Sample Code

Incorporating ClassKit into an Educational App

Walk through the process of setting up assignments and recording student progress.

Download

iOS 11.3+ | iPadOS 11.3+ | Xcode 10.2+

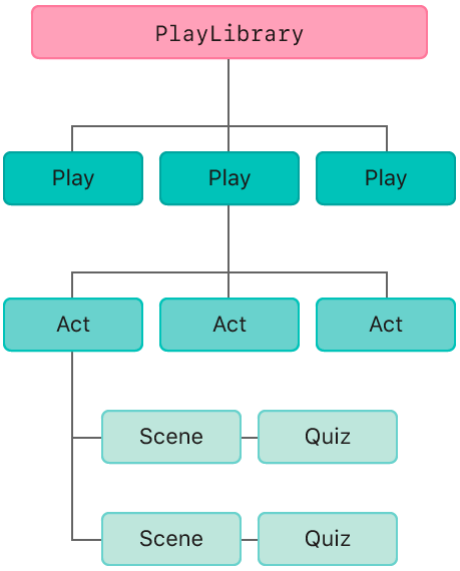
Overview

You adopt ClassKit in an existing educational app to enable teachers to create assignments and monitor students' progress through those assignments. This sample code project demonstrates ClassKit adoption in an app that lets users read plays.

Before starting, be sure to read Enabling ClassKit in Your App to learn how to configure your environment to work with ClassKit, and Testing Your App During Development to prepare to debug your ClassKit adoption.

Start with an Existing Educational App

The GreatPlays app provides a navigable hierarchy of plays, acts, and scenes, along with quizzes that test the reader's comprehension. The app uses a simple data model that represents a collection of plays—the shared `PlayLibrary` instance holds `Play` instances, each of which contain an array of `Act` instances, and so on. These all exist independent of ClassKit.



For this example, the structure of a single play—Shakespeare's *Hamlet*—is added to the library at launch, from inside the `application( _:didFinishLaunchingWithOptions:)` method.

> **Note**
>
> When writing an app for the educational market, consider supporting shared iPad, as described in Optimizing Apps for Shared iPad. This sample app does that by not using any persistent local storage and by setting the `NSSupportsPurgeableLocalStorage` key to `YES` in its `Info.plist` file.

In a real app, in addition to the structure, you would also add the play's text, along with quiz questions tailored to each scene. You might also support other plays, either distributed with the app, or downloaded later.

Define Assignable Content

Your first task in adopting ClassKit is to define your app's assignable content. You represent a unit of assignable content to ClassKit as a `CLSContext` instance, and then establish relationships between contexts by grouping them together into a hierarchy. For the play reader, teachers might want to assign a quiz, an individual scene, an act (with all its scenes), or even the whole play. So the existing model hierarchy provides a good template for a context hierarchy.

Because ClassKit layers on top of what your app already does, it's often best to isolate ClassKit support into class extensions. This scheme avoids disrupting the app's normal flow. The sample app therefore declares a `Node` protocol that model objects can adopt in an extension to readily associate with a related context:

```
protocol Node {
    var parent: Node? { get }
```

```
    var children: [Node]? { get }
    var identifier: String { get }
    var contextType: CLSContextType { get }
}
```

In adopting this protocol, a model object discloses its immediate ancestor and descendants, a unique identifier, and a `CLSContextType` value that indicates what kind of content it contains. For example, the `Node` extension to `Act`, shown below, defines its `parent` as the `play` that contains it, and its `children` as the `scenes` it contains. It provides an identifier that is unique to the act, and a context type of chapter, which is a reasonable approximation of the role of an act within a play.

```
extension Act: Node {
    var parent: Node? {
        return play
    }

    var children: [Node]? {
        return scenes
    }

    var identifier: String {
        return "Act \(number)"
    }

    var contextType: CLSContextType {
        return .chapter
    }
}
```

Further, an extension to the `Node` protocol provides default behavior to all model objects for handling identifiers. In particular, a model object that adopts the protocol gains the ability to report its own identifier path (a collection of `identifier` strings that trace through the hierarchy from one node to another), and to find a descendant node from an identifier path:

```
extension Node {
    var identifierPath: [String] {
        var pathComponents: [String] = [identifier]

        if let parent = self.parent {
            pathComponents = parent.identifierPath + pathComponents
```

```
        }

        return pathComponents
    }

    /// Finds a node in the play list hierarchy by its identifier path.
    func descendant(matching identifierPath: [String]) -> Node? {
        if let identifier = identifierPath.first {
            if let child = children?.first(where: { $0.identifier == identifier }) {
                return child.descendant(matching: Array(identifierPath.suffix(identi
            } else {
                return nil
            }
        } else {
            return self
        }
    }
}
```

Advertise Your Content to Teachers

Contexts are the mechanism by which your app advertises its assignable content to teachers. Contexts you tell ClassKit about appear as tasks in Apple's Schoolwork app, where teachers go to create assignments based on your content. So it's important to declare contexts as soon (and as atomically) as possible. Otherwise teachers won't see your app's content in Schoolwork, or might only see a partial list of tasks.

You deal with this by declaring the context hierarchy for static content at application launch, or immediately after you download dynamic content. In the play reader app, you do this by making context declaration an integral step of building a new play instance in the `addPlay` method:

```
func addPlay(_ play: Play, creatingContexts: Bool = true) {
    if !plays.contains(where: { $0.title == play.title }) {
        plays.append(play)

        // Give ClassKit a chance to set up its contexts.
        if creatingContexts {
            setupContext(play: play)
        }
    }
}
```

You declare an entire play context hierarchy by asking the data store for all the leaf nodes, which implicitly also declares all of the contexts that are ancestors of the leaf nodes:

```
func setupContext(play: Play) {
    for act in play.acts {
        for scene in act.scenes {

            // Get the deepest path: the quiz if there is one, or the scene if not.
            let path = scene.quiz?.identifierPath ?? scene.identifierPath

            // Asking for a context causes it (and its ancestors) to be built, as ne
            CLSDataStore.shared.mainAppContext.descendant(matchingIdentifierPath: pa
        }
    }
}
```

Because you're only declaring the contexts at this point, you don't need to do anything with the returned values.

Build Contexts on Demand

Any time you ask the data store (`CLSDataStore`) for a context, whether during declaration or because you want to activate the context, the data store first looks in its database of stored contexts. If the context is available there, perhaps from a previous launch of your app, the data store returns that. But if it's not available, the data store asks its `delegate` to build the context.

By defining contexts that parallel your model hierarchy, you facilitate the building of new contexts. In your implementation of the `CLSDataStoreDelegate` protocol's `createContext(forIdentifier:parentContext:parentIdentifierPath:)` method, you can use characteristics of your model objects to inform context creation.

In the play reader, the shared instance of the `PlayLibrary` class takes the role of delegate, again using an extension. Its `ClassKit` extension includes the `setupClassKit()` method that assigns itself as the delegate:

```
func setupClassKit() {
    CLSDataStore.shared.delegate = self
}
```

The extension also implements the delegate callback, relying on each model object's `Node` extension to provide data needed for context creation:

```
func createContext(forIdentifier identifier: String, parentContext: CLSContext, pare

    // Find a node in the model hierarchy based on the identifier path.
    let identifierPath = parentIdentifierPath + [identifier]

    guard let playIdentifier = identifierPath.first,
        let play = PlayLibrary.shared.plays.first(where: { $0.identifier == playIder
        let node = play.descendant(matching: Array(identifierPath.suffix(identifierP
        return nil
    }

    // Use the node to create and customize a context.
    let context = CLSContext(type: node.contextType, identifier: identifier, title:
    context.topic = .literacyAndWriting

    // Users of 11.3 rely on a user activity instead.
    if #available(iOS 11.4, *),
        let path = identifierPath.joined(separator: "/").addingPercentEncoding(withA

        // Use custom URLs to locate activities.
        //  Comment this assignment to rely on a user activity for all users.
        context.universalLinkURL = URL(string: "greatplays://" + path)
    }

    // No need to save: the framework handles that automatically.
    os_log("%s Built", node.identifierPath.description)
    return context
}
```

Build Contexts from an App Extension

The app includes a ClassKit context provider app extension by defining a target called `ClassKit ContextProvider`. A ClassKit extension's primary class conforms to the `CLSContext Provider` protocol. Schoolwork calls this protocol's `updateDescendants` method to create or update the children of a given context as the teacher browses assignable content. This enables Schoolwork to advertise the most up-to-date version of an app's assignable content, even before the teacher runs the main app for the first time.

Using the identifier path of the passed-in context, the update method finds the corresponding node in the data model, and then finds the children of that node:

```
// Start with plays as child nodes. Then look for a descendant.
var childNodes: [Node] = PlayLibrary.shared.plays

if let identifier = identifierPath.first,
    let play = PlayLibrary.shared.plays.first(where: { $0.identifier == identifier }
    let node = play.descendant(matching: Array(identifierPath.suffix(identifierPath.

    childNodes = node.children ?? []
}
```

The update method then finds existing child contexts at the same hierarchical level as the node, and creates any that are missing:

```
let predicate = NSPredicate(format: "%K = %@",
                            CLSPredicateKeyPath.parent as CVarArg,
                            context)
CLSDataStore.shared.contexts(matching: predicate) { childContexts, _ in
    for childNode in childNodes {
        if !childContexts.contains(where: { $0.identifier == childNode.identifier }
            let childContext = PlayLibrary.shared.createContext(forIdentifier: child
                                                                parentContext: conte
                                                                parentIdentifierPath

            context.addChildContext(childContext)
        }
    }
```

In this app, contexts never change, so the loop moves to the next iteration without taking any action when it finds an existing context. If your app has contexts that can change, use this opportunity to reconfigure the context. Either way, after finishing the loop, save the updates and call the completion handler:

```
    CLSDataStore.shared.save { error in
        if let error = error {
            os_log("Save error: %s", error.localizedDescription)
        } else {
            os_log("Saved")
        }
        completion(error)
    }
}
```

Record Progress with Activities

While contexts declare the structure of your app, you use `CLSActivity` instances to report progress through those contexts. For example, for a context representing a scene, the corresponding activity reports how much of the scene the student has read and how long they took to read it.

In addition to the identifier extension, the sample app includes another extension to `Node` that provides default behavior for working with ClassKit activities. Model objects use their own identifier path to locate the matching context, and then use the context to manage activities. For example, the `startActivity()` method defined in the extension begins an activity:

```
func startActivity(asNew: Bool = false) {
    os_log("%s Start", identifierPath.description)

    CLSDataStore.shared.mainAppContext.descendant(matchingIdentifierPath: identifier

        // Activate the context.
        context?.becomeActive()

        if asNew == false,
            let activity = context?.currentActivity {

            // Re-start the existing activity
            activity.start()

        } else {
            // Create and start an activity.
            context?.createNewActivity().start()
        }

        CLSDataStore.shared.save { error in
            guard error == nil else {
                os_log("%s Start save error: %s", self.identifierPath.description, e
                return
            }
        }
    }
}
```

`Node` also defines methods for reporting progress as a fraction of task completion and ending an activity. Notice that all these methods retrieve the context every time, rather than storing a

reference to it. It's important to do this, because the underlying instance could change between calls as a result of network synchronization.

Start Recording When the User Begins an Activity

You typically call methods to record activity from your view controllers. Consider an assignment to read a particular scene. The scene's view controller knows when the scene appears on screen and has a handle on the scene instance. So the controller is in the best position to tell the scene to begin recording an activity from its `viewDidAppear(_:)` method:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    scene?.startActivity()
}
```

Notice that when the student starts to read a scene, the controller doesn't set the `startActivity()` method's `asNew` parameter, leaving it to have the default value of `false`. As a result, a previously stopped activity, if available, is resumed. This allows the user to begin reading a scene, then navigate elsewhere in the app (for example to review an earlier scene) without finalizing the current attempt. When the user returns, progress and duration pick up from where the user left off.

In contrast, the `startQuiz()` method does set the `startActivity()` method's `asNew` parameter to `true`. Quizzes, once started, must be finished before the user can move to another task. So the app treats each new attempt as a new activity.

```
func start() {
    startActivity(asNew: true)
}
```

How you handle this for a particular activity depends on the characteristics of the tasks you define.

Add Progress When the User Scrolls

While the scene view controller (which manages a scroll view) is visible, it uses its knowledge of the content offset as an indication of how far through the scene the student has read. For each scroll view delegate update, the controller reports a new progress value to the scene by measuring scroll position as a proxy for how much the student has read:

```
func scrollViewDidScroll(_ scrollView: UIScrollView) {
    let position = sceneText.contentOffset.y + sceneText.frame.size.height
    let total = sceneText.contentSize.height

    // The scroll view can bounce, so use care to bound the progress.
    let progress = Double(max(0, min(1, position / total)))

    scene?.update(progress: progress)
}
```

Stop Recording When the User Stops an Activity

The controller informs the scene when the activity is over in its `viewWillDisappear(_:)` method:

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    scene?.stopActivity()
}
```

Report Additional Metrics with Activity Items

Activities report duration and progress automatically. But sometimes you want to provide additional metrics about an activity. For example, you might want to report a quiz score, or record how many times a hint was used in solving a problem. For this, you use activity items.

In GreatPlays, the `Node` protocol extension provides the `addScore()` and `addQuantity()` methods for this purpose. These are called when the user completes the quiz, but before ending the activity, to report the quiz results:

```
func record() {
    // The score is the primary metric for a quiz.
    addScore(score, title: "Score", primary: true)
    addQuantity(Double(hints), title: "Hints")
    markAsDone()
    stopActivity()
}
```

Mark an Activity as Done

While recording the quiz score, the app also calls the `markAsDone()` method, as shown in the previous section, which in turn calls the data store's `completeAllAssignedActivities(matching:)` method to indicate that the student has finalized the attempt.

```
func markAsDone() {
    if #available(iOS 12.2, *) {
        os_log("%s Done", identifierPath.description)
        CLSDataStore.shared.completeAllAssignedActivities(matching: identifierPath)
    }
}
```

The student can't go back and change anything after reaching the end of the quiz, so the app can safely mark the activity as done. In contrast, reading a scene has no easily detected end point, and so the app doesn't make a call to the completion method in that case. Instead, the student decides when to mark the activity complete in Schoolwork.

Designate a Primary Activity Item

When you add activity items, you can choose to make one of them the primary item. The primary gets a more prominent role in summarized results that teachers see. In the play reader example, for a quiz, the score is considered the primary item, as shown above. Alternatively, you can choose not to set any activity item as the primary, in which case progress becomes the most prominent result displayed to teachers. The `addScore()` method demonstrates the two ways in which you can register an activity item, either as primary or not:

```
if primary {
    activity.primaryActivityItem = item
} else {
    activity.addAdditionalActivityItem(item)
}
```

If you do decide to set a primary item, make sure you always set the same kind of primary item for a given activity. For example, once you register the score item as the primary for a quiz activity, you must always use score this way. Making the hint quantity the primary at a later time generates an error.

See Also

Essentials

Enabling ClassKit in your app

Prepare your app and your development environment to adopt ClassKit.

`ClassKit Environment Entitlement`

The ClassKit development or production environment for an education app that works with the Schoolwork app.

`class CLSDataStore`

A container for all the ClassKit data in your app.