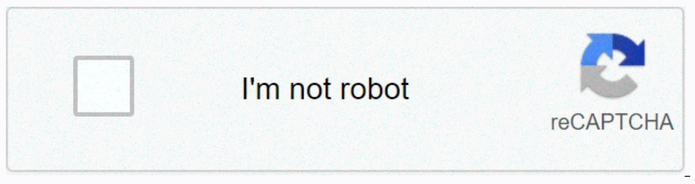

Continue

Autofill hints in android

Programs that use standard views work with an AutoComplete framework without requiring a custom configuration. However, you can optimize the operation of the application from the framework. For a guide to using interactive guides, see Optimize your autocomplete app. AutoComplete settings This section describes how to configure basic AutoComplete features for an app. The AutoComplete AutoComplete service setting must be configured on your device to use the AutoComplete framework. While most phones and tablets running Android 8.0 (API level 26) and above fly with autocomplete service, we recommend using a test service when testing your app, such as an AutoComplete service as part of an Android Java autocomplete sample | Cotlin. When you use the emulator, you must explicitly set the AutoComplete service because the emulator may not come with the default service. After you install the AutoComplete test service from the sample application, turn on AutoComplete by moving settings > System > Languages & input > Advanced > Input assistance > AutoComplete service. For more information about setting up an emulator to test autocomplete, see <a0><a1> Set up an emulator to test autocomplete </a1><a2></a2> </a0> . Provide autocomplete tips The AutoComplete service tries to determine the type of each view by using heuristics. However, if your app uses these heuristics, the autocomplete behavior may change unexpectedly when you update the app. For AutoComplete to correctly identify app form factors, provide autocomplete tips. You can set these hints using the android:autofillHints attribute. The following example sets the password hint to EditText: You can also set hints programmatically using <EditText android:layout_width=match_parent android:layout_height=wrap_content android:autofillhints=password></EditText> setAutofillHints() as shown in the following example: val password = findViewById<EditText>(R.id.password) password.setAutofillHints(View.AUTOFILL_HINT_PASSWORD) EditText password = findViewById(R.id.password); password.setAutofillHints(View.AUTOFILL_HINT_PASSWORD); Predefined tooltip constants AutoComplete frame does not check hints; they are simply transferred together without changes or checks for autocomplete service. Although you can use any value, the View class and the AndroidX HintConstants class contain lists of officially supported tooltips. Using a combination of these constants, you can create layouts for common AutoComplete scenarios: Account credentials When you autocomplete credentials, the sign-in form may contain hints, such as AUTOFILL_HINT_USERNAME and AUTOFILL_HINT_PASSWORD. When you create a new account or change your user name and password, you can use and AUTOFILL_HINT_NEW_PASSWORD. When asking for credit card information, you can use tips such as AUTOFILL_HINT_CREDIT_CARD_NUMBER and AUTOFILL_HINT_CREDIT_CARD_SECURITY_CODE. For credit card expiration, do one of the following: Physical Address During </EditText > </EditText> at a physical address, you can use the following tips: People names When autofilling people's names, you can use the following tips: Note: AUTOFILL_HINT_NAME removed and replaced with AUTOFILL_HINT_PERSON_NAME. Phone numbers For phone numbers, you can use the following: One-time password (OTP) For a one-time password in one view, you can use AUTOFILL_HINT_SMS_OTP. When you use multiple views where each map view is per OTP, you can use the generateSmsOptHintForCharacterPosition() method to create hint characters. Mark fields as important for AutoComplete You can tell the system whether to include individual fields in your application in the view structure for autocomplete purposes. By default, browsing uses IMPORTANT_FOR_AUTOFILL_AUTO mode, which allows Android to use its heristics to determine whether viewing is important for autocomplete. You can set importance with android:importantForAutofill attribute: <TextView android:layout_width=match_parent android:layout_height=wrap_content android:importantforautofill=no></TextView> The value of importantForAutofill can be any of the values defined in android:importantForAutofill: auto Let android system use its heretics to determine if the species is important for autofill. no This view is not important for autocomplete. You can also use the method setImportantForAutofill() : val captcha = findViewById<TextView>(R.id.captcha) captcha.setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_NO) TextView captcha = findViewById(R.id.captcha); captcha.setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_NO); And cases where views, view structure, or the whole thing are not important for AutoComplete: The CAPTCHA field in the sign-in case is not usually important for AutoComplete. In such cases, you can mark a view as IMPORTANT_FOR_AUTOFILL_NO. In a view where a user creates content, such as a text or spreadsheet editor, the entire view structure is not usually important for AutoComplete. In such cases, you can mark the view IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS make sure that all children are also marked as not important for autocomplete. In some activities in games, such as those that reflect gameplay, none of the case views are important for autocomplete. You can mark the root view as IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS to make sure that all views in the case are marked as not important for AutoComplete. Associated websites and mobile app data AutoComplete, such as AutoComplete with Google, can share user login data between browsers and Android devices after linking the app and website. When a user selects the same AutoComplete service on both platforms, sign in to </TextView>Sign-in credentials available for autocomplete when they sign in to the appropriate Android app. To link an Android app to a website, you'll need to link to delegate_permission/common.get_login_creds digital assets on your site. Then declare an association in the AndroidManifest file .xml application. For detailed instructions on linking a website to an Android app, see Turn on automatic sign-in apps and websites. End an AutoComplete workflow This section describes some scenarios in which you can take steps to improve autocomplete functionality for app users. Determine whether AutoComplete is turned on You can implement additional AutoComplete features in your app or even in certain app views if AutoComplete is available to the user. For example, TextView shows an AutoComplete entry in the overflow menu if AutoComplete is enabled for the user. To check whether AutoComplete is enabled for the user, call the IsEnabled() method of the AutofillManager object. Users can turn AutoComplete on or off, and change the AutoComplete service, and move in settings > System > Languages & input > Advanced > Input assistance > AutoComplete service. The application cannot override the user's AutoComplete settings. To help optimize sign-in and sign-in for users without autocomplete, try implementing Smart Lock for passwords. Force AutoComplete prompt Sometimes, you may need to force autocomplete requests to occur in response to user actions. For example, TextView offers an AutoComplete menu item when a user clicks a view for a long time. The following sample code shows how to force an AutoComplete request: fun eventHandler(view: View) { val afm = requireContext().getSystemService(AutoCompleteManager::class.java) afm?. requestAutofill(view) } public invalid eventHandler(View view) { AutoCompleteManager afm = context.getSystemService(AutoCompleteManager.class); if (afm != null) { afm.requestAutofill(view); } } You can also use the cancel() method to cancel the current autocomplete context. This can be useful, for example, if you have a button that clears the fields on the sign-in page. Use the correct AutoComplete type for data in selecting Pickers controls useful in some AutoComplete scenarios, providing a user interface that allows users to change the value of a field that stores date or time data. For example, in the credit card form, selecting a date allows users to enter or change their credit card expiration date. However, you must use a different view, such as EditText, to display data when the selection is not displayed. The EditText object first expects autocomplete data type AUTOFILL_TYPE_TEXT. If you are using a different data type, you must create a a view that is inherited from EditText and implements the methods required to process the appropriate data type. For example, if you have a date field, implement the methods with the logic that correctly handles the value of the AUTOFILL_TYPE_DATE. If you specify an AutoComplete data type, autocomplete AutoComplete to create an appropriate view of the data displayed in the view. For more information, see Use autocomplete picketers. Completion of autocomplete context AutoComplete frame stores user input for later use, showing the Save for AutoComplete dialog box when the AutoComplete context is complete. Typically, the AutoComplete context ends when the activity is complete. However, there are some situations where you need to explicitly report the framework; for example, if you use the same activity but different snippets for both your sign-in and content screens. In these special situations, you can explicitly end the context by calling AutofillManager.commit(). Note: It is important that you do not call commit() when the user has not completed the multi-sector form. For example, if you have a snippet of your username followed by a password snippet, wait for the password to be entered for the commit() call. Support for custom views Custom views can specify metadata that is exposed to autocomplete frames by using the AutoComplete API. Some views act as a container of virtual children, such as views that contain a user interface provided by OpenGL. These views must use apis to specify the structure of the information used in the application before they can work with the AutoComplete framework. If your program uses custom views, consider the following scenarios: Custom view contains a standard view structure or default view structure. The custom view has a virtual structure or view structure that is not available for the AutoComplete framework. Custom views with a standard custom view structure can determine the metadata for which you want to automatically fill in. Make sure that the custom view manages the metadata correctly to work with the AutoComplete frame. The custom view should take the following: Handle the AutoComplete value that the frame sends to the program. Specify the type and value of the AutoComplete frame. When AutoComplete is triggered, the AutoComplete frame calls autofill() in your view and sends the value that your view should use. You need to implement autofill() to specify how your custom view handles autofill values. Your view must specify the type and value of autocomplete by overracting the getAutofillType() and getAutofillValue() methods respectively. By adding this code, you will ensure that your view can provide appropriate autocomplete types and values within. Finally, AutoComplete should not fill out the view if the user cannot provide a value for the view in its current state (for example, if the view is disabled). In such cases, getAutofillType() should return AUTOFILL_TYPE_NONE, getAutofillValue() should return to zero, and AutoComplete() should do nothing. In the following cases, additional steps are required to Work within: You can edit a custom view. The custom view contains sensitive data. Custom view can be edited If the view can be edited, you should notify the AutoComplete frame of the changes by calling notifyValueChanged() on autocompleteManager AutoCompleteManager Custom view contains sensitive data If the view contains personal information (PII), such as email addresses, credit card numbers, and passwords, it must be marked as such. In general, views whose content comes from static resources do not contain sensitive data, but views whose content is dynamically configured may contain sensitive data. For example, a label that contains your user name type does not contain sensitive data, while the label that contains Hello John does. To mark whether a view contains sensitive data or not, make aProvideAutofillStructure() and call setDataIsSensitive() on the ViewStructure object. The following sample code shows you how to mark data in a view structure as sensitive or not: override fun onProvideAutofillStructure(outline: ViewStructure, tags: Int) { super.onProvideAutofillStructure(structure, flags) // Content that comes from static resources @Override usually not sensitive. int flags) { super.onProvideAutofillStructure (structure, flags); // Content that comes from static resources is generally not sensitive. Note: Frame: AutoComplete provides that all data is sensitive by default. If the view accepts only predefined values, you can use the setAutofillOptions() method to set options that AUTOFILL_TYPE_LIST you can use to autocomplete this view. , such as Counter, is a similar case. For example, a counter that provides dynamically created years (based on the current year) to use in credit card expiration fields can implement the getAutofillOptions() method of the adapter interface to provide a list of years. Views that use ArrayAdapter can also provide lists of values. ArrayAdapter automatically sets AutoComplete settings for static resources. If you provide a value dynamically, however, you must override getAutofillOptions(). Custom views with a virtual AutoComplete Frame structure require a view structure before it can edit and store information in the app user interface. There are some situations where the view structure is not available for the framework: the application uses a low-level rendering engine, such as OpenGL, to display the user interface. The application uses an instance of Canvas to draw a user interface. In these cases, you can specify a view structure by implementing theProvideAutofillVirtualStructure() and follow these steps: Increase the number of child view structures by calling addChildCount(). Add your child by calling newChild(). Set id for the child using the setAutofillId() call. Install Install properties, such as values and autocomplete type. If the data in the virtual child is sensitive, you should pass true to installDataIsSensitive() or false otherwise. The following code snippet shows you how to create a new child in a virtual structure: override fun onProvideAutofillVirtualStructure(outline: ViewStructure, tags: Int) { super.onProvideAutofillVirtualStructure(structure, check boxes) // Create a new child in a virtual structure. child.setAutofillId(structure.autofillId!!, childVirtualId) // Populate the child by providing properties such as value and type. child.setAutofillValue(childAutofillValue) child.setAutofillType(childAutofillType) // Some children may provide a list of values. like other types of views, mark the data as sensitive if // @Override is appropriate. View Child Structure = structure.newChild(childIndex); Set autofill ID for child. child.setAutofillId(structure.getAutofillId(), childVirtualId); Place your child by providing properties such as value and type. child.setAutofillValue(childAutofillValue); child.setAutofillType(childAutofillType); Some children may provide a list of valuables. CharSequence childAutofillOptions[] = { option1, option2 }; child.setAutofillOptions(childAutofillOptions); Just like other types of views, mark the data as sensitive if // is appropriate. boolean sensitive = !contentIsSetFromResources(); child.setDataIsSensitive(sensitive); } When virtual structure elements change, you should notify the frame by doing the following tasks: If the focus inside children changes, call NotifyViewEntered() and report it toViewExited() at the AutofillManager facility. If the child's value changes, call notifyValueChanged() on the AutofillManager object. If the view hierarchy is no longer available because the user performed a step in the workflow (for example, a user who is logged on by using the login form) to call commit() on the AutofillManager object. If the view hierarchy is no longer valid because the user canceled the workflow step (for example, if the user clicks the button that clears the sign-in form), call cancel() on the AutofillManager object. Use callbacks to AutoComplete events If your app provides its own AutoComplete views, you need a mechanism that tells the app to enable or disable the view in response to changes in the availability of the autocomplete user interface. AutoComplete </CharSequence>provides this mechanism as autocompleteCallback. This class provides the onAutofillEvent(View, int) method that the application calls after changing the autocomplete state associated with the view. There is also an overloaded version of this method that includes the childId parameter that your app can use with virtual views. Available states are defined as constants in the callback. You can register a callback using the AutofillManager class registerCallback() method. The following sample code shows how to declare a callback for AutoComplete events: val afm = context.getSystemService(AutoCompleteManager::class.java) afm?. registerCallback(object : AutofillManager.AutoFillCallback() { // For virtual structures, override // onAutofillEvent(View view, int childId, int event) instead. override fun onAutofillEvent(see: View, Event: Int) { super.onAutofillEvent (view, event) when (event) { EVENT_INPUT_HIDDEN -> { // AutoComplete associated with the view was hidden. } EVENT_INPUT_SHOWN -> { // View-related autofill availability is shown.} EVENT_INPUT_UNAVAILABLE -> - // AutoComplete is not available. } } } }) AutoCompleteManager afm = getContext().getSystemService(AutoCompleteManager.class); afm.registerCallback(new AutoCompleteManager.AutofillCallback() { // For virtual override structures // onAutofillEvent(View view, int childId, int event) instead. @Override public void onAutofillEvent(@NonNull View view, int childId, int event) int event) {

super.onAutofillEvent(view, event); switch (event) { case EVENT_INPUT_HIDDEN: // AutoComplete availability associated with viewing was hidden EVENT_INPUT_UNAVAILABLE EVENT_INPUT_SHOWN. When it is time to remove the callback, use the unregisterCallback() method. To indicate that the contents of the view have been automatically filled. reissing the android:autofiledHighlight element of the theme used by the application or activity, as shown in this <resources> <style name=MyAutofilledHighlight parent=... > <item name=android:autofilledHighlight>@drawable/my_drawable</item> </style> </resources > examples: res/values/styles.xml res/drawable/my_drawable.xml <shape xmlns:android= amp;gt; <solid android:color=#4DFF0000></solid> </shape> AndroidManifest.xml Note: When configuring this drawer, set the first two color bytes so that <application ...=android:theme=@style/MyAutofilledHighlight> <!-- or --> ...= android:theme=@style/MyAutofilledHighlight>колір був напівпрозорим. В іншому випадку текст автозаповнення не відображається. Автентифікація для</activity> </application> </application> The AutoComplete service may require the user to authenticate before the service can populate the fields in your app, in which case Android runs the service authentication as part of a stack of your activity. You do not have to update the application to support authentication because authentication occurs on the service. However, you should make sure that the activity view structure is preserved when you restart activities (for example, by creating a view structure in onCreate() instead of in onStart() or onResume()). You can check how your app behaves when the AutoComplete service requires heuristicsService authentication from the AutoComplete sampleframework and configure it to require authentication to populate the response. You can also use the BadViewStructureCreationSignInActivity role model for this issue. Assigning AutoComplete IDs to a redesigned view Containers that redescore views, such as the RecyclerView class, are very useful for applications to display scroll lists based on large data sets. When you scroll through a container, the system re-views the layout, but the views then contain new content. If the original content of the view is full, AutoComplete stores the logical value of the views by using autocomplete. The issue occurs when the system re-views the layout, the logical view IDs remain the same, causing the user's autocomplete data to be incorrect associated with the AutoComplete ID. To work around this issue on Devices running Android 9 (API level 28) and above, you can explicitly manage the AutoComplete view ID used by RecyclerView using these new methods: the getNextAutofillId() method gets a new AutoComplete ID unique to the activity. The setAutofillId() method sets the unique, logical identifier of the autocomplete of this view in the case. Solve known issues This section provides solutions to known issues within AutoComplete. AutoComplete crashes apps on Android 8.0, 8.1 In Android 8.0 Os (API level 26) and 8.1 (API level 27), AutoComplete may cause the app to crash in certain scenarios. To resolve any potential issues, you must mark any views that are not autocomplete importantforAutofill = no. You can also mark all activities using importantForAutofill =noExcludeDescendants. Note that these practices are generally recommended for any views that do not require autocomplete. Modified resolution dialog boxes are not considered for AutoComplete in Android 8.1 (API level 27), and below if the view in the dialog box is changed after it is already displayed, the view is not considered for AutoComplete. These views are not included in the object which Android system sends to the AutoComplete service. As a result, the service cannot fill out the submission. To work around this issue, replace the parameter property of the Activity Property Token dialog box that creates the dialog box. After you verify that AutoComplete is enabled, save the window settings in the class method that inherits the dialog box. Then, replace the Stored Settings token property with the Parent Activity Property property by using the onAttachedToWindow() method. The following code snippet shows the class that implements the workaround: MyDialog class(context: context) : Dialog(context) { // Used to store dialog Build.VERSION_CODES Build.VERSION.SDK_INT settings. O || Build.VERSION.SDK_INT != Build.VERSION_CODES. O_MR1) { return false } val autofillManager = if (Build.VERSION.SDK_INT > = Build.VERSION_CODES. M) { context.getSystemService(AutoCompleteManager::class.java) } yet { null } return AutoCompleteManager?. isEnabled ?: false } override fun atWindowAttributesChanged(params: WindowManager.LayoutParams) { if (params.token == null & token != null) { params.token = marker } super.onWindowAttributesChanged(params) } override the fun onAttachedToWindow() { if (isDialogResizedWorkaroundRequired) { Token = OwnerActivity!!. window.attributes.token } super.onAttachedToWindow() } } public class MyDialog extends dialog { public MyDialog(Context context) { super(context); } // Used to store dialog settings. private IBinder token; @Override public void onWindowAttributesChanged (WindowManager.LayoutParams params) { if (params.token == null & token != null) { params.token = marker; } super.onWindowAttributesChanged(params); } @Override public void onAttachedToWindow() { if (isDialogResizedWorkaroundRequired()) { marker = getOwnerActivity().).) getowWind().getAttributes().token; } super.onAttachedToWindow().) } private boolean is DialogueRequired() - if (Build.VERSION.SDK_INT != Build.VERSION_CODES. O || Build.VERSION.SDK_INT != Build.VERSION_CODES. O_MR1) { return false; } AutoComplete AutoCompleteManager = null; if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.M) { AutoCompleteManager = getContext().getSystemService(AutoCompleteManager.class); } AutoComplete ReturnManager != null & AutoCompleteManager.isEnabled(); } To avoid unnecessary operations, the following code snippet shows you how to check if autocomplete is supported on the device and enabled for the current user, and whether this method is required: // AutofillExtensions.kt fun Context.isDialogResizedWorkaroundRequired(): Boolean { // After as the problem is resolved on Android, you should check whether the // workaround is required for the current device Build.VERSION_CODES Build.VERSION.SDK_INT. O) { // AutoComplete frame is only available on Android 8.0 // or higher. return false } val afm = getSystemService(AutofillManager::class.java) // Return true if autocomplete is supported by the device and enabled // for the current user. afm != null & afm.isEnabled } публічний клас АвтозаповненняДодаток { публічний статичний булеан єДіалогРезифікованийЗавдячний (контекст) { { Once the problem is resolved on Android, you should check if // workaround is still required for the current device. return isAutofill Affordable (context); } public static logical isAutofillAvail (context context) { if (Build.VERSION.SDK_INT < Build.VERSION_CODES. O) { // AutoComplete frame is only available on Android 8.0 // or higher. AutoCompleteManager afm = context.getSystemService(AutofillManager.class); returns true if autofill is supported by device and enabled // for current user. return afm != null & afm.isEnabled(); } } } Check your app with AutoComplete Most apps work with AutoComplete services without any changes. However, you can optimize your app so that it's best with autocomplete services. After optimizing the program, you should check it to make sure it works as intended with autocomplete services. To test the app, you must use an emulator or physical device running Android 8.0 (Level 26 API) or higher. For more information about how to create an emulator, see Create and manage virtual devices. Install autocomplete service Before you can test an app with AutoComplete, you need to install another app that provides autocomplete services. You can use third-party apps for this purpose, but it's easier to use a sample AutoComplete service so you don't have to subscise with any third-party services. You can use Android AutoComplete Frame Sample Java | Kotlin to test your app using autocomplete services. The sample application provides autocomplete services and client activity classes that you can use to validate a workflow before you use it with an application. This page references the Android AutoCompleteFramework sample program. Once you have installed the app, you must enable the AutoComplete service in the system settings. You can enable this service, go to Settings > System > Languages & input > Advanced > input > AutoComplete services. Analyze data requirements To test an app using AutoComplete, the service must have data it can use to fill the app. The service should also understand what type of data is expected in your app's views. For example, if your app has a view that is waiting for a user name, the service must have a data set that contains a user name and some mechanism to know that the view is waiting for such data. You should tell the service what type of data is expected in your views by setting the android:autofillHints attribute. Some services use complex geristics to determine the type of data, but others, such as a sample application, rely on the developer to provide this information. Your app works better with autocomplete if you set the android:autofillHints attribute in views that are related to autocomplete. Run the test After you've analyzed the data requirements, you can run a test that includes storing test data in autocomplete and running autocomplete in your app. Save data to The following steps show you how to save data to autocomplete autocomplete Currently active: Open an application that contains a view that expects the type of data you want to use during the test. A sample Android-AutofillFramework app provides a user interface with views waiting for multiple types of data, such as credit card numbers and usernames. Tap the view that contains the data type you want. Enter a value in the view. Click a confirmation button, such as Sign in or Send. Typically, you must submit a form before the service attempts to save the data. The system opens a dialog asking you to grant your permission to save the data. The dialog shows the name of the currently active service. Make sure it's the service you want to use during the scan, and then tap Save. If Android doesn't display the permissions dialog box or the service isn't the one you want to use during the test, check to see if the service is currently active in system settings. Start AutoComplete in the app The following steps show you how to call autocomplete in the app: Open the app and go to the activity that has the views you want to test. Tap the view you want to fill. The system should display an AutoComplete interface that contains data sets that can fill the view as shown in Figure 1. Tap the data set that has the data you want. The view should display data that was previously stored in the service. In Figure 1. AutoComplete a user interface that displays available data sets. If Android doesn't display the AutoComplete interface, you can try the following troubleshooting options: Make sure the views in your app use the correct value in the android:autofillHints attribute. You can view a list of possible attribute values with the constant prefix AUTOFILL_HINT the View class. Make sure that the android:importantForAutofill attribute is set to a value other than either the view that you want to populate, or that the value other than noExcludeDescendants is set to appearance or to one of its parents. Parents.