Module 5 Critical Thinking Assignment: My Photo Gallery App

Alexander Ricciardi

Colorado State University Global

CSC475: Platform-Based Development

Professor Herbert Pensado

March 16, 2025

Module 5 Critical Thinking Assignment: My Photo Gallery

This documentation is part of the Module 5 Critical Thinking Assignment from CSC475: Platform-Based Development at Colorado State University Global. The documentation provides an overview of the Android application's functionality and testing scenarios, including the application's Kotlin code source overview and output screenshots. It also reflects on the obstacles faced during the application's development, and the skills acquired. The application is coded in Kotlin 2.0.21 and is named "My Photo Gallery."

The Assignment Direction:

Option #1: "Photo Gallery"
Challenge: Build a photo gallery application that displays a grid of images fetched from the device's storage or an online source. Implement basic image loading and rendering functionalities, allowing users to view and scroll through a collection of photos.
Please ensure that your submission includes the following components:

- Source code file(s) containing the program implementation.
- A 1-page paper explaining the program's purpose, the obstacles faced during its development, and the skills acquired. The paper should also include screenshots showcasing the successful execution of the program.
- Compile and submit your pseudocode, source code, and screenshots of the application executing the application, the results and GIT repository in a single document.

Program Description:

The program is a small Android application that allows a user to browse images from pexels.com (a website that provides free stock photos).

- When launched, the home page of the app displays a browsable list of curated professional photographs selected by Pexels.
- Search for specific images using keywords
- View detailed information about each photograph, including photographer credits
- The app User Interface (UI) follows Material Design principles

⚠️ My notes:

The application is developed using Kotlin 2.0.21 and the following:

- Jetpack Compose (2.7.x): UI
- Retrofit (2.9.0): API communication
- OkHttp (4.11.0): HTTP client
- Gson (1.6.0): JSON serialization/deserialization
- Coil (2.50): For asynchronous image loading with Compose integration
- Kotlin Coroutines (1.7.3):
- Navigation Compose (2.7.7): navigation between screens
- Material 3: Material Design components and theming

Table 1

Pexels API's Calls (Part of the notes)

Endpoint	URL	Description
Photo Endpoints		
Search Photos	https://api.pexels.com/v1/search	Search for photos with parameters like query, orientation, size, color, locale
Curated Photos	https://api.pexels.com/v1/curated	Get real-time photos curated by the Pexels team
Get a Photo	https://api.pexels.com/v1/photos/:id	Retrieve a specific photo by ID
Video Endpoints		
Search Videos	https://api.pexels.com/videos/search	Search for videos with parameters like query, orientation, size, locale
Popular Videos	https://api.pexels.com/videos/popular	Get current popular Pexels videos
Get a Video	https://api.pexels.com/videos/videos/:id	Retrieve a specific video by ID
Collection Endpoints		
Featured Collections	https://api.pexels.com/v1/collections/featured	Get all featured collections on Pexels
My Collections	https://api.pexels.com/v1/collections	Get all of your personal collections
Collection Media	https://api.pexels.com/v1/collections/:id	Get all media (photos and videos) within a single collection

Note: the table lists the different API calls available from Pexels. Data from "Pexls API" by Pexels (n.d.)

Git Repository:

This is a picture of my GitHub page: 

I use GitHub as my Distributed Version Control System (DVCS), the following is a link to my GitHub, Omegapy.

My GitHub repository that is used to store this assignment is named My-Academics-Portfolio and the link to this specific assignment is: https://github.com/Omegapy/My-Academics-Portfolio/tree/main/Platform-Based-Dev-Android-CSC475/Module-5-Critical-Thinking

Project Map:

- Module-5-CTA-MuPhotoGalery-App.docx (this file, App documentation)

The project files from the app, file structure:

```
myphotogallery_1/
 ├── AndroidManifest.xml
 ├── MainActivity.kt      # Main activity (VIEW)
 │                        # navigation and UI components
 │
 ├── data/                # MODEL LAYER
 │   │                    # data operations and business logic
 │   │
 │   ├── api/             # API service
 │   │   └── PexelsApiService.kt  # API interface for Pexels API
 │   │                    # fetching photos
 │   │
 │   ├── model/           # Data model classes
 │   │   ├── Photo.kt     # Data class
 │   │   │                # photo objects
 │   │   │
 │   │   └── PhotosResponse.kt  # API response data structure
 │   │                    # photo lists from API
 │   │
 │   ├── network/         # Network configuration
 │   │   └── NetworkModule.kt  # setup and API client
 │   │                         # Retrofit
 │   │
 │   └── repository/      # Repository layer - mediates between data sources and ViewModels
 │       ├── PhotoRepository.kt  # Repository interface
 │       │                       # data access methods
 │       │
 │       └── PhotoRepositoryImpl.kt   # Repository implementation
 │                                     # data access using API service
 │
 ├── ui/   # UI components (VIEW & VIEWMODEL)
 ├── components/    # UI components
 │   │   └── PhotoItem.kt # VIEW - photo card component
 │   │                    # individual photos in the grid
 │   │
 │   ├── navigation/  # Navigation components
 │   │   ├── AppNavHost.kt # VIEW - Navigation
 │   │   │                 # Manages navigation
 │   │   │
 │   │   └── NavRoute.kt  # VIEW - route definitions
 │   │                    # app's navigation paths
 │   │
 │   ├── screens/  # App screens (composables)
 │   │   ├── GalleryScreen.kt   # VIEW - gallery screen
 │   │   │                      # photo grid and search
 │   │   │
 │   │   └── PhotoDetailScreen.kt  # VIEW - Photo detail screen
```

```
│   │                            # detailed view of a selected photo
│   │
│   ├── state/           # UI state definitions
│   │   └── UiState.kt   # VIEWMODEL - UI state classes
│   │                    # Loading, Success, Empty, Error states
│   │
│   ├── theme/           # UI theming
│   │   └── Theme.kt     # VIEW - App theme
│   │                    # colors and shapes
│   │
│   └── viewmodel/       # ViewModels
│       └── PhotoViewModel.kt  # VIEWMODEL - Photo manager
│                              # state and user actions
│
└── util/                # Utility classes
    └── NetworkUtils.kt  # MODEL (utility) - Network connectivity
                         # Checks if device has internet connection
```

Reflection

As my last critical thinking assignment (My To Do App), this assignment was also challenging. I had to scrap and recreate my project on Android Studio a couple of times. One of those times was during the synchronization process of the Gradle file, my PC crashed and corrupted Android Studio files. I had to completely remove IDE and reinstall it. Another challenging part was understanding why I was getting the Java error `NumberFormatException` related to just one image ID when populating the home page after the app was launched. The error was occurring because the image ID exceeded the maximum value for Java's 32-bit integers. At first, I was thinking that it was maybe a problem on how I configured Retrofit, but instead it was just a data type issue, using long integer instead of a 32-bit integer for the image IDs fixed the error. The error was occurring in the `Photo.kt` file whiting in the `Photo` class:

```
@SerializedName("photographer_id")
val photographerId: Long,
```

Changing the declaration of the variable `photographerId` from *int* to *long* fixed the problem. Note that the Pexels API returns JSON documents, and when *GSON* deserializes Pexels JSON documents into the *Photo* object, it looks for the `@SerializedName` annotations. In this example, the `val photographerId` needs its type to be declared *long*, for *GSON* to convert the JSON value to the appropriate Kotlin integer type. Fixing this issue helped me sharpen my skills in troubleshooting data-type errors in a multilayer application (within the MVVM architecture).

Another challenging aspect of developing the app was managing UI state. For that I use a sealed interface, `PhotoUiState`, and its subclasses for different states (`Loading`, `Success`, `Empty`, `Error`). This allows the app to manage the UI state based on data type. For example, in the file `GalleryScreen.kt` the Composable function *`GalleryScreen`* uses the *`photostate`*, a *`PhotoUiState`* object, to check the data type/state of the list of the *Photo* objects being fetched. It checks the state as follows:

- If the `Photo` objects are loading, the `photostate` will return the *Loading* state.
- If the `Photo` objects are loaded in the list, `photostate` will return the `Success` state.
- If the loading was successful but no *Photo* objects are present (e.g. a search resulting in no photo found), *`photostate`* will return the `Empty` state.
- If an error occurs, `photostate` will return the `Error` state.

See Code Snippet 1 for part of the code. Note that the `PhotoViewModel` manages the states, it updates the states in response to user actions or data changes (monitors the state flow), in response, the *View* element, the `GalleryScreen` Composable function evaluates the new state and renders the appropriate UI components.

Code Snippet 1

UI Checking State

```
when (photoState) {
    is PhotoUiState.Loading -> {
        LoadingView()
    }
    is PhotoUiState.Success -> {
        PhotoGrid(
            photos = photoState.photos,
            onPhotoClick = onPhotoClick
        )
    }
    is PhotoUiState.Empty -> {
        EmptyView()
    }
    is PhotoUiState.Error -> {
        ErrorView(
            message = photoState.message,
            onRetry = onRefresh
        )
    }
}
```

Note: This Kotlin code snippet illustrates the functionality of the UI checking the state of the photos and calling the related Composable function which renders the appropriate UI components. From the `GalleryScreen.kt` file.

Developing the manager for the UI state and the related state flow in the *ViewModel* was extremely challenging, but also extremely rewarding as I learned through it the complexity of implementing the MVVM architecture and I also learned that implementing a sealed interface approach provided a very robust structure for handling different UI states.

The passing data between screens was also challenging, this required me to understand how to implement and navigate/transfer parameters and the management state of different screens. This was done by implementing and combining navigation arguments and *ViewModel* state. Additionally, I also acquire the skills of using Retrofit for API calls, handling authentication, and parsing responses with Gson.

While coroutines are not implicitly implemented within the codebase of the app, it implements them explicitly throughout the code using keywords such as *suspend* (e.g. `suspend fun getCuratedPhotos()`), *launch* (e.g. `ViewModelScope.launch {…}`), and *Flow* (e.g. `fun getCuratedPhotos(...): Flow<Result<PhotosResponse>>`).

Overall, the development of the app was challenging as I had to face issues with data types, Jetpack Compose dependencies, synchronization/building the Gradle file, and my PC crashing leading to corruption of Android Studio files and the reinstallation of it. However, developing the app allowed me to gain various skills using various libraries. Skills such as a deeper understanding of the MVVM, asynchronous programming with explicit coroutines, network requests, image loading, state

management, and Jetpack Compose navigation. The project allowed me to gain new skills to build more robust and better structured Android apps.

App MVVM Structure

The following section is a set of several tables and figures illustrating various app components and how they fit within the MVVM architecture and app's flow.

Table 2

Model Layer

Name	Type	File	Description
`Photo`	Data Class	`Photo.kt`	Data representing a photo with properties like ID, URL, photographer info, and image sources
`PhotoSource`	Data Class	`Photo.kt`	Nested data class containing URLs to different photo sizes
`PhotosResponse`	Data Class	`PhotosResponse.kt`	API response wrapper
`PhotoRepository`	Interface	`PhotoRepository.kt`	Interface for fetching photo data
`PhotoRepositoryImpl`	Class	`PhotoRepositoryImpl.kt`	PhotoRepository that communicates with the Pexels API
`PexelsApiService`	Interface	`PexelsApiService.kt`	Retrofit interface for making API calls to Pexels
`NetworkModule`	Object	`NetworkModule.kt`	network-related dependencies and configurations

Note: The table lists the various components of the *Model* Layer.

Table 3

ViewModel Layer

Name	Type	File	Description
`PhotoViewModel`	Class	`PhotoViewModel.kt`	Manager for photo data and user interactions, also provides UI state
`PhotoUiState`	Sealed Interface	`UiState.kt`	Defines possible UI states (Loading, Success, Empty, Error)
`PhotoUiState.Loading`	Object	`UiState.kt`	Loading state when data is being fetched
`PhotoUiState.Success`	Data Class	`UiState.kt`	Successful data fetch with a list of photos
`PhotoUiState.Empty`	Object	`UiState.kt`	When no photos are available
`PhotoUiState.Error`	Data Class	`UiState.kt`	Error state with an error message

Note: The table lists the various components of the *ViewModel* Layer.

Table 4

View Layer

Name	Type	File	Description
MainActivity	Class	MainActivity.kt	Main function sets up the theme and navigation
AppNavHost	Composable	AppNavHost.kt	Sets up navigation graph and manages *ViewModels*
NavRoute	Sealed Class	NavRoute.kt	Defines navigation routes
GalleryScreen	Composable	GalleryScreen.kt	Main screen displaying photo grid with search
SearchBar	Composable	GalleryScreen.kt	Search input field
PhotoGrid	Composable	GalleryScreen.kt	Grid display of photos using *LazyVerticalGrid*
LoadingView	Composable	GalleryScreen.kt	Loading indicator for data fetching
EmptyView	Composable	GalleryScreen.kt	Message displayed when no photos are found
ErrorView	Composable	GalleryScreen.kt	Error message with retry button
PhotoDetailScreen	Composable	PhotoDetailScreen.kt	Detailed view of a selected photo
PhotoItem	Composable	PhotoItem.kt	Displays a single photo in the grid

Note: The table lists the various components of the *ViewModel* Layer.

Figure 1

Data Flow

```
PexelsApiService ↔ PhotoRepositoryImpl ↔ PhotoViewModel ↔ Composable Screens
 (API Calls)            (Data Access)  (State Management)  (UI Rendering)
```

Note: the figure illustrates the app data flow.

Figure 2

State Flow

```
PhotoViewModel(_photoState) → PhotoViewModel(photoState) → Composable Screens → UI Components
      (Internal State)              (Exposed State)        (State Observation)  (UI Rendering)
```

Note: the figure illustrates the app state flow.

Figure 3

UI Flow

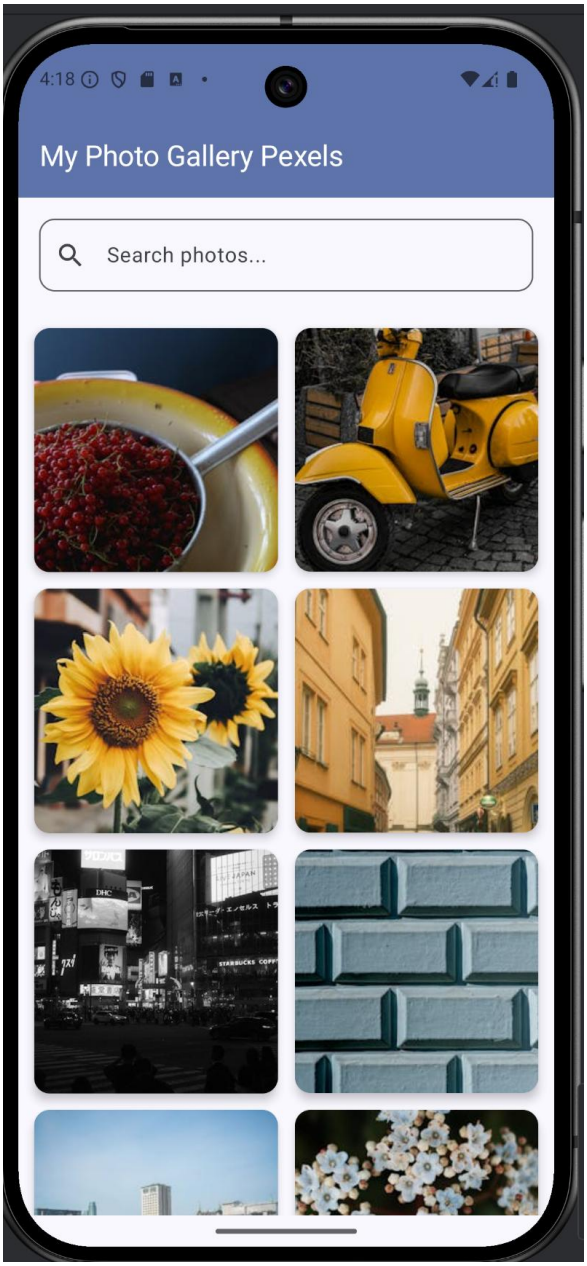
```
UI Components → Composable Screens → PhotoViewModel(Methods) → PhotoRepositoryImpl → PexelsApiService
(User Input)   (Event Handling)      (Action Processing)      (Data Operations)      (API Calls)
```

Note: the figure illustrates the app UI flow.

Screenshots

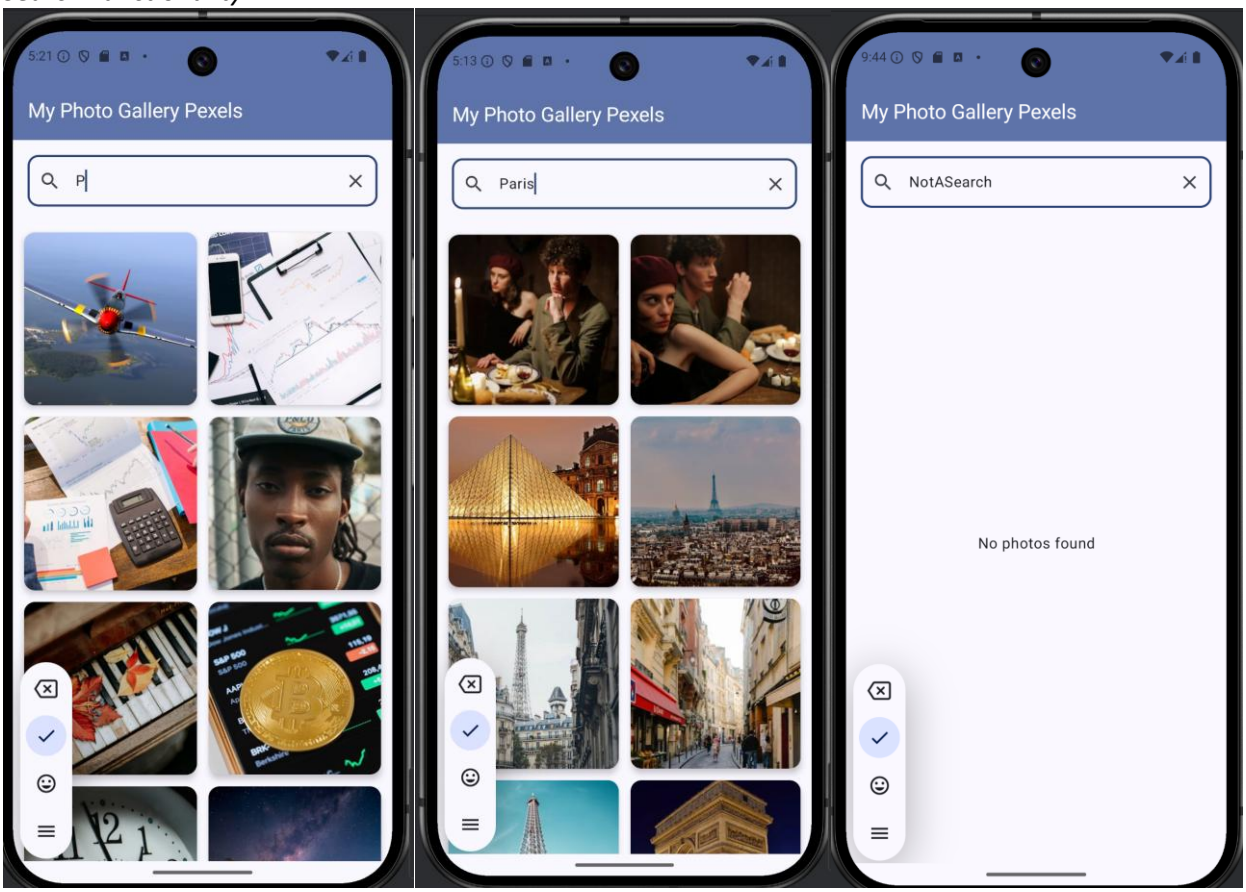
This section demonstrates the functionality of the application by using images illustrating the user interactions with the application and the application outputs from those interactions.

Figure 4
Home Screen



Note: The image shows the home screen of the app after launch. The images on the screens are from the Pexels Curated collection. The `loadCuratedPhotos()` function of the `PhotoViewModel` class calls for the curated image to be retrieved right after the app '`init`', followed by the `getCuratedPhotos()` function in the `PhotoRepositoryImpl` class, then by the `@GET("v1/curated") suspend fun getCuratedPhotos()` of the `PexelsApiService` interface.

Figure 5
Search Functionality

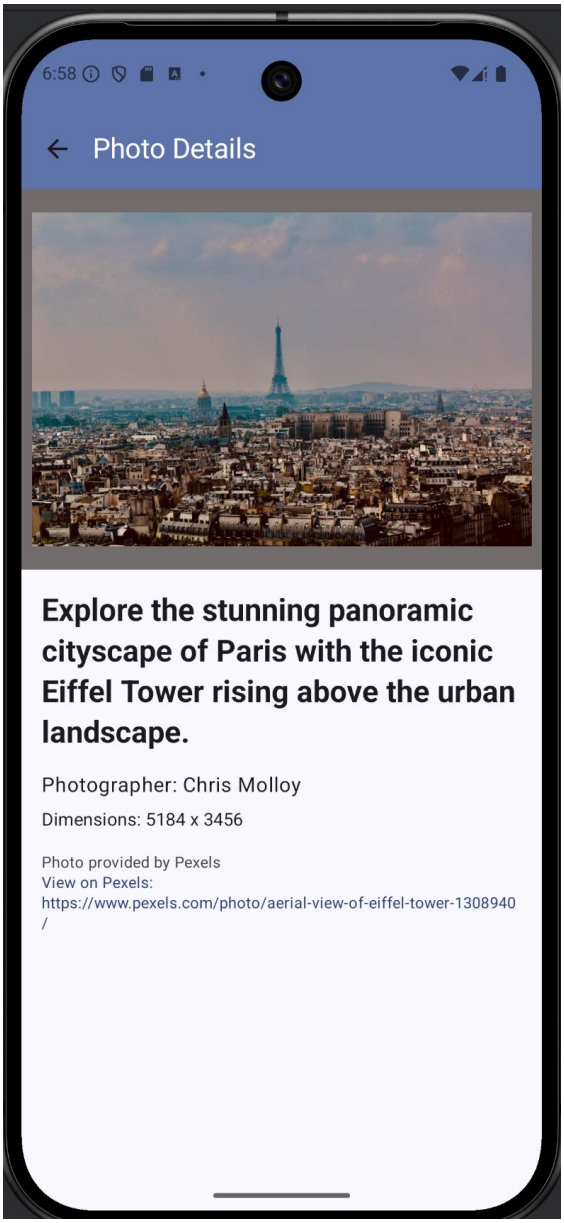


Note: Entering a text in the search field triggers a responsive free-text search, meaning that the app search functionality is reactive in real time, that is, as the user enters the text, the app searches photos displaying the result in real-time. This is done by using the flow state approach, and the `onValueChange` function embedded in the `OptIn(ExperimentalMaterial3Api::class) @Composable private fun SearchBar()` within the `GalleryScreen.kt` file. The data search itself is done starting with the fun search() function of the PhotoViewModel class, followed by the `fun searchPhotos()` function in the `PhotoRepositoryImpl` class, then by the `@GET("v1/search") suspend fun searchPhotos()` of the `PexelsApiService` interface.

See next page

Figure 6

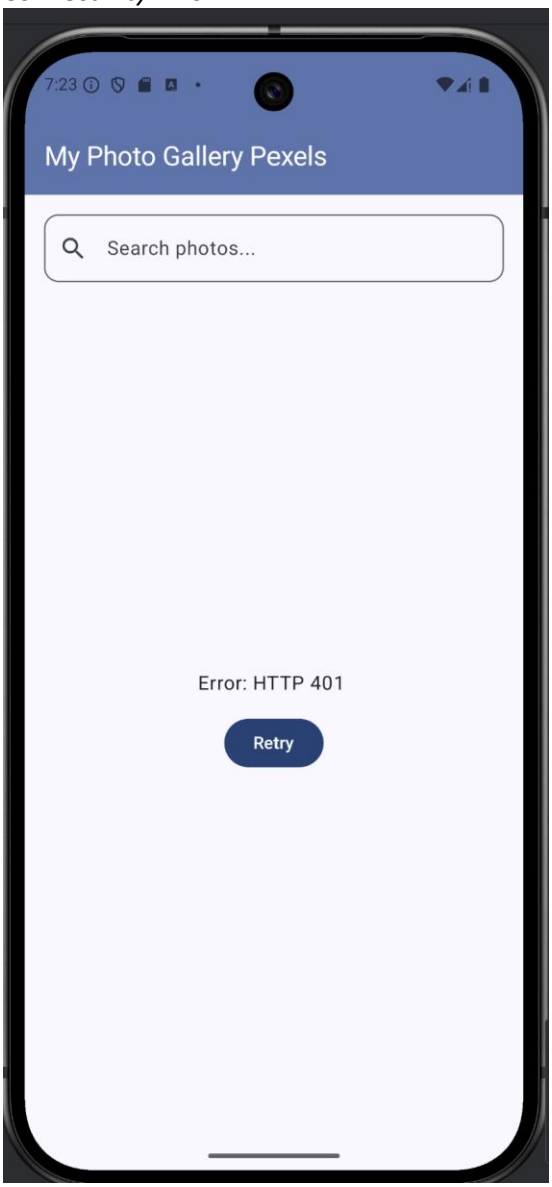
Photo Details



Note: Clicking on the Sort on a photo will display the photo details. In the `GalleryScreen.kt`, the `PhotoGrid()` function displays photos in a grid using `LazyVerticalGrid` and each photo is rendered as a `PhotoItem` with an `onClick` callback that activates the navigation feature of the app. In `AppNavHost.kt`, when a photo is clicked, the `onPhotoClick` functionality embedded in the `@Composable fun AppNavHost()` function is activated, creating a route (e.g. `NavRoute.PhotoDetail.createRoute(photo.id)`) that will fetch and display the photo's details using the `navController.navigate()` function and the `@OptIn(ExperimentalMaterial3Api::class) @Composable fun PhotoDetailScreen()` function from the `PhotDetailScreen.kt` file. Clicking on the arrow (<-) on top of the screen will activate the `onBackClick` functionality taking the user back to the previous screen.

Figure 7

Connectivity Error



Note: Here the Pexels API key was removed from the object `NetworkModule` in the `NetworkModule.kt` file to generate an Error. This error was generated after the '`init`' (after launching the app) of the app within the `loadCuratedPhotos()` function of the `PhotoViewModel` class, trying to fetch the image using the `PhotoRepositoryImpl` class `getCuratedPhotos()` function, followed by the `@GET("v1/curated") suspend fun getCuratedPhotos()` of the `PexelsApiService` interface and it returning authentication key error triggering the `Error` state in the `sealed interface PhotoUiState` within the file `UiState.kt`. This resulted in the error being displayed on the screen, with a button 'Retry', which is rendered using the `@Composable private fun ErrorView()` function in the `GalleryScreen.kt` file.

As shown in figures 4 to 7, the app works as intended.

References:

Pexels (n.d.). *Pexels API*. Pexels. https://www.pexels.com/api/documentation/#introduction