Article

Integrating custom data types into your intents

Provide the system with information about the types your app uses to model its data so that your intents can use those types as parameters.

Overview

Your app likely defines a number of custom types that model the data the app creates or consumes. For example, an app that enables people to view information about hiking trails might define types that describe trail information. Because those types are unique to your app, the framework can't interpret them until you expose them to system experiences like the Shortcuts app. *Entities* are lightweight types that provide information to the system about your app's data or concepts relating to that data. An entity identifies and queries the data it represents and describes how the system displays that data onscreen. Create an entity for each of the core types or concepts you want to use as an input parameter for your intents.

For more information on creating app intents and specifying their input parameters, see Creating your first app intent and Adding parameters to an app intent.

Define an entity that represents your data

To let an intent use one of your app's custom data types as a parameter, define a new structure in your app's target to represent that type. Then update the structure's definition to adopt the `AppEntity` protocol. Although custom types can directly conform to the protocol, prefer using distinct entity types that are lightweight and provide only the information the system requires. Distinct types let you separate your entities from the rest of your app's model and domain code. For example, the Accelerating app interactions with App Intents sample code project uses a `Trail` type and defines a corresponding `TrailEntity` type.

```
struct TrailEntity: AppEntity {
    // ...
}
```

If your app's model code is lightweight and the amount of data is small enough to fit into device memory, you can make your models conform to `AppEntity` to keep your code simple. However, models can contain larger data like images or your app could operate on large amounts of data. For those cases, use separate types for your entities so you can load only the data the entities need, and make the mechanism that provides the entities' data more performant. For example, if your app uses CloudKit to manage its data, set the `desiredKeys` property on your fetch operations to return only the data the entities need instead of fetching entire records.

Specify a unique identifier for your entity

Every entity must have a stable, unique identifier. The framework uses that identifier as a concrete reference to your entity while mediating between your app and other parts of the system. For example, when someone selects the value for an entity-based parameter in the shortcut editor, the system asks your app to resolve that parameter using the entity's identifier. The `AppEntity` protocol inherits the `Identifiable` protocol to enforce this prerequisite.

To add a unique identifier to your entity type, implement the protocol's `id` requirement and set its type to one of the three data types optimized for the framework: `String`, `Int`, or `UUID`.

```
var id: UUID
```

> **Note**
>
> Wherever possible, use one of the three optimized data types for entity identifiers. If you must use a different data type, extend that type and implement the required support. For more information, see `EntityIdentifierConvertible`.

Provide a visual representation for your entity

An entity represents a type and the data for that type. In your entity, describe how to display both elements onscreen. For example, the Shortcuts app uses this information to show type details in the Actions Library and to present entity data in the shortcut editor.

Add the `typeDisplayRepresentation` variable to your entity's structure and return a human-readable, localized string that describes the entity. For example, the hiking app from the previous

example displays the number of trails. The system displays this string whenever it needs to present your entity's type onscreen.

```
static var typeDisplayRepresentation: TypeDisplayRepresentation {
    TypeDisplayRepresentation(
        name: LocalizedStringResource("Trail", table: "AppIntents"),
        numericFormat: LocalizedStringResource("\(placeholder: .int) trails", table:
    )
}
```

The required `displayRepresentation` variable describes how to display an entity's represented data at runtime. Update your structure to include this variable and return an instance of `DisplayRepresentation`. Specify a localized `title` that lets people recognize the data.

Create a visually rich display of your entity by setting the representation's `subtitle` and `image` variables. For example, the Accelerating app interactions with App Intents sample code project displays the name of the trail, a region description, and an image.

```
var displayRepresentation: DisplayRepresentation {
    DisplayRepresentation(title: "\(name)",
                          subtitle: "\(regionDescription)",
                          image: DisplayRepresentation.Image(named: imageName))
}
```

For more information, see `DisplayRepresentation`.

Make your entity searchable

The framework requires entity types to be searchable so the system can resolve identifiers at runtime and request a list of suggested entities to display onscreen. For example, when a person sets a parameter to a specific entity in the shortcut editor, the system retains that entity's identifier. Later, when the intent runs, the framework asks your type to materialize the entity from its identifier. The framework then updates the relevant parameter with the materialized entity before invoking the intent's `perform()` function.

To make your entity searchable, define a new structure that adopts the `EntityQuery` protocol. Place this structure in the app's target alongside your entity. Add the `entities(for:)` function, and update the declaration so the element type of the `identifiers` array matches your entity's `id` variable. Use the provided identifiers to materialize and return the relevant entities.

```
struct TrailEntityQuery: EntityQuery {

    @Dependency
    var trailManager: TrailDataManager

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

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

To offer a better user experience, provide a list of suggested entities that the system displays, at appropriate times, to let people quickly make a selection. To provide those entities, add the `suggestedEntities()` method to your query structure. If your data generates a small number of entities, return them all; otherwise, return a subset of those entities relevant to the current context. For example, the Accelerating app interactions with App Intents sample code project suggests a person's favorite hiking trails.

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

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

To let people use arbitrary text to find specific entities, adopt the `EntityStringQuery` protocol instead. Queries that adopt this protocol cause the system to display a search field above the list of suggested entities. Implement the required `entities(matching:)` function, and use the provided string to match against your data. For example, the Accelerating app interactions with App Intents sample code project allows people to search for a specific trail. The following code snippet from the sample code project matches a person's search input to the app's trail information using the `name` property:

```
func entities(matching string: String) async throws -> [TrailEntity] {
    return trailManager.trails { trail in
        trail.name.localizedCaseInsensitiveContains(string)
    }.map { TrailEntity(trail: $0) }
}
```

After you implement your query, update the related entity's definition to include the `defaultQuery` variable, and specify an instance of your query type as the value. The system uses this variable at runtime to determine which type it can query on behalf of the related entity.

```
static var defaultQuery = TrailEntityQuery()
```

There are several subprotocols to `EntityQuery`, each of which enables different types of functionality. The Accelerating app interactions with App Intents sample code project implements all of them for demonstration purposes, but for a real app, you can choose the ones that meet your needs.

Enable Find intents

Apps implementing either the `EnumerableEntityQuery` or `EntityPropertyQuery` protocols 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 Accelerating app interactions with App Intents sample code project focuses its UI on providing trail information, but people could 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 an App Shortcut.

For more information about enabling Find intents, see Enable Find intents.

Enumerate your data type's static values

If a type has known fixed values at build time, such as a Swift enumeration, expose those types to the system by converting them to *app enums*, the static equivalent of entities. Because app enum values are constant, the compiler introspects them at build time and optimizes their use. The framework provides both an identity and a query by default, and the system can get type information at runtime without launching the app. For example, a music app might use an app enum to associate an album with an album type such as studio, live, or compilation.

To convert a common type to an app enum, update its declaration to adopt the `AppEnum` protocol. There's no need to create a separate type because the existing type is inherently lightweight and doesn't store additional data. The framework requires that app enums also conform to `RawRepresentable` and use `String` as their storage type, so modify your type to satisfy those requirements. Like with entities, specify a localized description of the type that the system can display onscreen.

```
enum ActivityStyle: String, Codable, Sendable {
    case biking
    case equestrian
    case hiking
    case jogging
    case crossCountrySkiing
    case snowshoeing

    // ...
}

extension ActivityStyle: AppEnum {

    static var typeDisplayRepresentation: TypeDisplayRepresentation {
        TypeDisplayRepresentation(
            name: LocalizedStringResource("Activity", table: "AppIntents"),
            numericFormat: LocalizedStringResource("\(placeholder: .int) activities'
        )
    }
```

Important

Don't adopt both protocols in the same type; use `AppEntity` for types that provide dynamic values and `AppEnum` for types that provide a limited set of static values.

To provide descriptions for each of your app enum's values, add the protocol's required `case DisplayRepresentations` variable. Return a dictionary that maps the values to their display representations.

```
static var caseDisplayRepresentations: [ActivityStyle: DisplayRepresentation] = [
    .biking: DisplayRepresentation(title: "Biking",
                                   subtitle: "Mountain bike ride",
                                   image: imageRepresentation[.biking]),

    .equestrian: DisplayRepresentation(title: "Equestrian",
                                       subtitle: "Equestrian sports",
                                       image: imageRepresentation[.equestrian]),

    .hiking: DisplayRepresentation(title: "Hiking",
                                   subtitle: "A lengthy outdoor walk",
                                   image: imageRepresentation[.hiking]),
```

```
    .jogging: DisplayRepresentation(title: "Jogging",
                                    subtitle: "A gentle run",
                                    image: imageRepresentation[.jogging]),

    .crossCountrySkiing: DisplayRepresentation(title: "Skiing",
                                               subtitle: "Cross-country skiing",
                                               image: imageRepresentation[.crossCour

    .snowshoeing: DisplayRepresentation(title: "Snowshoeing",
                                        subtitle: "Walking in the snow",
                                        image: imageRepresentation[.snowshoeing])
]
```

The example above initializes each representation with a string literal. To help people quickly understand the values, it also specifies a `subtitle` and an `image`.

For more information, see `DisplayRepresentation`.

See Also

Parameters, custom data types, and queries

- Adding parameters to an app intent

  Enable people to configure app intents with their custom input values.

- Parameter resolution

  Define the required parameters for your app intents and specify how to resolve those parameters at runtime.

- App entities

  Make core types or concepts discoverable to the system by declaring them as app entities.

- Entity queries

  Help the system find the entities your app defines and use them to resolve parameters.

- Resolvers

  Resolve the parameters of your app intents, and extend the standard resolution types to include your app's custom types.