|  |  |
| --- | --- |
| Logo AMW Gdynia | **AKADEMIA MARYNARKI WOJENNEJ**  **im. BOHATERÓW WESTERPLATTE**  **WYDZIAŁ MECHANICZNO-ELEKTRYCZNY**  **KATEDRA INFORMATYKI** |

**SPRAWOZDANIE**

**Aplikacja Informacyjna**

**Miasto Gdynia**

|  |  |
| --- | --- |
|  | Wykonawca: |
|  | **Patrycja Przybysz** |
|  | Grupa: |
|  | **215IC B2** |

GDYNIA 2024

1. Opis Aplikacji

Zaimplementowana przeze mnie aplikacja jest aplikacją informacyjną dotyczącą miasta Gdynia. Umożliwia użytkownikowi przeglądanie najnowszych artykułów dotyczących miasta, korzystanie z internetowej mapy zawierającej lokalizacje najważniejszych miejsc w mieście oraz historię miasta.

Budowa aplikacji:

1. WelcomeActivity

Pełni rolę ekranu powitalnego (splash screen), który jest wyświetlany użytkownikowi na początku uruchomienia aplikacji. Po upływie określonego czasu, ekran powitalny automatycznie przechodzi do głównej aktywności aplikacji.



1. MainActivity

Klasa MainActivity jest główną aktywnością aplikacji informacyjnej dotyczącej miasta Gdynia. Odpowiada za wyświetlanie interfejsu użytkownika, który umożliwia nawigację pomiędzy różnymi sekcjami aplikacji, takimi jak najnowsze artykuły, historia miasta oraz turystyka.

* aplikacja wyświetla listę artykułów dotyczących miasta Gdynia oraz szczegóły artykułu jak data tytuł treść oraz zdjęcia, korzystając z fragmentów ArticleListFragment i ArticleDetailFragment.
* Klasa MainActivity inicjalizuje bazę danych i wstawia przykładowe dane do tabeli artykułów.



1. TourismActivity

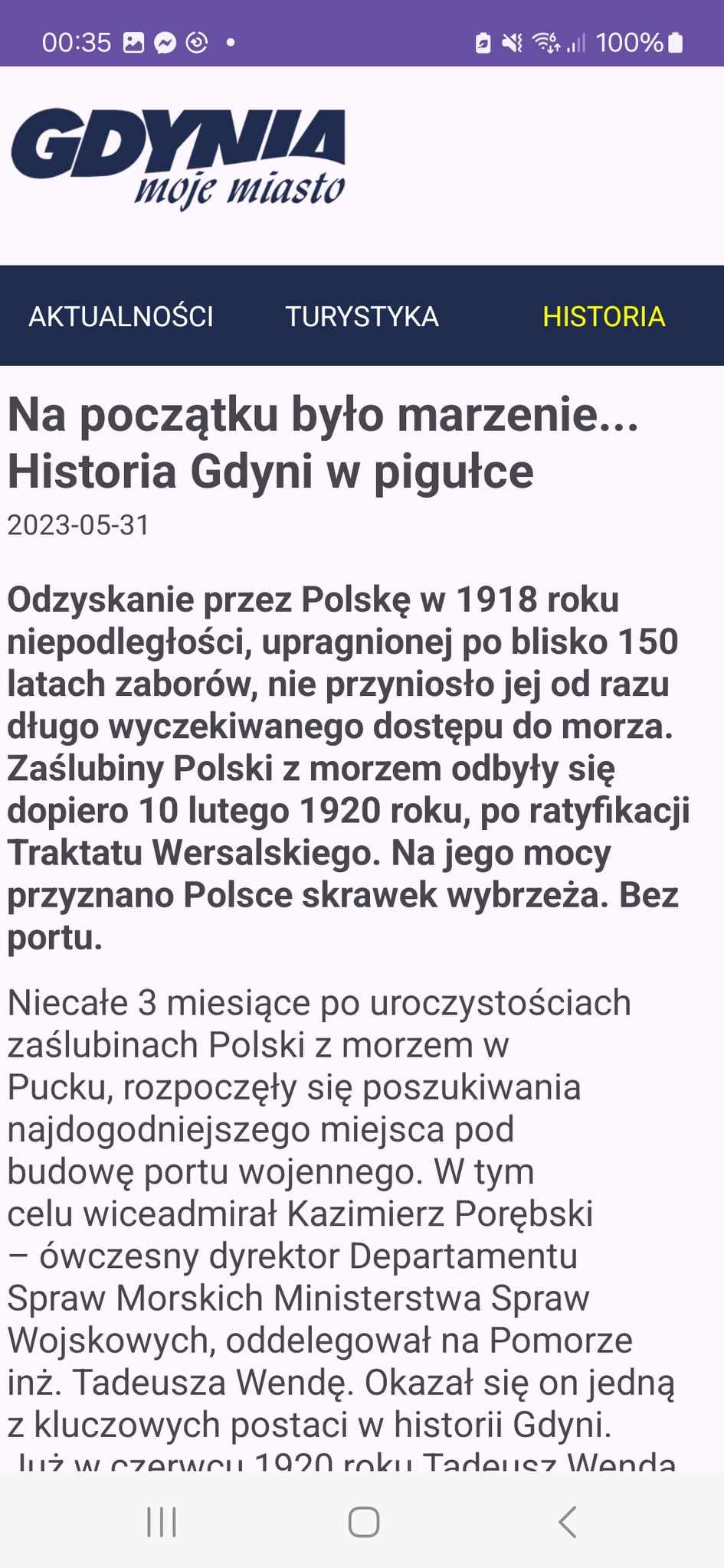
Klasa TourismActivity jest jedną z aktywności w aplikacji informacyjnej dotyczącej miasta Gdynia. Umożliwia użytkownikowi przejście do sekcji historii miasta, powrót do ekranu głównego oraz otwarcie zewnętrznej mapy z atrakcjami turystycznymi miasta. Po kliknięciu przycisku użytkownik zostaje przekierowany do zewnętrznej mapy z atrakcjami turystycznymi miasta Gdynia w przeglądarce internetowej.



1. HistoryActivity

Klasa HistoryActivity jest jedną z aktywności w aplikacji informacyjnej dotyczącej miasta Gdynia. Umożliwia użytkownikowi przeglądanie szczegółowego artykułu dotyczącego historii miasta. Klasa zapewnia również nawigację do innych sekcji aplikacji, takich jak ekran główny i sekcja turystyki.

Pobiera artykuł z bazy danych i wyświetla jego tytuł, datę oraz zawartość, w tym teksty i obrazy.



1. Zastosowane technologie
2. **SQLite** to lekka, wbudowana baza danych, która jest często używana w aplikacjach mobilnych do przechowywania danych lokalnie na urządzeniu. W aplikacji SQLite jest używana do przechowywania artykułów.

* DatabaseHelper: Jest to klasa pomocnicza, która zarządza tworzeniem i aktualizacją bazy danych. W tej klasie zdefiniowane są metody do dodawania, usuwania i pobierania danych z bazy danych.
* Przykładowe użycie: Artykuły są pobierane z bazy danych i wyświetlane w odpowiednich aktywnościach. Metoda getAllArticles() w DatabaseHelper zwraca listę wszystkich artykułów przechowywanych w bazie danych.
* package com.example.projekt  
    
  import android.content.ContentValues  
  import android.content.Context  
  import android.database.sqlite.SQLiteDatabase  
  import android.database.sqlite.SQLiteOpenHelper  
  import android.util.Log  
    
  class DatabaseHelper(context: Context) : SQLiteOpenHelper(context, DATABASE\_NAME, null, DATABASE\_VERSION) {  
    
   companion object {  
   private const val DATABASE\_NAME = "articles.db"  
   private const val DATABASE\_VERSION = 1  
    
   private const val TABLE\_ARTICLES = "articles"  
   private const val COLUMN\_ID = "id"  
   private const val COLUMN\_TITLE = "title"  
   private const val COLUMN\_DATE = "date"  
    
   private const val TABLE\_ELEMENTS = "elements"  
   private const val COLUMN\_ARTICLE\_ID = "article\_id"  
   private const val COLUMN\_TYPE = "type"  
   private const val COLUMN\_CONTENT = "content"  
   }  
    
   override fun onCreate(db: SQLiteDatabase) {  
   val createArticlesTable = """  
   CREATE TABLE $TABLE\_ARTICLES (  
   $COLUMN\_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
   $COLUMN\_TITLE TEXT,  
   $COLUMN\_DATE TEXT  
   )  
   """.*trimIndent*()  
   db.execSQL(createArticlesTable)  
    
   val createElementsTable = """  
   CREATE TABLE $TABLE\_ELEMENTS (  
   $COLUMN\_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
   $COLUMN\_ARTICLE\_ID INTEGER,  
   $COLUMN\_TYPE TEXT,  
   $COLUMN\_CONTENT TEXT,  
   FOREIGN KEY($COLUMN\_ARTICLE\_ID) REFERENCES $TABLE\_ARTICLES($COLUMN\_ID)  
   )  
   """.*trimIndent*()  
   db.execSQL(createElementsTable)  
   }  
    
   override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {  
   db.execSQL("DROP TABLE IF EXISTS $TABLE\_ELEMENTS")  
   db.execSQL("DROP TABLE IF EXISTS $TABLE\_ARTICLES")  
   onCreate(db)  
   }  
    
   fun insertArticle(article: Article): Long {  
   if (articleExists(article.title, article.date)) {  
   Log.d("DatabaseHelper", "Article already exists: ${article.title}")  
   return -1 // Return -1 if article already exists  
   }  
    
   val db = *writableDatabase* val values = ContentValues().*apply* **{** put(COLUMN\_TITLE, article.title)  
   put(COLUMN\_DATE, article.date)  
   **}** val articleId = db.insert(TABLE\_ARTICLES, null, values)  
    
   if (articleId == -1L) {  
   Log.e("DatabaseHelper", "Failed to insert article: ${article.title}")  
   return -1  
   }  
    
   article.elements.*forEach* **{** element **->** val elementValues = ContentValues().*apply* **{** put(COLUMN\_ARTICLE\_ID, articleId)  
   put(COLUMN\_TYPE, when (element) {  
   is ArticleElement.Text -> "text"  
   is ArticleElement.Image -> "image"  
   })  
   put(COLUMN\_CONTENT, when (element) {  
   is ArticleElement.Text -> element.content  
   is ArticleElement.Image -> element.imagePath  
   })  
   **}** val elementId = db.insert(TABLE\_ELEMENTS, null, elementValues)  
   if (elementId == -1L) {  
   Log.e("DatabaseHelper", "Failed to insert element for article ID: $articleId")  
   }  
   **}** Log.d("DatabaseHelper", "Inserted article with ID: $articleId")  
   return articleId  
   }  
    
   private fun articleExists(title: String, date: String): Boolean {  
   val db = *readableDatabase* val selection = "$COLUMN\_TITLE = ? AND $COLUMN\_DATE = ?"  
   val selectionArgs = *arrayOf*(title, date)  
   val cursor = db.query(TABLE\_ARTICLES, *arrayOf*(COLUMN\_ID), selection, selectionArgs, null, null, null)  
   val exists = cursor.*count* > 0  
   cursor.close()  
   return exists  
   }  
    
   fun getAllArticles(): List<Article> {  
   val db = *readableDatabase* val articles = *mutableListOf*<Article>()  
    
   val cursor = db.query(TABLE\_ARTICLES, null, null, null, null, null, null)  
   if (cursor.moveToFirst()) {  
   do {  
   val id = cursor.getLong(cursor.getColumnIndexOrThrow(COLUMN\_ID))  
   val title = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN\_TITLE))  
   val date = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN\_DATE))  
    
   val elements = getElementsForArticle(id)  
   articles.add(Article(id, title, date, elements))  
   } while (cursor.moveToNext())  
   }  
   cursor.close()  
    
   return articles  
   }  
    
   fun getArticle(articleId: Long): Article? {  
   val db = *readableDatabase* var article: Article? = null  
    
   val selection = "$COLUMN\_ID = ?"  
   val selectionArgs = *arrayOf*(articleId.toString())  
   val cursor = db.query(TABLE\_ARTICLES, null, selection, selectionArgs, null, null, null)  
   if (cursor.moveToFirst()) {  
   val title = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN\_TITLE))  
   val date = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN\_DATE))  
    
   val elements = getElementsForArticle(articleId)  
   article = Article(articleId, title, date, elements)  
   }  
   cursor.close()  
    
   return article  
   }  
    
    
    
    
   private fun getElementsForArticle(articleId: Long): List<ArticleElement> {  
   val db = *readableDatabase* val elements = *mutableListOf*<ArticleElement>()  
    
   val selection = "$COLUMN\_ARTICLE\_ID = ?"  
   val selectionArgs = *arrayOf*(articleId.toString())  
   val cursor = db.query(TABLE\_ELEMENTS, null, selection, selectionArgs, null, null, null)  
   if (cursor.moveToFirst()) {  
   do {  
   val type = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN\_TYPE))  
   val content = cursor.getString(cursor.getColumnIndexOrThrow(COLUMN\_CONTENT))  
   when (type) {  
   "text" -> elements.add(ArticleElement.Text(content))  
   "image" -> elements.add(ArticleElement.Image(content))  
   }  
   } while (cursor.moveToNext())  
   }  
   cursor.close()  
    
   return elements  
   }  
    
   fun clearDatabase() {  
   val db = *writableDatabase* db.delete(TABLE\_ELEMENTS, null, null)  
   db.delete(TABLE\_ARTICLES, null, null)  
   }  
  }

1. **View Binding** to funkcja dostępna w Android Studio, która ułatwia pracę z widokami w kodzie Kotlin/Java. Generuje ona klasy wiążące widoki z plików XML, co pozwala na bezpieczne typowo i wydajne odnajdywanie widoków w kodzie.

* Zastosowanie w aplikacji: ViewBinding umożliwia łatwe odwoływanie się do widoków zdefiniowanych w plikach XML bez potrzeby używania metod findViewById. Dzięki temu kod jest bardziej czytelny i mniej podatny na błędy.

Przykład użycia:

override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 binding = ActivityMainBinding.inflate(*layoutInflater*)  
 setContentView(binding.*root*)  
  
 dbHelper = DatabaseHelper(this)  
 dbHelper.clearDatabase()  
 insertSampleData()  
  
 binding.textViewMain.setOnClickListener **{** val intent = Intent(this, MainActivity::class.*java*)  
 startActivity(intent)  
 **}** binding.textViewHistory.setOnClickListener **{** val intent = Intent(this, HistoryActivity::class.*java*)  
 startActivity(intent)  
 **}**

1. **RecyclerView** to bardziej elastyczna i wydajna wersja ListView, przeznaczona do wyświetlania dużych zestawów danych.

* **Zastosowanie w aplikacji:** RecyclerView jest używane do wyświetlania listy artykułów w MainActivity. Dzięki niemu możliwe jest efektywne zarządzanie i prezentowanie dużych ilości danych.

1. **Fragments** to modułowe części interfejsu użytkownika, które mogą być wielokrotnie używane w różnych aktywnościach.

* **Zastosowanie w aplikacji:** Fragmenty są używane do wyświetlania listy artykułów (ArticleListFragment) oraz szczegółów artykułów (ArticleDetailFragment). Ułatwia to zarządzanie interfejsem użytkownika oraz poprawia modularność aplikacji.

1. **Intenty** to mechanizm umożliwiający komunikację między komponentami aplikacji, takimi jak aktywności i serwisy.

* Intenty są używane do nawigacji między różnymi aktywnościami, takimi jak przejścia z ekranu głównego do historii miasta czy sekcji turystyki.

1. **Handler i Looper** to klasy używane do zarządzania kolejkami wiadomości i zadaniami wykonywanymi w wątku głównym.

* Handler i Looper są używane w WelcomeActivity do implementacji opóźnienia wyświetlania ekranu powitalnego przed przejściem do ekranu głównego.