

Final project documentation

Event-driven programming

Katarzyna Zaleska, WCY19IJ1S1

21.06.2022

Contents

1	Introduction	2
2	Used technologies	2
3	Requirements	2
3.1	Multithreading - example of use	2
3.2	Asynchronous programming	3
3.3	Custom events	3
3.4	Using data taken from two external web services	3
3.5	Database	4
3.6	Config properties	5
3.7	Custom graphical components	5
3.8	Event-bus	6
3.9	Apache Commons	7
3.10	Example of use Singleton	8
4	Important dependencies	8
5	Final effect	8
6	Useful sources	12
7	Conslusions	12

1 Introduction

The WordCards application was created to support learning - mainly English. The user has the ability to create their own private account. After logging in, a dashboard is displayed - random advice, information about the user's account. The side menu allows you to switch scenes and check other functionality such as checking pronunciation or adding new words.

2 Used technologies

The application was written in Java SE programming language. JavaFX was used to create a graphical user interface and PostgreSQL as a database relational management system. All additional libraries were added using maven dependencies. Some data such as the database password has been added to the app.properties file to avoid harcoding this data in code. The Properties-Manager class allows you to read data from this file.

3 Requirements

3.1 Multithreading - example of use

There is a Label on the user's dashboard that contains advice. Every 5 seconds, the displayed label changes - this change occurs 10 times. The invokeCounterTask function is called in the initialize function of the DashboardController.

```
private void invokeCounterTask() {
    AdviceTask task = new AdviceTask(10);
    task.valueProperty().addListener((observable, oldValue, newValue) -> {
        randomAdviceLabel.setText(newValue);
    });

    Thread thread = new Thread(task);
    thread.setDaemon(true);
    thread.start();
}
```

AdviceTask class:

```
public class AdviceTask extends Task<String> {
    private final long limit;

    public AdviceTask(long limit) {this.limit = limit;}

    @Override
    protected String call() throws Exception {
        Slip adviceText = null;
        long count = 0;
        while (count < limit) {
            adviceText = generateAdvice().getSlip();
            Thread.sleep(5000);
            count++;
            updateValue("#" + adviceText.getId() + ": " + adviceText.getAdvice());
        }
        return "#" + adviceText.getId() + ": " + adviceText.getAdvice();
    }

    private Advice generateAdvice() {
        // code to return new advice
    }
}
```

3.2 Asynchronous programming

Asynchronous programming was used to send requests to receive information about a word. Once the relevant information is received, it is processed and presented to the user. This prevents the program from blocking as a result of waiting for a potentially long-running task.

```
CompletableFuture<Word> future = CompletableFuture.supplyAsync(() -> {
    // some code to get information about word
    return words;
}).thenApply(word -> {
    Platform.runLater(() -> {
        // make update of gui
    });
    return word[0];
});
```

3.3 Custom events

I created two custom events - LoginEvent and RegisterMsgEvent. The first one is called during login. The second runs in the background and displays the message on the registration screen. Example of handling a custom event:

```
loginButton.setOnAction(new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent event) {
        String username = usernameTextField.getText();
        String password = passwordPasswordField.getText();

        LoginUser loginUser = new LoginUser(username, password);
        loginButton.fireEvent(new LoginEvent(LoginEvent.LOGIN_USER_SAVE, loginUser));
    }
});

loginButton.addEventHandler(LoginEvent.ANY, this::loginButtonOnAction);
```

3.4 Using data taken from two external web services

[Advice API](#) gives you the opportunity to receive random advice in English. A special Task sends a certain number of requests every few seconds and changes the displayed advice on the user dashboard.

[Dictionary API](#) provides information about the word - pronunciation, link to a sample recording and much more.

I used POJO classes to store the acquired data and to organize the code. Based on the [jsonschema2pojo](#) website, I generated the appropriate classes. Below is an example of sending a request to the API to generate a new advice.

```
private Advice generateAdvice() {
    URL url = null;
    try {
        url = new URL(ADVICE_API_URL);
    } catch (Exception e) {
        e.printStackTrace();
    }

    InputStreamReader reader = null;
    try {
        reader = new InputStreamReader(url.openStream());
    } catch (Exception e) {
        e.printStackTrace();
    }

    Advice advice = new Gson().fromJson(reader, Advice.class);
}
```

```
        return advice;
    }
}
```

3.5 Database

I initially created a persistence.xml file, which is the central piece of configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="pu">
        <description>Jpa</description>
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <properties>
            // properties
        </properties>
    </persistence-unit>
</persistence>
```

The next important step was to create the appropriate entities. In my case, the User and Card classes. I used the Repository Design Pattern to create appropriate methods for the database data. Below is an example of using this design pattern for the User class.

Interface UserRepository:

```
public interface UserRepository {
    User getUserById(Long id);
    User getUserByUsername(String username);
    User saveUser(User user);
    void deleteUser(User user);
}
```

Implementing the repository with JPA and Hibernate:

```
public class UserRepositoryImpl implements UserRepository {
    private final EntityManager em;

    @Inject
    public UserRepositoryImpl(EntityManager em) {
        this.em = em;
    }

    @Override
    public User getUserById(Long id) {
        return em.find(User.class, id);
    }

    @Override
    public User getUserByUsername(String username) {
        TypedQuery<User> query = em.createQuery("SELECT u FROM User u WHERE
            u.username = :username", User.class);
        query.setParameter("username", username);
        return query.getSingleResult();
    }

    @Override
    public User saveUser(User user) {
        em.getTransaction().begin();
        em.persist(user);
        em.getTransaction().commit();
        return user;
    }
}
```

```

    }

    @Override
    public void deleteUser(User user) {
        em.getTransaction().begin();
        em.remove(user);
        em.getTransaction().commit();
    }
}

```

The implementation of the repository design pattern made it possible to store all queries in a single file and prevented code repetition.

3.6 Config properties

The app.properties file contains data used in the application (passwords, names). PropertiesManager class uses Singleton Design Pattern to ensure that only one instance of this class is created. Example of reading data from a file:

```

String DBASE_NAME = PropertiesManager.getInstance().getProperty("DBASE_NAME");
String DBASE_URL = PropertiesManager.getInstance().getProperty("DBASE_URL");

```

Content of app.properties file:

```

DBASE_NAME=<YOUR_DBASE_NAME>
DBASE_URL=<YOUR_DBASE_URL>
DBASE_USERNAME=<YOUR_DBASE_USERNAME>
DBASE_PASSWORD=<YOUR_DBASE_PASSWORD>
MAIL_NAME=<YOUR_MAIL_NAME>
MAIL_PASSWORD=<YOUR_MAIL_PASSWORD>
WORD_API=https://api.dictionaryapi.dev/api/v2/entries/en/
ADVICE_API_URL=https://api.adviceslip.com/advice

```

3.7 Custom graphical components

The CustomLabel class extends the Label class. The changeColorLabelText method changes the color of the text (shades of blue) every second 4 times.

```

public class CustomLabel extends Label {
    private ArrayList<String> colors = new ArrayList<String>(Arrays.asList("#145DA0",
        "#0C2D48", "#2E8BC0", "#B1D4E0"));
    private int colorIndex = 0;

    public CustomLabel() {
        this("");
    }

    public CustomLabel(String text) {
        super(text);
        changeColorLabelText();
    }

    private void changeColorLabelText() {
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                if(colorIndex < colors.size()) {
                    setTextFill(Color.web(colors.get(colorIndex)));
                    colorIndex++;
                } else {
                    timer.cancel();
                    timer.purge();
                }
            }
        })
    }
}

```

```

    }, 0, 1000);
}
}

```

A search button was created by extending the Button class. Its styles change when hovered over, pressed.

```

public class SearchButton extends Button {
    private ImageView imageView;

    public SearchButton() {
        super("Search");
        addImageView();
        updateStyles();
        registerListeners();
    }

    private void registerListeners() {
        setOnMouseEntered(event -> {stylesOnMouseEntered();});
        setOnMouseClicked(event -> {stylesOnMouseClicked();});
        setOnMouseExited(event -> {updateStyles();});
    }

    private void addImageView() {
        imageView = new
            ImageView(getClass().getResource("/images/icons/loupe.png").toExternalForm());
        imageView.setFitHeight(16);
        imageView.setFitWidth(16);

        setGraphic(imageView);
    }

    \\ methods to change styles
}

```

Changing the css styles allowed me to get additional effects for the CheckBox. All used CSS styles are located in the resources/custom-controls-css folder.



Figure 1: Custom CheckBox.

3.8 Event-bus

I used the Guava library to create the EventBus. Instantiating the EventBus class and registering listener. I've also registered a DeadEventListener class (if there is an event type which there are no subscribers).

```

EventBus eventBus = new EventBus();
EventListener eventListener = new EventListener();

eventBus.register(eventListener);
eventBus.register(new DeadEventListeners());

```

EventListener class:

```

public class EventListener {
    @Subscribe
    public void loginEventListener(LoginEvent event) {
        AlertDialog alertDialog = new AlertDialog();
        alertDialog.setDialogTitle("WordCards");
        alertDialog.setHeader("Hi " + event.getPerson().getUsername() + "!");
    }
}

```

```

        alertDialog.setContentLabel("You have logged in successfully!\n It's a
            pleasure to see you again!");
        alertDialog.showDialogAlert();
    }
}

```

DeadEventListener class:

```

public class DeadEventListener {
    @Subscribe
    public void deadEvent(DeadEvent event) {
        AlertDialog alertDialog = new AlertDialog();
        alertDialog.setTitle("WordCards");
        alertDialog.setContentLabel(event.getEvent().toString());
        alertDialog.showDialogAlert();
    }
}

```

An example of a custom AlertDialog:

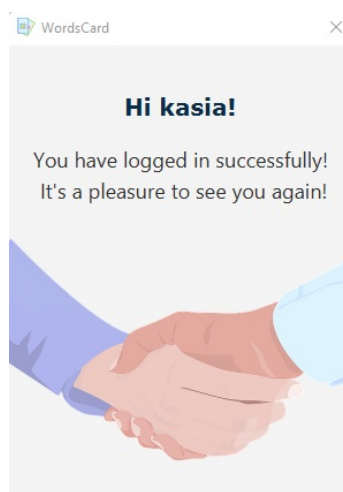


Figure 2: AlertDialog.

3.9 Apache Commons

Apache commons email API has classes for sending for example HTML emails. Added dependency in pom.xml:

```

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-email</artifactId>
    <version>1.5</version>
</dependency>

```

Added code to send HTML email:

```

HtmlEmail email = new HtmlEmail();
email.setHostName(HOST);
email.setSmtPort(PORT);
email.setAuthenticator(new DefaultAuthenticator(userName, password));
email.setSSLonConnect(SSL_FLAG);
email.setFrom(fromAddress);
email.setSubject(subject);
email.setHtmlMsg(htmlMessage);
email.setTextMsg("Your email client does not support HTML messages");
email.addTo(toAddress);
email.send();

```

3.10 Example of use Singleton

To share the same User class instance, I created a UserHolder class that uses the Singleton Design Pattern.

```
public class UserHolder {
    private User user;
    private final static UserHolder INSTANCE = new UserHolder();

    public UserHolder() {}

    public static UserHolder getInstance() {
        return INSTANCE;
    }

    public void setUser(User u) {
        this.user = u;
    }

    public User getUser() {
        return this.user;
    }
}
```

4 Important dependencies

- **Gson** - a Java library used to convert JSON objects to POJO
- **Spring Security RSA** - a small utility library for RSA ciphers
- **Google Guice** - an ultra-lightweight dependency injection framework
- **Apache Commons Email** - provide an API for sending email
- **Guava** - library provides the EventBus which allows publish-subscribe communication between components
- **Hibernate** - an relational mapping (ORM) tool that provides a framework to map object-oriented domain models to relational databases
- **JUnit** - a unit testing framework

5 Final effect

Connecting to the database is quite a long process so loading the login screen does not happen right away when you start the program. I used Preloader to pass the loading information to the user. "Use of a preloader can help to reduce perceived application startup time by showing some content to the user earlier, such as a progress indicator or login prompt." Based on an article available on the oracle website, I created the CustomPreloader class. In the Main class I added the init() function, which is responsible for creating the connection to the database. The preloader will start before the main application and will be hidden before it starts.

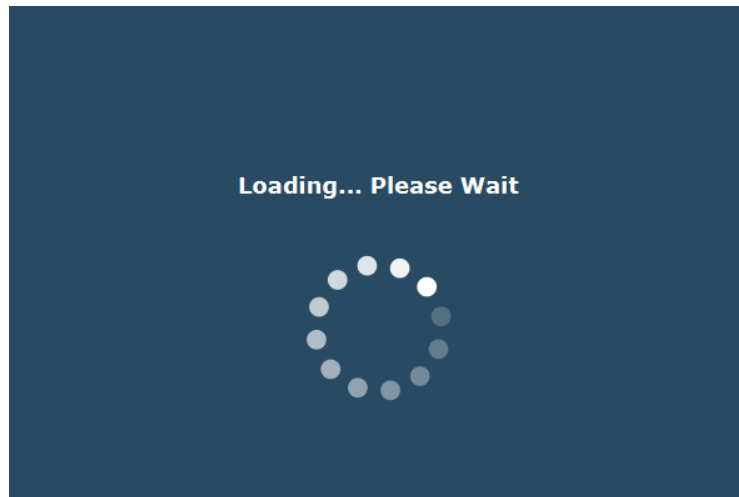


Figure 3: Custom preloader.

Starting the application displays the login window.

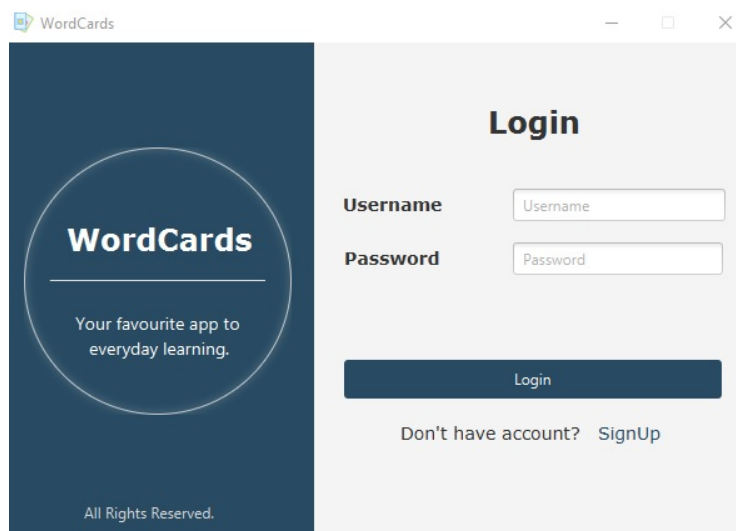
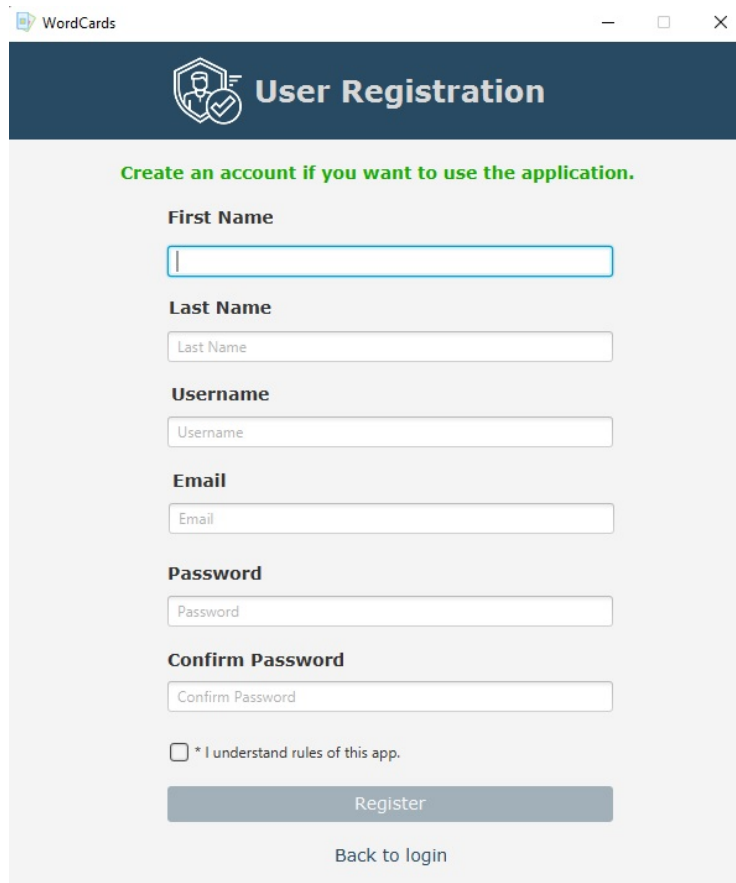


Figure 4: Login screen.

If the user does not have an account, they can create a new one - SignUp button. Entering incorrect data (login, registration) will result in an appropriate message.

The image shows a web browser window titled "WordCards" with a dark blue header. The header contains a logo of a shield with a checkmark and the text "User Registration". Below the header, there is a green message: "Create an account if you want to use the application." The registration form consists of several input fields: "First Name", "Last Name", "Username", "Email", "Password", and "Confirm Password". Each field has a placeholder text. Below the fields, there is a checkbox labeled "* I understand rules of this app." and a "Register" button. At the bottom, there is a link "Back to login".

WordCards

User Registration

Create an account if you want to use the application.

First Name

Last Name

Username

Email

Password

Confirm Password

☐ * I understand rules of this app.

Register

[Back to login](#)

Figure 5: User registration screen.

After successful login or registration, an alert will be displayed with a message.

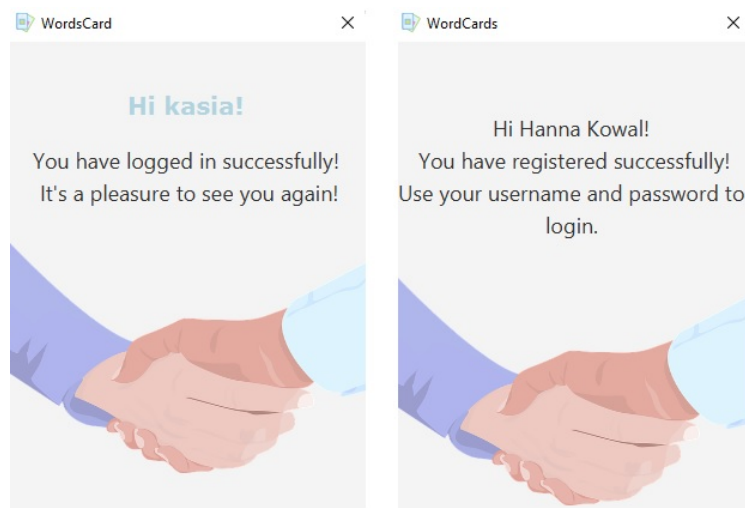


Figure 6: Alert.

After the successful login, the user will see the application with the start screen - dashboard.

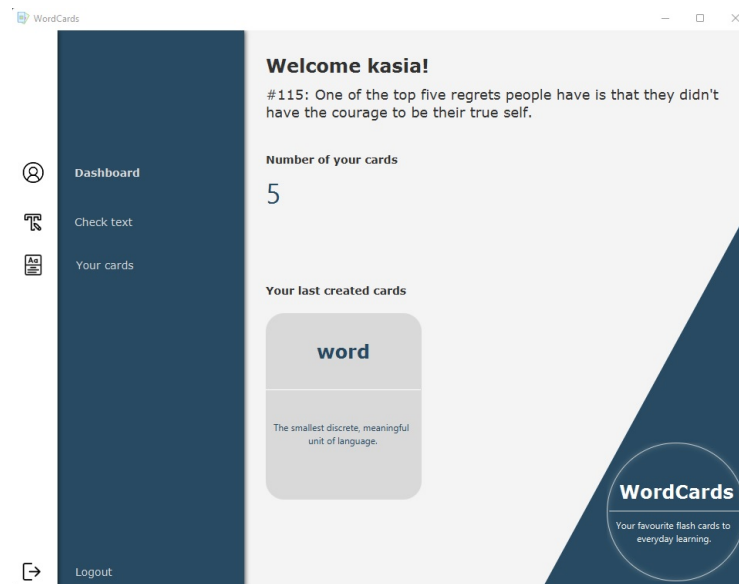


Figure 7: Dashboard.

The menu on the left allows you to switch scenes. The user can check the word - pronunciation information, definitions for different parts of speech, pronunciation recording.

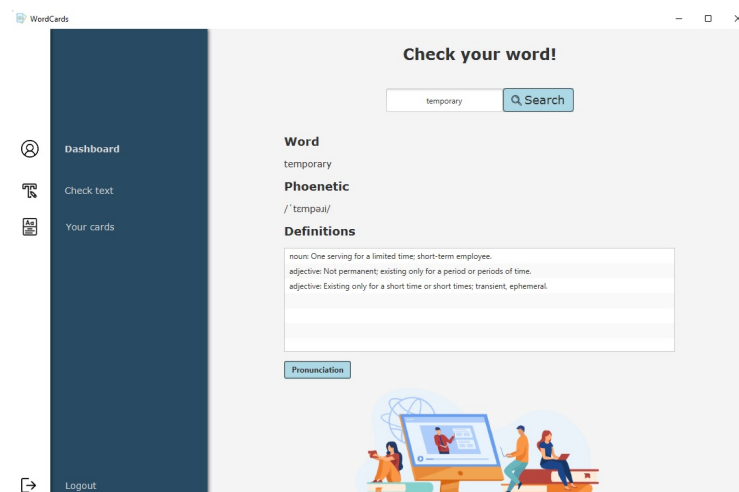


Figure 8: Check text screen.

If the word information is missing, the user will receive the message "No results found".

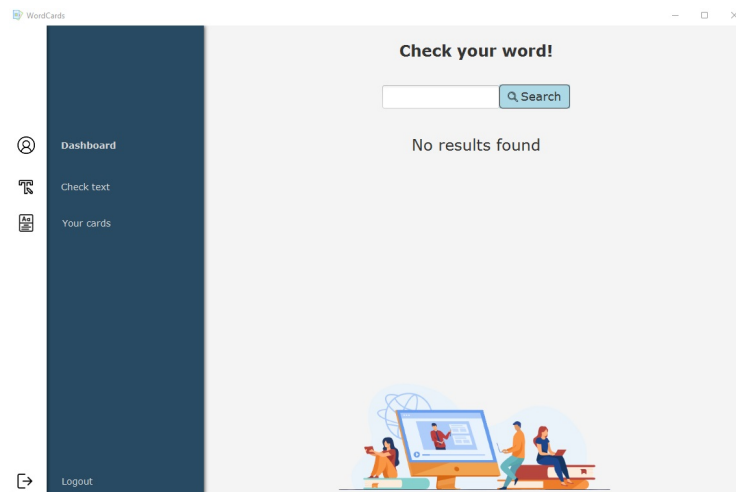


Figure 9: No results found.

New words can be added to the database along with their meaning, definition and example. Selecting the appropriate row and pressing the "Remove" button will remove that word from the database.

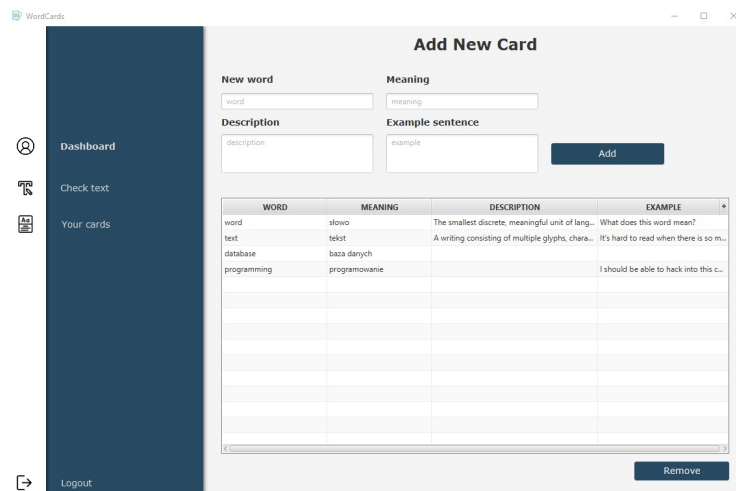


Figure 10: Add new card.

The "Logout" button allows the user to log out and return to the login screen.

6 Useful sources

I used a wide variety of articles, tutorials and presentations. Some of the aspects were implemented as presented during the lab activities. The photos and icons are from [Flaticon](#) and [Frepik](#). The materials used can be found in the bibliography.

7 Conslusions

The application is multi-threaded and uses aspects of asynchronous programming. Custom events allow you to log the user in or display an appropriate message. The data available from the two external web services is used when checking a word or displaying an advice. The words added to the user account are stored in the database. Config properties prevented the use of passwords or other sensitive information in code. Apache Commons made it possible to send welcome emails while the event bus was used to display alerts during login/registration. The design patterns used made it possible, for example, to save information about the logged-in user.

The project allowed a variety of issues to be realized. Some of the problems that appeared were solved by using additional libraries.

Bibliography

- [1] JavaFX custom event definitions and rules for building GUI custom components - presentation made with other faculty members as part of a lecture in the course "Event-driven programming".
- [2] ORACLE *Preloaders*, [Preloaders article](#).
- [3] Gerrit Grunwald, *Custom Controls in JavaFX (Part I)*, [Custom controls article - part I](#).
- [4] Gerrit Grunwald, *Custom Controls in JavaFX (Part III)*, [Custom controls article - part III](#).
- [5] baeldung, [Guide to Guava's EventBus](#).
- [6] Apache Commons, [Apache Commons Email](#)..
- [7] Ed Eden-Rump, [A Definitive Guide To JavaFX Events](#).
- [8] tutorialspoint, [JavaFX - Event Handling](#).
- [9] Manoj Debnath, [Multithreading in JavaFX](#).
- [10] [Refactoring guru](#).
- [11] [JUnit 5 tutorial](#).
- [12] Ben McCann, *Hibernate with JPA Annotations and Guice*, [Dependency injection article](#)..
- [13] Genuine Coder, *JavaFX Background Tasks — How to make your GUI smoother, faster and snappier*, [Video](#).
- [14] Miguel Manjarres, *JavaFX: 3 Ways of Passing Information Between Scenes!*, [Article](#).
- [15] Java Code Junkie, *Create Custom Events — JavaFX GUI Tutorial for Beginners*, [Video](#).