CIS 600: Android Programming

CUSE CONNECT

PROJECT REPORT

Team Members:

Group Members	**SUID**
Gouthami Venkat	443273126
Mohit Kambli	508469511
Mukkesh Medai Dalavoi Kumaraswamy	733546104

ABOUT APPLICATION

CuseConnect is a dynamic mobile application tailored for Syracuse University students and administrators. It's designed to enhance engagement with campus life and the local community. The app serves as a comprehensive campus companion, offering three main features:

- **Grievance Reporting:** Grievance Reporting streamlines issue reporting and resolution processes.
- **Event Discovery:** Event Discovery, using Ticketmaster API, simplifies finding and attending local events.
- **Dining Options:** Dining Options, using the Yelp API, assist in exploring local culinary choices. The app caters to students' academic, social, and dining needs while offering administrators efficient tools for managing student concerns.

GRIEVANCE REPORTING

BACKEND:

Firebase's Realtime Database:

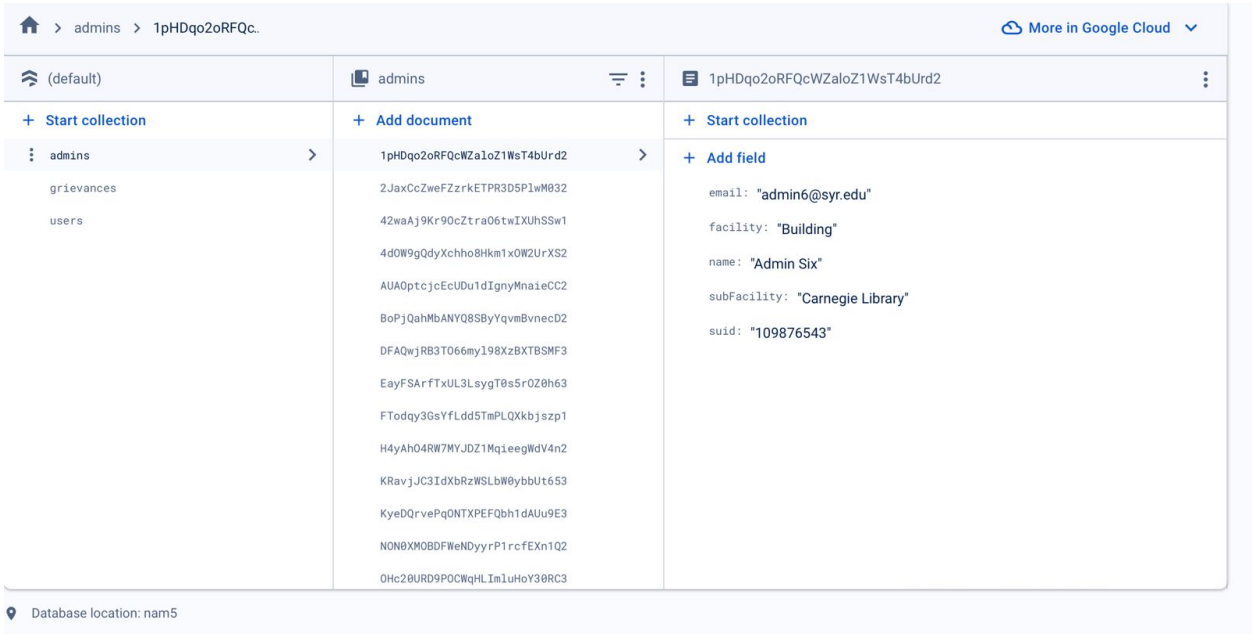
CLOUD FIRESTORE:

Cloud Firestore is a cloud-hosted, NoSQL database designed for global app development. It allows developers to store and sync data between users in real-time, offering seamless offline data access and ensuring end-user data consistency across multiple client apps. Firestore is designed to scale effortlessly, making it suitable for a wide range of applications, from small startups to large enterprises. It supports powerful querying, transaction operations, and it integrates seamlessly with other Firebase and Google Cloud services.

In the Cloud Firestore database for the CuseConnect app, **three collections** structure the data architecture:

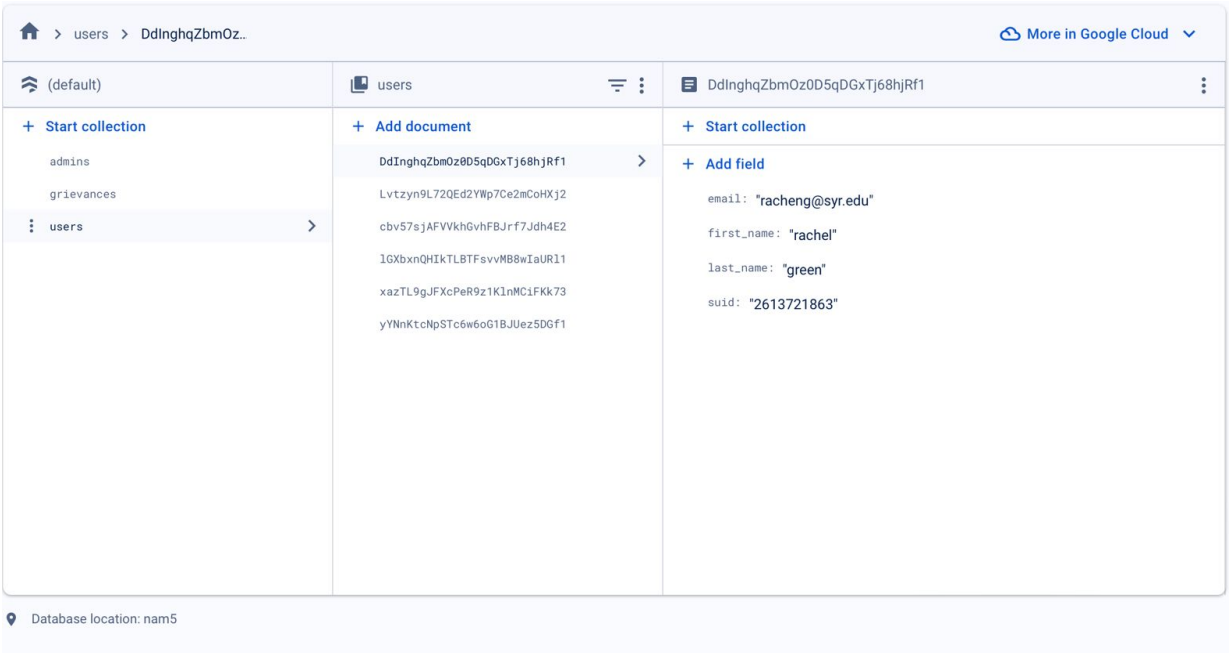
ADMINS: This collection contains documents with the following keys to manage admin profiles:

- email: Admin's email address
- name: Full name of the admin
- facility: Main administrative facility or department
- subfacility: Specific sub-department or area within the facility
- suid: Syracuse University identification number



USERS: This collection houses user information with keys for personal and school identification:

- email: User's email address
- first name: User's given name
- last name: User's family name
- suid: Syracuse University identification number



ADMIN & USER Credentials:

Admins:
Email-ID/Password: admin1@syr.edu/Password1
Email-ID/Password: admin2@syr.edu/Password2
.
.
Email-ID/Password: admin27@syr.edu/Password27

Users:
Email-ID/Password: gvenkat@syr.edu/Gouthami@123
Email-ID/Password: mkambli@syr.edu/Euphie017119#

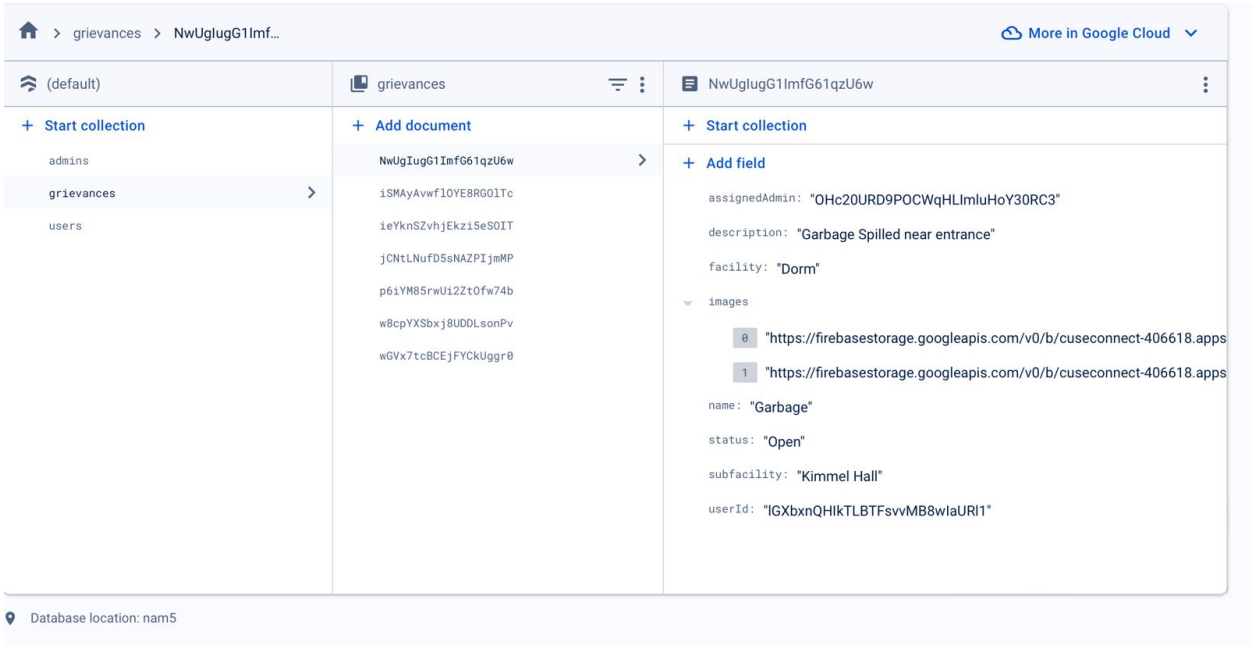
ADMIN-FACILITY-SUB FACILITY MAPPING

```
// Example list of admin details
private val adminsList = listOf(
    Admin("Admin One", "admin1@syr.edu", "Password1", "Building", "Bird Library","123456789", true),
    Admin("Admin Two", "admin2@syr.edu", "Password2", "Building", "Bird Library","987654321", true),
    Admin("Admin Three", "admin3@syr.edu", "Password3", "Building", "Bird Library","234567890", true),
    Admin("Admin Four", "admin4@syr.edu", "Password4", "Building", "Carnegie Library","876543210", true),
    Admin("Admin Five", "admin5@syr.edu", "Password5", "Building", "Carnegie Library","345678901", true),
    Admin("Admin Six", "admin6@syr.edu", "Password6", "Building", "Carnegie Library","109876543", true),
    Admin("Admin Seven", "admin7@syr.edu", "Password7", "Building", "Barnes Center","456789012", true),
    Admin("Admin Eight", "admin8@syr.edu", "Password8", "Building", "Barnes Center","890123456", true),
    Admin("Admin Nine", "admin9@syr.edu", "Password9", "Building", "Barnes Center","567890123", true),
    Admin("Admin Ten", "admin10@syr.edu", "Password10", "Dorm", "Booth Hall", "321098765", true),
    Admin("Admin Eleven", "admin11@syr.edu", "Password11", "Dorm", "Booth Hall", "678901234", true),
    Admin("Admin Twelve", "admin12@syr.edu", "Password12", "Dorm", "Booth Hall", "210987654", true),
    Admin("Admin Thirteen", "admin13@syr.edu", "Password13", "Dorm", "Marion Hall", "543210987", true),
    Admin("Admin Fourteen", "admin14@syr.edu", "Password14", "Dorm", "Marion Hall", "901234567", true),
    Admin("Admin Fifteen", "admin15@syr.edu", "Password15", "Dorm", "Marion Hall", "432109876", true),
    Admin("Admin Sixteen", "admin16@syr.edu", "Password16", "Dorm", "Kimmel Hall","789012345", true),
    Admin("Admin Seventeen", "admin17@syr.edu", "Password17", "Dorm", "Kimmel Hall","876543211", true),
    Admin("Admin Eighteen", "admin18@syr.edu", "Password18", "Dorm", "Kimmel Hall","123456788", true),
    Admin("Admin Nineteen ", "admin19@syr.edu", "Password19", "Dining Hall", "Sadler Dining Hall", "234567899", true),
    Admin("Admin Twenty ", "admin20@syr.edu", "Password20", "Dining Hall", "Sadler Dining Hall", "987654322", true),
    Admin("Admin Twenty One", "admin21@syr.edu", "Password21", "Dining Hall", "Sadler Dining Hall", "345678900", true),
    Admin("Admin Twenty Two", "admin22@syr.edu", "Password22", "Dining Hall", "Ernie Dining Hall", "109876544", true),
    Admin("Admin Twenty Three", "admin23@syr.edu", "Password23", "Dining Hall", "Ernie Dining Hall", "456789011", true),
    Admin("Admin Twenty Four", "admin24@syr.edu", "Password24", "Dining Hall", "Ernie Dining Hall", "890123455", true),
    Admin("Admin Twenty Five", "admin25@syr.edu", "Password25", "Dining Hall", "Shaw Dining Hall", "567890122", true),
    Admin("Admin Twenty Six", "admin26@syr.edu", "Password26", "Dining Hall", "Shaw Dining Hall", "321098766", true),
    Admin("Admin Twenty Seven", "admin27@syr.edu", "Password27", "Dining Hall", "Shaw Dining Hall", "678901233", true),
)
```

GRIEVANCES: This collection tracks issues or complaints with details for resolution:

- name: Title or name of the grievance
- description: Full description of the issue
- facility: Related facility or location of the grievance

- sub facility: More specific location within the facility
- images: Associated images for the grievance
- status: Current status of the grievance (e.g., open, resolved)
- userID: Identifier for the user who reported the grievance
- assignedAdmin: Admin assigned to handle the grievance

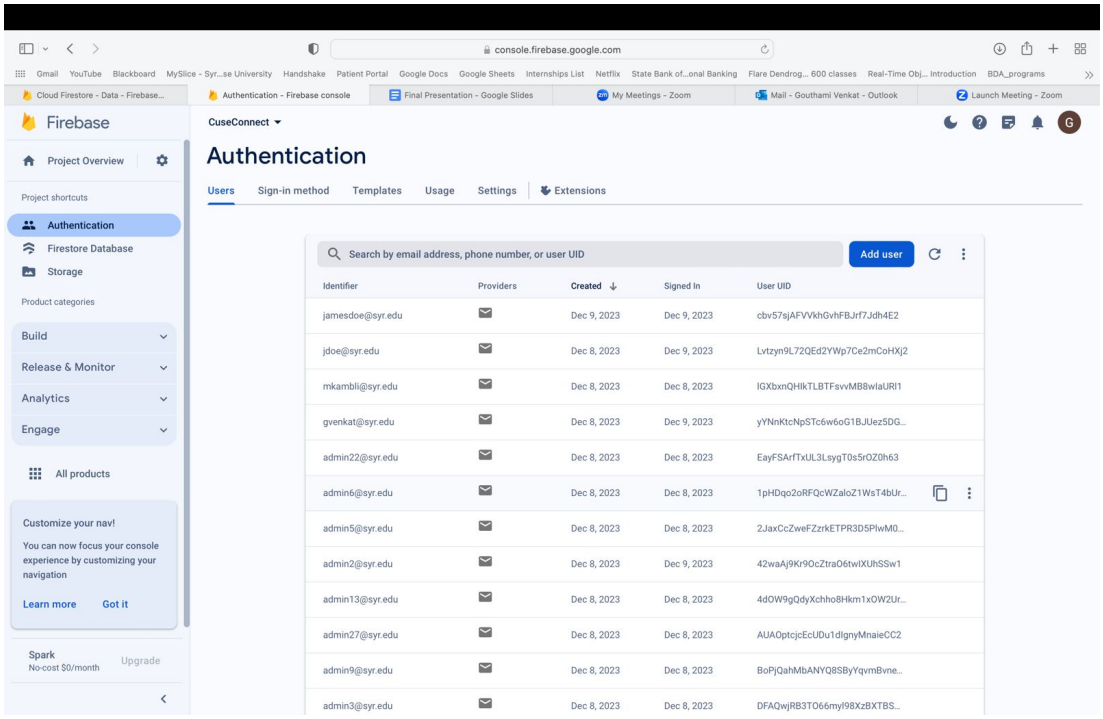


Each collection is optimized for its role in the application, ensuring efficient data management and operations within the app.

Firebase Authentication:

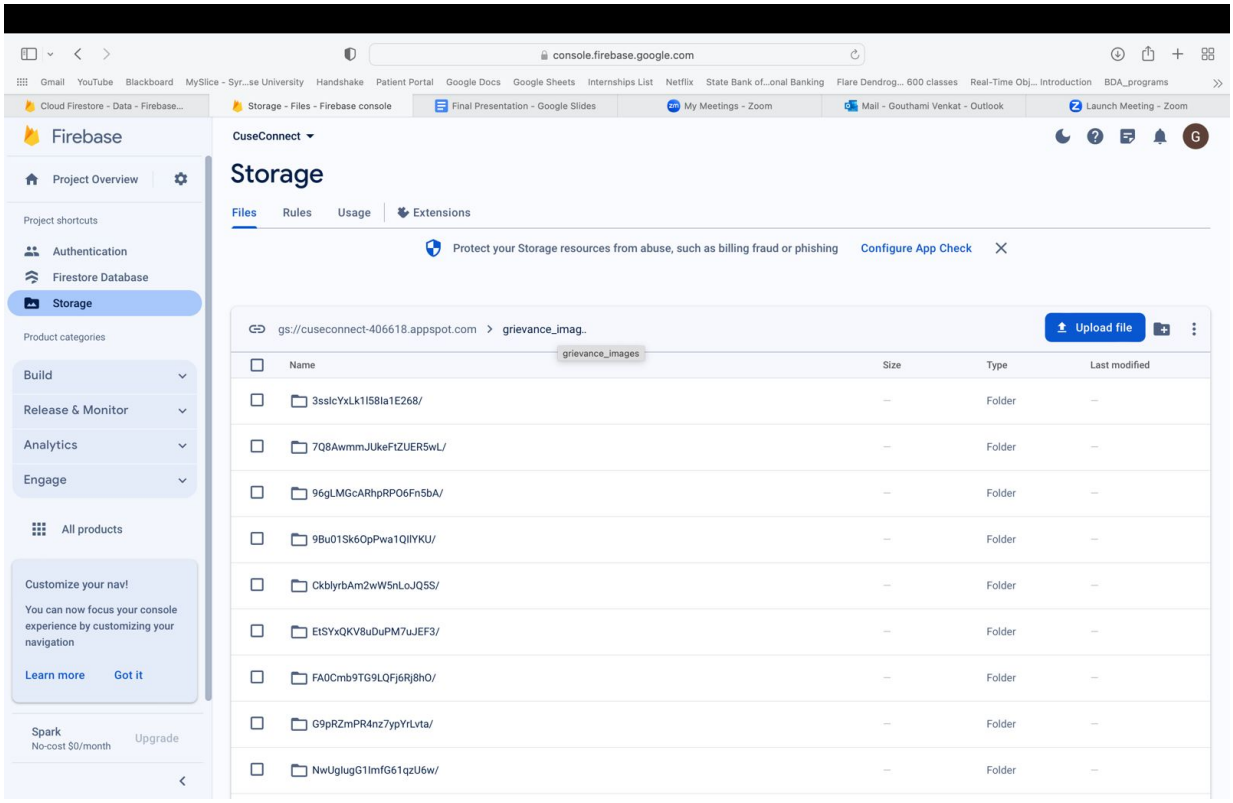
The CuseConnect application employs Firebase Authentication to manage secure sign-in and sign-up processes for Syracuse University users. This robust authentication system ensures that only authorized users gain access to the app's functionalities. For administrators, user accounts are pre-configured and managed directly from the backend, allowing for controlled access to

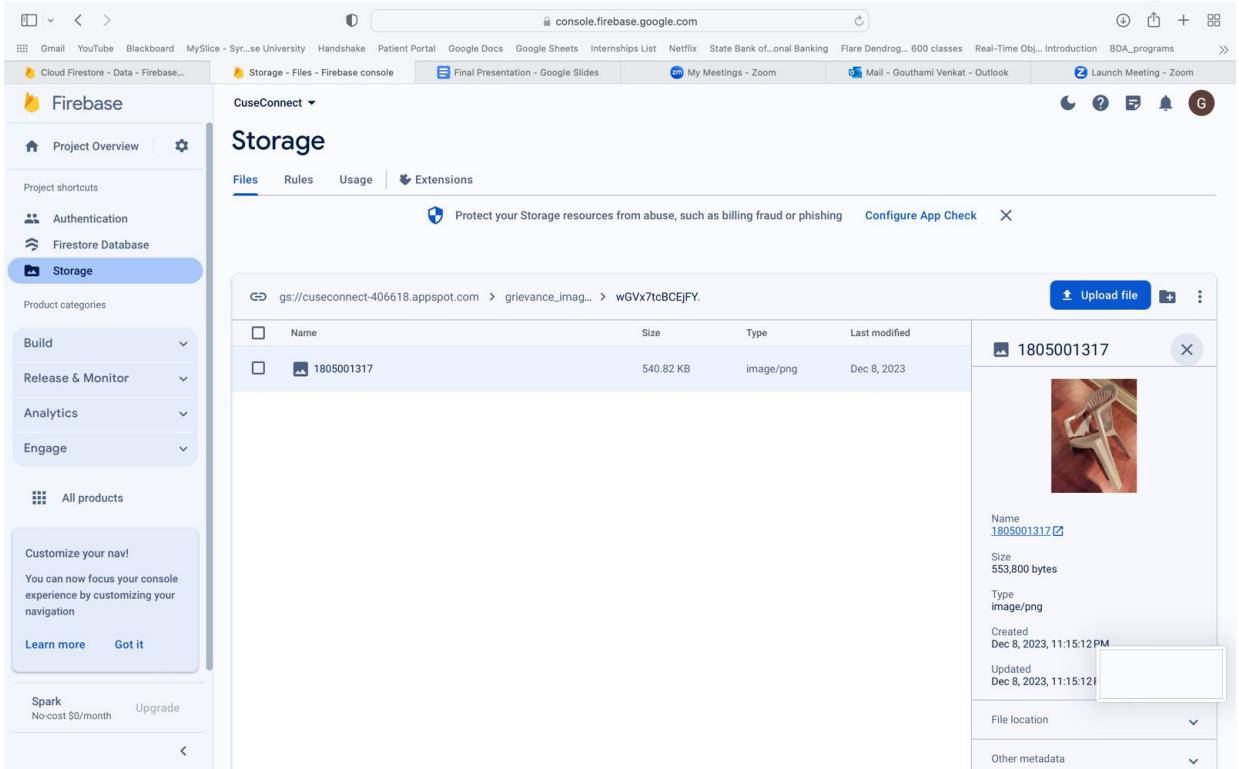
administrative features. This delineation between user and admin roles reinforces the app's security while maintaining a streamlined user experience for both parties.



Data Storage:

In the CuseConnect application, Firebase Storage is utilized to securely store **user-uploaded images associated with grievances**. When a user submits a grievance and includes images, these images are uploaded to Firebase Storage, which provides a powerful and secure way to handle file uploads and downloads. Each image is stored in a structured format, often in a dedicated 'grievances' folder, and can be retrieved through **unique URLs**. These URLs are then stored in the Cloud Firestore 'grievances' collection under the 'images' key, allowing the images to be efficiently fetched and displayed in the admin view for review and action. Firebase Storage ensures that image data is safely managed and seamlessly integrated with the app's data ecosystem.


console.firebase.google.com
Firebase
CuseConnect
Project Overview
Project shortcuts
Authentication
Firestore Database
Storage
Product categories
Build
Release & Monitor
Analytics
Engage
All products
Customize your nav!
You can now focus your console experience by customizing your navigation
Learn more
Got it
Spark
No-cost $0/month
Upgrade
Storage
Files
Rules
Usage
Extensions
Protect your Storage resources from abuse, such as billing fraud or phishing
Configure App Check
gs://cuseconnect-406618.appspot.com > grievance_imag..
Upload file
grievance_images
Name
Size
Type
Last modified
3ssIcYxLk1I58Ia1E268/
7Q8AwmmJUkeFtZUER5wL/
96gLMGcARhpRPO6Fn5bA/
9Bu01Sk6OpPwa1QIlYKU/
CkblyrbAm2wW5nLoJQ5S/
EtSYxQKV8uDuPM7uJEF3/
FA0Cmb9TG9LQFj6Rj8hO/
G9pRZmPR4nz7ypYrLvta/
NwUglugG1ImfG61qzU6w/
Folder


console.firebase.google.com
Firebase
CuseConnect
Project Overview
Project shortcuts
Authentication
Firestore Database
Storage
Product categories
Build
Release & Monitor
Analytics
Engage
All products
Customize your nav!
You can now focus your console experience by customizing your navigation
Learn more
Got it
Spark
No-cost $0/month
Upgrade
Storage
Files
Rules
Usage
Extensions
Protect your Storage resources from abuse, such as billing fraud or phishing
Configure App Check
gs://cuseconnect-406618.appspot.com > grievance_imag... > wGVx7tcBCEjFY.
Upload file
Name
Size
Type
Last modified
1805001317
540.82 KB
image/png
Dec 8, 2023
1805001317
Name
1805001317
Size
553,800 bytes
Type
image/png
Created
Dec 8, 2023, 11:15:12 PM
Updated
Dec 8, 2023, 11:15:12
File location
Other metadata

GRIEVANCE ADMIN PERSPECTIVE:

About Admin Role:

Within the CuseConnect app, admins play a crucial role in grievance management by monitoring issues within specific facilities and subfacilities. These are categorized into three main facilities—

1. Buildings
2. Dorms
3. Dining Halls.

Each main facility encompasses various subfacilities, which include individual buildings, dorms, and dining establishments. Admins are tasked with addressing grievances related to their designated subfacility, updating statuses, and providing feedback, thus ensuring that resolutions are effectively communicated and that user satisfaction is upheld across the university campus. Each grievance, presented in a color-coded card indicating status, requires review. Their role is pivotal in maintaining the grievance resolution workflow and enhancing user satisfaction within the university's community.

Landing Page (Grievance List Page):

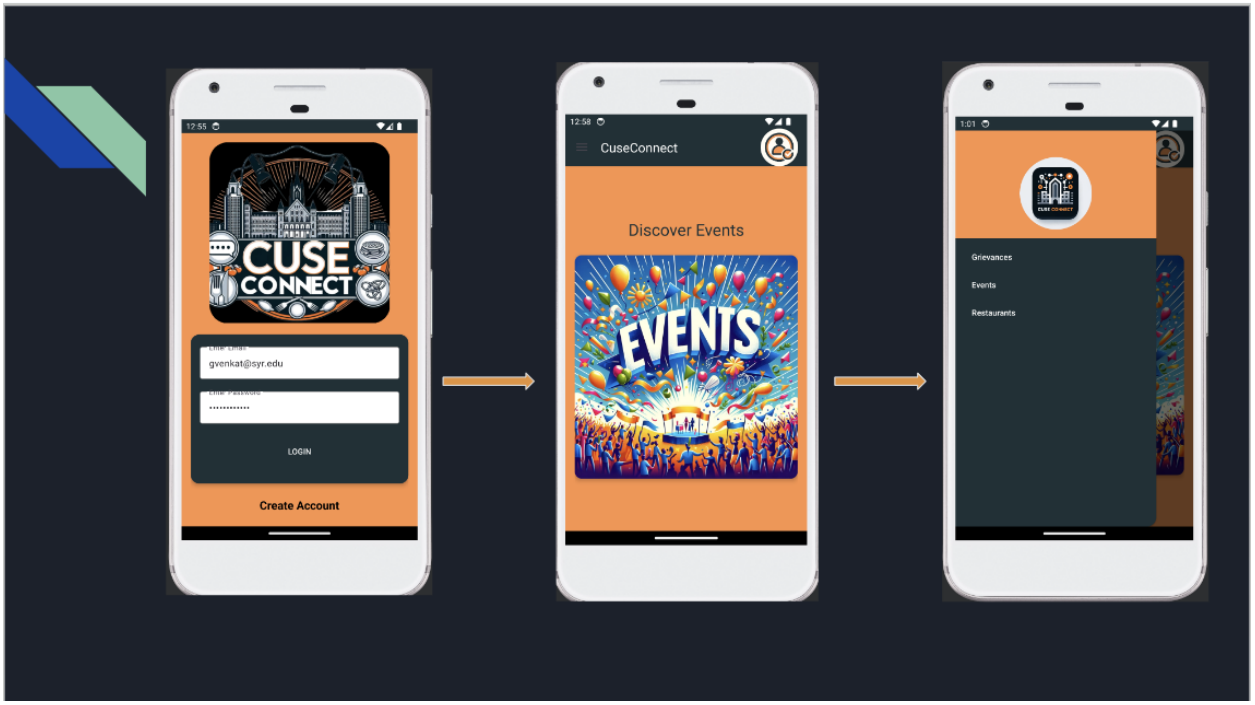
The admin module is engineered with a user-centric interface. Upon successful authentication, an admin is directed to a grievance list page. This page features a toolbar with an icon, which, when tapped, presents a pop-up containing the admin's profile information and a sign-out option.

In the app, grievances are systematically displayed to admins based on the sub facility they are assigned to. Each entry is presented in a CardView within a RecyclerView, with color indicators—**green signifies a resolved issue**, while **orange denotes one that needs attention**. This visual coding, combined with the displayed details of the grievance such as the **icon, title, the user who logged it, and its current status**, provides a clear and immediate overview, enabling admins to **efficiently manage and prioritize their tasks**.

Detail Page:
Upon accessing the detail page of a grievance, an admin is presented with comprehensive information including the Grievance Name, Description, Facility, Subfacility, and the Username of the reporting user, along with any images uploaded as part of the grievance report. A status indicator, implemented as a dropdown spinner, allows the admin to update the **grievance from 'Open' to 'Resolved'**. The admin can also **input feedback in a designated text box**. After updating the status and providing feedback, **clicking the 'Update Status' button will record the changes**, which are then reflected on the grievance list page: the CardView for the grievance will change color to GREEN to indicate its resolved status, enhancing visual acknowledgment and tracking.

Once the admin updates a grievance's status and enters feedback, **the user will see this reflected in their view**. The CardView will turn green to signify resolution, and the feedback provided by the admin will be visible to the user, indicating that the grievance has been addressed.

USER PERSPECTIVE:



User Sign in Page:
Upon initiating the CuseConnect app, users are greeted with a login screen, enhanced with the app's logo, which prompts for email and password entry. These credentials are authenticated via Firebase when the **'Login' button** is pressed. Successful verification grants access to the app's main interface, while a failed attempt triggers a toast bar for incorrect credentials. Additionally, users have the option to navigate to account creation by selecting the **'Create Account' button**.

Account Creation:
The signup process for the CuseConnect app is facilitated by Firebase Authentication, requiring users to input their First Name, Last Name, Email, Password, and Syracuse University Identification (SUID). Each field is compulsory, with stringent validation checks in place: the email address must follow the format of 'syr.edu', and the password is required to be a minimum of eight characters in length for security purposes. This ensures that all user accounts are verified and adhere to university standards.

User Navigation View and Toolbar:
From a user's perspective in the CuseConnect app, they interact with a navigation drawer that features the app's logo at the top and houses **three main menu items**:

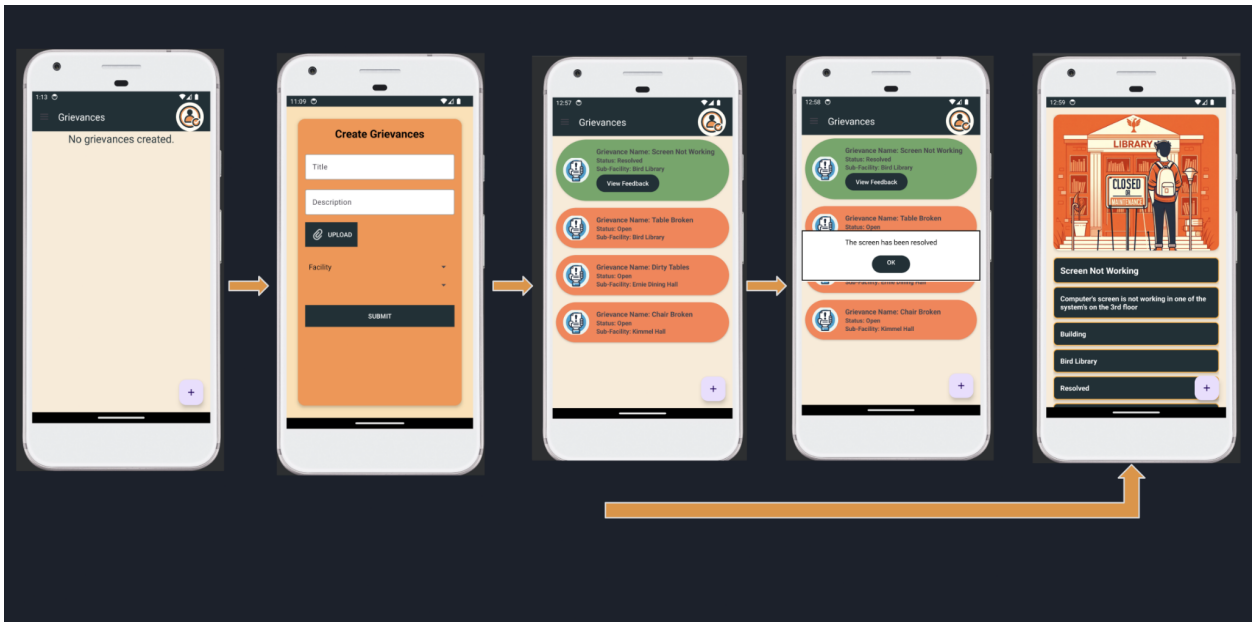
1. Grievances
2. Events
3. Restaurants.

This navigation drawer is consistently accessible across all list pages for ease of use.

Additionally, each page is equipped with a **toolbar that includes a profile icon** on the right. Users can tap this icon to access their **profile information or to sign out** of the app, ensuring a user-friendly experience with essential functionalities readily available.

User Landing Page:
Upon logging into the CuseConnect app, the user is greeted by a dynamic landing page. This page features a dynamic text view and an animated cardview that cycles through images, providing a visually engaging experience. From here, users can employ the navigation drawer to easily access different features of the app, such as grievances, events, and restaurant listings, making for a seamless transition to the various services offered by the app.

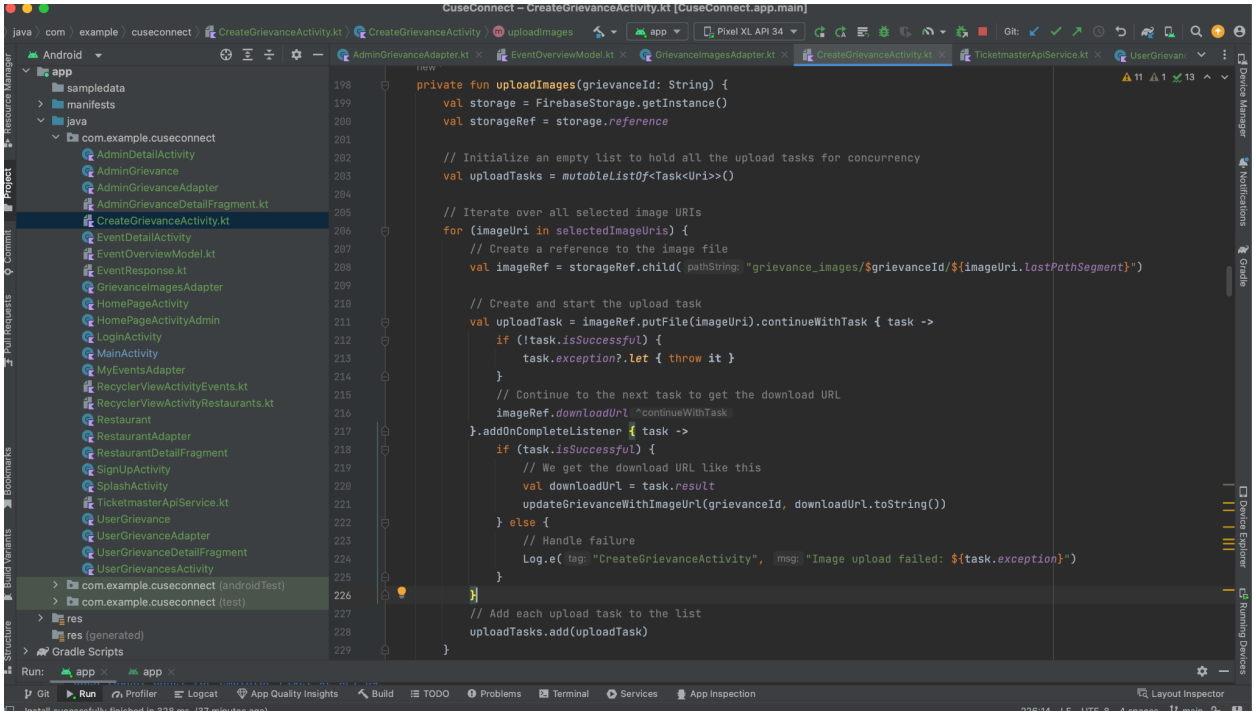
GRIEVANCE USER PERSPECTIVE:



The CuseConnect app's toolbar features a user icon, which, when clicked, reveals the user's account details and includes an option to sign out. This function provides easy access to personal settings and secure exit from the application.

ABOUT CAMERA FEATURE:
In the CreateGrievanceActivity class of the provided Android application, the camera upload feature is implemented to allow users to take photos or select them from the gallery for attaching to a grievance report. When the user clicks the upload button, the app first checks if the necessary permissions for camera and storage access are granted. If not, it requests these permissions. Once permissions are granted, the user is presented with options to either take a new photo using the camera or choose existing photos from the gallery. This is facilitated by the

showPhotoSourceOptions() method, which creates an AlertDialog with these options. If the user opts to take a photo, the app launches the camera using an intent with MediaStore.ACTION_IMAGE_CAPTURE. For choosing from the gallery, an intent with Intent.ACTION_PICK and MediaStore.Images.Media.EXTERNAL_CONTENT_URI is used. The selected or captured images are then stored in the selectedImageUris ArrayList. These images can be uploaded to a server or processed further as needed in the application. The onActivityResult() method handles the result returned from the camera or gallery and processes the images accordingly.



Grievances List:

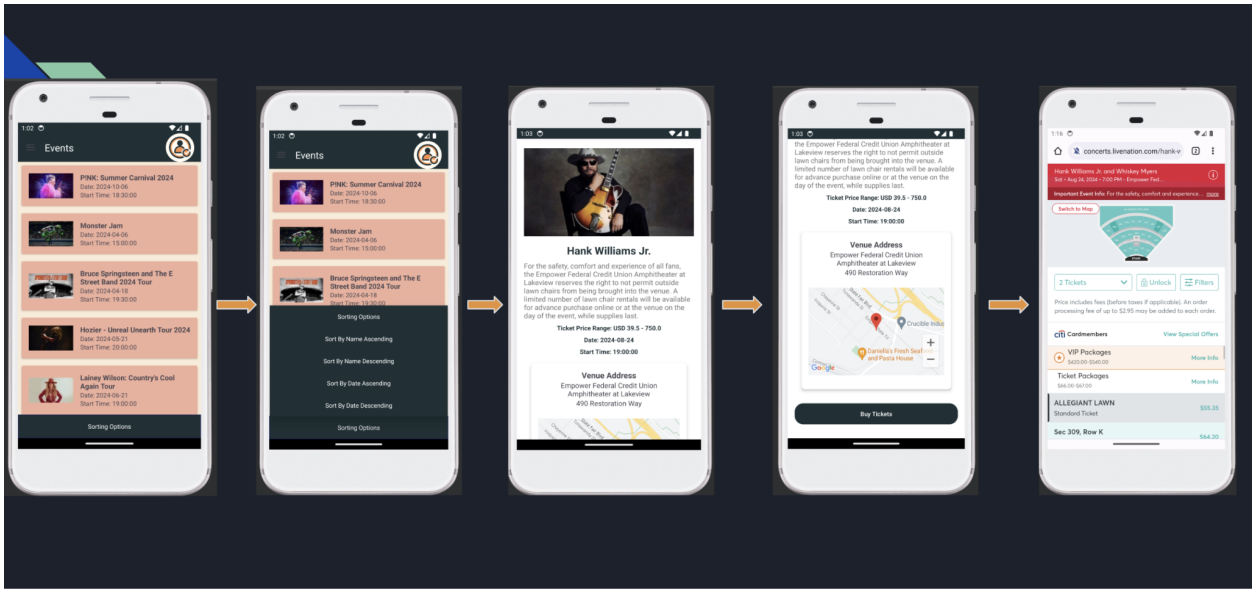
In the CuseConnect app's Grievance section, users encounter a list view where grievances are displayed in a card format. Each card represents a unique grievance submitted by the user, detailing the issue at a glance. In the app's grievance tracking system, card colors visually communicate the status of a user's report: orange signifies an open issue, while green denotes a resolved one. Additionally, for grievances that have been addressed, there is an option to view detailed feedback, providing users with closure and insight into the resolution process. This intuitive and interactive system enables users to easily track the progress of their reported grievances.

This section features a button for reporting grievances, guiding users to a comprehensive form where they can articulate concerns about campus facilities. This form, designed for clarity and ease of use, requests essential information such as the title, detailed description of the issue,

associated facility and also provides the functionality to upload relevant imagery which helps the campus administrators to promptly address student concerns.

Detail View Write-up:
Upon selecting a specific grievance card, users are presented with a detailed view that elaborates on the grievance they've submitted. This view includes all the particulars of the issue, such as the title, description, and any images uploaded as part of the report. It's a deeper dive into the user's submission, providing a full overview of the grievance to the user.

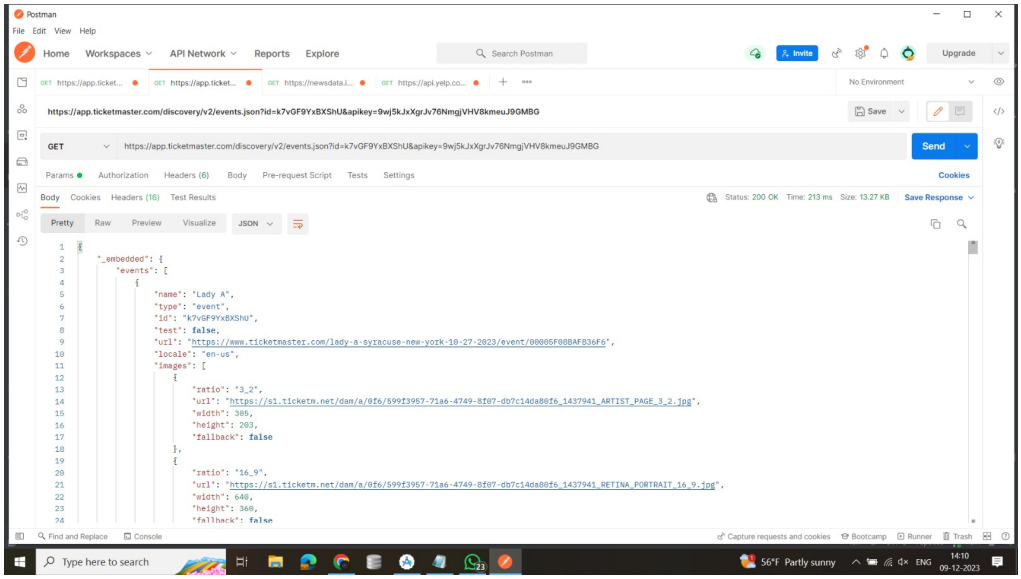
EVENT DISCOVERY:



Event Discovery in the CuseConnect app, utilizing the Ticketmaster API, involves fetching and displaying information about various events. This API provides access to a vast database of event-related data, including event names, dates, venues, and more. The app retrieves this information through a network request, processes it, and presents it in an easily navigable format for the users. This feature allows users to discover, search, and sort events based on their preferences, enhancing their engagement with local community events.

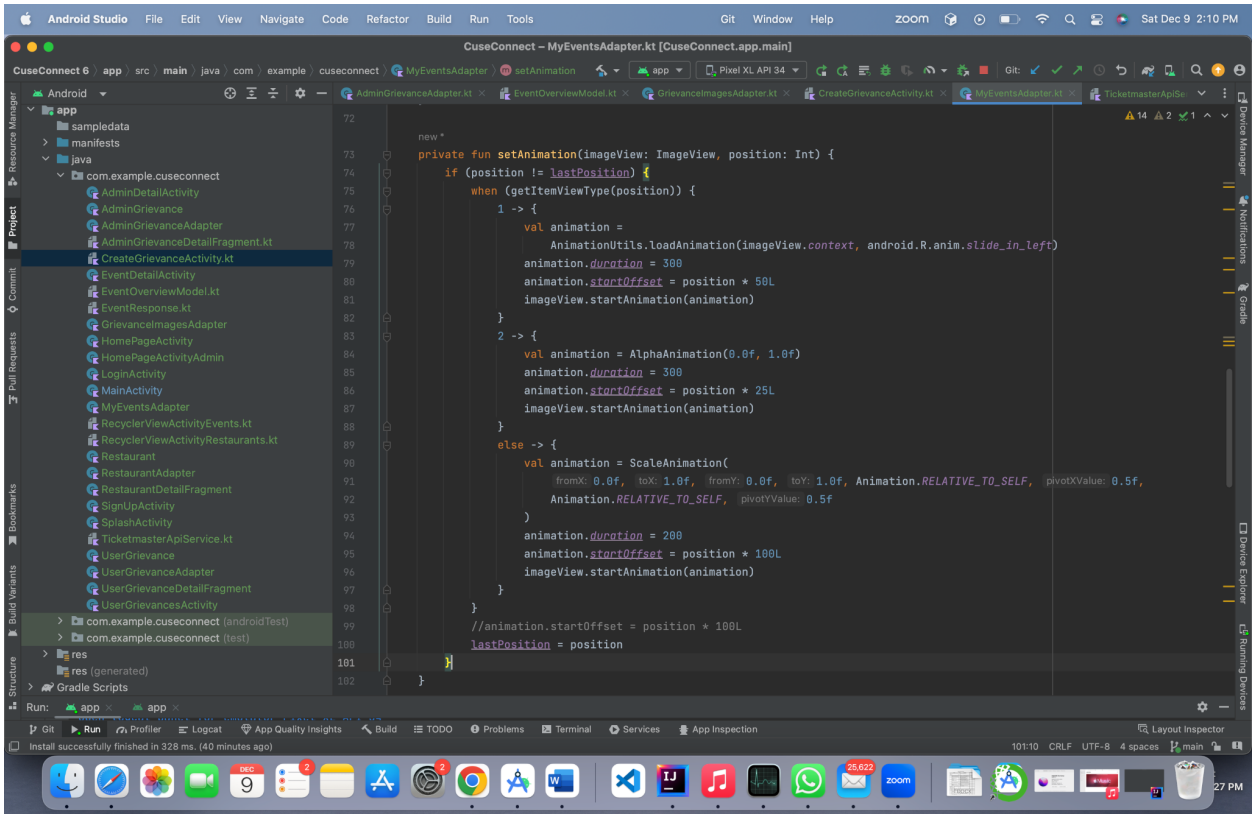
TicketMaster API
The Ticketmaster Discovery API is a tool for developers that enables integration with Ticketmaster's vast database of live event information. It allows applications to search and retrieve detailed listings of events, including dates, venues, artist details, and high-resolution images. The API also provides access to important information like ticket pricing and availability, helping to create comprehensive event discovery experiences within third-party

services. This API makes it possible to connect fans with their favorite events seamlessly and in real time



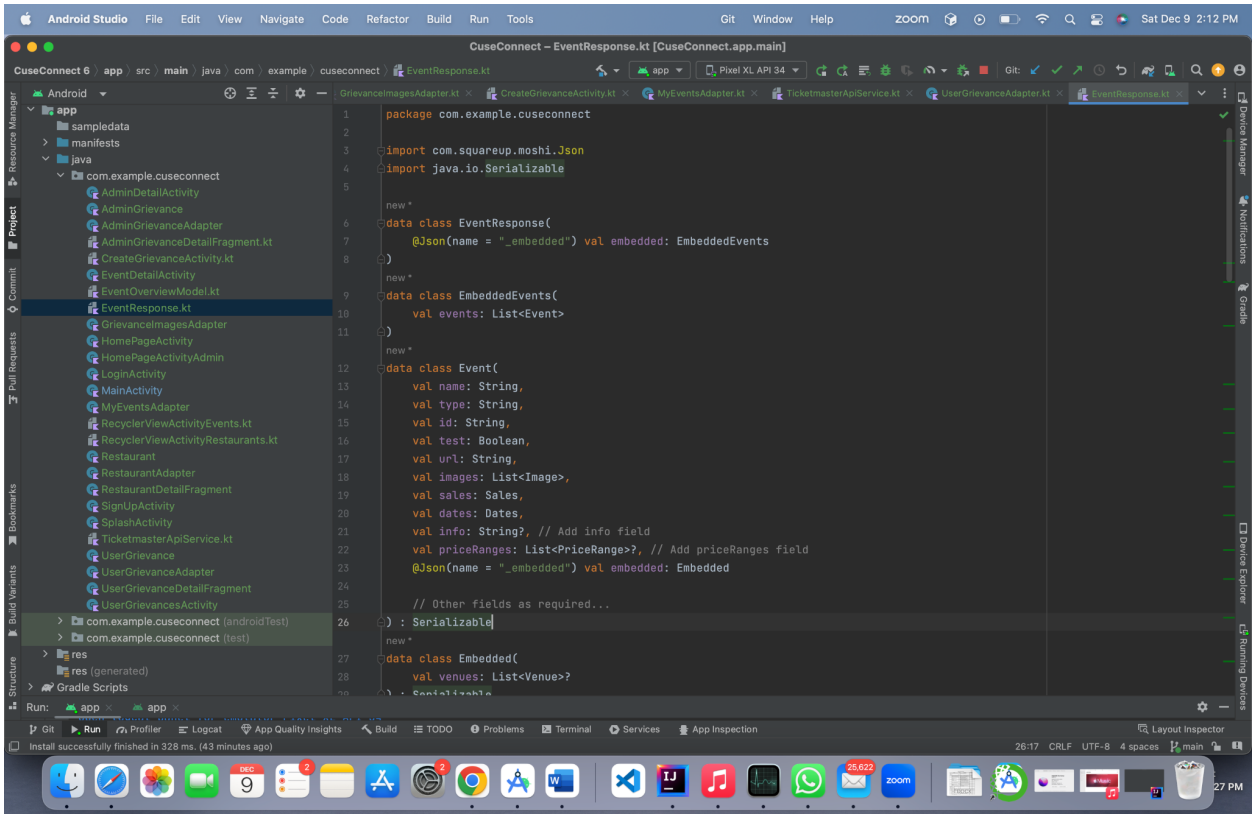
ABOUT IMAGE ANIMATION:

The setAnimation function applies different animations to an ImageView based on its position in a list. When a new item position is detected (different from lastPosition), it selects an animation type according to the getItemViewType(position). For type 1, it uses a slide-in animation from the left, with a duration of 300 milliseconds and a staggered start offset based on the item's position. Type 2 triggers a fade-in animation from transparent to opaque over 300 milliseconds, with a faster staggered start. Other types use a scale animation, expanding the item from zero to its full size in 200 milliseconds, also with a staggered start. The animation start offset is calculated uniquely for each item, creating a cascading appearance effect. After animating, lastPosition is updated to avoid re-animating the same item. This method enhances user engagement by adding dynamic visual effects as items appear on the screen.



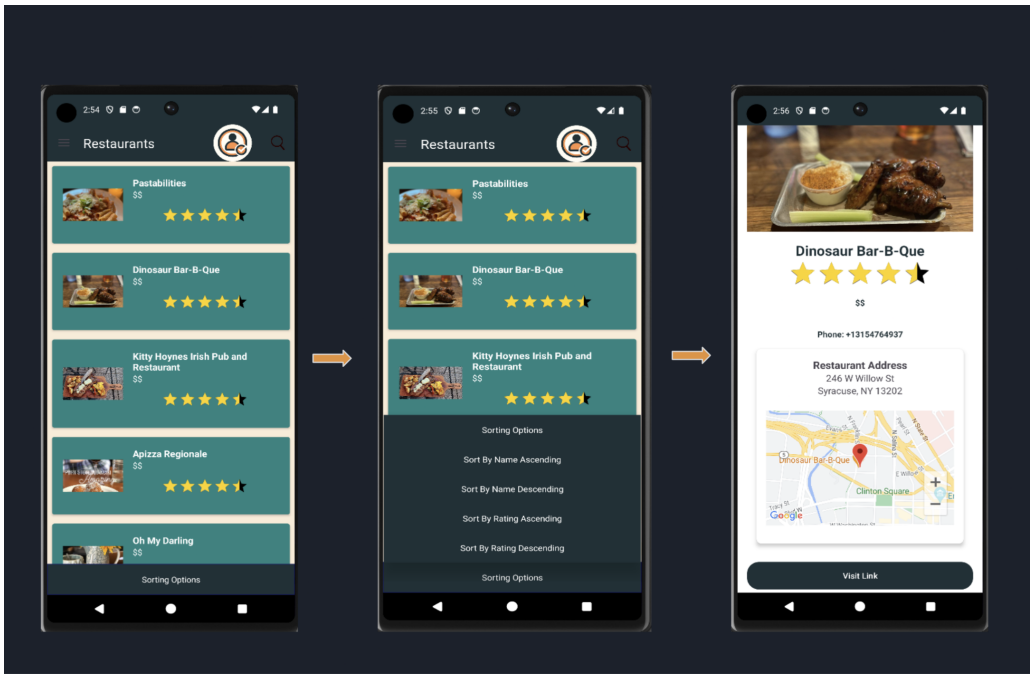
ABOUT RETROFIT and MOSHI libraries:
Retrofit is a type-safe HTTP client for Android and Java, designed for interacting with APIs and sending network requests. It turns an API interface into a Java interface, streamlining the process of connecting to a web service. Moshi, complementing Retrofit, is a JSON parser that helps convert JSON data into Java and Kotlin objects. It's particularly efficient with Kotlin, supporting its unique features.

In the code, Retrofit is set up with a base URL for the Ticketmaster API and configured to use Moshi as its converter. The TicketmasterApiService interface defines an API method getEvents with @GET, indicating an HTTP GET request. This method uses Kotlin's coroutines (suspend function) for asynchronous calls and accepts parameters marked with @Query for dynamic query building. Finally, TicketmasterApi provides a lazy instantiation of the Retrofit service, ready to fetch event data from Ticketmaster based on city input, demonstrating an efficient integration of network operations and JSON parsing in a Kotlin context.



RESTAURANTS

Dining Options:



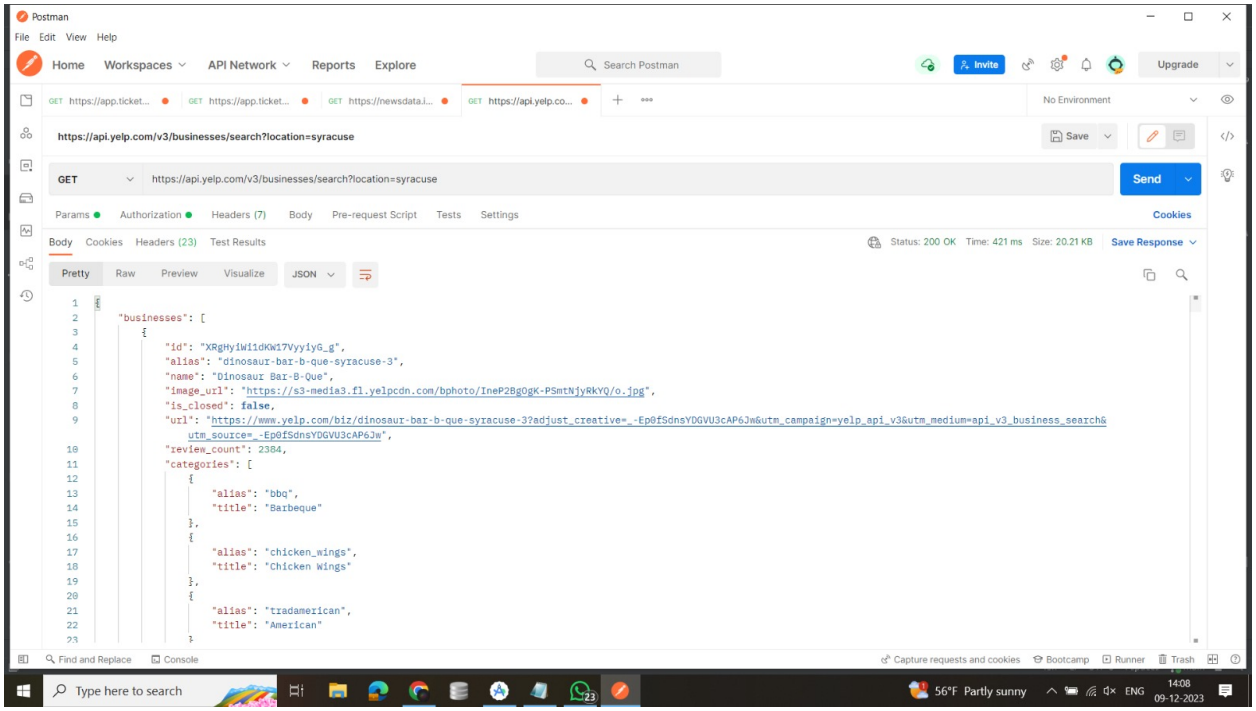
The restaurant feature of the CuseConnect app dynamically pulls real-time data from the Yelp API to display a variety of dining options available to users. It delivers essential information about the restaurants. With this integration, users can effortlessly explore, evaluate, and select dining venues that match their preferences, thus streamlining their culinary adventures within the Syracuse area.

Restaurant List Page:
The application's cardview interface serves as an interactive gateway, presenting a snapshot of Syracuse's diverse dining scene. It dynamically fetches and displays a list of restaurants from the **Yelp API**, offering a glance at key details such as the establishment's name, cost estimation indicated by price point symbols, and the aggregated customer ratings represented by stars. The app's list page offers a **search function** in the **action bar** that simplifies the process of finding restaurants for users. Additionally, a spinner in the bottom action bar offers four sophisticated sorting options: **Sort by Name Ascending, Sort by Name Descending, Sort by Rating Ascending, and Sort by Rating Descending**. This high-level overview is designed for at-a-glance decision making, enriched with sorting capabilities for a personalized browsing experience.

Restaurant Detail Page:
Upon selection of a restaurant from the cardview, the app transitions to a detailed view, enriching the user's interface with comprehensive information. This includes the restaurant's full name, its Yelp star rating, price category, contact number, and a **location map equipped with GPS functionality using Google Maps API**. This feature ensures a seamless user journey from discovery to dining, providing all necessary details to assist users in making informed dining decisions and facilitating the navigation to their chosen restaurant. The app features a **button that routes users straight to the restaurant's page on Yelp**, where they can delve into additional details and user reviews.

YELP API

The Yelp API, also known as Yelp Fusion, offers developers a gateway to one of the largest repositories of local business information and user reviews. It empowers applications to conduct searches for a wide array of business types, including restaurants, bars, and cafes, providing detailed results such as business hours, location, contact information, and ratings. Additionally, developers can leverage the API to access rich content like user-generated reviews and photos, enhancing the user experience with authentic visuals and feedback. The Yelp API's robust search capabilities enable applications to deliver personalized local search experiences to users, with filters for categories, price levels, and specific services like delivery or pickup. This integration facilitates the connection between users and their ideal local experiences, all through a seamless digital interface powered by Yelp's extensive data platform



About MapView Feature

```
private lateinit var mapView: MapView
private lateinit var googleMap: GoogleMap

new *
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    mapView = view.findViewById(R.id.mapView)
    mapView.onCreate(savedInstanceState)
    mapView.getMapAsync( callback: this)
}

new *
override fun onMapReady(map: GoogleMap) {
    googleMap = map

    // Get restaurant coordinates and set a marker on the map
    //val restaurantCoordinates = LatLng(43.05269, -76.1546)
    val restaurantCoordinates = LatLng(restaurantLatitude.toDouble(), restaurantLongitude.toDouble())
    googleMap.addMarker(MarkerOptions().position(restaurantCoordinates).title("Restaurant Location"))
    googleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(restaurantCoordinates,  zoom: 15f))
    googleMap.uiSettings.isZoomControlsEnabled = true
    googleMap.uiSettings.isZoomGesturesEnabled = true
    googleMap.uiSettings.isScrollGesturesEnabled = true
    googleMap.uiSettings.isScrollGesturesEnabledDuringRotateOrZoom = true
    googleMap.uiSettings.isMapToolbarEnabled = true
}
```

```
new *
override fun onResume() {
    super.onResume()
    mapView.onResume()
}

new *
override fun onPause() {
    super.onPause()
    mapView.onPause()
}

new *
override fun onDestroy() {
    super.onDestroy()
    mapView.onDestroy()
}

new *
override fun onLowMemory() {
    super.onLowMemory()
    mapView.onLowMemory()
}
```

Handling MapView Lifecycle Events:
The first screenshot shows functions that override the onResume(), onPause(), onDestroy(), and onLowMemory() lifecycle callbacks of an Android Activity or Fragment. These methods ensure that the MapView instance's lifecycle is tied to the lifecycle of the hosting Activity or Fragment. This is necessary for the MapView to behave correctly, such as pausing the rendering of the map when the Activity or Fragment is not active.

Setting Up the MapView:
The second screenshot shows the onViewCreated() method, which is typically used in a Fragment. In this method, the MapView is initialized with the savedInstanceState and an asynchronous request is made to get the GoogleMap instance. This is the setup phase where the MapView is tied to a layout (R.id.mapView) and prepared to display the map.

Configuring GoogleMap Settings:
Once the GoogleMap is ready (in the onMapReady() callback), several settings are applied:
A marker is added to the map at the specified restaurant coordinates (latitude and longitude).
The camera is moved and zoomed to the location of the marker.
Zoom controls are enabled, allowing users to zoom in and out on the map.
Gesture controls for zooming and scrolling are enabled, enhancing the user's ability to interact with the map.

GPS Support:
While the code snippets do not explicitly show GPS functionality, the use of location coordinates suggests that the application may retrieve the user's current location via GPS to display on the map. The GoogleMap API allows for real-time location updates and can show the user's current position on the map if GPS support is enabled and the necessary permissions are granted.

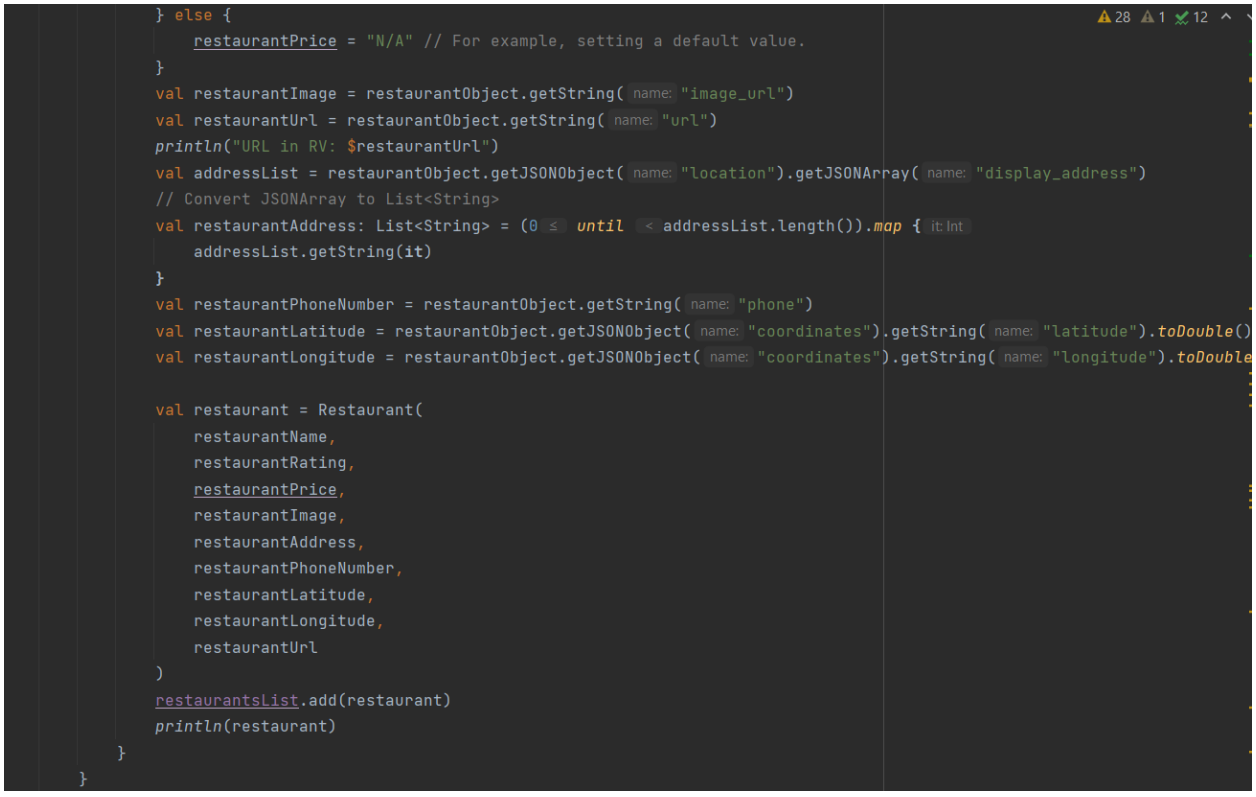
About okhttp3:

```
val apiKey = "GZkg6b2vZHnIqulSuuRxPD0r74Ny53I6jbkxlXZFcR51lL0D3yyn1ZYiK2qwjHiuLcJlMD10nP1VPuOQGd-lHRQ7MmO2MBY
val location = "SYR"
val baseUrl = "https://api.yelp.com/v3/businesses/search?location=$location"

val client = OkHttpClient()

val request = Request.Builder()
    .url(baseUrl)
    .header( name: "Authorization",  value: "Bearer $apiKey")
    .build()

try {
    val response = client.newCall(request).execute()
    if (!response.isSuccessful) {
        println("Error: ${response.code}: ${response.message}")
    } else {
        val responseBody = response.body?.string()
        val jsonObject = JSONObject(responseBody)
        val restaurantsArray = jsonObject.getJSONArray( name: "businesses")
        for (i in 0 ≤ until  < restaurantsArray.length()) {
            val restaurantObject = restaurantsArray.getJSONObject(i)
            val restaurantName = restaurantObject.getString( name: "name")
            val restaurantRating = restaurantObject.getString( name: "rating").toFloat()
            var restaurantPrice = ""
            if (restaurantObject.has( name: "price")) {
                restaurantPrice = restaurantObject.getString( name: "price")
            } else {
                restaurantPrice = "N/A" // For example, setting a default value.
            }
            val restaurantImage = restaurantObject.getString( name: "image_url")
```



The code snippet demonstrates the use of the Yelp Fusion API to retrieve information about restaurants in a specified location (in this case, "SYR" for Syracuse). The API request is constructed with the provided Yelp API key and location, and an OkHttp client is used to execute the request. The response is then processed, checking for success and parsing the JSON data. For each restaurant in the response, relevant details such as name, rating, price, image URL, website URL, address, phone number, latitude, and longitude are extracted. The information is used to create Restaurant objects, which are added to a restaurantsList. The code also handles cases where certain information may be missing, setting default values accordingly. Finally, a callback is invoked upon successful execution, and any exceptions are caught and printed. This code essentially fetches and parses restaurant data from the Yelp API, creating a list of Restaurant objects for further use in the application, such as displaying them in a RecyclerView.

About user state:

```
private val SPLASH_TIME_OUT: Long = 2000
private lateinit var auth: FirebaseAuth

new *
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_splash)

    auth = FirebaseAuth.getInstance()

    Handler().postDelayed({
        // Check if the user is already authenticated
        val userUid = auth.currentUser?.uid
        if (userUid != null) {
            // The user is already authenticated, start the main activity accordingly
            checkLoggedInUser(userUid)
        } else {
            // The user is not authenticated, start the login activity
            startActivity(Intent( packageContext: this, LoginActivity::class.java))
        }

        // Close this activity
        //finish()
    }, SPLASH_TIME_OUT)
}
```

```
private fun checkLoggedInUser(userId: String) {
    val usersCollection = FirebaseFirestore.getInstance().collection( collectionPath: "users")
    val adminsCollection = FirebaseFirestore.getInstance().collection( collectionPath: "admins")

    // Check if the user is in the "users" collection
    usersCollection.document(userId).get()
        .addOnSuccessListener { userDocument ->
            if (userDocument.exists()) {
                // User is not an admin, start regular user activity
                startRegularUserActivity()
            } else {
                // User is not in the "users" collection, check the "admins" collection
                adminsCollection.document(userId).get()
                    .addOnSuccessListener { adminDocument ->
                        if (adminDocument.exists()) {
                            // User is an admin, start admin activity
                            startAdminActivity()
                        } else {
                            // User is not in the "admins" collection either
                            // Handle accordingly (e.g., show an error message)
                            Toast.makeText( context: this, text: "User not found in 'users' or 'admins' collection", Toast.LENGTH_SHORT).show
                        }
                    }
                    .addOnFailureListener { e ->
                        // Handle failure when checking "admins" collection
                        Toast.makeText( context: this, text: "Error checking 'admins' collection: ${e.message}", Toast.LENGTH_SHORT).show()
                    }
            }
        }
        .addOnFailureListener { e ->
```

```
                        // Handle failure when checking "admins" collection
                        Toast.makeText( context: this,  text: "Error checking 'admins' collection: ${e.message}", Toast.LENGTH_SHORT).show()
                    }
                }
            }
            .addOnFailureListener { e ->
                // Handle failure when checking "users" collection
                Toast.makeText( context: this,  text: "Error checking 'users' collection: ${e.message}", Toast.LENGTH_SHORT).show()
            }
    }

    new *
    private fun startRegularUserActivity() {
        // Logic to start regular user activity
        startActivity(Intent( packageContext: this, HomePageActivity::class.java))
        finish() // Optional: finish the current activity if needed
    }

    new *
    private fun startAdminActivity() {
        // Logic to start admin activity
        startActivity(Intent( packageContext: this, HomePageActivityAdmin::class.java))
        finish() // Optional: finish the current activity if needed
    }

}
```

This code snippet defines a SplashActivity, serving as a splash screen that briefly appears upon app launch. The onCreate method initializes the Firebase authentication object and uses a Handler with a delayed post to check if a user is already authenticated. If so, the checkLoggedInUser function is called to determine whether the user is a regular user or an admin. This function queries the Firebase Firestore database to check if the user's ID exists in the "users" or "admins" collection and starts the corresponding activity based on the result. The startRegularUserActivity and startAdminActivity functions initiate the regular user and admin activities, respectively. Error handling is implemented using Toast messages to notify the user of any issues during the authentication or database queries. The splash screen remains visible for a defined duration (SPLASH_TIME_OUT) before redirecting the user to the appropriate activity, enhancing the user experience by providing a seamless transition from the splash screen to the main application.

We have covered the following basic and advanced features:

BASIC FEATURES:

- **Navigation/Drawer/ViewPager**
- **Toolbar**
- **Multiple Fragments and Activities**
- **Layout & Orientation Changes**
- **Cloud database**
- **Dialog/Custom View/Animation**

ADVANCED FEATURES:

- **Camera/Gallery support**
- **Maps**
- **GPS**
- **Real API Data:** TicketMaster API and Yelp API
- **Retrofit/Moshi Libraries**
- **Firebase Security**