Documentary: API Calls in the Kstudents Android App

The **Kstudents** Android app is designed to facilitate seamless interaction with backend services through a well-structured API layer. This layer is implemented using **Retrofit**, a type-safe HTTP client for Android and Java. Below, we explore the key API services and their functionalities.

---

1. EventApiService
This service handles event-related operations and analytics.

Endpoints:
- **Get All Events:**
  ```java
  @GET("Exam/m_events.php?action=getAllEvents")
  Call<EventsResponse> getAllEvents();
  ```
  Retrieves all events from the server.

- **Get Updated Events:**
  ```java
  @GET("Exam/m_events.php?action=getUpdatedEvents")
  Call<EventsResponse> getUpdatedEvents(@Query("lastUpdate") String lastUpdateTime);
  ```
  Fetches events updated after a specific timestamp.

- **Submit Analytics:**
  ```java
  @POST("Exam/EventAnalytics.php")
  Call<AnalyticsSubmissionResponse> submitAnalytics(@Body AnalyticsRequest analyticsRequest);
  ```
  Submits analytics data to the server.

- **Get Event Analytics Report:**
  ```java
  @GET("Exam/EventAnalytics.php?action=eventReport")
  Call<EventAnalyticsReport> getEventAnalyticsReport(@Query("event_id") int eventId);
  ```
  Retrieves analytics for a specific event.

- **Get Sponsor Report:**
  ```java
  @GET("Exam/EventAnalytics.php?action=sponsorReport")
  Call<SponsorReportResponse> getSponsorReport(@Query("sponsor_id") int sponsorId);
  ```
  Fetches analytics for a specific sponsor.

- **Get Overall Stats:**
  ```java
  @GET("Exam/EventAnalytics.php?action=overallStats")
  Call<OverallStatsResponse> getOverallStats();
  ```
  Provides overall analytics statistics.

---

2. Entrance_Api
This service manages notifications related to entrance exams.

Endpoints:
- **Get Notifications:**
  ```java
  @GET("Exam/Entrance.php")
  Call<EntranceResponse> getEntranceNotifications();
  ```
  Retrieves entrance exam notifications.

- **Add Notification:**
  ```java
  @POST("Exam/Entrance.php")
  Call<Map<String, String>> addEntranceNotification(@Body Map<String, Object>
notification);
  ```
  Adds a new notification.

- **Update Notification:**
  ```java
  @PUT("Exam/Entrance.php")
  Call<Map<String, String>> updateEntranceNotification(@Body Map<String, Object>
notification);
  ```
  Updates an existing notification.

- **Delete Notification:**
  ```java
  @DELETE("Exam/Entrance.php")
  Call<Map<String, String>> deleteEntranceNotification(@Body Map<String, Object>
notification);
  ```
  Deletes a notification.

---

3. ContactApiService

This service facilitates user communication through a contact form.

Endpoints:
- **Submit Contact Form:**
  ```java
  @FormUrlEncoded
  @POST("Exam/m_contact.php?action=submitContactForm")
  Call<ContactResponse> submitContactForm(
          @Field("name") String name,
          @Field("email") String email,
          @Field("phone") String phone,
          @Field("subject") String subject,
          @Field("message") String message
  );
  ```
  Sends a contact form submission to the server.

- **Get Contact Status:**
  ```java
  @GET("Exam/m_contact.php?action=getContactStatus")
  Call<ContactStatusResponse> getContactStatus(@Query("messageId") String messageId);
  ```
  Checks the status of a submitted message.

---

4. ApiService_Questions
This service handles operations related to university questions and resources.

Endpoints:
- **Get Universities:**
  ```java
  @GET("Exam/degrees.php?type=university")
  Call<List<University>> getUniversities();
  ```
  Retrieves a list of universities.

- **Get Degrees:**
  ```java
  @GET("Exam/degrees.php")
  Call<List<String>> getDegrees(
          @Query("type") String type,
          @Query("university_id") int universityId
  );
  ```
  Fetches degrees offered by a specific university.

- **Get Questions:**
  ```java
  @GET("m_getpdf.php")
  Call<ResponseBody> getQuestions(
          @Query("degree") String degree,
          @Query("semester") String semester,
          @Query("university_id") int universityId
  );
  ```
  Retrieves question papers for a specific degree and semester.

- **Download PDF:**
  ```java
  @GET("HOD/{filename}")
  Call<ResponseBody> downloadPdf(@Path("filename") String filename);
  ```
  Downloads a PDF file from the server.

---

5. ApiClient
The `ApiClient` class provides a centralized Retrofit instance for all API services.

Key Features:
- **Base URL:**
  ```java
  private static final String BASE_URL = "https://keralatechreach.in/";
  ```
  All API calls are made to this base URL.

- **Retrofit Initialization:**
  ```java
  public static Retrofit getClient() {
          if (retrofit == null) {
          Gson gson = new GsonBuilder()
                  .setLenient()
                  .create();

          retrofit = new Retrofit.Builder()
                  .baseUrl(BASE_URL)
                  .addConverterFactory(GsonConverterFactory.create(gson))
                  .build();
          }
          return retrofit;
  }
  ```
  Ensures a single Retrofit instance is used throughout the app.

---

6. Authentication with ApiAuthInterceptor
The `ApiAuthInterceptor` ensures secure communication by appending an API key to every request.

Key Features:
- **API Key Retrieval:**
  ```java
  String apiKey = KeystoreManager.decryptApiKey(context);
  ```
  Retrieves the encrypted API key from the keystore.

- **Query Parameter Injection:**
  ```java
  url += separator + "api_key=" + apiKey;
  ```
  Appends the API key as a query parameter to the request URL.

Conclusion
The **Kstudents** app's API layer is robust and modular, enabling efficient communication with backend services. By leveraging Retrofit, the app ensures maintainability, scalability, and security in its API interactions.

Documentation: SQLite Database in the Kstudents Android App

The **Kstudents** Android app uses SQLite as its local database to store and manage data efficiently. SQLite is a lightweight, embedded database engine that is well-suited for mobile applications. Below, we document the structure, purpose, and usage of SQLite in the app.

Purpose of SQLite in Kstudents
SQLite is used in the app to:
1. Cache data fetched from the server for offline access.
2. Store user-specific data such as preferences or analytics.
3. Manage event-related data, including categories, districts, and notes.
4. Provide a local database for faster access to frequently used data.

Database Schema
The app likely uses a structured schema to organize its data. Based on the `EventsResponse` class, the following tables are inferred:

1. Events Table
Stores information about events.

Column Name	Data Type	Description
`id`	INTEGER	Primary key for the event.
`category_id`	INTEGER	Foreign key linking to the category.
`district_id`	INTEGER	Foreign key linking to the district.
`event_start`	TEXT	Start date and time of the event.
`event_end`	TEXT	End date and time of the event.
`name`	TEXT	Name of the event.
`place`	TEXT	Location of the event.
`link`	TEXT	URL link for more details.
`map_link`	TEXT	Google Maps link for the location.
`description`	TEXT	Description of the event.
`created_at`	TEXT	Timestamp when the event was created.
`updated_date`	TEXT	Timestamp when the event was updated.
`is_sponsored`	INTEGER	Indicates if the event is sponsored.
`sponsor_name`	TEXT	Name of the sponsor.
`sponsor_logo_url`	TEXT	URL of the sponsor's logo.
`sponsor_level`	TEXT	Sponsorship level.
`sponsored_until`	TEXT	Sponsorship expiration date.

---

2. Categories Table
Stores event categories.

Column Name	Data Type	Description
`category_id`	INTEGER	Primary key for the category.
`category_name`	TEXT	Name of the category.
`created_at`	TEXT	Timestamp when the category was created.
`updated_date`	TEXT	Timestamp when the category was updated.

---

3. Districts Table
Stores district information.

Column Name	Data Type	Description
`district_id`	INTEGER	Primary key for the district.
`district_name`	TEXT	Name of the district.
`created_at`	TEXT	Timestamp when the district was created.
`updated_date`	TEXT	Timestamp when the district was updated.

---

4. Notes Table
Stores notes related to events or academics.

Column Name	Data Type	Description
`approved_id`	INTEGER	Primary key for the note.
`degree`	TEXT	Degree associated with the note.
`semester`	TEXT	Semester associated with the note.
`module`	TEXT	Module associated with the note.
`subject`	TEXT	Subject of the note.
`qtext`	TEXT	Text of the question or note.
`qyear`	TEXT	Year associated with the note.
`q_id`	INTEGER	Unique ID for the question.
`university`	INTEGER	University ID associated with the note.
`updated_date`	TEXT	Timestamp when the note was updated.

SQLite Helper Class
The app likely uses a helper class (e.g., `SQLiteOpenHelper`) to manage database creation and versioning. Below is a typical structure for such a class:

```java
public class DatabaseHelper extends SQLiteOpenHelper {
	private static final String DATABASE_NAME = "kstudents.db";
	private static final int DATABASE_VERSION = 1;
```

```
public DatabaseHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
// Create Events table
db.execSQL("CREATE TABLE events (" +
        "id INTEGER PRIMARY KEY, " +
        "category_id INTEGER, " +
        "district_id INTEGER, " +
        "event_start TEXT, " +
        "event_end TEXT, " +
        "name TEXT, " +
        "place TEXT, " +
        "link TEXT, " +
        "map_link TEXT, " +
        "description TEXT, " +
        "created_at TEXT, " +
        "updated_date TEXT, " +
        "is_sponsored INTEGER, " +
        "sponsor_name TEXT, " +
        "sponsor_logo_url TEXT, " +
        "sponsor_level TEXT, " +
        "sponsored_until TEXT)");

// Create Categories table
db.execSQL("CREATE TABLE categories (" +
        "category_id INTEGER PRIMARY KEY, " +
        "category_name TEXT, " +
        "created_at TEXT, " +
        "updated_date TEXT)");

// Create Districts table
db.execSQL("CREATE TABLE districts (" +
        "district_id INTEGER PRIMARY KEY, " +
        "district_name TEXT, " +
        "created_at TEXT, " +
        "updated_date TEXT)");

// Create Notes table
db.execSQL("CREATE TABLE notes (" +
        "approved_id INTEGER PRIMARY KEY, " +
        "degree TEXT, " +
        "semester TEXT, " +
        "module TEXT, " +
        "subject TEXT, " +
```

```
                "qtext TEXT, " +
                "qyear TEXT, " +
                "q_id INTEGER, " +
                "university INTEGER, " +
                "updated_date TEXT)");
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Handle database upgrades
        db.execSQL("DROP TABLE IF EXISTS events");
        db.execSQL("DROP TABLE IF EXISTS categories");
        db.execSQL("DROP TABLE IF EXISTS districts");
        db.execSQL("DROP TABLE IF EXISTS notes");
        onCreate(db);
        }
}
```

Data Access
The app likely uses DAO (Data Access Object) classes or raw SQL queries to interact with the database. For example:

Insert Event:
```java
public void insertEvent(Event event) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put("id", event.getId());
        values.put("name", event.getName());
        values.put("place", event.getPlace());
        // Add other fields...
        db.insert("events", null, values);
        db.close();
}
```

Fetch Events:
```java
public List<Event> getAllEvents() {
        List<Event> events = new ArrayList<>();
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor cursor = db.rawQuery("SELECT * FROM events", null);
        if (cursor.moveToFirst()) {
        do {
        Event event = new Event();
```

```
        event.setId(cursor.getInt(cursor.getColumnIndex("id")));
        event.setName(cursor.getString(cursor.getColumnIndex("name")));
        event.setPlace(cursor.getString(cursor.getColumnIndex("place")));
        // Set other fields...
        events.add(event);
        } while (cursor.moveToNext());
        }
        cursor.close();
        db.close();
        return events;
}
```

Conclusion
The SQLite database in the **Kstudents** app is a critical component for managing offline data and improving app performance. Its schema is designed to support the app's core functionalities, including event management, category and district organization, and academic notes storage. By leveraging SQLite, the app ensures a smooth user experience even in offline scenarios.

Similar code found with 1 license type

Documentation: Events Module in the Kstudents Android App

The **Events** module in the Kstudents Android app is a core feature that allows users to view, filter, and interact with event data. This module integrates with the backend API, local SQLite database, and UI components to provide a seamless experience for managing and exploring events.

Overview of the Events Module
The Events module consists of several key components:
1. **Events Activity**: The main screen for viewing events in a calendar or list format.
2. **Event List Activity**: A detailed list view of events with filtering options.
3. **Event Detail Activity**: A detailed view of a single event, including its description, location, and links.
4. **Event Adapter**: A RecyclerView adapter for displaying event data in a list format.

---

1. Events Activity (`Events.java`)
This is the main activity for the Events module. It provides a calendar view and a list of events with filtering and pagination.

Key Features:
- **Calendar Integration**: Uses `MaterialCalendarView` to display events on specific dates.
- **Filters**: Allows filtering by category, date, and district.
- **Pagination**: Loads events in chunks to improve performance.
- **Bottom Navigation**: Provides navigation to other app modules.

Important Methods:
- **`setupCalendarView()`**: Configures the calendar to display events and handle date selection.
- **`setupFilterButtons()`**: Sets up filter buttons for category, date, and district.
- **`fetchEvents()`**: Fetches events from the local database or API with the applied filters.
- **`markEventDates()`**: Highlights event dates on the calendar.

Code Snippet:
```java
private void fetchEvents() {
        isLoading = true;
        progressBar.setVisibility(View.VISIBLE);

        // Load filtered events for the list with pagination
        eventDataProcessing.loadEvents(selectedCategoryId, selectedDate,
selectedDistrictId, currentPage, ITEMS_PER_PAGE);

        // Load all events for the calendar
        eventDataProcessing.loadAllEventsForCalendar(this);
```

}
```

---

### **2. Event List Activity (`EventListActivity.java`)**
This activity provides a detailed list view of events with advanced filtering options.

#### **Key Features:**
- **RecyclerView Integration**: Displays events in a scrollable list.
- **Filter Chips**: Allows users to apply and clear filters dynamically.
- **Pagination**: Supports infinite scrolling to load more events.

#### **Important Methods:**
- **`setupRecyclerView()`**: Configures the RecyclerView for displaying events.
- **`loadEvents()`**: Loads events based on the current filters and pagination state.
- **`applyFilters()`**: Applies selected filters and reloads the event list.

#### **Code Snippet:**
```java
private void loadEvents() {
        isLoading = true;
        currentPage = 0;
        hasMoreEvents = true;
        eventsList.clear();
        eventAdapter.notifyDataSetChanged();
        dataProcessing.loadEvents(selectedCategoryId, selectedDate, selectedDistrictId,
currentPage, ITEMS_PER_PAGE);
}
```

---

3. Event Detail Activity (`EventDetail2Activity.java`)
This activity provides a detailed view of a single event, including its description, location, and links.

Key Features:
- **Event Details**: Displays event name, date, location, description, and links.
- **Calendar Integration**: Highlights the event's date range on a calendar.
- **Map and Website Links**: Allows users to open the event location in Google Maps or visit the event's website.
- **Sharing**: Enables users to share event details with others.

Important Methods:
- **`fetchEventDetailsFromDatabase()`**: Fetches event details from the local SQLite database.
- **`openLocationInMap()`**: Opens the event location in Google Maps.
```

- **`openEventWebsite()`**: Opens the event's website in a browser.

Code Snippet:
```java
private void openLocationInMap() {
        if (eventLocation != null && !eventLocation.isEmpty()) {
        Uri gmmIntentUri = Uri.parse("geo:0,0?q=" + Uri.encode(eventLocation));
        Intent mapIntent = new Intent(Intent.ACTION_VIEW, gmmIntentUri);
        mapIntent.setPackage("com.google.android.apps.maps");

        if (mapIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(mapIntent);
        } else {
        // Open in browser if Google Maps is not installed
        Uri locationUrl = Uri.parse("https://www.google.com/maps/search/?api=1&query=" +
Uri.encode(eventLocation));
        Intent browserIntent = new Intent(Intent.ACTION_VIEW, locationUrl);
        startActivity(browserIntent);
        }
        }
}
```

---

4. Event Adapter (`EventAdapter.java`)
The `EventAdapter` is a RecyclerView adapter that binds event data to the UI components in the event list.

Key Features:
- **Dynamic Data Binding**: Binds event data to the list item views.
- **Sponsored Events**: Highlights sponsored events with a special background and badge.
- **Click Listener**: Handles click events to navigate to the event detail screen.

Important Methods:
- **`onBindViewHolder()`**: Binds event data to the list item views.
- **`updateEvents()`**: Updates the event list when new data is loaded.

Code Snippet:
```java
@Override
public void onBindViewHolder(@NonNull EventViewHolder holder, int position) {
        EventDataProcessing.EventDate event = events.get(position);

        holder.tvEventName.setText(event.getName());
        holder.tvEventDate.setText(formatEventDate(event));
        holder.tvEventDistrict.setText(event.getDistrictName());
```

```
        if (event.isSponsored()) {

        holder.itemView.setBackgroundResource(R.drawable.sponsored_event_background
);
        holder.sponsorBadge.setVisibility(View.VISIBLE);
        holder.sponsorName.setText(event.getSponsorName());
        } else {
        holder.itemView.setBackgroundResource(R.drawable.regular_event_background);
        holder.sponsorBadge.setVisibility(View.GONE);
        }

        holder.itemView.setOnClickListener(v -> clickListener.onEventClick(event.getId()));
}
```

---

5. Event Data Processing
The `EventDataProcessing` class is responsible for fetching and processing event data from the local SQLite database or API.

Key Features:
- **Data Loading**: Loads events based on filters and pagination.
- **Filter Dialogs**: Provides dialogs for selecting categories, dates, and districts.
- **Calendar Integration**: Prepares data for marking event dates on the calendar.

---

6. Analytics Integration
The Events module integrates with the `EventAnalyticsService` to track user interactions and impressions. This helps in understanding user behavior and improving the app.

Tracked Events:
- **Calendar Month Change**: Tracks when users navigate between months in the calendar.
- **Event Clicks**: Tracks when users click on an event in the list.
- **Map and Website Interactions**: Tracks when users open the event location in Google Maps or visit the event's website.

Conclusion
The Events module in the Kstudents app is a comprehensive feature that combines calendar views, list views, and detailed event information. It provides a user-friendly interface for exploring events while leveraging advanced features like filtering, pagination, and analytics tracking. This module is a critical part of the app, enhancing user engagement and accessibility.

Similar code found with 1 license type

Documentation: Initiatives Module in the Kstudents Android App

The **Initiatives** module in the Kstudents Android app is designed to display and manage government initiatives. It provides users with detailed information about various initiatives, including descriptions, eligibility criteria, and links to official websites. This module integrates with the backend API and uses a RecyclerView to display the initiatives in a user-friendly format.

Overview of the Initiatives Module
The Initiatives module consists of the following components:
1. **Initiatives Activity**: The main screen for displaying a list of government initiatives.
2. **Initiatives Adapter**: A RecyclerView adapter for binding initiative data to the UI.
3. **Govt_ApiService**: A Retrofit-based API service for fetching initiatives from the backend.
4. **ApiResponseInitiatives**: A generic class for handling API responses.

---

1. Initiatives Activity (`InitiativesActivity.java`)
This is the main activity for the Initiatives module. It fetches data from the backend API and displays it in a RecyclerView.

Key Features:
- **Retrofit Integration**: Uses the `Govt_ApiService` to fetch initiatives from the backend.
- **RecyclerView**: Displays initiatives in a scrollable list.
- **Bottom Navigation**: Allows navigation to other app modules.

Important Methods:
- **`setupRetrofit()`**: Initializes the Retrofit instance for API calls.
- **`loadInitiatives()`**: Fetches initiatives from the backend and updates the RecyclerView.
- **`setupBottomNavigation()`**: Configures the bottom navigation for seamless navigation between modules.

Code Snippet:
```java
private void loadInitiatives() {
        Call<ApiResponseInitiatives<List<Initiative>>> call = apiService.getInitiatives();
        call.enqueue(new Callback<ApiResponseInitiatives<List<Initiative>>>() {
        @Override
        public void onResponse(Call<ApiResponseInitiatives<List<Initiative>>> call,
                               Response<ApiResponseInitiatives<List<Initiative>>> response) {
        if (response.isSuccessful()) {
                ApiResponseInitiatives<List<Initiative>> apiResponse = response.body();
                if (apiResponse != null && "success".equals(apiResponse.getStatus())) {
                initiatives.clear();
```

```
                    if (apiResponse.getData() != null) {
                    initiatives.addAll(apiResponse.getData());
                    }
                    adapter.notifyDataSetChanged();
                    } else {
                    String error = apiResponse != null ? apiResponse.getMessage() : "Unknown
error";
                    Toast.makeText(InitiativesActivity.this, error,
Toast.LENGTH_SHORT).show();
                    }
            } else {
                    Toast.makeText(InitiativesActivity.this,
                    "Error: " + response.code(), Toast.LENGTH_SHORT).show();
            }
            }

            @Override
            public void onFailure(Call<ApiResponseInitiatives<List<Initiative>>> call, Throwable
t) {
            Toast.makeText(InitiativesActivity.this,
                    "Error: " + t.getMessage(), Toast.LENGTH_SHORT).show();
            }
            });
}
```

---

2. Initiatives Adapter (`InitiativesAdapter.java`)
The `InitiativesAdapter` is a RecyclerView adapter that binds initiative data to the UI components in the list.

Key Features:
- **Dynamic Data Binding**: Binds initiative data (name, description, eligibility, website) to the UI.
- **Dialog Display**: Shows detailed descriptions and eligibility criteria in a dialog.
- **Website Links**: Opens initiative websites in a browser.

Important Methods:
- **`onBindViewHolder()`**: Binds initiative data to the list item views.
- **`showDialog()`**: Displays a dialog with detailed information about the initiative.

Code Snippet:
```java
@Override
public void onBindViewHolder(ViewHolder holder, int position) {
        Initiative initiative = initiatives.get(position);
        holder.nameButton.setText(initiative.getName());
```

```
        holder.descriptionButton.setText("Description");
        holder.eligibilityButton.setText("Eligibility");
        holder.websiteButton.setText("Website");

        holder.descriptionButton.setOnClickListener(v -> {
        showDialog("Description", initiative.getDescription());
        });

        holder.eligibilityButton.setOnClickListener(v -> {
        showDialog("Eligibility", initiative.getEligibility());
        });

        holder.websiteButton.setOnClickListener(v -> {
        String url = initiative.getWebsiteUrl();
        if (url != null && !url.isEmpty()) {
        if (!url.startsWith("http://") && !url.startsWith("https://")) {
                url = "https://" + url;
        }
        try {
                Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
                context.startActivity(intent);
        } catch (Exception e) {
                Toast.makeText(context, "Cannot open this website",
Toast.LENGTH_SHORT).show();
        }
        } else {
        Toast.makeText(context, "No website available", Toast.LENGTH_SHORT).show();
        }
        });
}
```

---

3. Govt_ApiService (`Govt_ApiService.java`)
The `Govt_ApiService` is a Retrofit-based API service for fetching government initiatives from the backend.

Key Features:
- **API Endpoint**: Fetches initiatives from the `m_get_initiatives.php` endpoint.
- **Authentication**: Uses `ApiAuthInterceptor` to add authentication headers to API requests.

API Endpoint:
```java
@GET("Exam/m_get_initiatives.php")
Call<ApiResponseInitiatives<List<Initiative>>> getInitiatives();
```

Factory Class:
The `Factory` class provides a singleton instance of the `Govt_ApiService`.

```java
public static Govt_ApiService getInstance(Context context, String baseUrl) {
	if (service == null) {
	OkHttpClient client = new OkHttpClient.Builder()
		.addInterceptor(new ApiAuthInterceptor(context))
		.build();

	Retrofit retrofit = new Retrofit.Builder()
		.baseUrl(baseUrl)
		.client(client)
		.addConverterFactory(GsonConverterFactory.create())
		.build();

	service = retrofit.create(Govt_ApiService.class);
	}
	return service;
}
```

---

4. ApiResponseInitiatives (`ApiResponseInitiatives.java`)
This is a generic class for handling API responses.

Key Features:
- **Status**: Indicates the success or failure of the API call.
- **Data**: Contains the list of initiatives fetched from the backend.
- **Message**: Provides additional information about the API response.

Code Snippet:
```java
public class ApiResponseInitiatives<T> {
	private String status;
	private T data;
	private String message;

	public String getStatus() { return status; }
	public void setStatus(String status) { this.status = status; }
	public T getData() { return data; }
	public void setData(T data) { this.data = data; }
	public String getMessage() { return message; }
	public void setMessage(String message) { this.message = message; }
}
```

---

5. Initiative Class
The `Initiative` class represents a single government initiative. It contains fields such as:
- **Name**: The name of the initiative.
- **Description**: A detailed description of the initiative.
- **Eligibility**: Eligibility criteria for the initiative.
- **Website URL**: A link to the official website for the initiative.

---

6. User Interaction
- **View Details**: Users can view detailed descriptions and eligibility criteria by clicking the respective buttons.
- **Visit Website**: Users can open the initiative's website in a browser.
- **Error Handling**: Displays appropriate error messages if the API call fails or the website URL is invalid.

Conclusion
The Initiatives module in the Kstudents app provides a comprehensive interface for exploring government initiatives. By integrating Retrofit for API calls, RecyclerView for displaying data, and intuitive UI components, this module ensures a seamless user experience. It is a valuable feature for users seeking information about government programs and initiatives.

Similar code found with 3 license types

Documentation: Jobs Module in the Kstudents Android App

The **Jobs** module in the Kstudents Android app is designed to provide users with job-related information, including job listings, detailed job descriptions, and application links. It integrates with a backend API to fetch job data and displays it in a user-friendly interface using RecyclerView and other Android UI components.

Overview of the Jobs Module
The Jobs module consists of the following components:
1. **Job View Activity (`Job_View.java`)**: Displays a list of available jobs.
2. **Job Details Activity (`JobDetailsActivity.java`)**: Provides detailed information about a specific job.
3. **Job Adapter (`JobAdapter.java`)**: A RecyclerView adapter for binding job data to the UI.
4. **Job Model (`Job.java`)**: Represents the structure of a job object.
5. **API Client (`ApiClient.java`)**: Handles API calls to fetch job data from the backend.

---

1. Job View Activity (`Job_View.java`)
This activity displays a list of jobs fetched from the backend API. It uses a RecyclerView to show job details such as the post name, company name, and application deadline.

Key Features:
- **RecyclerView Integration**: Displays jobs in a scrollable list.
- **Swipe-to-Refresh**: Allows users to refresh the job list manually.
- **Empty State View**: Displays a message when no jobs are available.
- **Bottom Navigation**: Provides navigation to other app modules.

Important Methods:
- **`loadJobs()`**: Fetches the list of jobs from the backend API.
- **`onJobClick()`**: Handles click events on a job item and navigates to the Job Details screen.

Code Snippet:
```java
private void loadJobs() {
        progressBar.setVisibility(View.VISIBLE);
        emptyStateView.setVisibility(View.GONE);

        ApiClient.getInstance(this).getJobs(new ApiClient.JobsResponseListener() {
        @Override
        public void onResponse(List<Job> jobs) {
        progressBar.setVisibility(View.GONE);
```

```
        swipeRefreshLayout.setRefreshing(false);

        if (jobs.isEmpty()) {
                emptyStateView.setVisibility(View.VISIBLE);
        } else {
                emptyStateView.setVisibility(View.GONE);
        }

        jobList.clear();
        jobList.addAll(jobs);
        jobAdapter.notifyDataSetChanged();
        }

        @Override
        public void onError(String message) {
        progressBar.setVisibility(View.GONE);
        swipeRefreshLayout.setRefreshing(false);
        emptyStateView.setVisibility(View.VISIBLE);
        Toast.makeText(Job_View.this, "Error: " + message,
Toast.LENGTH_SHORT).show();
        }
        });
}
```

---

2. Job Details Activity (`JobDetailsActivity.java`)
This activity provides detailed information about a specific job, including the post name, company name, description, qualifications, and application deadline.

Key Features:
- **Deep Linking**: Supports deep links to open specific job details directly.
- **Website Integration**: Opens the job's application website in a browser.
- **Sharing**: Allows users to share job details with others.
- **Bottom Navigation**: Provides navigation to other app modules.

Important Methods:
- **`loadJobDetails()`**: Fetches detailed information about a specific job from the backend API.
- **`websiteClicked()`**: Opens the job's application website in a browser.
- **`setupShareButton()`**: Configures the share button to share job details.

Code Snippet:
```java
private void loadJobDetails(int jobId) {
        progressBar.setVisibility(View.VISIBLE);
```

```
        ApiClient.getInstance(this).getJobDetails(jobId, new ApiClient.JobDetailsListener() {
        @Override
        public void onResponse(Job job) {
        progressBar.setVisibility(View.GONE);
        currentJob = job;

        // Set job details
        textPostName.setText(job.getPostName());
        textCompanyName.setText(job.getCompanyName());

        // Format deadline date
        try {
                SimpleDateFormat serverFormat = new SimpleDateFormat("yyyy-MM-dd",
Locale.getDefault());
                SimpleDateFormat displayFormat = new SimpleDateFormat("MMMM dd,
yyyy", Locale.getDefault());
                Date date = serverFormat.parse(job.getDeadline());
                textDeadline.setText("Application Deadline: " + displayFormat.format(date));
        } catch (ParseException e) {
                textDeadline.setText("Application Deadline: " + job.getDeadline());
        }

        textDescription.setText(job.getDescription());
        textQualifications.setText(job.getQualifications());
        textEmail.setText(job.getEmail());
        textPhone.setText(job.getPhone());
        }

        @Override
        public void onError(String message) {
        progressBar.setVisibility(View.GONE);
        Toast.makeText(JobDetailsActivity.this, "Error: " + message,
Toast.LENGTH_SHORT).show();
        finish();
        }
        });
}
```

---

3. Job Adapter (`JobAdapter.java`)
The `JobAdapter` is a RecyclerView adapter that binds job data to the UI components in the job list.

Key Features:
- **Dynamic Data Binding**: Binds job data (post name, company name, deadline) to the UI.
- **Click Listener**: Handles click events to navigate to the Job Details screen.

Important Methods:
- **`onBindViewHolder()`**: Binds job data to the list item views.
- **`formatDate()`**: Formats the job deadline date for display.

Code Snippet:
```java
@Override
public void onBindViewHolder(@NonNull JobViewHolder holder, int position) {
        Job job = jobList.get(position);

        holder.textPostName.setText(job.getPostName());
        holder.textCompanyName.setText(job.getCompanyName());

        // Format deadline date
        String deadlineText = "Deadline: " + formatDate(job.getDeadline());
        holder.textDeadline.setText(deadlineText);

        // Set click listeners
        holder.cardJob.setOnClickListener(v -> listener.onJobClick(job));
}
```

---

4. Job Model (`Job.java`)
The `Job` class represents the structure of a job object. It includes fields such as:
- **ID**: Unique identifier for the job.
- **Post Name**: The name of the job post.
- **Company Name**: The name of the company offering the job.
- **Deadline**: The application deadline for the job.
- **Description**: A detailed description of the job.
- **Qualifications**: The required qualifications for the job.
- **Email**: Contact email for the job.
- **Phone**: Contact phone number for the job.
- **Website**: The application website for the job.

Code Snippet:
```java
public class Job implements Serializable {
        private int id;
        @SerializedName("post_name")
        private String postName;

        @SerializedName("company_name")
        private String companyName;
        private String deadline;
        private String description;
```

```
    private String qualifications;
    private String email;
    private String phone;
    private String website;

    // Getters and setters
    public int getId() { return id; }
    public String getPostName() { return postName; }
    public String getCompanyName() { return companyName; }
    public String getDeadline() { return deadline; }
    public String getDescription() { return description; }
    public String getQualifications() { return qualifications; }
    public String getEmail() { return email; }
    public String getPhone() { return phone; }
    public String getWebsite() { return website; }
}
```

---

5. API Client (`ApiClient.java`)
The `ApiClient` class handles API calls to fetch job data from the backend.

Key Features:
- **Job List Endpoint**: Fetches a list of jobs from the `get_job.php` endpoint.
- **Job Details Endpoint**: Fetches detailed information about a specific job from the `get_job_details.php` endpoint.
- **Authentication**: Uses `ApiAuthInterceptor` to add authentication headers to API requests.

Code Snippet:
```java
public void getJobs(final JobsResponseListener listener) {
    apiService.getJobs().enqueue(new Callback<JobResponse>() {
    @Override
    public void onResponse(Call<JobResponse> call, Response<JobResponse>
response) {
    if (response.isSuccessful() && response.body() != null) {
        JobResponse jobResponse = response.body();
        if (jobResponse.isSuccess()) {
        listener.onResponse(jobResponse.getJobs());
        } else {
        listener.onError(jobResponse.getMessage());
        }
    } else {
        listener.onError("Server error: " + response.code());
    }
    }
```

```
        @Override
        public void onFailure(Call<JobResponse> call, Throwable t) {
        listener.onError("Network error: " + t.getMessage());
        }
        });
}
```

Conclusion
The Jobs module in the Kstudents app provides a comprehensive interface for exploring job opportunities. By integrating Retrofit for API calls, RecyclerView for displaying data, and intuitive UI components, this module ensures a seamless user experience. It is a valuable feature for users seeking job-related information and application links.

Documentation: Message Module in the Kstudents Android App

The **Message** module in the Kstudents Android app allows users to send messages or inquiries to the app's administrators or support team. This module provides a simple form for users to input their details and message, which is then submitted to the backend server using Retrofit.

Overview of the Message Module
The Message module consists of the following components:
1. **Messageus Activity (`Messageus.java`)**: The main activity for submitting messages.
2. **Contact Response (`ContactResponse.java`)**: Represents the response from the server after submitting a message.
3. **Contact Status Response (`ContactStatusResponse.java`)**: Represents the status of a previously submitted message.

---

1. Messageus Activity (`Messageus.java`)
This activity provides a form for users to input their name, email, phone number, subject, and message. It validates the input and submits the data to the backend server.

Key Features:
- **Form Validation**: Ensures that all required fields are filled before submission.
- **Progress Dialog**: Displays a loading indicator while the message is being sent.
- **API Integration**: Uses Retrofit to send the message to the backend server.
- **Success Dialog**: Displays a confirmation dialog when the message is successfully sent.
- **Error Handling**: Handles network errors and server-side errors gracefully.

Important Methods:
- **`submitContactForm()`**: Validates the form inputs and sends the message to the server.
- **`showSuccessDialog()`**: Displays a dialog to confirm that the message was sent successfully.
- **`clearFormFields()`**: Clears the form fields after a successful submission.

Code Snippet:
```java
private void submitContactForm() {
        String name = nameEditText.getText().toString().trim();
        String email = emailEditText.getText().toString().trim();
        String phone = phoneEditText.getText().toString().trim();
        String subject = subjectEditText.getText().toString().trim();
        String message = messageEditText.getText().toString().trim();

        // Validate inputs
```

```
        if (name.isEmpty() || email.isEmpty() || subject.isEmpty() || message.isEmpty()) {
        Toast.makeText(this, "Please fill all required fields",
Toast.LENGTH_SHORT).show();
        return;
        }

        ProgressDialog progressDialog = new ProgressDialog(this);
        progressDialog.setMessage("Sending message...");
        progressDialog.setCancelable(false);
        progressDialog.show();

        String baseUrl = getString(R.string.api_base_url);
        ContactApiService apiService = ContactApiService.Factory.getInstance(this,
baseUrl);

        Call<ContactResponse> call = apiService.submitContactForm(name, email, phone,
subject, message);
        call.enqueue(new Callback<ContactResponse>() {
        @Override
        public void onResponse(Call<ContactResponse> call, Response<ContactResponse>
response) {
        progressDialog.dismiss();
        if (response.isSuccessful() && response.body() != null) {
                ContactResponse contactResponse = response.body();
                if (contactResponse.isSuccess()) {
                showSuccessDialog(contactResponse.getMessage());
                clearFormFields();
                } else {
                Toast.makeText(Messageus.this, contactResponse.getMessage(),
Toast.LENGTH_LONG).show();
                }
        } else {
                Toast.makeText(Messageus.this, "Failed to send message. Please try
again.", Toast.LENGTH_LONG).show();
        }
        }

        @Override
        public void onFailure(Call<ContactResponse> call, Throwable t) {
        progressDialog.dismiss();
        Toast.makeText(Messageus.this, "Network error. Please check your connection.",
Toast.LENGTH_LONG).show();
        }
        });
}
```

---

2. Contact Response (`ContactResponse.java`)
The `ContactResponse` class represents the response from the server after a message is submitted. It contains the following fields:
- **`success`**: A boolean indicating whether the message was sent successfully.
- **`message`**: A string containing a success or error message.
- **`messageId`**: A unique ID for the submitted message.

Code Snippet:
```java
public class ContactResponse {
	@SerializedName("success")
	private boolean success;

	@SerializedName("message")
	private String message;

	@SerializedName("messageId")
	private String messageId;

	public boolean isSuccess() {
	return success;
	}

	public String getMessage() {
	return message;
	}

	public String getMessageId() {
	return messageId;
	}
}
```

---

3. Contact Status Response (`ContactStatusResponse.java`)
The `ContactStatusResponse` class represents the status of a previously submitted message. It contains the following fields:
- **`success`**: A boolean indicating whether the status retrieval was successful.
- **`status`**: A string representing the current status of the message.
- **`message`**: A string containing additional information about the status.

Code Snippet:
```java
public class ContactStatusResponse {
	@SerializedName("success")
	private boolean success;
```

```
    @SerializedName("status")
    private String status;

    @SerializedName("message")
    private String message;

    public boolean isSuccess() {
    return success;
    }

    public String getStatus() {
    return status;
    }

    public String getMessage() {
    return message;
    }
}
```

---

4. User Interaction
- **Form Submission**: Users fill out the form and click the "Submit" button to send their message.
- **Validation**: If any required fields are empty, the app displays a toast message prompting the user to fill them.
- **Success Feedback**: If the message is sent successfully, a dialog is displayed with a success message.
- **Error Handling**: If the message fails to send, the app displays an appropriate error message.

---

5. API Integration
The `ContactApiService` is used to send the message to the backend server. The API endpoint for submitting the message is:
```java
@FormUrlEncoded
@POST("Exam/m_contact.php?action=submitContactForm")
Call<ContactResponse> submitContactForm(
    @Field("name") String name,
    @Field("email") String email,
    @Field("phone") String phone,
    @Field("subject") String subject,
    @Field("message") String message
);
```

```

---

### **6. Error Handling**
- **Validation Errors**: If required fields are empty, the app displays a toast message.
- **Network Errors**: If there is a network issue, the app displays a toast message indicating a connection problem.
- **Server Errors**: If the server returns an error, the app displays the error message from the server.

---

### **Conclusion**
The Message module in the Kstudents app provides a simple and effective way for users to communicate with the app's administrators or support team. By integrating Retrofit for API calls, form validation, and user-friendly feedback mechanisms, this module ensures a smooth and reliable user experience.
```

Documentation: Analytics Module in the Kstudents Android App

The **Analytics** module in the Kstudents Android app is designed to track, store, and report user interactions and impressions related to events. It provides insights into user engagement, event performance, and sponsor analytics. The module integrates with both local SQLite storage and backend APIs to ensure data is captured and synchronized effectively.

Overview of the Analytics Module
The Analytics module consists of the following components:
1. **Event Analytics Service (`EventAnalyticsService.java`)**: Tracks and stores analytics events locally.
2. **Analytics Sync Service (`AnalyticsSyncService.java`)**: Synchronizes local analytics data with the backend server.
3. **Event Analytics Reporter (`EventAnalyticsReporter.java`)**: Generates detailed reports on event performance.
4. **Analytics Request (`AnalyticsRequest.java`)**: Represents the structure of analytics data sent to the server.
5. **Analytics Submission Response (`AnalyticsSubmissionResponse.java`)**: Represents the server's response after submitting analytics data.
6. **Sponsor Report Response (`SponsorReportResponse.java`)**: Provides analytics data for sponsors.
7. **Overall Stats Response (`OverallStatsResponse.java`)**: Provides overall analytics statistics.
8. **Event Analytics Report (`EventAnalyticsReport.java`)**: Provides detailed analytics for a specific event.

---

1. Event Analytics Service (`EventAnalyticsService.java`)
The `EventAnalyticsService` is responsible for tracking and storing analytics events such as impressions and interactions. These events are stored locally in SQLite until they are synchronized with the backend.

Key Features:
- **Impression Tracking**: Tracks when an event is viewed by a user.
- **Interaction Tracking**: Tracks user interactions with events (e.g., clicks, shares).
- **Batch Processing**: Stores events in batches and processes them when a threshold is reached.
- **Install ID**: Generates a unique ID for each app installation to track user-specific data.

Important Methods:
- **`trackImpression(int eventId, String source)`**: Tracks an impression for a specific event.

- **`trackInteraction(int eventId, String interactionType)`**: Tracks a user interaction with an event.
- **`flushEvents()`**: Forces the processing and saving of pending events to the database.

Code Snippet:
```java
public void trackImpression(int eventId, String source) {
        if (eventId <= 0) return; // Skip invalid events

        AnalyticsEvent event = new AnalyticsEvent(
        eventId,
        "impression",
        source,
        getCurrentTimestamp(),
        installId
        );
        queueAnalyticsEvent(event);
        Log.d(TAG, "Tracked impression for event: " + eventId + " from source: " + source);
}
```

---

2. Analytics Sync Service (`AnalyticsSyncService.java`)
The `AnalyticsSyncService` is a background service that periodically synchronizes local analytics data with the backend server.

Key Features:
- **Job Scheduling**: Uses Android's `JobScheduler` to schedule periodic syncs.
- **Batch Processing**: Sends analytics data to the server in batches.
- **Network Awareness**: Ensures syncs only occur when the device is connected to a network.
- **Error Handling**: Reschedules jobs if a sync fails.

Important Methods:
- **`scheduleJob(Context context)`**: Schedules the analytics sync job.
- **`syncAnalytics(JobParameters params)`**: Synchronizes unsynced analytics events with the server.
- **`markEventsAsSynced()`**: Marks events as synced in the local database after a successful sync.

Code Snippet:
```java
public static void scheduleJob(Context context) {
        ComponentName serviceComponent = new ComponentName(context, AnalyticsSyncService.class);
        JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceComponent);
```

```
        builder.setMinimumLatency(SYNC_INTERVAL) // Wait at least 6 hours between
syncs
        .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY) // Run on any network
        .setPersisted(true) // Survive reboots
        .setBackoffCriteria(30 * 60 * 1000, JobInfo.BACKOFF_POLICY_LINEAR); // 30 min
backoff

        JobScheduler jobScheduler = (JobScheduler)
context.getSystemService(Context.JOB_SCHEDULER_SERVICE);
        if (jobScheduler != null) {
        jobScheduler.schedule(builder.build());
        Log.d(TAG, "Analytics sync job scheduled");
        }
}
```

---

3. Event Analytics Reporter (`EventAnalyticsReporter.java`)
The `EventAnalyticsReporter` generates detailed reports on event performance, including impressions, interactions, and sponsor details.

Key Features:
- **Impression Reporting**: Counts total and unique impressions for an event.
- **Interaction Reporting**: Counts total and unique interactions for an event.
- **Sponsor Details**: Includes sponsor information if the event is sponsored.

Important Methods:
- **`generateEventReport(int eventId, AnalyticsReportCallback callback)`**: Generates a detailed report for a specific event.
- **`addImpressionSource(String source, int count, int uniqueCount)`**: Adds impression data to the report.
- **`addInteractionType(String type, int count, int uniqueCount)`**: Adds interaction data to the report.

Code Snippet:
```java
public void generateEventReport(int eventId, AnalyticsReportCallback callback) {
        new AsyncTask<Integer, Void, EventAnalyticsReport>() {
        @Override
        protected EventAnalyticsReport doInBackground(Integer... params) {
        int eventId = params[0];
        SQLiteDatabase db = dbHelper.getReadableDatabase();
        EventAnalyticsReport report = new EventAnalyticsReport(eventId);

        // Count impressions
        String impressionQuery = "SELECT COUNT(*) as count, COUNT(DISTINCT
install_id) as unique_count, source " +
```

```
                "FROM event_impressions WHERE event_id = ? GROUP BY source";
        Cursor impressionCursor = db.rawQuery(impressionQuery, new
String[]{String.valueOf(eventId)});
        while (impressionCursor.moveToNext()) {
                int count =
impressionCursor.getInt(impressionCursor.getColumnIndex("count"));
                int uniqueCount =
impressionCursor.getInt(impressionCursor.getColumnIndex("unique_count"));
                String source =
impressionCursor.getString(impressionCursor.getColumnIndex("source"));
                report.addImpressionSource(source, count, uniqueCount);
        }
        impressionCursor.close();

        return report;
        }

        @Override
        protected void onPostExecute(EventAnalyticsReport report) {
        if (report != null) {
                callback.onReportGenerated(report);
        } else {
                callback.onReportFailed("Failed to generate report");
        }
        }
        }.execute(eventId);
}
```

---

4. Analytics Request (`AnalyticsRequest.java`)
The `AnalyticsRequest` class represents the structure of analytics data sent to the server. It includes:
- **Install ID**: A unique identifier for the app installation.
- **Events**: A list of analytics events (impressions or interactions).

Code Snippet:
```java
public class AnalyticsRequest {
        private String install_id;
        private List<AnalyticsEvent> events;

        public AnalyticsRequest(String installId, List<AnalyticsEvent> events) {
        this.install_id = installId;
        this.events = events;
        }
```

```
        public static class AnalyticsEvent {
        private int event_id;
        private String event_type;
        private String timestamp;
        private String source;

        public AnalyticsEvent(int eventId, String eventType, String timestamp, String source)
{
        this.event_id = eventId;
        this.event_type = eventType;
        this.timestamp = timestamp;
        this.source = source;
        }
        }
}
```

---

5. Analytics Submission Response (`AnalyticsSubmissionResponse.java`)
The `AnalyticsSubmissionResponse` class represents the server's response after submitting analytics data. It includes:
- **Success**: Indicates whether the submission was successful.
- **Processed**: The number of events processed by the server.
- **Successful**: The number of events successfully synced.
- **Failed**: The number of events that failed to sync.

---

6. Sponsor Report Response (`SponsorReportResponse.java`)
The `SponsorReportResponse` provides analytics data for sponsors, including:
- **Sponsor Info**: Details about the sponsor (name, logo, website).
- **Event Reports**: Analytics data for events sponsored by the sponsor.
- **Summary**: Overall statistics for the sponsor's events.

---

7. Overall Stats Response (`OverallStatsResponse.java`)
The `OverallStatsResponse` provides aggregated analytics statistics, including:
- **Event Stats**: Total events, sponsored events, upcoming events.
- **Impression Stats**: Total impressions, unique users, events with impressions.
- **Interaction Stats**: Total interactions, unique users, events with interactions.
- **Engagement Rate**: The ratio of interactions to impressions.

---

8. Event Analytics Report (`EventAnalyticsReport.java`)
The `EventAnalyticsReport` provides detailed analytics for a specific event, including:

- **Event Info**: Details about the event (name, description, sponsor info).
- **Impressions**: Total and unique impressions, grouped by source.
- **Interactions**: Total and unique interactions, grouped by type.
- **Summary**: Aggregated statistics for the event.

Conclusion
The Analytics module in the Kstudents app is a robust system for tracking, storing, and reporting user engagement data. By integrating local storage, background synchronization, and detailed reporting, this module provides valuable insights into event performance and user behavior. It is a critical component for improving user experience and optimizing event management.

Documentation: Notes Module in the Kstudents Android App

The **Notes** module in the Kstudents Android app provides users with a platform to browse, view, upload, save, and manage academic notes. It integrates with the backend API, local SQLite database, and file storage to ensure seamless functionality for both online and offline use.

Overview of the Notes Module
The Notes module consists of the following components:
1. **Notes Activity (`Notes.java`)**: The main screen for browsing and searching notes.
2. **Notes View Activity (`NotesView.java`)**: Displays a filtered list of notes based on user selection.
3. **View PDF Activity (`ViewPdf.java`)**: Allows users to view a specific note in a PDF viewer.
4. **Upload Note Activity (`UploadNoteActivity.java`)**: Enables users to upload new notes.
5. **Saved Notes Activity (`SavedNotesActivity.java`)**: Displays notes saved locally for offline access.
6. **Saved Notes Manager (`SavedNotesManager.java`)**: Handles saving and managing notes locally.
7. **Notes Adapter (`NotesAdapter.java`)**: A RecyclerView adapter for displaying notes in a list.
8. **Recent Notes Adapter (`RecentNotesAdapter.java`)**: Displays recently accessed notes.
9. **Notes Data Processing (`NotesDataProcessing.java`)**: Handles data fetching, filtering, and processing.

---

1. Notes Activity (`Notes.java`)
This is the main entry point for the Notes module. It allows users to search for notes by selecting a university, degree, and semester.

Key Features:
- **Dropdown Filters**: Users can select a university, degree, and semester to filter notes.
- **Recent Notes**: Displays a list of recently accessed notes.
- **Navigation**: Provides access to other app modules via bottom navigation.

Important Methods:
- **`fetchUniversities()`**: Fetches the list of universities from the backend.
- **`fetchDegrees(int universityId)`**: Fetches the list of degrees for a selected university.
- **`notesSubmit(View view)`**: Navigates to the `NotesView` activity with the selected filters.

Code Snippet:

```java
public void notesSubmit(View view) {
        String university = universitySpinner.getText().toString();
        String degree = degSpinner.getText().toString();
        String semester = semSpinner.getText().toString();

        if (university.isEmpty() || degree.isEmpty() || semester.isEmpty()) {
        Toast.makeText(this, "Please select all fields", Toast.LENGTH_SHORT).show();
        return;
        }

        Integer universityId = universityIdMap.get(university);
        if (universityId == null) {
        Toast.makeText(this, "Error: University ID not found",
Toast.LENGTH_SHORT).show();
        return;
        }

        Intent intent = new Intent(this, NotesView.class);
        intent.putExtra("university_name", university);
        intent.putExtra("university_id", universityId);
        intent.putExtra("deg", degree);
        intent.putExtra("sem", semester);
        startActivity(intent);
}
```

---

2. Notes View Activity (`NotesView.java`)
This activity displays a list of notes based on the selected university, degree, and semester. It also allows filtering by module.

Key Features:
- **Module Filters**: Users can filter notes by module using chips.
- **RecyclerView Integration**: Displays notes in a scrollable list.
- **Deep Linking**: Supports deep links to open specific notes directly.

Important Methods:
- **`loadNotes(String moduleFilter)`**: Loads notes based on the selected filters.
- **`loadModuleFilters(String degree, String semester)`**: Loads available modules for filtering.

Code Snippet:
```java
private void loadModuleFilters(String degree, String semester) {
        moduleFilterChips.removeAllViews();
```

```
        Chip allChip = new Chip(this);
        allChip.setText("All Modules");
        allChip.setCheckable(true);
        allChip.setChecked(true);
        allChip.setOnClickListener(v -> {
        mModule = null;
        loadNotes(null);
        });
        moduleFilterChips.addView(allChip);

        notesDataProcessing.getModules(degree, semester, modules -> {
        for (String module : modules) {
        Chip chip = new Chip(this);
        chip.setText(module);
        chip.setCheckable(true);
        chip.setOnClickListener(v -> {
                mModule = module;
                loadNotes(module);
        });
        moduleFilterChips.addView(chip);
        }
        });
}
```

---

3. View PDF Activity (`ViewPdf.java`)
This activity allows users to view a specific note in a PDF viewer.

Key Features:
- **PDF Viewer**: Uses a `WebView` to display PDFs via Google Docs Viewer or direct links.
- **Error Handling**: Displays appropriate messages if the PDF URL is invalid or unavailable.
- **Sharing**: Allows users to share the PDF link.

Important Methods:
- **`loadPdf(String pdfUrl)`**: Loads the PDF in the `WebView`.
- **`fetchPdfUrlFromServer(int noteId)`**: Fetches the PDF URL from the server if not provided.

Code Snippet:
```java
private void loadPdf(String pdfUrl) {
        webView.setWebViewClient(new WebViewClient() {
        @Override
        public void onPageFinished(WebView view, String url) {
        progressBar.setVisibility(View.GONE);
        webView.setVisibility(View.VISIBLE);
```

```
        }
        });

        WebSettings webSettings = webView.getSettings();
        webSettings.setJavaScriptEnabled(true);
        webSettings.setBuiltInZoomControls(true);
        webSettings.setDisplayZoomControls(false);

        if (pdfUrl.toLowerCase().endsWith(".pdf")) {
        String googleDocsUrl = "https://docs.google.com/viewer?url=" + Uri.encode(pdfUrl) +
"&embedded=true";
        webView.loadUrl(googleDocsUrl);
        } else {
        webView.loadUrl(pdfUrl);
        }
}
```

---

4. Upload Note Activity (`UploadNoteActivity.java`)
This activity allows users to upload new notes to the server.

Key Features:
- **File Picker**: Allows users to select a PDF file for upload.
- **Form Validation**: Ensures all required fields are filled before uploading.
- **Progress Indicator**: Displays upload progress.

Important Methods:
- **`uploadNote()`**: Handles the note upload process.
- **`setupFilePicker()`**: Configures the file picker for selecting PDF files.

Code Snippet:
```java
private void uploadNote() throws FileNotFoundException {
        if (universitySpinner.getText().toString().isEmpty() ||
degreeSpinner.getText().toString().isEmpty()) {
        Toast.makeText(this, "Please fill in all fields", Toast.LENGTH_SHORT).show();
        return;
        }

        InputStream inputStream = getContentResolver().openInputStream(selectedFileUri);
        byte[] fileBytes = IOUtils.toByteArray(inputStream);

        MultipartBody.Part filePart = MultipartBody.Part.createFormData("file", "note.pdf",
        RequestBody.create(MediaType.parse("application/pdf"), fileBytes));

        uploadService.uploadNote(filePart).enqueue(new Callback<ResponseBody>() {
```

```
        @Override
        public void onResponse(Call<ResponseBody> call, Response<ResponseBody>
response) {
        if (response.isSuccessful()) {
                Toast.makeText(UploadNoteActivity.this, "Note uploaded successfully",
Toast.LENGTH_SHORT).show();
        } else {
                Toast.makeText(UploadNoteActivity.this, "Upload failed",
Toast.LENGTH_SHORT).show();
        }
        }

        @Override
        public void onFailure(Call<ResponseBody> call, Throwable t) {
        Toast.makeText(UploadNoteActivity.this, "Network error",
Toast.LENGTH_SHORT).show();
        }
        });
}
```

---

5. Saved Notes Activity (`SavedNotesActivity.java`)
This activity displays notes saved locally for offline access.

Key Features:
- **Offline Access**: Allows users to view saved notes without an internet connection.
- **Delete and Share**: Users can delete or share saved notes.

Important Methods:
- **`loadSavedNotes()`**: Loads saved notes from local storage.
- **`deleteSavedPdf(SavedNotesManager.SavedNote note)`**: Deletes a saved note.

---

6. Saved Notes Manager (`SavedNotesManager.java`)
This class handles saving and managing notes locally.

Key Features:
- **File Management**: Saves notes as PDF files in local storage.
- **Metadata Storage**: Stores metadata (e.g., subject, module, year) in shared preferences.

Important Methods:
- **`saveNote()`**: Saves a note to local storage.
- **`getSavedNotes()`**: Retrieves a list of saved notes.

---

7. Notes Adapter (`NotesAdapter.java`)
The `NotesAdapter` binds note data to the UI components in the RecyclerView.

Key Features:
- **Dynamic Data Binding**: Binds note data (e.g., subject, module, year) to the UI.
- **Save and Share**: Allows users to save or share notes.

---

8. Notes Data Processing (`NotesDataProcessing.java`)
This class handles data fetching, filtering, and processing for the Notes module.

Key Features:
- **Database Integration**: Fetches notes from the local SQLite database.
- **Recent Notes**: Saves and retrieves recently accessed notes.

Conclusion
The Notes module in the Kstudents app is a comprehensive feature that allows users to browse, view, upload, and manage academic notes. By integrating backend APIs, local storage, and intuitive UI components, this module ensures a seamless user experience for both online and offline scenarios.

Documentation: Upload Module in the Kstudents Android App

The **Upload** module in the Kstudents Android app allows users to upload academic resources, such as question papers and notes, to the backend server. This module provides a user-friendly interface for selecting files, filling in metadata, and uploading them securely using Retrofit.

Overview of the Upload Module
The Upload module consists of the following components:
1. **Upload Service (`UploadService.java`)**: Defines the API endpoints for uploading files.
2. **Upload PDF Activity (`UploadPdf.java`)**: Handles the UI and logic for uploading question papers.
3. **Upload Note Activity (`UploadNoteActivity.java`)**: Handles the UI and logic for uploading notes.

---

1. Upload Service (`UploadService.java`)
The `UploadService` interface defines the API endpoints for uploading files to the backend server. It uses Retrofit to handle HTTP requests.

Key Features:
- **Multipart Uploads**: Supports uploading files along with metadata using `@Multipart`.
- **Endpoints**:
  - **`uploadPdf`**: Uploads question papers.
  - **`uploadNote`**: Uploads notes.

Endpoints:
- **Upload Question Papers:**
  ```java
  @Multipart
  @POST("Exam/mobile_question_up.php")
  Call<ResponseBody> uploadPdf(
          @Part("file_id") RequestBody fileId,
          @Part("degree") RequestBody degree,
          @Part("subj") RequestBody subj,
          @Part("sem") RequestBody sem,
          @Part("qyear") RequestBody qyear,
          @Part("university") RequestBody university,
          @Part MultipartBody.Part file
  );
  ```

- **Upload Notes:**
  ```java
  @Multipart
  @POST("Exam/notes_upload.php")
  Call<ResponseBody> uploadNote(
          @Part("file_id") RequestBody fileId,
          @Part("title") RequestBody title,
          @Part("module") RequestBody module,
          @Part("degree") RequestBody degree,
          @Part("semester") RequestBody semester,
          @Part("university") RequestBody university,
          @Part("year") RequestBody year,
          @Part MultipartBody.Part file
  );
  ```

---

2. Upload PDF Activity (`UploadPdf.java`)
This activity provides a user interface for uploading question papers. It allows users to select a PDF file, fill in metadata (e.g., university, degree, semester, year, subject), and upload the file to the server.

Key Features:
- **File Picker**: Allows users to select a PDF file from their device.
- **Dynamic Dropdowns**: Fetches universities and degrees dynamically from the backend.
- **Form Validation**: Ensures all required fields are filled before uploading.
- **Progress Indicator**: Displays upload progress and status.

Important Methods:
- **`setupPdfPicker()`**: Configures the file picker for selecting PDF files.
- **`fetchUniversities()`**: Fetches the list of universities from the backend.
- **`fetchDegrees(int universityId)`**: Fetches the list of degrees for a selected university.
- **`uploadPdf()`**: Handles the file upload process.

Code Snippet:
```java
private void uploadPdf() throws FileNotFoundException {
        // Validate inputs
        if (universityAutoComplete.getText().toString().isEmpty()) {
        Toast.makeText(this, "Please select a university", Toast.LENGTH_SHORT).show();
        return;
        }

        progressBarUpload.setVisibility(View.VISIBLE);
        tvUploadStatus.setVisibility(View.VISIBLE);

        InputStream inputStream = getContentResolver().openInputStream(selectedPdfUri);
```

```
        byte[] fileBytes = IOUtils.toByteArray(inputStream);

        String timestamp = new SimpleDateFormat("yyyyMMdd_HHmmss",
Locale.getDefault()).format(new Date());
        String fileName = "file_" + timestamp + ".pdf";

        MultipartBody.Part filePart = MultipartBody.Part.createFormData("file", fileName,

        RequestBody.create(MediaType.parse(getContentResolver().getType(selectedPdfUri
)), fileBytes));

        RequestBody fileId = RequestBody.create(MediaType.parse("text/plain"),
timestamp);
        RequestBody universityPart = RequestBody.create(MediaType.parse("text/plain"),
universityId.toString());
        RequestBody degreePart = RequestBody.create(MediaType.parse("text/plain"),
degree);
        RequestBody subjPart = RequestBody.create(MediaType.parse("text/plain"), subj);
        RequestBody semPart = RequestBody.create(MediaType.parse("text/plain"), sem);
        RequestBody qyearPart = RequestBody.create(MediaType.parse("text/plain"),
qyear);

        uploadService.uploadPdf(fileId, degreePart, subjPart, semPart, qyearPart,
universityPart, filePart)
        .enqueue(new Callback<ResponseBody>() {
        @Override
        public void onResponse(Call<ResponseBody> call, Response<ResponseBody>
response) {
                progressBarUpload.setVisibility(View.GONE);
                if (response.isSuccessful()) {
                Toast.makeText(UploadPdf.this, "Upload successful",
Toast.LENGTH_SHORT).show();
                resetForm();
                } else {
                Toast.makeText(UploadPdf.this, "Upload failed: " + response.message(),
Toast.LENGTH_SHORT).show();
                }
        }

        @Override
        public void onFailure(Call<ResponseBody> call, Throwable t) {
                progressBarUpload.setVisibility(View.GONE);
                Toast.makeText(UploadPdf.this, "Upload failed: " + t.getMessage(),
Toast.LENGTH_SHORT).show();
        }
        });
}
```

Dynamic Dropdowns:
- **Universities**: Populated dynamically using the `fetchUniversities()` method.
- **Degrees**: Populated dynamically based on the selected university using the `fetchDegrees()` method.

---

3. Upload Note Activity (`UploadNoteActivity.java`)
This activity provides a user interface for uploading notes. It is similar to the `UploadPdf` activity but includes additional fields for specifying the title and module of the note.

Key Features:
- **File Picker**: Allows users to select a PDF file from their device.
- **Form Validation**: Ensures all required fields are filled before uploading.
- **Progress Indicator**: Displays upload progress and status.

Important Methods:
- **`uploadNote()`**: Handles the file upload process for notes.

Code Snippet:
```java
private void uploadNote() throws FileNotFoundException {
        InputStream inputStream = getContentResolver().openInputStream(selectedFileUri);
        byte[] fileBytes = IOUtils.toByteArray(inputStream);

        MultipartBody.Part filePart = MultipartBody.Part.createFormData("file", "note.pdf",
        RequestBody.create(MediaType.parse("application/pdf"), fileBytes));

        RequestBody fileId = RequestBody.create(MediaType.parse("text/plain"), "note_id");
        RequestBody titlePart = RequestBody.create(MediaType.parse("text/plain"), title);
        RequestBody modulePart = RequestBody.create(MediaType.parse("text/plain"),
module);
        RequestBody degreePart = RequestBody.create(MediaType.parse("text/plain"),
degree);
        RequestBody semesterPart = RequestBody.create(MediaType.parse("text/plain"),
semester);
        RequestBody universityPart = RequestBody.create(MediaType.parse("text/plain"),
university);
        RequestBody yearPart = RequestBody.create(MediaType.parse("text/plain"), year);

        uploadService.uploadNote(fileId, titlePart, modulePart, degreePart, semesterPart,
universityPart, yearPart, filePart)
        .enqueue(new Callback<ResponseBody>() {
        @Override
        public void onResponse(Call<ResponseBody> call, Response<ResponseBody>
response) {
                if (response.isSuccessful()) {
```

```
                Toast.makeText(UploadNoteActivity.this, "Note uploaded successfully",
Toast.LENGTH_SHORT).show();
                } else {
                Toast.makeText(UploadNoteActivity.this, "Upload failed",
Toast.LENGTH_SHORT).show();
                }
        }

        @Override
        public void onFailure(Call<ResponseBody> call, Throwable t) {
                Toast.makeText(UploadNoteActivity.this, "Network error",
Toast.LENGTH_SHORT).show();
        }
        });
}
```

---

4. User Interaction
- **File Selection**: Users can select a PDF file using the file picker.
- **Metadata Input**: Users fill in metadata such as university, degree, semester, year, and subject.
- **Upload**: Users click the upload button to send the file and metadata to the server.
- **Feedback**: The app provides feedback on the upload status (e.g., success, failure, or progress).

---

5. Error Handling
- **Validation Errors**: Ensures all required fields are filled before uploading.
- **Network Errors**: Displays appropriate messages if the upload fails due to network issues.
- **Server Errors**: Handles server-side errors and displays the error message to the user.

Conclusion
The Upload module in the Kstudents app provides a robust and user-friendly interface for uploading academic resources. By integrating Retrofit for API calls, dynamic dropdowns for metadata selection, and progress indicators for feedback, this module ensures a seamless user experience. It is a valuable feature for users contributing to the app's academic resource repository.

Documentation for `questions` Class

The `questions` class is an Android `AppCompatActivity` that serves as the main interface for users to browse and interact with question papers. It provides functionality for selecting universities, degrees, and semesters, and displays recent questions viewed by the user.

Class Overview

Package
```java
package com.keralatechreach.kstudents.Question_View;
```

Inheritance
```java
public class questions extends AppCompatActivity
```

Key Features

1. **Dropdown Selection**:
   - Allows users to select a university, degree, and semester using `AutoCompleteTextView` dropdowns.
   - Dynamically loads degrees based on the selected university.

2. **Recent Questions**:
   - Displays a list of recently viewed questions using a `RecyclerView`.
   - If no recent questions are available, a placeholder message is shown.

3. **Navigation**:
   - Integrates with a `BottomNavigationView` for seamless navigation between different sections of the app.

4. **Data Fetching**:
   - Fetches universities and degrees from a `DataRepository`.
   - Loads recent questions from the local database.

5. **Actions**:
   - Allows users to submit their selected filters to view questions.
   - Provides options to upload new questions, view saved questions, and access notes.

UI Components

XML Layout
The activity uses the layout file `activity_questions.xml`. Key components include:
- **AutoCompleteTextView**:
  - `universitySpinner`: Dropdown for selecting a university.
  - `degSpinner`: Dropdown for selecting a degree.
  - `semSpinner`: Dropdown for selecting a semester.
- **RecyclerView**:
  - `recentQuestionsRecyclerView`: Displays the list of recent questions.
- **TextView**:
  - `recentQuestionsEmptyText`: Placeholder text when no recent questions are available.
- **BottomNavigationView**:
  - `bottom_navigation`: Provides navigation options.

Key Methods

`onCreate(Bundle savedInstanceState)`
- Initializes the activity and sets up the UI components.
- Configures dropdowns, navigation, and the recent questions list.
- Fetches universities and recent questions.

`setupBottomNavigation()`
- Configures the `BottomNavigationView` to handle navigation between different sections of the app.

`fetchUniversities()`
- Fetches the list of universities from the `DataRepository`.
- Populates the `universitySpinner` dropdown with the fetched data.

`fetchDegrees(int universityId)`
- Fetches the list of degrees for the selected university from the `DataRepository`.
- Populates the `degSpinner` dropdown with the fetched data.

`loadRecentQuestions()`
- Loads the list of recently viewed questions from the local database.
- Updates the `RecyclerView` with the fetched data or displays a placeholder if no data is available.

`degsubmit(View view)`
- Handles the submission of selected filters (university, degree, semester).
- Launches the `QuestionView` activity with the selected parameters.

`uploadquestion(View v)`

- Launches the `UploadPdf` activity for uploading new questions.

`savedQuestions(View v)`
- Launches the `SavedQuestionsActivity` to view saved questions.

`notes(View v)`
- Launches the `Notes` activity.

Data Handling

`DataRepository`
- Used to fetch data such as universities and degrees.
- Provides methods for retrieving recent questions from the local database.

`RecentQuestionsAdapter`
- Adapter for the `RecyclerView` to display recent questions.
- Updates the UI dynamically when new data is loaded.

Logging and Debugging

- Logs are added using `Log.d` and `Log.e` for debugging purposes.
- Example:
  ```java
  Log.d(TAG, "University: " + university + " (ID: " + universityId + "), Degree: " + degree + ", Semester: " + semester);
  ```

Error Handling

- Displays `Toast` messages for errors such as missing data or failed API calls.
- Example:
  ```java
  Toast.makeText(this, "Please select all fields", Toast.LENGTH_SHORT).show();
  ```

Dependencies

- **AndroidX**: For modern Android components like `AppCompatActivity` and `RecyclerView`.
- **Material Components**: For UI elements like `BottomNavigationView`.

- **Retrofit**: For API calls (used indirectly via `ApiService_Questions`).

Example Usage

Viewing Questions
1. Select a university, degree, and semester from the dropdowns.
2. Click the "Submit" button to view questions for the selected filters.

Viewing Recent Questions
- Recent questions are displayed automatically in the `RecyclerView` when the activity is launched.

Uploading Questions
- Click the "Upload Questions" button to navigate to the `UploadPdf` activity.

Notes

- The `questions` class relies heavily on the `DataRepository` for data fetching and storage.
- The activity is designed to be responsive, with dynamic updates to the UI based on user interactions and data availability.

SavedQuestion Documentation

Overview
The SavedQuestion system is a feature that allows users to save, manage, and view question papers within the KStudents Android application. It consists of three main components:

1. SavedQuestionsManager
Central class that handles saving, retrieving, and managing saved questions.

```java
public static class SavedQuestion {
	// Question attributes
	private String qtext;	// Question text
	private String fileName;	// PDF file name
	private String subject;	// Subject name
	private String year;	// Academic year
	private String semester;	// Semester
	private String degree;		// Degree program
}
```

Key Methods
- `addSavedQuestion()`: Saves a new question to SharedPreferences
- `getSavedQuestions()`: Retrieves all saved questions
- `removeSavedQuestion()`: Deletes a saved question

2. SavedQuestionsActivity
Main UI for displaying and managing saved questions.

Features
- Displays list of saved questions in cards
- Empty state view when no questions are saved
- Options to view, delete and share questions
- Bottom navigation integration
- Browse questions button to find new questions

3. SavedPdfViewerActivity
PDF viewer for saved question papers.

Features
- PDF viewing with zoom and scroll
- Share functionality
- Delete option
- Deep link support

- Error handling for missing files

Storage
- Questions metadata stored in SharedPreferences
- PDF files stored in app's private directory: `/saved_questions/`
- Uses Android FileProvider for sharing

Usage Example

```java
// Save a question
SavedQuestionsManager.addSavedQuestion(context,
        "Question text",
        "exam2023.pdf",
        "Mathematics",
        "2023",
        "Semester 1",
        "B.Tech");

// Get saved questions
List<SavedQuestion> questions = SavedQuestionsManager.getSavedQuestions(context);

// Remove a question
SavedQuestionsManager.removeSavedQuestion(context, "exam2023.pdf");
```

Deep Linking
The system supports deep links in the format:
```
kstudents://viewer?file=filename.pdf&subject=Math&year=2023
```

Dependencies
- `com.github.barteksc:android-pdf-viewer`: For PDF rendering
- Android SharedPreferences
- Android FileProvider

Compatibility
- Maintains backward compatibility for older saved questions without semester/degree
- Handles missing files gracefully
- Supports Android file sharing conventions

Security Implementation Documentation

Overview
The security system in the KStudents Android application implements encryption and secure storage mechanisms for sensitive data, particularly API keys. The system consists of three main components:

1. KeystoreManager
Primary class for handling encryption operations using Android Keystore System.

Key Features
- Uses Android Keystore for secure key storage
- Implements AES/GCM/NoPadding encryption
- Provides encryption and decryption methods for API keys

```java
// Constants
private static final String KEYSTORE_PROVIDER = "AndroidKeyStore";
private static final String KEY_ALIAS = "ApiKeyAlias";
private static final String TRANSFORMATION = "AES/GCM/NoPadding";
private static final int GCM_TAG_LENGTH = 128;
```

Main Methods
- `encryptApiKey()`: Encrypts API keys
- `decryptApiKey()`: Decrypts stored API keys
- `getOrCreateSecretKey()`: Manages secret key generation/retrieval
- `getSecretKey()`: Retrieves existing secret key

2. SharedPrefsManager
Manages secure storage of encrypted data using SharedPreferences.

Storage Keys
```java
private static final String PREF_NAME = "secure_api_prefs";
private static final String PREF_ENCRYPTED_API_KEY = "encrypted_api_key";
private static final String PREF_IV = "encryption_iv";
private static final String KEY_CONTACT_MESSAGE_ID = "contact_message_id";
```

Key Methods
- `saveEncryptedApiKey()`: Stores encrypted API key
- `getEncryptedApiKey()`: Retrieves encrypted API key
- `saveIV()`: Stores initialization vector
- `getIV()`: Retrieves initialization vector

3. ApiAuthInterceptor
Handles API authentication by intercepting network requests.

```java
public Response intercept(Chain chain) throws IOException {
        // Adds API key to requests
        // Manages authentication headers
}
```

Security Features

Encryption
- **Algorithm**: AES (Advanced Encryption Standard)
- **Mode**: GCM (Galois/Counter Mode)
- **Padding**: NoPadding
- **Key Size**: 128-bit
- **IV**: Unique initialization vector for each encryption

Key Storage
1. **Android Keystore System**
   - Hardware-backed key storage when available
   - Protected from extraction
   - Bound to device security

2. **SharedPreferences**
   - Encrypted data storage
   - IV storage
   - Contact message IDs

Usage Example

```java
// Encrypting an API key
String encryptedKey = KeystoreManager.encryptApiKey(context, "api-key-value");

// Storing encrypted key
SharedPrefsManager.saveEncryptedApiKey(context, encryptedKey);

// Retrieving and decrypting
String decryptedKey = KeystoreManager.decryptApiKey(context);
```

Security Best Practices
1. Never store API keys in plaintext
2. Use unique IVs for each encryption
3. Implement hardware-backed keystore when available

4. Log security-related events for debugging
5. Handle encryption/decryption errors gracefully

Error Handling
- Extensive logging for debugging
- Null checks for missing keys
- Exception handling for cryptographic operations
- Fallback mechanisms for missing security components

Dependencies
- Android Keystore System
- Android SharedPreferences
- OkHttp for network interceptors

Compatibility
- Android API 23+ (Marshmallow)
- Supports hardware-backed keystore where available
- Graceful degradation on unsupported devices

ShareLinkGenerator Documentation

Overview
The `ShareLinkGenerator` class provides a centralized way to create and share deep links for various features in the KStudents Android application. It handles URL generation and sharing functionality for question papers, study notes, jobs, and events.

Base Configuration
```java
private static final String BASE_URL = "https://www.keralatechreach.in/Exam";
private static final String PLAY_STORE_LINK =
"https://play.google.com/store/apps/details?id=com.keralatechreach.kstudents";
```

Features

1. Question Papers Sharing
```java
createQuestionViewLink(String universityName, int universityId, String degree, String semester)
shareQuestionViewLink(Context context, String universityName, int universityId, String degree, String semester)
```
- Creates URLs for question papers
- Format: `BASE_URL/questions.php?university_name=X&university_id=Y&deg=Z&sem=W`

2. Study Notes Sharing
```java
createNotesViewLink(String universityName, int universityId, String degree, String semester)
shareNotesViewLink(Context context, String universityName, int universityId, String degree, String semester)
```
- Creates URLs for study notes
- Format: `BASE_URL/notes?university_name=X&university_id=Y&deg=Z&sem=W`

3. Job Opportunities Sharing
```java
createJobDetailLink(String jobTitle, String companyName, int jobId)
shareJob(Context context, String jobTitle, String companyName, int jobId)
```
- Creates URLs for job listings
- Format: `BASE_URL/JobRedirect.php?job_title=X&company_name=Y&job_id=Z`

4. Events Sharing
```java
```

createEventDetailLink(String eventName, int eventId)
shareEvent(Context context, String eventName, int eventId)
```
- Creates URLs for events
- Format: `BASE_URL/EventRedirect.php?event_name=X&event_id=Y`

## Usage Example

```java
// Share a question paper
ShareLinkGenerator.shareQuestionViewLink(
        context,
        "University of Kerala",
        1,
        "B.Tech",
        "Semester 4"
);

// Share a job opportunity
ShareLinkGenerator.shareJob(
        context,
        "Software Engineer",
        "Tech Company",
        12345
);
```

Features
- URL parameter encoding for special characters
- Automatic Play Store link addition
- Share intent creation with chooser dialog
- Deep linking support
- Consistent sharing format across features

Share Message Format
Each shared message includes:
1. Content description
2. Deep link URL
3. Play Store link for app download

Security Considerations
- All parameters are URI encoded to prevent injection
- Uses HTTPS for secure communication
- Implements standard Android sharing intents

Dependencies
- Android URI builder
- Android Intent system
```

- Android Context for sharing

Best Practices
1. Always provide valid parameters to avoid malformed URLs
2. Use appropriate context (usually Activity context)
3. Handle potential null values in parameters
4. Consider URI encoding for special characters

Notes
- All sharing methods are synchronous
- Requires Android sharing permissions
- Compatible with standard Android share sheet
- Maintains consistent branding across shares

Version Checker Documentation

Overview
The Version Checker system provides automatic version checking and update management for the KStudents Android application. It consists of three main components working together to handle version verification and update prompts.

Components

1. VersionResponse
Data class that models the server response for version information.

```java
public class VersionResponse {
	@SerializedName("version_code")
	private int versionCode;	// Internal version number
	@SerializedName("version_name")
	private String versionName;		// User-visible version
	@SerializedName("update_url")
	private String updateUrl;	// URL for update
	@SerializedName("update_message")
	private String updateMessage;	// Custom update message
}
```

2. VersionCheckApi
Retrofit interface defining the API endpoint.

```java
public interface VersionCheckApi {
	@GET("VersionCheck/check_version.php")
	Call<VersionResponse> checkVersion();
}
```

3. VersionCheckManager
Main class handling version checking logic and update dialogs.

Key Features
- Automatic version comparison
- Update dialog display
- Play Store redirection
- Error handling
- Force check capability

Usage Example

```java
// Initialize manager
VersionCheckManager versionManager = new VersionCheckManager(context);

// Check for updates
versionManager.checkAppVersion();

// Force check (e.g., from settings)
versionManager.forceVersionCheck();
```

Server Response Format
Expected JSON response:
```json
{
    "version_code": 123,
    "version_name": "1.2.3",
    "update_url": "https://play.google.com/store/apps/...",
    "update_message": "New features available!"
}
```

Update Dialog
The update dialog includes:
- Title: "Update Available"
- Custom message or default version info
- Update button: Opens Play Store/download URL
- Cancel button: Dismisses dialog
- Non-cancelable behavior

Technical Details

Dependencies
- Retrofit2 for network calls
- GSON for JSON parsing
- AndroidX for UI components

API Configuration
```java
baseUrl = "https://www.keralatechreach.in/"
endpoint = "VersionCheck/check_version.php"
```

Version Comparison
```java
private boolean isUpdateAvailable(int currentVersionCode, int latestVersionCode) {
```

```
        return latestVersionCode > currentVersionCode;
}
```

Error Handling
- Package manager exceptions
- Network failures
- Invalid server responses
- Missing version information

Best Practices
1. Check version on app launch
2. Provide force check option
3. Handle offline scenarios gracefully
4. Log version check failures
5. Use non-blocking network calls

Security Considerations
- HTTPS for secure communication
- Version code verification
- Valid update URL validation
- Protected API endpoints

Note
The system uses Android's `versionCode` for comparison while displaying the user-friendly `versionName` in dialogs.

KStudents Android App Documentation

Core Activities Documentation

1. MainActivity
Main landing page of the application with key features:

Features
- Bottom navigation integration
- Google Mobile Ads implementation
- Version checking
- Analytics sync service
- Activity navigation

```java
private static final String BANNER_AD_UNIT_ID = "ca-app-pub-
5726640321358021/3777989098";
```

Key Methods
- `initializeMobileAds()`: Sets up Google Mobile Ads SDK
- `setupBannerAd()`: Configures adaptive banner advertisements
- `setupBottomNavigation()`: Handles navigation between main sections
- `checkForUpdates()`: Initiates version check

2. Intro (Splash Screen)
Initial loading screen with animations.

Features
- Logo and title animations
- API key initialization
- Database setup
- Timed transition to MainActivity

```java
private static final int SPLASH_DURATION = 1000; // 1.0 seconds
```

3. MoreActivity
Additional features and settings page.

Features
- Navigation to secondary features:
  - Saved Notes
  - Message Us

  - Exams
  - About
  - Entrance Exams
  - Government Initiatives
  - Saved Questions
- External links:
  - Play Store rating
  - Privacy policy

4. AboutView
Information about the application.

Features
- Bottom navigation integration
- Static information display
- Navigation back to main sections

Navigation Structure

```
MainActivity
├── Questions
├── Course List
├── Upload Question
├── Events
├── Initiatives
├── Jobs
├── Exams
└── More
        ├── Saved Notes
        ├── Message Us
        ├── Exams
        ├── About
        ├── Entrance
        ├── Initiatives
        ├── Saved Questions
        ├── Rate App
        └── Privacy Policy
```

Common Components

Bottom Navigation
Present in all main activities with 5 sections:
- Home
- Questions
- Jobs
- Events

- More

```java
bottomNavigationView.setOnItemSelectedListener(item -> {
        // Navigation logic
});
```

Animation Resources
Located in `res/anim/`:
- `logo_animation.xml`
- `title_animation.xml`

Security Features
- API key management
- EdgeToEdge enabled activities
- Secure transitions
- Protected navigation

Best Practices
1. Activity state management
2. Proper activity lifecycle handling
3. Consistent navigation patterns
4. Ad implementation guidelines
5. Version control integration

Dependencies
- Google Mobile Ads SDK
- AndroidX
- Material Design Components
- Custom animations
- Database helpers

Notes
- The app uses a bottom navigation pattern for main navigation
- Activities handle their own state management
- Ad implementation follows Google AdMob guidelines
- Version checking ensures app updates
- Database initialization occurs at app launch