|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Firebase | <chare> | 1 | <pext> | Firebase is a product of Google which helps developers to build,  manage, and grow their apps easily.  It helps developers to build their apps faster and in a more secure way.  No programming is required on the firebase side  which makes it easy to use its features more efficiently.  It provides services to android, ios, web, and unity.  It provides cloud storage.  It uses NoSQL for the database for the storage of data.  -------Firebase Features----------  1. Authentication  2. Realtime Database  3. Cloud storage  4. Crash Analytics  5. Analytics  6. Testing for release apps  ---------Firebase pros-------------  1. Real-time updating of data  2. Offline availability  3. Centralized database for all connected devices  4. Easy understanding  5. Cloud messaging | </end> |
| <hitle> | Firebase Authentication | <chare> | 2 | <pext> | Firebase Authentication provides backend services, easy-to-use SDKs,  and ready-made UI libraries to authenticate users to your app.  It supports authentication using passwords, phone numbers,  popular federated identity providers like Google, Facebook and Twitter, and more.  --------Register/Signup/Create user email password-----------  We can use createUserWithEmailAndPassword() method.  This method takes email and password as a parameter,  validates them and then create a new user.  private lateinit var auth: FirebaseAuth  auth = FirebaseAuth.getInstance()  auth.createUserWithEmailAndPassword(email, password)  .addOnCompleteListener(this, OnCompleteListener{ task ->  if(task.isSuccessful){  Toast.makeText(this, "Successfully Registered", Toast.LENGTH\_LONG).show()  val intent = Intent(this, MainActivity::class.java)  startActivity(intent)  finish()  }else {  Toast.makeText(this, "Registration Failed", Toast.LENGTH\_LONG).show()  }  })  User will be registered and you can see the registered user  on the Firebase website under the Authentication section  and then in the Users tab.  If you are trying to register with the same email more than once,  then you will receive an error or in simple words your task is unsuccessful.  -----------Login/Signin user email password--------  We can use signInWithEmailAndPassword().  This method takes email and password as a parameter,  validates them and then signin a user in your application  if the validation is successful.  private lateinit var auth: FirebaseAuth  auth = FirebaseAuth.getInstance()  auth.signInWithEmailAndPassword(email, password)  .addOnCompleteListener(this, OnCompleteListener { task ->  if(task.isSuccessful) {  Toast.makeText(this, "Successfully Logged In", Toast.LENGTH\_LONG).show()  val intent = Intent(this, MainActivity::class.java)  startActivity(intent)  finish()  }else {  Toast.makeText(this, "Login Failed", Toast.LENGTH\_LONG).show()  }  })  ------------forgot/change/reset password link-------------  We can use the sendPasswordResetEmail()  to send the reset email to the user.  This method takes email as a parameter.  private lateinit var auth: FirebaseAuth  auth = FirebaseAuth.getInstance()  auth.sendPasswordResetEmail(email)  .addOnCompleteListener(this, OnCompleteListener { task ->  if (task.isSuccessful) {  Toast.makeText(this, "Reset link sent to your email", Toast.LENGTH\_LONG)  .show()  } else {  Toast.makeText(this, "Unable to send reset mail", Toast.LENGTH\_LONG)  .show()  }  })  ---------------Check user logged/signed/login status/state---------------  We can use the getCurrentUser() method to get the status of a user.  if(auth.currentUser == null){  val intent = Intent(this, LoginActivity::class.java)  startActivity(intent)  finish()  }else{  Toast.makeText(this, "Already logged in", Toast.LENGTH\_LONG).show()  }  ------------ updatePassword------------  We can update the password of a user by using the updatePassword().  auth.currentUser?.updatePassword(password)  ?.addOnCompleteListener(this, OnCompleteListener { task ->  if (task.isSuccessful) {  Toast.makeText(this, "Password changes successfully", Toast.LENGTH\_LONG)  .show()  finish()  } else {  Toast.makeText(this, "password not changed", Toast.LENGTH\_LONG)  .show()  }  })  --------------updateEmail()-------------  We can update the Email of a user.  -------------Logout/signOut user--------------  FirebaseAuth.getInstance().signOut() | </end> |
| <hitle> | Phone Authentication | <chare> | 3 | <pext> | https://www.ericthecoder.com/2019/03/18/2-factor-authentication-email-password-phone-with-firebase-authentication-on-android/ | </end> |
| <hitle> | multiple auth providers | <chare> | 3 | <pext> | https://en.proft.me/2018/02/6/how-link-multiple-auth-providers-firebase-android/ | </end> |
| <hitle> | Firebase Analytics | <chare> | 2 | <pext> | Firebase Analytics is a free app measurement solution that provides insight on app usage and user engagement, which means that we will have a lot of data to make decisions regarding behaviors, patterns and demography among others when users use our app.  -------Firebase Analytics howto/walkthrough-------  1. Add Dependency in app’s build.gradle  2. Make FirebaseAnalytics object.  private lateinit var firebaseAnalytics: FirebaseAnalytics  firebaseAnalytics = FirebaseAnalytics.getInstance(context)  3. Log the events we’d need to track.  LogEvent method receives Key-Value pair as params.  val bundle = Bundle()  bundle.putString(FirebaseAnalytics.Param.ITEM\_NAME, tvShow.name)  firebaseAnalytics?.logEvent(FirebaseAnalytics.Event.LOGIN.name(), bundle) | </end> |
| <hitle> | Firebase Crashlytics | <chare> | 2 | <pext> | Firebase Crashlytics is a mobile application crash-reporting tool  that provides you with real-time insights into the stability of your app.  It helps you identify and prioritize issues based on the frequency  and severity of crashes, allowing you to focus your efforts  on the most critical problems.  With Firebase Crashlytics, you can receive crash reports directly  in your Firebase console or through email alerts,  allowing you to quickly identify and resolve issues.  You can also track the performance of specific app versions and user groups,  enabling you to identify trends and patterns in crashes.  ------------Firebase Crashlytics internal work--------  Firebase Crashlytics SDK automatically detects and reports crashes,  even if your app has gone offline or if the crash occurred  outside of your app’s main thread.  When a crash occurs, Firebase Crashlytics collects important information  about the crash, including the stack trace, device information,  and app state at the time of the crash.  It then sends this data to the Firebase servers,  where it is processed and analyzed.  -------howto walkthrough Firebase Crashlytics-------  Step 1: Create a Firebase project,  and connect the android project to firebase project  and add dependency to use library.  Step 2: Initialize Firebase Crashlytics:  FirebaseCrashlytics.getInstance().setCrashlyticsCollectionEnabled(true)  Use this in Application class.  class MyApplication : Application() {  override fun onCreate() {  super.onCreate()  FirebaseCrashlytics.getInstance().setCrashlyticsCollectionEnabled(true)  }  }  Step 3: Test Firebase Crashlytics:  FirebaseCrashlytics.getInstance().crash()  To test Firebase Crashlytics, we can force a crash in your app.  Step 4: Review crash reports:  Once we done the integration in android project,  we can view crash reports in the Firebase Console.  Firebase Console/My project/click “Crashlytics” | </end> |
| <hitle> | Firebase Realtime Database | <chare> | 2 | <pext> | Firebase Realtime Database is a solution that stores data in the cloud  and provides an easy way to sync your data among various devices.  Realtime means that any changes in data are reflected immediately  across all platforms and devices within milliseconds.  Realtime Database uses data synchronization and subscriber mechanisms  instead of typical HTTP requests, which allows you to build  more flexible real-time apps, easily, with less effort and without the need  to worry about networking code.  --------------Firebase Database Offline support--------  Realtime Database provides great offline support  because it keeps an internal cache of all the data you’ve queried.  When there’s no Internet connection,  the app uses the data from the cache, allowing apps to remain responsive.  When the device connects to the Internet,  the Realtime Database synchronizes the local data changes  with the remote updates that occurred while the client was offline,  resolving any conflicts automatically.  --------------Firebase Database Rule--------  Client devices access the Realtime Database directly,  without the need for an application server.  Security rules take care of who has access to what data,  and how they can access it.  1. Default  // These rules don't allow anyone read or write access to your database  {  "rules": {  ".read": false,  ".write": false  }  }  2. Public  // These rules give anyone, even people who are not users of your app,  // read and write access to your database  {  "rules": {  ".read": true,  ".write": true  }  }  3. User  // These rules grant access to a node matching the authenticated  // user's ID from the Firebase auth token  {  "rules": {  "users": {  "$uid": {  ".read": "$uid === auth.uid",  ".write": "$uid === auth.uid"  }  }  }  }  ------------howo/walkthrough Firebase Realtime Database--------  1. We need to connect the project to Firebase Console.  2. Creating Model/Data class  @IgnoreExtraProperties  data class UserInfo(  var name: String? = "",  var mobile: String? = ""  )  3. Get a Database Reference  private lateinit var dbReference: DatabaseReference  dbReference = FirebaseDatabase.getReference("users")  4. Read and Write Data by setValue()  private fun createUser(name: String, mobile: String) {  val user = UserInfo(name, mobile)  dbReference.child(userId).setValue(user)  }  The setValue() method, overwrite the data that is present with the new data.  Instead of updating the entire object, we can update a particular child.  dbReference.child("users").child(userId).setValue(user)  dbReference.child("users").child(userId).child("name").setValue(name)  5. To listen for changes in the database,  we can use the addValueEventListener() method.  onDataChange() will be called when there is a change in the data.  If the application is unable to read data from the database,  then the onCancelled() method will be called.  private fun addUserChangeListener() {  // User data change listener  dbReference.child(userId).addValueEventListener(object : ValueEventListener {  override fun onDataChange(dataSnapshot: DataSnapshot) {  val user = dataSnapshot.getValue(UserInfo::class.java)  // Check for null  if (user == null) {  return  }  // Display newly updated name and email  userNameTv.setText(user?.name).toString()  userMobileTv.setText(user?.mobile).toString()  // clear edit text  userNameEt.setText("")  userMobileEt.setText("")  }  override fun onCancelled(error: DatabaseError) {  // Failed to read value  }  })  }  6. Delete a data  To delete a data from a specific location,  you can use the removeVale() method on a reference  to the location of that data. | </end> |
| <hitle> | FCM Firebase Cloud Messaging | <chare> | 2 | <pext> | Firebase Cloud Messaging (FCM) is a set of tools  that sends push notifications and small messages of up to 4 KB  to different platforms: Android, iOS and web.  The article:  <https://www.kodeco.com/9227276-firebase-cloud-messaging-for-android-sending-push-notifications>  -----FCM architecture main parts-------  1. A service, API or console that sends messages to targeted devices.  3. The Firebase Cloud Messaging back end,  where all the processing happens.  3. A transport layer that’s specific to each platform.  In Android’s case, this is called the Android Transport Layer.  3. The SDK on the device where you’ll receive the messages.  In this case, called the Android Firebase Cloud Messaging SDK.  ---------4KB Max size of notifications FCM data playload-----------  up to 4 KB  ------------ default expiry period of FCM notificaition----------  The default expiration for push notifications without an expiration date is 30 days for APNs and 28 days for FCM.  -----------FCM Topic usecase---------  Based on the publish/subscribe model,  FCM topic messaging allows you to send a message to multiple devices  that have optioned in to a particular topic.  You compose topic messages as needed,  and FCM handles routing and delivering the message  reliably to the right devices.  For example, users of a local weather forecasting app  could option in to a “severe weather alerts” topic  and receive notifications of storms threatening specified areas.  Users of a sports app could subscribe to automatic updates  in live game scores for their favorite teams.  Important things of FCM topics:  1. Topic messaging supports unlimited topics  and subscriptions for each app.  2. Topic messaging is best suited for content  such as news, weather, or other publicly available information.  3. Topic messages are optimized for throughput rather than latency.  For fast, secure delivery to single devices or small groups of devices,  target messages to registration tokens, not topics.  4. If you need to send messages to multiple devices per user,  consider device group messaging for those use cases.  -----------FCM subscribeTopic()----------  fun subscribeTopic(context : Context,topic : String)  This method is used to Subscribe a device to “topic”  which is passed in a parameter of the method.  fun subscribeTopic(context: Context, topic: String) {  FirebaseMessaging.getInstance().subscribeToTopic(topic).addOnSuccessListener {  Toast.makeText(context, "Subscribed $topic", Toast.LENGTH\_LONG).show()  }.addOnFailureListener {  Toast.makeText(context, "Failed to Subscribe $topic", Toast.LENGTH\_LONG).show()  }  }  -----------FCM unsubscribeTopic()----------  fun unsubscribeTopic(context : Context,topic : String)  This method is used to Unsubscribe a device from a “topic”  that is passed in a parameter of the method.  fun unsubscribeTopic(context: Context, topic: String) {  FirebaseMessaging.getInstance().unsubscribeFromTopic(topic).addOnSuccessListener {  Toast.makeText(context, "Unsubscribed $topic", Toast.LENGTH\_LONG).show()  }.addOnFailureListener {  Toast.makeText(context, "Failed to Unsubscribe $topic", Toast.LENGTH\_LONG).show()  }  }  ---------Howto Notification FCM sendMessage()--------------  fun sendMessage(title: String, content: String,topic: String)  This method is used to send Messages as Push Notifications  to devices that are subscribed to a particular topic.  1. It uses HttpUrlConnection internally to trigger an API.  2. For security purposes, it requires “authorization” in its header  which contains the server key.  3. Set the title and content in the “data” of payload  which will be sent as a Push Notification.  4. Add “topic” under the payload to make it a valid payload  or else you will get a 400 Bad request error.  5. Once the above steps are done, HttpUrlConnection will do its work  and return 200 in response code on success.  fun sendMessage(title: String, content: String,topic: String) {  GlobalScope.launch {  val endpoint = "https://fcm.googleapis.com/fcm/send"  try {  val url = URL(endpoint)  val httpsURLConnection: HttpsURLConnection = url.openConnection() as HttpsURLConnection  httpsURLConnection.readTimeout = 10000  httpsURLConnection.connectTimeout = 15000  httpsURLConnection.requestMethod = "POST"  httpsURLConnection.doInput = true  httpsURLConnection.doOutput = true  // Adding the necessary headers  httpsURLConnection.setRequestProperty("authorization", "key=$key")  httpsURLConnection.setRequestProperty("Content-Type", "application/json")  // Creating the JSON with post params  val body = JSONObject()  val data = JSONObject()  data.put("title", title)  data.put("content", content)  body.put("data",data)  body.put("to","/topics/$topic")  val outputStream: OutputStream = BufferedOutputStream(httpsURLConnection.outputStream)  val writer = BufferedWriter(OutputStreamWriter(outputStream, "utf-8"))  writer.write(body.toString())  writer.flush()  writer.close()  outputStream.close()  val responseCode: Int = httpsURLConnection.responseCode  val responseMessage: String = httpsURLConnection.responseMessage  Log.d("Response:", "$responseCode $responseMessage")  var result = String()  var inputStream: InputStream? = null  inputStream = if (responseCode in 400..499) {  httpsURLConnection.errorStream  } else {  httpsURLConnection.inputStream  }  if (responseCode == 200) {  Log.e("Success:", "notification sent $title \n $content")  // The details of the user can be obtained from the result variable in JSON format  } else {  Log.e("Error", "Error Response")  }  } catch (e: Exception) {  e.printStackTrace()  }  }  }  Article:  <https://proandroiddev.com/send-topic-based-push-notification-in-android-b47d1ff6d764>  ----------FCM Notification onMessageReceived()-----------  override fun onMessageReceived(p0: RemoteMessage)  This method will be triggered whenever a device receives  a Push Notification from the topic for which it is subscribed.  override fun onMessageReceived(p0: RemoteMessage) {  super.onMessageReceived(p0)  Log.e("onMessageReceived: ", p0.data.toString())  val title = p0.data.get("title")  val content = p0.data.get("content")  val defaultSound = RingtoneManager.getDefaultUri(RingtoneManager.TYPE\_NOTIFICATION)  val intent = Intent(this,MainActivity::class.java)  intent.flags = Intent.FLAG\_ACTIVITY\_CLEAR\_TOP or Intent.FLAG\_ACTIVITY\_SINGLE\_TOP  val pendingIntent = PendingIntent.getActivity(applicationContext,0,  intent,  PendingIntent.FLAG\_UPDATE\_CURRENT)  if (android.os.Build.VERSION.SDK\_INT >= android.os.Build.VERSION\_CODES.O) {  checkNotificationChannel("1")  }  // val person = Person.Builder().setName("test").build()  val notification = NotificationCompat.Builder(applicationContext,"1")  .setSmallIcon(R.mipmap.ic\_launcher)  .setContentTitle(title)  .setContentText(content)  // .setStyle(NotificationCompat.MessagingStyle(person)  // .setGroupConversation(false)  // .addMessage(title,  // currentTimeMillis(), person)  // )  .setContentIntent(pendingIntent)  .setPriority(NotificationCompat.PRIORITY\_HIGH)  .setSound(defaultSound)  val notificationManager : NotificationManager =  getSystemService(Context.NOTIFICATION\_SERVICE) as NotificationManager  notificationManager.notify(1,notification.build())  }  ----------FCM FirebaseMessagingService Manifest---------  Make sure to register MyFirebaseMessagingService.kt  in AndroidManifest.xml under a <service> to receive Push Notification  if your application is in the background or killed state.  <service android:name=”.MyFirebaseMessagingService”>  <intent-filter>  <action android:name=”com.google.firebase.MESSAGING\_EVENT” />  </intent-filter>  </service> | </end> |
| <hitle> | Retrofit | <chare> | 1 | <pext> | Retrofit is a type-safe HTTP networking library used for Android  built by Square and maintained by Google.  It is the most used and most efficient library  for network requests in applications.  It is Annotation based library.  Retrofit was even better since it was super fast,  offered better functionality, and even simpler syntax.  -----------Retrofit pros-----------------  1. It is very fast and easy to use as it is annotation based library.  2. It supports request cancellation.  3. It supports post requests and multipart uploads.  4. It enables direct communication with the web service.  5. Supports dynamic URLs.  6. Supports convertors.  7. It supports both synchronous and asynchronous network requests.  8. It caches responses to avoid sending duplicate requests.  -----------Retrofit cons-----------------  1. It does not support setting priorities.  2. Retrofit is not having retry policy concept inbuild as Volley have.  3. Retrofit is not having request prioritization inbuild.  4. It does not support image loading.  It requires other libraries such as Picasso and Glide.  -----------Retrofit Annotations-------------  Retrofit provides an annotation  for each of the main standard request methods.  You simply use the appropriate Retrofit annotations  for each HTTP method  1. @GET : It will allow to fetch available data request.  2. @POST: It will allow to post a data to server as request.  3. @PUT: It will allow to update a existing data  to server as a whole request as body object.  4. @PATCH: It will allow to make an API request  to update a particular field in exsiting data on server database.  5. @DELETE: It will allow to make a delete API request. | </end> |
| <hitle> | Retrofit Interceptor | <chare> | 2 | <pext> | interceptors are a powerful concept that can monitor, rewrite,  authenticate, cache and retry the API call.  So basically, when we request API call, we can monitor the call  or perform some tasks or set of interceptors in one call is also allowed.  ------------- Interceptor Types ----------  Application Interceptors, Network Interceptors  -----------Application Interceptors---------  1. Application Interceptors  Interceptors added between the Application Code  and the OkHttp Core Library are referred to as application interceptors.  These are the interceptors that we add with addInterceptor().  ------------Network Interceptors----------  2. Network Interceptors  Interceptors on the network:  These are interceptors placed between the OkHttp Core Library and the server.  These can be added to OkHttpClient by using the addNetworkInterceptor().  ---------- interceptors use-cases--------  Logging interceptor, Retry interceptor, API Key Interceptor, Cache interceptor,  Header via Interceptor,  ----------Logging interceptor-------  In Retrofit 2, all network operations are performed via OkHttp library.  OkHttp provides HttpLoggingInterceptor which logs HTTP request  and response data.  HttpLoggingInterceptor has 4 levels of logging:  BASIC – Logs request and response lines.  BODY – Logs request and response lines  and their respective headers and bodies (if present). This is the only log level where we will get the response body data.  HEADERS – Logs request and response lines and their respective headers.  NONE – No logs. Use this log level for production environments  to enhance the apps performance.  val loggingInterceptor = HttpLoggingInterceptor().apply {  level = HttpLoggingInterceptor.Level.BODY  }  ------------ Retry interceptor---------  Retry intercepter is relly useful when API’s are interminent  in giving response also when bandwidth is low  and API is taking much time to give the response.  class RetryInterceptor(private val retryAttempts: Int) : Interceptor {  override fun intercept(chain: Interceptor.Chain): Response {  for (i in 1..retryAttempts) {  try {  return chain.proceed(chain.request())  } catch (e: SocketTimeoutException) {  e.printStackTrace()  }  }  throw RuntimeException("failed to compile the request")  }  }  ----------- API Key Interceptor---------------  API key interceptors are really needed one  for most of the realtime projects where you need to add one API Key  to make the request so rather than making it many times into many calls  have same duplicated code to add a API key we should use an interceptor  and enjoy it in all requests via OKHttp client.  class APIKeyInterceptor : Interceptor {  override fun intercept(chain: Interceptor.Chain): Response {  val currentUrl = chain.request().url  val newUrl = currentUrl.newBuilder().addQueryParameter("api\_key", "api\_key").build()  val currentRequest = chain.request().newBuilder()  val newRequest = currentRequest.url(newUrl).build()  return chain.proceed(newRequest)  }  }  ---------- Cache interceptor---------  If we want to cache the response of the API call  so that if we call the API again, the response comes out from Cache.  Let’s say we have the API call from Client to Server  and Cache-Control header is enabled from the server,  then OkHttp Core will respect that header and cache the response  for a certain time being which was sent from the server.  But what if the Cache-Control is not enabled from the server.  We still can cache the response from OkHttp Client using Interceptor.  class CacheInterceptor: Interceptor {  override fun intercept(chain: Interceptor.Chain): Response {  val originalResponse = chain.proceed(chain.request())  return if (Utils.isNetworkAvailable(applicationContext)) {  val maxAge = 60  originalResponse.newBuilder()  .addHeader("Cache-control", "public, max-age = $maxAge")  .build()  } else {  val maxStale = 60 \* 60 \* 24 \* 28 // 4 weeks  originalResponse.newBuilder()  .addHeader("Cache-control", "public, only-if-cache max-age = $maxStale")  .build()  }  }  }  ----------- Header via Interceptor----------  Let’s say that we have to make the API calls and we have to add  Authorization Header in all the API calls.  Either we can use it individually or we can centralize that using the Interceptor.  class AuthInterceptor: Interceptor {  override fun intercept(chain: Interceptor.Chain): Response {  val currentRequest = chain.request().newBuilder()  currentRequest.addHeader(AUTHORIZATION, TOKEN)  val newRequest = currentRequest.build()  return chain.proceed(newRequest)  }  }  -----------enqueue() vs execute()-------------  1. Enqueue() is Asynchronous calls which makes non-blocking UI,  In this way, android executes the request in a separate thread  and does not block the main thread.  2. Execute() is Synchronous calls, and the request is blocked  and the response is available immediately to the next statement,  and executed in the main thread blocking UI.  -----------cancel retrofit------------  We can use call.cancel() method or we can use CancellableCallback.  if (some condition && callback != null) {  callback.cancel();  } | </end> |
| <hitle> | Retrofit Async main/backround thread | <chare> | 2 | <pext> | Retrofit methods can be declared for either synchronous  or asynchronous execution.  A method with a return type will be executed synchronously.  @GET("/user/{id}/photo")  fun getUserPhoto(@Path("id") int id): Photo  Asynchronous execution requires  the last parameter of the method be a Callback.  @GET("/user/{id}/photo")  fun getUserPhoto(@Path("id") int id, Callback<Photo> cb)  On Android, callbacks will be executed on the main thread.  Retrofit also integrates RxJava to support methods  with a return type of rx.Observable  @GET("/user/{id}/photo")  Observable<Photo> getUserPhoto(@Path("id") int id);  Observable requests are subscribed asynchronously and observed  on the same thread that executed the HTTP request.  To observe on a different thread we have to call observeOn(Scheduler)  on the returned Observable. | </end> |
| <hitle> | Retrofit vs Volley | <chare> | 1 | <pext> | 1. Code complexity  Retrofit treats the Api calls as simple java method calls  and it also handle the Json/Xml parsing itself,  while with Volley we have to do some extra code  to achieve the same for parsing like need to create a custom request.  2. Image loading  Retrofit does not come with support for loading images  from the network by itself.  Volley has basic support for image loading.  3. Caching  Retrofit: caching should “just work”  if your server sets the correct cache control headers.  Volley definitely has a flexible caching mechanism.  4. Request prioritization, cancellation and retrying  Retrofit: does not support setting priority, supports cancellation;  supports manual request retries.  Volley: supports prioritization; supports cancellation  and customizable retry with exponential backoff out-of-the-box.  5. POST requests + multipart uploads  Retrofit has full support for POST requests and multi part file uploads.  Volley supports POST requests but you’ll have to convert  your Java objects to JSONObjects yourself,  also supports multi part requests  but you need to add additional classes or equivalent. | </end> |
| <hitle> | ConverterFactory | <chare> | 1 | <pext> | If your application is Restful, so gets and sends data from or to server,  Converter factory need to be added,  just for retrofit can convert JSON data into java objects like POJOs,  to use in Android Project.  There are some converter libraries for converting JSON to Java objects,  Like GSON, Moshi, etc. | </end> |
| <hitle> | Koin | <chare> | 1 | <pext> | <starting>Koin</heading> Koin is a pragmatic and lightweight dependency injection framework  for Kotlin developers. Koin is a DSL, a light container and a pragmatic API. Koin provides a DSL to help your describe your app  instead of annotate it or generate code for it. With its Kotlin DSL, Koin offers a smart functional API  to achieve to prepare your dependency injection.  startKoin: means run a KoinApplication instance into the GlobalContext.  A Koin module will inject/combine for your application.  module - create a Koin Module To describe your content in a module,  you can use the following functions: factory - provide a factory bean definition single - provide a singleton bean definition (also aliased as bean) get() - resolve a component dependency (also can use name, scope or parameters) bind() - add type to bind for given bean definition binds() - add types array for given bean definition scope - define a logical group for scoped definition scoped - provide a bean definition that will exists only in a scope </ending> | </end> |
| <hitle> | Moshi | <chare> | 1 | <pext> | <starting>Moshi</heading> Moshi is a modern JSON library for Android, Java and Kotlin. It makes it easy to parse JSON into Java and Kotlin classes </ending> | </end> |
| <hitle> | OkHttp | <chare> | 1 | <pext> | <starting>OkHttp</heading> OkHttp is an efficient HTTP & HTTP/2 client for Android and Java applications.  It comes with advanced features, such as connection pooling  (if HTTP/2 isn't available), transparent GZIP compression,  and response caching, to avoid the network completely for repeated requests. OkHTTP is an open source project designed to be an efficient HTTP client. It supports the SPDY protocol. SPDY is the basis for HTTP 2.0 and allows multiple HTTP requests  to be multiplexed over one socket connection. </ending> | </end> |
| <hitle> | Glide | <chare> | 1 | <pext> | <starting>Glide</heading> Glide, like Picasso, can load and display images from many sources,  while also taking care of caching and keeping a low memory impact  when doing image manipulations.  It provides animated GIF support and handles image loading/caching.  Animated GIF support is currently not implemented in Picasso. Yes, images play a major role in making the UI of an App more interactive  and user-friendly too. So, as an Android Developer, we should take care of using images in App. We should handle the different aspects of an image  like the slow unresponsive image, memory issues, loading errors,  and many more. If you are not handling these aspects in your application,  then your app will make a bad impression on your users. </ending> | </end> |
| <hitle> | Guice | <chare> | 1 | <pext> | <starting>Guice</heading> Guice is an open-source software framework for the Java platform  released by Google under the Apache License. It provides support for dependency injection using annotations  to configure Java objects. Guice allows implementation classes  to be bound programmatically to an interface,  then injected into constructors, methods or fields  using an @Inject annotation. When more than one implementation of the same interface is needed,  the user can create custom annotations that identify an implementation,  then use that annotation when injecting it. </ending> | </end> |
| <hitle> | Jenkins | <chare> | 1 | <pext> | <starting>Jenkins</heading> Jenkins is an open-source free automation tool  used to build and test software projects. The tool makes it painless for developers to integrate changes to the project.  Jenkins' primary focus is to keep track of the version control system  and initiate and monitor a build system if there are any changes. It keeps an eye on the entire process  and provides reports and notifications to alert. </ending> | </end> |
| <hitle> | DevOps | <chare> | 1 | <pext> | <starting>DevOps</heading> DevOps is a software engineering approach or culture  that unifies the Development and Operations team  under an umbrella of tools to automate every stage. The Development team is responsible for coding or scripting the software  and integrating all the components needed to run that software. Meanwhile, the Operations team is responsible  for managing the organization's diverse workforce  to ensure effective and efficient daily operations. </ending> | </end> |
| <hitle> | Continuous Integration (CI) | <chare> | 1 | <pext> | <starting>Continuous Integration (CI)</heading> Continuous Integration is a practice  that integrates code into a shared repository. It uses automated verifications for the early detection of problems. Continuous Integration doesn't eliminate bugs  but helps in finding and removing them quickly. </ending> | </end> |
| <hitle> | Continuous Delivery (CD) | <chare> | 1 | <pext> | <starting>Continuous Delivery (CD)</heading> Continuous Delivery is the phase  where the changes are made in the code before deploying. The team in this phase decides  what is to be deployed to the customers and when. The final goal of the pipeline is to make deployments. </ending> | </end> |
| <hitle> | Groovy Jenkins | <chare> | 1 | <pext> | <starting>Groovy Jenkins</heading> 1. Apache Groovy is a dynamic object-oriented programming language  used as a scripting language for Java platforms.  2. Groovy is used to orchestrate the Jenkins pipeline  and enables different teams to contribute to the work in different languages.  3. Groovy's syntax is very similar to that of Java,  making it more seamless with the Java interface.  4. The language has several features  like Java compatibility and Development support. </ending> | </end> |
| <hitle> | CI/CD pipeline | <chare> | 1 | <pext> | <starting>CI/CD pipeline</heading> CI/CD Pipeline or Continuous Integration/ Continuous Delivery  is considered the DevOps approach's backbone. The pipeline is responsible  for building codes, running tests, and deploying new software versions. </ending> | </end> |
| <hitle> | Continuous Deployment (CD) | <chare> | 1 | <pext> | <starting>Continuous Deployment (CD)</heading> Continuous Deployment (CD) is the ultimate stage in the DevOps pipeline. It refers to automatic release of any developer changes from the repository  to the production stage.  </ending> | </end> |
| <hitle> | Jenkins pipeline | <chare> | 1 | <pext> | <starting>Jenkins pipeline</heading> 1. The pipeline represents the continuous delivery and continuous integration  of all the jobs in the SDLC and DevOps life cycle.  2. The Jenkins pipeline is a set of plugins  that support implementation and integration  of continuous delivery pipelines into Jenkins. It connects this pipeline in a particular format by Jenkins. 3. The Jenkins pipeline solves several problems  like the maintenance of thousands of jobs and maintaining deployment  with needing to resort to other powerful methods. </ending> | </end> |
| <hitle> | Jenkins plugins | <chare> | 1 | <pext> | <starting>Jenkins plugins</heading> 1. Maven 2 project 2. Amazon EC2 3. Copy artifact 4. Join 5. HTML publisher 6. Green Balls </ending> | </end> |
| <hitle> | Jenkins process | <chare> | 1 | <pext> | <starting>Jenkins process</heading> 1. Jenkins checks changes in repositories regularly,  and developers must secure their code regularly.  2. Once the changes are defined,  Jenkins detects them and uses them to prepare a new build. 3. After that,  Jenkins will transverse through various stages in its usual pipeline. As one stage completes, the process will move further on to the next stage. 4. If a stage fails, the Jenkins build will stop there,  and the software will email the team using it. When completed successfully,  the code implements itself in the proper server so that testing begins. 5. After the successful testing phase,  Jenkins shares the results with the team using it. </ending> | </end> |
| <hitle> | Jenkins file | <chare> | 1 | <pext> | <starting>Jenkins file</heading> Jenkins file is a text file that has a definition of a Jenkins pipeline  and is checked into the source control repository. It enables code review and iteration on the pipeline. It also permits an audit trail for the pipeline. </ending> | </end> |
| <hitle> | Jenkins DSL | <chare> | 1 | <pext> | <starting>Jenkins DSL</heading> DSL stands for Domain Specific Language. Jenkins job DSL is a plugin that allows us  to define jobs in the programmatic form with minimal effort. You can describe your jobs in Jenkins using a Groovy Based Language. They designed Jenkins job DSL plugin to create versions of the job,  manage the records. </ending> | </end> |
| <hitle> | Jenkins environmental variables | <chare> | 1 | <pext> | <starting>Jenkins environmental variables</heading> $JOB\_NAME - The name that you give your job when it is first set up. $NODE\_NAME - This is the name of the node  on which the current build is running. $WORKSPACE - Refers to the path of the workspace $BUILD\_URL - Indicates the URL where the results of the builds can be found. $JENKINS\_URL - This is set to the URL of the Jenkins master  that is responsible for running the build.  </ending> | </end> |
| <hitle> | OkHttp | <chare> | 1 | <pext> | <starting>OkHttp</heading> OkHttp is an efficient HTTP & HTTP/2 client for Android and Java applications. It comes with advanced features,  such as connection pooling (if HTTP/2 isn’t available),  transparent GZIP compression, and response caching,  to avoid the network completely for repeated requests. It's also able to recover from common connection problems;  on a connection failure, if a service has multiple IP addresses,  it can retry the request to alternate addresses. </ending> | </end> |
| <hitle> | Mockito | <chare> | 1 | <pext> | <starting>Mockito</heading> Mockito is an open source testing framework for Java  released under the MIT License. The framework allows the creation of test double objects  in automated unit tests for the purpose of test-driven development  or behavior-driven development. Mockito allows developers to verify  the behavior of the system under test (SUT)  without establishing expectations beforehand. One of the criticisms of mock objects is that  there is a tight coupling of the test code to the system under test,  mockito attempts to eliminate the expect-run-verify pattern  by removing the specification of expectations. Mockito also provides some annotations for reducing boilerplate code. </ending> | </end> |
| <hitle> | Mockito pros/cons | <chare> | 1 | <pext> | <starting>Mockito pros/cons</heading> pros: 1. It has support for return values. 2. It supports exceptions. 3. It has support for the creation of mock using annotation. 4. There is no need to write mock objects on your own with Mockito. 5. The test code is not broken when renaming interface method names  or reordering parameters. cons: 1. It cannot mock constructors or static methods. 2. It requires Java version 6 plus to run. 3. It also cannot mock equals(), hashCode() methods. 4. VM mocking is only possible on VMs that are supported by Objenesis. </ending> | </end> |
| <hitle> | Hamcrest | <chare> | 1 | <pext> | <starting>Hamcrest</heading> Hamcrest is a framework used for writing customized assertion matchers  in the Java programming language. It allows the match rules to be defined declaratively. This makes the hamcrest valuable in UI validation, data filtering,  writing flexible tests, etc. It can also be used with mock objects by using adaptors. Hamcrest can also e used with JUnit and TestNG. </ending> | </end> |
| <hitle> | Mockito Assert vs Verify | <chare> | 1 | <pext> | <starting>Mockito Assert vs Verify</heading> The Assert command is used to validate critical functionality. If this validation fails, then the execution of that test method is stopped  and marked as failed. In the case of Verify command, the test method continues the execution  even after the failure of an assertion statement. The test method will be marked as failed  but the execution of remaining statements of the test method  is executed normally. </ending> | </end> |
| <hitle> | ArgumentCaptor | <chare> | 1 | <pext> | <starting>ArgumentCaptor</heading> ArgumentCaptor is a class that is used to capture the argument values  for future assertions. This class is defined in the org.mockito package and can be imported from it. Some of the methods present in this class are capture(), getValue(), getAllValues(), and ArgumentCaptor <U> forClass. </ending> | </end> |
| <hitle> | Mockito annotations | <chare> | 1 | <pext> | <starting>Mockito annotations</heading> @Mock - It is used to create and inject mocked instances. @Spy - It is used to create a real object and spy on the real object. @Captor - It is used to create an ArgumentCaptor. @InjectMocks - It is used to create an object of a class  and insert its dependencies. </ending> | </end> |
| <hitle> | PowerMock | <chare> | 1 | <pext> | <starting>PowerMock</heading> PowerMock is a Java framework for unit testing purposes. This framework extends from other mock libraries  with more powerful capabilities. It uses custom classloader and bytecode manipulation  for mocking the static methods, constructors, final classes,  private methods, and more. It normally lets you test the code that is regarded as untestable. </ending> | </end> |
| <hitle> | EasyMock | <chare> | 1 | <pext> | <starting>EasyMock</heading> EasyMock is a framework for creating mock objects as it uses Java reflection  to create it for a given interface. It relieves the user of hand-writing mock objects  as it uses a dynamic mock object generator. EsayMock pros: 1. exception support, 2. return value support, 3. refactoring scale, 4. annotation support and order check support. </ending> | </end> |
| <hitle> | Junit | <chare> | 1 | <pext> | <starting>JUnit</heading> JUnit is a unit testing framework for Java that helps developers  test individual pieces of code to ensure that they are functioning as intended. In Android development, JUnit is used to test the functionality  of Android applications before they are released to the public. By using JUnit, developers can catch bugs and errors early on,  which makes for a better overall user experience. </ending> | </end> |
| <hitle> | Unit testing | <chare> | 1 | <pext> | <starting>Unit testing</heading> Unit testing is a software testing strategy that tests single entities  like methods or classes at a time. This helps to ensure that the product quality is met  as per the business requirements. It also helps in reducing the technical debt of the application  by helping developers discover issues in the code logic due to any changes. It also gives insights into  how the code logic implemented could impact future changes.  </ending> | </end> |
| <hitle> | Junit features | <chare> | 1 | <pext> | <starting>Junit features</heading> 1. JUnit is an open-source framework. 2. Supports automated testing of test suites. 3. Provides annotations for identifying the test methods. 4. Provides assertions to test expected results or exceptions of the methods  under test. 5. Provides a platform for running test cases automatically  and checking their results and giving feedback. </ending> | </end> |
| <hitle> | JUnit annotations | <chare> | 1 | <pext> | <starting>JUnit annotations</heading> @Test: This annotation over a public method of void return type  can be run as a test case. This is a replacement of the org.junit.TestCase annotation. @Before: This is used when we want to execute the preconditions  or any initialisation based statements before running every test case. @BeforeClass: This is used when we want to execute statements  before all test cases. The statements may include test connections, common setup initialisation etc. @After: This is used when we want to execute statements after each test case.  The statements can be resetting the variables, deleting extra memory used etc. @AfterClass: This is used when certain statements are required to be executed  after all the test cases of the class are run. Releasing resource connections post-execution of test cases is one such example. @Ignores: This is used when some statements are required to be ignored  during the execution of test cases. @Test(timeout=x): This is used when some timeout  during the execution of test cases is to be set. The value of x is an integer that represents the time  within which the tests have to be completed. @Test(expected=NullPointerException.class): This is used  when some exception thrown by the target method needs to be asserted. </ending> | </end> |
| <hitle> | JUnit fixture | <chare> | 1 | <pext> | <starting>JUnit fixture</heading> Fixture represents a fixed state of object sets  used as a baseline for running test methods. This is to ensure there is a fixed and well-known environment  where the results of the test methods are repeatable when run multiple times.  The fixture has the following 2 methods: 1. setUp()This runs before every test case is run. 2. tearDown() This method is run after every test is run. </ending> | </end> |
| <hitle> | mocking vs stubbing | <chare> | 1 | <pext> | <starting>mocking vs stubbing</heading> Mocking is a phenomenon where an object mimics a real object.  Whereas, stubbing are the codes responsible for taking place  of another component.  Mockito, EasyMock are some of the mocking frameworks in Java. </ending> | </end> |
| <hitle> | stubbing vs mocking | <chare> | 1 | <pext> | <starting>stubbing vs mocking</heading> Mocking is a phenomenon where an object mimics a real object.  Whereas, stubbing are the codes responsible  for taking place of another component. Mockito, EasyMock are some of the mocking frameworks in Java. </ending> | </end> |
| <hitle> | JUnit Assert | <chare> | 1 | <pext> | <starting>JUnit Assert</heading> Assert methods are utility methods that support assert conditions in test cases.  They belong to the Assert class in JUnit 4 and the Assertions class in JUnit 5. It is recommended to import the assert methods statically to the test class  for avoiding using the class as a prefix to the method. </ending> | </end> |
| <hitle> | Espresso | <chare> | 1 | <pext> | <starting>Espresso</heading> Espresso is a testing tool for Android that allows you  to test the functionality of your app from within the Android Studio IDE. With Espresso, you can write tests that simulate user interactions  and verify that the results are as expected. </ending> | </end> |
| <hitle> | Volley | <chare> | 1 | <pext> | Volley is an HTTP library that’s used  for caching and making a network request in Android applications.  --------Volley cons-------------  Volley is not suitable for large download or streaming operations  since Volley holds all responses in memory during parsing.  -------Features of Volley-----------  1. Request queuing and prioritization  2. Effective request cache and memory management  3. Extensibility and customization of the library to our needs  4. Cancelling the requests  -----------Volley thread--------------  Request is processed in Main Thread.  Cache is processed in Cache Thread.  Actual network request is processed in Network Thread or IO Thread.  -----------Classes in Volley------------  Request Queue, Request  ----------Volley Request Queue--------------  It is the interest one used for dispatching requests to the network.  We can make a request queue on demand if required,  but typically it is created early on, at startup time,  and keep it around and use it as a Singleton.  ------------Volley Request-----------------  All the necessary information for making web API call is stored in it.  It is the base for creating network requests(GET, POST).  StringRequest, JSONObjectRequest, ImageRequest, etc.  ------------Volley Request an API-----------  val requestQueue: RequestQueue = Volley.newRequestQueue(context)  val request = StringRequest(  Request.Method.GET,  Constants.BASE\_URL,  { apiResponse: String ->  //success result  }, { error ->  // failed result  }  )  requestQueue.add(request)  -------------Volley add header---------  Val stringRequest = object: StringRequest(Request.Method.GET, url,  Response.Listener { //success },{ //failed }) {  Override fun getHeaders(): MutableMap<String, String> {  val header = HashMap<String, String>()  header[“Authorization”] = “API\_KEY”  return header  }  }  }  -----------Volley Request Priority------------  private fun showPriorityDemo() {  val requestQueue = Volley.newRequestQueue(context)  val request1 = object : StringRequest(  Method.GET, "URL", {//success}, {//failed} ) {  {  override fun getPriority(): Priority {  return Priority.LOW  }  }  val request2 = object : StringRequest(  Method.GET, "URL", {//success}, {//failed} ) {  {  override fun getPriority(): Priority {  return Priority.HIGH  }  }  requestQueue.add(request1)  requestQueue.add(request2)  }  ------------Volley cancel request----------------  val request1 = object : StringRequest(  Method.GET,  ) {  override fun cancel() {  super.cancel()  //after Api Got Cancel, this block got execute  }  }  And then we call requestQueue.cancelAll(TAG)  ------------ Caching using Volley------------------  class CustomVolleyRequestQueue(context: Context) {  val imageLoader: ImageLoader  private var requestQueue: RequestQueue  init {  val cache1: Cache = DiskBasedCache(  context.cacheDir,  10 \* 1024 \* 1024  )  val network: Network = BasicNetwork(HurlStack())  requestQueue = RequestQueue(cache1, network).also {  it.start()  }  imageLoader = ImageLoader(requestQueue,  object : ImageLoader.ImageCache {  private val cache = LruCache<String, Bitmap>(20)  override fun getBitmap(url: String?): Bitmap? {  return cache[url]  }  override fun putBitmap(url: String?, bitmap: Bitmap?) {  cache.put(url, bitmap)  }  })  }  } | </end> |
| <hitle> | REST API Authentication | <chare> | 1 | <pext> | ------------HTTP basic authentication--------------  1. HTTP basic authentication.  An HTTP Authorization header containing  a base64-encoded username:password string is passed  in the request header.  ----------API keys-----------------------  2. API keys.  A third-party application is granted permission to use an API  by issuing a key which may have specific rights or be restricted  to a particular domain.  The key is passed in every request in the HTTP header  or on the querystring.  -----------OAuth--------------  3. OAuth.  A token is obtained before any request can be made  by sending a client ID and possibly a client secret to an OAuth server.  The OAuth token is then sent with each API request until it expires.  ------------JSON Web Tokens (JWT)------------  4. JSON Web Tokens (JWT).  Digitally signed authentication tokens are securely transmitted  in both the request and response header.  JWTs allow the server to encode access rights  so calls to a database or other authorization system is not necessary. | </end> |
| <hitle> | REST API Security | <chare> | 1 | <pext> | 1. use HTTPS  2. use a robust authentication method  3. use CORS to limit client-side calls to specific domains  4. provide minimum functionality—  that is, don’t create DELETE options which are not required.  5. validate all endpoint URLs and body data  6. avoid exposing API tokens in client-side JavaScript  7. block access from unknown domains or IP addresses  8. block unexpectedly large payloads  9. consider rate limiting—  That is, requests using the same API token  or IP address are limited to N per minute  10. respond with an appropriate HTTP status code and caching header  11. log requests and investigate failures | </end> |
| <hitle> | editpoint | <chare> | 1 | <pext> |  | </end> |