Android save instance state object

\$\begingroup\$ I'm making one of my first android apps. Every time I change the screen orientation, I met this problem that destroys the Android app and installs it again (I know this is more complicated, but this is not the subject of this post.) I wonder if it's a good idea that a Game game variable in my MainActivity.java will only implement Serializable. This way I can do this: public class MainActivity AppCompatActivity { private MagicSquareGame game; @Override protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.activity_main); game = MagicSquareGame.getEngine(); inititTextReferences(); initButtonReferences(); updateSumResultsOnScreen(); } (all other things ...) @Override protected void onSaveInstanceState (Bundle outState) { outState.putSerializable(game, game); Log.i(MSG, Saving games); super.onSaveInstanceState(outState); } @Override protected void onRestoreInstanceState (Bundle savedInstanceState) { super.onRestoreInstanceState(savedInstanceState); game = (MagicSquareGame) savedInstanceState.getSerializable(game); updateSumResultsOnScreen(); Log.i(MSG, installation Game); Is that a good approach? The smell of code? What problems can I face? \$\endgroup\$ Maintaining and retating an activity's UI state in a timely way between system-initiated activity or application destruction is an important part of the user experience. In such cases, the user waits for the User UI state to remain the same, but destroys system activity and any stored state. To bridge the gap between user expectation and system behavior, use a combination of ViewModel objects, the onSaveInstanceState() method, and/or local storage to maintain UI state between such application and activity instance transitions. How we combine these options depends on the complexity of your user interface data, the use of cases for your application, and the speed at which it is received against memory usage. No matter what approach you take, you should make sure that your application meets user expectations for user interface status and provides a smooth and fast user interface (especially after frequent configuration changes, such as rotation, which prevents latency in loading data into the UI). In most cases, you must use both ViewModel and onSaveInstanceState(). This page explaines user expectations about UI status, options available to maintain the state, balances and limitations of each. User expectations and system behavior Depending on a user's action, they expect the activity status to be cleared or the state to be protected. In some cases, the system as expected by the user. In other cases, the system does the opposite of what the user expects. User-initiated User Status denial When the User initiates an activity, the temporary User UI status of that activity remains the same until the user completely rejects the activity. The user can turn off an activity alumn alms: Scrolling up from the activity that killed the app by completing some kind of recent activity by pressing the back button of the Overview (Recently) Screen (supported by Activity.finish()) In these full dismissal situations, the user's assumption is that they are permanently away from the activity. and reopen the activity, they expect the activity to start from a clean situation. The underlying system behavior for these dismissal scenarios matches user expectation - the activity instance is destroyed and removed from memory, along with any state stored in it and any saved instance status records associated with the activity. There are some exceptions to the full dismissal of this rule - for example, a user can expect a browser to use the back button to take them to the webpage where they look exactly before exiting the browser before exiting the browser. System-initiated User Side state denial A user waits for an activity's User Agent status to remain the same throughout a configuration change, such as rotating or switching to multi-window mode. By default, however, the system destroys activity when such a configuration change occurs and deletes any UI state stored in the activity instance. For more information about device configurations, see the Configuration reference page. Remember, it is possible to override the default behavior (even if it is not recommended) for configuration changes. For more details, see Use Configuration Change Yourself. The user also waits for your activity's User-I Check-in status to remain the same if they temporarily go to a different application and then return to your application. For example, the user makes a call in your search activity and then presses the home button or responds to a phone call - they expect to find the search keyword and its results exactly there when they return to the call activity. In this scenario, your application is placed in the background and the system does its best to keep your application process in memory. However, the system can destroy the application process while the user interacts with other applications away. In such a case, the activity instance is destroyed along with any state stored in it. When the user restarts the application, the activity is unexpectedly clean. To learn more about process death, see Processes and Application Lifecycle. When user interface state protection options do not match the default system behavior, the user's expectations for the User's UI state are you need to save and restore the status. Each user interface state protection options varies according to the following dimensions that affect the user experience: View Model Registered instance state Permanent Storage Storage location location Survives configuration change of disk or disk serialized memory on the network Yes Yes Yes The process initiated by the system survives death No Yes Survives user full activity dismissal/onFinish() No Yes Data limitations complex objects fine, however, space is limited only to primitive types and small objects such as simple, string-only disk space or cost/write time from the network resource fast read/write time (requires memory access only) slow (requires serialization/deserialization and disk access) slow (requires disk access or network movement) Use ViewModel to handle configuration changes ViewModel is ideal for storing and managing user-related data when the user is actively using the application while the user is actively using the application. Provides quick access to UI data and helps prevent you from re-importing data from the network or disk, window resizing, and other frequent configuration changes. To learn how to implement ViewModel, see the ViewModel guide. ViewModel protects data in memory, which means it's cheaper than getting data from a disk or network. ViewModel is associated with one activity (or another lifecycle owner) - it remains in memory during the configuration change, and the system automatically associates the ViewModel with the new activity instance that originated from the configuration change. ViewModels are automatically destroyed by the system when your user withdraws from your activity or part, or if you call an end(), which means that the status will be cleared as the user expects in these scenarios. In addition to the saved instance state, ViewModels are destroyed during a system-initiated process death. Therefore, you must use ViewModel objects in the same way as onSaveInstanceState() (or any other disk pereration) and store identifiers in savedInstanceState to help view data after system death. If you already have an in-memory solution to store your UI state among configuration changes, you may not need to use ViewModel. Use onSaveInstanceState() as a backup to handle process death from the process initiated on the system OnSaveInstanceState() search stores the data needed to reinstall the state of a UI controller, such as an activity or fragment, if the system destroys this controller and then rebuilds it. To learn how to apply the saved instance state, see Save and restore activity status in the Activity Lifecycle guide. Saved instance state buckets continue both configuration changes and processing death, but are limited by the amount of storage and speed because onSavedInstanceState() serializes data to disk. Serialized if complex, serialization can consume a lot of memory. Because this process occurs on the main thread during the configuration change, it can cause dropped frames and visual stuttering if serialization takes too long. Do not use save the Store on the EdInstanceState() amounts of data, such as bitmaps or complex data structures that require long serialization or deserialization. Instead, store only primitive types and simple, small objects such as String. Therefore, if other perdurance mechanisms fail, use SaveInstanceState() to store the least amount of data, such as identity, to recreate the data needed to restore the UI to its previous state. Most applications must implement SaveInstanceState() to handle system-initiated process death. Depending on the usage status of your application, you may not need to use it at all on SaveInstanceState(). For example, a browser can take a user back to the full webpage they looked at before exiting the browser. If your activity is acting this way, you can give up using onSaveInstanceState() and resume everything locally instead. Also, when you open an activity for a purpose, the extras package is delivered to the activity both when the configuration changes and when the system restores the activity. If a search status data part, such as a search query, is transferred as a intentional attachment when the activity starts, you can use the extras package instead of the onSaveInstanceState() package. See Intent and Intent Filters to learn more about intent extras. In any of these scenarios, you must use a ViewModel to avoid wasting re-installation cycles from the database during the configuration change. Where storeable UI data is simple and lightweight, you can only use the Save What's recorded() to protect your status data. Note: You can now access a saved state on ViewModel objects with the Saved Status module for ViewModel (currently in Alpha). This saved state can be accessed through an object named SavedStateHandle. You can see how Android lifecycle-sensitive components are used in codelab. Use local perduration to handle transaction death for complex or large data, such as databases or shared preferences, as long as your application is installed on the user's device (unless the user cleans up your app's data). While such local storage survives system-initiated activity and application process death, it can be expensive to get because local storage in memory will need to be read. This persistent local storage can be part of your application architecture to store all the data you don't want to lose if you turn activity on or off. ViewModel or saved instance state are not long-term storage solutions, and therefore are not backups for local storage, such as databases. Instead, you should use these mechanisms to temporarily store temporary UI state and use persistent storage for other application data You can use your application model data in the long run (e.g. (e.g. device). Managing UI state: You can efficiently record and restore UI state by dividing the job between various perdurity mechanisms. In most cases, each of these mechanisms should store different types of data used in the activity, depending on data complexity, access speed, and lifetime balances: Local perdurability: If you turn activity on or off, it stores all the data you don't want to lose. Example: A collection of song objects that can contain audio files and metadata. ViewModel: Store all the data needed to view the associated Web Information Controller in memory. Example: Song objects of the most recent search and most recent search query. onSaveInstanceState(): Stores the small amount of data needed to easily restore activity status if the system stops, and then recreates the UI Controller. Instead of storing complex objects here, permanently store complex objects in local storage and save a unique ID for themInstanceState(). Example: Store the most recent search query. For example, consider an activity that lets you search your song library. Here's how to handle different events: When a user adds a song, ViewModel permanently re-handles this data locally. If this newly added song is something that should be shown in the User UI, you must also update the data in the ViewModel object to reflect the addition of the song. Be remember to make all database attachments for the parent thread. When a user calls a song, the complex song data that you download from the database for the UI Controller must be stored immediately in the ViewModel object. You must also save the search query itself in the ViewModel object. When the activity enters the background, the system calls it SaveInstanceState(). You must save the search query to the onSaveInstanceState() package. It is easy to save this small amount of data. It's also all the information you need to restore activity to its current state. Restore complex states: recompragmenting parts When the time comes for the user to return to the activity, there are two possible scenarios for re-creating the activity: The activity is recreated after it is stopped by the system. The activity has a query saved in an onSaveInstanceState() package and must pass the query to viewmodel. ViewModel sees that there are no cached search results, and delegates who load search results using the given search query. The activity is created after the configuration change. The activity has a query saved in an onSaveInstanceState() package, and search results are already cached in viewmodel. From the query onSaveInstanceState() package, you have already loaded the required data and do not need to query the database again You pass it to ViewModel. Note: An event activity The onSaveInstanceState() package originally created does not contain data, and the ViewModel object is empty. When you create a ViewModel object, it will swallow an empty query and tell the ViewModel object that there is no data that needs to be loaded yet. Therefore, the activity starts in an empty state. Additional resources For more information about saving UI statuses, contact the following resources. Blogs Blogs