



A P A C H E
P O I

Czym jest Apache POI?

Apache POI (Poor Obfuscation Implementation) to biblioteka napisana w języku Java, służąca **do pracy z dokumentami w formatach Microsoft Office**, takich jak Excel, Word, i PowerPoint. Umożliwia tworzenie, modyfikowanie oraz odczytywanie danych z plików .xls, .xlsx, .doc, .docx, .ppt, i .pptx. Jest szczególnie popularna w środowisku Java, ponieważ pozwala na wygodną manipulację dokumentami bez potrzeby korzystania z oprogramowania Microsoft Office, co sprawia, że jest bardzo przydatna w aplikacjach biznesowych, systemach raportowania i automatyzacji pracy z dokumentami.

W P R O W A D Z E N I E



Czym różni się od innych bibliotek do obsługi dokumentów?

Apache POI wyróżnia się tym, że jest **darmowa** i **open-source**, co sprawia, że jest szeroko stosowana w projektach, które potrzebują wsparcia dla formatów Microsoft Office, a jednocześnie nie mogą korzystać z płatnych rozwiązań. W przeciwieństwie do niektórych komercyjnych bibliotek, takich jak Aspose, Apache POI oferuje podstawową, ale skuteczną funkcjonalność, która pozwala na tworzenie i edycję plików biurowych. Nie obsługuje wszystkich zaawansowanych funkcji Microsoft Office, ale **dzięki modułowej budowie jest bardziej elastyczna**. Jest także bardzo **dobrze zintegrowana z ekosystemem Javy**, co ułatwia jej implementację i współpracę z innymi bibliotekami.

W P R O W A D Z E N I E



STRUKTURA BIBLIOTEKI APACHE POI

Moduły biblioteki:

- **POIFS** (czyli OLE 2 Compound Document Format (format używany przez starsze wersje plików MS Office)) – do pracy z plikami binarnymi, umożliwia odczytanie struktury starszych plików Microsoft Office i wyciąganie z nich danych.
- **HSSF** (Horrible Spreadsheet Format) – do obsługi plików Excela w formacie .xls (starszy format excela).
- **XSSF** (XML Spreadsheet Format) – do obsługi plików Excela w formacie .xlsx.
- **HWPF** i **XWPF** – moduły do obsługi plików Worda (.doc, .docx), przykład: generowanie dokumentów tekstowych, raportów, dodawanie tekstu, obrazów, tabel i akapitów.
- **HSLF** i **XSLF** – do pracy z prezentacjami PowerPoint (.ppt, .pptx), przykład: automatyczne generowanie slajdów, dodawanie tekstu, obrazów i podstawowych animacji do prezentacji.



INSTALACJA APACHE POI

Aby rozpocząć korzystanie z funkcji Apache POI,
dostępne są dwie opcje:

1. Dodanie zależności do projektu w pliku pom.xml.

Idealne rozwiązanie dla projektów korzystających z
Maven.

2. Pobranie bibliotek bezpośrednio.

Można również ręcznie pobrać biblioteki Apache POI
i dodać je do projektu jako zewnętrzne pliki .jar.



Dodanie zależności w projekcie Maven

pom.xml

<dependencies>

<!-- Obsługa formatów opartych na XML (.xlsx, .docx, .pptx) -->

<!-- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml -->

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi-ooxml</artifactId>

<version>5.2.5</version>

</dependency>

<!-- Obsługa starszych formatów binarnych (.xls, .doc, .ppt) -->

<!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi</artifactId>

<version>5.2.5</version>

</dependency>

</dependencies>

Jeśli projekt korzysta z Maven, dodanie zależności Apache POI do pliku pom.xml jest najprostszym i najwygodniejszym rozwiązaniem. Dzięki temu wszystkie wymagane biblioteki są automatycznie pobierane i aktualizowane, co ułatwia zarządzanie wersjami i kompatybilnością.

Link do strony z wersjami zależności są podane nad każdą zależnością.

Dodanie zależności w projekcie Maven c.d.

pom.xml

```
<!-- Opcjonalnie:
<!-- Log4j API -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.20.0</version>
</dependency>

<!-- Log4j Core -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.20.0</version>
</dependency>

<!-- Log4j SLF4J binding, jeśli używasz SLF4J -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j-impl</artifactId>
  <version>2.20.0</version>
</dependency>
```

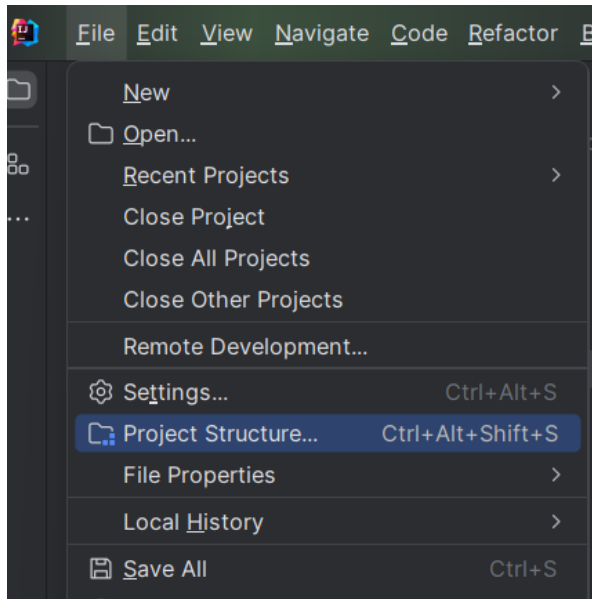
Opcjonalnie można dodać zależności **log4j-api** oraz **log4j-core**. Odpowiadają one za interfejsy logowania, zapisywanie logów i realizację konfiguracji logowania (np. do pliku lub konsoli). Dzięki tym zależnościom możliwe jest szczegółowe monitorowanie pracy POI, co może być pomocne przy rozwiązywaniu problemów z plikami Office. Warto dodać, że Apache POI działa poprawnie również bez nich, choć logowanie będzie wtedy ograniczone. Włączenie log4j-core jest przydatne w aplikacjach produkcyjnych, gdzie monitorowanie i debugowanie są kluczowe, zwłaszcza **przy przetwarzaniu dużej liczby dokumentów**, co często ma miejsce przy używaniu Apache POI.

Pobranie bibliotek ze strony Apache

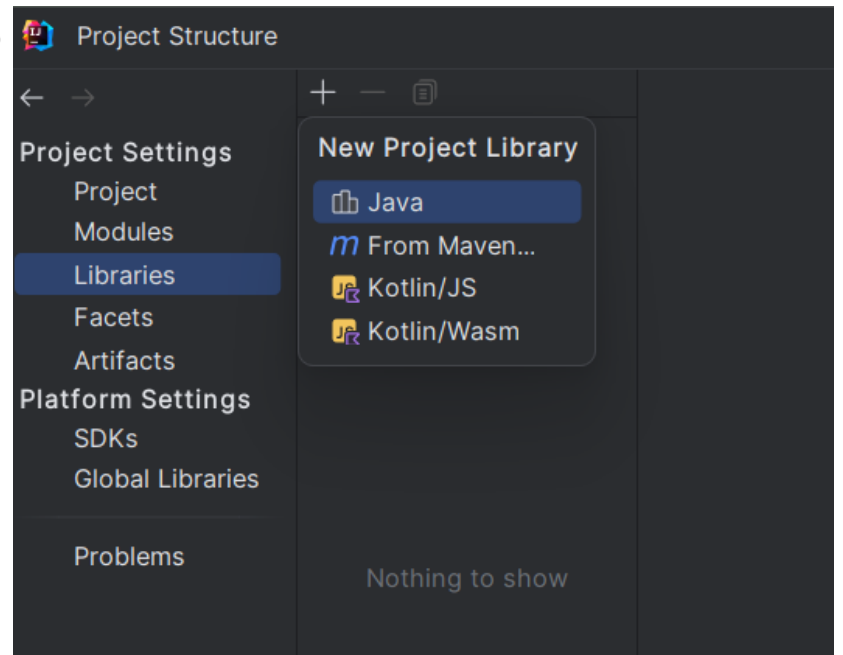
(Ręczne pobieranie bibliotek ma sens jedynie jeśli w projekcie nie korzystamy z systemów zarządzania zależnościami)

1. Wchodzimy na stronę: <https://archive.apache.org/dist/poi/release/bin/>
2. Pobieramy najnowszy (na dole strony) plik .zip lub .tgz w zależności od systemu operacyjnego
3. Po pobraniu pliku .zip lub .tgz, rozpakuj go w wybranym folderze na komputerze. Zazwyczaj znajdziesz tam folder poi-{wersja}, w którym będą pliki .jar oraz folder lib zawierający zależności Apache POI.
4. W projekcie w IntelliJ Idea z menu górnego wybieramy File -> Project Structure i dodajemy pobrane biblioteki.
5. Biblioteki powinny pojawić się w folderze External Libraries.

4.a

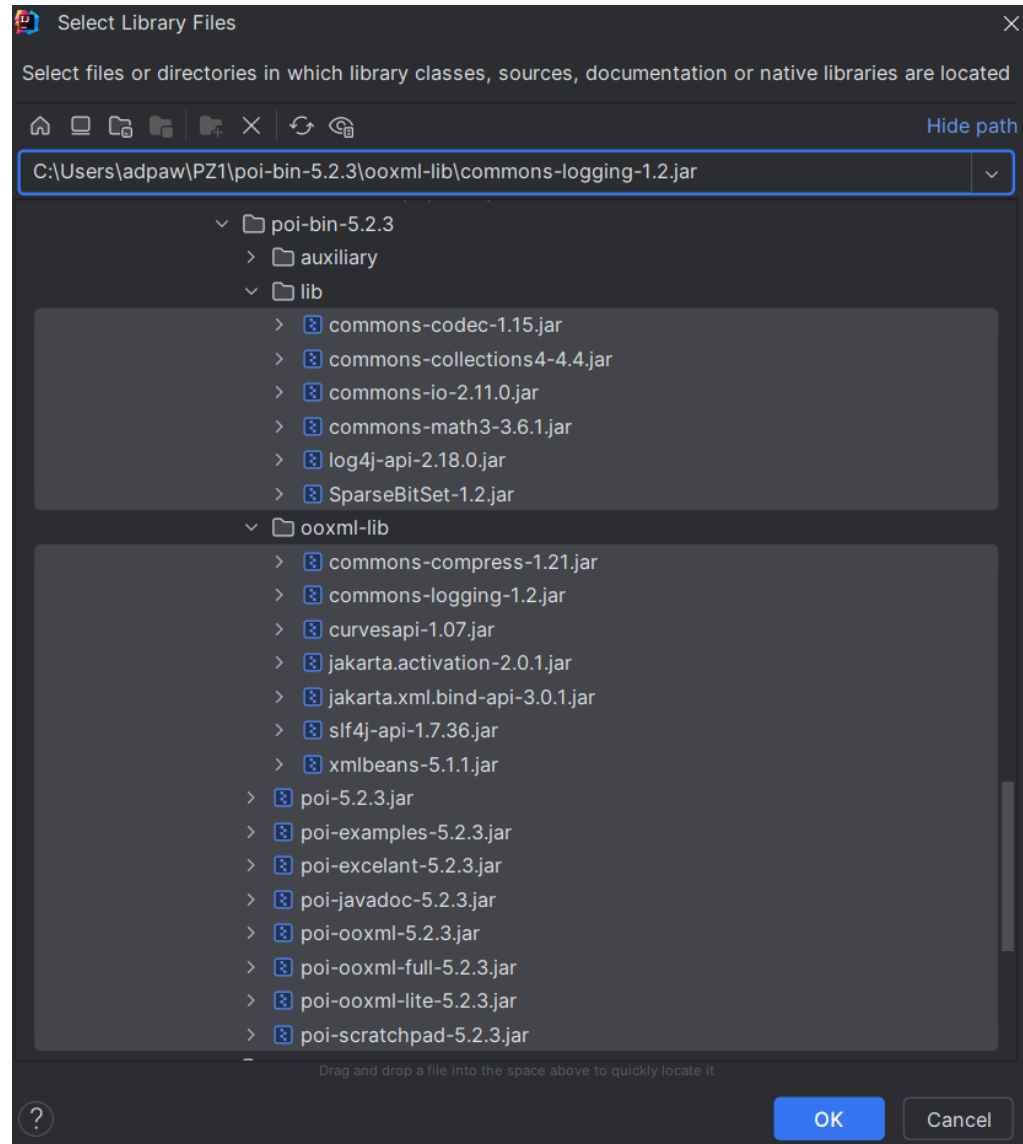


4.b

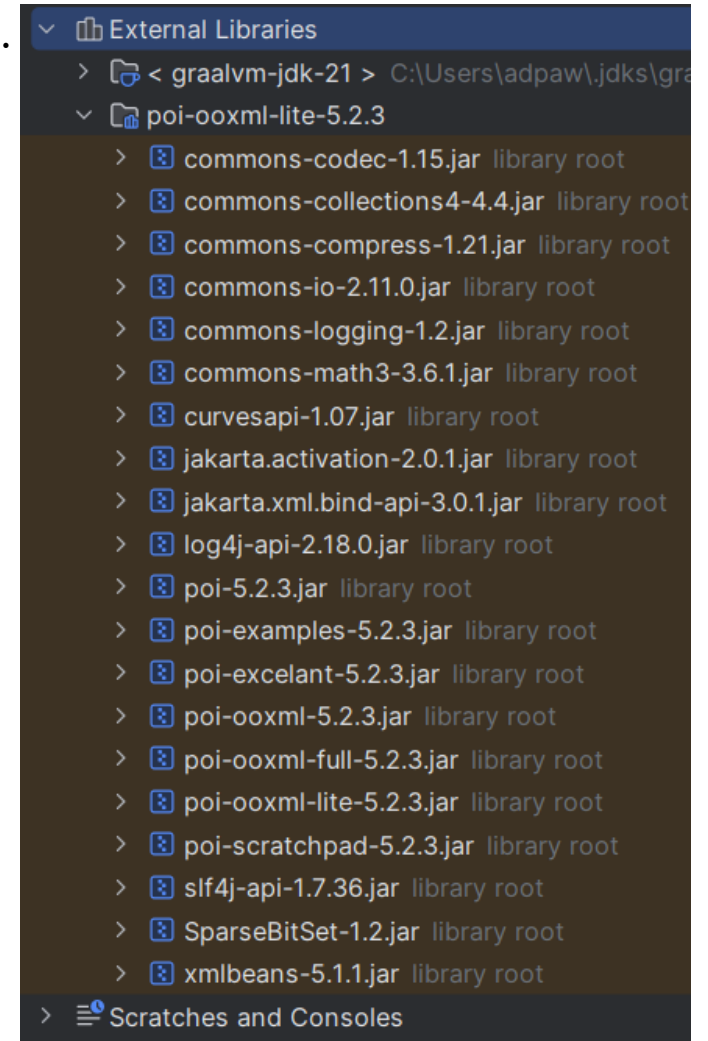


Pobranie bibliotek ze strony Apache c.d.

4.c



5.



Efekt końcowy - biblioteki dodane do projektu

PRACA Z
ARKUSZAMI
EXCEL

- Tworzymy nowy plik Excela w formacie **.xlsx** – **XSSFWorkbook** służy do obsługi nowoczesnych plików **.xlsx**.
- Dodajemy arkusz o nazwie "ExampleSheet".
- Dodajemy pierwszy wiersz i komórkę w kolumnie A1, ustawiając w niej tekst "Hello, Apache POI!".
- Formatujemy komórkę – ustawiamy pogrubienie dla tekstu. **CellStyle** tworzy styl, a **Font** pozwala na dostosowanie tekstu (np. pogrubienie).
- Zapisujemy plik na dysku jako **example.xlsx**, korzystając z **FileOutputStream**



TWORZENIE PLIKU EXCEL (XSSF)

```
import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import java.io.FileOutputStream;
import java.io.IOException;

public class ExcelExample {
    public static void main(String[] args) {
        Workbook workbook = new XSSFWorkbook();
        Sheet sheet = workbook.createSheet("ExampleSheet");

        Row row = sheet.createRow(0);
        Cell cell = row.createCell(0);
        cell.setCellValue("Hello, Apache POI!");

        CellStyle style = workbook.createCellStyle(); // Formatowanie komórki
        Font font = workbook.createFont();
        font.setBold(true);
        style.setFont(font);
        cell.setCellStyle(style);

        try (FileOutputStream fileOut = new FileOutputStream("example.xlsx")) { // Zapis do pliku
            workbook.write(fileOut);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



- Deklarujemy ścieżkę do pliku Excela (excelFilePath), który chcemy odczytać, tutaj plik `example.xlsx`.
- Tworzymy obiekt FileInputStream do odczytu pliku oraz obiekt Workbook (XSSFWorkbook), który otwiera plik Excela.
- Pobieramy pierwszy arkusz za pomocą getSheetAt(0) (arkusze w Excelu są numerowane od zera).
- Iterujemy przez wszystkie wiersze i komórki:
 - Dla każdej komórki sprawdzamy jej typ (tekstowy, numeryczny, logiczny) za pomocą getCellType() i wyświetlamy odpowiednią wartość.
 - Jeżeli typ jest nieznany, wyświetlamy Unknown Type,,
- Wyświetlamy wartości komórek w konsoli, a po każdym wierszu dodajemy nową linię.

Dzięki temu kodowi możemy odczytywać zawartość arkusza kalkulacyjnego i wypisywać dane w kolejności wiersz-po-wierszu oraz komórka-po-komórce.



ODCZYTYWANIE DANYCH Z PLIKU EXCEL

```
public class ExcelReadExample {
    public static void main(String[] args) {
        String excelFilePath = "example.xlsx"; // sciezka

        try (FileInputStream fileInputStream = new FileInputStream(excelFilePath);
            Workbook workbook = new XSSFWorkbook(fileInputStream)) {

            Sheet sheet = workbook.getSheetAt(0); // otwieramy pierwszy arkusz z pliku

            for (Row row : sheet) { // iterujemy po wierszach
                for (Cell cell : row) { // iterujemy po komorkach danego wiersza
                    switch (cell.getCellType()) {
                        case STRING:
                            System.out.print(cell.getStringCellValue() + " ");
                            break;
                        case NUMERIC:
                            System.out.print(cell.getNumericCellValue() + " ");
                            break;
                        case BOOLEAN:
                            System.out.print(cell.getBooleanCellValue() + " ");
                            break;
                        default:
                            System.out.print("Unknown Type ");
                            break;
                    }
                }
                System.out.println(); // nowa linia po wczytaniu danych z wiersza
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



PRACA Z
DOKUMENTAMI
WORD

- Tworzymy nowy dokument Word w formacie `.docx`, używając `XWPFDocument`.
- Dodajemy akapit do dokumentu przez `XWPFPparagraph`.
- Dodajemy element tekstowy (**`XWPFRun`**) w akapicie z tekstem "Hello, Apache POI!" oraz ustawiamy go na pogrubiony.
- Mamy opcję dodania obrazu (komentarz w kodzie pokazuje, jak wstawić obraz w formacie `.jpg`).
- Zapisujemy dokument na dysku jako `example.docx` za pomocą `FileOutputStream`



TWORZENIE DOKUMENTU WORD (XWPF)

```
import org.apache.poi.xwpf.usermodel.*;
import java.io.FileOutputStream;
import java.io.IOException;

public class WordExample {
    public static void main(String[] args) {
        XWPFDocument document = new XWPFDocument();

        // Dodanie akapitu
        XWPFParagraph paragraph = document.createParagraph();
        XWPFRun run = paragraph.createRun();
        run.setText("Hello, Apache POI!");
        run.setBold(true);

        // Dodanie obrazu
        // run.addPicture(new FileInputStream("image.jpg"), XWPFDocument.PICTURE_TYPE_JPEG, "image.jpg", Units.toEMU(200), Units.toEMU(200));

        // Zapis dokumentu
        try (FileOutputStream out = new FileOutputStream("example.docx")) {
            document.write(out);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



PRACA Z
PREZENTACJAMI
POWERPOINT

```

public class PowerPointExample {
    public static void main(String[] args) {
        XMLSlideShow ppt = new XMLSlideShow();

        XSLFSlide slide = ppt.createSlide();

        XSLFTextShape title = slide.createTextBox();
        title.setAnchor(new java.awt.Rectangle(50, 50, 400, 100));
        title.setText("Hello, Apache POI!");

        XSLFTextParagraph paragraph = title.addNewTextParagraph();
        XSLFTextRun run = paragraph.addNewTextRun();
        run.setFontSize(24.0);
        run.setBold(true);

        try (FileInputStream inputStream = new FileInputStream("image.jpg")) {
            byte[] pictureData = IOUtils.toByteArray(inputStream);
            XSLFPictureData pd = ppt.addPicture(pictureData, XSLFPictureData.PictureType.PNG);
            XSLFPictureShape pic = slide.createPicture(pd);
            pic.setAnchor(new java.awt.Rectangle(50, 150, 300, 200)); // Pozycja i rozmiar obrazka
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (FileOutputStream out = new FileOutputStream("example.pptx")) {
            ppt.write(out);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Tworzenie prezentacji PowerPoint (XSLF)

- Tworzymy nową prezentację PowerPoint w formacie .pptx przy użyciu XMLSlideShow.
- Dodajemy nowy slajd do prezentacji.
- Dodajemy pole tekstowe na slajdzie i ustawiamy jego tekst na "Hello, Apache POI!".
- Formatujemy tekst w polu tekstowym, ustawiając rozmiar czcionki na 24 i stylizując tekst jako pogrubiony.
- Dodajemy obrazek podając poprawną ścieżkę do pliku .jpg oraz formatujemy jego rozmiar i pozycję.
- Zapisujemy prezentację na dysku jako example.pptx, używając FileOutputStream

```

public class PowerPointReader {
    public static void modifySlideText(String inputFilePath, String outputFilePath) {
        try (FileInputStream inputStream = new FileInputStream(inputFilePath);
            XMLSlideShow ppt = new XMLSlideShow(inputStream)) {

            XSLFSlide slide = ppt.getSlides().get(0);

            for (XSLFShape shape : slide.getShapes()) {
                if (shape instanceof XSLFTextShape) {
                    XSLFTextShape textShape = (XSLFTextShape) shape;

                    textShape.clearText(); // Usunięcie istniejącego tekstu
                    textShape.setText("Zmodyfikowany tekst!"); // Ustawienie nowego tekstu
                }
            }

            for (XSLFPictureData pictureData : ppt.getPictureData()) {
                byte[] bytes = pictureData.getData();
                String fileName = pictureData.getFileName();

                Path filePath = Paths.get(".", fileName); // "." oznacza aktualny folder projektu
                Files.write(filePath, bytes);
            }

            try (FileOutputStream outputStream = new FileOutputStream(outputFilePath)) {
                ppt.write(outputStream);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Odczyt i zaktualizowanie prezentacji

Klasa PowerPointReader demonstruje, jak:

- wczytać istniejącą prezentację .pptx,
- przeiterować po elementach (shapes) prezentacji
- zmienić tekst na pierwszym slajdzie,
- pobrać obrazek z prezentacji
- zapisać zmodyfikowaną wersję do nowego pliku.



DZIĘKUJEMY ZA UWAGĘ

Piotr Pączek i Paweł Adamczyk