

[App Intents](#) / Accelerating app interactions with App Intents

## Sample Code

# Accelerating app interactions with App Intents

Enable people to use your app's features quickly through Siri, Spotlight, and Shortcuts.

Download

iOS 18.1+ | iPadOS 18.1+ | macOS 15.1+ | visionOS 2.1+ | watchOS 11.0+ | Xcode 26.0+

## Overview

The app in this sample code project provides information on trails, allowing people to check on conditions, search for trails that allow activities like skiing, and record which trails they visit. Expressing these features as intents allows people to use them through Siri, Spotlight search, and Shortcuts. Additionally, the project integrates workout tracking on Apple Watch, and shows how to implement Action button support on Apple Watch Ultra. The intents also appear as actions in the Shortcuts app. People can combine these actions to build entirely new features in Shortcuts because the intents provide custom data types that match each other's inputs.

## Identify common actions

The sample app includes two key features that people are likely to use frequently: looking up information on a trail, and recording activity on a trail. To make it easy for people to use these features without even opening the app, the sample code creates intents for them to use with Siri, Spotlight search, and Shortcuts. For example, if someone saves their favorite trails in the app and wants to get the current conditions for those trails, the app implements the `OpenFavorites` structure, which conforms to `AppIntent`. When someone runs this intent, the app opens and navigates to the Favorites view.

```

/// Each intent needs to include metadata, such as a localized title. The title of t
static let title: LocalizedStringResource = "Open Favorite Trails"

/// An intent can optionally provide a localized description that the Shortcuts app
static let description = IntentDescription("Opens the app and goes to your favorite

/// Tell the system to bring the app to the foreground when the intent runs.
static let openAppWhenRun: Bool = true

/**
When the system runs the intent, it calls `perform()`.

Intents run on an arbitrary queue. Intents that manipulate UI need to annotate `per
so that the UI operations run on the main actor.
*/
@MainActor
func perform() async throws -> some IntentResult {
    navigationModel.selectedCollection = trailManager.favoritesCollection

    /// Return an empty result, indicating that the intent is complete.
    return .result()
}

```

## Create App Shortcuts

People may ask Siri to show their favorite trails, or they may find this suggested action through a Spotlight search. To support both of these options, the app implements an [AppShortcut](#) using `OpenFavorites`. An App Shortcut combines an intent with phrases people may use with Siri to perform the action, and additional metadata, such as an icon, and then uses this information in a Spotlight search. People can invoke the App Shortcut with a suggested phrase, or other similar words, because the system uses a semantic similarity index to help identify people's requests — automatically matching phrases that are similar, but not identical.

```

AppShortcut(intent: OpenFavorites(), phrases: [
    "Open Favorites in \(.applicationName)",
    "Show my favorite \(.applicationName)"
],
shortTitle: "Open Favorites",
systemImageName: "star.circle")

```

To register the App Shortcut with the system, the app calls `updateAppShortcutParameters` on its `AppShortcutsProvider` during the `init` of the `App` structure.

To aid the system's presentation of the App Shortcut, the sample app includes a short title and an SF Symbols name that represent the App Shortcut. Further, the sample app's `Info.plist` file declares `NSAppIconActionTintColorName` with the app's primary color and two contrasting colors in an array for the `NSAppIconComplementingColorNames` key. The system uses these colors when displaying the App Shortcuts, such as in Spotlight or the Shortcuts app. The specified values of the color names for these keys come from the app's asset catalog.

After registering an App Shortcut with the system, people can begin using the intent through Siri without any further configuration. To teach people a phrase to use the intent, the app provides a `SiriTipView` in the associated view.

```
SiriTipView(intent: OpenFavorites(), isVisible: $displaySiriTip)
```

The `SiriTipView` takes a binding to a visibility Boolean so that the app hides the view if an individual chooses to dismiss it.

Aside from intents for people to quickly view their favorite trails and track their workouts, the sample app provides extensive search capabilities through intents. The app doesn't provide App Shortcuts for intents that people use less commonly. Best practice is to provide App Shortcuts for only the most common actions in an app — usually between two and five intents, and not more than ten.

## Design custom responses

Even though the app doesn't provide `GetTrailInfo` as an App Shortcut, people may still interact with it through Siri, such as including the intent in a shortcut they create in the Shortcuts app. For a good user experience, this intent provides its result with a visual response using a custom UI snippet, and as a dialog for Siri to communicate the same information. It does so by conforming the return type of the intent's `perform` function to both `ProvidesDialog` and `ShowsSnippetView`.

```
func perform() async throws -> some IntentResult & ReturnsValue<TrailEntity> & ProvidesDialog
```

The app provides both visual experiences and voice-only experiences because people may be in a context where they can't see information in a custom UI (such as when the intent runs on HomePod), or when displaying the custom UI may be inappropriate (such as when the intent runs through CarPlay). This implementation provides a custom UI with a shorter supporting dialog to use when the custom UI is visible, and a different dialog containing additional information if the system can't show the snippet. The sample uses a transparent background for the custom UI

because the system displays it over a translucent background material. Avoiding opaque backgrounds provides the best results.

```
let snippet = TrailInfoView(trail: trailData, includeConditions: true)

/**
 This intent displays a custom view that includes the trail conditions as part of the
 the system can only read the response, but not display it. When the system can display
 conditions.
 */
let dialog = IntentDialog(full: "The latest reported conditions for \((trail.name) in
                        supporting: "Here's the latest information on trail conditions"

return .result(value: trail, dialog: dialog, view: snippet)
```

This sample app provides custom dialog throughout its intents. `SuggestTrails` validates the parameters that people provide and uses the custom dialog to prompt them for additional information. For example, if the provided location parameter isn't specific enough, the intent prompts the individual to choose from a list of locations related to their input. The app does this by throwing `needsDisambiguationError` with a value for the dialog parameter.

```
let dialog = IntentDialog("Multiple locations match \((location). Did you mean one of
let disambiguationList = suggestedMatches.sorted(using: KeyPathComparator(\.self, compare: <
throw $location.needsDisambiguationError(among: disambiguationList, dialog: dialog)
```

## Add parameters to an intent

An app intent can optionally require certain parameters to complete its action. For example, the `GetTrailInfo` intent declares a `trail` parameter by decorating the property with the `IntentParameter` property wrapper.

```
@Parameter(title: "Trail", description: "The trail to get information for.")
var trail: TrailEntity
```

The system supports parameters using common Foundation types, such as `String`, and those for custom data types in an app. The app makes its trail data available in an app intent through the `TrailEntity` type, which is a structure conforming to the `AppEntity` protocol.

To allow the system to query the app for `TrailEntity` data, the entity implements the `Identifiable` protocol with values that are stable and persistent. `TrailEntity` declares

defaultQuery, which the system uses to perform queries to receive `TrailEntity` structures.

```
static let defaultQuery = TrailEntityQuery()
```

An `AppEntity` makes its properties available to the system by decorating it with the `EntityProperty` property wrapper.

```
/**
 The trail's name. The `EntityProperty` property wrapper makes this property's data
 such as when an intent returns a trail in a shortcut.

 The system automatically generates the title for this property from the variable name.
 Generated titles are available for both `EntityProperty` and `IntentIntentParameter`.
 */
@property var name: String

/**
 A description of the trail's location, such as a nearby city name, or the national
 park name.

 If you want the displayed title for the property to be different from the variable name,
 use the `title` parameter of the `EntityProperty` property wrapper.
 */
@property(title: "Region")
var regionDescription: String
```

## Provide the app's data through queries

The system queries the app for its trail data through `TrailEntityQuery`, a type conforming to `EntityQuery`. For example, if someone saves a specific value as the trail parameter for `GetTrailInfo`, the system locates the `TrailEntity` by using the `defaultQuery` and requesting the entity by its ID from the `Identifiable` protocol. All types conforming to `EntityQuery` need to implement this method.

```
func entities(for identifiers: [TrailEntity.ID]) async throws -> [TrailEntity] {
    Logger.entityQueryLogging.debug("[TrailEntityQuery] Query for IDs \(identifiers)")

    return trailManager.trails(with: identifiers)
        .map { TrailEntity(trail: $0) }
}
```

The app also provides a list of common trail suggestions by implementing the optional `suggestedEntities` function.

```
func suggestedEntities() async throws -> [TrailEntity] {
    Logger.entityQueryLogging.debug("[TrailEntityQuery] Request for suggested entities")

    return trailManager.trails(with: trailManager.favoritesCollection.members)
        .map { TrailEntity(trail: $0) }
}
```

There are several subprotocols to `EntityQuery`, each of which enables different types of functionality. The sample app implements all of them for demonstration purposes, but a real app can use only the ones that meet its needs.

The app implements `EntityStringQuery` to help people configure `GetTrailInfo`. When people configure this intent in the Shortcuts app, they first see the list of trails from suggested Entities. The Shortcuts app provides a search field, enabling people to search for results that appear in the list of suggested trails. The app provides results for the search term by implementing `entities(matching:)`.

```
func entities(matching string: String) async throws -> [TrailEntity] {
    Logger.entityQueryLogging.debug("[TrailEntityQuery] String query for term \(string)")

    return trailManager.trails { trail in
        trail.name.localizedCaseInsensitiveContains(string)
    }.map { TrailEntity(trail: $0) }
}
```

## Enable Find intents

Apps implementing either the `EnumerableEntityQuery` or the `EntityPropertyQuery` protocol automatically add a Find intent in the Shortcuts app. These intents enable people to build powerful new features for themselves in Shortcuts, powered by the app's data — without requiring the app to implement that feature itself. For example, the sample app focuses its UI on providing trail information, but people can also use its data to plan activities for a vacation. The app doesn't need to build vacation-planning features because it implements these entity query protocols to provide an interface to the data through a Shortcut.

The sample app groups trails into collections based on geographic region, and implements the collections as a type called `TrailCollection` that conforms to `AppEntity`. The list of geographic regions is small, and a `TrailCollection` is a simple structure with the collection name and a list of trail IDs that require little memory. To make this information available through a

Find intent, the app implements `FeaturedCollectionEntityQuery` with conformance to `EnumerableEntityQuery`. The app uses `EnumerableEntityQuery` here because the data for the featured trail collections is a small and fixed set of values, and doesn't require a large amount of memory. The app implements `allEntities` to return all of the values, which people can filter by name in the Shortcuts app.

```
func allEntities() async throws -> [TrailCollection] {
    Logger.entityQueryLogging.debug("[FeaturedCollectionEntityQuery] Request for all")
    return trailManager.featuredTrailCollections
}
```

The app also implements `EntityPropertyQuery` for `TrailEntity`. This query type is ideal for large data sets that may have large numbers of entities, or entities that have higher memory consumption. Implementing this query adds a Find intent to the Shortcuts app, enabling people to run predicate searches on entity properties. For example, someone planning a vacation around seeing waterfalls that are easily accessible can configure the Find intent with criteria for trails containing *fall* in the trail name, and a trail distance of less than 1 kilometer. An implementation of `EntityPropertyQuery` includes several required functions and properties. `TrailEntityQuery+PropertyQuery.swift` contains the complete implementation.

Designing great intents for integration with the system means that the intents work as standalone intents with their parameters, and also work with other intents the app provides, or with other apps that may be installed. People can create shortcuts that use the output of one intent the app provides and use it as input to another intent the app provides, like the following examples:

- `SuggestTrails` can use the output of the Find intent for trail collections as input.
- The Find intent for trails can use the output of `SuggestTrails` to further refine the results.
- The Find intent for trails can also work alone, searching for matching trail properties from all of the trail data the app provides.

## Contribute entities to Spotlight

The sample app provides its trail data to Spotlight when the app first runs. The app declares a `Trail` structure for this data, containing the app's internal representation of that data. The app maps its data from the structure to searchable attributes in a `CSSearchableItemAttributeSet`.

```
var searchableAttributes: CSSearchableItemAttributeSet {
    let attributes = CSSearchableItemAttributeSet()

    attributes.title = name
```

```

attributes.namedLocation = regionDescription
attributes.keywords = activities.localizedElements

attributes.latitude = NSNumber(value: coordinate.latitude)
attributes.longitude = NSNumber(value: coordinate.longitude)
attributes.supportsNavigation = true

return attributes
}

```

The app also declares a `TrailEntity` structure to make the trail data available to the rest of the system as part of its App Intents integration. To integrate `TrailEntity` with Spotlight, `TrailEntity` conforms to `IndexedEntity`. The app associates the searchable attributes from the `Trail` structure with the `TrailEntity` by calling `associateAppEntity(_:priority:)` before contributing the data to the Spotlight index.

```

// Create an array of the searchable information for each `Trail`.
let searchableItems = trails.map { trail in
    let item = CSSearchableItem(uniqueIdentifier: String(trail.id),
                               domainIdentifier: nil,
                               attributeSet: trail.searchableAttributes)

    let isFavorite = favoritesCollection.members.contains(trail.id)
    let weight = isFavorite ? 10 : 1
    let intent = TrailEntity(trail: trail)

    /**
     Associate `TrailEntity` with the data that the `Trail` structure provides so that
     both types represent the same data. You need to create this association before
     adding the item to a `CSSearchableIndex`.
     */
    item.associateAppEntity(intent, priority: weight)
    return item
}

do {
    // Add the trails to the search index so people can find them through Spotlight.
    // You need to do this as part of the app's initial setup on launch.
    let index = CSSearchableIndex.default()
    try await index.indexSearchableItems(searchableItems)
    Logger.spotlightLogging.info("[Spotlight] Trails indexed by Spotlight")
} catch let error {
}

```

```
Logger.spotlightLogging.error("[Spotlight] Trails were not indexed by Spotlight.
```

1

## Integrate universal links


The sample app offers an `OpenTrail` intent so that people can open the app to a specific trail's information from a shortcut. Rather than adding code to configure the app's UI for displaying a trail's information just for this intent, the app uses the same URL scheme it uses to implement universal links. The app declares the URL for a trail's details through conformance to `URLRepresentableEntity`.


```
extension TrailEntity: URLRepresentableEntity {
    static var urlRepresentation: URLRepresentation {
        // Use string interpolation to fill values from your entity necessary for co
        // This example URL uses the unique and persistent identifier for the `Trail
        "https://example.com/trail/\(.id)/details"
    }
}
```


To leverage the app's existing code for handling a universal link, the app conforms the `OpenTrail` intent to both `OpenIntent` and `URLRepresentableIntent`. These conformances allow the app to skip implementing a `perform()` method on `OpenTrail`. When the intent runs, the system automatically passes the URL to the app using the standard mechanisms required for handling universal links.

## See Also

### Essentials

 **App Intents updates**  
Learn about important changes in App Intents.

 **Making actions and content discoverable and widely available**  
Adopt App Intents to make your app discoverable with Spotlight, controls, widgets, and the Action button.

 **Creating your first app intent**  
Create your first app intent that makes your app available in system experiences like Spotlight or the Shortcuts app.

## { } Adopting App Intents to support system experiences

Create app intents and entities to incorporate system experiences such as Spotlight, visual intelligence, and Shortcuts.