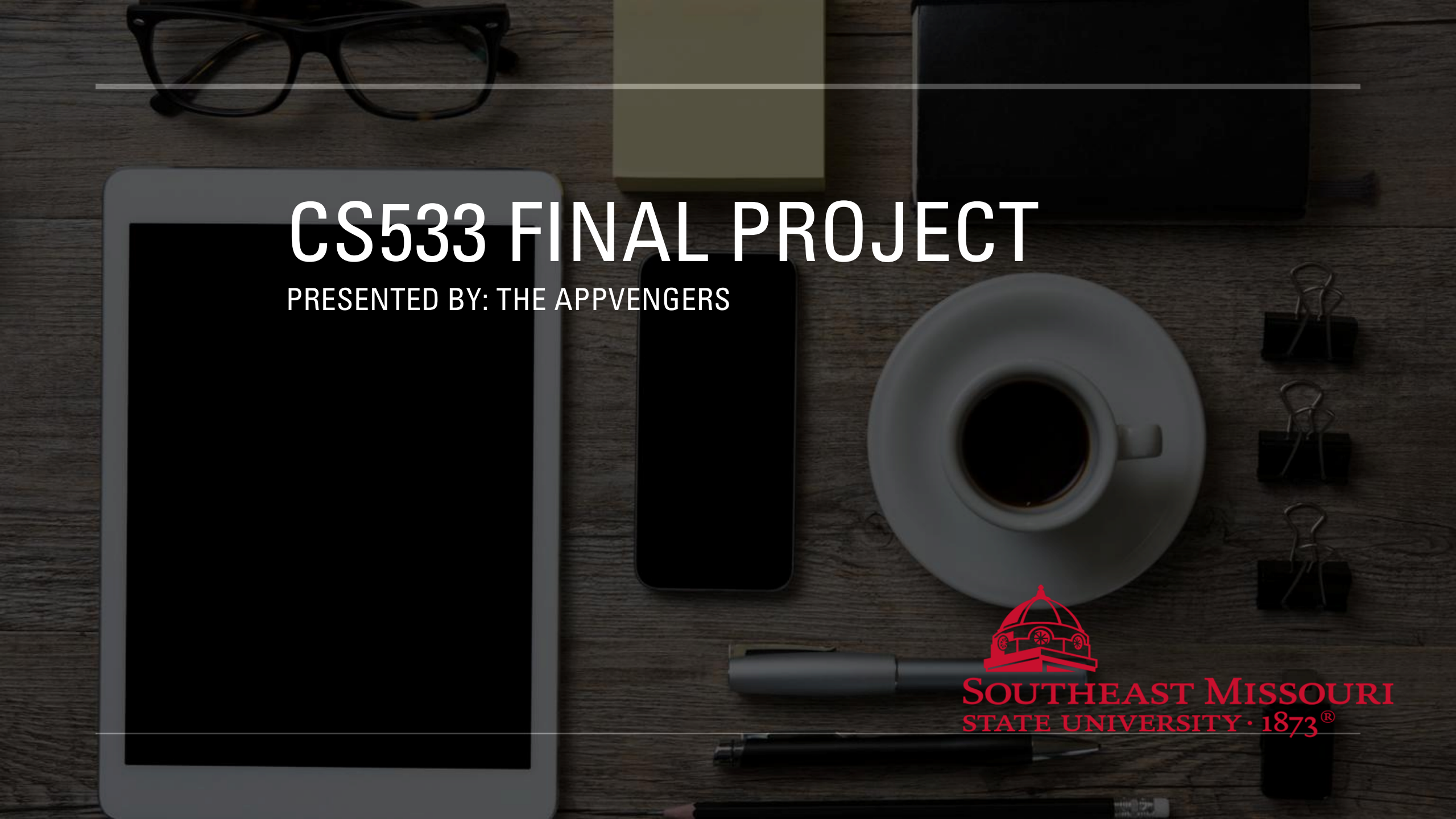


CS533 FINAL PROJECT

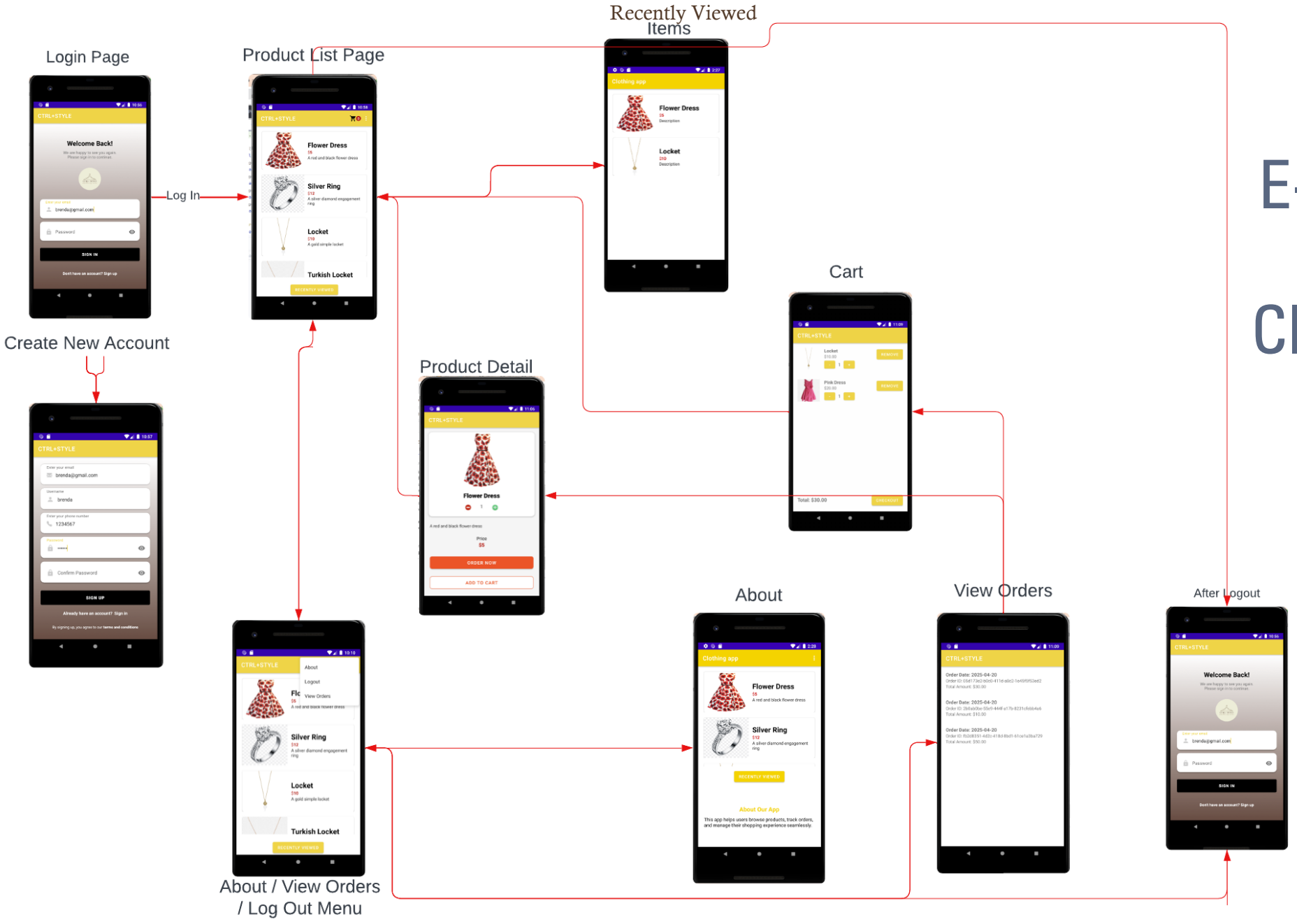
PRESENTED BY: THE APPVENGERS

SOUTHEAST MISSOURI STATE UNIVERSITY · 1873®

Project Analysis

Software Development Life Cycle (SDLC)

E-commerce Clothing App Flowchart



PROJECT ANALYSIS
FUNCTIONAL REQUIREMENTS

User Authentication

- Users must be able to sign up and log in.
- Authentication should use email & password.
- Users should be able to log out.

Product Listing & Search

- Users should see a list of available products.
- Each product should display name, image, price, and description.
- Users should be able to s
for products by scrolling

PROJECT ANALYSIS
FUNCTIONAL REQUIREMENTS

Product Details

- Clicking on a product should take the user to a Product Detail Page.
- The detail page should show image, price, description, and an option to increase and decrease quantity, order, or add to cart.

About Page

- The app should have an “About” page to display general app information.

PROJECT ANALYSIS
FUNCTIONAL REQUIREMENTS

Firebase and SQLite

- Cart items and recently viewed items will be stored in an SQLite database.
- Orders and login/authorization information will be stored in a Firebase Firestore for persistence across sessions/devices.
- Firestore will store and sync data in real time.

Shopping Cart

- Users should be able to add products to the cart.
- Users should be able to view and remove cart items.
- Users should be able to increase or decrease the quantity of items in the cart.
- Users should be able to order listed in the cart.

PROJECT ANALYSIS
FUNCTIONAL REQUIREMENTS

Performance

- The app should load products within 2 seconds.
- The app should be responsive of different screen sizes.

Usability

- The UI should be easy to navigate.
- Buttons and actions should be intuitive.

SDLC - PROJECT ANALYSIS
NON-FUNCTIONAL REQUIREMENTS

Scalability

- The app should handle many users and products without crashing.

Maintainability

- The code should be modular and well-documented.
- Future updates should be easy to implement.
- Resources should be used i
  place of hard-coded tex
  colors.

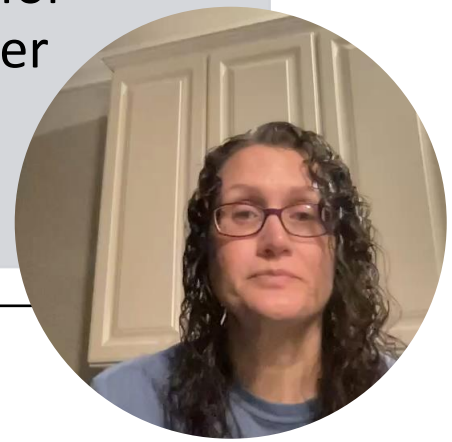
SDLC - DESIGN

Activities

- MainActivity – Displays the product list
- DetailActivity – Shows product details
- OrderActivity – Displays placed orders
- RecentlyViewedActivity – Shows recently viewed products
- CartActivity – Displays products placed in the cart

Database

- Use SQLite (DbHelper.java) for storing items in the cart
- Design a new "Recently Viewed" table to store clicked products
- Implement Google Firebase for storing placed orders and user information



SDLC - IMPLEMENTATION

Product List

- Used RecyclerView in MainActivity to display products from products.json
- MainAdapter.java binds data to the UI

Recently Viewed Activity

- Implemented a button-based navigation to a new RecentlyViewedActivity
- Used SQLite to store and retrieve recently viewed products

SDLC - IMPLEMENTATION

Database Integration

- Extended DbHelper.java to support Recently Viewed Products
- Added necessary methods to store and retriev product data with json

SDLC – TESTING AND DEBUGGING

Key Fixes and Improvements

- Fixed missing UI elements (RecyclerView & Button placement)
- Debugged issue where Recently Viewed list was not updating
- Improved layout for better UI/UX

SDLC – DEPLOYMENT AND MAINTENANCE

GitHub Integration

- Used GitHub Desktop to commit and push changes
- Resolved issues with syncing local commits to the GitHub repository

Future Enhancements

- Improve UI/UX for a more modern design
- Add filtering & sorting for products

ALTERNATIVE SOLUTION ANALYSIS

PRODUCT DATA SOURCE

Current Solution (JSON File)

- Does not impact performance speed
- Cannot change products without updating the app

Alternative Solution (Remote API or Firestore)

- More complex
- Can dynamically change product list

CART & RECENTLY VIEWED ITEMS

Current Solution (SQLite)

- Simple – no required dependencies
- Error handling is harder

Alternative Solution (Room Database)

- Creates an abstract layer over SQLite
- Verifies queries at compile time
- Can respond to live changes
- Larger learning curve

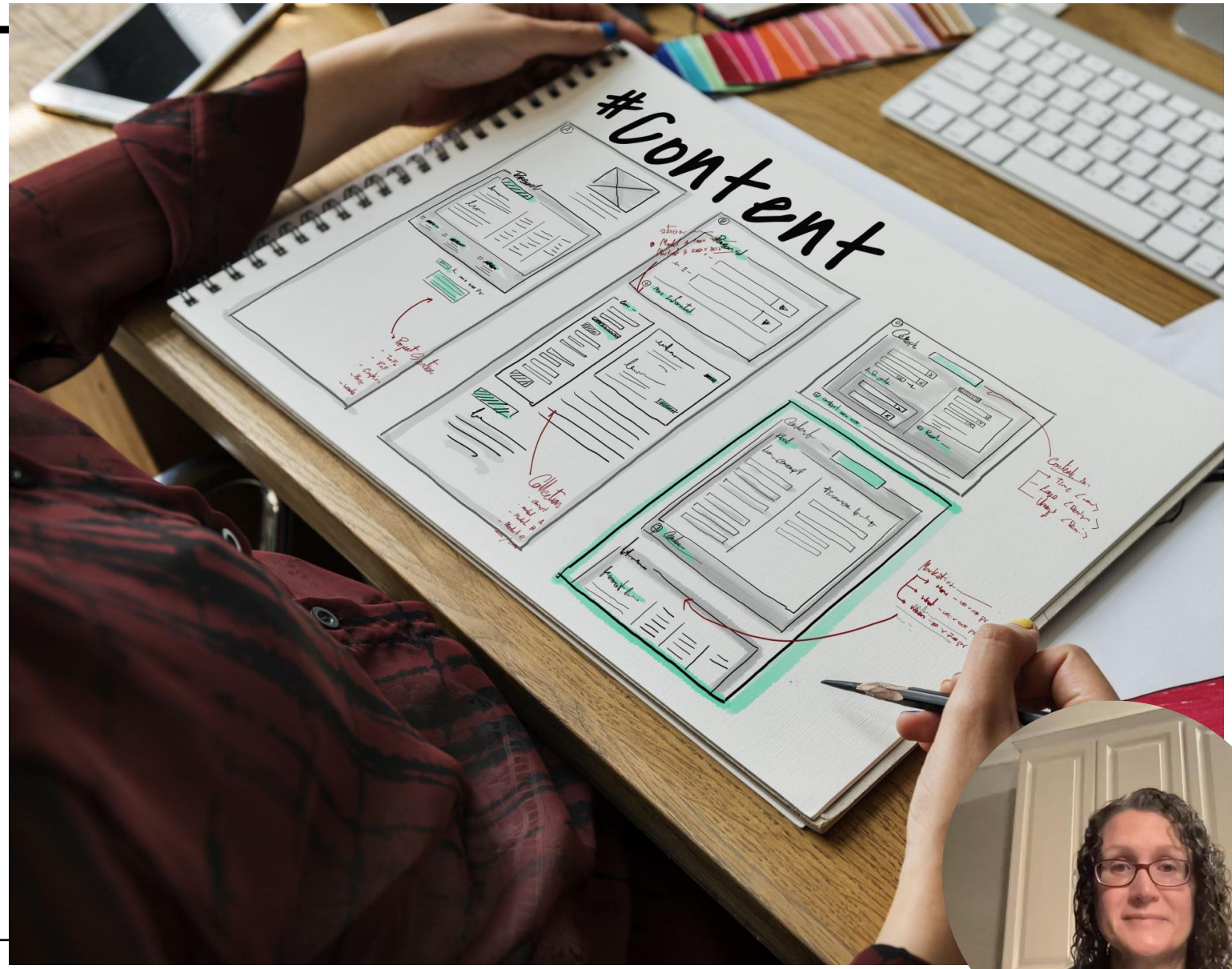
ORDERS AND AUTHENTICATION

Current Solution (Firebase Firestore)	Alternative Solution (Amazon Web Services)
• Less expensive/less flexible • Real time data sync • Only NoSQL	• More expensive/more complex/more flexibility • Does not have real-time data sync • Supports SQL and NoSQL

PROJECT DESIGN

Brenda Nahlik

SHOPPING APP PROJECT CONTRIBUTIONS



OVERVIEW

My contribution improved the mobile shopping app by:

Adding a “Recently Viewed” feature using SQLite

Creating a dedicated activity for recently viewed products

Adding an About page fragment from the menu

Integrating RecyclerViews and persistent storage

FEATURE 1:
RECENTLY SAVED ITEMS

Save
- Objective: Save clicked products to the added recently_viewed table.

Insert
- On product click, insert product into 'recently_viewed' table

Open
- Then open DetailActivity with product data via Intent

INSERT PRODUCT INTO RECENTLY VIEWED TABLE

```
// Insert a recently viewed product
1 usage
public void insertRecentlyViewed(Product product) {
    SQLiteDatabase db = getWritableDatabase();

    // Check if the product already exists
    Cursor cursor = db.rawQuery( sql: "SELECT * FROM " + RECENTLY_VIEWED_TB + " WHERE name=?", new String[]{product.getName()});
    if (cursor.getCount() > 0) {
        cursor.close();
        return; // Product is already in recently viewed, no need to insert
    }
    cursor.close();

    ContentValues values = new ContentValues();
    values.put("name", product.getName());
    values.put("description", product.getDescription());
    values.put("price", product.getPrice());
    values.put("image", product.getImage());

    db.insert(RECENTLY_VIEWED_TB,  nullColumnHack: null, values);
}
```

RETRIEVE RECENTLY VIEWED PRODUCTS FROM TABLE

```
// Retrieve recently viewed products
2 usages
public ArrayList<Product> getRecentlyViewed() {
    ArrayList<Product> recentlyViewedList = new ArrayList<>();
    SQLiteDatabase database = this.getReadableDatabase();
    Cursor cursor = database.rawQuery( sql: "SELECT * FROM " + RECENTLY_VIEWED_TB,  selectionArgs: null);

    while (cursor.moveToNext()) {
        Product product = new Product(
                cursor.getString( i: 1),  // name
                cursor.getString( i: 2),  // description
                cursor.getString( i: 3),  // price
                cursor.getString( i: 4)   // image
        );
        recentlyViewedList.add(product);
    }
    cursor.close();
    return recentlyViewedList;
}
```

VIEW RECENTLY VIEWED ITEMS

Objective: Show list of recently viewed items.

Uses RecyclerView to show items

Uses RecentlyViewedAdapter to bind product data retrieved from SQLite

Loads data from DbHelper.getRecentlyViewed()

VIEW RECENTLY VIEWED ITEMS WITH RECYCLERVIEW

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

DISPLAY RECENTLY VIEWED ITEMS

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="10dp">

    <ImageView
        android:id="@+id/product_image"
        android:layout_width="80dp"
        android:layout_height="80dp"
        android:scaleType="centerCrop"
        android:src="@drawable/ic_launcher_foreground" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:paddingLeft="10dp">

        <TextView
            android:id="@+id/product_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```

FEATURE 2: "ABOUT" FRAGMENT

- Objective: Add informational 'About' page.
- Files:
- AboutFragment.java
- fragment_about.xml
- MainActivity.java
- Selected via menu item
- → triggers loadFragment(new AboutFra

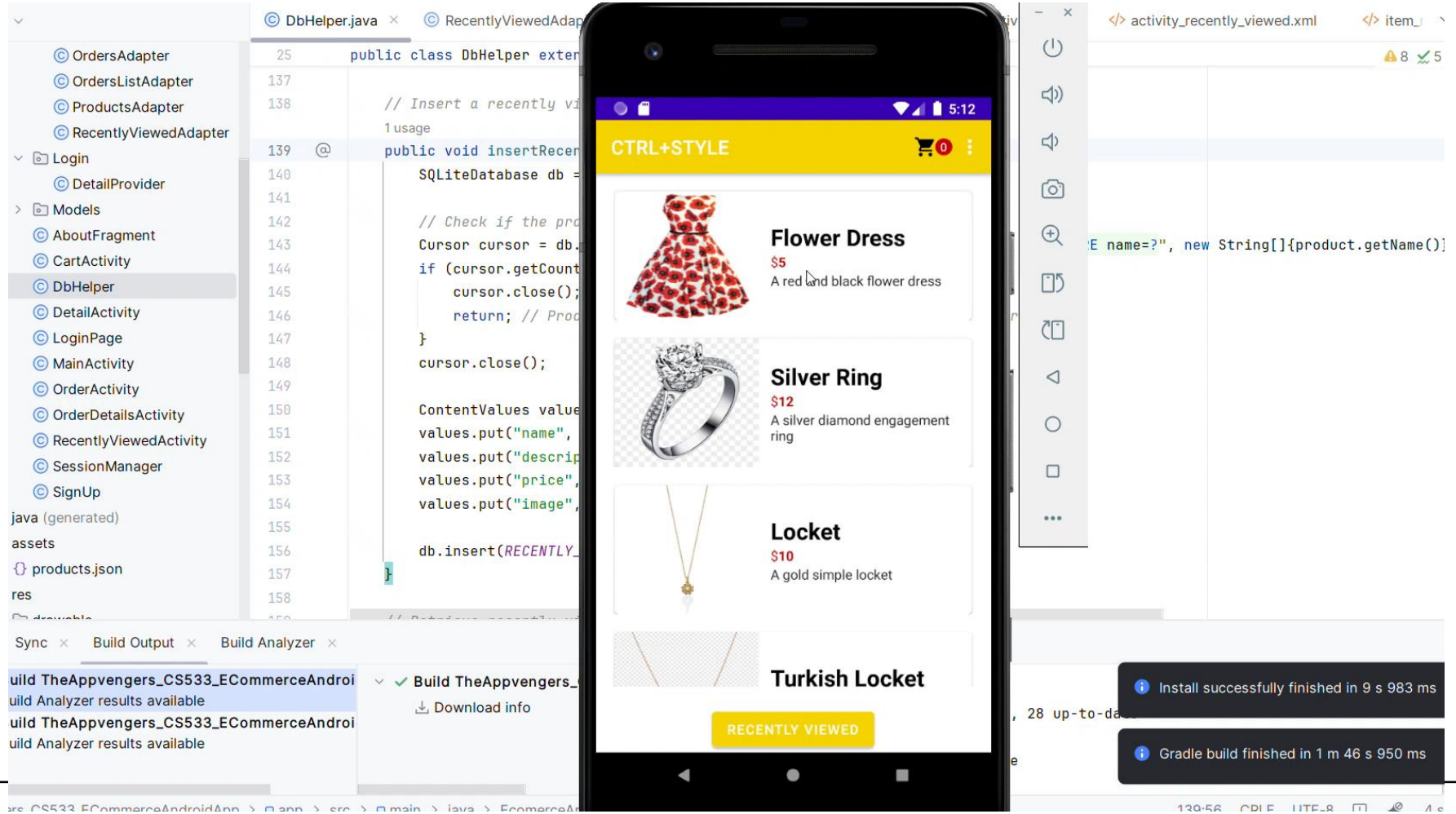
“ABOUT” SECTION LAYOUT

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="20dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="About Our App"
        android:textSize="20sp"
        android:textStyle="bold"
        android:textColor="@color/yellow"
        android:paddingBottom="10dp"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/about_description"
        android:textColor="@color/black"
        android:textSize="16sp"/>
</LinearLayout>
```

DEMONSTRATION



Samana Dahal

Shopping App Project Contributions

PROJECT OVERVIEW

- Implemented Session Manager to maintain user login across app sessions
- Redesigned Sign-Up UI for improved user experience
- Added Logout Button for better session control
- Updated colors.xml for consistent design language
- Contributed to README.md for documentation clarity
- Participated in merge conflicts resolution and code integration

Feature 1: Implemented Session Manager

PROBLEM: USERS WERE LOGGED OUT

EVERY TIME THEY NAVIGATED BACK.

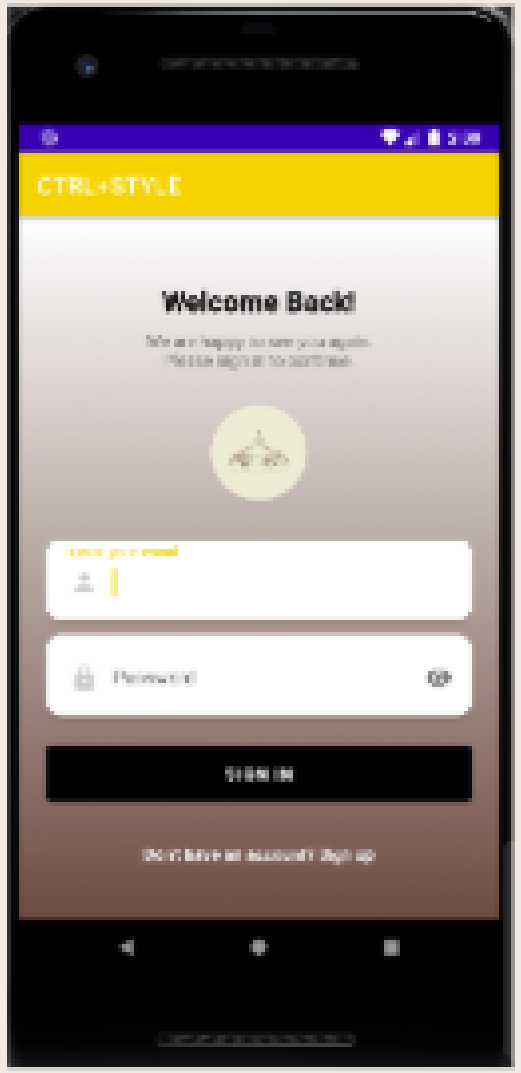
SOLUTION:

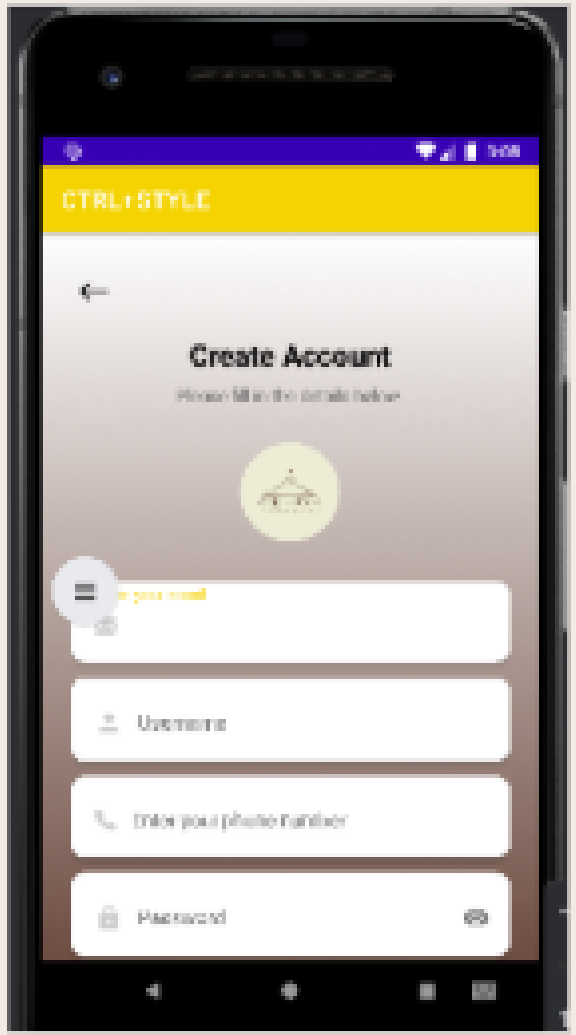
- Introduced a Session Manager using SharedPreferences
- Stored login state and user information securely
- Ensured seamless user experience across sessions

IMPROVED SIGN-UP UI

Changes Made:

- Enhanced form layout and error handling
- Added input validation
- Cleaned up UI to align with modern Android design standards





ADDED A LOGOUT BUTTON

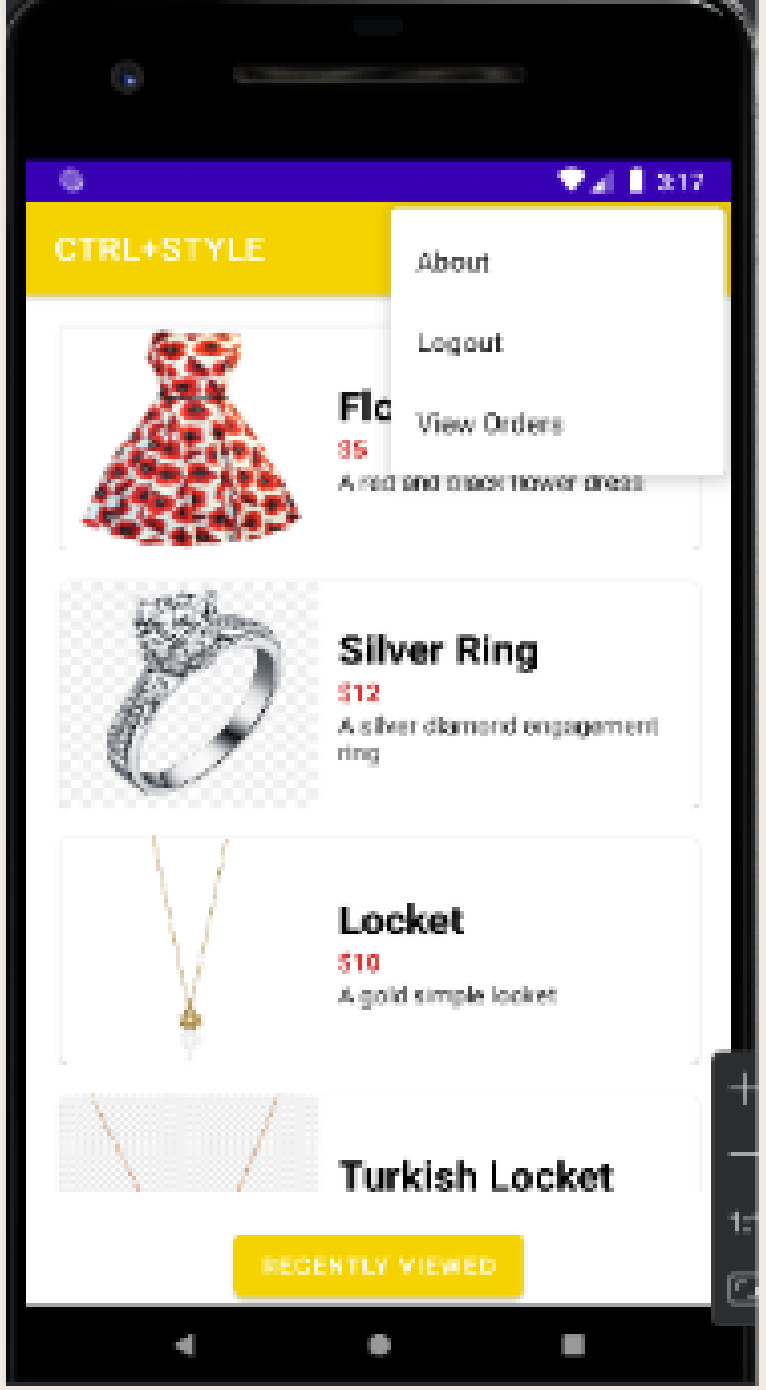
Purpose:
To allow users to explicitly end their session, enhancing both user control and security.

Problem Identified:

- Previously, users had no clear way to log out from the app.
- This led to potential confusion and left sessions open on shared devices.

What I Did:

- Placed a logout icon/button in the top-right corner of the main activity screen.
- Linked it to the Session Manager to clear login data.
- Redirected users to the Login screen upon logout.



ADDITIONAL CONTRIBUTIONS

Minor Enhancements:

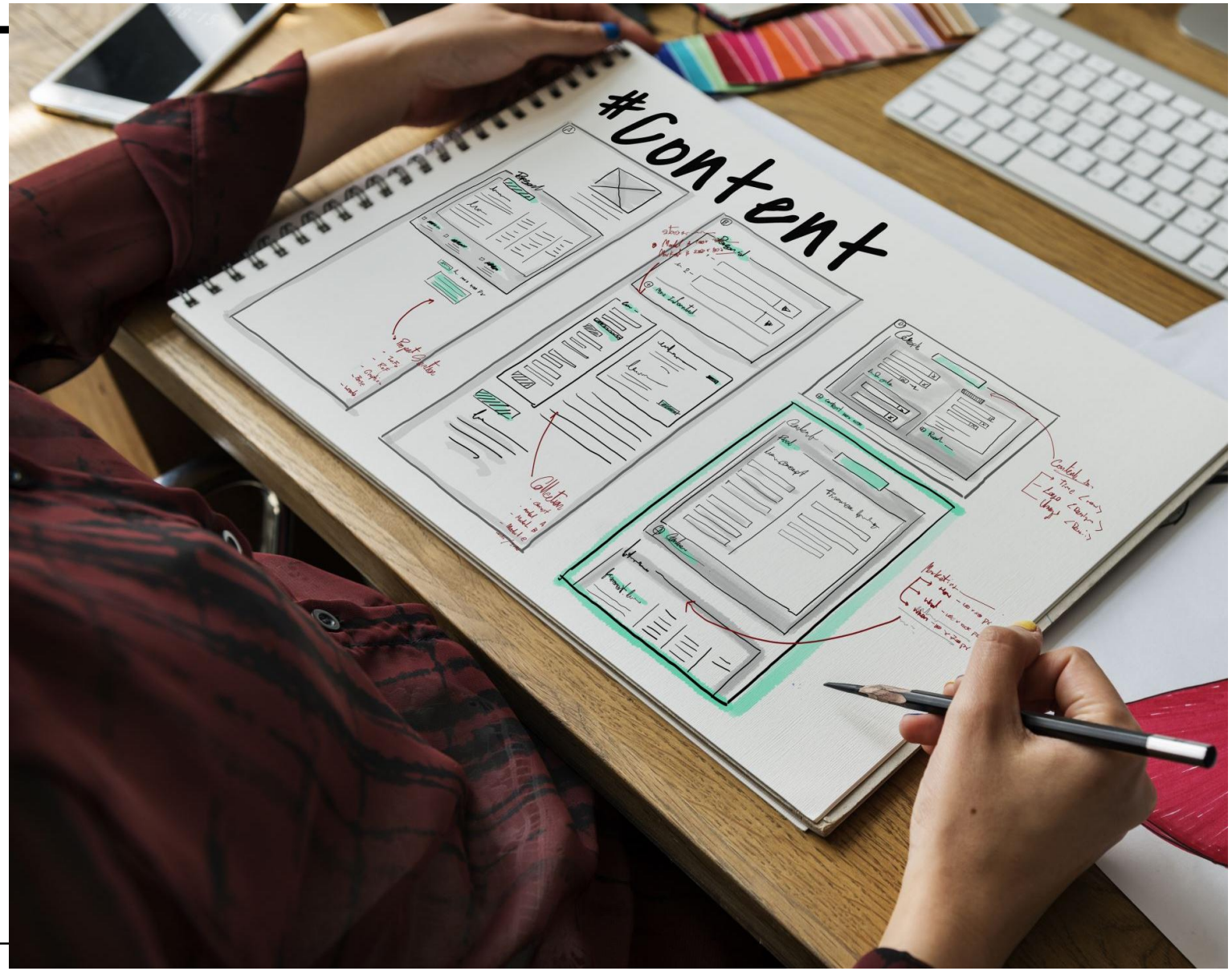
- Updated color themes for branding consistency
- README improvements to guide new developers

Reflection:

- Learned about session persistence, UI/UX principles, and collaborative version control
- Learned about version control using Git and GitHub
- Enjoyed contributing to a real-world e-commerce app with a full team

Morgan Snider

SHOPPING APP PROJECT CONTRIBUTIONS



products.json

```
[
  {
    "name": "Flower Dress",
    "description": "A red and black flower dress",
    "price": "5",
    "image": "dress"
  },
  {
    "name": "Silver Ring",
    "description": "A silver diamond engagement ring",
    "price": "12",
    "image": "ring"
  },
  {
    "name": "Locket",
    "description": "A gold simple locket",
    "price": "10",
    "image": "locket"
  },
  {
    "name": "Turkish Locket",
    "description": "A bronze simple locket",
    "price": "12",
    "image": "lockets"
  },
  {
```

ADDING JSON FUNCTIONALITY

A JSON file is used to load the product information as shown here

products.json | MainActivity.java | Product.java

```java
package EcomerceApp.ShoppingApp.Models;

public class Product {
    private String name;
    private String description;
    private String price;
    private String image;

    // Add getters and setters
    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getDescription() { return description; }

    public void setDescription(String description) { this.description = description; }

    public String getPrice() { return price; }

    public void setPrice(String price) { this.price = price; }

    public String getImage() { return image; }

    public void setImage(String image) { this.image = image; }
}
```

ADDING JSON FUNCTIONALITY

A Product model creates objects from the JSON input

```
// Read JSON from assets
try {
    AssetManager assetManager = getAssets();
    InputStream inputStream = assetManager.open( fileName: "products.json");
    InputStreamReader reader = new InputStreamReader(inputStream);

    String jsonString = convertStreamToString(inputStream);  // Assume this is the JSON file content
    Gson gson = new Gson();
    List<Product> productList = gson.fromJson(jsonString, new TypeToken<List<Product>>(){}.getType());


    ArrayList<MainModel> list = new ArrayList<>();
    for (Product product : productList) {
        // Get the drawable resource ID based on the image name from JSON
        String imageName = product.getImage();
        int imageResId = getResources().getIdentifier(imageName,  defType: "drawable", getPackageName());

        MainModel model = new MainModel(
                product.getName(),
                product.getDescription(),
                product.getPrice(),
                imageName
        );
        list.add(model);
    }
```

ADDING JSON FUNCTIONALITY

In the MainActivity, product.json is read and create a list of Product objects which are then translated into the MainModel objects of the base code and displayed in the recyclerviewer

```
//increase or decrease quantity buttons
        binded.plus.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View view) {
                count=Integer.parseInt(binded.quantity.getText().toString())+1;
                binded.quantity.setText(String.valueOf(count));
                binded.detailprice.setText(String.valueOf(Integer.parseInt(binded.detailprice.getText().toString())+localvarprice));
                Toast.makeText( context: DetailActivity.this,  text: "Quantity Increased "+binded.quantity.getText(), Toast.LENGTH_SHORT).show();
            }
        });
        //decrease quantity
        binded.minus.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {

                if(!binded.quantity.getText().equals("1"))
                {
                    count=Integer.parseInt(binded.quantity.getText().toString())-1;
                    binded.quantity.setText(String.valueOf(count));
                    binded.detailprice.setText(String.valueOf(Integer.parseInt(binded.detailprice.getText().toString())-localvarprice));
                    Toast.makeText( context: DetailActivity.this,  text: "Quantity Decreased "+binded.quantity.getText(), Toast.LENGTH_SHORT).show();

                }
            }
        });
    }
    }
}
```

ERROR FIXING

Fixed an error where the quantity and price of an order was displaying incorrectly

(Item of price $5, quantity of 3 was displaying as $20 total instead of $15)

SHOPPING APP PROJECT CONTRIBUTIONS

Samikshya Adhikari

BRANDING AND LOGIN PAGE DESIGN

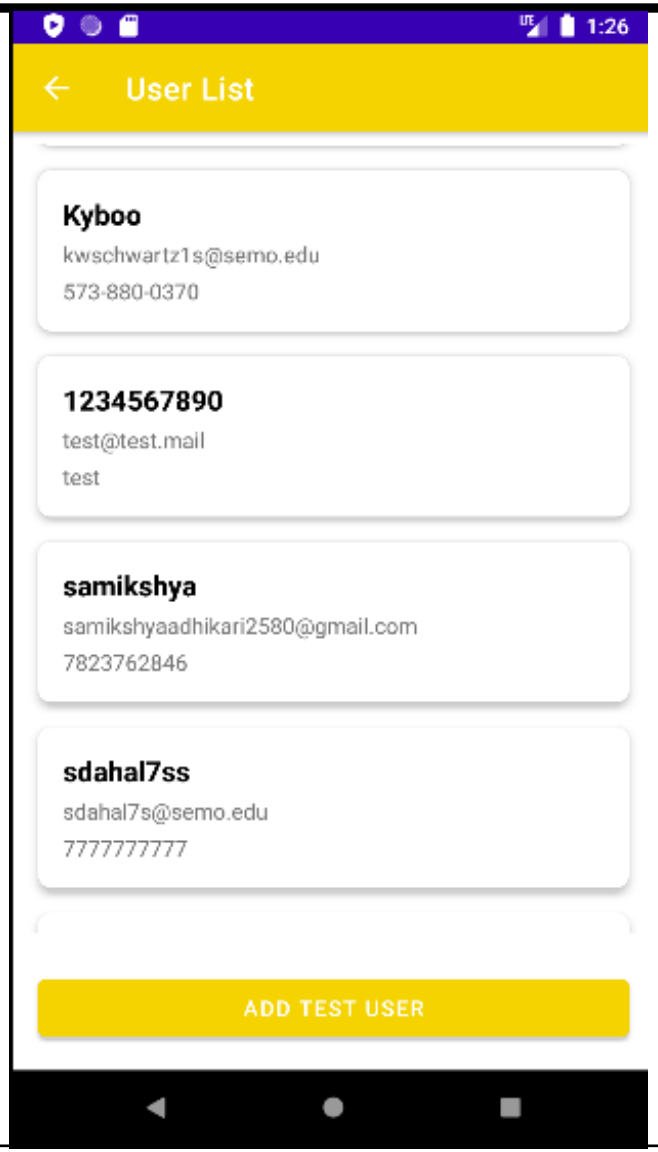
- Created a custom Login Page to establish the app's first impression.
- Branded the app as CTRL+STYLE, setting the theme of the project.
- Designed and applied a new logo and modernized color palette.
- Updated the AppBar to better match the overall background and brand aesthetic.

SPLASH SCREEN AND THEME UPDATES

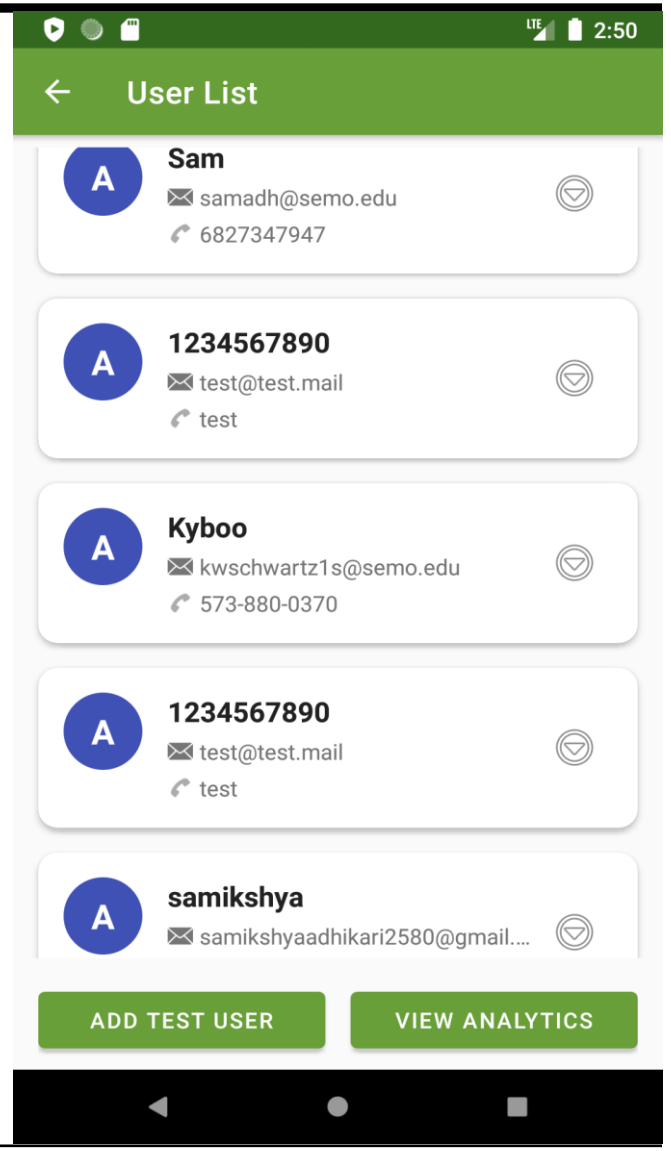
- Developed a new Splash Screen featuring the CTRL+STYLE logo and green background.
- Replaced the earlier yellow theme with a cohesive green-based color theme for visual harmony.
- Enhanced user experience with smoother app loading animations.

THEME UPDATE

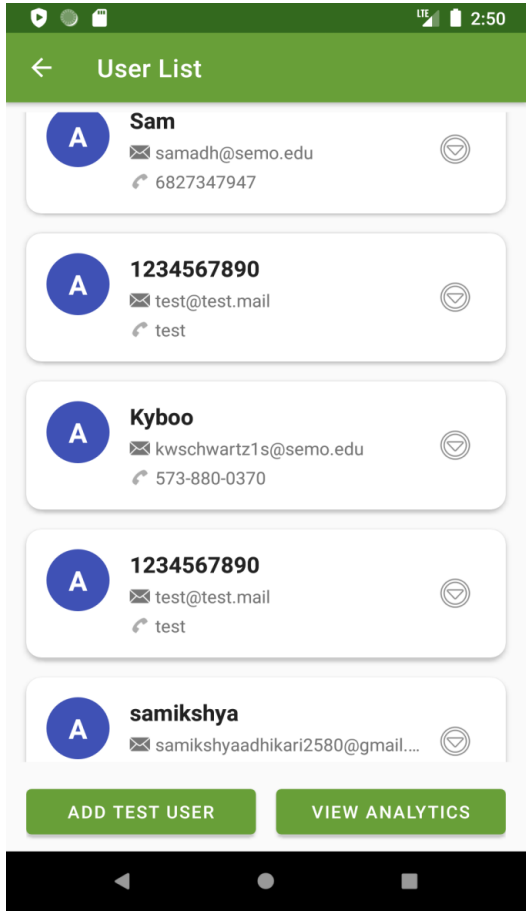
- Before



- After



FIREBASE USER DATA INTEGRATION



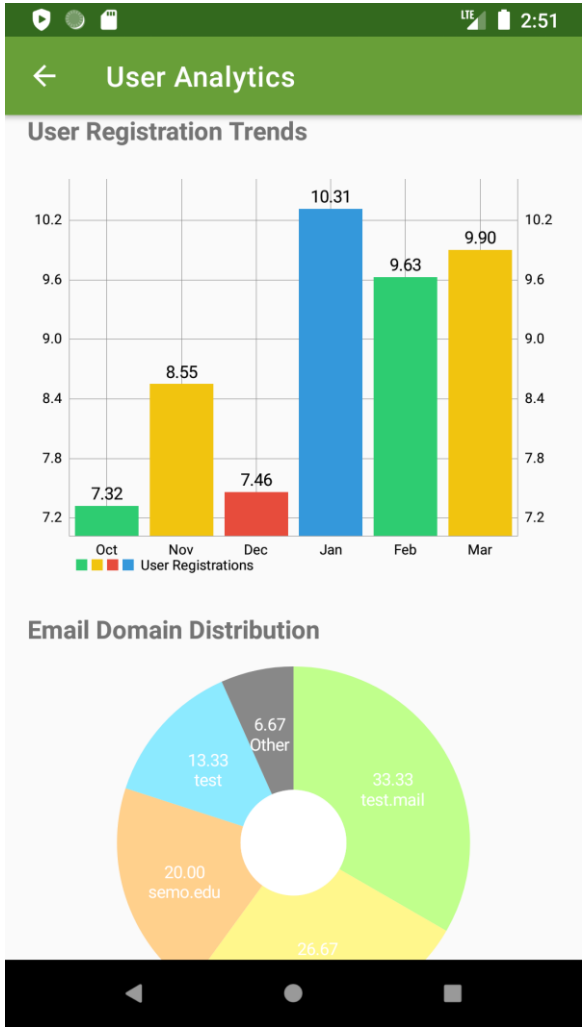
- Fetched registered user details (name, email, phone) from Firestore.
- Displayed user information in a structured User List inside the app.
- Enabled real-time updates whenever a user signed up or edited their details.

USER ANALYTICS DASHBOARD

Built a Bar Chart showing simulated 6-month registration trends.

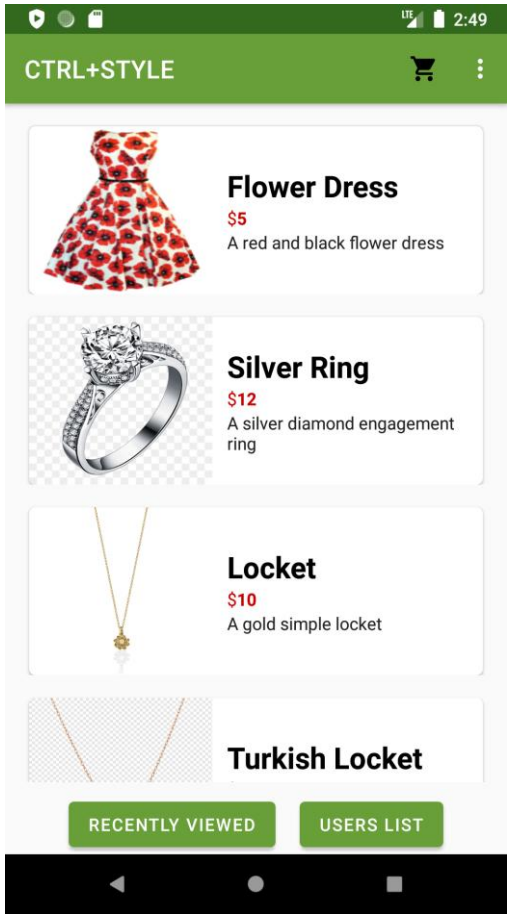
Developed a Pie Chart based on real Firestore data to visualize email domain distribution.

Enabled dynamic, visual insights directly from database queries.



FINAL INTEGRATION AND PROJECT COMPLETION

- Merged all UI, functional, and visualization updates into the Main branch.
- Coordinated last-minute improvements without breaking existing features.
- Contributed to delivering a fully functional, branded, and polished final shopping app.



Kyle Schwartz

SHOPPING APP PROJECT CONTRIBUTIONS

OVERVIEW

Upgraded project to latest Gradle and Java technologies supported by Firebase.
Implemented Google Firebase for storing user and order information
Developed a cart activity using SQLLite and a checkout feature that sends cart data from SQLLite to Firebase. Also added an “Add to Cart” icon in the detail activity, and a cart icon in the menu bar.
Implemented more advanced authentication with Google Firebase Authenticate
Finalized “Order Now” and “Checkout” buttons to send place an order by sending user and order information to Firebase.
Re-coded the orders activity to retrieve order data from Firebase.

UPGRADES TO PROJECT ARCHITECTURE

(REQUIRED TO USE FIREBASE APIS)

- This was the most difficult part of the project for me because it required major changes to the Gradle code, and a good understanding of how each component in the app worked.
- Firebase required the project to use a more modern build setup. To resolve several errors, I made the following upgrades to the original, somewhat outdated, template.
  - Upgraded Gradle compile SDK from 32 to 35.
  - Modified existing build files to use the more modern Kotlin programming language instead of Groovy.
  - Upgraded the Java version from 1.8 to 11.
  - Re-coded any code that threw depreciation errors.
  - Began to use Android Studio Meerkat for Development.

FIREBASE INTEGRATION

Store

- Objective: Store user an order information from all app instances in one cloud-based database.

Upload

- After clicking the “Order Now” or “Add to Cart” buttons order and user information is uploaded to the cloud.

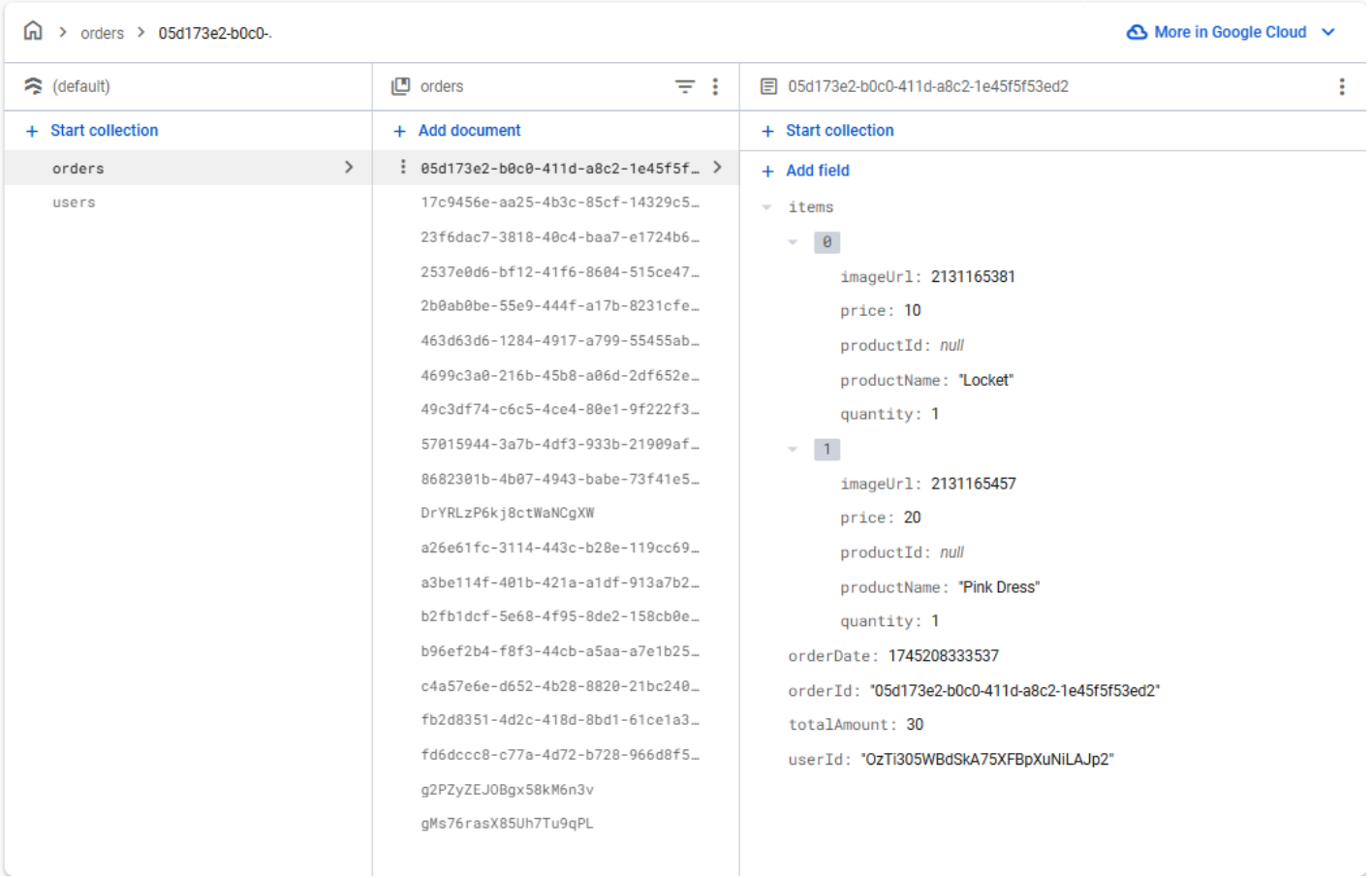
Retrieve

- The data can then be collected and retrieved from every app instance in Google Firebase.

FIREBASE INTEGRATION

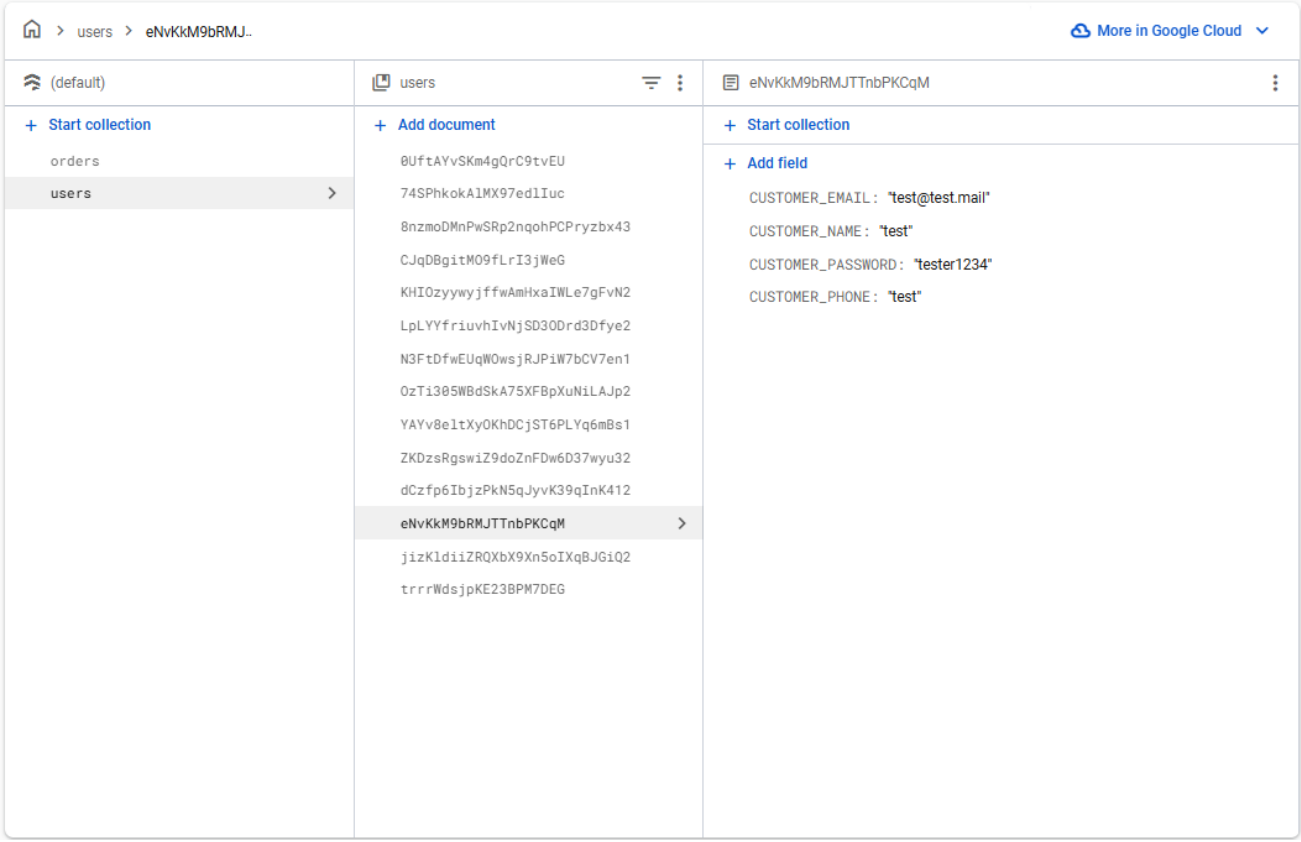
- The google-services.json file is what ensures that all instances of the app are tied to the same database and authentication systems.
- Necessary libraries and dependencies that are needed to work with the Firebase API were added to the Gradle build files.
- The code largely involved creating a Firebase instance, establishing a connection, and creating a document with keys and data.
- Most of the data in Firebase was designed to closely mimick the structure originally in SQLLite to ease in the process of integrating Firebase with existing features.

SCREENSHOT OF AN ORDER WITHIN FIREBASE



EXAMPLE USER DATA STORED IN FIREBASE

(PLAIN TEXT PASSWORD IS STORED FOR DEBUGGING AND DEMONSTRATION PURPOSES ONLY AND WOULD NOT BE RETAINED IN A DEPLOYED APPLICATION.)



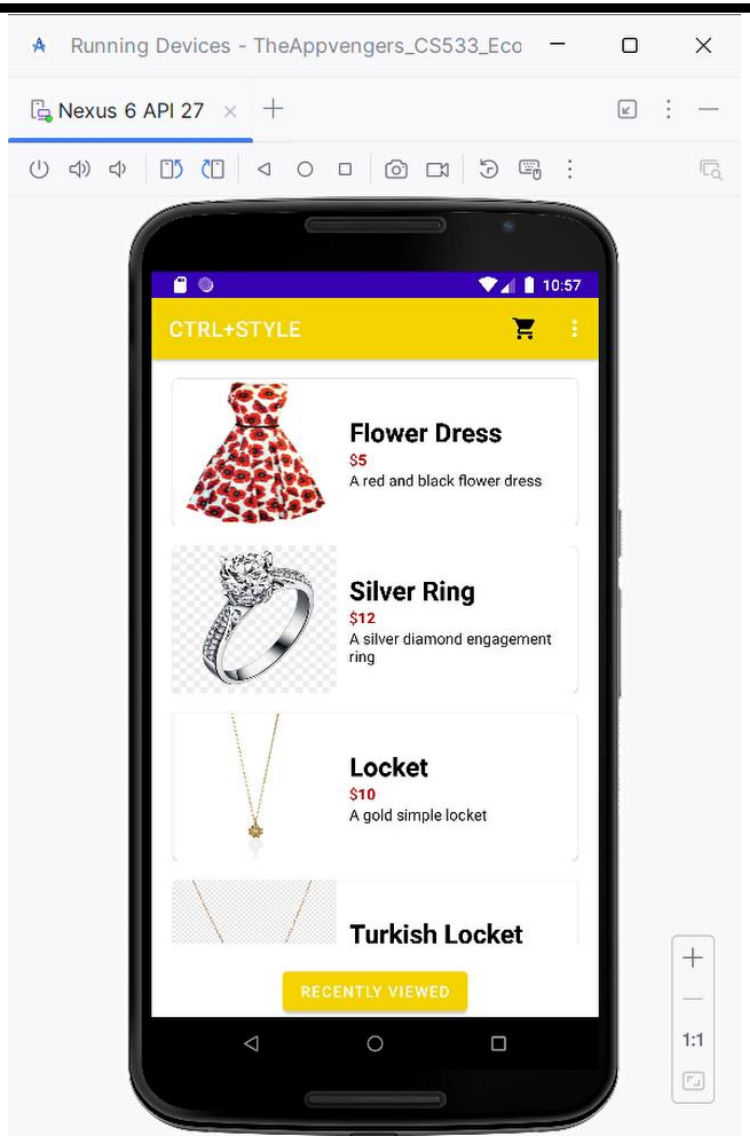
CART FEATURES

The cart allows users to:

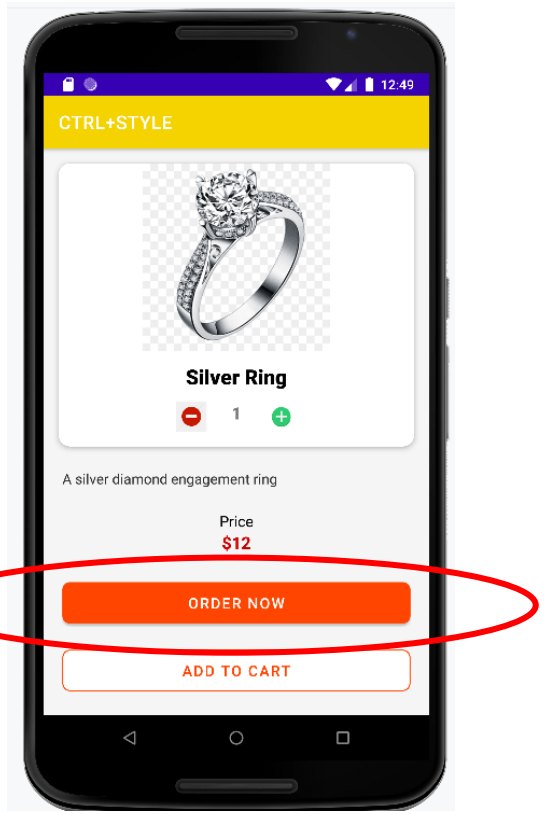
- Select a list of items before checking out.
- Modify the quanitity of each item.
- Remove any unwanted items.
- View the total price of all items in the cart.
- Order all the items currently in the cart.
- Navigate to the orders activity after the checkout button is clicked

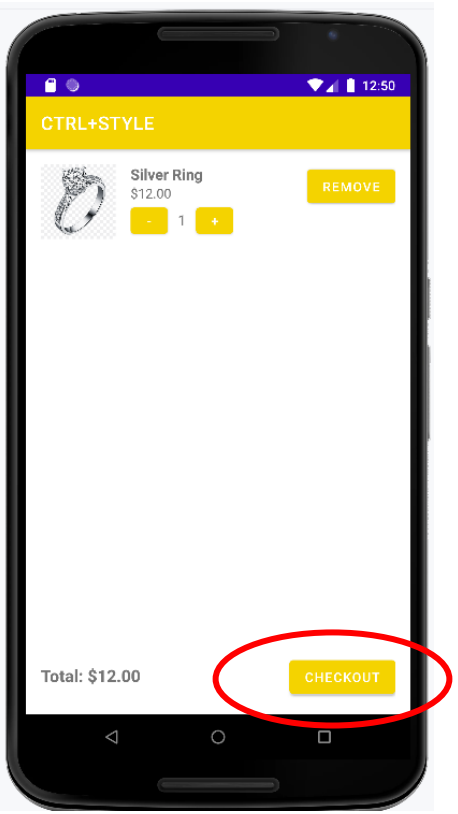
- The cart required the implementation of lists within the order document in Firebase.
- The cart uses a SQLLite cart table to store the data after the "Add to Cart" button in the product details activity is clicked.
- It also required a modification the menu bar with a new icon, anew data model, a new adapter, a new activity, and two new layout files to implement.

CART DEMO VIDEO



CLICKING "ORDER NOW" IN THE DETAIL ACTIVITY OR "CHECKOUT" IN THE CART SENDS THE NECESSARY ORDER INFORMATION TO GOOGLE FIREBASE





GOOGLE FIREBASE AUTHENTICATION OBJECTIVES

Authenticate

- Authenticate users using cloud based encrypted services that retrieve account information from the web.

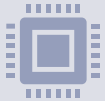
Manage

- Allow the organization using the app to view and manage users of the app

Consolidate

- Ensure sensitive user authentication data is handled by an authentication service, while user data such as mailing address or phone number is handled in a users table.

GOOGLE FIREBASE AUTHENTICATION



Similar to Firebase, Google's authentication service required the google-services.json file

It also required two necessary libraries that needed to be added to the Gradle build files.

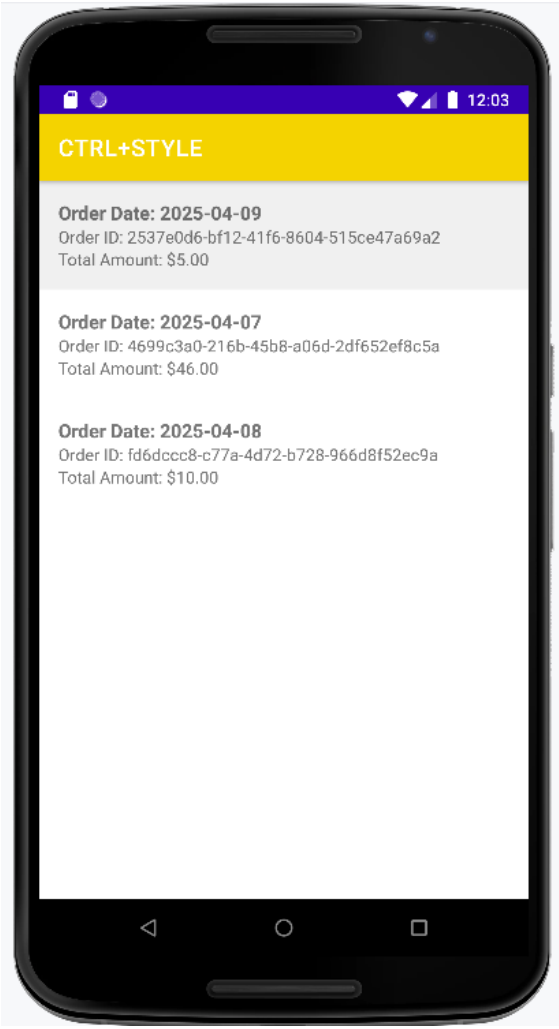
After Google Firebase Authentication and its necessary libraries were imported, coding was simple and consisted of establishing a connection and modifying the existing SQLLite log in and sign up code to instead send the data to Firebase.

ORDERS ACTIVITY IMPLEMENTATION

- The cart and the storing of multiple items per order as opposed to the single item per order of the original template meant that the order activity had to be coded almost from scratch.
- The order activity was divided into two parts. A list of all the orders, and a list of all the products in an order.
- This information is retrieved from the orders table in Firebase. The userId field that is stored in each orders document, and the userId field that is stored in the authentication system ensures that only orders that a given logged in user placed are displayed.

ORDER ACTIVITY (ORDER LIST)

The orders activity consists of clickable tiles called order list items. Clicking on an order list item takes the user to the order details activity, which displays the products list associated with a given order number.



METHOD TO RETRIEVE ORDER DATA FROM FIREBASE

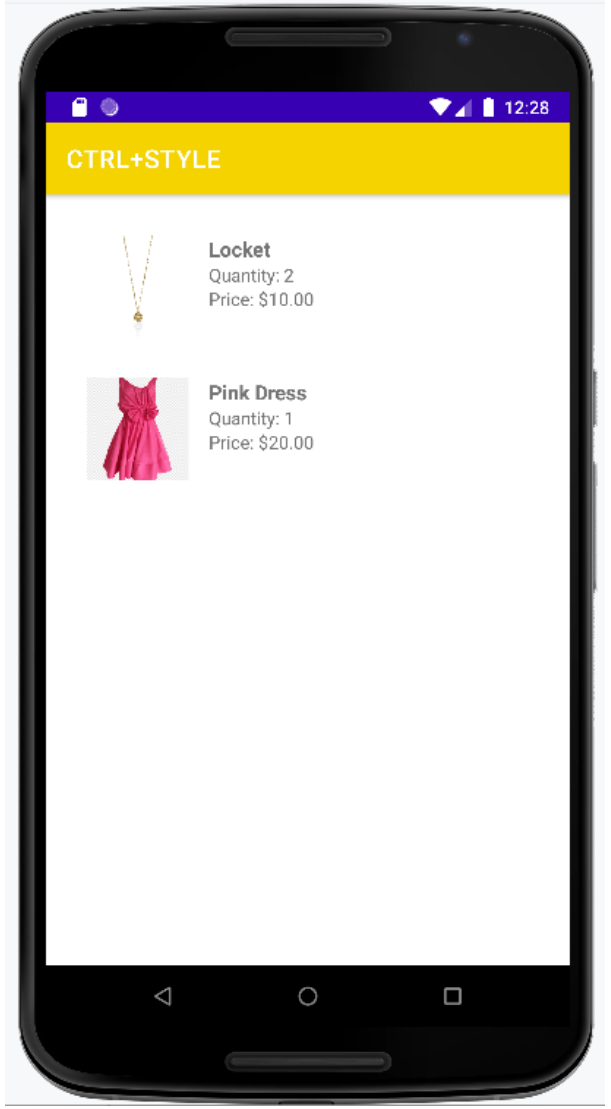
```
private void fetchOrdersFromFirestore() {
    String userId = firebaseAuth.getCurrentUser().getUid(); // Get the current user's UID

    firestore.collection( collectionPath: "orders") CollectionReference
            .whereEqualTo( field: "userId", userId) // Filter orders by the current user's ID
            .get() Task<QuerySnapshot>
            .addOnSuccessListener( QuerySnapshot queryDocumentSnapshots -> {
                if (!queryDocumentSnapshots.isEmpty()) {
                    List<OrdersModel> fetchedOrders = queryDocumentSnapshots.toObjects(OrdersModel.class);
                    ordersList.clear();
                    ordersList.addAll(fetchedOrders);
                    adapter.notifyDataSetChanged(); // Notify the adapter of data changes
                } else {
                    Toast.makeText( context: OrderActivity.this,  text: "No orders found", Toast.LENGTH_SHORT).show();
                }
            })
            .addOnFailureListener( Exception e -> {
                Toast.makeText( context: OrderActivity.this,  text: "Error fetching orders: " + e.getMessage(), Toast.LENGTH_SHORT).show();
                Log.e( tag: "OrderActivity",  msg: "Error fetching orders", e);
            });
}
```

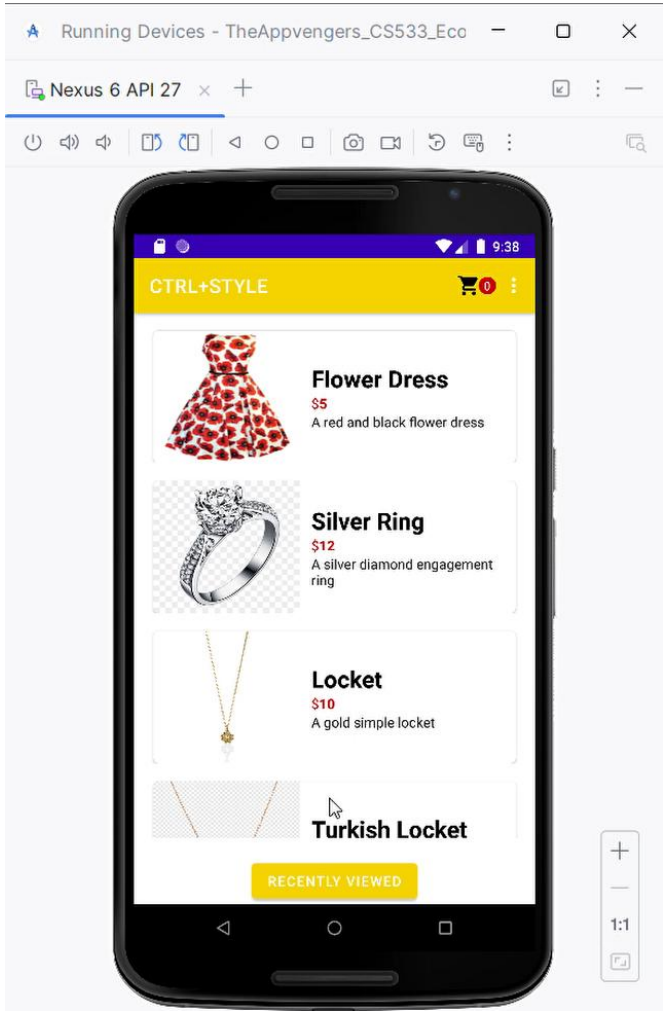
ORDER ACTIVITY (PRODUCT LIST)

The order details activity displays its information based on the orderId field that is retrieved from the intent when an order list item is clicked.

```
// Fetch products for the selected order
String orderId = getIntent().getStringExtra( name: "orderId");
fetchOrderDetails(orderId);
```



ORDERS ACTIVITY VIDEO DEMO



BUG FIXES

- Changed the scaleType of the images in the activity_detail.xml to use fitCenter instead of centerCrop to prevent parts of the images from being hidden when cropped.
- To fix the item counter on cart icon not updating, I created a global variable to store the cart_badge element, assigned the value to it in the onCreateOptionsMenu function, and updated in each time the MainActivity resumes.
- The total price and cart badge did not update until each activity was reloaded. I fixed this by adding listeners to update the TextViews live whenever the SQLLite cart table was modified.

SHOPPING APP PROJECT CONTRIBUTIONS

NAYEEM HOSSAIN

OVERVIEW

- What I Worked On:

- Integration of different app modules

- Managing GitHub branches and pull requests

- Designing and executing test cases

- Automating tests using JUnit

- Bug tracking on Google Sheets

TOOLS & TECHNOLOGIES I USED

- GITHUB: BRANCHING, PRS, ISSUE TRACKING
- ANDROID STUDIO: DEVELOPMENT, EMULATOR
- FIREBASE: AUTH, DATABASE
- JUNIT: TESTING
- GOOGLE SHEETS: MANUAL BUG LOGGING

KEY TASKS COMPLETED

Integration Tasks:	- Connected Firebase auth to login	- Integrated cart with checkout buttons	- Handled backend-frontend API flows
Testing Tasks:	- Created unit test cases	- UI/UX testing on devices	- Full integration testing

CHALLENGES FACED & SOLUTIONS

Challenges:

- Merge conflicts
- API call failures
- UI layout issues

Solutions:

- Better Git practices
- Improved API integration strategies
- Responsive UI design

REFLECTIONS AND LEARNINGS

- Improved GitHub collaboration

- Early and continuous testing

- Delivered better user experience

- Gained hands-on Firebase and JUnit skills

PROJECT PROGRESS EVALUATION

CHALLENGES AND RISKS

Challenges:	Upgrading project environment (Gradle, SDK, Kotlin DSL) to support Firebase integration.	Migrating from local SQLite to Firebase without breaking existing features.	Creating real-time dashboards with partial data (no timestamps initially available).
Risks:	Potential app crashes due to large Firebase updates.	Compatibility issues when shifting to newer Android build versions.	Risk of user session loss if authentication was not managed properly.

LIMITATIONS AND TRADEOFFS

Limitations:

- Registration trends (bar chart) use simulated data due to missing timestamps.
- Plain-text storage of some debugging credentials temporarily for demo purposes (not production secure).
- Limited product details due to scope/time constraints (no payment integration).

Tradeoffs:

- Prioritized Firebase setup and core functionality over implementing a full admin panel or payment system.
- Focused on front-end polish and user analytics rather than expanding product categories.

SOLUTIONS IMPLEMENTED

- Upgraded Gradle, SDKs, and Java versions to resolve Firebase compatibility issues.
- Redesigned database structure in Firebase to mirror relational logic and ease migration from SQLite.
- Simulated timestamp data to create a functional bar chart for registration trends.
- Session manager implemented to retain user login even after app closure.
- Image scaling issues fixed by changing scaleType to fitCenter in layouts.
- Created real-time listeners to update Cart badge and Total price dynamically.

DISCUSSION AND OVERALL OUTCOMES

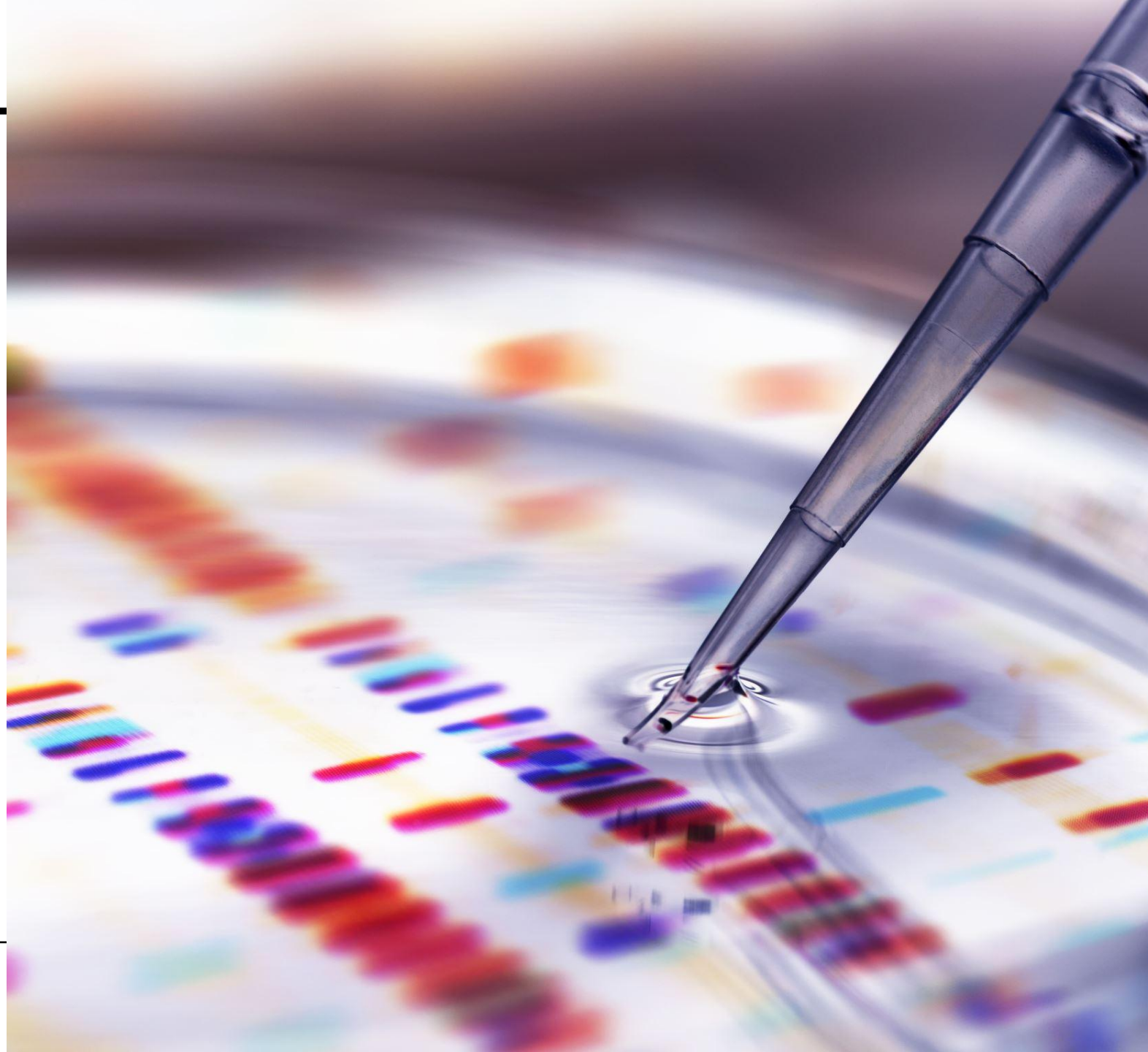
- Successfully built a branded and functional e-commerce app (CTRL+STYLE) despite late feature additions.
- Managed last-minute changes (splash screen update, user detail fetching, data visualization) without major bugs.
- Developed a scalable base for future features such as secure payments, admin dashboards, and real-time notifications.
- Final testing showed a stable app with live Firebase integration, polished UI, and a consistent shopping experience.

DEVELOPMENT

INTEGRATION AND TESTING - OVERVIEW

- Purpose of this Section:
- To ensure that all modules of the shopping app integrate smoothly and the app functions reliably under real-world conditions.
- Key Areas Covered:

- Module Integration
- GitHub-based Collaboration
- Functional, UI/UX & Security Testing
- Automation & Bug Tracking



INTEGRATION APPROACH

- Development & Collaboration Strategy:
- Modular Design: Features like cart, checkout, and user profile built as separate components.
- Version Control with GitHub: Branching strategy used for individual features, merged via Pull Requests.
- Continuous Integration: Frequent merges and testing helped catch issues early.

GITHUB INTEGRATION WORKFLOW

Tools & Practices:

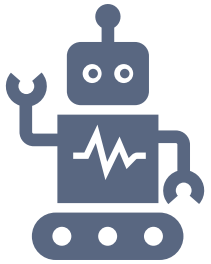
- Feature Branching: Each teammate worked on a dedicated branch.

- Pull Requests (PRs): Used for peer review and code quality checks.

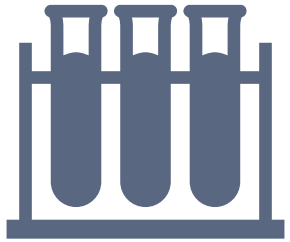
- Issues & Projects: Tasks, bugs, and progress tracked on GitHub board.

- Commits & Tags: Structured commits for easy traceability and tagging for major releases.

TESTING TYPES IMPLEMENTED



1. Unit Testing – Individual logic like login verification, cart updates.



2. System Testing – Entire app tested end-to-end.

3. User Acceptance Testing – Conducted mock runs and collected user feedback.

UI/UX TESTING

- Visual & Interactive Consistency:
- Tested across different screen sizes and devices.
- Ensured smooth navigation and minimal latency in screen transitions.
- Verified proper display of images, fonts, and layouts.
- Employed both emulators and real Android devices for thorough validation.

SECURITY & PERFORMANCE TESTING

Security Measures:

- Used Firebase Authentication for secure login.

- Firebase APIs ensured secure data transfer.

Performance Checks:

- Load testing of product browsing and checkout flow.

- Monitored response time of key actions like checking out and viewing product details.

AUTOMATION & BUG TRACKING

Automated Testing:

- JUnit used for unit and functional test cases.
- Automated test runs triggered before major PRs.

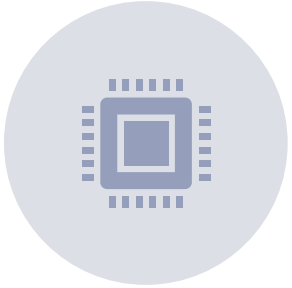
Bug Tracking with GitHub:

- Bugs logged as GitHub Issues, linked to commits and PRs.
- Prioritized using labels like 'bug', 'high priority', 'enhancement'.

KEY OUTCOMES & LEARNINGS

- GitHub streamlined team collaboration and issue tracking.

- Integration testing revealed issues with the data sent to and from Firebase.



- Using multiple emulators helped catch UI edge cases missed.

- Lesson: Integration + Testing = Reliable, User-Friendly App.

IMPLEMENTATION

RESOLUTION OF PROBLEMS

- *Most problems and solutions have been addressed earlier in the presentation, as we explained each feature implementation.*
- **Problems Resolved:**
- Incorrect image scaleType causing images to scale offscreen in product details – Kyle Schwartz changed scaleType to use fitCenter instead of centerCrop.
- Incorrect total price being calculated when adding a quantity greater than 3 in the cart – Morgan Snider created a variable, localVarPrice, to track the price of each item.
- User logged out each time the app is closed – Samana Dahal added a session manager to keep the user logged in, and a logout button to log the user out.

LAST MINUTE FIXES

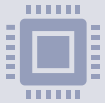
(AFTER SOME OF THE PRESENTATION WAS COMPLETED)

- Item counter on cart icon not updating – Kyle Schwartz created a global variable to store the cart_badge element, assigned the value to it in the onCreateOptionsMenu function, and updated in each time the MainActivity resumes.
- Total price and cart badge did not update until each activity was reloaded – Kyle Schwartz added listeners to update the TextViews live whenever the SQLLite cart table was modified.

REAL WORLD SCENARIO

Key Requirements for e-commerce = View products + Place Orders

The app would be suitable to implement into a real world scenario thanks to its implementation of basic features required for an e-commerce app.



This is a basic demo that could be expanded on to create a fully functional e commerce app.

THE PRODUCTS

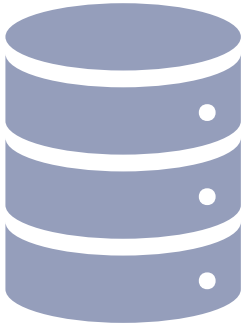
For demonstration purposes our app uses a JSON file stored in the app, but uses the same process of reading the JSON as though the JSON was being pulled from a web service.

The JSON file the products are loaded from mimics a JSON web service which could pull product data from a organizations existing product management systems.

The process of reading the JSON would be similar with the addition of the necessary code and a library for handling requests from cloud services or the web.

THE ORDERS

Google Firebase provides a centralized location where all order and user data can be stored from each app instance



The data from Firebase can then be used with backend order processing systems to fulfill customer orders.

WHAT WE STILL NEED

- A real e-commerce app would need
  - to store more data than we collect in our app
  - a payment system
  - more advanced security measures
  - an order processing system connected to Firebase
  - a more robust product management system

EVALUATION AND FUTURE WORK

Enhanced Product Filtering and Search
- Implement advanced filtering options (e.g., by size, brand, price range).
- Add a smart search with auto-suggestions and recent searches.

User Profiles and Order History
- Allow users to view and manage their profile.
- Display previous orders and order status updates.

Payment Gateway Integration
- Integrate secure payment options (credit/debit, PayPal, digital wallets).

Wishlist and Favorites
- Enable users to save items they like for future purchases.

Admin Dashboard
- Build an admin panel to manage products, view analytics, and track sales.

Real-time Notifications
- Push notifications for new arrivals, order updates, and exclusive deals.

CREDITS

- Kyle Schwartz – Project Leader / Firebase Developer
- Nayeem Hossain - Quality Assurance Specialist (Mobile Application Tester)
- Morgan Snider – Database Developer
- Brenda Nahlik – Mobile Application Architect
- Samana Dahal – Mobile Application Developer
- Samikshya Adhikari – Data Scientist

REFERENCES

- CoderMian. (n.d.). *Shopping-App* [Source code]. GitHub. https://github.com/CoderMian/Shopping-App.git

THANK YOU!