Building widgets for iOS applications with WidgetKit and SwiftUI

Starting with iOS 14, Apple introduced widgets that allow users to show a piece of the app's content on the home screen.

Widgets display relevant non-interactive content letting users quickly open the app for more details. Widgets support multiple sizes and are built using SwiftUI views.

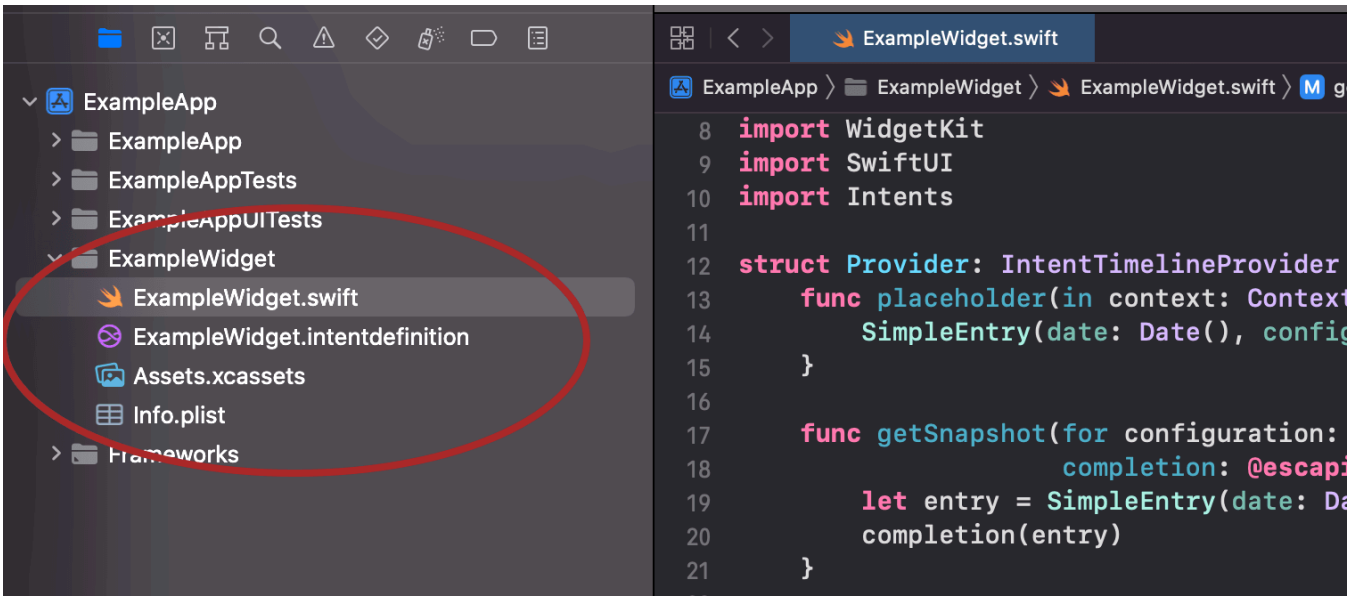
Let's dive in and see how we can use the `WidgetKit` framework to add widgets for an iOS application.

1. Adding a widget extension

A starting point for creating a widget is the `Widget Extension` template.

1. Select `File > New > Target` in Xcode's menu.
2. Search for and select `Widget Extension`.
3. Enter the name for the extensions.

After confirming, Xcode creates a new widget target with the following structure:



Structure of the new widget target.

Configuring the widget

The created `ExampleWidget.swift` file provides sample code for the widget configuration. The entry point for the widget extension conforms to the `Widget` protocol and is attributed with `@main`:

```
@main
struct ExampleWidget: Widget {

    var body: some WidgetConfiguration {
        // return configuration
    }
}
```

When returning a widget configuration, we can choose between two different widget types:

- `StaticConfiguration` for widgets with **no user-configurable properties** i.e. for widgets that display the same content for all users.
- `IntentConfiguration` for widgets with **user-configurable properties** i.e. for widgets that display content depending on some user input like their location, preferences or any other input.

Widgets with `StaticConfiguration`

In case our widget is independent from any user properties, we can return a `StaticConfiguration` instance:

```
var body: some WidgetConfiguration {
    StaticConfiguration(
        kind: "com.tanaschita.ExampleWidget",
        provider: ExampleTimelineProvider()) { entry in
            ExampleWidgetEntryView(entry: entry)
    }
    .configurationDisplayName("My Widget")
}
```

To do that, we provide the following initialiser parameters:

- `kind` - Widget's identifier.
- `provider` - An object that conforms to `TimelineProvider` and tells `WidgetKit` when to update the content of the widget.
- `content` - A closure that contains a SwiftUI view with the widget's content.

We can use modifiers for additional configuration details like a display name, description or supported families.

```
.configurationDisplayName("My Widget")
.description("This is an example widget.")
.supportedFamilies([.systemSmall, .systemMedium, .systemLarge, .syst
```

Widgets with `IntentConfiguration`

If the widget's content is dependent on some user properties, we can return an `IntentConfiguration` object instead.

The initializer looks exactly the same as for `StaticConfiguration` with one additional parameter called `intent` that defines user-configurable properties.

Updating the widget's content with a timeline provider

When we created a `WidgetConfiguration` in the code above, we passed in an instance of `ExampleTimelineProvider` that conforms to the `TimelineProvider` protocol.

A timeline provider generates a timeline that consists of timeline entries. The following methods need to be implemented to conform to the protocol:

```
struct ExampleTimelineProvider: TimelineProvider {
    typealias Entry = ExampleTimelineEntry

    // Provides a timeline entry representing a placeholder version
```

```
    func placeholder(in context: Context) -> ExampleTimelineEntry {
    }

    // Provides a timeline entry that represents the current time ar
    func getSnapshot(in context: Context, completion: @escaping (Exa
    }

    // Provides an array of timeline entries for the current time ar
    func getTimeline(in context: Context, completion: @escaping (Tim
    }
}
```

Each time line entry specifies a date when to update the widget's content and contains any custom properties needed to display the widget.

```
struct ExampleTimelineEntry: TimelineEntry {
    let date: Date
    let title: String
}
```

Let's look at each method that needs to be implemented for the timeline provider.

1. `placeholder(in:)`

WidgetKit may need to render the widget's content as a placeholder giving the user a general idea of what the widget shows, for example:

- while the actual data is loading in the background
- when we enable the Data Protection capability for the widget extension to hide sensitive information
- when a user chooses to display the placeholder in certain presentation contexts such as on the Lock Screen

```
func placeholder(in context: Context) -> ExampleTimelineEntry {
      return ExampleTimelineEntry(date: Date(), title: "Quote of the
  }
```

2. `getSnapshot(in:completion:)`

WidgetKit calls this method when the widget appears in transient situations. The `completion` handler should be called as quickly as possible supplying sample data if the data for the widget's current state is not available yet.

```
var quoteOfTheDay: String?

func getSnapshot(in context: Context, completion: @escaping (Example
    completion(ExampleTimelineEntry(date: Date(), quoteOfTheDay: quo
}
```

3. `getTimeline(in:completion:)`

After requesting the initial snapshot, WidgetKit calls this method to request a regular timeline from the provider.

```
func getTimeline(in context: Context, completion: @escaping (Timelin
    loadQuoteOfTheDay { quoteOfTheDay in
        let entry = ExampleTimelineEntry(date: Date(), quoteOfTheDay
        let timeline = Timeline(entries: [entry],
                                policy: .after(Date.tomorrow) )
        completion(timeline)
    }

}
```

The timeline consists of one or more timeline entries and a reload policy informing WidgetKit when to request the next timeline.

Building the widget's view with SwiftUI

The last step is to build the widget view with SwiftUI. We initialised the view when returning a `WidgetConfiguration` earlier passing in a time line entry from the time line provider.

We can now use this entry to display relevant information to the user:

```
struct ExampleWidgetEntryView: View {
    var entry: ExampleTimelineProvider.Entry

    var body: some View {
        Text(entry.quoteOfTheDay)
    }
}
```

When adding a widget, users choose a specific family from the ones the widget supports. WidgetKit sets the chosen family in the SwiftUI environment.

```
struct ExampleWidgetEntryView : View {
    @Environment(\.widgetFamily) var family: WidgetFamily
    var entry: ExampleTimelineProvider.Entry

    @ViewBuilder
    var body: some View {
        switch family {
        case .systemSmall: Text(entry.quoteOfTheDay)
        case .systemMedium: ...
        // Handle all cases the widget supports
        }

    }
}
```

After adding the environment property, we can provide a different view for each family our widget supports.

Responding to user's interactions

Since widgets present read-only information, user interaction is limited to tapping on a widget. When a user does that, the system launches our app to handle the request.

We can add a URL by using the `widgetURL(_:)` view modifier.

```
Text(entry.quoteOfTheDay)
  .widgetURL(entry.widgetURL)
```

When a user taps on the widget, we can use the provided URL for example to root the user to a certain part of the app. For that, we can either use the

- `application(_:open:options:)` method when working with AppDelegate or
- `onOpenURL(perform:)` method on a SwiftUI view