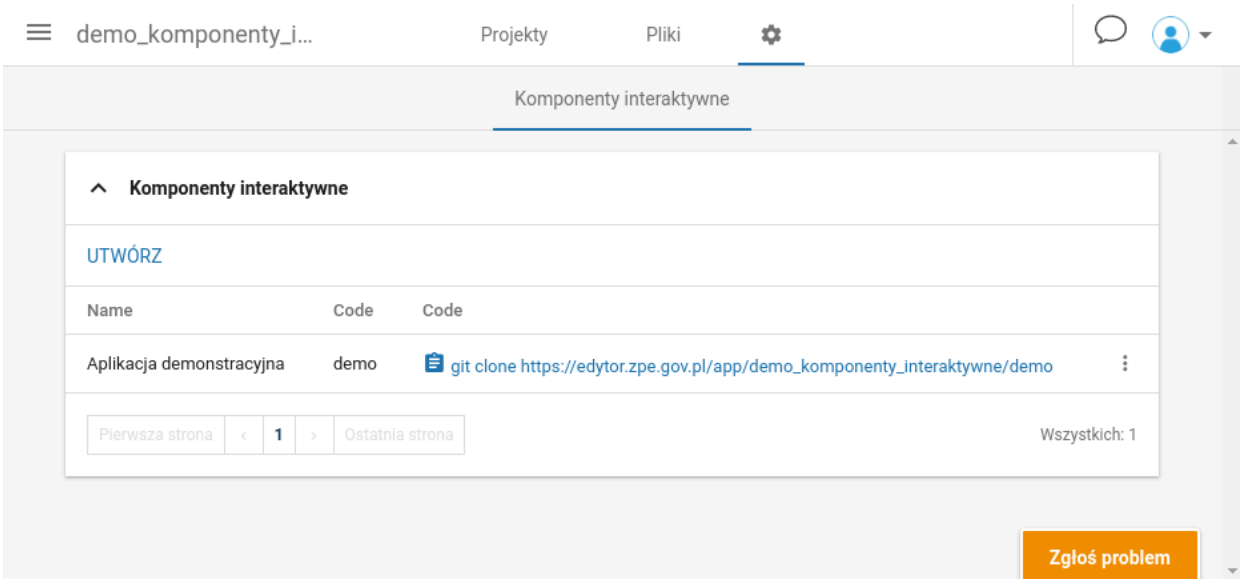
Dokumentacja techniczna komponentów interaktywnych

Data aktualizacji	03.02.2026
Status	Aktualny
Autor	CIE

Wprowadzenie

Dokumentacja opisuje techniczne zagadnienia związane z tworzeniem własnych komponentów interaktywnych.

Przed utworzeniem własnego komponentu należy zarezerwować nazwę w systemie. Nazwa składa się z dwóch członów oddzielonych znakiem slash: nazwa przestrzeni/kod silnika (np. core/geogebra).



Po zarezerwowaniu nazwy, system utworzy repozytorium GIT, do którego należy wgrać zminifikowane źródła aplikacji oraz inne potrzebne pliki (obrazy, CSS które są używane zawsze).

Pliki, które dotyczą konkretnej instancji ćwiczenia, powinny być dostarczone z danymi.

Wymagania techniczne

Kod silnika musi być zgodny z ECMAScript5. Wszystkie pliki tekstowe muszą być zakodowane w UTF-8 bez sygnatury BOM.

Jeśli przy tworzeniu komponentu, używane są nowoczesne struktury składniowe, kod **musi** zostać skompilowany do składni ECMAScript5 (np. przy użyciu kompilatora babeljs).

Obiekty interaktywne muszą zapewniać responsywność i kompatybilność z:

- Firefox (ostatnia wersja -1)
- Chrome (ostatnia wersja -1)
- Opera (ostatnia wersja -1)
- Microsoft Edge 94
- Safari 17+
- iOS Safari 17+

Architektura silnika

Silnik jest aplikacją zgodną z interfejsem platformy. Wszystkie pliki silnika umieszczone są w repozytorium GIT utworzonym przez system.

Na silnik składają się:

- engine.json **(wymagany)**
- punkt wejścia **(wymagany)**, jego położenie jest określone w engine.json
- punkt wejścia edytora *(opcjonalny)*, jego położenie jest określone w engine.json
- inne pliki statyczne *(opcjonalnie)*

Aplikacja może odnosić się **tylko** do plików znajdujących się w repozytorium (używając metody enginePath) lub w konkretnej instacji (używając metody dataPath).

Aby uruchomić silnik, należy utworzyć instancję.

Silnik powinien zawierać tylko kluczowe pliki konieczne do uruchomienia widżetu. Zbyt duży silnik może utrudnić pobieranie materiałów do trybu offline.

Definicja silnika - engine.json

Plik engine.json definiuje jakie funkcje dostarcza oraz jakich funkcji wymaga komponent.

Włączenie niektórych opcji, może nieść za sobą konieczność implementacji dodatkowych metod.

Minimalna (wymagana) zawartość pliku:

```
{
    "entry": "entry.js"
}
```

Wszystkie opcje

```
{
    "entry": "entry.js",
    "stateful": true | false,
    "printable": true | false,
    "validation": "auto" | "manual" | "none",
    "useWebGL": true | false,
    "editor": {
    "entry": "editorEntry.js",
        "defaultData": {},
        "demoData": {}
    },
    "collaboration": true | false,
    "awards": [
        {
            "code": "smog",
            "name": "Ukończyłeś mój kurs!",
            "description: "Gratulacje, teraz już wiesz, co to jest smog.",
            "icon": "image/icon1.png"
        }
    ]
}
```

Funkcje poszczególnych pól:

- **entry**: określa plik wejścia.
- **data**: dane, które przekazywane są potem w metodzie init.
- **stateful**: określa czy aplikacja zapisuje stan.

  Wymaga implementacji dodatkowych metod. Szczegóły implementacyjne zostały opisane w sekcji stateful.
- **printable**: aplikację można drukować (wraz z nim dostarczone są style CSS do druku).
- **useWebGL**: deklaracja czy komponent używa WebGL.
- **validation**: określa czy ćwiczenie może podlegać ocenie.

  Możliwe wartości:

  - auto - komponent sam sprawdza poprawność wykonania.
    Wymaga implementacji interfejsu validation.
  - manual - poprawność wykonania sprawdzana jest manualnie, przez użytkownika systemu.
    Wymaga implementacji interfejsu stateful.
  - none (*domyślnie*) - poprawność wykonania nie podlega ocenie.
- **editor**: określa parametry umożliwiające utworzenie edytora komponentu.
  - **entry**: określa plik wejścia dla modułu edytora.

    - **defaultData**: domyślne dane, przy tworzeniu nowego komponentu.
    - **demoData**: dane wykorzystywane przy użyciu opcji “Wczytaj demo”.
- **collaboration**: komponent wspiera tryb współpracy.
- **awards**: lista nagród możliwa do nadania przez ćwiczenie. Nadanie nagrody odbywa się poprzez wywołanie `api.grantAward('code')`.

Punkt wejścia (entry)

Dostarczany jest w formie asynchronicznego modułu AMD lub kompatybilnego (np. UMD).

Moduł musi eksportować fabrykę (funkcję, która utworzy silnik) lub konstruktor.

Przykłady eksportu fabryki (moduł AMD):

```
define([], function () {
    return function () {
        return {
            init: function () {
            },
            destroy: function () {
            }
        }
    }
})
```

Przykłady eksportu konstruktora (moduł AMD):

```
define([], function () {
    function Engine() {
    }

    Engine.prototype.init = function () {
    }
    Engine.prototype.destroy = function () {
    }

    return Engine;
})
```

Przykłady eksportu fabryki (moduł AMD) z żądaniem biblioteki jQuery:

```
define(['jquery:3'], function ($) {
    return function () {
        return {
            init: function () {
            },
            destroy: function () {
            }
        }
    }
})
```

Przykładowy fragment konfiguracji webpack pozwalający na eksport biblioteki:

```
module.exports = {
    // ...
    output: {
        libraryTarget: 'amd'
    },
    externals: {
        jquery: 'jquery:3'
    }
};
```

Przykładowy punkt wejścia dla powyższej konfiguracji:

```
import $ from 'jquery'

// jQuery w wersji 3

function Engine() {
}

Engine.prototype.init = function () {
}
Engine.prototype.destroy = function () {
}

export default Engine
```

Używanie bibliotek

Razem z systemem dostarczone są popularne biblioteki, które można wykorzystać do tworzenia widżetów. Używając załadowaną już bibliotekę, można skutecznie ograniczyć rozmiar tworzonych komponentów.

Biblioteki dostępne do użycia:

- vue (wersja 2)
- vue:3 (wersja 3)
- jquery:3 (wersja 3)
- underscore
- backbone
- axios
- react:16 (wersja 16)

Bibliotek tych, nie należy modyfikować (np. dodawać pluginów).

Jeśli istnieje potrzeba użycia innych popularnym frameworków, które warto dostarczyć w systemie, prosimy o kontakt z administracją portalu.

Dostawca może użyć dowolnego otwartego oprogramowanie i dostarczyć go razem z komponentem.

Kryteria jakościowe

Warunki, jakie musi spełnić kod silnika (kod komponentu oraz całego dostarczonego z nim oprogramowania)

- nie można modyfikować obiektów DOM poza kontenerem dostarczonym w metodzie `init()`.
- nie można modyfikować zmiennych publicznie (w szczególności, w obiekcie `window`).
- nie można dostarczać rozwiązań typu polyfill, które uzupełniają brakujące w przeglądarce funkcję (wynika z punktu powyżej).
- nie można ładować czcionek w głównej ramce oraz styli CSS, które mogłyby zmieniać wygląd reszty strony.
  Do ładowania styli zalecamy użycie metody `loadCSS()`.
- nie można wywoływać żadnych zewnętrznych usług.
- komponent nie może wprowadzać podatności na ataki do systemu, np. umożliwiać wykonanie kodu przedmioty trzecie czy kradzież sesji.
- komponent powinien działać poprawnie, jeśli zostanie uruchomione wiele jego kopii lub inne komponenty.

  Wyjątek: komponent wielokrotnie zainicjalizowany działa poprawnie, lecz ograniczenia przeglądarek nie pozwala na jego poprawne działanie w większej ilości (np. jeśli widżet korzysta z WebGL)

Interfejs silnika

Bazowy interfejs silnika. Implementacja tego interfejsu jest zawsze wymagana.

```
interface Engine {
    init(container: Element, api, options): Promise<void> | void

    destroy(container: Element): void
}
```

- `init(container, api, options)`

  Metoda inicjalizująca komponent

  - `container` - obiekt DOM, w którym należy umieścić komponent.
  - `api` - obiekt API pozwalający na komunikację z systemem
  - `options` - opcje dodatkowe:

    ```
    {
        id: "...",
        contrastMode: false | "yellowOnBlack" | "blackOnYellow" | "whiteOnBlack",
        userRole: "student" | "teacher",
        locale: "pl_PL",
        showAnswers: true | false,
        data: {...}
    }
    ```

    - `id` - unikalny identyfikator osadzenia.
    - `userRole` - rola użytkownika, który uruchamia ćwiczenie (uczeń czy nauczyciel).
    - `contrastMode` - jaki tryb kontrastowy wybrał użytkownik
    - `locale` - preferowany język
    - `showAnswers` - czy widżet powinien eksponować prawidłowe odpowiedzi użytkownikowi (poprzez przyciski “pokaż odpowiedź”, “sprawdź”).
    - `files` - tablica plików utworzona przez edytor
    - `data` - przekazane są wartości z pliku manifest.json instancji.

      Uwaga: dane przekazane w tym polu powinny być traktowane jako niebezpieczne/nieprzefiltrowane. Ich poprawne wykorzystanie leży po stronie silnika.

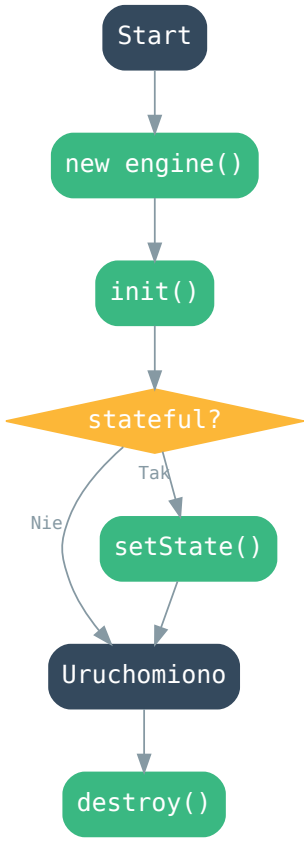
  Obiekt zwracając obietnice, sygnalizuje, że nie zakończył jeszcze inicjalizacji. System wyświetli informację o ładowaniu do czasu ukończenia obietnicy. W przypadku wystąpienia błędu wyświetlony zostanie systemowy komunikat o braku możliwości uruchomienia komponentu.

  Przykład ładowania css:

  ```
  class Example {
      init(container, api, options) {
          return api.loadCss(
              api.enginePath('dist/entry.css')
          ).then(function() {
              console.log('CSS został załadowany');
          });
      }
  }
  ```

- `destroy(container)`

Metoda, która jest wykonywana przed zniszczeniem komponentu.
Jej zadaniem jest usunięcie wszystkich elementów DOM, zdarzeń i uwolnienie wszystkich innych zasobów, które są używane przez komponent.



Uruchomienie aplikacji.

Stateful

Po zmianie stanu komponent powinien wywołać metodę API `triggerStateSave()`. System wkrótce wywoła metodę `getState()`. Zwrócona wartość zostanie zapisana.

```
interface Stateful {
    setState(stateData: any): void

    getState(): any

    setStateFrozen(isFrozen: boolean): void
}
```

- `setState(stateData)` - przywraca stan komponentu. Przy pierwszym uruchomieniu stateData będzie wartością NULL.

  Uwaga: Dane zawarte w `stateData` powinny być traktowane jako niebezpieczne. Ich poprawne wykorzystanie leży po stronie silnika (patrz bezpieczeństwo).

- `getState()` - Zwraca aktualny stan komponentu. Wartość musi być serializowalna.

- `setStateFrozen(isFrozen)` - Ustawia stan zamrożenia. Metoda wywoływana jest zawsze po metodzie setState.

  Po wywołaniu `setStateFrozen(true)`, komponent musi być zablokowany. Nie może wtedy zmieniać stanu.

  Po wywołaniu `setStateFrozen(false)`, komponent musi powrócić do normalnego trybu pracy.

Walidacja stanu

Wymaga implementacji interfejsu stateful.

```
interface AutoValidable extends Stateful {
    isStateValid(stateData: any): Boolean

    showStateValidation(isValidationVisible: Boolean): void
}
```

- `isStateValid(stateData)` - Zwraca informację czy stan komponentu `stateData` odpowiada poprawnemu wykonaniu ćwiczenia.
- `showStateValidation(isValidationVisible)` - Wymusza prezentację walidacji stanu (czy ćwiczenie zostało wykonane poprawnie). Metoda wywoływana jest zawsze po metodzie `setState(stateData)`.

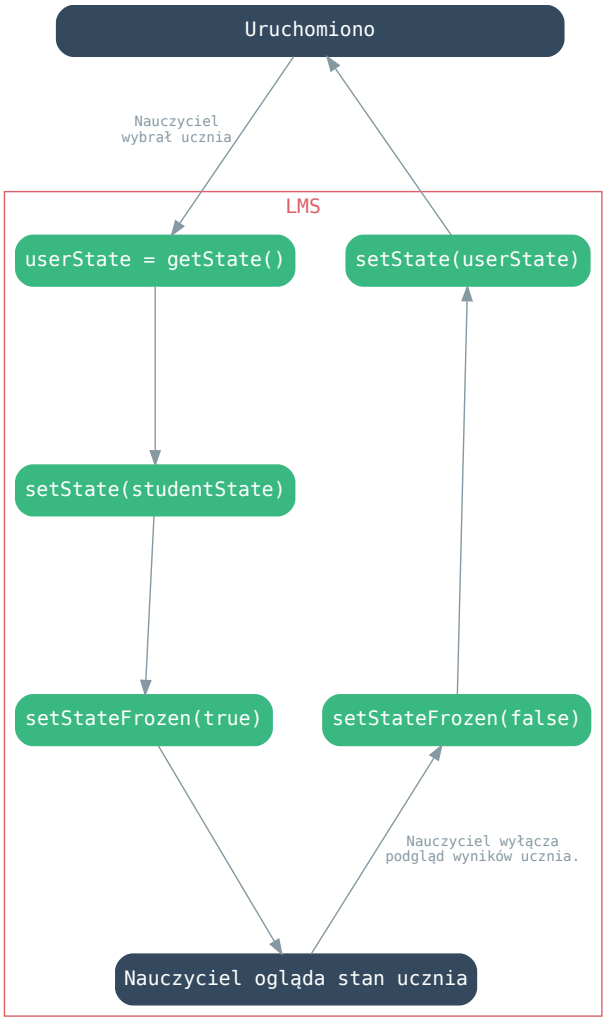
  Po wywołaniu `showStateValidation(true)`, komponent musi zaprezentować walidację stanu. Nie może wtedy zmieniać stanu.

  Po wywołaniu `showStateValidation(false)`, komponent musi ukryć walidację stanu.

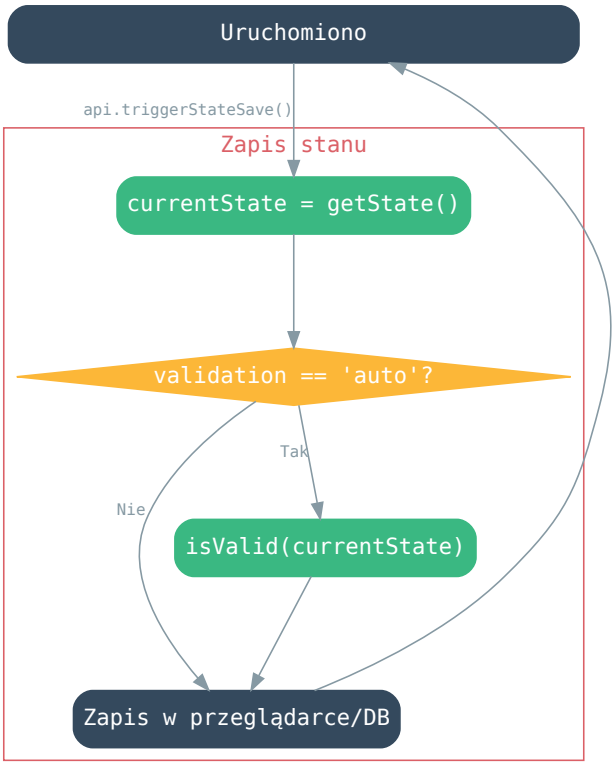
Upload plików przez ucznia

```
interface AllowUserFileUpload extends Stateful {
    getFiles(stateData): Array<string>
}
```

- `getFiles(stateData)` - zwraca tablicę kodów, które mogą być użyte dla obecnego stanu. Pliki wgrane przez ucznia o kodach, które nie znajdują się w tej tablicy, zostaną skasowane. Metoda wywoływana jest podczas wywołania `api.uploadFile(fileId, file)`. Jeśli `fileId` nie będzie znajdować się w tej tablicy, wgranie pliku przez ucznia nie będzie możliwe.



Współpraca modułu z modułem LMS. Prezentacja odpowiedzi ucznia.



Współpraca modułu z modułem LMS. Weryfikacja poprawności wykonanego zadania.

API

Obiekt API jest przekazywany w metodzie init. Umożliwia on komunikację z systemem.

```
interface ExerciseApi {

    triggerStateSave(): Promise<void>;

    triggerStateRestore(): Promise<void>;

    enginePath(path): string;

    dataPath(path): string;

    loadCss(realPath): Promise<void>;

    typesetMath(dom): Promise<void>;

    // Obsługa slotów
    embedWidget(container: Element, manifest: string | object, widgetOptions: object): Promise<void>;

    createWidgetSlot(container: Element, slotName, slotOptions: object, widgetOptions: object): Promise<void>;

    isWidgetSlotFilled(slotName);

    // Obsługa trybu pełnoekranowego
    requestFullscreen(container: Element, onFullscreenExit): Promise<void>;

    toggleFullscreen(container: Element, onFullscreenExit): Promise<void>;

    exitFullscreen(): Promise<void>;

    // Obsługa klawiatury ekranowej
    inputFocusIn(input: Element);

    inputFocusOut(input: Element);

    // Wgrywanie plików przez ucznia
    uploadFile(fileId: string, file: File | Blob): Promise<void>;

    removeUploadedFile(fileId: string): Promise<void>;

    removeUploadedFiles(): Promise<void>;

    // Otwarcie galerii
    openGallery(images);

    // Nadanie nagrody uczniowi
    grantAward(awardCode: string): void;

    // Tryb współpracy
    set awarenessData(value: object);

    sendRoomMessage(message: string | object): void;
}
```

- `triggerStateSave()` - metoda powinna zostać wywołana po zmianie stanu komponentu.
- `triggerStateRestore()` - dodatkowa metoda, która wymusza przywrócenie stanu komponentu z bazy danych. Używanie jej nie jest konieczne.
- `enginePath(path)` - zwraca ścieżkę relatywną względem katalogu silnika.
- `dataPath(path)` - zwraca ścieżkę relatywną względem katalogu danych.
- `loadCss(realPath)` - ładuje plik CSS

- `typesetMath(dom)` - modyfikuje wskazany element DOM. Konwertuje elementy `<math>` tak, aby były poprawnie wyświetlone na wszystkich przeglądarkach.
- `embedWidget(container, manifest, widgetOptions)` - ładuje inną aplikację w elemencie dom `container`.
  W drugim argumencie można podać cały manifest lub identyfikator komponentu. W przypadku podania identyfikatora musi on być zawarty w `manifest.json` w polu `dependencies`.
- `createWidgetSlot(container, slotName, slotOptions, widgetOptions)` - ładuje inną aplikację w elemencie dom `container`. W drugim argumencie podajemy nazwę slotu (dowolna). Slotami zarządza edytor. Jeśli slot został uzupełniony, załadowana zostanie inna aplikacja. W innym przypadku funkcja zwróci błąd (obietnica zostanie odrzucona).
- `isWidgetSlotFilled(slotName)` - zwraca informację czy slot jest uzupełniony.
- `openGallery(images)` - uruchamia systemową galerię zdjęć. W argumencie należy przekazać jeden lub tablicę elementów `<img>`.
- `requestFullscreen(container, onFullscreenExit)` - uruchom kontener w trybie pełnoekranowym.
- `toggleFullscreen(container, onFullscreenExit)` - uruchom kontener w trybie pełnoekranowym. Jeśli jest już w trybie pełnoekranowym, to go opuść.
- `exitFullscreen()` - opuść tryb pełnoekranowy.
- `inputFocusIn(input)` - informuje system, że do elementu formularza należy wyświetlić klawiaturę ekranową.
- `inputFocusOut(input)` - informuje system, że do elementu formularza należy przestać wyświetlać klawiaturę ekranową.
  Uwaga: nie należy bazować tu na zdarzeniu `focus`. W trakcie obsługi klawiatury ekranowej może ona przejmować fokus.
- `api.grantAward(awardCode: string)` - nadaje uczniowi `awardCode` (zdefiniowana w `engine.json`). Nagroda nadawana jest zawsze indywidualnie (nawet w trybie współpracy). Ponowne wywołanie tej samej metody nie powoduje ponownego nadania nagrody. W przypadku problemów z połączeniem internetowym system poinformuje o tym fakcie użytkownika i podejmie ponowne próby nadania nagrody. W przypadku próby nadania nagrody, która nie jest zdefiniowana w `engine.json`, metoda wyrzuci wyjątek `ApiError(name: "AwardNotDefined")`
- `set awarenessData(value: object)` - umożliwia modyfikację stanu obecnego ucznia w trybie współpracy. Wewnątrz można przechowywać dane o dowolnej strukturze. Maksymalna ilość danych wynosi 1 KB. W przypadku przekroczenia limitów setter wyrzuci wyjątek `ApiError(name: "LimitExceeded")`. Dane będą automatycznie synchronizowane pomiędzy połączonymi użytkownikami, ale nie częściej niż 5 razy w ciągu sekundy (nie każda aktualizacja trafi zawsze do wszystkich użytkowników). Dane są ulotne i są tracone po zakończeniu połączenia. Mechanizm został szczegółowo opisany w sekcji informowanie o połączonych uczniach i ich wewnętrznym stanie.
- `sendRoomMessage(message: string | object)` - umożliwia wysłanie wiadomości do innych połączonych użytkowników w trybie współpracy. Mechanizm został szczegółowo opisany w sekcji wysyłanie ulotnych wiadomości do wszystkich podłączonych uczniów.

Błędy wywołania API

Nieprawidłowe wywołania API skutkują wyrzuceniem wyjątku o następującej strukturze:

```
class ApiError extends Error {
    name: string;
    message: string;
}
```

- `name` - zawiera kod błędu.
- `message` - zawiera szczegółowy opis błędu.

Parametry CSS

Z poziomu CSS, do dyspozycji są następujące zmienne:

- `--font-sans` - systemowy font bezszeryfowy.
- `--font-serif` - systemowy font szeryfowy.
- `--font-mono` - systemowy font o stałej szerokości znaku.

Umożliwiają one wykorzystanie tej samej czcionki, co reszta elementów systemu.

Przykład użycia:

```
.my-class {
    font-family: var(--font-sans);
}
```

Współpraca

Uruchomienie trybu współpracy umożliwia użytkownikom wspólną pracę nad zadaniami. Rozwiązania umożliwiające pracę w grupie:

- budowanie stanu wspólnego dla całej grupy przechowywanego na serwerze
- informowanie o połączonych uczniach i ich wewnętrznym stanie
- wysyłanie ulotnych wiadomości do wszystkich podłączonych uczniów

Uruchomienie trybu współpracy

Do pliku `engine.json` należy dodać deklarację informującą, że komponent wspiera tryb współpracy:

```
{
    "entry": "entry.js",
    "stateful": true,
    "collaboration": true
}
```

System decyduje czy komponent należy uruchomić w trybie współpracy (multiplayer), czy nie (singleplayer). Ćwiczenie musi zweryfikować wartość przekazaną w `options.collaboration` i na jej podstawie dostosować zachowanie.

```
define([], function () {
    function Engine() {
    }

    Engine.prototype.init = function (container, api, options) {
        if (options.collaboration) {
            // collaboration enabled
        } else {
            // singleplayer mode
        }
    }

    Engine.prototype.destroy = function () {
    }

    return Engine;
})
```

Budowanie wspólnego stanu

Metody `setState(state)`, `isStateValid(state)` przyjmują dodatkowy argument `collaborationState`. `getState()` zostaje bez zmian i zwraca stan dla ucznia. Po modyfikacji `collaborationState` nie ma konieczności informowania api o zmianie poprzez wywołanie `api.triggerStateSave()`.

W przypadku uruchomienia podglądu przez nauczyciela, po wybraniu grupy system wywoła `setState(state, collaborationState)` oraz `setStateFrozen(true)`. W tym przypadku stan `collaborationState` jest zablokowany i nie można na nim wprowadzać zmian.

Wymagana jest implementacja interfejsu `StatefulCollaborative`:

```
interface StatefulCollaborative {
    setState(stateData: any, collaborationState: CollaborationState): void

    getState(): any

    setStateFrozen(isFrozen: boolean): void
}

interface CollaborationState {

    getMap(name: string): SyncedMap;

    getArray(name: string): SyncedArray;
}

interface SyncedMap {
    onChange: (key: string, newValue: any) => void;
    onDelete: (key: string) => void;

    get(key: string): any;

    set(key: string, newValue: any): void;

    keys(): string[];

    delete(key: string): void;

    setSyncedWith(variable: any): void;

    toJSON(): any;
}

interface SyncedArray {
    // Callback pozwalający na
    onSplice: (start, deleteCount, ...items) => void;

    get length(): number;

    get(key: number): any;

    set(key: number, newValue: any);

    push(...newValue): void;

    splice(start, deleteCount, ...items): void;

    setSyncedWith(variable: any): void;

    toJSON(): any;
}
```

`collaborationState` jest zbiorem zmiennych synchronizowanych z serwerem. Zmienne mogą mieć dwa typy `SyncedArray` i `SyncedMap`.

System zapewnia spójność danych niezależnie od kolejności wykonywanych operacji. Oba typy pozwalają na przechowanie dowolnej struktury (nie tylko prymitywnych typów).

System ogranicza synchronizację tylko do danych na najwyższym poziomie. Zalecane jest stosowanie płaskich struktur (oddzielne klucze na oddzielne pola), aby ograniczyć konflikty przy równoległej edycji.

Przykład:

```
collaborationState.getArray('users').push(
        {
            "name": "Adam",
            "surname": "Nowak"
        }
)
```

W tym przypadku nie ma możliwości, aby jeden z użytkowników edytował pole `surname` a drugi `name`. Obiekt jest przechowywany w całości.

W przypadku utraty połączenia z serwerem system będzie wyświetlać komunikat o ponownej próbie połączenia. Po odzyskaniu połączenia dane zostaną ponownie synchronizowane.

Zmiana danych przez ucznia wywołuję następujące callbacki u reszty uczniów:

- dla `SyncedArray` wywoływany jest callback `onSplice`.
- dla `SyncedMap` wywoływany jest callback `onChange` lub `onDelete`.

Oba typy zawierają metodę `setSyncedWith(variable)`. Metoda konfiguruje callbacki `onSplice`/`onChange`/`onDelete` tak, aby zmiany w strukturze były odzwierciedlone w natywnej zmiennej javascript.

Przykład wykorzystania listy we frameworku Vue:

```
<template>
    <div>
        <template v-for="(item, key) of list">
            <div>
                <label>
                    Element: {{ key + 1 }}
                </label>
                <input
                        type="text"
                        :value="item.text"
                        @change="todoList.set(key, {
                        text: $event.target.value
                    })"
                />
                <button @click="todoList.splice(key, 1)">
                    Usuń
                </button>
            </div>
        </template>
        <div>
            Ilość elementów: {{ list.length }}
        </div>
        <button @click="todoList.splice(0, 0, {
            text: 'New item'
        })">
            Dodaj na początku
        </button>
        <button @click="todoList.push({
            text: 'New item'
        })">
            Dodaj na końcu
        </button>
    </div>
</template>
<script>
    import {markRaw} from "vue";

    export default {
        name: "TodoList",
        props: {
            collaborationState: {}
        },
        data() {
            return {
                todoList: null,
                list: []
            };
        },
        mounted() {
            this.todoList = markRaw(this.collaborationState.getArray('todoList'));
            this.todoList.setSyncedWith(this.list);
        }
    }
</script>
```

W powyższym przykładzie z `collaborationState` pobrano `SyncedArray` o nazwie `todoList`. Zmiany w liście przepisywane są na reaktywną zmienną `list`, która jest używana do wyświetlania listy. Zmiany w liście wprowadzane są bezpośrednio w obiekcie `SyncedArray`.

Walidacja stanu dla trybu współpracy

Jeśli komponent wspiera tryb współpracy, to weryfikacja stanu jest oparta tylko na `collaborationState`. Pierwszy argument metody `isStateValid` ma zawsze wartość `null`.

```
interface AutoValidableCollaborative extends StatefulCollaborative {
    isStateValid(stateData: null, collaborationState: CollaborationState): Boolean;

    showStateValidation(isValidationVisible: Boolean): void;
}
```

Informowanie o połączonych uczniach i ich wewnętrznym stanie

Aby uzyskać dostęp do danych o połączonych uczniach należy zaimplementować poniższy interfejs:

```
interface RoomUsersAware {
    onRoomActiveUsersChange(activeUsers: Array<UserConnectionData>);

    onRoomSignedUpUsersChange(signedUpUsers: Array<UserData>);
}

interface UserData {
    // Identyfikator użytkownika
    id: string,
    // Wyświetlana nazwa użytkownika
    name: string,
    // URL avatara
    avatar: string,
    // Rola użytkownika
    userRole: Role;
}

interface UserConnectionData extends UserData {
    // Identyfikator połączenia
    connectionId: string,
    // Aktualny stan użytkownika
    awarenessData: any;
}

enum Role {
    Student = "student",
    Teacher = "teacher",
}
```

- `onRoomActiveUsersChange(activeUsers)` - funkcja wywoływana po połączeniu/odłączeniu ucznia oraz po zmianie `api.awarenessData` przez dowolnego ucznia. `activeUsers` zawiera listę podłączonych uczniów razem z ID połączenia oraz ich stanem.
- `onRoomSignedUpUsersChange(signedUpUsers)` - funkcja wywoływana po uruchomieniu i po dodaniu nowego ucznia do grupy. `signedUpUsers` zawiera listę wszystkich zapisanych uczniów w danej grupie.

Pole `api.awarenessData` pozwala na zmianę stanu obecnie zalogowanego ucznia.

Przykładowy kod śledzący położenie kursora myszy innych uczniów:

```
define([], function () {
    function Engine() {
    }

    Engine.prototype.init = function (container, api, options) {
        this.api = api;
        window.addEventListener('mousemove', this._onMouseMove)
    }

    Engine.prototype.destroy = function (users) {
        window.removeEventListener('mousemove', this._onMouseMove)
    }

    Engine.prototype._onMouseMove = function (event) {
        this.api.awarenessData = {
            x: event.screenX,
            y: event.screenY
        };
    }

    Engine.prototype.onRoomActiveUsersChange = function (activeUsers) {
        // Zmiena activeUsers zawiera listę uczniów którzy mają aktywnie uruchomiony komponent.
        // Przykładowa zawartość zmiennej:
        [
            {
                // Dane użytkownika
                id: '3a7f3717-b753-48d8-9fbf-db31e3e2f41e',
                name: 'Jan Kowalski',
                avatar: 'https://static.zpe.gov.pl/avatar.png',

                // ID połączenia
                connectionId: '52aa178d-9618-4b30-b67a-7add84a20b65',
                // Aktualny stan użytkownika
                awarenessData: {
                    x: 0,
                    y: 0
                },
            },
            {
                id: '5c58ac45-ce0f-4c6a-a138-3670eefbca86',
                name: 'Adam Nowak',
                avatar: 'https://static.zpe.gov.pl/avatar.png',

                connectionId: 'd331abc4-3779-4810-ae3d-ecc650b29844',
                awarenessData: {
                    x: 231,
                    y: 555
                },
            }
        ]
    }

    Engine.prototype.onRoomSignedUpUsersChange = function (signedUpUsers) {
        // Zmiena signedUpUsers zawiera listę wszystkich współpracujących uczniów.
        // Przykładowa zawartość zmiennej:
        [
            {
                id: '3a7f3717-b753-48d8-9fbf-db31e3e2f41e',
                name: 'Jan Kowalski',
                avatar: 'https://static.zpe.gov.pl/avatar.png',
            },
            {
                id: '5c58ac45-ce0f-4c6a-a138-3670eefbca86',
                name: 'Adam Nowak',
                avatar: 'https://static.zpe.gov.pl/avatar.png',
            }
        ]
```

```
    }

    return Engine;
})
```

Wysyłanie ulotnych wiadomości do wszystkich podłączonych uczniów

Aby otrzymywać wiadomości od innych użytkowników w grupie, należy zdefiniować interfejs:

```
interface RoomMessageAware {
    onRoomMessage(sender: UserConnectionData, message: string | object): void;
}

interface UserData {
    // Identyfikator użytkownika
    id: string,
    // Wyświetlana nazwa użytkownika
    name: string,
    // URL avatara
    avatar: string,
    // Rola użytkownika
    userRole: Role;
}

interface UserConnectionData extends UserData {
    // Identyfikator połączenia
    connectionId: string,
    // Aktualny stan użytkownika
    awarenessData: any;
}

enum Role {
    Student = "student",
    Teacher = "teacher",
}
```

- **onRoomMessage(sender, message)**: Funkcja będzie wywoływana, gdy inna instancja komponentu wywoła metodę `api.sendRoomMessage(message)`.

Przykład:

```
define([], function () {
    function Engine() {
    }

    Engine.prototype.init = function (container, api, options) {
        api.sendRoomMessage(
                'Wysyłam wiadomość do wszystkich innych uczniów w grupie'
        );
        api.sendRoomMessage({
            'text': "Mogę przekazać także obiekt"
        });
    }

    Engine.prototype.destroy = function (users) {
    }

    Engine.prototype.onRoomMessage = function (sender, message) {
        // Wiadomość otrzymana od ucznia sender
        console.log(message)
    }

    return Engine;
})
```

Maksymalny rozmiar przesyłanej wiadomości to 1 KB. Maksymalna ilość wysyłanych wiadomości to 5 wiadomości / sekundę dla każdego użytkownika. Wysłane wiadomości nie są przechowywane na serwerze. Użytkownik nie otrzyma wiadomości, jeśli nie będzie połączony w trakcie wysyłania wiadomości.

W przypadku przekroczenia limitów funkcja wyrzuci wyjątek `ApiError(name: "LimitExceeded")`. W przypadku braku aktywnego połączenia wywołanie metody `api.sendRoomMessage(message)` wyrzuci wyjątek `ApiError(name: "Disconnected")`.

Status połączenia

Aby otrzymywać informację o stanie połączenia, należy zdefiniować interfejs:

```
interface ConnectionAware {
    onConnectionStatusChange(status: ConnectionStatus): void;
}

enum ConnectionStatus {
    Connected = "connected",
    Disconnected = "disconnected",
}
```

Przykład:

```
define([], function () {
    function Engine() {
    }

    Engine.prototype.init = function (container, api, options) {
    }

    Engine.prototype.destroy = function (users) {
    }

    Engine.prototype.onConnectionStatusChange = function (status) {
        if (status === 'connected') {
            // połączono
        }
        if (status === 'disconnected') {
            // użytkownik utracił połączenie
        }
    }

    return Engine;
})
```

Edytor komponentów

Edytor jest osobną aplikacją, z osobnym punktem wejścia (zdefiniowanym w `engine.json`). Umożliwia stworzenie komponentów, które korzystają z utworzonego wcześniej silnika. Jego implementacja nie jest wymagana do poprawnego działania komponentu.

Interfejs edytora

```
interface EngineEditor {

    init(api, options): void

    destroy(): void

    initTab(tab, container: Element, api): Promise<void> | any

    destroyTab(tab, container): void

    setState(stateData): void

    getState(): Object
}
```

- `init(api, options)`

  Metoda inicjalizująca edytor. Należy w niej zadeklarować zakładki, jakie mają być utworzone.

- `destroy()`

  Metoda, która jest wykonywana przed zniszczeniem edytora.
  Jej zadaniem jest usunięcie wszystkich elementów DOM, zdarzeń i uwolnienie wszystkich innych zasobów, które są używane przez edytor.

- `initTab(tab, container, api)`

  Metoda inicjalizująca tab. Jest wywoływana po wywołaniu `init()` lub po `destroyTab()` (w przypadku zmiany zakładki na inną).

- `destroyTab(tab, container)`

  Metoda czyszcząca zakładkę.

  Jej zadaniem jest usunięcie wszystkich elementów DOM, zdarzeń i uwolnienie wszystkich innych zasobów które są używane przez zakładkę.

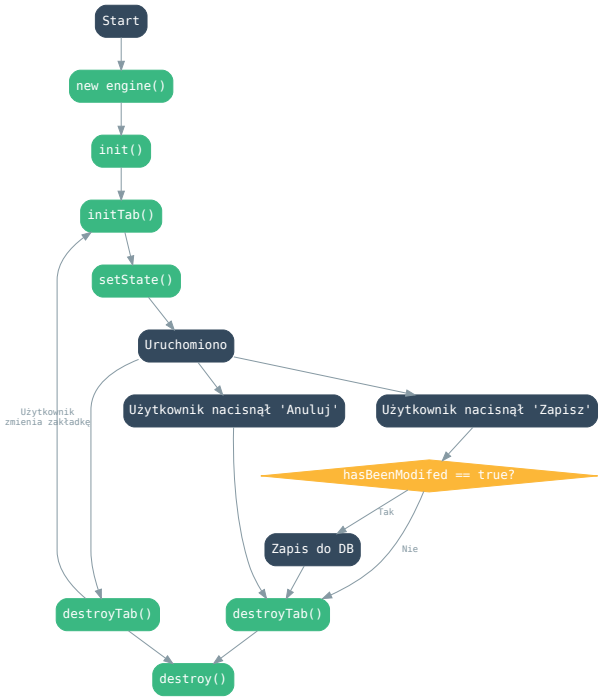
- `setState(stateData)`

  Ustawienie aktualnego stanu dla edytora.

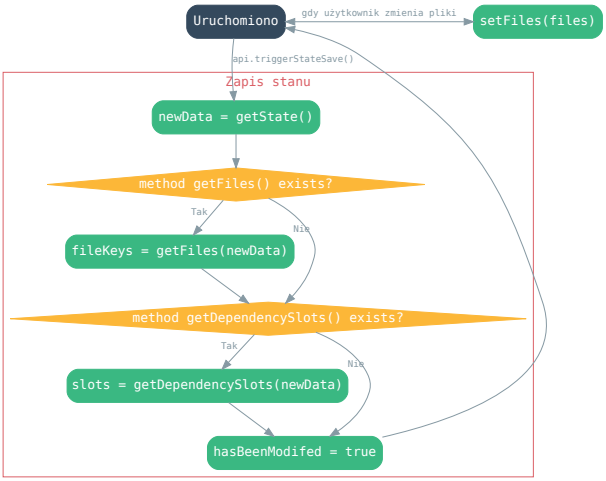
  Przy pierwszym tworzeniu komponentu, do `stateData` przekazana będzie wartość `defaultData` zawartą w engine.json lub NULL.

- `getState()`

  Zwraca aktualny stan edytora. Wartość musi być serializowalna.



Uruchomienie edytora.



Zapis danych w edytorze.

API dla edytora

Obiekt API jest przekazywany w metodzie init. Umożliwia on komunikację z systemem. Oprócz wymienionych poniżej

```
interface ExerciseEditorApi {

    addEditorTab(id, name): ExerciseEditorApi;

    triggerStateSave(): Promise<void>;

    enginePath(path): string;

    dataPath(path): string;

    loadCss(realPath): Promise<void>;

    typesetMath(dom): Promise<void>;

    embedWidget(container: Element, manifest, widgetOptions): Promise<void>;

    createWidgetSlot(container: Element, slotName, slotOptions, widgetOptions): Promise<void>;

    isWidgetSlotFilled(slotName): boolean;
}
```

- `addEditorTab(id, name)` - metoda pozwala na utworzenie nowej zakładki.
- `triggerStateSave()` - metoda powinna zostać wywołana po zmianie stanu edytora.
- `createWidgetSlot(container, slotName, slotOptions, widgetOptions)` - ładuje inną aplikację w elemencie dom `container`.
- `isWidgetSlotFilled(slotName)`

Metody `enginePath()`, `dataPath()`, `loadCss()`, `typesetMath()`, `embedWidget()` działają identycznie jak w przypadku API instancji.

Pliki

Interfejs pozwala na zadeklarowanie zapotrzebowania na pliki, które użytkownik tworzący widżet musi wgrać.

```
interface HasFileUploadSupport extends EngineEditor {

    getFiles(state: any, options: object): Record<string, RequestedFile>;

    setFiles(files: Record<string, File>): void
}

interface RequestedFile {
    // Nazwa pliku wyświetlana w edytorze
    label: string;
}

interface File {
    // Scieżka relatywna względem folderu
    path: string;
    // Absolutna ścieżka do pliku
    url: string;
    // Opis WCAG pliku wpisany w edytorze
    wcag: string;
}

class HasFileUploadSupportExample implements HasFileUploadSupport {
    getFiles(state: any, options: object): Record<string, RequestedFile> {
        return {
            'KOD': {
                'label': 'Nazwa pliku'
            },
            // ....
        };
    }

    setFiles(files: Record<string, File>): void {

    }
}
```

- `getFiles(state, options)` - metoda powinna zwrócić listę plików dla zadanego stanu.
- `setFiles(files)` - metoda wywoływana po wgraniu/usunięciu pliku przez użytkownika. System przekazuje aktualną listę plików.

Pliki dostępne są w instancji w metodzie `init()` w parametrze options.files

```
class ExampleFilesApp {
    init(container, api, files) {
        let url = api.dataPath(files['IMG1']);
        console.debug(
                url ? 'Url pliku:' + url : 'Plik nie został wgrany'
        );
    }
}
```

DependencySlots

```
interface HasDependencySlotsSupport extends EngineEditor {
    getDependencySlots(state);
}
```

Dodatkowy interfejs, wymagany w przypadku użycia metody `createWidgetSlot()`. Metoda `getDependencySlots()` powinna zwrócić wszystkie używane obecnie sloty (tablicę z nazwami), dla danego stanu edytora.

W przypadku zwrócenia niepustej tablicy pojawi się dodatkowa zakładka umożliwiająca uzupełnienie slotu oraz edycję komponentu wewnątrz. Może to być inne ćwiczenie, bądź film.

System w żaden sposób nie sygnalizuje zmiany slotu. Po zmianie jego zawartości przez użytkownika, zawartość kontenera wskazanego w `createWidgetSlot()` zostaje zaktualizowana.

Instancja komponentu

Gotowy silnik można wykorzystać, tworząc instancję komponentu.

Instancję można utworzyć:

- przeciągając komponent z paska narzędziowego, jeśli został dla niego utworzony edytor.

  Uwaga: edytor jest widoczny na pasku narzędziowym, tylko w przestrzeni, w której został utworzony komponent.

- wgrywając paczkę ZIP zawierającą plik `manifest.json` oraz inne pliki, z których korzysta silnik, a które są specyficzne dla danej instancji.

  Struktura pliku `manifest.json`:

  ```
  {
    "engine": "nazwa przestrzeni/kod silnika",
    "dependencies": [],
    "data": {}
  }
  ```

  Plik `manifest.json` musi znajdować się na samym szycie struktury archiwum (nie może znajdować się w żadnym katalogu).

Komponent w wersji drukowanej (PDF)

System, podczas generowania wersji PDF wykorzystuje mechanizmy przeglądarki do wydruku podstrony.

- komponent, który można drukować, należy oznaczyć w engine.json `"printable": true`.
- instancja musi być gotowa do druku, po zakończeniu wywołania `init()` lub po skończeniu zwróconego `Promise`.
- druk musi uwzględniać ustawiony stan ćwiczenia.
- walidacja musi być widoczna w druku.
- można wykorzystywać reguły `@media print` do ukrycia elementów interfejsu.

  ```
  @media print {
      .example {
         display: none;
      }
  }
  ```

Użycie systemowego odtwarzacza multimediów wewnątrz widżetu

- simple *(opcjonalny)* - uproszczony odtwarzacz (tylko odtwórz/zatrzymaj).
- parametry źródła:
  - mediaType **(wymagany)** - rodzaj źródła. Dozwolone wartości: `audio`, `video`.
  - src **(wymagany)** - adres URL źródła w formie mp3 lub mp4.
  - name *(opcjonalny)* - opis wyświetlony tylko w przypadku użycia playlisty.
  - poster *(opcjonalny)* - adres URL pliku `png` lub `jpg`. Obraz zostanie wyświetlony przed uruchomieniem filmu.
  - cam360 *(opcjonalny)* - czy film 360. Obsługiwane jest wyłącznie odwzorowanie walcowe równoodległościowe (equirectangular projection)
- parametry odtwarzacza
  - controls
    - volume: czy pokazywać kontrolę głośności.
    - quality: czy pokazywać wybór jakości (jeśli dostępne).
    - playbackRate: czy pokazywać wybór prędkości odtwarzania.
    - fullscreen: czy pokazywać przycisk do uruchomienia filmu na pełnym ekranie.

```
api.embedWidget(
        container,
        {
            "engine": "core/media-player",
            "data": {
                "simple": false,
                "playlist": [
                    {
                        "id": 1,
                        "mediaType": "video" | "audio",
                        "src": source,
                        "name": "Nazwa ścieżki",
                        "cam360": false,
                        "poster": videoPoster,
                    }
                ]
            }
        },
        {
            controls: {
                volume: true,
                quality: true,
                playbackRate: true,
                fullscreen: true
            }
        }
)
```

Dodatkowe silniki

Umieszczanie pakietów HTML przy pomocy `core/iframe`

Silnik umożliwia załączenie w elemencie `<iframe>` pakietu HTML (składającego się z pliku głównego HTML, oraz w razie potrzeby dodatkowych plików: JS, CSS, graficznych).

przykładowy plik `manifest.json`:

```
{
    "engine": "core/iframe",
    "data": {
        "index": "index.html",
        "fullscreenButton": true,
        "heightRatio": 0.5625
    }
}
```

opcje:

- **index**: relatywny (względem `manifest.json`) adres pliku HTML który zostanie uruchomiony w ramce.
- **fullscreenButton**: opcja umożliwiająca pokazanie przycisku do uruchomienia ramki na pełnym ekranie
- **heightRatio**: stosunek wysokości ramki do szerokości.
  np. 0.5625 da proporcji panoramicznych

Plik `manifest.json` należy dodać do istniejącego pakietu HTML i spakować całość w archiwum ZIP. Plik `manifest.json` musi znajdować się na samym szycie struktury archiwum (nie może znajdować się w żadnym katalogu).