Q1. Consider an app similar to Google Lens that identifies the object captured by the camera. Assuming that a user frequently uses it, which hardware unit would you prefer to be used to identify the objects? Justify.

For an app similar to Google Lens that identifies objects captured by a camera, several hardware units can be considered for efficient and effective object recognition. The choice of hardware depends on factors such as speed, accuracy, power consumption, and cost. Here are some options along with their justifications:

1. **Dedicated Neural Processing Units (NPUs)**

Justification:
- **Efficiency:** NPUs are designed specifically for running machine learning algorithms efficiently. They are optimized for tasks like image recognition, making them highly efficient for this purpose.
- **Speed:** NPUs can process large amounts of data in parallel, which is crucial for real-time object identification.
- **Low Power Consumption:** They are generally more power-efficient compared to running the same tasks on CPUs or GPUs, which can be important for mobile devices.
- **Accuracy:** Due to their design for AI tasks, NPUs can provide high accuracy in object recognition.
- **Cost:** While NPUs can add to the cost of the device, they are becoming more common in smartphones and other devices, so the cost is decreasing.

2. **System on Chip (SoC) with Integrated AI Accelerators**

Justification:
- **Integration:** Many modern SoCs come with integrated AI accelerators like Qualcomm's Hexagon DSP or Huawei's Kirin NPU. These are specifically designed to handle tasks like image recognition efficiently.
- **Power Efficiency:** The integration of these accelerators into the SoC can lead to better power efficiency compared to using separate components.
- **Performance:** They offer good performance for AI tasks without the need for additional dedicated hardware.
- **Cost:** Since these components are part of the SoC, they might not significantly increase the overall cost of the device.

3. **Graphics Processing Units (GPUs)**

Justification:
- **Parallel Processing:** GPUs excel at parallel processing, which is crucial for tasks like image recognition.

- **Availability:** Most smartphones and modern devices already have GPUs for rendering graphics. Utilizing them for AI tasks can be cost-effective.
- **Community Support:** There is a wealth of knowledge and libraries available for running machine learning models on GPUs.
- **Upgradability:** With the improvement of GPU technology, software updates can potentially improve the app's object recognition capabilities.
- **Cost:** GPUs can be more cost-effective than specialized hardware like NPUs, especially if the device already has a capable GPU.

4. **Hybrid Approaches (Combining Hardware Units)**

Justification:
- **Optimal Performance:** Combining NPUs, GPUs, and other accelerators in a heterogeneous computing environment can offer the best of all worlds.
- **Task Offloading:** Different units can handle different parts of the object recognition pipeline, optimizing performance and efficiency.
- **Flexibility:** This approach allows for flexibility in adapting to different device architectures and user requirements.
- **Future-Proofing:** By combining different units, the app can be prepared for future advancements in hardware technology.

Conclusion

The ideal hardware unit for object identification in an app like Google Lens would depend on a variety of factors, including the device's intended use, budget constraints, power efficiency requirements, and desired level of performance.

- For high-performance, low-power mobile devices, NPUs or integrated AI accelerators within SoCs might be the best choice.
- For devices where cost is a significant factor, utilizing the existing GPU capabilities can be a sensible approach.
- A hybrid approach could offer the best of both worlds, optimizing performance, efficiency, and flexibility.

Ultimately, the choice of hardware should be based on a careful consideration of these factors, keeping in mind the target user experience and the app's intended use cases.

Q2. Consider a smartphone messaging app that syncs with its server once every 100 ms. The advantage of this app is that it receives the message very quickly. Is there any disadvantage of using it?

Syncing a smartphone messaging app with its server once every 100 milliseconds does offer the advantage of near-instant message reception, which can be beneficial for real-time communication. However, there are several potential disadvantages to consider with such a high syncing frequency:

1. **Battery Drain:**
   - Constantly syncing with the server at a high frequency consumes more battery power.
   - This can lead to faster battery drain, especially for users who frequently use the messaging app throughout the day.

2. **Data Usage:**
   - More frequent syncing means more data being transmitted between the app and the server.
   - Users with limited data plans may find that their data is used up more quickly than expected.

3. **Network Congestion:**
   - If a large number of users are using the app simultaneously with frequent syncs, it could lead to network congestion.
   - This congestion may result in slower data speeds or delays for all users on that network.

4. **Server Load:**
   - The server hosting the messaging app will experience a higher load with frequent sync requests from numerous users.
   - This could potentially affect the server's performance and responsiveness.

5. **App Responsiveness:**
   - While messages are received quickly, the app's overall responsiveness might suffer.
   - Other functions of the app, such as typing, sending messages, or accessing other features, may slow down due to the constant syncing process.

6. **User Privacy:**
   - More frequent syncing means more data being sent back and forth between the device and the server.
   - Users concerned about privacy may be wary of the amount of data being transmitted, especially if the messages contain sensitive information.

7. **Notification Overload:**
   - With messages arriving every 100 milliseconds, users may receive a barrage of notifications.
   - This can be overwhelming and lead to notification fatigue, where users start ignoring or disabling notifications altogether.

8. **Potential for Message Loss:**
   - If the app is unable to handle the high volume of incoming messages or if there are network issues, messages could potentially be lost.
   - This could result in users missing important information or conversations.

9. **Device Heating:**

   - Constant data transfers and processing can cause the device to heat up.
   - Prolonged exposure to high temperatures can affect the device's performance and battery life over time.

10. **Higher Costs for Users:**
   - Users with limited data plans might incur additional charges due to increased data usage.
   - Continuous syncing may also require users to have higher data plans, leading to increased monthly costs.

11. **Security Risks:**
   - More frequent communication increases the attack surface for potential security vulnerabilities.
   - The app and server must be secured against potential threats to protect user data and privacy.

12. **Testing and Maintenance Challenges:**
   - Implementing and maintaining such a high-frequency syncing mechanism requires thorough testing and monitoring.
   - Any bugs or issues related to syncing could be more challenging to diagnose and fix.

Conclusion:
While receiving messages quickly is undoubtedly advantageous for real-time communication, the drawbacks of a smartphone messaging app syncing every 100 milliseconds should be carefully considered. Balancing the need for instant communication with factors such as battery life, data usage, network congestion, server load, user experience, privacy, and security is crucial for providing a seamless and efficient messaging experience for users. Developers must weigh these considerations to find the optimal syncing frequency that meets user expectations without compromising the app's performance or usability.

Q3. Assume that we want to retreive and display a list with around 1000 elements in
an app. The developer decides to show them using a listview. Is this the right
approach? Justify.

Using a `ListView` to display a list of around 1000 elements in a mobile app can be a reasonable approach, but it comes with considerations regarding performance, user experience, and the nature of the data. Here are some points to consider:

1. **Performance:**
   - **Rendering:** `ListView` efficiently renders a subset of elements that are visible on the screen.
   - **Virtualization:** It uses a concept called "virtualization" where only the visible items are actually rendered, improving performance.
   - **Optimized Scrolling:** Users can scroll through the list smoothly, even with a large number of items.

2. **Memory Usage:**
   - `ListView` helps manage memory efficiently by recycling views as they scroll off the screen.
   - It does not need to keep all 1000 elements in memory simultaneously, which reduces the memory footprint.

3. **User Experience:**
   - **Navigation:** Users can easily scroll through the list to find items of interest.
   - **Search and Filter:** A `ListView` can also work well with search or filter functionalities to help users find specific items in a large list.
   - **Familiarity:** `ListView` is a common UI pattern in mobile apps, so users are likely familiar with how to interact with it.

4. **Data Loading:**
   - Developers can implement pagination or lazy loading techniques to fetch and display data in chunks.
   - This means not all 1000 elements need to be loaded at once, improving the initial load time and reducing network usage.

5. **Screen Size and Density:**
   - Consideration should be given to the screen size and density of devices.
   - On smaller screens, displaying 1000 items might make each item too small to interact with comfortably.

6. **Customization and Performance Tuning:**
   - Developers can optimize `ListView` for performance by customizing item rendering, using view holders, and implementing efficient adapters.
   - Customizations can include different item layouts based on data types or states.

7. **Alternatives to Consider:**
   - **Pagination:** If users typically interact with a smaller subset of the list, consider loading more items as they scroll down.
   - **Search/Filter:** Implement search or filter functionality to help users find specific items quickly.
   - **Sectioned Lists:** Grouping items into sections can make it easier for users to navigate large lists.

When Might `ListView` Not Be Ideal?

   - **Heavy Interactivity:** If each list item is highly interactive (e.g., contains complex UI elements or actions), rendering 1000 items might lead to performance issues.
   - **Data Complexity:** If each item requires heavy data processing or fetching, it might impact performance.

  - **Limited Screen Space:** On smaller screens, displaying a long `ListView` might not provide a good user experience due to small item sizes.

Conclusion:
Using a `ListView` to display around 1000 elements can be an appropriate choice for many mobile apps, especially when considering performance optimizations, memory management, and user experience. However, developers should also consider factors such as device screen size, data complexity, and the need for interactivity. Customizations, such as lazy loading, pagination, or search/filter functionalities, can further enhance the usability of the list. It's essential to balance the benefits of displaying a large list with the potential challenges it might pose for performance and user interaction.

Q4. Suppose you have a large number of UI elements. How would you navigate its semantic tree?

In Kotlin Jetpack Compose, which is a modern reactive UI toolkit for building native Android UIs, navigating the UI tree involves using Compose's own set of functions and APIs. Jetpack Compose uses a declarative approach to building UIs, which means you describe what the UI should look like based on its current state.

Here are some common approaches to navigate the UI tree and interact with elements in Jetpack Compose:

1. **Referencing Elements:**
  - In Compose, you create and reference UI elements using composable functions.
  - You can create custom composable functions to encapsulate UI components.

  ```kotlin
  @Composable
  fun MyCustomButton(text: String, onClick: () -> Unit) {
      Button(onClick = onClick) {
          Text(text)
      }
  }
  ```

2. **Using State and State Hoisting:**
  - State is an essential concept in Compose for managing UI data.
  - You can define state variables and pass them down to composable functions.

  ```kotlin
  @Composable
  ```

   fun MyScreen() {
       var count by remember { mutableStateOf(0) }

       MyCustomButton("Increment", onClick = { count++ })

       Text("Count: $count")
   }

3. **Handling User Interactions:**
   - Compose provides `Clickable` and `Modifier.clickable` to handle user clicks.

   ```kotlin
   @Composable
   fun MyClickableText(text: String, onClick: () -> Unit) {
       Text(
           text = text,
           modifier = Modifier.clickable(onClick = onClick)
       )
   }
   ```

4. **Using Navigation Composables:**
   - For navigation, you can define composable functions for different screens.
   - Navigate between screens using `NavController`.

   ```kotlin
   @Composable
   fun ScreenA(navController: NavController) {
       Button(onClick = { navController.navigate("screenB") }) {
           Text("Go to Screen B")
       }
   }

   @Composable
   fun ScreenB() {
       Text("This is Screen B")
   }
   ```

5. **Using `rememberCoroutineScope` for Side Effects:**
   - For operations such as fetching data or performing animations, you can use
`rememberCoroutineScope`.

```kotlin
  @Composable
  fun MyFetchDataButton(fetchData: () -> Unit) {
      val coroutineScope = rememberCoroutineScope()

      Button(onClick = {
          coroutineScope.launch {
              fetchData()
          }
      }) {
          Text("Fetch Data")
      }
  }
  ```

6. **Using Composition Local:**
  - `CompositionLocal` allows you to access objects throughout the composition tree.
  - Useful for passing down objects like themes, settings, or authentication status.

  ```kotlin
  val CurrentUser = compositionLocalOf<User> { error("No user found!") }

  @Composable
  fun Greeting() {
      val user = LocalUser.current
      Text("Hello, ${user.name}")
  }
  ```

7. **Testing with `ComposeTestRule` and `onNode`:**
  - For testing UI interactions, you can use `ComposeTestRule` and `onNode` to find and interact with elements.

  ```kotlin
  @get:Rule
  val composeTestRule = createComposeRule()

  @Test
  fun testButtonClick() {
      composeTestRule.setContent {
          MyClickableText(text = "Click Me", onClick = { /* Perform test action */ })
      }

      composeTestRule.onNodeWithText("Click Me").performClick()
  ```

  }

8. **Using `LazyColumn` or `LazyRow` for Large Lists:**
  - For displaying large lists efficiently, use `LazyColumn` or `LazyRow`.
  - Items are only composed when they are visible on the screen.

  ```kotlin
  @Composable
  fun MyList(items: List<String>) {
      LazyColumn {
          items(items) { item ->
              Text(text = item)
          }
      }
  }
  ```

9. **Using `rememberScrollState` for Scrolling:**
  - For scrollable elements, you can use `rememberScrollState` to manage scroll position.

  ```kotlin
  @Composable
  fun ScrollableContent() {
      val scrollState = rememberScrollState()

      Column(Modifier.verticalScroll(scrollState)) {
          // Content here
      }
  }
  ```

10. **Interoperability with ViewModels:**
  - Compose works well with the ViewModel architecture component.
  - You can use `viewModel()` to access a ViewModel within a composable.

  ```kotlin
  @Composable
  fun MyScreen(viewModel: MyViewModel = viewModel()) {
      val data by viewModel.myData.observeAsState()

      // Use data in the UI
  }
  ```

These are just some examples of how you can navigate and interact with the UI tree in Jetpack Compose. Compose provides a powerful and flexible toolkit for building modern, reactive UIs in a more concise and declarative way compared to traditional Android Views.

Q5. Suppose you have designed that sends a heart patient's data using a service. Is
this a good approach, and why or why not?

Sending a heart patient's data using a service involves transmitting sensitive medical information over a network. Whether this approach is good or not depends on several factors, including data security, privacy regulations, encryption methods, compliance with healthcare standards, and patient consent. Here are some considerations:

Pros:

1. **Remote Monitoring:**
   - Sending patient data via a service enables remote monitoring of the heart patient's health.
   - Doctors and healthcare providers can receive real-time updates on the patient's condition, allowing for timely interventions.

2. **Faster Response Times:**
   - Immediate transmission of data allows healthcare professionals to respond quickly to any critical changes in the patient's health.
   - This can potentially save lives in emergency situations.

3. **Improved Patient Care:**
   - Continuous monitoring and analysis of patient data can lead to better treatment plans and personalized care.
   - Patterns and trends in the data can help doctors make informed decisions about medication adjustments or lifestyle recommendations.

4. **Efficient Data Management:**
   - Storing patient data in a centralized system allows for easy access by authorized healthcare professionals.
   - Electronic records can be updated, shared securely, and integrated with other medical systems.

5. **Data Analytics and Research:**
   - Aggregated and anonymized patient data can be used for medical research, leading to advancements in heart disease treatment and prevention.

Cons:

1. **Data Security Risks:**
   - Sending sensitive medical data over a network introduces security vulnerabilities.
   - Without proper encryption and security measures, data can be intercepted or accessed by unauthorized parties.

2. **Privacy Concerns:**
   - Patients' health information is highly confidential, and its exposure can lead to privacy breaches.
   - Compliance with regulations such as HIPAA (in the U.S.) or GDPR (in Europe) is crucial to protect patient privacy.

3. **Legal and Regulatory Compliance:**
   - Healthcare organizations must adhere to strict regulations regarding the handling, storage, and transmission of patient data.
   - Failure to comply with these regulations can result in legal consequences and fines.

4. **Reliability of the Service:**
   - The service used for data transmission must be reliable and have minimal downtime.
   - Interruptions in data transmission can affect patient care and the monitoring process.

5. **Costs and Infrastructure:**
   - Implementing and maintaining a secure data transmission service requires financial investment.
   - Healthcare organizations need robust infrastructure and skilled personnel to manage and secure the system.

6. **Informed Patient Consent:**
   - Patients must be fully informed about the data transmission process, its risks, and benefits.
   - Obtaining explicit consent from patients is essential to ensure transparency and respect for their rights.

7. **Data Accuracy and Integrity:**
   - Errors or inaccuracies in transmitted data can lead to incorrect diagnoses or treatment decisions.
   - Measures must be in place to ensure the accuracy and integrity of the data being transmitted.

8. **Interoperability Challenges:**
   - Ensuring that the service can communicate and integrate seamlessly with existing healthcare systems and devices is crucial.
   - Lack of interoperability can lead to data silos and inefficiencies in patient care.

Conclusion:

Sending a heart patient's data using a service can offer significant benefits in terms of remote monitoring, faster response times, and improved patient care. However, it also comes with inherent risks related to data security, privacy, regulatory compliance, and patient consent. To make this approach effective and ethical:

- **Implement Robust Security Measures:** Encryption, access controls, and secure protocols must be in place to protect patient data.
- **Ensure Regulatory Compliance:** Adhere to healthcare regulations and standards to avoid legal and financial repercussions.
- **Obtain Informed Patient Consent:** Patients should be aware of how their data will be used, shared, and protected.
- **Invest in Reliable Infrastructure:** A reliable service with minimal downtime is crucial for continuous monitoring.
- **Maintain Data Accuracy:** Regular checks and validations should be performed to ensure the integrity of transmitted data.
- **Respect Patient Privacy:** Safeguard patient confidentiality and privacy rights throughout the data transmission process.

By addressing these considerations, healthcare organizations can leverage the benefits of transmitting patient data while mitigating the associated risks. Collaborating with IT security experts, legal advisors, and patient advocates can help in designing a secure and ethical system for sending heart patient data using a service.

Q6. Suppose I write an SQL query to create schemas and retrieve data whenever an entry in the schema is updated. Is this a good approach and why/why not?

Writing an SQL query to create schemas and retrieve data whenever an entry in the schema is updated can be a valid approach in certain scenarios. However, whether it is a good approach or not depends on various factors, including the complexity of the schema, the frequency of updates, performance considerations, and the specific use case. Here are some considerations:

Pros:

1. **Real-time Data Retrieval:**
   - By creating a query that retrieves data whenever an entry is updated, you ensure that you always have the latest information.
   - This can be beneficial for applications where real-time or near-real-time data is required.

2. **Automated Updates:**
   - The SQL query can automate the process of retrieving updated data, eliminating the need for manual intervention.

   - This reduces the chances of errors and ensures data consistency.

3. **Simplicity and Ease of Implementation:**
   - For simple schemas and straightforward data retrieval needs, this approach can be quick and easy to implement.
   - It may involve a single SQL query or a simple trigger to achieve the desired functionality.

4. **Useful for Reporting or Monitoring:**
   - If you need to generate reports or monitor changes in the database in real-time, this approach provides a convenient solution.

Cons:

1. **Performance Impact:**
   - Continuous monitoring and retrieval of updated data can put a strain on the database, especially in high-traffic or high-update scenarios.
   - This approach might lead to increased server load and slower response times.

2. **Complexity with Large Schemas:**
   - For large and complex schemas with many tables and relationships, writing and maintaining the SQL query can become cumbersome.
   - Updates to the schema or changes in data retrieval requirements might require significant modifications to the query.

3. **Potential for Data Redundancy:**
   - If the SQL query retrieves the entire dataset every time an entry is updated, it could lead to redundant data transmission.
   - This is inefficient, especially when only a subset of the data has changed.

4. **Concurrency and Locking Issues:**
   - Continuous querying for updated data might lead to concurrency issues, such as locking conflicts.
   - If the query holds locks on tables for extended periods, it can affect the performance and concurrency of other transactions.

5. **Increased Network Traffic:**
   - Transmitting large datasets frequently over the network can lead to increased network traffic and potential bandwidth issues.

6. **Security Concerns:**
   - Depending on how the SQL query is implemented, it might expose sensitive data or open up security vulnerabilities.
   - Ensure that proper access controls and encryption are in place to protect the transmitted data.

7. **Scalability Challenges:**
   - As the database grows or the application scales, this approach might face challenges in handling increased data volumes and processing requirements.
   - Scaling up the database or the application infrastructure might be necessary.

8. **Alternative Approaches:**
   - **Change Data Capture (CDC):** Use CDC tools or features provided by the database management system to capture and replicate changes.
   - **Event-Driven Architecture:** Implement an event-driven system where database updates trigger events that are processed by subscribers.
   - **Batch Processing:** Instead of real-time retrieval, periodically retrieve updated data in batches to reduce the impact on the database.

Conclusion:
The decision to write an SQL query to create schemas and retrieve data whenever an entry is updated depends on the specific requirements, the size and complexity of the database, performance considerations, and scalability needs.

- **For Small, Simple Databases:** This approach can be suitable and straightforward to implement, providing real-time data retrieval without significant performance impact.
- **For Large, Complex Databases:** Careful consideration should be given to potential performance bottlenecks, data redundancy, concurrency issues, and scalability challenges.
- **Consider Alternatives:** Explore alternative approaches such as CDC, event-driven architecture, or batch processing for more efficient and scalable solutions.

Overall, while this approach can be effective in certain scenarios, it is essential to weigh the pros and cons, monitor its impact on performance, and ensure that it meets the specific requirements of the application and database architecture.

Q7. Suppose I utilize threads instead of co-routines to run a background task. Is
this a feasible idea, and is there any disadvantage of this approach.

Utilizing threads instead of coroutines to run background tasks is a feasible idea, especially in languages or environments where coroutines are not available or well-supported. However, there are some disadvantages and considerations to keep in mind when using threads for concurrency. Here are some points to consider:

Feasibility:

1. **Works in Most Programming Environments:**

   - Threads are a fundamental construct in many programming languages and platforms.
   - They provide a way to achieve concurrent execution and perform background tasks.

2. **Provides Concurrency:**
   - Using threads allows you to perform multiple tasks concurrently, which is useful for tasks like network I/O, file I/O, or heavy computations.

3. **Flexibility:**
   - Threads offer flexibility in terms of control over the execution of tasks.
   - You can manage thread creation, scheduling, and synchronization according to your application's needs.

Disadvantages and Considerations:

1. **Complexity and Overhead:**
   - Threads come with overhead in terms of memory and CPU usage.
   - Creating and managing threads can be complex, especially in scenarios with a large number of tasks.

2. **Resource Intensive:**
   - Each thread consumes system resources, such as stack memory and CPU time.
   - Running too many threads concurrently can lead to resource exhaustion and degraded performance.

3. **Synchronization and Deadlocks:**
   - When multiple threads access shared resources, you must implement synchronization mechanisms to prevent race conditions and data corruption.
   - Incorrect synchronization can lead to deadlocks, where threads are stuck waiting for each other indefinitely.

4. **Context Switching:**
   - Context switching between threads adds overhead and can impact performance, especially in scenarios with many threads competing for CPU time.

5. **Scalability Challenges:**
   - As the number of threads increases, the overhead of managing them also grows.
   - Scaling a thread-based application to handle a large number of concurrent tasks can be challenging.

6. **Difficulty in Error Handling:**
   - Debugging and error handling in threaded applications can be more complex than with sequential code.
   - Threaded code can lead to hard-to-reproduce bugs and issues.

7. **Lack of Structured Concurrency:**
   - Threads typically lack built-in support for structured concurrency, which makes it harder to manage the lifecycle of concurrent tasks.
   - This can lead to issues with task cancellation, resource leaks, or unintended task lifetimes.

8. **Blocking Nature:**
   - Traditional threads often rely on blocking I/O operations, which can lead to inefficiencies.
   - Blocked threads can't be used for other tasks, reducing the overall concurrency of the application.

9. **Performance Bottlenecks:**
   - In some cases, using threads for concurrency might not scale well with the number of CPU cores.
   - Task scheduling and contention for shared resources can create performance bottlenecks.

Alternatives and Modern Solutions:

1. **Coroutines:**
   - Coroutines provide a more lightweight and structured approach to concurrency.
   - They offer many advantages over traditional threads, such as better resource management, structured concurrency, and simplified error handling.

2. **Thread Pools:**
   - If using threads, consider using thread pools to manage and reuse threads efficiently.
   - Thread pools help reduce the overhead of thread creation and destruction.

3. **Asynchronous Programming:**
   - Modern languages and frameworks often provide asynchronous programming models.
   - Asynchronous tasks can be more efficient than traditional threading for I/O-bound operations.

4. **Event-Driven Architectures:**
   - Event-driven programming can provide concurrency without the need for explicit threading.
   - It utilizes callbacks or event handlers to respond to asynchronous events.

Conclusion:
Using threads for running background tasks is a feasible approach, especially in environments where coroutines or modern concurrency constructs are not available. However, it comes with its set of challenges, including complexity, resource consumption, synchronization issues, and scalability concerns.

If you choose to use threads, consider using best practices such as thread pools, proper synchronization mechanisms, and careful resource management. Alternatively, explore modern solutions like coroutines, asynchronous programming, or event-driven architectures, which offer more efficient and structured ways to achieve concurrency with fewer downsides. Always consider the specific requirements, constraints, and characteristics of your application when deciding on the concurrency model to use.

https://www.howtodoandroid.com/jetpack-compose-retrofit-recyclerview/

https://www.geeksforgeeks.org/android-json-parsing-using-retrofit-library-with-jetpack-compose/

https://medium.com/@Night_Goat/mastering-kotlinpoet-for-android-jetpack-compose-5ddccccbf232#:~:text=If%20you%20want%20to%20efficiently,JSON%20data%20into%20structured%20objects

https://www.youtube.com/watch?v=6_wK_Ud8--0

```
package com.example.my

import android.os.Bundle
import android.widget.ProgressBar
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.material3.Button
import androidx.compose.material3.LinearProgressIndicator
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.example.my.ui.theme.MyTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyTheme {
                var count by remember { mutableStateOf(0) }
                val listState = rememberLazyListState()
                val limitedCount = count.coerceIn(0, 20)
                var isDistanceInKilometers by remember { mutableStateOf(true) }
                var distance = if (isDistanceInKilometers) count * 10 else (count * 10) * 0.621371
                var distanceCovered = if (isDistanceInKilometers) limitedCount * 10 else (limitedCount
* 10) * 0.621371
                val distanceRemaining = if (isDistanceInKilometers) (20 - limitedCount) * 10 else ((20
- limitedCount) * 10) * 0.621371
                val listItems = listOf(
                    "  Destination 1",
                    "\n",
                    "  Destination 2",
                    "\n",
                    "  Destination 3",
                    "\n",
                    "  Destination 4",
                    "\n",
                    "  Destination 5",
                    "\n",
                    "  Destination 6",
                    "\n",
                    "  Destination 7",
                    "\n",
                    "  Destination 8",
                    "\n",
                    "  Destination 9",
                    "\n",
                    "  Destination 10",
                    "\n",
                    "  Destination 11",
                    "\n",
                    "  Destination 12",
                    "\n",
                    "  Destination 13",
                    "\n",
                    "  Destination 14",
                    "\n",
                    "  Destination 15",
                    "\n",
                    "  Destination 16",
                    "\n",
                    "  Destination 17",
                    "\n",
                    "  Destination 18",
                    "\n",
                    "  Destination 19",
                    "\n",
                    "  Destination 20",
                    "\n",
                    "\n"
                )
                Column {
                    Row {
                        Button(
                            onClick = {

                                if (count == listItems.size - 21){
                                   distance = 0
                                }else(count++)
                                 },
                            modifier = Modifier.padding(8.dp)
                        ) {
                            Text("Next stop")
                        }

                    }
                    Spacer(modifier = Modifier.height(16.dp))


                    Column {
```

```
                    Text("  Distance covered: ${distanceCovered} ${if (isDistanceInKilometers) "km"
else "miles"}")
                    Text(
                        text = "Next is in ${distance} ${if (isDistanceInKilometers) "km" else "miles"}",
                        modifier = Modifier.padding(8.dp),

                        )
                    Text("  Distance remaining: ${distanceRemaining} ${if (isDistanceInKilometers)
"km" else "miles"}")

                    Button(
                        onClick = {
                            isDistanceInKilometers = !isDistanceInKilometers
                        },
                        modifier = Modifier.padding(8.dp)
                    ) {
                        Text("Convert to ${if (isDistanceInKilometers) "Miles" else "Kilometers"}")
                    }
                }
                Spacer(modifier = Modifier.height(16.dp))
                ProgressforDistance(limitedCount)

                Spacer(modifier = Modifier.height(16.dp))

                LazyColumn {
                    items(listItems) { item ->
                        Text(item)

                    }
                }
            }
        }
    }

}



@Composable
fun ProgressforDistance(limitedCount: Int) {
    val progress = limitedCount / 20f
    val percentage = (progress * 100).toInt()
    val cappedProgress = progress.coerceIn(0f, 1f)
    Column {
            LinearProgressIndicator(
                progress = progress,
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(horizontal = 20.dp)
                    .height(12.dp)
            )
            val percentage = (cappedProgress * 100).toInt()
            Text(
                text = "Covered: $percentage%",
                modifier = Modifier.padding(top = 4.dp, start = 20.dp)
            )
        }
    }
}
```