ChatGPT

Persistent Device Pairing via QR Code and CloudKit Sharing

Overview of CloudKit Sharing for Device Pairing

Apple's CloudKit provides a built-in **sharing mechanism** (using the `CKShare` class) to allow one user's private database records to be securely shared with another user's iCloud account. In a pairing scenario (e.g. parent–child apps), you can leverage CloudKit sharing to establish a persistent link between two devices, even if they use different iCloud accounts [1] [2]. After a one-time invite/accept process, the two devices will have a shared CloudKit record or zone that both can access, enabling ongoing data sync (such as screen time alerts, settings, etc.) between them.

Same iCloud Account vs Different Accounts: If both devices are signed into the **same iCloud account**, explicit sharing isn't necessary – all devices under one Apple ID automatically sync data in that account's private CloudKit database. In other words, the private database is only accessible to that user's devices [3]. You might still implement an in-app "pairing" step to assign roles (parent vs child), but no CloudKit invitation is needed for data sync. By contrast, devices on **different iCloud accounts** *must* use CloudKit sharing to collaborate – there is no automatic family-sharing of CloudKit data without an invite [4]. This means one device (the "owner") will share specific records or an entire record zone to the other device's user (the "participant") via a CloudKit share invitation.

CloudKit Sharing Basics (Private, Shared Databases and CKShare)

Every app with CloudKit has: a **private database** for each user's personal data, a **shared database** (for any records other users have shared with them), and a public database (not needed for our use-case). In a sharing flow, the owner's private DB contains the original records and a `CKShare` record; once the invite is accepted, the participant's **shared DB** will contain a copy of those records (kept in sync with the owner) [5] [6]. Key points about CloudKit sharing:

- **CKShare Record:** A `CKShare` is a special kind of record that references the data being shared. You create a `CKShare` and save it **alongside the record(s)** you want to share in the owner's private database. (Important: the root record and the CKShare must be saved in the same operation to avoid errors.) The `CKShare` defines who the participants are and what permissions they have.

- **Sharing Scope:** You can share a single record (and its children) or an entire record zone. For a persistent parent–child pairing, a convenient approach is often to share a **custom record zone** that contains all data relevant to the pairing. Zone sharing (introduced in iOS 15) lets you share a whole zone so that any new records in that zone are automatically shared [7]. Alternatively, you can use a single root record with child records; CloudKit will include all records in the hierarchy below the root in the share [8]. Choose a model that fits your data (for example, sharing a "ChildDevice" record and linking all screen-time entries as children of that record).

- **Owner and Participants:** The device initiating the share is the **owner** of the data (the share resides in their private DB). The receiving side becomes a **participant** with access via their shared DB. Owners can have multiple participants in a share (up to 100 per Apple's limits), so one child device could theoretically share with two parent devices, etc., if needed. Each participant can have a permission level set by the owner – typically `.readWrite` (if the parent should also update data, e.g. send restrictions or feedback) or `.readOnly` (if the parent should only view data). The owner can configure these permissions on the CKShare for each participant or set a *public permission* for anyone with the link [9] [10].

- **UICloudSharingController:** Apple provides a built-in UI controller for sharing invites. **UICloudSharingController** presents a share sheet allowing the owner to add participants (via contacts) or generate a share link, and to set permissions, without writing much code [11] [12]. Under the hood it creates the CKShare and either sends an invite via Messages/Mail or gives a URL. For a custom QR-code flow, you might bypass the full UI and directly use the `CKShare` API to get the share URL (see below), but UICloudSharingController can still be used in a minimal way to set permissions or just to obtain the share link for QR. (It's also possible to add participants programmatically via `CKShare.Participant` and `CKFetchShareParticipantsOperation` if you prefer a fully custom flow [13] [14].)

Implementing the QR Code Pairing Flow

Using a **QR code** for pairing can make the user experience very straightforward: for example, the child's device shows a QR code on screen, and the parent's device scans it to instantly establish the link. Here's a step-by-step outline of an efficient implementation:

1. Setup CloudKit and Permissions: In Xcode, enable the iCloud capability (with CloudKit) for your app. Ensure both apps (or both instances of the app) use the same CloudKit container. In your app's Info.plist, set `CKSharingSupported` to `YES` [15] – this flag tells iOS that your app can handle CloudKit share invitations. It allows the system to route acceptance events to your app. Also be prepared to handle iCloud user auth: use `CKContainer.default().accountStatus` to check if the user is logged into an iCloud account, and prompt them if not. (If iCloud is unavailable, you'll need to disable pairing features.) It's also wise to handle the case of iCloud being turned off or the user revoking CloudKit permissions, using the `CKAccountChanged` notification to detect changes [16].

2. Initiating the Share on Device A (e.g. Child's Device): When pairing starts, Device A will create a record to share – e.g. a "ChildProfile" record containing the child's name or an identifier. This record should be saved in the owner's private database (possibly in a dedicated zone if using zone sharing). Next, create a `CKShare` object linked to that record. Save **both** the CKShare and the record together in one operation (e.g. using a `CKModifyRecordsOperation` or just `CKDatabase.save(_:)` for both). Once saved, CloudKit will assign a **share URL** to the CKShare. You can retrieve this URL either by examining the `CKShare.url` property [10] or via the completion handler of `UICloudSharingController` (if you used one). This URL is essentially an invitation link that any iCloud user can use to join the share.

3. Generating a QR Code: Take the CKShare's URL (it will be an `URL` object) and generate a QR code from it. You can use CoreImage filters (e.g. `CIFilter(name: "CIQRCodeGenerator")`) or Apple's Vision framework to generate the QR image. **Tip:** Make sure the URL is not too large; CloudKit share URLs are

reasonably short web links, so they fit in a QR code without issue. Some developers choose to embed a shorter code instead – for example, one app **WebbyGuard** describes generating a **temporary pairing code** and storing the share URL in CloudKit public DB keyed by that code [17] [18]. In their approach, the child app writes a public "pairing record" (with a unique code and the share URL), and the QR contains just that code. The parent scans the code, looks up the share URL from the public DB, and then proceeds to accept the share. This indirection can be useful for security (the link isn't directly visible) or to allow the code to expire after a short time, but it's optional. For simplicity, you can embed the actual share URL in the QR code and skip using the public database – scanning that URL will still let the parent join the share securely.

4. Scanning the QR Code on Device B (e.g. Parent's Device): On the parent's side, implement a QR scanning view. You can use **AVFoundation** (with `AVCaptureSession` and `AVMetadataOutput` for `.qr` codes) or the newer **VisionKit** DataScanner API for a SwiftUI-friendly approach. Either will give you the contents of the QR code as a string. When a URL string is obtained from the QR, you have two choices:

- **(a) Use System URL Handling:** Simply call `UIApplication.shared.open(url)` with the scanned CloudKit share URL. Because your app declared `CKSharingSupported` and if the URL is a valid CloudKit share link, iOS will prompt the user to accept the share and automatically invoke your app's delegate callback when done. This is essentially what happens if you send the link via Messages and the user taps it [19]. The system will handle retrieving the share metadata and calling your app. In your app delegate or scene delegate, implement `userDidAcceptCloudKitShareWith` to receive the `CKShareMetadata`. From there, call `CKContainer.default().acceptShare(metadata)` to finalize acceptance (if you use the default AppDelegate template, this call might be made for you). The Apple sample app demonstrates that when the user follows the link, the system launches the app and you accept the share in `SceneDelegate.userDidAcceptCloudKitShareWith` [20].

- **(b) Programmatic Acceptance in-app:** Alternatively, you can handle the acceptance within your app's flow (e.g. show a loading indicator during pairing). Use the CloudKit API to fetch share metadata and accept, rather than bouncing out to Safari. For example, Apple's Tech Talks show a method:

  ```
  func confirmShareParticipation(from url: URL) async throws {
      let container = CKContainer(identifier: <# your container ID #>)
      let shareMetadata = try await container.shareMetadata(for: url)
      try await container.accept(shareMetadata)
  }
  ```

  This code fetches the `CKShare.Metadata` for the invite URL and then accepts the share in one go [21]. You can do this asynchronously (as above) or with callbacks/operations (`CKFetchShareMetadataOperation` and `CKAcceptSharesOperation` if not using async/await). After this completes, the share is accepted *silently* without leaving the app UI. (Do note that even for programmatic acceptance, the user must be logged in to iCloud and will implicitly "join" the share – you cannot bypass the requirement that a user consents to join. However, by scanning the code and calling accept, you have that user action as consent.)

5. Completing the Pairing: Once the parent device has accepted, you now have a persistent link. The child's shared record(s) will appear in the parent's **shared database**. You can verify this by fetching from

`CKContainer.default().sharedCloudDatabase`. For example, you might fetch the shared zone or query for the shared child profile record. In Apple's sample, after acceptance, the shared contact appears under a "Shared" list in the second user's app [22]. In your case, you might show that the child device is now paired. This relationship will remain until someone explicitly breaks the share.

6. Post-Pairing Data Sync: With the share in place, any future data that needs to go from child to parent (or vice-versa) can be saved into the shared zone/record. For instance, the child app could create new "ScreenTimeAlert" records in the shared zone; the parent app, when online, can fetch or subscribe to changes in that zone to see those alerts in near-real-time. CloudKit will handle syncing those records to all participants. If the parent needs to send data back (e.g. a "limit reached, lock device" command or feedback on an alert), it can modify records in the shared zone as long as it has write permission. Because you likely gave the parent `.readWrite` access, the parent can add child records or edit fields. One practical tip: if adding new records to a shared zone, you must use the zone from the share (you cannot arbitrarily create a new zone on the participant's side). Usually, you get the zone ID from the `CKShareMetadata` or the shared record. Some developers store the share's recordID or zoneID for future use [23] [24]. But you can also query the shared database for the record by record name if known.

7. Handling Same-Account Pairing: In the case that the parent and child devices are using the **same iCloud account**, you won't use CKShare at all. Instead, both apps share the same private database data. You might still generate a QR code to easily designate one device as the "child" and one as the "parent" within your app's logic. For example, the child app could create a record with a unique ID (in private DB) and show a QR of that ID; the parent scans it and finds that record in its own copy of the private DB (since it's the same account) to confirm pairing. But strictly speaking, if it's the same user, you could skip QR entirely and just have the user press a button on each device to mark their roles, as all data is already shared via iCloud. The QR flow is most useful for cross-account pairing.

User Authentication and Share Permissions Considerations

iCloud Authentication: Both users must have iCloud accounts and CloudKit enabled. If the child device's user is not signed in or has iCloud Drive disabled, you cannot create or share records. Similarly, the parent must be signed in to accept the share. Be prepared to detect and inform the user if iCloud is unavailable (for example, by checking `CKContainer.default().accountStatus` on launch and when attempting pairing). Apple also recommends monitoring the `CKAccountChanged` notification to handle the case where a user signs out or switches accounts during usage [16].

Participant Identity & Permissions: When creating the CKShare, decide how you will invite the other user. There are two models: **specific invite or shareable link**. For a **specific invite**, you'd look up the user's identity (via email/phone, using `CKContainer.fetchShareParticipants`) and add them as a `CKShare.Participant` on the share, then the system will send an invite to that user's iCloud account. This requires you to know some identifier of the other user (and the user must have allowed discoverability by email or phone). In a parent-child scenario with a QR code, you typically **don't know the other's iCloud ID upfront**, so instead you'll use a **shareable link**. By setting the share's `publicPermission` to allow access (e.g. `.readWrite` or at least `.readOnly`), **anyone with the link can join** [9] [10]. This is the approach when sharing via QR code or general URL – it doesn't require email lookup. You can still secure it by making the link single-use or expiring. For instance, WebbyGuard's implementation writes the share info

to a temporary public record and deletes it after use [25] [18], effectively expiring the pairing code once the parent has scanned it.

When using UICloudSharingController, if you choose "Copy Link" or "Share Options", you can set whether the link is for "Anyone with the link" or "Only people you invite" and the permission (view or edit). For a seamless QR experience, configure the share for **"anyone with link" and allow edit** (if two-way communication is needed). That way the parent can join via QR without additional identity confirmation, and can modify shared data. If you only want parent to view data, set the participant permission to read-only before finalizing the share. (UICloudSharingController's delegate methods like `itemPermission` can set default permissions.)

User Experience: The user's acceptance step is crucial. Apple's security model requires that the invited user *consents* to accepting the share. In our QR flow, the act of scanning and perhaps tapping an "Accept" prompt in-app serves this purpose. Keep the UX clear: for example, on the parent app, after scanning the code you might show a message like "Join child's device data?" to confirm before calling the accept API. If using the system openURL method, iOS will show a dialog asking to open in the app and potentially another prompt to accept the share (especially if using the built-in UI). Once accepted, you should inform the user that pairing was successful.

Edge Cases: If the child cancels the share or the link is invalid/expired, handle those errors (CloudKit will return an error if you try to accept an invalid share metadata). If the parent tries to reuse a QR code that's already been accepted, you might detect that by the code being gone (in the public DB method) or an error on accept saying "already accepted." In practice, each share link can only be used once per participant – if needed to re-pair, the owner should generate a new CKShare or re-share the record (note: attempting to share a record that's already shared will throw an `alreadyShared` error [26], so you might reuse the existing CKShare if it still exists rather than creating duplicates).

Common Challenges and Developer Tips

Building a CloudKit-based pairing system comes with a few challenges that others have noted, but there are well-known solutions:

- **Ensuring the Share Flow Completes:** One common hurdle is correctly handling the acceptance callback. In the older UIKit app lifecycle, the **AppDelegate** method `application(_:userDidAcceptCloudKitShareWith:)` is called when the user accepts an invite (e.g. via the link) [27]. In a modern SwiftUI lifecycle app, there is no scene delegate by default, so you can do one of two things: use an `UIApplicationDelegateAdaptor` to hook into that delegate call, or handle the URL in `onOpenURL` and call the acceptance APIs manually. Several developers have pointed out that Apple hasn't (as of iOS 16/17) provided a pure SwiftUI approach for this, so using an app delegate is the workaround [28]. In any case, test that the acceptance is being processed – e.g., log or alert when you get the metadata and finish the `accept()` call – otherwise the share won't actually be added.

- **Testing with Multiple Accounts:** During development, you'll need at least two iCloud accounts (e.g. your personal Apple ID and a secondary test Apple ID) to try out the full flow. You can use a simulator or second device logged into the secondary account. The Apple sample project "CloudKit

Sharing" suggests using one device (or simulator) for *User One* and another for *User Two* [29] [30]. Ensure your iCloud container is set up with the proper entitlements and that both accounts have the app installed. If using Family Sharing, note that it doesn't bypass the invite process – even family members must accept a CloudKit share explicitly [4].

- **Dealing with CloudKit Constraints:** Keep in mind CloudKit operations are asynchronous and can fail (if the network is down, iCloud servers unreachable, etc.). Implement proper error handling and perhaps retry logic for critical ops like saving the share or accepting it. Also, CloudKit sharing is only for iCloud users – you cannot share to non-Apple accounts. The number of participants per share is limited (100), which is far above what a parent-child scenario needs (just something to be aware of). And if you anticipate the need to share a lot of data right after pairing, be mindful of CloudKit **quota and rate limits** – e.g., the first sync of many records might be slow. Use notifications (`CKSubscription`) or periodic fetch to get updates rather than continuously polling.

- **Permission Changes and Revocation:** As part of best practices, consider how to handle if a user revokes a share. CloudKit allows the owner to **remove a participant** or stop sharing entirely (by deleting the CKShare) [31] [32]. A participant can also stop participating (e.g., a parent could delete the shared record from their shared DB which removes their access [33]). In your app's UI, you might provide a "Unpair" option which on the owner side would remove the share (thus evicting the participant), or on the participant side simply stops showing shared data (and perhaps deletes local references). Ensure your app appropriately handles the scenario where the share is gone – e.g., stop expecting data, maybe inform the user. If using UICloudSharingController, you get a built-in UI for managing participants and stopping sharing [34], which you could present in an "Manage pairing" screen for the owner.

- **Data Consistency:** When both devices can modify shared data, conflicts could theoretically occur (CloudKit will attempt to merge or you may get version conflicts). For simple parent-child use (e.g., child generates events, parent only reads or sends simple commands), conflicts are rare. But if you do collaborative edits, be sure to handle CKError.serverRecordChanged by refetching and merging changes as needed. This is more of a concern for collaborative document apps than for a screen time controller, but it's worth noting.

- **Real-World Example:** *WebbyGuard*, a real App Store app for parent-child web safety, uses **precisely** this technique. According to their documentation, the child app generates a QR code that encodes a unique identifier, writes a temporary record with that and the share info to CloudKit public DB, and the parent app scans it to accept a CloudKit share. The share establishes a secure link between the two iCloud accounts, and thereafter the child's alerts are sent through a shared CloudKit zone to the parent [17] [35]. This confirms that Apple's CloudKit + QR approach is being used successfully in production for persistent pairing. Apple's own sample code on GitHub ("CloudKit Sharing") is also a great reference to see the sharing mechanics and acceptance flow in action [36] [37].

Additional Resources and References

- **Official Apple Documentation:**
- *Sharing CloudKit Data with Other iCloud Users* – Apple's guide explaining how to share records, use UICloudSharingController, and handle acceptance [38] [19].

- *CKShare, CKShare.Metadata, CKAcceptSharesOperation* – API reference docs for the CloudKit sharing classes (useful for details on properties like `publicPermission`, and how to fetch/accept shares).
- *UICloudSharingController* – Documentation on using the built-in sharing UI (helpful if you want the default interface for invites).

- **Apple Sample Code:**

- *CloudKit Sharing Example (Apple Sample Code Project)* – A full Xcode project demonstrating record sharing between two users [39] [5]. It shows creating a share, sending the invitation link, and accepting it on another device, as well as fetching shared records. This can be a template for implementing your pairing logic.
- *WWDC Videos:* **"Get the most out of CloudKit Sharing" (Tech Talks 2023)** – covers best practices, including zone sharing and using share URLs programmatically [40] [21]. **"What's New in CloudKit" (WWDC21)** – introduces zone sharing and improvements in sharing APIs. These videos (and transcripts on developer.apple.com) provide insight into designing a good sharing UX.

- **Developer Articles & Tutorials:**

- *"SwiftUI: Simple Content Collaboration with iCloud" by Itsuki* – A 2025 tutorial walking through building a multi-user CloudKit sharing app, with code samples for creating CKShare, generating share links, and syncing data. It also discusses problems encountered and how to solve them (e.g. needing a custom record zone, handling share UI updates) [41] [42].
- *Swift with Majid – Zone sharing in CloudKit* – Explains the difference between record sharing and zone sharing, with code examples. Useful if you plan to share a whole zone (for persistent pairing, a zone share can simplify syncing many records) [7].
- *Kodeco (Ray Wenderlich) – "Sharing Core Data with CloudKit in SwiftUI"* – While focused on Core Data, the latter sections show how to invite users to a share and accept the invitation in a SwiftUI context [43] [44]. It's a practical guide that might illuminate the CloudKit pieces even if you aren't using Core Data.

By combining CloudKit's robust sharing capabilities with a user-friendly QR code exchange, you can implement a secure and persistent device pairing system. This approach handles the heavy lifting of data syncing and access control through iCloud, while keeping the user experience as simple as "scan this code to connect." Developers who have built similar systems report that once the initial hurdles of CloudKit setup and share acceptance are resolved, the solution is reliable and scales well for real-world use [25] [35]. Good luck with building your pairing system!

---

1 5 19 20 22 29 30 36 37 38 39 GitHub - apple/sample-cloudkit-sharing
https://github.com/apple/sample-cloudkit-sharing

2 6 7 8 9 10 11 12 13 14 15 21 31 32 33 34 40 Get the most out of CloudKit Sharing - Tech Talks - Videos - Apple Developer
https://developer.apple.com/videos/play/tech-talks/10874/

3 4 General Questions about CloudKit and Family Sharing : r/iOSProgramming
https://www.reddit.com/r/iOSProgramming/comments/szds7m/general_questions_about_cloudkit_and_family/

16 Handling Account Status Changes With CloudKit - Cocoacasts
https://cocoacasts.com/handling-account-status-changes-with-cloudkit

17 18 25 35 Privacy Policy - WebbyGuard
https://privacy.webbyguard.app/

23 24 27 CloudKit Series — Participant Acceptance | by Cory D. Wiles | Medium
https://kwylez.medium.com/cloudkit-series-participant-acceptance-f39a8d7f7a53

26 41 42 SwiftUI: A Simple Simultaneous Content Collaboration App with iCloud | by Itsuki | Level Up Coding
https://levelup.gitconnected.com/swiftui-a-simple-simultaneous-content-collaboration-app-with-icloud-b50802aa4007?gi=27cef9e0ee55

28 How to accept CloudKit shares with the new SwiftUI app lifecycle?
https://stackoverflow.com/questions/63296317/how-to-accept-cloudkit-shares-with-the-new-swiftui-app-lifecycle

43 Accepting Share Invitations in a SwiftUI App - Apple Developer
https://developer.apple.com/documentation/coredata/accepting-share-invitations-in-a-swiftui-app

44 SwiftUI + CloudKit + Sharing? - Reddit
https://www.reddit.com/r/SwiftUI/comments/1gq876r/swiftui_cloudkit_sharing/