Dokumentacja biblioteki UI JavaScript/React

Na bazie projektu CKTechnik

Autor: Miłosz Ziernik

Wersja: 1

Ostatnia modyfikacja: 26/09/2017

1. Architektura

Projekt stworzony jest w strukturze NPM bazującym na pliku `package.json`. Za proces renderowania odpowiada biblioteka ReactJS. Rolę buildera/runnera pełni Webpack. Całość zawiera w sobie część bazową (framework) oraz właściwą aplikację. Część bazową można wyodrębnić do oddzielnego projektu. Poszczególne katalogi odzwierciedlają moduły aplikacji:

- Biblioteka-core
  - Aplikacja – część bazowa odpowiedzialna za cały cykl życia aplikacji
  - Komponenty – zestaw gotowych do użycia komponentów React
  - Repozytoria – wszystko co związane z przetwarzaniem danych
  - WebApi – komunikacja z serwerem back-end
  - Strony – wbudowane strony – głownie na potrzeby deweloperskie
  - Utils – różne narzędzia
- Aplikacja
  - Model – repozytoria, konfiguracja, api
  - Widok – strony (formularze)

2. Wymagania

a. NodeJS – środowisko uruchomieniowe - https://nodejs.org/en/download/
b. IDE – dowolne wspierające NodeJS – WebStorm (zalecany), VisualStudio Code, Atom itp.

** w środowisku programistycznym należy włączyć wsparcie dla FLOW (w przypadku WebStorm: File -> Settings -> Languages & Frameworks -> JavaScript -> Javascript language version: Flow*

3. Uruchomienie projektu

- Instalujemy NodeJS - https://nodejs.org/en/download/
- pobieramy pliki źródłowe z repozytorium SVN (svn://10.0.0.2/CKTechnik/UI)
- w katalogu projektu wykonujemy komendy:
  - `npm install` – instalacja wszystkich niezbędnych modułów (zależności)
  - `npm start` – kompilacja i uruchomienie wbudowanego serwera http

Tryby pracy (profile) aplikacji

- Deweloperski – aktywne opcje ułatwiające debugowanie aplikacji,
- Testowy – wyłączone niektóre mechanizmy deweloperskie (np. weryfikacje typów), bundle.js w postaci zminimalizowanej
- Demonstracyjny – podobny do testowego, komunikaty błędów uogólnione, dane zbliżone do produkcyjnych
- Produkcyjny – kod zoptymalizowany, wyłączone wszystkie funkcje weryfikujące typy, opcjonalne zaciemnienie kodu (uglify)

Skrypty dostępne w pliku `package.json`:

- "`start`" – uruchomienie serwera deweloperskiego
- "`build DEV`" – zbudowanie aplikacji w trybie deweloperskim
- "`build TEST`" – zbudowanie aplikacji w trybie testowym
- "`build DEMO`" – zbudowanie aplikacji w trybie demonstracyjnym
- "`build PROD`" – zbudowanie aplikacji w trybie produkcyjnym
- "`flow`" – uruchomienie weryfikacji poprawności typów przy użyciu biblioteki flow (`npm install --global flow-bin`)
- "`CKTechnik Sync`" – synchronizacja plików z projektem CKTechnik
- "`doc`": wygenerowanie dokumentacji kodu przy użyciu biblioteki esdoc (`npm install esdoc`) *Wymaga konfiguracji
- "`devtool`" – uruchomienie narzędzia deweloperskiego (chromium) umożliwiającego debugowanie środowiska npm (`npm install devtool -g`)

Uruchomienie: `npm run [nazwa skryptu]`

4. Wydawanie wersji testowej/produkcyjnej

Aby wydać wersję aplikacji należy wykonać jeden ze skryptów opisanych wcześniej, np. „test” `npm run "build TEST"`. Proces budowania potrwa ok 2 minuty. W jego wyniku powstanie w katalogu `public` plik `bundle.js` zawierającą kod całej aplikacji oraz `bundle.js.map` zawierający mapowania kodu źródłowego i wynikowego. Należy zwrócić uwagę aby nie plik `bundle.js.map` nie znalazł się w wersji produkcyjnej (względy bezpieczeństwa). Można również włączyć zaciemnianie kodu konfigurując odpowiednie flagi w pliku `webpack.config.js`. Włączenie zaciemniania utrudni jednak znacznie diagnozowanie błędów.

Oprócz wspomnianych wcześniej plików utworzony zostanie katalog `assets` zawierający pliki projektu będące zasobami (pliki nie dołączone do bundle.js).

W momencie buildowania automatycznie zostanie „podbita” wersja aplikacji oraz data buildu w pliku `package.json` za pomocą modułu webpack-auto-inject-version

5. Wsparcie przeglądarek:

a. Chrome: >= 58
b. Firefox: >= 53
c. Internet Explorer – brak wsparcia (teoretycznie możliwe jest dostosowanie wersji aplikacji do współpracy z I.E., jednak wymagałoby to dodatkowego nakładu pracy)
d. Edge: >= 38
e. Android: tak (wymaga testów)
f. Inne bazujące na silniku chromium, np. Opera, Maxthon

6. Konfiguracja Webpack-a, dodawanie pliku zasobu

Procesem budowania aplikacji odpowiada narzędzie webpack (obecnie 2.3.3). Konfiguracja zawarta jest w pliku `webpack.config.js` i bazuje na wzorcu ze strony producenta. Zastosowane moduły/funkcjonalności

- Babel – transpilacja plików jsx (i nie tylko) do standardu ECMA2015
- OpenBrowser – automatyczne otwieranie domyślnej przeglądarki w trybie deweloperskim
- StringReplace – dodanie informacji o modułach w powiązaniu z klasą Bootstrap.js
- WebpackAutoInject – automatyczne wersjonowanie projektu

Dodane jest wsparcie dla `jQuery`. Aby dodać plik zasobu należy jego ścieżkę (lub katalog nadrzędny) dodać do listy w zmiennej `nodeModulesWhiteList`.

7. Ogólne założenia funkcjonalne

- Aplikacja funkcjonuje w modelu Single Page Application. Całość aplikacji (skrypty i szablony) wczytywana jest w momencie uruchomienia. Możliwe jest cachowanie za pomocą pliku manifestu HTML lub nagłówków protokołu HTTP.
- Kod aplikacji nie powinien być generowany przez serwer i wczytywany np. za pośrednictwem mechanizmu ajax, jak to się odbywa w klasycznych rozwiązaniach.
- Protokół komunikacji z serwerem: WebSocket (biblioteka SignalR).
- Format wymiany danych: JSON.
- (Możliwie) cała logika biznesowa i kontrola spójności danych leży po stronie backend-u
- Dane przechowywane są w formie repozytoriów.
- W każdym repozytorium musi być wskazanie na kolumnę zawierającą klucz głowny
- Po uruchomieniu aplikacji aktualizowana jest lista repozytoriów oraz ich meta dane.
- Po zaktualizowaniu listy wczytywane są dane (rekordy)
- Serwer zwraca wszystkie dane niezbędne do działania aplikacji
- Serwer decyduje czy zwrócić dane (uprawnienia CRUD) oraz w jaki zakres wierszy ma być zwrócony (w niektórych przypadkach można o ograniczyć ilość zwracanych danych stosując stronicowanie)
- Aktualizacja danych opiera się na wbudowanym mechanizmie edycji rekordów.
- Edycja bazuje na kopii danych – dane źródłowe nigdy nie są bezpośrednio modyfikowane.
- Modyfikacja danych odbywa się w systemie transakcyjnym.
- Zatwierdzenie transakcji skutkuje wysłaniem żądania do serwera zawierającego dane zmodyfikowanych rekordów.
- W odpowiedzi na żądanie serwer przetwarza dane, zapisuje w bazie i rozgłasza dokonane zmiany dla wszystkich klientów lub zwraca komunikat błędu w przypadku niepowodzenia operacji.
- Serwer rozsyła informacje o wszelkich zmianach do wszystkich aktywnych (podłączonych) klientów. Aplikacje klienckie automatycznie aktualizują stan repozytoriów.

8. Core aplikacji, cykl życia

Uruchamianie aplikacji rozpoczyna się od wykonania pliku Index.js. Następuje ładowanie importowanych zależności. Należy je deklarować w taki sposób, aby na początku wczytywały się moduły nie zawierające zależności – czyli zaczynamy od klas typu utils a kończymy na komponentach i stronach.

Główne klasy:

- Application – główna klasa –
  - scala wszystkie gałęzie,
  - renderuje elementy react najwyższego poziomu,
  - odpowiada za generowanie zdarzeń zmiany adresu URL (nawigacji)
- AppNode – gałąź aplikacji, zarządza pojedynczą niezależną sekcją na stronie (np. pasek narzędzi, pasek nawigacyjny, kontener stron, pasek statusu)
- Endpoint – definicja punktu nawigacji,
  - bezpośrednio powiązane z procesem routingu,
  - definiuje atrybuty strony takie jak: klucz, nazwa, ścieżka, ikona i inne
  - odpowiada za proces nawigacji do danej strony
- API – interfejs zawierający zbiór domyślnych metod obsługi WebApi. Aby móc z niej korzystać należy wywołać metodę set() podając jako argument handler obsługujący metody api
- Event – obsługa zdarzeń aplikacji
  - AppEvent – instancja obiektu zdarzenia – zawiera wszystkie dane dotyczące konkretnego zdarzenia
  - EventType – definicja zdarzenia, np. („navigate”, „resize”)
- ContextObject - umożliwia zarządzanie cyklem życia obiektów powiązanych z danym kontekstem. Jeśli dany kontekst (np. strona) zostaje zniszczony, to automatycznie zostaje również wywołana funkcja zwrotna i wykonywane są zadeklarowane operacje (np. wyrejestrowanie obserwatorów, czyszczenie danych)
- Skin – obsługa „skórek” – klasa zawiera definicje selektorów css, które użytkownik sam może modyfikować i widzieć rezultat w czasie rzeczywistym. Modyfikacja skutkuje ustawianiem zmiennych w arkuszach styli (nie obsługiwane przez I.E. i Edge)
- Status – klasa pełniąca rolę interfejsu do obsługi komunikatów niskiego priorytetu. Interfejs ten implementowany jest domyślnie przez klasę StatusHint, która wyświetla statusy w formie paneli wysuwanych z prawej górnej krawędzi ekranu. Dostępne są następujące typy statusów: debug, info, warning, error. Każdy status wyświetlany jest przez określony czas po upływie którego znika.
- UserData – zawiera podstawowe dane na temat zalogowanego użytkownika: identyfikator, imię, nazwisko, nazwę wyświetlaną, e-mail. Dane te wykorzystywane są do filtrów związanych z użytkownikiem (np. „Moje zadania”) oraz do zarządzania procesem logowania/wylogowania użytkownika.

9. Nawigacja

Po załadowaniu strony lub po zmianie adresu URL następuje proces nawigacji składający się z kolejnych kroków:

- Klasa `Application` generuje zdarzenie `AppEvent.APPLICATION__HASH_CHANGE` lub `AppEvent.APPLICATION__LOCATION_CHANGE` w zależności od tego czy zmieniła się znacząca część adresu czy tylko hasz.
- Zdarzenie APPLICATION__LOCATION_CHANGE nasłuchiwane jest przez `PageContainer`, który w momencie zmiany adresu URL wyrenderuje docelową stronę
- O tym który komponent uznany zostanie za bieżącą stronę decyduje proces routingu realizowany przez bibliotekę react-router
- Router analizuje kolejno zadeklarowane endpointy i próbuje dopasować bieżący adres URL do maski endpointu
- Po dopasowaniu renderowany jest komponent (strona) powiązany z danym endpointem lub strona błędu jeśli nie dopasowano żadnego adresu

Należy zwrócić uwagę, że w standardzie SPA (Single Page Application) zmiana adresu URL nie powoduje wysłania żądania http. Taka operacja możliwa jest dzięki standardowi HTML5 oraz interfejsowi obsługi historii przeglądania. Nawigacja technicznie jedynie dodaje dany adres URL do stosu wywołań i generuje stosowne zdarzenie systemowe.

Aby proces ten działał zawsze prawidłowo wymagane jest skonfigurowanie serwera http w taki sposób aby w momencie odebrania żądania dotyczącego nieistniejącego zasobu (pliku) zawsze zwracał plik indeksu (index.html) zamiast strony błędu 404.

10. WebApi

Mechanizm komunikacji z serwerem backend bazuje na bibliotece SignalR, protokołem transportowym jest WebSocket.

Klasy:

- `WebApi` – główna klasa odpowiedzialna za przetwarzanie żądań i odpowiedzi serwera
- `WebApiRequest` – Klasa odpowiedzialna za zbudowanie struktury danych żądania
- `WebApiResponse` – Odpowiada za przetworzenie odpowiedzi, sparowanie z żądaniem (`WebApiRequest`)
- `WebApiMessage` – obsługa komunikatów (alertów) wysyłanych z serwera (obecnie nie wykorzystywane)
- `Transport` – odpowiada za obsługę protokołu transportowego, zarządzanie połączeniem

Metody API:

- `authorizeUser` – autoryzacja użytkownika na podstawie loginu i hasła, zwraca dane zalogowanego użytkownika
- `repoList` – zwraca listę repozytoriów (meta dane)
- `repoGet` – zwraca dane repozytoriów (wiersze)
- `repoEdit` – wykonanie akcji CRUD na repozytorium
- `repoAction` – wykonanie predefiniowanej akcji na repozytorium
- `recordCallback` – wykonanie funkcji zwrotnej w momencie tworzenia lub edycji repozytorium
- `downloadFile` – inicjalizacja mechanizmu pobierania pliku. Serwer przetwarza żądanie pobrania pliku i zwraca identyfikator, nazwę, rozmiar i adres URL. Następnie aplikacja wykona nawigację do wskazanego adresu URL i pobierze plik.
- `uploadFile` – inicjalizacja mechanizmu wysyłania pliku. Działa analogicznie jak mechanizm pobierania
- `getConfiguration` – zwraca konfigurację serwera, informacje o buildzie, wymagane adresy itp

11. Repozytoria

Repozytoria pełnią rolę magazynu danych. Zawierają zbiór meta danych opisujących strukturę danych, dzięki czemu możliwe jest:

- wyświetlenie danej (formatowanie do odpowiedniej postaci)
- automatyczne dobór kontrolki odpowiedzialnej za podgląd oraz edycję danej
- parsowanie i serializacja – konwersja z i do formatu akceptowanego przez backend
- walidacja poprawności danych:
    - Wymagalność
    - Unikalność
    - Minimalna/maksymalna wartość/długość tekstu
    - Zgodność z wyrażeniem regularnym
- Tworzenie zależności z innymi repozytoriami za pomocą kluczy obcych oraz referencji
- Definiowanie akcji na poziomie repozytorium lub rekordu wykonywanych po stronie backend-u,
- Definiowanie uprawnień CRUDE
- Określanie maski wyświetlania danego rekordu

Elementy (klasy) repozytorium:

- `Repository` – główna klasa przechowująca strukturę danego repozytorium. Odpowiednik tabeli w strukturze baz danych. Każda klasa ma jedną instancję (singleton).
- `RepoConfig` – konfiguracja repozytorium
- `Record` – klasa obsługująca dany wiersz repozytorium (analogia do rekordu bazy danych). Klasa tworzona jest automatycznie w momencie dokonywania operacji na rekordzie, operuje na kopii danych, niszczona jest w momencie zniszczenia komponentu, który ją powołał.
- `Column` – przechowuje konfigurację pól rekordu lub kolumn (w odniesieniu do tabeli bazy danych)
- `Field/Cell` – reprezentuje pojedynczą komórkę zawierającą daną wartość o typie zdefiniowanym w klasie `Column`
- `DataType` – definicje typów danych oraz mechanizmy ich obsługi
- `RepoCursor` – kursor iterujący się po repozytorium. Używany głownie do filtrowania/wyszukiwania danych. Iteracja nie tworzy klasy `Record` dla każdego rekordu, pobieranie danych odbywa się przez funkcję `getValue`, w której argumencie definiujemy interesującą nas kolumnę. Ma to na celu zwiększenie wydajności algorytmu dla dużej ilości danych.
- `RepoFlag` – klasa obsługująca niektóre atrybuty atrybuty (np. hidden lub akcje) uzależnione od miejsca gdzie są wyświetlane (widok edycji rekordu, lista rekordów)
    - L - ukrycie na liście
    - H - wyłączenie na liście
    - C - ukrycie w formularzu dodawania
    - U - ukrycie w formularzu edycji
    - A - wyłączenie wszędzie
- `Action` – obsługa akcji repozytorium lub rekordu definiowane po stronie backend-u
- `Foreign` – mechanizm zarządzania relacjami danego repozytorium z innymi za pomocą kluczy obcych lub referencji
- `RepoTree` – konwersja płaskiej struktury repozytorium do struktury drzewiastej bazującej na polu `parentColumn` konfiguracji repozytorium

- `RepositoryStorage` – obsługa magazynu danych repozytorium – wczytywanie i zapisywanie danych. Implementacje:
    - `WebApiRepoStorage` – dane wczytywane i zapisywane są za pośrednictwem modułu `WebApi` (komunikacja z backendem)
    - `BrowserRepoStorage` – dane przechowywane w lokalnym magazynie przeglądarki (local storage)
    - `NullRepoStorage` – dane nie są nigdzie zapisywane

12. Komponenty

Komponenty aplikacji stworzone są w formie komponentów ReactJS zapisanym w strukturze JSX. Klasą bazową dla wszystkich komponentów jest `Component` (która rozszerza `ReactComponent`).Klasa ta zawiera szereg funkcjonalności umożliwiających integrację komponentów z resztą aplikacji. Komponenty odpowiedzialne za obsługę typów danych są ściśle związane z klasą Field.

Lista komponentów:

- `FCtrl` – jeden z najważniejszych komponentów, który automatycznie wybiera docelowy komponent na podstawie pola Field
- `Alert` – wyświetlanie komunikatów (okien modalnych, popup) przy pomocy biblioteki sweetalert2
- `Bootstrap` – zestaw styli odpowiedzialnych za responsywne wyświetlanie treści
- `CodeMirror` – wyświetlanie i edycja różnych typów plików tekstowych
- `Tree` – drzewo, pasek nawigacyjny
- `Icon` – zbiór ikon w postaci czcionek FontAwesome, Fontello i ionicons
- `Breadcrumbs` – komponent wyświetlający ścieżkę formatek w panelu górnym
- `Busy` – wyświetla animowany wskaźnik zajętości wewnątrz komponentu
- `Spinner` – podobnie jak `Busy`, z tym że wyświetlane jest na środku ekranu wraz z dodatkową warstwą blokującą kliknięcia
- Button – przycisk, styl bootstrap-a, przyjmuje jeden z typów: "`basic`", "`default`", "`primary`", "`success`", "`info`", "`warning`", "`danger`", "`link`"
- `Btn` – kontroler przycisku Button
- `IconEdit` – pole edycyjne z ikoną z lewej strony – występuje np. w oknie logowania
- `JsonViewer` – podgląd struktury JSON
- `Link` – element „a” html rozszerzony o dodatkowe opcje
- `PopupMenu` – menu kontekstowe
- `TabSet` – komponent listy zakładek
- `Hint` – hint używany np. w FCtrl
- `Input` – odpowiednik input-a html
- `Memo` – odpowiednik text-area html
- `Select` – ComboBox, lista rozwijalna, odpowiednik select-a html
- `CheckBox` – dwu lub trójstanowa kontrola boolean w formie zaznaczanego pola

- `Toogle` – podobnie jak `CheckBox`, z tym że w formie slidera
- `RadioButton` – wyświetlanie elementów enumeraty w formie przycisków typu radio
- `List` – umożliwia dodawanie i usuwanie elementów zdefiniowanych w obiekcie Field
- `Multiple` – tworzy komponent złożony z kilku innych (np. wartość + jednostka)
- `DatePicker` – wybór daty i/lub czasu
- `ImageViewer` – podgląd i edycja pliku graficznego (podmiana)
- `Attributes` – zbiór atrybutów (obiektów typu Field) rysowanych w formie tabelarycznej
- `Panel` – komponent wizualny grupujący kontrolki, ma możliwość zmiany rozmiaru, przewijana, zwijania
- `Resizer` – trójkącik rysowany w prawym dolnym rogu komponentu, umożliwia zmianę jego rozmiaru
- `Scrollbar` – pełni role suwaka, pojawia się i znika automatycznie gdy kursor znajdzie się w obszarze kontrolowanego komponentu
- `Splitter` – umożliwia zmianę rozmiaru komponentów potomnych
- `Modal` – okno modalne umożliwiające wyświetlenie dowolnej treści – również stron/formatek
- `Dragger` – umożliwia przeciąganie elementów metodą drag and drop
- `AttributesRecord` – rysuje automatycznie zawartość rekordu w formie komponentu `Attributes`
- `DTO` – rozszerzenie `JsonViewer`, automatycznie wyświetla DTO dla rekordu
- `Table` – tabela bazująca na komponencie `ReactTable`
- `RepoTable` – rozszerzenie `Table` automatycznie rysujące zawartość repozytorium
- `RecordCtrl` – kontroler obsługujący operacje na rekordzie, rysuje przyciski „Zapisz”, „Usuń”, tworzy okno modalne edycji rekordu
- `RepoCtrl` – kontroler repozytorium - wyświetla tabelę `RepoTable` z listą rekordów, rysuje przycisk „Dodaj”
- `ReferenceRecordCtrl` – umożliwia edycję repozytoriów powiązanych przez referencje, ma postać komponentu List

13. Przykład strony:

Poniższy przykład zawiera:

- Rozszerzenie klasy `Page`
- Deklaracja endpoint-u
- Nawigacji do strony głównej
- Użycie komponentu
    - Deklarację właściwości wymaganej i opcjonalnej (`propTypes`)
    - Deklarację domyślnej wartości właściwości (`defaultProps`)
    - Zmianę stanu (`state`)
    - Przepisanie właściwości do stanu
    - Deklarację styli śród liniowych
    - Obsługę zdarzeń (`click, mouse enter, mouse leave`)
    - Dostęp do gałęzi drzewa DOM
    - Integrację z zewnętrzną biblioteką na przykładzie jQuery i funkcji hover

Aby strona zadziałała musimy ją dodać do importów, np. w pliku Index.js : `import "./page/PTest";`
Utworzone zostaną deklaracje dwóch klas oraz instancja obiektu Endpoint, który mapuje adres URL „/test” z komponentem `PTest`. Etykieta „Strona testowa” automatycznie pojawi się w pasku nawigacyjnym. Wysołując adres http://localhost:8080/test zobaczymy naszą stronę, na której widoczny będzie czarny prostokąt z zielonym tekstem oraz przycisk przenoszący do strony głównej. Po najechaniu kursorem myszy na prostokąt tekst się powiększy (`zoom: 1.4`) oraz pojawi się efekt zaniknięcia i ponownego wyświetlenia (`$.hover()`). Po kliknięciu wylosowany zostanie kolor i przekazany do komponentu `Label` jako stan.

```
import {React, PropTypes, Endpoint} from "../core/core";
import {Component, Page, Panel} from "../core/components";

class PTest extends Page {

    render() {
        this.title.set("Przykładowa strona");
        return <Panel fit>
            <Label color="lime" info="Po kliknięciu pojawi się na chwilę podkreślenie i pogrubienie">
                <div>Przykładowa etykieta</div>
            </Label>;
            <div>
                <button onClick={(e: Event) => Endpoint.navigate("/", e)}>Przejdź do strony
głównej</button>
            </div>
        </Panel>
    }

}

class Label extends Component {

    state = {
        color: null
    };

    static propTypes = {
        color: PropTypes.string.isRequired,
        info: PropTypes.string
```

```
    };

    static defaultProps = {
        color: "yellow"
    };

    constructor() {
        super(...arguments);
        this.state.color = this.props.color;
    }

    render() {
        return <div
            ref={(tag: HTMLDivElement) => {
                if (tag)
                    $(tag).hover(function () {
                        $(this).fadeOut(1000);
                        $(this).fadeIn(1000);
                    })
            }
            }
            style={{
                color: this.state.color,
                cursor: "pointer",
                backgroundColor: "black",
                padding: "30px"
            }}
            onClick={(e: Event) => {
                let color = Math.floor(Math.random() * 16777216).toString(16);
                color = '#000000'.slice(0, -color.length) + color;
                this.setState({
                    color: color
                });
            }}
            onMouseEnter={(e: Event) => e.currentTarget.style.zoom = "1.4"}
            onMouseLeave={(e: Event) => e.currentTarget.style.zoom = null}
        >
            {this.props.children}
            <div style={{fontSize: "0.7em"}}>{this.props.info}</div>
        </div>
    }
}

new Endpoint("test", "Strona testowa", "/test", PTest);
```

14. Operacje na repozytoriach

Poniższy kod demonstruje w jaki sposób odwołać się do rekordu, wyświetlić jego wybrane pola oraz zapisać zmiany na przykładzie rekordu danych użytkownika. Klasa `PTest` rozszerza `RepoPage` – klasa ta kontroluje gotowość repozytorium. Jeśli dane nie zostały jeszcze wczytane, wyświetlony zostanie komponent oczekiwania typu spinner. Po wczytaniu danych strona zostanie automatycznie odświeżona.

Z repozytorium użytkowników (`R_USER`) odwołujemy się do interesującego nas rekordu na podstawie klucza głównego – w tym przypadku w postaci GUID-a). metoda R_USER.get() zwróci obiekt `EUser` typu `Record`, które zawiera wszystkie interesujące nas dane. Deklarujemy kolejno komponenty `FCtrl` podając w parametrze jakie pole rekordu użytkownika ma być wyświetlone oraz w jakiej postaci (np. nazwa, wartość, tylko do odczytu itp.). Utworzony został obiekt `RecordCtrl` automatyzujący operacje na rekordzie. Widoczne są również 3 przyciski umożliwiające zapis danych – każdy w odmienny sposób. Ostatni przycisk („metoda 3”) demonstruje zapis z pominięciem kontrolera.

```
import {React, Endpoint, CRUDE, Repository} from "../core/core";
import {Panel, FCtrl} from "../core/components";
import {EUser, R_USER} from "../model/Repositories";
import RepoPage from "../core/page/base/RepoPage";
import RecordCtrl from "../core/component/repository/RecordCtrl";

class PTest extends RepoPage {

    constructor() {
        super(R_USER, ...arguments);
    }

    render() {
        const user: EUser = R_USER.get(this, "044a0652-3012-41b5-8a79-260d9b313d9f");
        const controller: RecordCtrl = new RecordCtrl(user);

        this.title.set("Edycja użytkownika " + user.displayValue);
        return <Panel fit>
            <ul>
                <li>Login (tylko do odczytu): <FCtrl field={user.LOGIN}/></li>
                <li>Imię (edycja): <FCtrl value field={user.FIRST_NAME}/></li>
                <li>Nazwisko (edycja): <FCtrl value field={user.LAST_NAME}/></li>
            </ul>
            <div>
                {controller.btnSave.$}
                {controller.btnDelete.$}
                <button onClick={e => controller.commit(CRUDE.UPDATE, () => alert("Zapis pomyślny"))}>
                    Zapisz - metoda 2
                </button>
                <button
                    onClick={e => Repository.commit(this, [user])
                        .then(data => alert("Zapis pomyślny"))
                        .catch(e => alert("Błąd: " + e))
                    }>
                    Zapisz - metoda 3
                </button>
                <button onClick={e => controller.modalEdit()}>Edytuj w oknie modalnym</button>
            </div>
        </Panel>
    }
}

new Endpoint("test", "Strona testowa", "/test", PTest);
```

Przykład obiektu DTO wysyłanego do serwera


```
{
	#action:	"update",
	#uid:	"mmRr0dZtAI",
	id:	"044a0652-3012-41b5-8a79-260d9b313d9f",
	login:	"akowalska",
	password:	"QEthCg4E",
	firstName:	"Anna",
	lastName:	"Kowalska",
	active:	true,
	archive:	false,
	email:	"anna.kowalska@email.pl",
	mobile:	null,
	type: 0,
	picture:	{
		id:	"d3151d9c-1f2c-42d2-8d7a-267f27d487e6",
		name: "zdjecie.png",
		size: 47033
	},
	bussinessId: null
}
```