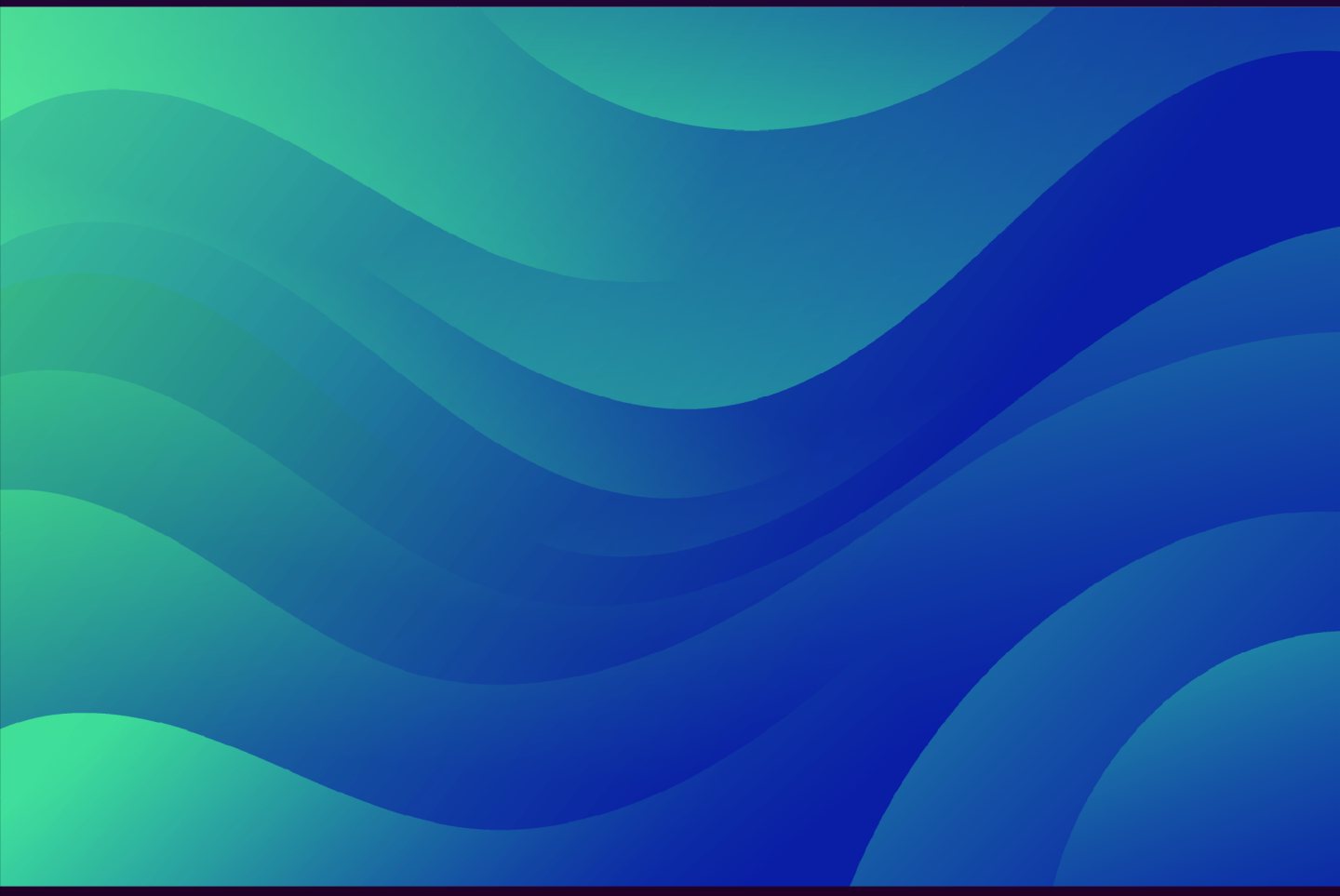
Android Studio Narwhal Essentials



Java Edition

Payload
publishing

Android Studio Narwhal Essentials

Java Edition

Android Studio Narwhal Essentials – Java Edition

ISBN: 978-1-965764-22-0

Rev: 1.0

https://www.payloadbooks.com

Chapter 74

74. A Gemini AI Tutorial

Now that the preparation stage is complete, work can begin on the first screen of the GeminiDemo project. The Text + Image screen will integrate AI capabilities using the Gemini Developer API to allow the user to interact with Gemini by entering text prompts and uploading images for analysis.

74.1 Opening the GeminiDemo project

If you are reading the AI chapters sequentially, launch Android Studio and open the GeminiDemo project from the previous chapter. Alternatively, open the *GeminiDemo_step1* project from the source code samples download. If you have not already done so, you can download the sample code using the following link:

https://www.payloadbooks.com/narwhaljava-prag

Before proceeding, follow the steps in the *"Preparing the Gemini Firebase AI Logic Project"* chapter to add your *google-services.json* file to the project.

74.2 Getting started

The first step is to test that images are available in the photo library for testing the Gemini API code we will add later. Launch Android Studio and run the app on a device or emulator. With the "Text + Image" screen displayed, click the Select a Photo button to test that the visual media picker activity appears:

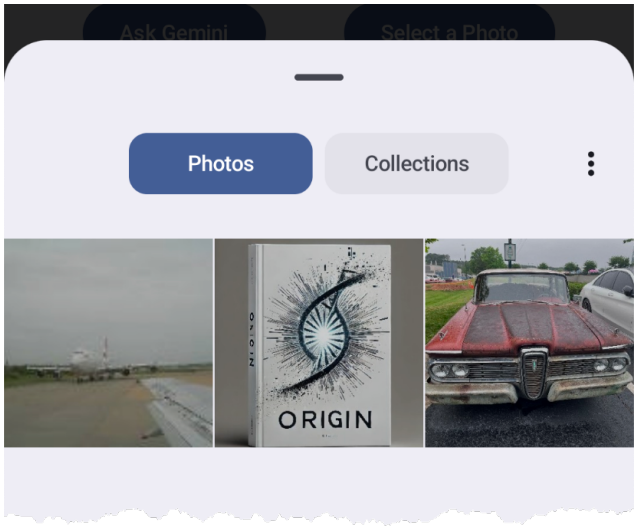


Figure 74-1

If you are using an emulator, the photo library may be empty. To add photos, drag and drop them from the host computer onto the emulator home screen. Next, open the Files app in the emulator, navigate to the Downloads folder, and perform a long press on the image to select it (marked A in Figure 74-2). Select the menu button (B) followed by the *Open with* option (C):

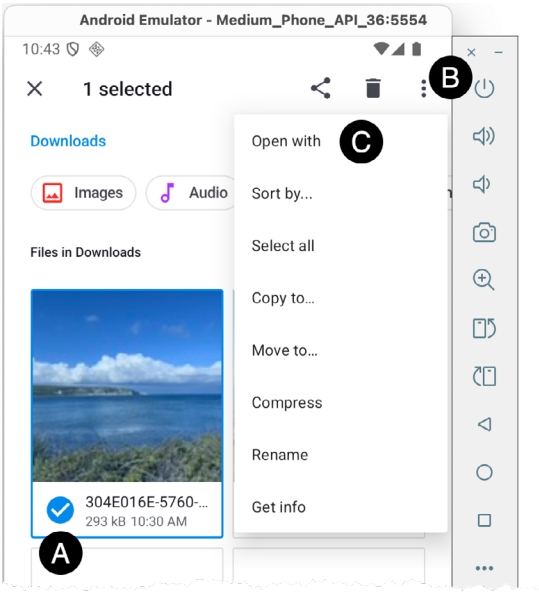


Figure 74-2

When prompted, select the Photos app. Stop and restart the GeminiDemo app, and open the media picker to access the added image.

When an image is selected, the app displays it on the ImageView located above the prompt text field, and the corresponding Uri is stored in the *selectedImageUri* variable declared within the *TextImageFragment.java* class file:

```
public class TextImageFragment extends Fragment {
.
.
    private Uri selectedImageUri;
.
.
    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        imagePickerLauncher = registerForActivityResult(
                  new ActivityResultContracts.GetContent(),
                uri -> {
                    if (uri != null) {
                        binding.imageView.setImageURI(uri);
                        selectedImageUri = uri;
                    }
                });
    }
.
.
```

74.3 Calling the Gemini API

The Ask Gemini button is pre-configured to call a method called *askGemini()*, the code for which reads as follows:

```
public void askGemini(String message) {
    binding.resultText.setText("Generating response...");
    Bitmap bitmap = uriToBitmap(requireContext(), selectedImageUri);
    viewModel.sendTextAndImage(bitmap, message);
}
```

The method displays a message that the request is processing before converting the selected image to a bitmap. Both the prompt text and bitmap are then passed to the stub *sendTextAndImage()* method declared in the *GeminiViewModel.java* file:

```
public void sendTextAndVideo(Context context, Uri videoUri, String prompt) {

}
```

Edit the *GeminiViewModel.java* file and make the following modifications:

```
.
.
import androidx.annotation.NonNull;
import com.google.common.util.concurrent.FutureCallback;
import com.google.common.util.concurrent.Futures;
import com.google.common.util.concurrent.ListenableFuture;
import com.google.firebase.ai.type.Content;
import com.google.firebase.ai.type.GenerateContentResponse;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class GeminiViewModel extends ViewModel {

    final private ExecutorService executor = Executors.newSingleThreadExecutor();
.
.
    public void sendTextAndImage(Bitmap bitmap, String prompt) {
        Content.Builder contentBuilder = new Content.Builder();

        if (bitmap != null) {
            contentBuilder.addImage(bitmap);
        }

        contentBuilder.addText(prompt);
        Content content = contentBuilder.build();

        ListenableFuture<GenerateContentResponse> response =
                                model.generateContent(content);
```

```
        Futures.addCallback(response,
                new FutureCallback<GenerateContentResponse>() {
            @Override
            public void onSuccess(GenerateContentResponse result) {
                genResult.postValue(result.getText());
            }

            @Override
            public void onFailure(@NonNull Throwable t) {
                // Handle error
            }
        }, executor);
}
```

The method creates a content builder instance and populates it with the prompt text and the image bitmap if one was provided. The Futures wrapper classes and an executor service are then used to call the *generateContent(content)* method asynchronously, thereby allowing the app to remain responsive while awaiting a response. On a successful operation, the *onSuccess()* method is called by the future listener and passed the generated result. The result is then posted to the *genResult* LiveData instance, where it will be detected by the observer in the TextImageFragment class and displayed to the user.

74.4 Testing the app

The first stage of the GeminiDemo project is now ready for testing. Run the app, ask Gemini a question, and await a response. Next, select a photo and ask Gemini questions related to the image:

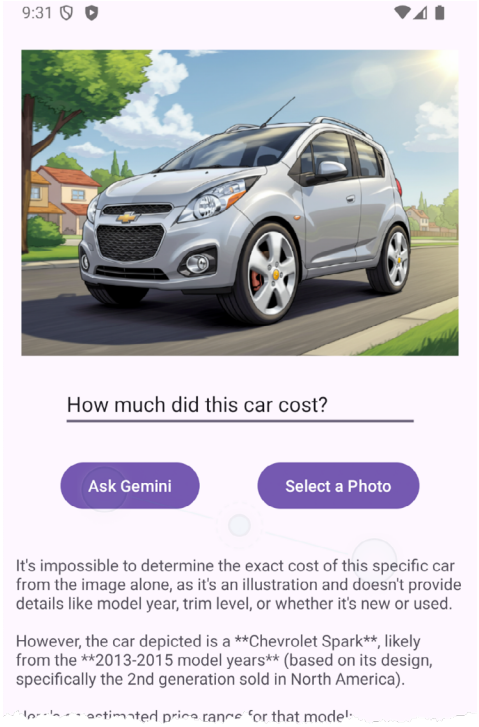


Figure 74-3

74.5 Migrating to chat

If we were to ask a follow-up question on the same topic without providing context, the model would be unable to answer it because it doesn't remember the previous interaction. As the final step in this chapter, we will convert the example to a chat session. Edit the *GeminiViewModel.kt* file and make the following changes:

```
.
.
import com.google.firebase.ai.java.ChatFutures;
.
.
GenerativeModelFutures model = GenerativeModelFutures.from(ai);

ChatFutures chatSession = model.startChat();

public void sendTextAndImage(Bitmap bitmap, String prompt) {
    Content.Builder contentBuilder = new Content.Builder();
.
.
    ListenableFuture<GenerateContentResponse> response =
            model.generateContent(content);
            chatSession.sendMessage(content);
.
.
```

Rerun the app and repeat the previous steps to select an image and ask a question. Ask a follow-up question without providing context and verify that the model can provide an answer:

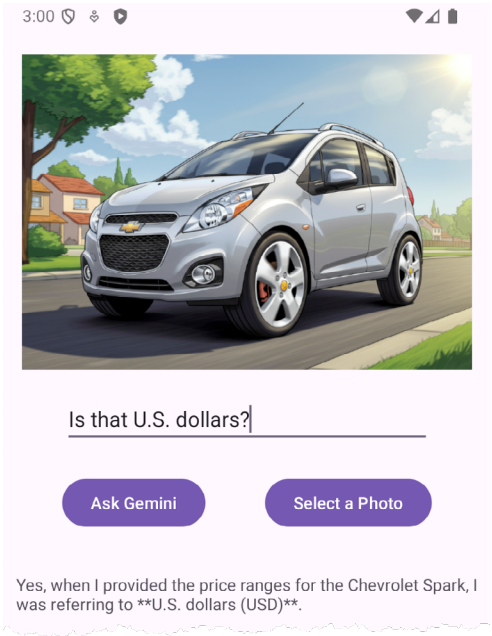


Figure 74-4