Programowanie obiektowe i wieloplatformowe - projekt
Kierunek: Informatyka w systemach i układach elektronicznych
Semestr: V

Weather-Calendar

Prowadzący projekt: …

Członkowie sekcji:
Tomasz Wojtasek
Paweł Kurek

Gliwice, 2025

1 Wstęp

1.1 Cel projektu

Celem projektu było stworzenie aplikacji umożliwiającej użytkownikowi zaplanowanie spotkań oraz przeglądanie prognozy pogody w zaplanowanym dniu. Użytkownik będzie mógł wybrać konkretny dzień w kalendarzu, a aplikacja wyświetli zapisane spotkanie oraz prognozę pogody, w tym informacje między innymi o temperaturze, wilgotności, ciśnieniu, opadach oraz ikonę wskazującą warunki atmosferyczne.

1.2 Zakładane funkcje i cele

- Możliwość dodawania spotkania w określonym dniu, wykorzystując mechanizm kalendarza.
- Wyświetlanie prognozy pogody oraz informacji o spotkaniu po kliknięciu w dany dzień w kalendarzu. Automatyczne aktualizacje interfejsu użytkownika, gdy dane ulegną zmianie.
- Informowanie użytkownika o wszelkich błędach czy sukcesach wykonywanych operacji. Wprowadzenie mechanizmu wyjątków.
- Aplikacja ma pobierać najnowsze dane pogodowe z zewnętrznego API.
- Aplikacja ma przechowywać dane o spotkaniach w plikach binarnych, a w przyszłości również w plikach JSON.
- Rozdzielenie logiki interfejsu użytkownika od logiki aplikacji.
- Opracowanie styli CSS w celu stworzenia przejrzystego graficznego interfejsu użytkownika.

2 Projekt oprogramowania

2.1 Architektura aplikacji

W projekcie aplikacji Weather-Calendar skupiono się na rozdzieleniu logiki interfejsu użytkownika od logiki aplikacji poprzez zastosowanie warstw.

1. **Controller**: warstwa kontrolerów odpowiada za interakcje z użytkownikiem, aktualizację widoku i delegowanie zadań do warstwy serwisów. Na przykład, `CalendarController` zarządza widokiem kalendarza i przekazuje dane do `CalendarService`, który dostarcza informacje o rozmieszczeniu elementów w interfejsie. Podobnie, `EventController` współpracuje z `EventService` oraz `WeatherService`, aby zarządzać wydarzeniami i danymi pogodowymi.
2. **Service**: warstwa serwisów przetwarza dane wprowadzone przez użytkownika i integruje różne komponenty aplikacji. Przykładowo `CalendarService` odpowiada za generowanie i aktualizację kalendarza, współpracując z innymi serwisami (np. `WeatherService`). Serwisy zajmują się również walidacją danych i rzucają wyjątki, które kontrolery mogą obsłużyć.
3. **Manager**: przechowuje dane aplikacji, zapewniając jednolity dostęp do nich. Klasy menadżerów (np. `WeatherManager`, `EventManager`) zarządzają stanem aplikacji, umożliwiając dodawanie, usuwanie i pobieranie danych.
4. **Model**: modele reprezentują dane (np. zaplanowane wydarzenia, prognozy pogody) i zawierają podstawową logikę biznesową (np. pomocnicze metody do serializacji, operację na kolekcjach danych). Są używane do przechowywania informacji, które użytkownik widzi w interfejsie.
5. **Uiutil**: warstwa „uitutil" wspiera interakcję z użytkownikiem. Zawiera klasy pomocnicze, takie jak `CalendarButton`, które dodają funkcjonalność do standardowych komponentów GUI. Zajmuje się również obsługą komunikatów o błędach (`AlertError`, `AlertException`) oraz przełączaniem scen w aplikacji (`StageAssistant`).
6. **Util**: warstwa "util" zawiera narzędzia wspierające inne warstwy, takie jak `QueryAssistant` do obsługi zapytań HTTP oraz `GlobalStateAssistant` do zapisu i odczytu stanu aplikacji. Zawiera również klasy wyjątków, jak `ApiException` i `GlobalStateException`, które obsługują błędy specyficzne dla tych operacji.

2.2 Wybrane technologie

- **Język programowania:** Java 23
- **Framework:** JavaFX 23 (do budowy GUI)
- **System zarządzania zależnościami i budowy projektu:** Maven 3.8.0
- **Pliki FXML:** Do definiowania układu interfejsu użytkownika.
- **Zaprojektowanie GUI:** SceneBuilder
- **Pliki CSS:** Do stworzenia stylu GUI.
- **Zewnętrzne API:** Visual Crossing Weather API (do pobierania danych pogodowych).
- **Serializacja, deserializacja z API:** biblioteka Gson 2.11.0
- **Zapis danych:** Pliki binarne oraz planowany zapis do plików JSON.

2.3 Zakładany sposób przepływu danych

1. **Uruchomienie aplikacji**: Aplikacja jest uruchamiana, a metoda `init` wykonuje **wstępną konfigurację**. Stan kolekcji spotkań (z klasy `EventManager`) jest ładowany z pliku binarnego. Tworzony jest obiekt `AutoWeatherService`, który wykonuje zapytania do API pogodowego automatycznie co określony czas, działając na osobnym wątku. Aktualizuje pogodę dla wszystkich lokalizacji (pobranych z `EventManager`).
2. **Wybór roku i miesiąca**: Użytkownik wybiera rok oraz miesiąc w kalendarzu za pomocą kontrolek generowanych przez `CalendarController`. Po zmianie miesiąca lub roku, generowany jest nowy kalendarz. `CalendarService` dostarcza dane, które są prezentowane w interfejsie użytkownika. W przypadku przypisanych spotkań lub prognoz pogody, odpowiednie dane (np. nazwa spotkania lub ikona pogody) są wyświetlane na elemencie kalendarza.
3. **Dodanie nowego spotkania**: Użytkownik wybiera pusty element kalendarza, co uruchamia kontroler `NewEventController`. Po przypisaniu parametrów spotkania i naciśnięciu „Zapisz spotkanie”:
    a. **`EventService`** waliduje dane, a jeśli są poprawne, zaktualizowuje stan aplikacji za pomocą EventManager.
    b. **`WeatherService`** wykonuje zapytanie do API pogodowego, aby zaktualizować dane pogodowe dla wybranej lokalizacji i zaktualizować stan aplikacji poprzez `WeatherManager`.
    c. **Jeśli dane są niepoprawne**, `NewEventController` przechwytuje wyjątki i wyświetla komunikaty błędów przy użyciu klasy `AlertException`.
4. **Zakończenie dodawania spotkania**: Po dodaniu spotkania, okno jest zamykane, a użytkownik otrzymuje alert o sukcesie operacji (za pomocą klasy `AlertSuccess`).
5. **Wyświetlanie spotkania**: Użytkownik wybiera nowo dodane spotkanie z kalendarza. Uruchamia to kontroler `EventController`, który pobiera dane spotkania za pomocą `EventService` oraz prognozy pogody przy użyciu `WeatherService`. Dane są wstawiane do kontrolek tekstowych w JavaFX.
6. **Edycja i usuwanie spotkania**: Użytkownik ma możliwość edytowania lub usunięcia spotkania, co wywołuje odpowiednie metody w `EventService`.
    a. W przypadku edycji, po pomyślnej zmianie, kontrolki JavaFX są automatycznie zaktualizowane, a użytkownik otrzymuje informację o sukcesie operacji (poprzez `AlertSuccess`).
    b. Przy usunięciu spotkania, dane są usuwane z `EventManager`, a interfejs zostaje odpowiednio zaktualizowany. Ikona pogody i nazwa spotkania w kalendarzu są resetowane.
7. **Automatyczna aktualizacja prognozy pogody**: Jeśli prognoza pogody ulegnie zmianie (np. dzięki działaniu `AutoWeatherService`), kontrolki JavaFX zostaną automatycznie zaktualizowane, korzystając z mechanizmu `Observable` w JavaFX.
8. **Zamknięcie aplikacji**: Użytkownik zamyka aplikację, co uruchamia metodę `stop`. Stan kolekcji spotkań w `EventManager` jest zapisywany do pliku binarnego, zapewniając zapisany stan aplikacji. Wątek związany z obiektem `AutoWeatherService` jest zakańczany.

3 Opis realizacji

3.1 Menadżer aplikacji

```
public class EventManager {
    private static EventManager instance;
    private ObservableMap<LocalDate, ScheduledEvent> events;

    private EventManager() {
        this.events = FXCollections.observableHashMap();
    }
    public static synchronized EventManager getInstance() {
        if (instance == null) {
            instance = new EventManager();
        }
        return instance;
    }
```

Rys. Wzorzec singleton dla klas typu Manager, na przykładzie `EventManager`.

W aplikacji zastosowano **wzorzec singleton** dla klas zarządzających globalnym stanem, takich jak **`EventManager`** i **`WeatherManager`**. Dzięki temu każda z tych klas ma tylko jedną instancję dostępną globalnie. W metodzie `getInstance()` sprawdzane jest, czy instancja już istnieje. Jeśli nie, tworzony jest nowy obiekt; jeśli tak, zwracana jest już istniejąca instancja.

```
public void addEvent(LocalDate date, ScheduledEvent event) {
    events.put(date, event);
}

public void deleteEvent(LocalDate date) {
    if (!events.containsKey(date)) {
        throw new IllegalArgumentException(
                "Event not found for the given date: " + date);
    }
    events.remove(date);
}
```

Rys. Przykładowe metody do operacji na kolekcji danych w `EventManager`.

Przykłady metod operujących na kolekcjach danych w `EventManager` to np. **`addEvent`** i **`deleteEvent`**, które umożliwiają dodawanie i usuwanie spotkań z kolekcji `ObservableMap`. Dzięki wykorzystaniu kolekcji – mapy, zapewniona jest unikalność kluczy (dat), jednak takie podejście ogranicza możliwość przechowywania wielu spotkań na jeden dzień.

```
private final ObservableMap<String, WeatherForecast> weatherLocations;
```

Rys. Kolekcja danych w `WeatherManager`.

W przypadku **`WeatherManager`**, kolekcja przechowuje prognozy pogodowe, przypisane do lokalizacji. Pozwala to na elastyczne zarządzanie prognozami, zapewniając unikalność danych dla każdej lokalizacji.

3.2 Zapis/odczyt stanu aplikacji

```
@Override
public void init() {
    System.out.println("Inicjalizacja aplikacji");
    EventManager eventManager = EventManager.getInstance();
    try {
        eventManager.loadEventsState();
        this.autoWeatherService = new AutoWeatherService();
    }
    catch(GlobalStateException ex) {}
}
```

Rys. Metoda `init` w `AppService`.

Metoda **`init`** to metoda inicjalizacyjna aplikacji, wykonująca się po stworzeniu instancji klasy `Application`. Ładowany jest stan klasy singleton `EventManager` metodą `loadEventsState`. Zapis stanu tej kolekcji (`saveEventsState`) występuje w metodzie `stop`, która wykonuje się po zamknięciu aplikacji, czyli gdy ostatnie otwarte okno zostanie zamknięte lub gdy wywołano metodę Platform.exit().

```
public final void loadEventsState() throws GlobalStateException {
    this.events = FXCollections.observableMap(GlobalStateAssistant.loadState(AppConstants.PATH_EVENTS_STATE));
}
public final void saveEventsState() throws GlobalStateException {
    GlobalStateAssistant.saveState(new HashMap<>(this.events), AppConstants.PATH_EVENTS_STATE);
}
```

Rys. Zapis/odczyt stanu w `EventManager`

Metody **`loadEventsState`** i **`saveEventsState`** służą do zapisywania i odczytywania stanu kolekcji `events` znajdującej się w `EventManager` za pomocą klasy `GlobalStateAssistant`. Ponieważ `ObservableMap` nie jest serializowalne, jest przekształcane na `HashMap` przed zapisaniem. Choć dostęp do tych metod jest globalny, lepszym rozwiązaniem byłoby przeniesienie odpowiedzialności za zapis i odczyt do dedykowanego serwisu.

Metody **`loadState`** i **`saveState`** w `GlobalStateAssistant` obsługują serializację i deserializację obiektów do/z pliku binarnego, upraszczając operacje wejścia-wyjścia i eliminując potrzebę zarządzania strumieniami plików w innych częściach aplikacji.

3.3 Modele danych

Modele **`ScheduledEvent`** i **`WeatherDay`** wykorzystują mechanizm właściwości „**`Property`**”, co pozwala na łatwą integrację z UI oraz automatyczną aktualizację interfejsu w przypadku zmian danych.

```
private transient StringProperty eventName;
private transient StringProperty eventDesc;
private transient StringProperty location;
```

Rys. Atrybuty `StringProperty` z klasy `ScheduledEvent`.

Aby umożliwić **deserializacje** danych pogodowych z api za pomocą biblioteki Gson do postaci `StringProperty`, konieczne było stworzenie dedykowanego konwertera (klasa `WeatherDayAdapter`).

Aby umożliwić zapis właściwości klasy `ScheduledEvent` do pliku binarnego (z użyciem obiektu klasy `ObjectOutputStream`), opracowano dodatkową metodę `writeObject` (w klasie `ScheduledEvent`).

Dodatkowo aby uniknąć problemów z serializacją, atrybuty oznaczone zostały jako `Transient`.

```
public class WeatherForecast {
    private final Map<LocalDate, WeatherDay> weatherDays = new HashMap();
```

Rys. Klasa kontenera przechowująca dane pogodowe.

`WeatherForecast` jest klasą kontenera do przechowywania prognoz pogodowych, umożliwiającą łatwe zarządzanie i dostęp do danych. Udostępnia także metody do operowania na tej kolekcji. W przeciwieństwie do klas typu Manager, nie zaimplementowano w niej wzorca singleton.

```
private final LocalDate date;
private final int column;
private final int row;
private final CalendarButton dayButton;
private final String initialText;
```

Rys. Atrybuty klasy `CalendarItem` reprezentujące pojedyńczy element kalendarza.

`CalendarItem` przechowuje dane dotyczące pojedynczego elementu kalendarza, takie jak `date`, `column`, `row`, `CalendarButton`, oraz `initialText`, co ułatwia manipulację elementami kalendarza.

3.4 Uruchamianie nowych okien w aplikacji

```
public <T> T openNewStage(String fxmlPath, String title, boolean isModal,
        double minWidth, double minHeight) throws IOException
{
    FXMLLoader loader = new FXMLLoader(getClass().getResource(fxmlPath));
    Parent root = loader.load();
    Scene scene = new Scene(root);
    this.loadCssStylesheet(scene);
    Stage stage = new Stage();
    stage.setTitle(title);
    stage.setScene(scene);
    stage.setMinWidth(minWidth);
    stage.setMinHeight(minHeight);
    if (isModal) {
        stage.initModality(Modality.APPLICATION_MODAL);
    }
    stage.show();
    return loader.getController();
}
```

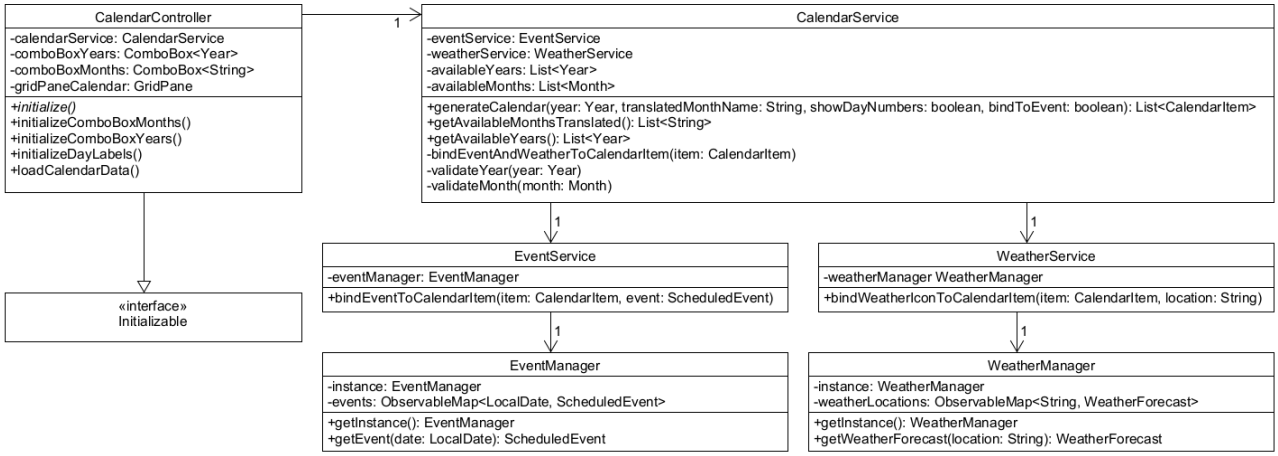
Rys. Metoda klasy `StageAssistant`, służąca do otwierania nowych okien w aplikacji.

Metoda **`openNewStage`** w klasie `StageAssistant` służy do dynamicznego otwierania nowych okien w aplikacji. Ładuje plik FXML przy użyciu `FXMLLoader`, tworzy obiekt `Scene`, ustawia tytuł, wymiary, tryb modalny (jeśli określony) oraz ładuje arkusz stylów CSS. Zwraca kontroler powiązany z danym widokiem, umożliwiając dalsze operacje z poziomu pierwotnego kontrolera.

```
@Override
public void start(Stage stage) throws IOException {
    StageAssistant.getInstance().openNewStage(
            AppConstants.PATH_FXML_CALENDAR,
            AppConstants.CALENDAR_STAGE_NAME,
            true,
            AppConstants.CALENDAR_STAGE_MIN_WIDTH,
            AppConstants.CALENDAR_STAGE_MIN_HEIGHT
    );
}
```

Rys. Przykład wykorzystania metody `openNewStage` w metodzie `start` głównej klasy aplikacji .

3.5 Generowanie kalendarza



Rys. Uproszczony diagram klas związanych z generowaniem kalendarza.

Na rysunku powyżej zamieszczono uproszczony diagram klas związanych z generowaniem kalendarza. Na diagramie zamieszczono wyłącznie najważniejsze metody i atrybuty klas związane z tą operacją.

3.5.1 Implementacja metody loadCalendarData w CalendarController

```
@FXML
public void loadCalendarData(){
    Year year = this.comboBoxYears.getSelectionModel().getSelectedItem();
    String month = this.comboBoxMonths.getSelectionModel().getSelectedItem();
    this.cleanCalendar();
    try {
        this.calendarService.generateCalendar(year, month, true, true).forEach(item -> {
            CalendarButton button = item.getButton();
            button.setOnAction(e -> calendarButton_click(item));
            this.gridPaneCalendar.add(button, item.getColumn(), item.getRow());
        });
    }
    catch(ApiException ex) {
        new AlertException(ex).showAndWait();
    }
}
```

Rys. Metoda klasy `CalendarController`, służąca do wyświetlenia kalendarza w interfejsie użytkownika.

Metoda **`loadCalendarData`** odpowiada za ładowanie danych do kalendarza. Najpierw pobiera wybrany rok i miesiąc z odpowiednich `comboboxów`. Następnie czyści istniejący kalendarz, a następnie generuje nowy kalendarz przy pomocy serwisu `calendarService`. Dla każdego elementu w wygenerowanej liście (reprezentującej dni miesiąca) tworzony jest przycisk `CalendarButton`, który po kliknięciu wywołuje metodę `calendarButton_click`. Przyciski są dodawane do `gridPaneCalendar` na odpowiednich kolumnach i wierszach. W przypadku błędu podczas generowania kalendarza (np. problem z API), rzucany jest wyjątek, który jest obsługiwany i wyświetlany za pomocą `AlertException`.

3.5.2 Implementacja metody generateCalendar w CalendarService

```
public List<CalendarItem> generateCalendar(Year year, Month month, boolean showDayNumbers,
        boolean bindToEvent) throws ApiException {
    this.validateYear(year);
    this.validateMonth(month);

    List<CalendarItem> calendarItems = new ArrayList<>();
    int daysInMonth = month.length(year.isLeap());
    int shift = LocalDate.of(year.getValue(), month, 1).getDayOfWeek().getValue() - 1;
    int row = 1;

    for (int col = 1 + shift; col <= daysInMonth + shift; col++) {
        int day = col - shift;
        LocalDate date = LocalDate.of(year.getValue(), month, day);
        CalendarItem item = createCalendarItem(date, col, row, showDayNumbers);
        if (bindToEvent) {
            this.bindEventAndWeatherToCalendarItem(item);
        }
        calendarItems.add(item);
        if (col % 7 == 0) {
            row++;
        }
    }
    return calendarItems;
}
```

Rys. Metoda klasy `CalendarService`, służąca do generowania kalendarza.

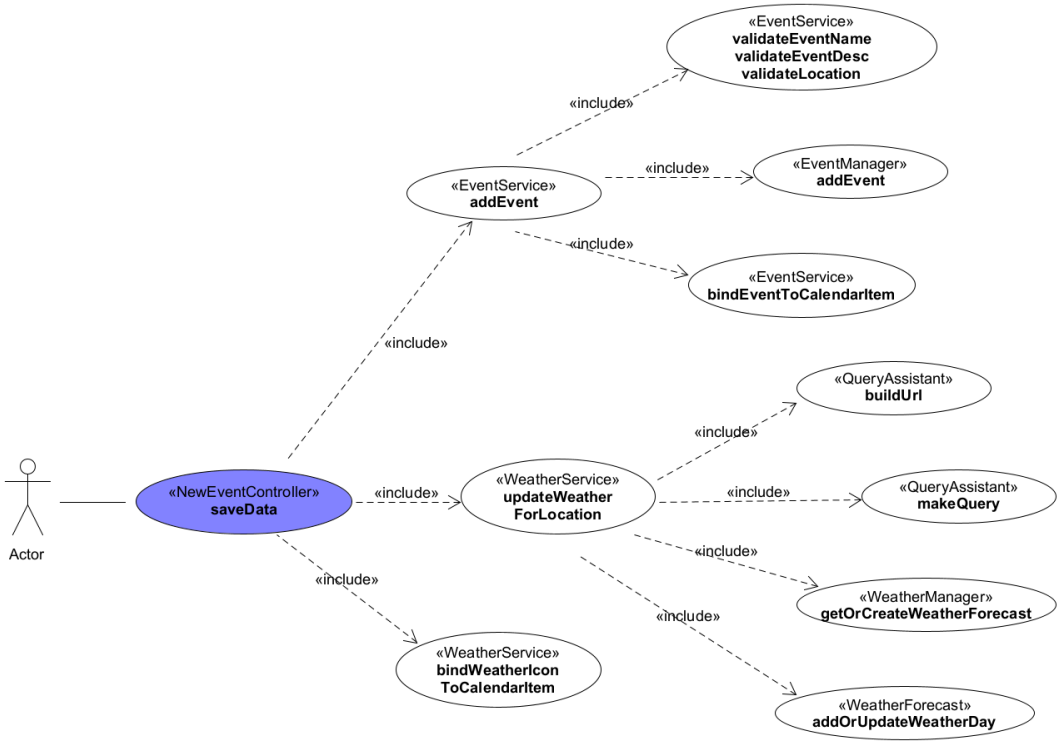
Metoda **`generateCalendar`** tworzy strukturę kalendarza dla określonego roku i miesiąca, generując listę obiektów `CalendarItem` reprezentujących dni miesiąca. Po walidacji parametrów `year` i `month`, inicjalizowana jest lista `calendarItems`. Zmienna `daysInMonth` przechowuje liczbę dni w miesiącu, a `shift` określa przesunięcie pierwszego dnia w tygodniu.

Pętla iteruje po dniach miesiąca, tworząc obiekt `LocalDate` dla każdego dnia i generując `CalendarItem` przy pomocy metody `createCalendarItem`. Jeśli `bindToEvent` jest ustawione na `true`, metoda **`bindEventAndWeatherToCalendarItem`** łączy dzień z wydarzeniami i prognozami pogodowymi wykorzystując metody zaimplementowane w innych serwisach. Po dodaniu `CalendarItem` do listy, metoda sprawdza czy należy przejść do nowego tygodnia (przy wielokrotności 7 w kolumnie). Na końcu zwraca listę `calendarItems` do wyświetlenia w UI.

```
private void bindEventAndWeatherToCalendarItem(CalendarItem item) {
    try {
        ScheduledEvent event = this.eventService.getEvent(item);
        if (event != null) {
            this.eventService.bindEventToCalendarItem(item, event);
            this.weatherService.bindWeatherIconToCalendarItem(item, event.getLocation());
        }
    } catch (Exception ex) {}
}
```

Rys. Metoda wiązania właściwości elementu kalendarza z spotkaniami oraz pogodą.

3.6 Proces dodawania spotkania



Rys. Diagram use-case: dodawanie spotkania przez użytkownika w NewEventController.

3.6.1 Implementacja metody saveData w NewEventController

```
@FXML
private void saveData() {
    try {
        String location = this.textFieldEventLocation.getText();
        String eventName = this.textFieldEventName.getText();
        String eventDesc = this.textAreaEventDesc.getText();
        this.eventService.addEvent(selectedItem, eventName, eventDesc, location);
        this.weatherService.updateWeatherForLocation(location);
        this.weatherService.bindWeatherIconToCalendarItem(selectedItem, location);

        Stage currentStage = (Stage) buttonAddEvent.getScene().getWindow();
        currentStage.close();
        new AlertSucces("Pomyślnie udało się zapisać nowe spotkanie.").showAndWait();
     }
    catch(Exception ex) {
        new AlertException(ex).showAndWait();
    }
}
```

Rys. Metoda klasy `NewEventController` służąca do zapisu spotkania oraz aktualizacji pogody.

Metoda **`saveData`** jest wywoływana po kliknięciu przycisku "**Zapisz spotkanie**" w kontrolerze `NewEventController` i odpowiada za dodanie nowego spotkania oraz zaktualizowanie związanych z nim danych pogodowych. Użytkownik wprowadza dane wydarzenia (lokalizację, nazwę, opis) w odpowiednich polach tekstowych.

Po kliknięciu przycisku, metoda pobiera te dane, a następnie wywołuje metodę **`addEvent`** z serwisu `eventService`, aby zapisać nowe spotkanie w systemie.

Równocześnie, dla podanej lokalizacji, metoda **`updateWeatherForLocation`** w serwisie

`weatherService` pobiera i aktualizuje dane pogodowe. Dodatkowo, metoda `bindWeatherIconToCalendarItem` aktualizuje ikonę pogody powiązaną z wydarzeniem w kalendarzu.

Po zakończeniu tych operacji, aktualne okno (widok) zostaje zamknięte, a użytkownik otrzymuje powiadomienie o pomyślnym zapisaniu spotkania za pomocą komunikatu wyświetlanego przez `AlertSucces`. Jeśli wystąpi jakikolwiek błąd, zostanie wyświetlony komunikat o błędzie za pomocą `AlertException`.

3.6.2 Implementacja metody addEvent w EventService

```
public void addEvent(CalendarItem item, String eventName, String eventDesc, String location) {
    this.validateCalendarItem(item);
    this.validateEventName(eventName);
    this.validateEventDesc(eventDesc);
    ScheduledEvent newEvent = new ScheduledEvent(eventName, eventDesc, location);
    this.eventManager.addEvent(item.getDate(), newEvent);
    this.bindEventToCalendarItem(item, newEvent);
}
```

Rys. Dodawanie spotkania w serwisie `EventService`

Metoda **`addEvent`** w `EventService` dodaje nowe wydarzenia, wykonując walidację danych (np. `eventName`, `eventDesc`). Po walidacji tworzony jest obiekt `ScheduledEvent`, który jest przekazywany do `EventManager` i powiązany z odpowiednim `CalendarItem` w kalendarzu.

Metoda **`bindEventToCalendarItem`** przypisuje właściwości `eventNameProperty` do właściwości przycisku kalendarza, co pozwala na automatyczne aktualizacje tekstu kalendarza, gdy nazwa spotkania jest zmieniana.

```
private void validateEventDesc(String eventDesc) {
    if(eventDesc.length() > 500) {
        throw new IllegalArgumentException("Opis spotkania nie może przekraczać 500 znaków.");
    }
}
```

Rys. Przykładowa walidacja danych w klasie `EventService`.

W `EventService` wykonywana jest **walidacja danych**, np. sprawdzenie długości `eventDesc`. W przypadku błędu rzucany jest wyjątek `IllegalArgumentException`.

Edytowanie i usuwanie spotkań działa podobnie jak dodawanie. **Edytowanie spotkań** obejmuje walidację, a zmiany są zapisywane w istniejącym obiekcie `ScheduledEvent`. **Usuwanie spotkań** wymaga weryfikacji istnienia spotkania w kolekcji, a następnie jego usunięcia.

3.6.3 Implementacja metody updateWeatherForLocation w WeatherService

```
public void updateWeatherForLocation(String location) throws ApiException {
        WeatherQuery result;
         try {
             String url = QueryAssistant.buildUrl(this.apiBaseUrl, location, this.apiQueryParams);
             result = QueryAssistant.makeQuery(url, WeatherQuery.class);
         }
         catch(ApiException ex) {
           throw new ApiException(
                   "Wystąpił błąd podczas aktualizowania danych pogodowych dla lokalizacji: " + location, ex);
         }

         WeatherForecast weatherLocation = this.weatherManager.getOrCreateWeatherForecast(location);

         for(WeatherDay weatherDay : result.getDays()) {
             weatherLocation.addOrUpdateWeatherDay(weatherDay);
         }
}
```

Rys. Metoda aktualizacji pogody w `WeatherService`.

Metoda **`updateWeatherForLocation`** pobiera dane pogodowe dla określonej lokalizacji. Wykonuje zapytanie HTTP do API, korzystając z metody `makeQuery` z klasy `QueryAssistant`. Url jest tworzony metodą `buildUrl` z `QueryAssistant`. Odpowiedź jest deserializowana do obiektu `WeatherQuery`. Po pobraniu danych, metoda aktualizuje prognozę pogodową dla danej lokalizacji w `WeatherManager`, dodając lub aktualizując dane w `WeatherForecast`.

```
public static <T> T makeQuery(String url, Class<T> classOfT) throws ApiException {
```

Rys. Metoda pomocnicza do zapytania HTTP w `QueryAssistant`.

Metoda **`makeQuery`** wykonuje zapytanie HTTP, używając biblioteki `Gson` do deserializacji odpowiedzi na obiekt `WeatherDay`.

W aplikacji zaimplementowano także automatyczne aktualizacje pogody. Zajmuje się tym klasa **`AutoWeatherService`**, która wykorzystuje w tym celu klasę `ScheduledExecutorService` (java.util.concurrent).

Klasa `AutoWeatherService` w celu aktualizacji pogody wykorzystuje serwis `WeatherService` (opisaną wyżej metodę `updateWeatherForLocation`).

3.7 Wyświetlanie informacji o spotkaniu, pogodzie

```
private void setControls() {
        ScheduledEvent event = this.eventService.getEvent(this.selectedItem);

        this.labelEventName.textProperty().bind(event.eventNameProperty());
        this.labelEventDesc.textProperty().bind(event.eventDescProperty());
        this.labelLocation.textProperty().bind(event.locationProperty());
        this.textFieldEditEventName.setText(event.getEventName());
        this.textAreaEditEventDesc.setText(event.getEventDesc());
        this.textFieldEditEventLocation.setText(event.getLocation());

        WeatherDay weather = this.weatherService.getWeatherDay(this.selectedItem, event.getLocation());

        this.labelDatetime.textProperty().bind(weather.datetimeProperty());
        this.labelTemperature.textProperty().bind(weather.tempProperty().asString());
        this.labelHumidity.textProperty().bind(weather.humidityProperty().asString());
        this.labelPrecip.textProperty().bind(weather.precipProperty().asString());
        this.labelPrecipprob.textProperty().bind(weather.precipprobProperty().asString());
        this.labelSnow.textProperty().bind(weather.snowProperty().asString());
        this.labelPressure.textProperty().bind(weather.pressureProperty().asString());
        this.labelCloudcover.textProperty().bind(weather.cloudcoverProperty().asString());
        this.labelSunrise.textProperty().bind(weather.sunriseProperty());
        this.labelSunset.textProperty().bind(weather.sunsetProperty());
        this.labelConditions.textProperty().bind(weather.conditionsProperty());
        this.labelDescription.textProperty().bind(weather.descriptionProperty());
        this.imageView.setImage(new Image("img/weather-icon-trsp/" + weather.getIcon() + ".png"));
}
```

Rys. Metoda klasy `EventController`, służąca do wyświetlania informacji o spotkaniu i pogodzie w interfejsie użytkownika.

Metoda **`setControls`** jest wywoływana podczas inicjalizacji kontrolera `EventController` oraz po edytowaniu spotkania i służy do ustawiania danych dla widoków związanych z wydarzeniem oraz prognozą pogody. Na początku metoda pobiera obiekt `ScheduledEvent` za pomocą `eventService`, bazując na wybranym elemencie `selectedItem`, który pochodzi z klasy `CalendarController`. Wartości właściwości tego wydarzenia (nazwa, opis, lokalizacja) są bindowane do odpowiednich etykiet w interfejsie użytkownika. Następnie metoda ustawia wartości pól tekstowych (`TextField` i `TextArea`) na aktualne dane wydarzenia.

Po pobraniu danych pogodowych dla danego wydarzenia i lokalizacji z `weatherService`, właściwości pogodowe (takie jak data, temperatura, wilgotność, opady, itp.) są bindowane do etykiet w interfejsie użytkownika. Na koniec, na podstawie ikony prognozy, wczytywana jest odpowiednia grafika, która jest przypisywana do widoku obrazu. Dzięki temu wszystkie dane są automatycznie aktualizowane w interfejsie po ich załadowaniu.

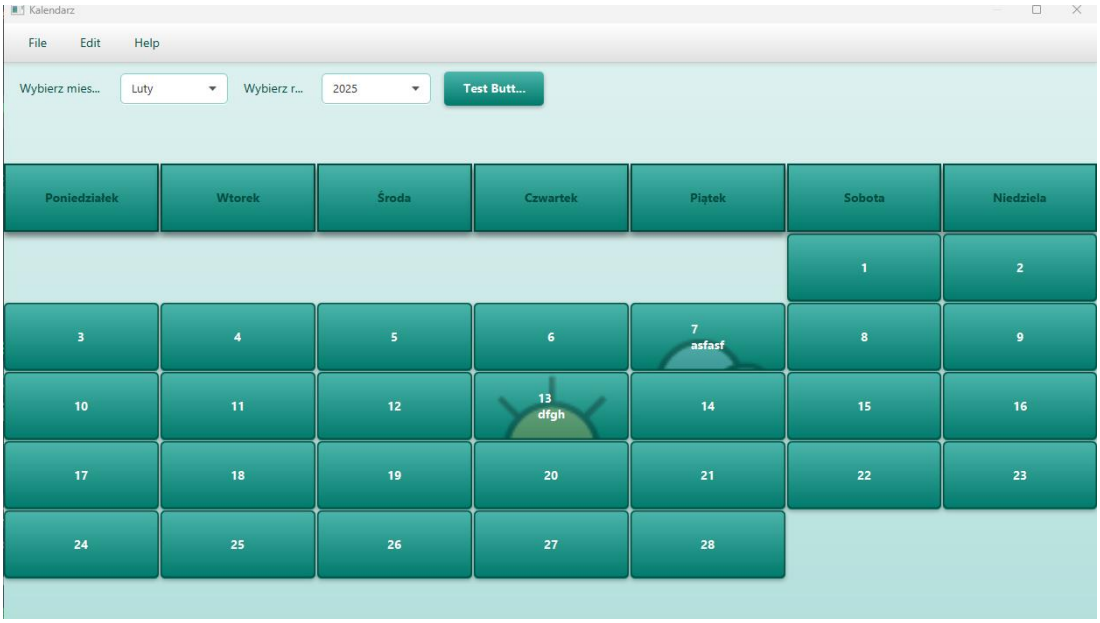
4 Instrukcja obsługi/działania aplikacji

1. Uruchomienie aplikacji

- należy kliknąć dwukrotnie plik Weather-Calendar.jar.

Aplikacja otworzy główne okno kalendarza.

- należy wybrać miesiąc oraz rok kalendarza, w którym będziemy chcieli przypisać spotkanie.

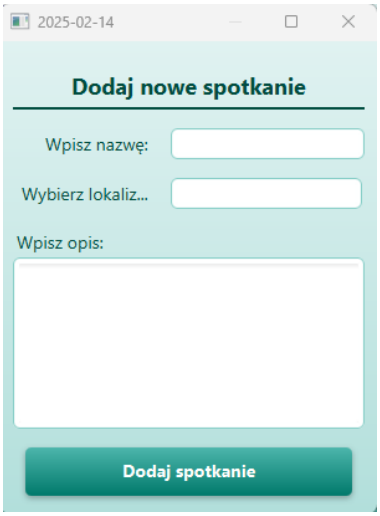


Rys. Okno kalendarza (kontroler: `CalendarController`).

2. Dodawanie nowego wydarzenia

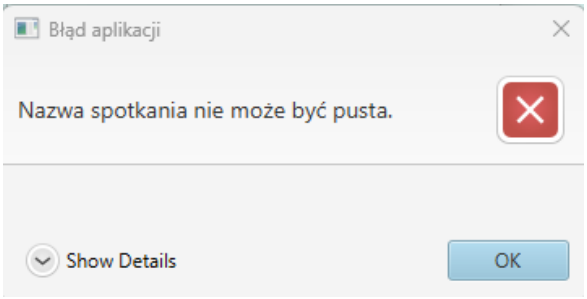
- należy kliknąć w dowolny pusty element kalendarza.
- następnie wpisać nazwę, lokalizację oraz opcjonalnie opis spotkania.
- ostatecznie kliknąć przycisk "Dodaj spotkanie".

Wydarzenie pojawi się w kalendarzu z ikoną pogodową.



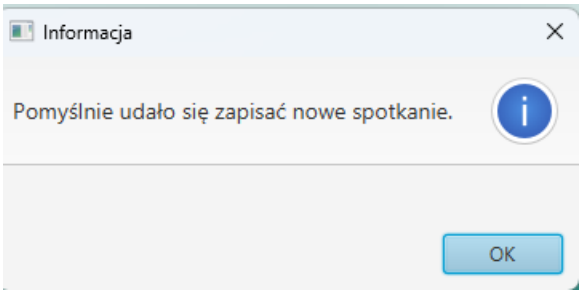
Rys. Okno dodawania nowego spotkania (kontroler: `NewEventController`) .

3. W przypadku gdy podamy niepoprawne dane, otrzymamy stosowny komunikat, który poinformuje nas o błędzie.



Rys. Okno wyświetlające błąd aplikacji (klasa `AlertError`).

4. Jeśli uda się dodać nowe spotkanie, dostaniemy komunikat informujący o sukcesie operacji.



Rys. Okno informacyjne (klasa: `AlertSucces`).

5. Edycja i usuwanie wydarzenia

- należy wybrać nowo dodany element kalendarza.

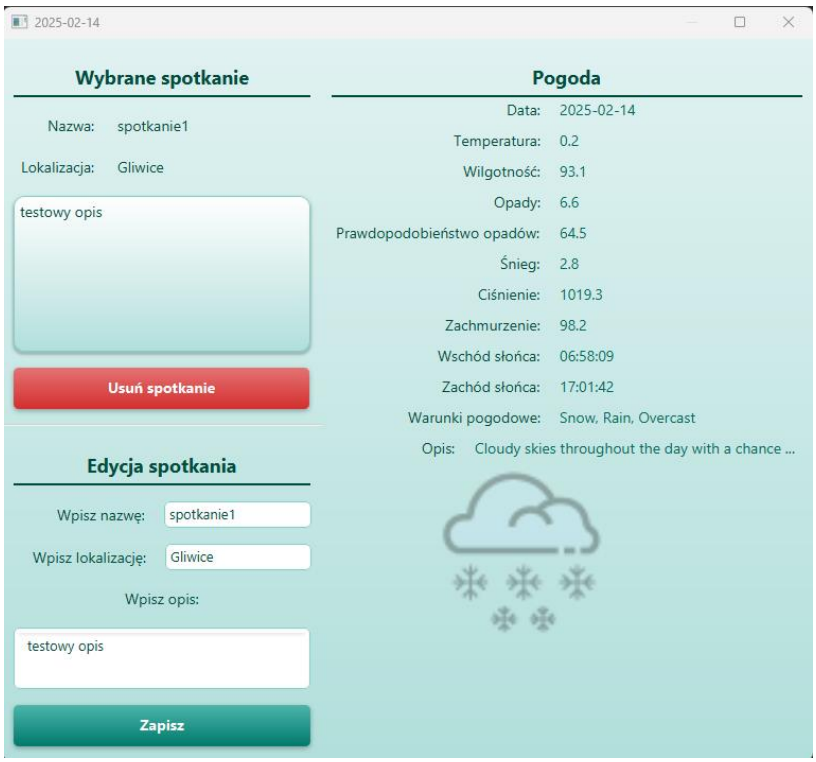
- można zmodyfikować dane i kliknąć "Zapisz", lub usunąć wydarzenie.

- w przypadku wystąpienia błędu bądź gdy operacja zakończy się sukcesem, otrzymamy stosowny komunikat.

6. Wyświetlanie spotkania oraz prognozy pogody

- klikając na dzień w kalendarzu, zobaczymy także szczegóły pogody (temperatura, opady, zachmurzenie itp.) oraz informacje o przypisanym spotkaniu.

- dane pogodowe są aktualizowane na podstawie lokalizacji spotkania.



Rys. Okno wyświetlania informacji o spotkaniu oraz pogodzie, umożliwiające modyfikowanie spotkania (kontroler: `EventController`).

7. Zapisywanie i wczytywanie danych

- dane zapisywane są automatycznie podczas zamykania aplikacji.

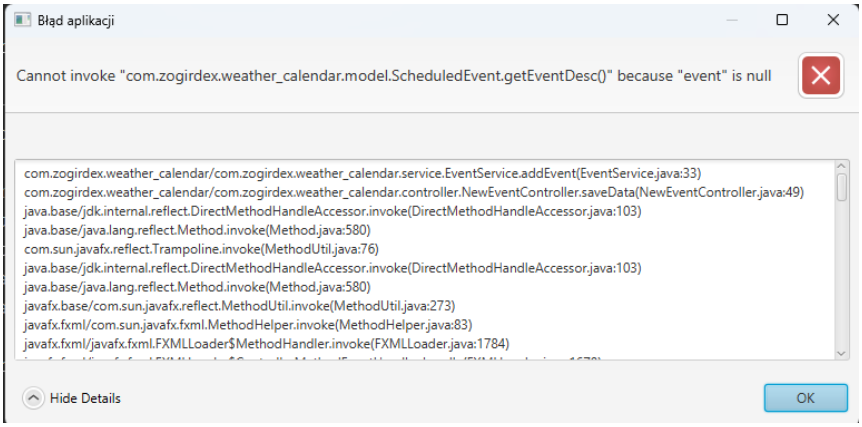
8. Rozwiązywanie problemów

- Brak danych pogodowych - sprawdź połączenie z internetem, bądź upewnij się, że lokalizacja spotkania jest prawidłowa.

- Aplikacja nie uruchamia się - upewnij się, że masz zainstalowaną Javę.

- Błąd podczas dodawania wydarzenia - sprawdź, czy wszystkie pola są wypełnione.

- W przypadku wystąpienia nieoczekiwanego błędu należy skontaktować się z administratorem aplikacji



Rys. Okno wyświetlające przechwycone wyjątki, oraz nieoczekiwane błędy (klasa: `AlertException`).

5 Podsumowanie i wnioski

Aplikacja posiada **mechanizmy obsługi błędów**, zapewniając użytkownikowi jasne komunikaty przy problemach z API czy spotkaniami, co poprawia stabilność i doświadczenie użytkownika.

Mechanizm wiązania danych z interfejsem użytkownika pozwala na automatyczną aktualizację widoków po zmianach, co zapewnia płynność działania i spójność wizualną aplikacji.

Komunikacja z zewnętrznymi API została poprawnie zaimplementowana i może być rozszerzana o dodatkowe źródła danych.

Obecnie aplikacja zapisuje stan w metodzie „stop" podczas normalnego zamknięcia, co może prowadzić **do utraty danych w przypadku nieoczekiwanego wyłączenia**. Aby temu zapobiec, sekcja rozważała dodanie listenera do mapy `ObservableMap` spotkań, który umożliwi zapisywanie danych na bieżąco, np. do bazy danych SQL lub plików JSON.

Kolejnym problemem jest **przechowywanie klucza API w kodzie aplikacji**. Rozwiązaniem mogłoby być stworzenie własnego REST API, które pełniłoby rolę pośrednika, przechowując klucze tylko na zewnętrznym serwerze. W przyszłości interfejs mógłby zostać rozszerzony o autoryzację użytkowników oraz przesyłanie danych o spotkaniach do serwerowej bazy danych.

Brak mechanizmu walidacji lokalizacji wynika z problemów z dokładnością oraz ogromną liczbą możliwych miejsc. API pogodowe obsługuje te lokalizacje, więc walidacja po stronie aplikacji mogłaby prowadzić do błędnych odrzuceń.

Aplikacja wyświetla prognozy w ujęciu dobowym, ale w przyszłości planujemy dodać **prognozę godzinową**, co wymaga rozbudowy obecnej struktury danych. Obecna architektura jest jednak przystosowana na te zmiany.

Sekcja planowała także dodanie **mechanizmu logowania błędów** przy użyciu loggera, który rejestrowałby wszystkie wyjątki i problemy aplikacji do plików tekstowych. Dzięki temu możliwe byłoby łatwiejsze diagnozowanie błędów oraz analiza ich przyczyn.

Dodatkowo, rozważano wprowadzenie **plików konfiguracyjnych** umożliwiających definiowanie ustawień aplikacji, takich jak klucze API, domyślna lokalizacja użytkownika czy interwały odświeżania danych pogodowych. Mechanizm ładowania konfiguracji pozwoliłby na łatwiejsze dostosowywanie aplikacji bez konieczności ingerencji w kod.

Podsumowując, udało się zrealizować podstawowe założenia aplikacji, a zaprojektowana architektura pozwala na dalszy rozwój.

6 Spis źródeł

- [JavaFX, FXML]: https://openjfx.io/
- [Java 23]: https://docs.oracle.com/en/java/javase/23/
- [Maven]: https://maven.apache.org/
- [SceneBuilder]: https://gluonhq.com/products/scene-builder/
- [Diagramy UML]: https://www.umlet.com/
- [Visual Crossing Weather API]: https://www.visualcrossing.com/weather-api
- [Gson Documentation]: https://github.com/google/gson