Sample Code

Adopting drag and drop using SwiftUI

Enable drag-and-drop interactions in lists, tables and custom views.

Download

iOS 18.0+ | iPadOS 18.0+ | macOS 15.0+ | visionOS 2.0+ | Xcode 16.0+

Overview

This sample code project demonstrates how a SwiftUI view can act as a drag source or drop destination.

To enable drag interactions, add the `draggable(_:)` modifier to a view to send or receive `Transferable` items within an app, among a collection of your own apps, or between your apps and others that support the import or export of a specified data format. To handle dropped content, use the `dropDestination(for:action:isTargeted:)` modifier to receive the expected dropped item.

In the sample app, people can drag a contact from a list of contacts and drop it into another app; such as Contacts, Notes, or Messages. Additionally, people can drag and drop new contacts from other apps, like Contacts or Notes, into the sample app.

On iPad, people can use this sample with a second app such as Contacts or Notes in Split View. People can drag a row from this app into Contacts or drag a contact from Contacts into this app.

Enable Drag Interactions

To enable dragging, add the `draggable(_:)` modifier to a view to send items that conform to `Transferable` protocol.

```
List {
    ForEach(dataModel.contacts) { contact in
        NavigationLink {
            ContactDetailView(contact: contact)
        } label: {
            CompactContactView(contact: contact)
                .draggable(contact) {
                    ThumbnailView(contact: contact)
                }
        }
    }
}
```

When someone drags a contact from a list of contacts, the app uses the `draggable(_:preview:)` modifier to define a custom preview for the dragged item.

To learn more about adopting the draggable API to provide items for drag operations, see Making a view into a drag source.

Enable drop interactions

Use the `dropDestination(for:action:isTargeted:)` modifier to receive dragged items and define the destination that handles the dropped content.

```
.dropDestination(for: Contact.self) { droppedContacts, index in
    dataModel.handleDroppedContacts(droppedContacts: droppedContacts, index: index)
}
```

The modifier expects a type `Contact` which conforms to the `Transferable` protocol. The implementation of the `dropDestination(for:action:isTargeted:)` modifier uses the `transferRepresentation` to receive a dragged item representing the dropped contact information.

The app defines the transfer representations in order of preference. The app uses the most suitable representation to create and initialize a `Contact` object.

```
static var transferRepresentation: some TransferRepresentation {
    // Allows a Contact to be transferred with a custom content type.
    CodableRepresentation(contentType: .exampleContact)
    // Allows importing and exporting Contact data as a vCard.
    DataRepresentation(contentType: .vCard) { contact in
```

```
            try contact.toVCardData()
        } importing: { data in
            try await parseVCardData(data)
        }
        .suggestedFileName { $0.fullName }
        // Enables exporting the `phoneNumber` string as a proxy for the entire `Contact
        ProxyRepresentation { contact in
            contact.phoneNumber
        } importing: { value in
            Contact(id: UUID().uuidString, givenName: value, familyName: "", phoneNumber
        }
    }
```

If the app receives an item with a custom uniform type identifier, for example `com.example.contact`, it uses the `CodableRepresentation` to represent the `Contact` data structure.

`com.example.contact` declared by the sample app conforms to public.contact. To ensure the operating system understands how to handle the content type, it should conform to either `UTType.data`, `UTType.package`, or one of the types that inherit from these two.

`ProxyRepresentation` serves as an alternative representation that allows people to drag and drop a contact into any text editor that doesn't support `com.example.contact` or `vCard` content but works with text formats. In this case, the app exports the `phoneNumber` as a string. When someone drops a string on the `List` or `Table`, the sample app uses the `ProxyRepresentation` to convert that string into a `Contact` object.

Finally, `DataRepresentation` creates a binary representation of the `Contact` object and constructs the value for the receiver that supports the `vCard` content type.

When someone drops the contact on the `Table` or `List`, the completion handler inserts the dropped contact into the collection of contacts at the drop location. If the drop doesn't specify an index, the completion handler adds the dropped contact to the end of the collection.

```
func handleDroppedContacts(droppedContacts: [Contact], index: Int? = nil) {
    guard let firstContact = droppedContacts.first else {
        return
    }
    // If the ID of the first contact exists in the contacts list,
    // move the contact from its current position to the new index.
    // If no index is specified, insert the contact at the end of the list.
    if let existingIndex = contacts.firstIndex(where: { $0.id == firstContact.id })
        let indexSet = IndexSet(integer: existingIndex)
        contacts.move(fromOffsets: indexSet, toOffset: index ?? contacts.endIndex)
    } else {
        contacts.insert(firstContact, at: index ?? contacts.endIndex)
    }
}
```

Finally, the `dropDestination(for:action:isTargeted:)` modifier can receive drag interactions that start in any other view or app.

For design guidance on adopting drag and drop, see Human Interface Guidelines > Drag and drop.

See Also

Essentials

Making a view into a drag source

Adopt draggable API to provide items for drag-and-drop operations.