**Outfit of the Day (https://github.com/Akhil-Kokkula/OutfitOfTheDay)**

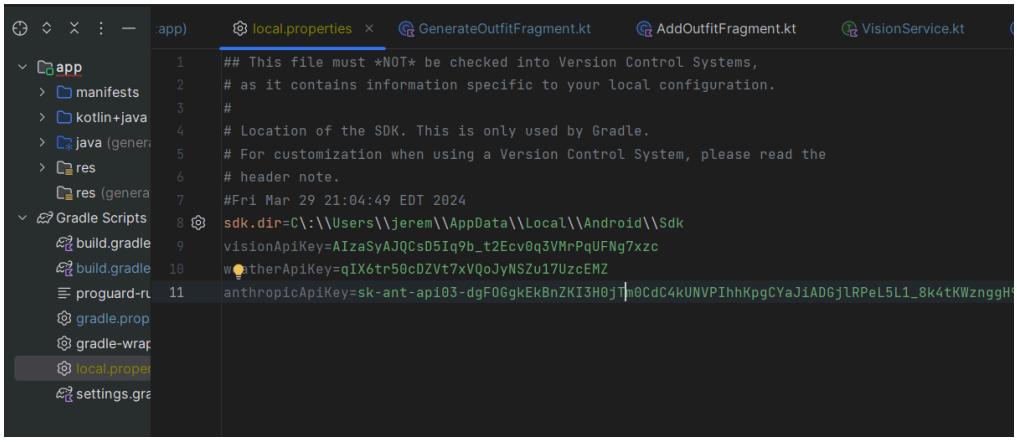
Authors: Jeremy, Akhil, Prathmesh, and Zuizz

**Purpose:** Everyone spends energy in choosing the best outfit for a given occasion. However, in many cases, people dedicate too much time into deciding their outfits for a given day. In addition, people often forget to factor in certain events in their day, the weather, and other features when determining their outfits, resulting in regret and unneeded stress. Through Outfit of the Day, a given user will be able to utilize weather and calendar information to create smart outfits, serving as a solution.

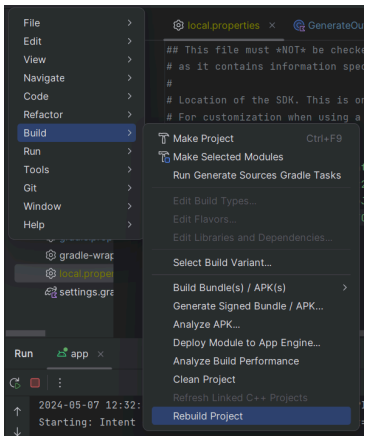
**API Key Setup:**

Copy these keys into local.properities:

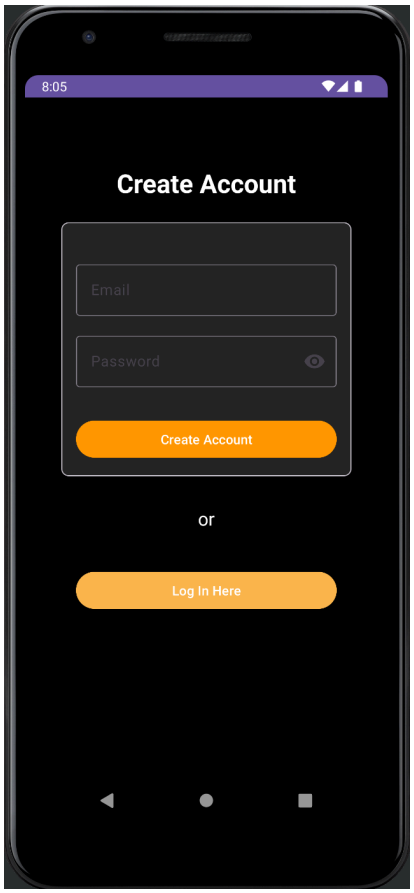
```
visionApiKey=AIzaSyClw-8cPXAOCg9hPqVR74By9mGU-q-yl6c
weatherApiKey=qIX6tr50cDZVt7xVQoJyNSZu17UzcEMZ
anthropicApiKey=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
```

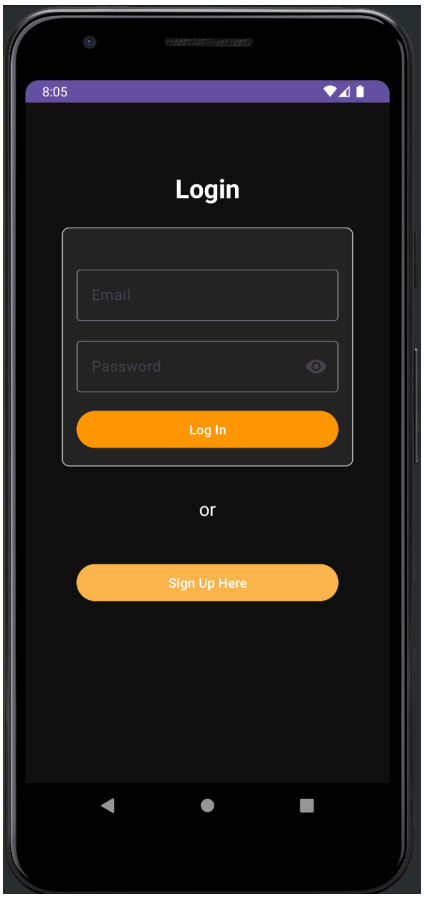



If there are problems loading the variables, make sure to sync the gradle files and rebuild the project:



## Authentication (SignUpFragment.kt and LoginFragment.kt)




The SignUpFragment.kt and LoginFragment.kt provides authentication for the app. Authentication is powered by Firebase Authentication.

### 1. createAccount Function

This function in the SignUpFragment.kt creates the user's account with email and password input text using Firebase Authentication.

```
private fun createAccount() {
    val email = emailInputField.text.toString().trim()
    val password = passwordInputField.text.toString().trim()

    if (email == "" || password == "") {
        Toast.makeText(getContext(), getString(R.string.signUp_incomplete),
Toast.LENGTH_SHORT).show();
        return
    }

    auth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                Log.d(TAG, "successfully created new user account")
                goToLoginFragment()
```

```
            } else {

Toast.makeText(getContext(),getString(R.string.signUp_accountCreationFail) ,
Toast.LENGTH_SHORT).show();
                Log.e(TAG, "could not create new user account due to ",
task.exception)
            }
        }
}
```

**2. logIn Function**

This function in the LoginFragment.kt signs the user in using Firebase Authentication by making sure the user has an account.

```
private fun logIn() {
    val email = emailInputField.text.toString().trim()
    val password = passwordInputField.text.toString().trim()

    auth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                Log.d(TAG, "successfully logged in user")
                goToHomeFragment()
            } else {
                Toast.makeText(getContext(), getString(R.string.failed_Login),
Toast.LENGTH_SHORT).show()
                Log.e(TAG, "could not log in user", task.exception)
            }
        }
}
```

**Home Frament**

After the user logs into the application, they will be redirected to the home fragment which serves as the application's landing page. The landing page displays the user's username, weather data, and wardrobe statistics.

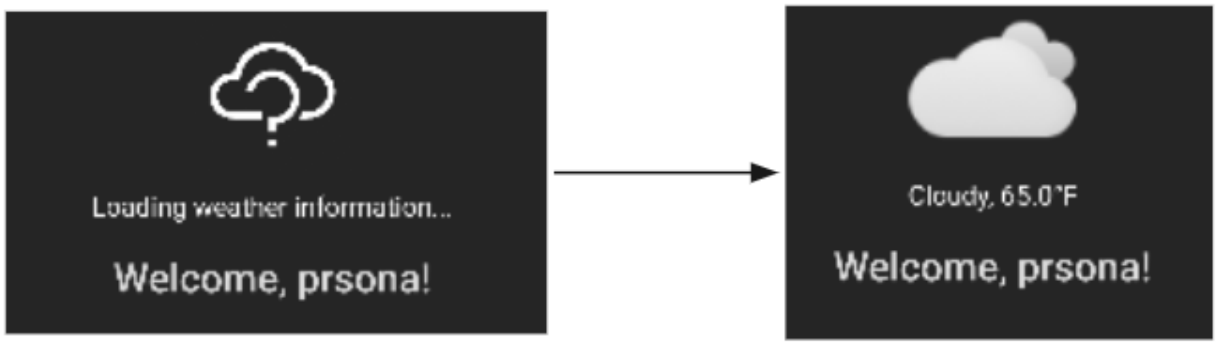
**1. Welcome Message**

This fragment, like many, captures the user's username by getting an instance of their Firebase authentication account. With this information, the application fetches the username of the user, displaying it on the application.

**2. Weather Information**

Upon landing on the landing page, the user is prompted to allow the application to access location data via device permissions if not already allowed. Once the permissions have been granted, a coroutine is set and passes the user's location data to the Tomorrow.io API. Upon

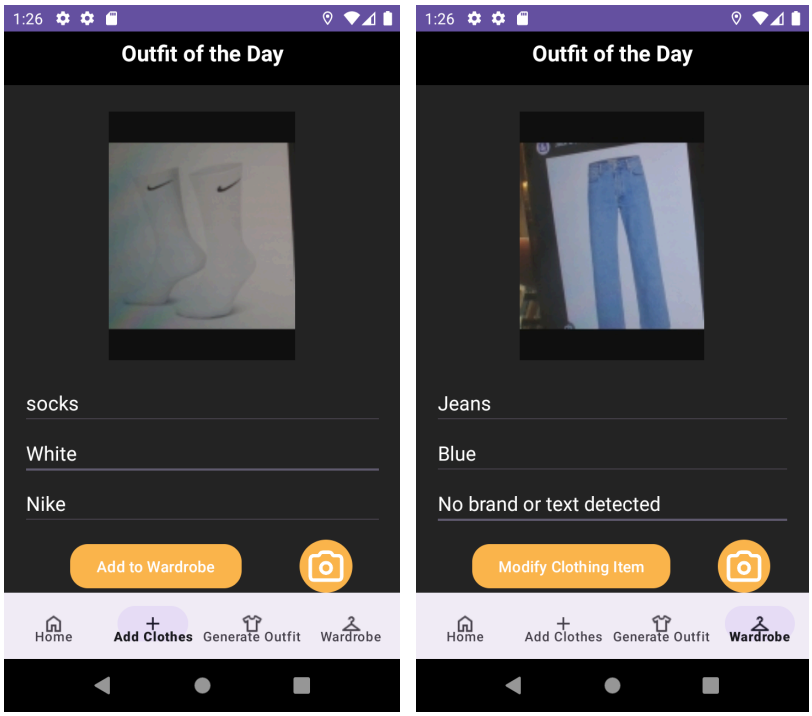
receiving the response, our application not only stores the temperature, but also the type of weather (sunny, cloudy, heavy rain, etc.). This information is then used to update the UI of the home fragment, showcasing the weather information.




### 3. Wardrobe Distribution Chart

When the user reaches the landing page, the user's firebase authentication account is used to fetch a count of each category of clothing they have. Using the count of each category, the percentage of clothing in each category is calculated. Using these percentages, a pie chart is displayed.

**AddOutfitFragment.kt**



This fragment allows users to take a picture of their clothing and Google Cloud Vision will generate what type of clothing item it is, the color, and text/brands. From there, the user will have to confirm it is correct and then add it to their wardrobe. Users can also manually input the clothing item. Moreover, the user can modify an existing clothing item which will be done by clicking on the item and holding on until a “modify” popup occurs in the wardrobe fragment.

**1. getClosestColorName function**

Since Google Cloud Vision returns the color of the picture as RGB, we need to convert this for easier readability for the user. We can do this by calculating the Euclidean distance between a given RGB color and predefined colors to determine and return the closest color name.

```
private fun getClosestColorName(r: Int, g: Int, b: Int): String {
    return colorList.minByOrNull { color ->
        // Calculate Euclidean distance between the color parameter and each
color in the list
        sqrt(((color.r - r) * (color.r - r) + (color.g - g) * (color.g - g) +
(color.b - b) * (color.b - b)).toDouble())
    }?.name ?: "Unknown Color"
}
```

**2. modifyOutfitData function**

Allows the user to modify existing clothing items from their account. In the function, it would check for the user id and then take in arguments the label, color, and branch which is passed from the wardrobe fragment. Moreover, it will change the button text from “add to fragment” to “modify clothing item.” By clicking the button, it will modify the existing item on Firebase.

```
private fun modifyOutfitData(itemId: String) {
    val userId = FirebaseAuth.getInstance().currentUser?.uid
    if (userId == null) {
        Toast.makeText(context, "User not logged in", Toast.LENGTH_SHORT).show()
        return
    }

    // Extract data from input fields
    val label = editTextLabel.text.toString().trim()
    val color = editTextColor.text.toString().trim()
    val brand = editTextBrand.text.toString().trim()

    if (label.isEmpty() || color.isEmpty() || brand.isEmpty() ||
capturedImageBitmap == null) {
        Toast.makeText(context, "Please fill all fields and capture an image.",
Toast.LENGTH_SHORT).show()
        return
    }

    val imageBase64 = encodeImageToBase64(capturedImageBitmap!!)
    val category = getCategoryFromLabel(label)
```

```
    // Create a map of data to update
    val outfitUpdates = mapOf(
        "label" to label,
        "color" to color,
        "brand" to brand,
        "category" to category,
        "imageBase64" to imageBase64
    )

    // Reference to the specific outfit item in Firebase
    val outfitRef =
FirebaseDatabase.getInstance().getReference("users/$userId/outfits/$itemId")

    // Update the data in Firebase
    outfitRef.updateChildren(outfitUpdates).addOnCompleteListener { task ->
        if (task.isSuccessful) {
            Toast.makeText(context, "Outfit updated successfully!",
Toast.LENGTH_SHORT).show()
            // Optionally navigate back or clear the form
        } else {
            Toast.makeText(context, "Failed to update outfit.",
Toast.LENGTH_SHORT).show()
        }
    }
}
```

**3. analyzeImage Function**

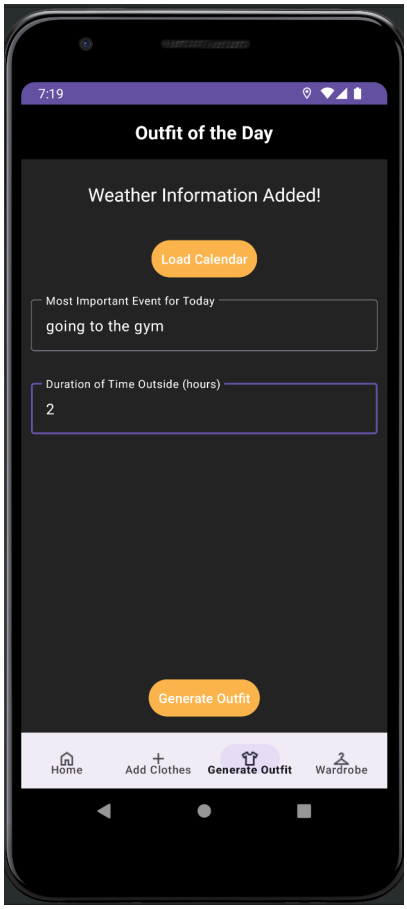
The analyzeImage function processes an image to identify clothing-related information. It compresses the image, converts it to a Base64 string, and sends it to the Google Vision API for analysis, requesting the detection of labels, logos, text, and colors. When the API returns results, the function filters and extracts useful details such as clothing types and brands, and identifies the dominant color in the image. These results are then displayed in the app, helping users quickly categorize and describe their clothing items without manual input. The challenge that we faced here was that Google Cloud Vision did not always return the most accurate information. Although it did offer a confidence score, selecting the highest confidence was not always the best path. For example, when showing a picture of a shirt, the API would return “t-shirt (65%)” and “pants (80%).” Therefore, we have decided to allow the user to ultimately decide the inputs after getting “suggestions” from the API.

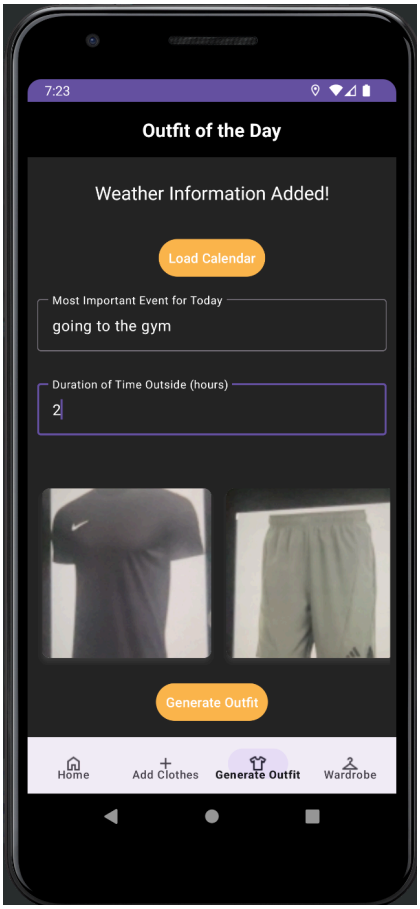
```
fun analyzeImage(imageBitmap: Bitmap) {
    val visionService = setupRetrofit()
    val byteArrayOutputStream = ByteArrayOutputStream()
    imageBitmap.compress(Bitmap.CompressFormat.JPEG, 90, byteArrayOutputStream)
    val base64Image = Base64.encodeToString(byteArrayOutputStream.toByteArray(),
Base64.DEFAULT)

    val request = VisionModel.VisionRequest(
        requests = listOf(
            VisionModel.AnnotateImageRequest(
```

```
                image = VisionModel.Image(content = base64Image),
                features = listOf(
                    VisionModel.Feature(type = "LABEL_DETECTION", maxResults =
20),
                    VisionModel.Feature(type = "LOGO_DETECTION", maxResults =
5),
                    VisionModel.Feature(type = "IMAGE_PROPERTIES", maxResults =
1),
                    VisionModel.Feature(type = "TEXT_DETECTION", maxResults = 5)
                )
            )
        )
    )

    visionService.annotateImage(request).enqueue(object :
retrofit2.Callback<VisionModel.VisionResponse> {
        override fun onResponse(call:
retrofit2.Call<VisionModel.VisionResponse>, response:
retrofit2.Response<VisionModel.VisionResponse>) {
            if (response.isSuccessful) {
                response.body()?.responses?.firstOrNull()?.let {
                    val gson = GsonBuilder().setPrettyPrinting().create()
                    Log.d("AddOutfitFragment", "Vision API Response:
${gson.toJson(it)}")

                    val descriptions = it.labelAnnotations?.map { label ->
                        label.description.toLowerCase(Locale.ROOT)
                    }?.filter { desc ->
                        allowedClothingTypes.any { type -> type.equals(desc,
ignoreCase = true) }
                    }?.joinToString(", ")

                    val brandNames = it.logoAnnotations?.map { logo ->
logo.description }?.joinToString(", ")
                    val textDescriptions = it.textAnnotations?.map { text ->
text.description }?.joinToString(" ")

                    // Extracting dominant color and converting it to a named
color
                    val dominantColorInfo =
it.imagePropertiesAnnotation?.dominantColors?.colors?.maxByOrNull { color ->
color.pixelFraction }
                    val closestColorName = dominantColorInfo?.color?.let { color
->
                        getClosestColorName(color.red, color.green, color.blue)
// Converting RGB to color name
                    } ?: "No color detected"

                    activity?.runOnUiThread {
```

```
                        editTextLabel.setText(descriptions ?: "No label
detected")
                        editTextColor.setText(closestColorName) // Using the
named color

                        editTextBrand.setText(listOfNotNull(brandNames,
textDescriptions).joinToString(" ").ifBlank { "No brand or text detected" })
                    }
                }
            } else {
                Log.e("AddOutfitFragment", "API Error Response:
${response.errorBody()?.string()}")
                Toast.makeText(context, "API request failed with code:
${response.code()}", Toast.LENGTH_LONG).show()
            }
        }
        override fun onFailure(call: retrofit2.Call<VisionModel.VisionResponse>,
t: Throwable) {
            Toast.makeText(context, "API request failed: ${t.message}",
Toast.LENGTH_LONG).show()
        }
    })
}
```

## GenerateOutfitFragment.kt



This fragment allows users to generate an outfit from their wardrobe based on the user's occasion and duration of the occasion. The user can also load their google calendar (and other features) for more context of the user's day.

**1. generateOutfitAction Function**

This function is the driver for the generate outfit action. A user can generate an outfit only once the user has entered their occasion, duration for occasion, and the weather information is added. Some challenges for this function was handling asynchronous code correctly for good user experience.

```
private fun generateOutfitAction() {
    if (occasionInputText.text.toString() == "") {
        Toast.makeText(context, getString(R.string.generateOutfitEventFail),
Toast.LENGTH_SHORT).show()
    } else if (durationInputText.text.toString() == "" ||
durationInputText.text.toString().toInt() == 0) {
        Toast.makeText(context,
getString(R.string.generateOutfitEventDurationFail) ,
Toast.LENGTH_SHORT).show()
    } else if (weatherJSONObject == null) {
        Toast.makeText(context, getString(R.string.generateOutfitWeatherFail) ,
Toast.LENGTH_SHORT).show()
    } else {
        loadingIndicator.visibility = View.VISIBLE
        getHourlyWeatherData()
        val aiTextStrBuilder = StringBuilder()
        aiTextStrBuilder.append("You are a fashion stylist and you must give the
user a full outfit for the day. ")
        aiTextStrBuilder.append("Here is the expected output format you need to
provide and please only answer in this format:\n" +
                "item -NvcyBYui5a1z-UH8Yfh: T-shirt, White, Casual\n" +
                "item -NvcyBYui5a1z-UH7Ghy: Jeans, Light Blue, Casual\n" +
                "item -NvcyBYui5a1z-UH6Ynh: Sneakers, White, Casual\n" +
                "item -NvcyBYui5a1z-UH5Gfl: Aviator Sunglasses, Black,
Casual\n\n")
        aiTextStrBuilder.append("Weather Information:\n")
        aiTextStrBuilder.append(hourlyWeatherJSONString)
        aiTextStrBuilder.append("\n\n")
        aiTextStrBuilder.append("The most important event of the user's day:\n")
        aiTextStrBuilder.append(occasionInputText.text.toString())
        aiTextStrBuilder.append("\n\n")

        //Calendar Information:
        aiTextStrBuilder.append("The itinerary for the user today:\n")
        aiTextStrBuilder.append(stringOfEvents)
        aiTextStrBuilder.append("\n\n")
        aiTextStrBuilder.append("User wardrobe:\n")

        fetchDataFromDatabase { items ->
```

```
            wardrobeList = items
            for (clothingItem in items) {
                Log.d("GenerateOutfitFragment", "item ${clothingItem.id}:
${clothingItem.label}, ${clothingItem.color}, ${clothingItem.brand}")
                aiTextStrBuilder.append("item ${clothingItem.id}:
${clothingItem.label}, ${clothingItem.color}, ${clothingItem.brand}\n")
            }

            aiTextStrBuilder.append("\n\n")
            aiTextStrBuilder.append("Please generate an outfit for me")

            val aiTextStr = aiTextStrBuilder.toString()
            println("will send to ai")
            println(aiTextStr)

            sendAndReceiveMessageFromClaude(aiTextStr) { outfitAIResponseStr ->
                val regex = Regex("""item ([\w-]+):""")
                val clothingIds = outfitAIResponseStr
                    .split("\n")
                    .filter { it.startsWith("item") }
                    .mapNotNull { regex.find(it)?.groupValues?.get(1) }
                println(clothingIds)

                val outfitImages = mutableListOf<ClothingItem>()
                wardrobeList.forEach { item ->
                    if (item.id in clothingIds) {
                        outfitImages.add(item)
                    }
                }

                GlobalScope.launch(Dispatchers.Main) {
                    loadingIndicator.visibility = View.GONE
                    outfitGalleryAdapter.updatingOutfitList(outfitImages)
                    outfitPhotoGallery.scrollToPosition(0)
                }
            }
        }
    }
}
```

**2. Prompt Engineering**

We prompt engineer the text sent to Claude API by specifying its role and expected output format.

```
aiTextStrBuilder.append("You are a fashion stylist and you must give the user a
full outfit for the day. ")
        aiTextStrBuilder.append("Here is the expected output format you need to
provide and please only answer in this format:\n" +
                "item -NvcyBYui5a1z-UH8Yfh: T-shirt, White, Casual\n" +
                "item -NvcyBYui5a1z-UH7Ghy: Jeans, Light Blue, Casual\n" +
```

```
                "item -NvcyBYui5a1z-UH6Ynh: Sneakers, White, Casual\n" +
                "item -NvcyBYui5a1z-UH5Gfl: Aviator Sunglasses, Black,
Casual\n\n")
```

### 3. Claude API Text Input

We then share today's weather information during the duration of the occasion, the user's itinerary from their Google Calendar via the API, and the user's wardrobe by fetching the information from the Firebase Database to the Claude API. Next, we send the clear request to Claude by asking it to generate an outfit.

```
aiTextStrBuilder.append("Weather Information:\n")
        aiTextStrBuilder.append(hourlyWeatherJSONString)
        aiTextStrBuilder.append("\n\n")
        aiTextStrBuilder.append("The most important event of the user's day:\n")
        aiTextStrBuilder.append(occasionInputText.text.toString())
        aiTextStrBuilder.append("\n\n")

        //Calendar Information:
        aiTextStrBuilder.append("The itinerary for the user today:\n")
        aiTextStrBuilder.append(stringOfEvents)
        aiTextStrBuilder.append("\n\n")
        aiTextStrBuilder.append("User wardrobe:\n")

        fetchDataFromDatabase { items ->
            wardrobeList = items
            for (clothingItem in items) {
                Log.d("GenerateOutfitFragment", "item ${clothingItem.id}:
${clothingItem.label}, ${clothingItem.color}, ${clothingItem.brand}")
                aiTextStrBuilder.append("item ${clothingItem.id}:
${clothingItem.label}, ${clothingItem.color}, ${clothingItem.brand}\n")
            }

            aiTextStrBuilder.append("\n\n")
            aiTextStrBuilder.append("Please generate an outfit for me")
```

### 4. Parsing Claude API Response

The last step is to receive the response from Claude and parse the response to get the ID's of the wardrobe the API recommends. We then update the horizontal RecyclerView with the user's wardrobe items that Claude has recommended.

```
sendAndReceiveMessageFromClaude(aiTextStr) { outfitAIResponseStr ->
                val regex = Regex("""item ([\w-]+):""")
                val clothingIds = outfitAIResponseStr
                    .split("\n")
                    .filter { it.startsWith("item") }
                    .mapNotNull { regex.find(it)?.groupValues?.get(1) }
                println(clothingIds)

                val outfitImages = mutableListOf<ClothingItem>()
                wardrobeList.forEach { item ->
```

```
            if (item.id in clothingIds) {
                outfitImages.add(item)
            }
        }

        GlobalScope.launch(Dispatchers.Main) {
            loadingIndicator.visibility = View.GONE
            outfitGalleryAdapter.updatingOutfitList(outfitImages)
            outfitPhotoGallery.scrollToPosition(0)
        }
```

**5. Tomorrow.io API Usage**

The Generate Outfit fragment utilizes the Tomorrow.io API to fetch weather information using the user's location. By doing so, our application uses weather as a feature in determining the best outfit for a given condition. For example, if a user was going to class on a rainy day, the application would automatically fetch and take that weather data into account to ensure the user had an appropriate outfit.

Firstly, the application requests device permissions to fetch the user's location.

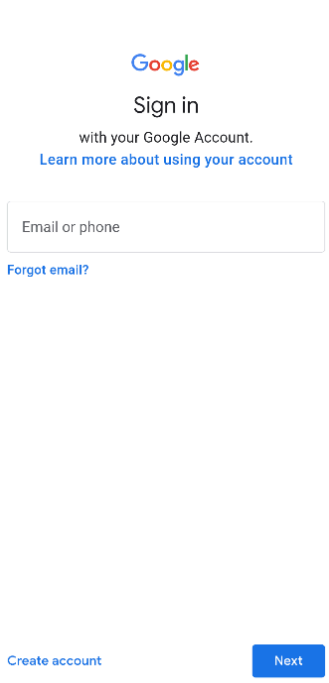
```
PrathmeshS
private fun startLocationGathering() {
    locationManager = requireContext().getSystemService(Context.LOCATION_SERVICE) as LocationManager
    if (ContextCompat.checkSelfPermission(requireContext(), Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
        // Request location updates
        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,  minTimeMs: 0,  minDistanceM: 0f, locationListener)
    } else {
        //Ask user for permission and then load location
        requestPermissions(arrayOf(Manifest.permission.ACCESS_FINE_LOCATION), LOCATION_PERMISSION_REQUEST_CODE)
    }
}
```

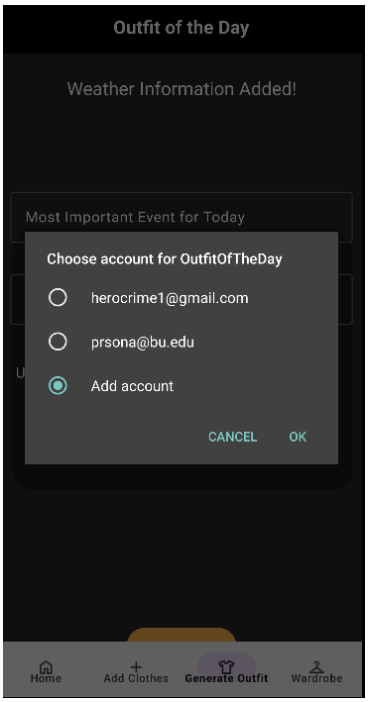
After those permissions have been granted, the user's latitude and longitude is captured and fed into our Tomorrow.io API. Within the "run" function in our Generate Outfits fragment, the API is called using the user's location and a variety of API endpoints. The JSON response returned is then saved and parsed to retrieve temperature, precipitation, and humidity data, all of which are fed into Claude API during outfit generation.

**6. Google Calendar API Usage**

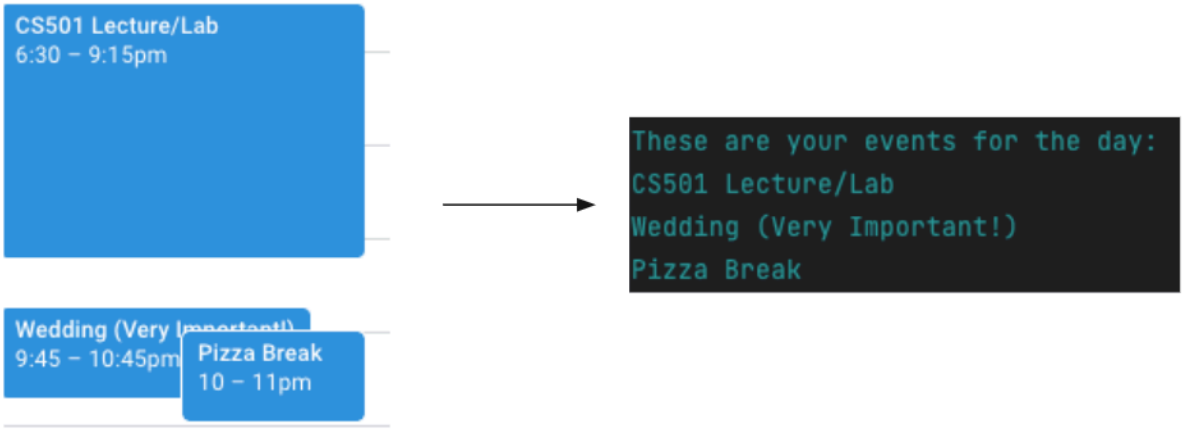
The Google Calendar API is used to retrieve a user's itinerary for the given day, serving as an additional feature during outfit generation.

Upon a user's request to load their calendar, they are prompted to choose a variety of Google accounts they have already signed up with on their device. If none are present, the user is directed to add a google account.




Upon signing in or choosing an account, a coroutine is called to fetch the user's itinerary from their google calendar. Through parsing and specifying variables to the Google Calendar API, the application stores all needed information and marks it ready for use during outfit generation.




**ClothingItem.kt**

This file contains the definition for the clothing item data class. Each clothing item in the wardrobe can consist of an id (for database purposes), label, color, brand, category (what type of clothing is it, ie tops, bottoms, etc), the imagebase64 string for the image, and a url for cases when the picture of the clothing item is some link.

**WardrobeFragment.kt**

This fragment is where the user can view their clothing items that they have added to their wardrobe. The fragment utilizes WardrobeAdapter to bind clothing items to views. It displays the items that are stored in the Firebase Database. The function below fetches the data items from the database and displays them in the user's wardrobe screen:

```
private fun fetchDataFromDatabase() {
   databaseReference.addValueEventListener(object : ValueEventListener {
       override fun onDataChange(dataSnapshot: DataSnapshot) {
           allItems.clear()
           for (snapshot in dataSnapshot.children) {
               val item = snapshot.getValue(ClothingItem::class.java)
               item?.let {
                   it.id = snapshot.key
                   allItems.add(it)
               }
           }
           updateUI(allItems)  // Initial load with all items
       }

       override fun onCancelled(databaseError: DatabaseError) {
           Log.e("WardrobeFragment", "Failed to read wardrobe data",
databaseError.toException())
       }
   })
}
```

The user has the ability to search through their wardrobe with the search bar at the top, where they can search for keywords associated with their clothing item, including its label, color, brand, etc. Below are the two functions that set up the search view and filter the wardrobe/update the UI according to the search text submitted.

```
private fun setupSearchView() {
   binding.searchView.setOnQueryTextListener(object :
SearchView.OnQueryTextListener {
       override fun onQueryTextSubmit(query: String?): Boolean {
           return false // Let the SearchView handle the default behavior of
the query text submission
       }

       override fun onQueryTextChange(newText: String?): Boolean {
           filterWardrobeBySearch(newText)
           return true
       }
```

```
    })
}

private fun filterWardrobeBySearch(query: String?) {
    val filteredList = if (!query.isNullOrEmpty()) {
        allItems.filter {
            it.label?.contains(query, ignoreCase = true) == true ||
                    it.color?.contains(query, ignoreCase = true) == true ||
                    it.brand?.contains(query, ignoreCase = true) == true
        }
    } else {
        allItems // Return all items if search query is empty
    }
    updateUI(filteredList)
}
```

The user can also choose to view their wardrobe using selected filters. They can select to filter their wardrobe to only view “all”, “hats”, “tops”, “bottoms”, “footwear”, and “miscellaneous” clothing items. This can help the user look at certain items that they may be interested in. Below is the function for filtering the wardrobe, where it uses the category that the user selected to display only clothing items of that category.

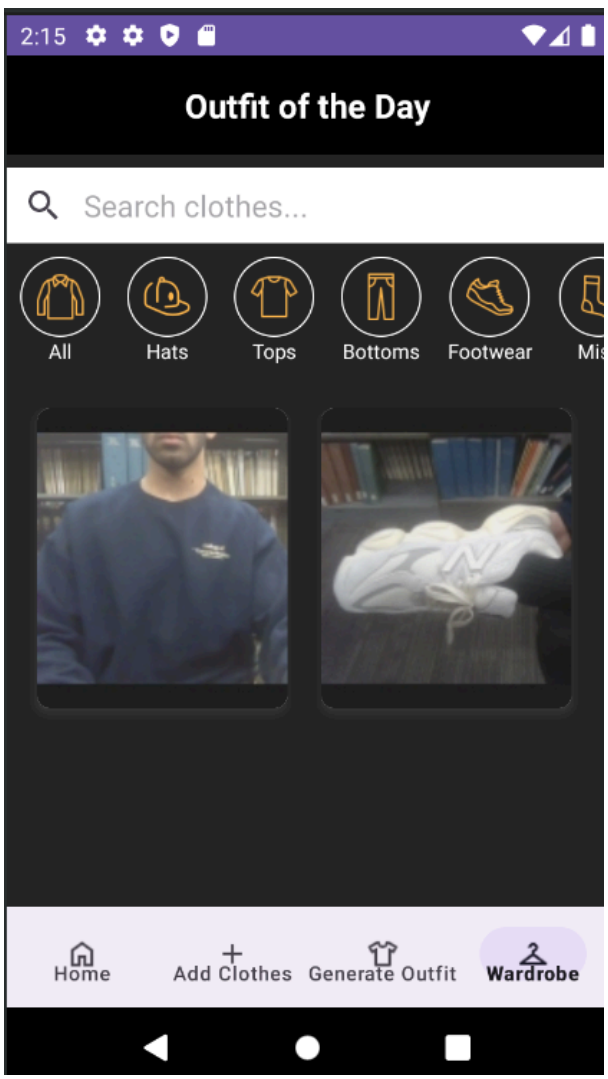
```
private fun filterWardrobe(category: String) {
    val filteredItems = if (category == "all") {
        allItems
    } else {
        allItems.filter { it.category?.equals(category, ignoreCase = true) ?:
false }
    }
    updateUI(filteredItems)
}
```

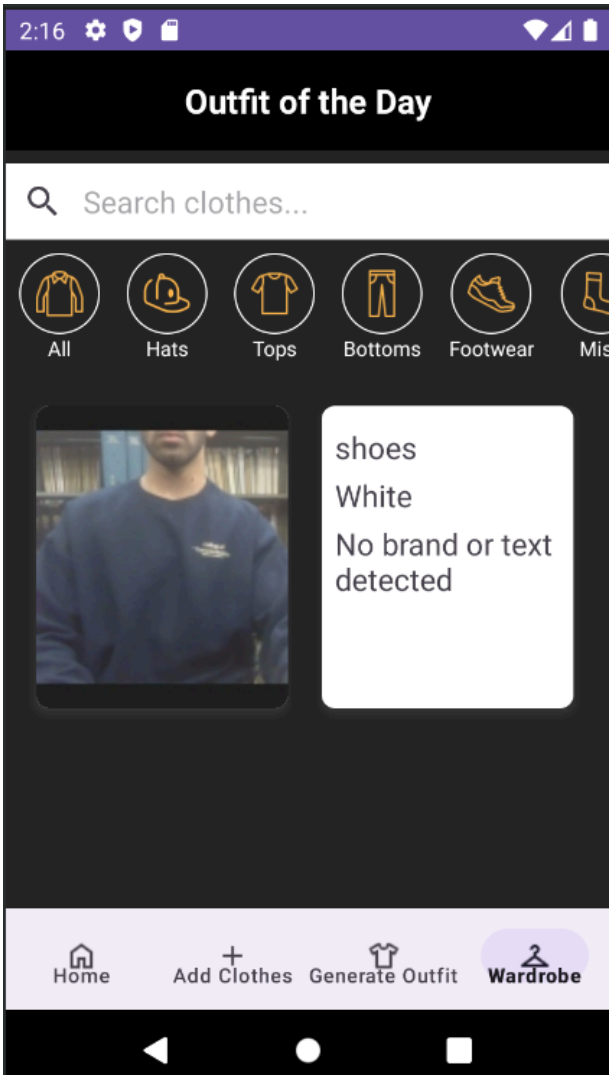
Remember, each clothing item has an associated category with it. The category is assigned to a clothing item based on the clothing item’s name (there is a set of allowed/possible clothing item labels) when it is added to the wardrobe in AddOutfitFragment.kt (function getCategoryFromLabel shown below).

```
private fun getCategoryFromLabel(label: String): String {
    val normalizedLabel = label.toLowerCase(Locale.ROOT).trim()
    return when {
        "t-shirt" in normalizedLabel || "shirt" in normalizedLabel || "blouse"
in normalizedLabel || "jacket" in normalizedLabel || "coat" in normalizedLabel
|| "hoodie" in normalizedLabel || "cardigan" in normalizedLabel || "blazer" in
normalizedLabel -> "Tops"
        "skirt" in normalizedLabel || "shorts" in normalizedLabel || "leggings"
in normalizedLabel -> "Bottoms"
        "pant" in normalizedLabel || "jeans" in normalizedLabel -> "Bottoms"
        "hat" in normalizedLabel || "cap" in normalizedLabel -> "Hats"
```

```
        "shoes" in normalizedLabel || "boots" in normalizedLabel || "sandals" in
normalizedLabel || "high heels" in normalizedLabel || "loafers" in
normalizedLabel-> "Footwear"
        // Add more conditions as necessary
        else -> "Miscellaneous"
    }
}
```

The user also has the ability to modify and delete clothing items from their wardrobe. They can do this by clicking on a certain clothing item, after which the information for that clothing item will be displayed (shown below).
Clicking on the shoe clothing item:



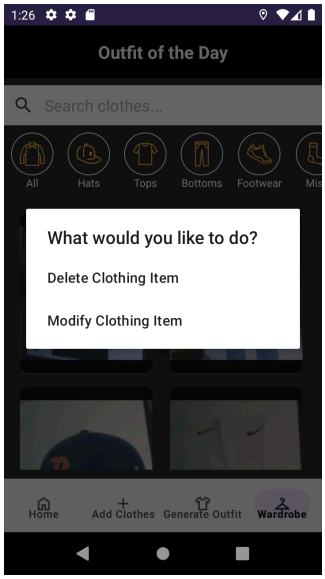
The below code handles the deleting and modifying of clothing items. If the delete option is selected, the item is deleted from the database, and if the modify option is selected, then the user is taken to the AddOutfitFragment where they can modify the clothing item's details.

```
private fun handleItemAction(item: ClothingItem, action: String) {
    when (action) {
        "delete" -> showDeleteConfirmationDialog(item)
        "modify" -> navigateToAddOutfitFragment(item)
    }
}
```

The below code deletes an item from the database.

```
private fun deleteItemFromFirebase(item: ClothingItem) {
    val userId = auth.currentUser?.uid
    if (userId != null && item.id != null) {

FirebaseDatabase.getInstance().getReference("users/$userId/outfits/${item.id}")
            .removeValue()
            .addOnSuccessListener {
                allItems.remove(item)
                adapter.notifyDataSetChanged()
                Log.d("WardrobeFragment", "Item deleted successfully")
            }
            .addOnFailureListener {
                Log.e("WardrobeFragment", "Failed to delete item", it)
            }
    }
}
```

This code, when run, results in the following functionality for the user when they click and hold a given item in the wardrobe fragment.



The below code takes the user to the AddOutfitFragment where they can modify the clothing item's details.

```
private fun navigateToAddOutfitFragment(item: ClothingItem) {
    val fragment = AddOutfitFragment().apply {
        arguments = Bundle().apply {
            putString("item_id", item.id) // pass other details as needed
            putString("label", item.label)
            putString("color", item.color)
            putString("brand", item.brand)
            putString("category", item.category)
            putString("imageBase64", item.imageBase64)
            putString("imageUrl", item.imageUrl)
        }
    }
    activity?.supportFragmentManager?.beginTransaction()?.apply {
        replace(R.id.nav_host_fragment, fragment)
        addToBackStack(null)
        commit()
    }
}
```

## Small Bonuses

### Multiple Locales

Our application is accessible in two locales, that being predominantly English speaking countries and Japan. Our group traversed the entirety of our application to remove hard-coded strings and used string resources to display text on our application. By doing so, we ended up with a string resource file almost 100 lines long. After finishing our default string resource file, we added a new string resource file for the Japan locale and added Japanese translations for each string.

By doing so, if a user switches their primary language to Japanese, the application will be loaded using Japanese strings, providing increased accessibility for those that might not speak English.

### Device Permissions

Our application properly requests the user for their contacts and location data using device permissions. The contacts and location permissions are granted when a user successfully retrieves their Google calendar and weather data, respectively.

**Recycler View**

Our application properly utilizes recycler views when loading in a user's wardrobe and generating outfits. These can be found on the "Generate Outfit" and "Wardrobe" fragments.

**Menus**

Our application utilizes a bottom navigation bar with icons to help users navigate through the different fragments of our application. In addition, our "Wardrobe" fragment also hosts a small menu to filter clothing based on categories, making it easier for a user to look at any particular type of clothing.

**Use Cases**

In our initial presentation pitching the idea of *Outfit of the Day*, we had three use cases for our application, each emphasizing a different demographic. In our examples, we stated that students, professionals, and adults in general all shared the same problem when it came to deciding on an outfit for a given day or occasion: All of them took too long looking through their wardrobe and remembering their day's plans to create outfits that were not the best they could be.

With our application now being finished, we are proud to say that all of our initial use cases have been implemented. No matter what individual you are, we make it easy to automate data collection and outfit generation, saving you time and work.

## References


- https://chat.openai.com/
- Google Calendar API using Kotlin
- Tomorrow.io (Weather Information API Docs)
- Claude API Docs
- Cloud Vision documentation | Cloud Vision API | Google Cloud
- Firebase Realtime Database (google.com)
- android - how to convert rgb color to hex color - kotlin - Stack Overflow
- Download 1,348,800 free icons (SVG, PNG) (icons8.com)
- Android Camera Integration with Kotlin: A Step-by-Step Guide (dopebase.com)