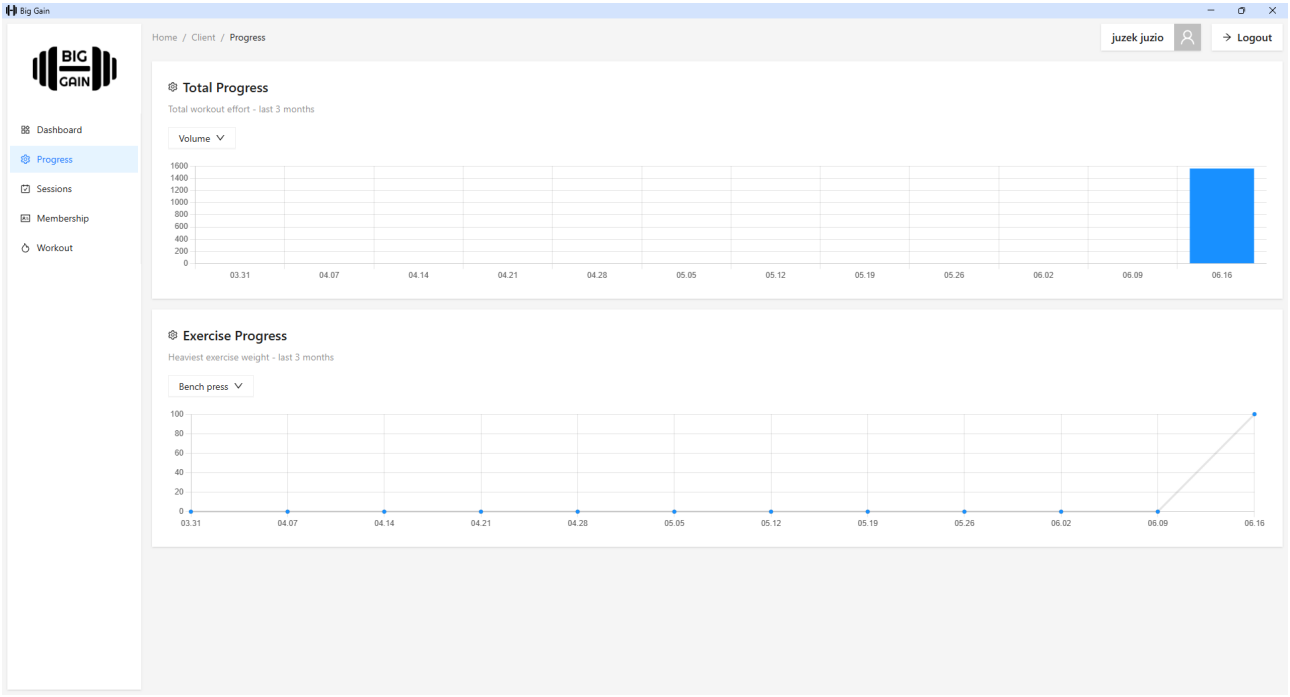
Aplikacja obsługuje różne widoki dla użytkowników w zależności od ich roli i zapewnia prosty oraz intuicyjny interfejs użytkownika, co przedstawiono na poniższych zrzutach ekranu.

Ze względu na dużą ilość zakładek pokazane zostały najczęściej używane ekrany.

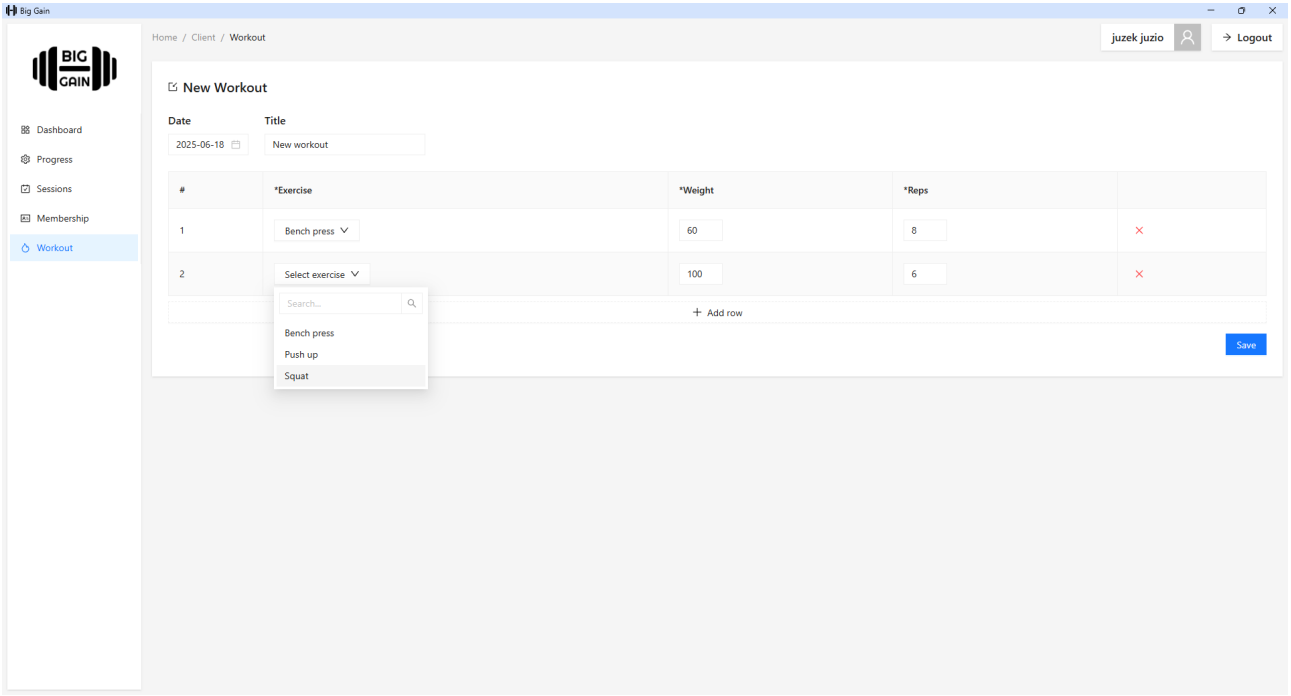
2.2 Prezentacja zrzutów ekranu (screeny) prezentujących działanie aplikacji



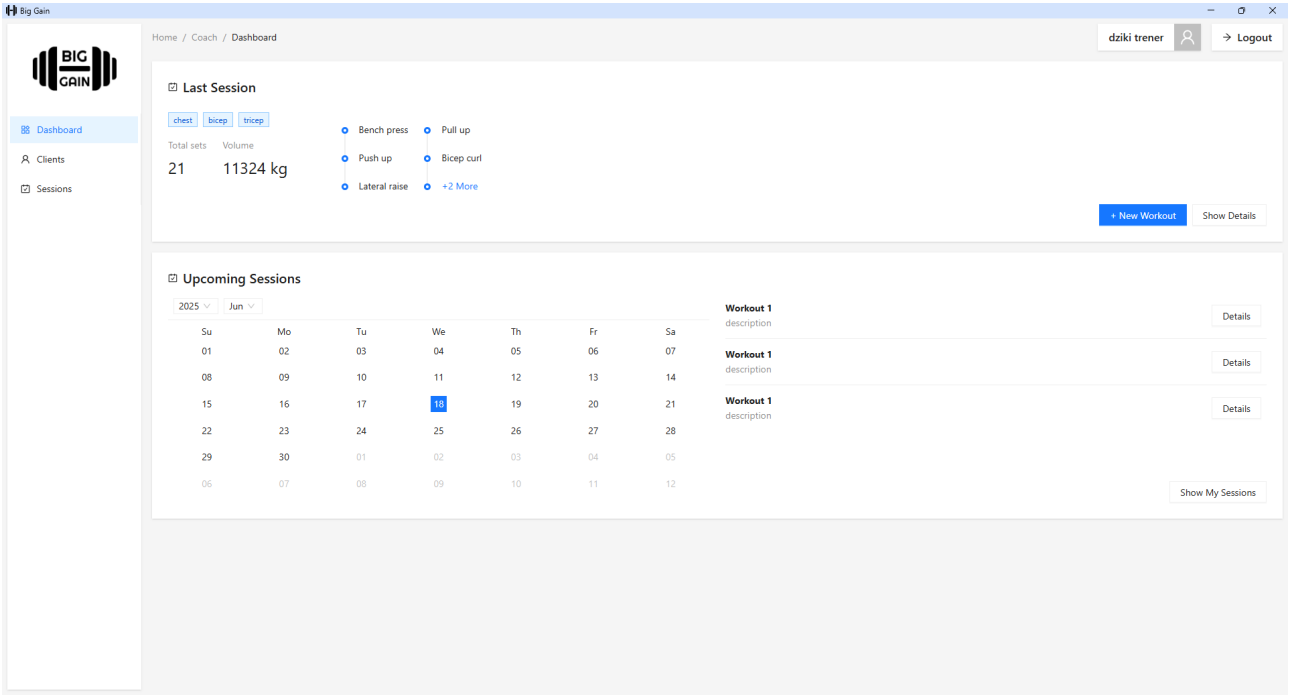
Rysunek 1: Panel klienta – przegląd statusu członkostwa, progresu oraz kalendarz sesji.



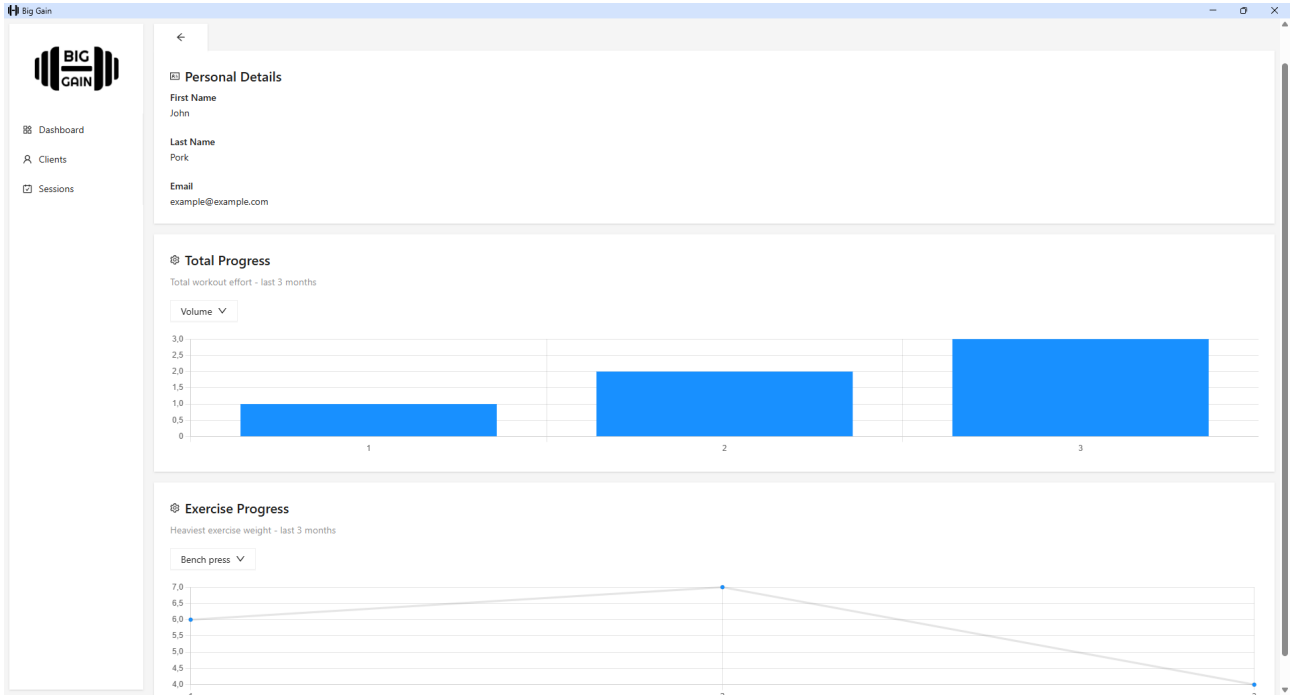
Rysunek 2: Klient może śledzić swoje statystyki.



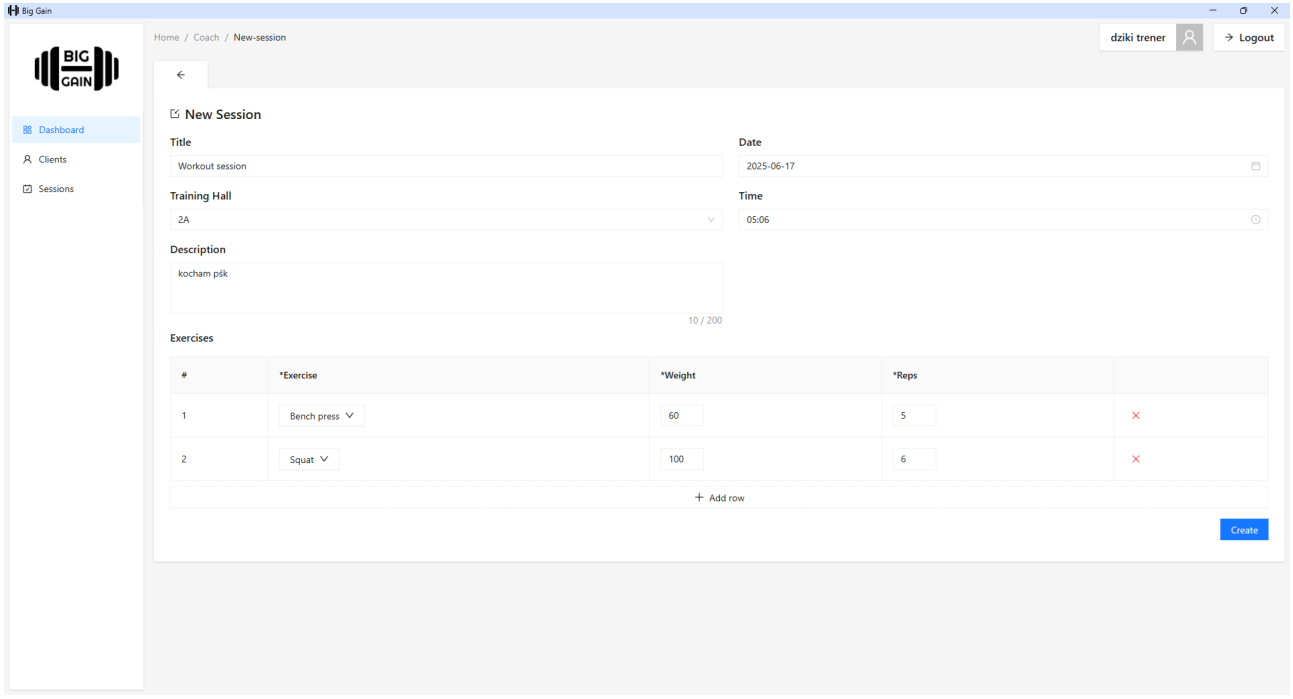
Rysunek 3: Nowy trening - klient może tworzyć własne sesje.



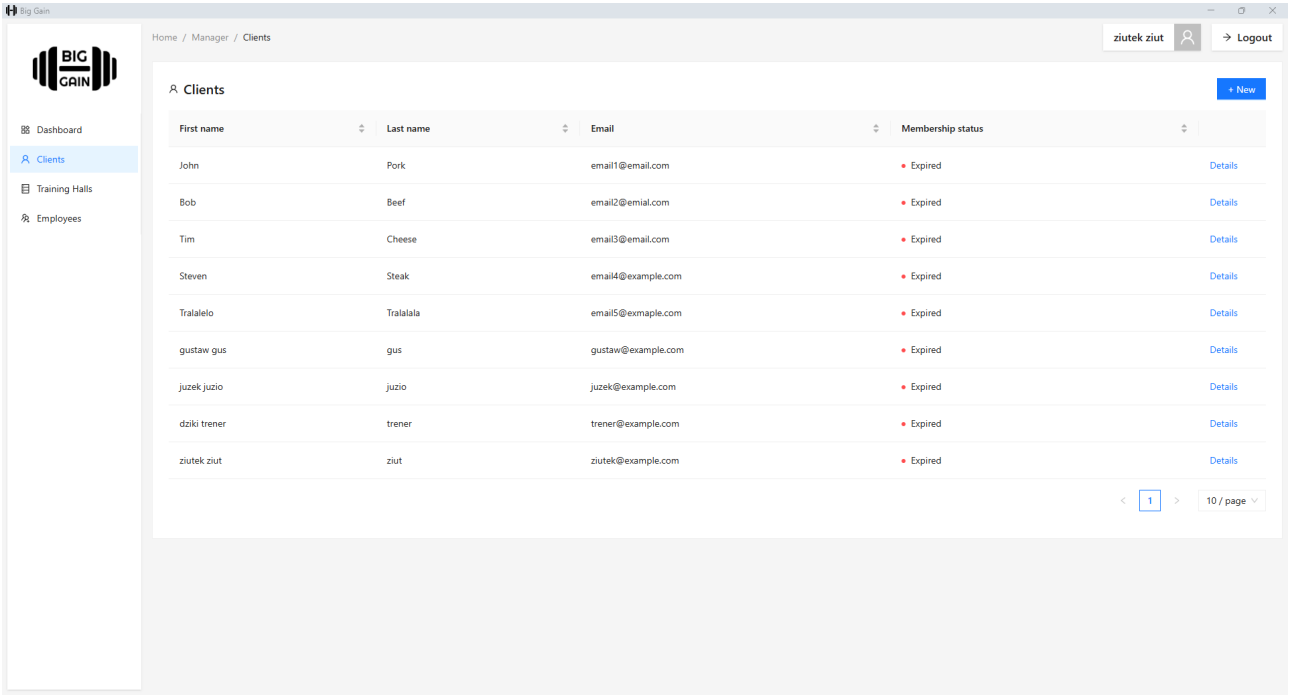
Rysunek 4: Panel trenera - ostatni trening, kalendarz nadchodzących sesji.



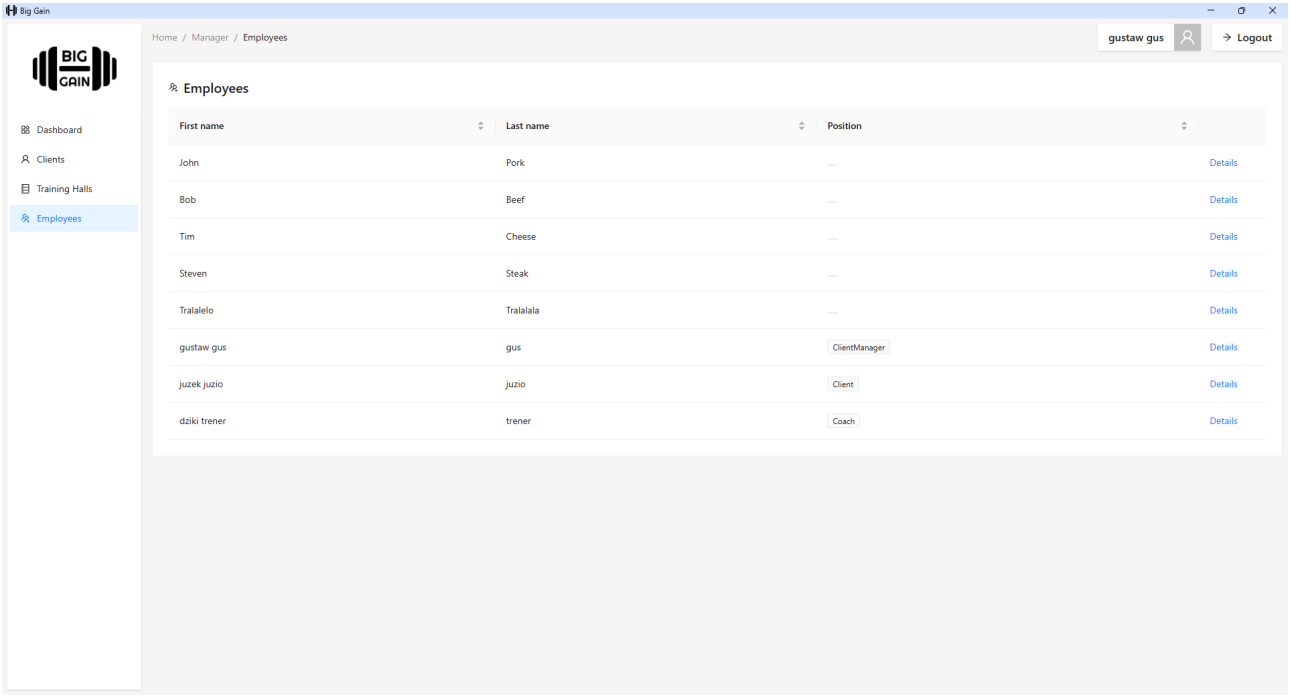
Rysunek 5: Również trener może przeglądać statystyki treningowe klienta.



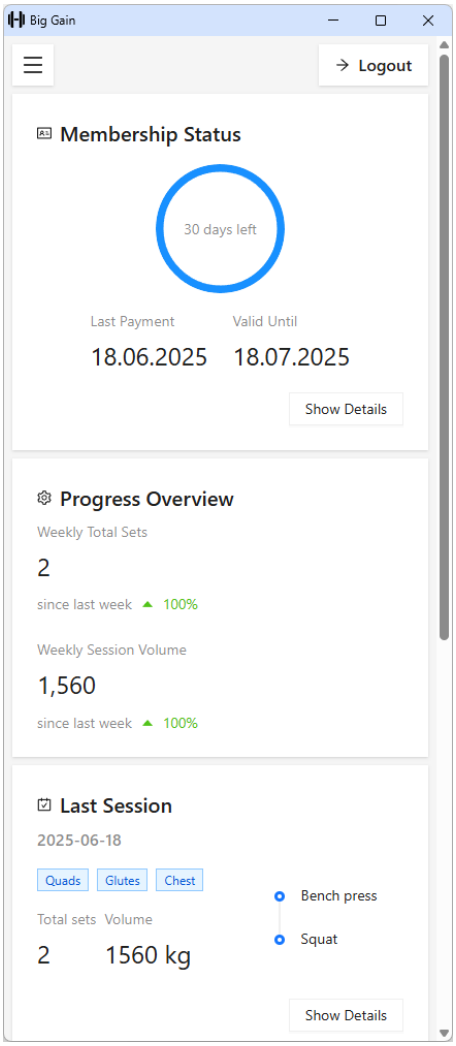
Rysunek 6: Tworzenie nowej sesji treningowej przez trenera.



Rysunek 7: Manager jak i pracownik mają dostęp do wszystkich klientów.



Rysunek 8: Manager może przeglądać dane pracowników.



Rysunek 9: Panel użytkownika na urządzeniu mobilnym - cała aplikacja jest responsywna.

2.3 Wybrane fragmenty kodu z kluczowymi funkcjonalnościami

```
function getValidityLabel(lastPayment?: Dayjs, validUntil?: Dayjs) {
    if (!lastPayment || !validUntil) {
        return 'Expired';
    }

    const now = dayjs();
    const until = dayjs(validUntil);

    if (until.isBefore(now)) {
        return 'Expired';
    }

    const diffDays = until.diff(now, 'day');
    const diffDayjsWithToday = diffDays + 1;

    if (diffDays >= 1) {
        return `${diffDayjsWithToday} day${diffDayjsWithToday === 1 ? '' : 's'} left`;
    }

    const diffHours = until.diff(now, 'hour');
    return `${diffHours} hour${diffHours === 1 ? '' : 's'} left`;
}

function getCirclePercentage(lastPayment?: Dayjs, validUntil?: Dayjs): number {
    if (!lastPayment || !validUntil) return 0;

    const now = dayjs();

    if (now.isAfter(validUntil, 'day')) return 0;
    if (now.isBefore(lastPayment, 'day')) return 100;

    const total = validUntil.diff(lastPayment, 'days');
    const elapsed = now.diff(lastPayment, 'days');

    return Math.min(100, Math.max(0, (1 - elapsed / total) * 100));
}

// ...
```

```
const { lastPayment, validUntil } = props;
const lastPaymentDate = dayjs(lastPayment);
const validUntilDate = dayjs(validUntil);

<Progress
        type='circle'
        percent={
            lastPayment && validUntilDate
                ? getCirclePercentage(lastPaymentDate, validUntilDate)
                : 0
        }
        format={() => (
            <div className='text-font-secondary text-sm'>
                {getValidityLabel(lastPaymentDate, validUntilDate)}
            </div>
        )}
        size={125}
    />
// ...
```

Listing 1: Przykład logiki obliczeniowej w komponencie. Funkcje służą do wyznaczania wskaźników statusu karnetu dla komponentu *Progress* z *Ant Design*.

```
export function Chart(props: ChartProps) {
    const { chartData, type, className, dropdownType } = props;

    const initialChartDataEntry = chartData?.data?.at(0);

    const [chartDataEntry, setChartDataEntry] = useState<ChartEntry | undefined>(
        initialChartDataEntry
    );

    useEffect(() => {
        setChartDataEntry(initialChartDataEntry);
    }, [initialChartDataEntry]);

    if (!chartData) {
        return <Empty />;
    }

    const menuItems = chartData.data.map(({ title }) => ({ key: title, label: title }));

    const dataExists =
        chartDataEntry &&
        chartDataEntry.timeSeries.labels.length > 0 &&
        chartDataEntry.timeSeries.values.length > 0;

    const chartComponentData = {
        labels: chartDataEntry?.timeSeries.labels,
        datasets: [
            {
                data: chartDataEntry?.timeSeries.values,
                backgroundColor: getCSSVariable('--color-primary')
            }
        ]
    };

    const handleMenuItemSelect = (item: { key: string; label: string }) => {
        const dataEntry = chartData.data.find(({ title }) => title === item.key);
        setChartDataEntry(dataEntry);
    };

    const menu =
        dropdownType === 'search' ? (
            <SearchDropdown
                placeholder={initialChartDataEntry?.title ?? 'Select'}
                menuItems={menuItems}
                onSelect={handleMenuItemSelect}
            />
        ) : (
            <Dropdown
                placeholder={initialChartDataEntry?.title ?? 'Select'}
                menuItems={menuItems}
                onSelect={handleMenuItemSelect}
            />
        );

    const chart =
        type === 'line' ? (
            <Line options={chartComponentOptions} data={chartComponentData} />
        ) : (
            <Bar options={chartComponentOptions} data={chartComponentData} />
        );

    return (
        <Flex vertical>
            <Text className='text-font-secondary mb-middle'>{chartData.description}</Text>
            {menu}
            <div className={`pt-middle min-h-50 ${className}`}>
                {dataExists ? chart : <Empty />}
            </div>
        </Flex>
    );
}
```

Listing 2: Przykład tworzenia wykresów. Komponent generuje wykres *Chart.js* na podstawie przekazanych parametrów. Możliwe jest wybranie typu wykresu oraz rozwijanej listy z kategoriami.

```
const columns: TableColumnsType<DataType> = [
    {
        title: 'First name',
        dataIndex: 'firstName',
        key: 'firstName',
        fixed: 'left',
        sorter: (a, b) => a.firstName.localeCompare(b.firstName)
    },
    {
        title: 'Last name',
        dataIndex: 'lastName',
        key: 'lastName',
        sorter: (a, b) => a.lastName.localeCompare(b.lastName)
    },
    {
        title: 'Position',
        dataIndex: 'position',
        key: 'position',
        sorter: (a, b) => a.position.localeCompare(b.position),
        render: (position: string) => <Tag>{position}</Tag>
    },

    {
        key: 'detailsHref',
        dataIndex: 'detailsHref',
        fixed: 'right',
        width: 100,
        render: (href: string) => <Link to={href}>Details</Link>
    }
];

export function EmployeesTableCard({ employees = [], newEmployeeHref }: EmployeesTableCardProps) {
    return (
        <Card className='w-full'>
            <Flex vertical className='gap-layout'>
                <Flex justify='space-between'>
                    <CardTitle title='Employees' icon='employees' />
                    {newEmployeeHref && (
                        <ActionButton primary href={newEmployeeHref}>
                            + New
                        </ActionButton>
                    )}
                </Flex>
                <Table<DataType>
                    pagination={false}
                    columns={columns}
                    dataSource={employees.map((e, i) => ({ key: i, ...e }))}
                    scroll={{ x: 'max-content' }}
                />
            </Flex>
        </Card>
    );
}
```

Listing 3: Przykład generowania tabeli. Tworzenie tabeli w każdym przypadku odbywa się za pośrednictwem komponentu *Table* z *Ant Design*.

```
export function HallCreationCard({ hallTypes = [], onCreate = () => {} }: HallCreationCardProps) {
    const [form] = Form.useForm();

    return (
        <Card>
            <CardTitle title='Create Hall' icon='training-halls' />
            <Form
                form={form}
                layout='vertical'
                onFinish={onCreate}
                requiredMark={label => <span>{label}</span>}
                className='pt-small'
            >
                <Flex className='gap-layout'>
                    <Flex vertical className='w-full'>
                        <Form.Item
                            label='Hall Number'
                            name='hallNumber'
                            rules={[{ required: true, message: '' }]}
                        >
                            <Input placeholder='Enter hall number' />
                        </Form.Item>

                        <Form.Item
                            label='Type'
                            name='hallType'
                            rules={[{ required: true, message: '' }]}
                        >
                            <Select placeholder='Select hall type'>
                                {hallTypes.filter(Boolean).map(type => (
                                    <Select.Option key={type.name} value={type.uuid}>
                                        {type.name}
                                    </Select.Option>
                                ))}
                            </Select>
                        </Form.Item>
                    </Flex>

                    <Flex className='w-full' vertical>
                        <Form.Item
                            label='Description'
                            name='hallDescription'
                            rules={[{ required: true, message: '' }]}
```

```
                        >
                            <TextArea rows={5} placeholder='Enter description' allowClear />
                        </Form.Item>
                    </Flex>
                </Flex>

                <Flex justify='end'>
                    <Button type='primary' htmlType='submit'>
                        Create
                    </Button>
                </Flex>
            </Form>
        </Card>
    );
}
```

Listing 4: Przykład obsługi formularza. Wykorzystywany jest *hook useForm* z *Ant Design* który pozwala na łatwą walidację i przesyłanie wartości z formularza.

```
export function UserProvider({ children }: { children: JSX.Element }) {
    const [userDetails, setUserDetails] = useState<UserDetails>();

    const updateUser = (updates: Partial<UserDetails>) => {
        setUserDetails(prev => (prev ? { ...prev, ...updates } : prev));
    };

    useEffect(() => {
        async function initUser() {
            const initialized = await keycloak.init({ onLoad: 'login-required' });
            if (!initialized) return;

            initializeAxios(keycloak);

            const userProfile = await keycloak.loadUserProfile();
            const userRoles = keycloak.tokenParsed?.['roles'] as UserRole[];
            console.log(keycloak.tokenParsed);
            const significantRole = userRoles
                .sort((a: UserRole, b: UserRole) => rolesPriority[b] - rolesPriority[a])
                .at(0);

            const whoAmIData = await whoAmI()
                .then(data => data?.data)
                .catch(() => {
                    console.error('failed to fetch user data. Is backend API online?');
                    return undefined;
                });

            const userDetails: UserDetails = {
                firstName: userProfile.firstName as string,
                lastName: userProfile.lastName as string,
                email: userProfile.email as string,
                role: significantRole ?? 'client',
                id: whoAmIData?.uuid,
                hasValidMembership: dayjs().isBefore(dayjs(whoAmIData?.membership?.validUntil))
            };

            setUserDetails(userDetails);
        }

        if (import.meta.env.VITE_AUTH_ENABLED === 'true') {
            initUser();
        }
    }, []);

    return (
        <UserContext.Provider value={{ user: userDetails, updateUser }}>
            {children}
        </UserContext.Provider>
    );
}
```

Listing 5: Przykład globalnego stanu aplikacji. *UserProvider* dostarcza innym komponentom kontekst użytkownika pobrany z serwisu *Keycloak*.

```
export function useHTMLElementResizeObserver<T extends HTMLElement = HTMLDivElement>() {
    const ref = useRef<T>(null);
    const [width, setWidth] = useState(0);

    useEffect(() => {
        if (!ref.current) return;
        const observer = new ResizeObserver(entries => {
            for (const entry of entries) {
                const { width } = entry.contentRect;
                setWidth(width);
            }
        });

        observer.observe(ref.current);
        return () => observer.disconnect();
    }, []);

    return [ref, Math.floor(width)];
}
```

Listing 6: Przykład niestandardowego *hooka*. Służy do obserwacji szerokości elementu i jest wykorzsytany przy obsłudze responsywności.

```tsx
export function MembershipStatusCardWithData({ horizontal, showDetails }: MembershipStatusCardWithDataProps) {
    const { user } = useUser();
    const [membership, setMembership] = useState<Membership>();

    useEffect(() => {
        async function getMembershipStatus() {
            if (!user?.id) return;
            const userMembership = (await getUser(user.id)).data.membership;
            setMembership(userMembership);
        }

        getMembershipStatus();
    }, [user?.id]);

    return (
        <MembershipStatusCard
            detailsHref={showDetails ? `/${user?.role}/membership` : undefined}
            lastPayment={membership?.validFrom}
            validUntil={membership?.validUntil}
            horizontal={horizontal}
        />
    );
}
```

Listing 7: Przykład komponentu pobierającego dane z zewnętrznego API. Wykorzystywane są funkcje generowane przez *Orval* co eliminuje konieczność ręcznego korzystania z biblioteki *axios*.

```tsx
export function Router() {
    return (
        <BrowserRouter>
            <AuthGuard>
                <Routes>
                    <Route path='renew-membership' element={<ClientRenewMembershipPage />} />
                    <Route
                        path={rolesConfig['client'].routePrefix}
                        element={<AuthenticatedLayout renderChat={false} role='client' />}
                    >
                        <Route path='dashboard' element={<ClientDashboardPage />} />
                        <Route path='progress' element={<ClientProgressPage />} />
                        <Route path='sessions' element={<SessionsPage />} />
                        <Route path='membership' element={<ClientMembershipPage />} />
                        <Route path='workout' element={<ClientWorkoutPage />} />
                        <Route path='workout/:id' element={<WorkoutDetailsPage />} />
                    </Route>

                    <Route
                        path={rolesConfig['employee'].routePrefix}
                        element={<AuthenticatedLayout renderChat={false} role='employee' />}
                    >
                        <Route path='dashboard' element={<EmployeeDashboardPage />} />
                        <Route path='clients' element={<ClientsPage />} />
                        <Route path='training-halls' element={<HallsPage />} />
                        <Route path='create-membership' element={<MembershipCreationPage />} />
                        <Route path='clients/:id' element={<ClientDetailsPage />} />
                        <Route path='training-halls/:id' element={<EmployeeHallDetailsPage />} />
                    </Route>

                    <Route
                        path={rolesConfig['coach'].routePrefix}
                        element={<AuthenticatedLayout renderChat={false} role='coach' />}
                    >
                        <Route path='dashboard' element={<CoachDashboardPage />} />
                        <Route path='clients' element={<ClientsPage />} />
                        <Route path='sessions' element={<SessionsPage />} />
                        <Route path='workout/:id' element={<WorkoutDetailsPage />} />
                        <Route path='clients/details/:id' element={<CoachClientDetailsPage />} />
                        <Route path='new-session' element={<CoachNewSessionPage />} />
                    </Route>

                    <Route
                        path={rolesConfig['manager'].routePrefix}
                        element={<AuthenticatedLayout renderChat={false} role='manager' />}
                    >
                        <Route path='dashboard' element={<ManagerDashboardPage />} />
                        <Route path='create-membership' element={<MembershipCreationPage />} />
                        <Route path='clients' element={<ClientsPage />} />
                        <Route path='clients/:id' element={<ClientDetailsPage />} />
                        <Route path='employees' element={<ManagerEmployeesPage />} />
                        <Route path='create-employee' element={<ManagerEmployeeCreationPage />} />
                        <Route path='employees/:id' element={<ManagerEmployeeDetailsPage />} />
                        <Route path='client-details' element={<ClientDetailsPage />} />
                        <Route path='training-halls' element={<HallsPage />} />
                        <Route path='training-halls/details/:id' element={<div></div>} />
                        <Route path='create-hall' element={<ManagerHallCreationPage />} />
                    </Route>
                </Routes>
            </AuthGuard>
        </BrowserRouter>
    );
}
```

Listing 8: Komponent realizujący *routing* w aplikacji. Definiuje ścieżki i komponenty generujące poszczególne ekrany.

```tsx
export function AuthGuard({ children }: AuthGuardProps) {
    const { user } = useUser();
    const location = useLocation();
    const [loading, setLoading] = useState(false);

    useEffect(() => {
        setLoading(user === undefined);
    }, [user]);

    const isPublicPath = PUBLIC_PATHS.some(path => location.pathname.startsWith(path));

    if (import.meta.env.VITE_AUTH_ENABLED === 'false' || isPublicPath) {
        return children;
    }

    if (loading || !user) {
        return <Loader />;
    }

    const { routePrefix, defaultRoute } = rolesConfig[user?.role];
    const isPathAllowed = location.pathname.startsWith(routePrefix);

    if (!isPathAllowed) {
        return <Navigate to={routePrefix + defaultRoute} replace />;
    } else if (isPathAllowed && user.role === 'client' && !user.hasValidMembership) {
        return <Navigate to='/renew-membership' />;
    } else {
        return children;
    }
}
```

Listing 9: Komponent chroniący podstrony - zapewnia przekierowania i wymusza logowanie użytkownika.

3 Testy

3.1 Opis metod testowania (np. testy manualne i automatyczne)

W projekcie zastosowano kombinację testów manualnych i automatycznych w celu zapewnienia jakości oprogramowania. Testy automatyczne skupiają się na komponentach interfejsu użytkownika, sprawdzając ich zachowanie w różnych scenariuszach. Testy manualne wykorzystano do weryfikacji ogólnej funkcjonalności systemu oraz integracji między modułami.

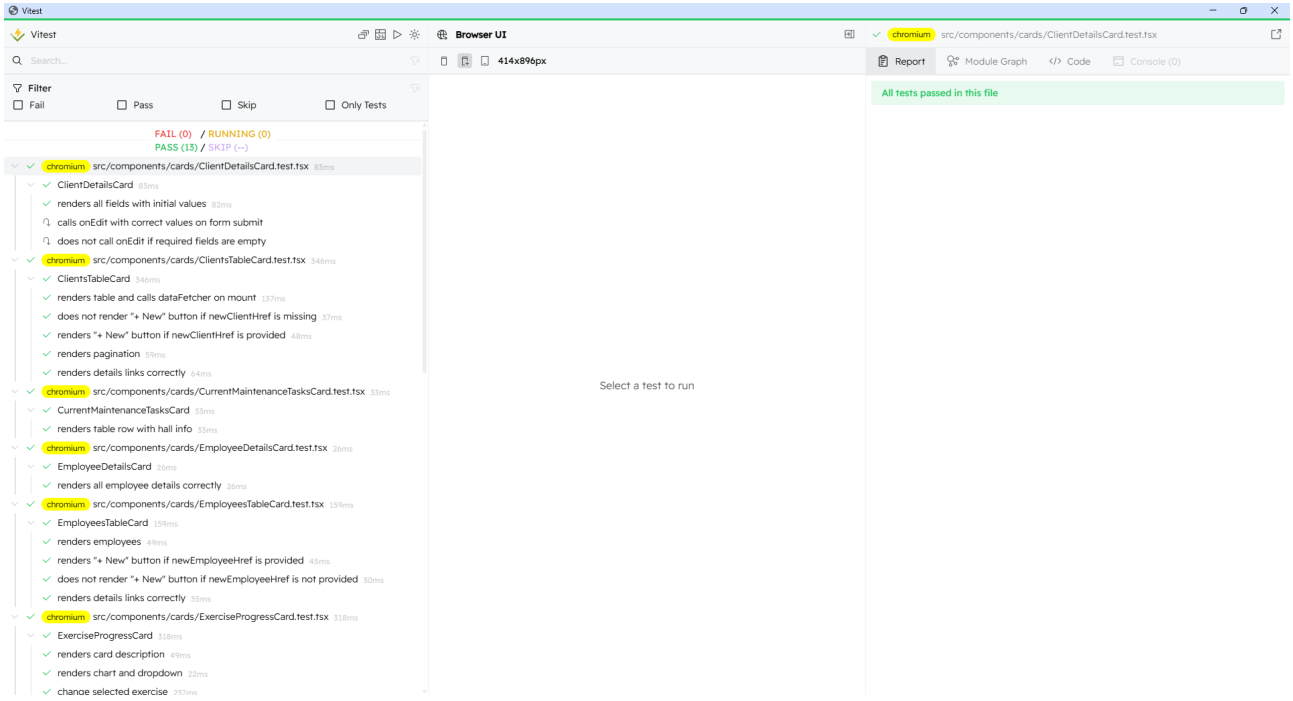
3.2 Wyniki testów, napotkane błędy oraz zastosowane rozwiązania

Przeprowadzone testy objęły wszystkie karty wykorzystywane w interfejsie użytkownika. Weryfikowano m.in. poprawność renderowania pól z określonymi wartościami, wywołania funkcji po wysłaniu formularza oraz widoczność elementów generowanych warunkowo.

Do testowania wykorzystano framework *Vitest* oraz tryb przeglądarkowy, umożliwiający podgląd na żywo przebiegu testów. Próby użycia trybu *headless* (bez interfejsu graficznego) skutkowały fałszywymi błędami, wynikającymi m.in. z braku interakcji z oknem (np. kliknięcia) lub problemów z mockowaniem natywnych funkcji przeglądarki, ponieważ testy w tym trybie uruchamiane są w środowisku *Node.js*. Wprowadzenie testowania w trybie przeglądarkowym wyeliminowało te problemy.

Podczas testów nie stwierdzono błędów w działaniu interfejsu – wszystkie kluczowe funkcjonalności działały zgodnie z oczekiwaniami. Jedyną napotkaną trudnością było połączenie interfejsu z API, co wymagało refaktoryzacji nazw atrybutów komponentów oraz ich dostosowania do przypadków, w których dane mogą nie zostać pobrane.

Znaczącym wyzwaniem okazała się również obsługa autoryzacji. Konieczne było utworzenie kontekstu użytkownika zintegrowanego z danymi pobieranymi z serwisu *Keycloak* oraz implementacja *interceptora* dla biblioteki *axios*, umożliwiającego prawidłowe odświeżanie sesji użytkownika. Dodatkowo potrzebny okazał się komponent odpowiedzialny za kontrolę dostępu do chronionych ścieżek i przekierowywanie niezalogowanych użytkowników. Ostatecznie proces logowania został pomyślnie wdrożony i działa poprawnie.



Rysunek 10: Wyniki testów interfejsu w środowisku *Vitest*.

4 Podsumowanie

4.1 Wnioski z realizacji projektu

Realizacja aplikacji do zarządzania siłownią pozwoliła zdobyć praktyczne doświadczenie w projektowaniu i implementacji złożonego interfejsu użytkownika z wykorzystaniem nowoczesnych rozwiązań frontendowych. W trakcie prac możliwe było zastosowanie w praktyce wiedzy dotyczącej architektury aplikacji typu SPA (*Single Page Application*), obsługi routingu, stanu aplikacji oraz integracji z zewnętrznymi usługami. Projekt umożliwił także lepsze zrozumienie zasad projektowania komponentów, tworzenia struktury kodu możliwej do wykorzystania w różnych częściach aplikacji oraz skutecznego testowania aplikacji — wcześniej wydawało się, że pisanie testów w aplikacjach *frontendowych* jest mało przydatne lub trudne do zrealizowania. Okazało się, że każdy z tych aspektów jest kluczowy dla prawidłowego działania aplikacji.

4.2 Ocena osiągniętych rezultatów i refleksje na temat procesu implementacji

Zrealizowana aplikacja spełnia założone wymagania funkcjonalne i techniczne. Interfejs użytkownika został zaprojektowany w sposób intuicyjny, a podział na role użytkowników (klient, pracownik, trener, menadżer) pozwala na klarowną separację uprawnień i dostępnych funkcji. Największym wyzwaniem okazała się integracja z backendem (spodziewaliśmy się, że pójdzie sprawniej), szczególnie podczas tworzenia i walidacji formularzy.

Pomimo relatywnie niewielkiej liczby funkcjonalności, projekt okazał się trudny i wyjątkowo czasochłonny. Każdy etap — od zaprojektowania interfejsu, przez jego wdrożenie i przetestowanie, aż po integrację z API — wymagał dużo zaangażowania i dokładności. Cały proces jest złożony i wymaga starannego planowania, co pokazuje, jak czasochłonne jest tworzenie nawet pozornie prostych aplikacji *frontendowych*.

4.3 Propozycje usprawnień lub dalszego rozwoju aplikacji

Aplikacja ma duży potencjał dalszego rozwoju. Do możliwych usprawnień należą:

- Wprowadzenie systemu powiadomień dla użytkowników (np. przypomnienia o wygasającym karnecie).
- Umożliwienie komunikacji pomiędzy trenerem a klientem (chat, wiadomości).
- Rozbudowa sekcji analitycznej dla menadżera, np. o raporty frekwencji czy obłożenia sal.
- Obsługa różnych rodzajów karnetów.
- Dodanie możliwości tworzenia i zarządzania kontami użytkowników bezpośrednio w aplikacji, zamiast manipulowania nimi w systemie *Keycloak*.
- Implementacja systemu ocen i opinii na temat trenerów i zajęć.

5 Podział pracy

- Filip Stępień – projekt graficzny aplikacji, przygotowanie i konfiguracja środowiska *developerskiego* oraz struktury projektu, implementacja komponentów, zapewnienie responsywności interfejsu, integracja z *backendem*, testy, dokumentacja.
- Rafał Grot – wdrożenie procesu automatyzacji (klient API), integracja z *backendem*, implementacja mechanizmów autoryzacji.
- Bartłomiej Karkoszka – implementacja komponentów, integracja z *backendem*, dokumentacja.

6 Literatura

- React – dokumentacja oficjalna, `https://react.dev/`, dostęp: 18.06.2025.
- Vite – dokumentacja oficjalna, `https://vite.dev/`, dostęp: 18.06.2025.
- Vitest – dokumentacja oficjalna, `https://vitest.dev/`, dostęp: 18.06.2025.
- React Router – dokumentacja oficjalna, `https://reactrouter.com/`, dostęp: 18.06.2025.
- Orval – dokumentacja oficjalna, `https://orval.dev`, dostęp: 18.06.2025.
- AntDesign – dokumentacja oficjalna, `https://ant.design/`, dostęp: 18.06.2025.
- Chart.js – dokumentacja oficjalna, `https://www.chartjs.org/`, dostęp: 18.06.2025.
- Keycloak JS – dokumentacja oficjalna, `https://www.keycloak.org/securing-apps/javascript-adapter`, dostęp: 18.06.2025.
- Tailwind CSS – dokumentacja oficjalna, `https://tailwindcss.com/`, dostęp: 18.06.2025.
- Day.js – dokumentacja oficjalna, `https://day.js.org/`, dostęp: 18.06.2025.
- Pawełkiwicz, Mateusz – wykłady z przedmiotu *Zaawansowane aplikacje frontendowe*, Politechnika Świętokrzyska, semestr letni 2025:
  - *Wprowadzenie do Frontend Developmentu*,
  - *Narzędzia i Frameworki*,
  - *Stylowanie i układ strony*,
  - *Zaawansowane techniki CSS*,
  - *Podstawy JavaScript*,
  - *Nowoczesny JavaScript, narzędzia i programowanie asynchroniczne*,
  - *Frameworki i Biblioteki Frontendowe – React, Angular, Vue.js*,
  - *Testowanie aplikacji frontendowych*.