By default, lists do not have animations enabled for them. For that, you must configure this behavior.

Ways to perform animations

1. `.animation(_:value:)` — Animate whenever the value changes. Quick, global animations.

Attach an animation to a view that runs whenever the given value changes. Good for “always animate this view when its model changes” without having to wrap every mutation in withAnimation.

Example

```
@State private var items = ["A","B","C"]

var body: some View {
    List {
        ForEach(items, id: \.self) { item in
            Text(item)
        }
    }
    .animation(.default, value: items) // animate list changes whenever `items` changes
}
```

Pros

- Simple and declarative: one modifier controls animation for all changes to the watched value.
- Less boilerplate: you don’t have to wrap every mutation in withAnimation.
- Works for many common changes (insert/remove/reorder) when identity is stable.

Cons

- Coarse control: every change to the watched value uses the same animation; harder to customize per-action.
- Can produce unexpected animations if the watched value changes for reasons you didn’t intend to animate.
- The modifier applies at the View tree location you place it—placement matters for scope.

When to choose this

- Use `.animation(_:value:)` when you want a simple, consistent animation for updates to a specific state (e.g., whole list contents) and don't need per-action customization. Prefer this over the older, global `.animation(_:)` so you only animate on the exact value changes you expect.

2. `.transition(...)` — Custom enter/exit transitions for views

Define how a view appears/disappears (move, scale, opacity, or a combination). Transitions apply to view insertions/removals and must be used together with an animation context (e.g., withAnimation or .animation modifier) to be visible.

Example (use a ScrollView + LazyVStack for full control — List may ignore custom transitions)

```
@State private var items = ["A","B","C"]

var body: some View {
    ScrollView {
        LazyVStack {
            ForEach(items, id: \.self) { item in
                Text(item)
                    .padding()
                    .background(Color.secondary.opacity(0.1))
                    .cornerRadius(8)
                    .transition(.move(edge: .leading).combined(with:
.opacity))
            }
        }
    }
    Button("Remove first") {
        withAnimation {
            items.removeFirst()
        }
    }
}
```

Pros

- Fine-grained control over how individual rows appear/disappear (in/out animations).
- Can combine multiple transition effects and customize insertion vs removal animations.
- Great for custom visual behaviors (slide+fade, asymmetric transitions).

Cons

- Transitions normally don't work well inside List due to List being a platform-native

wrapper; you often need ScrollView + LazyVStack.
- Requires explicit animation context (e.g., withAnimation or .animation modifier) to animate.
- Needs stable identity (id) on rows to match before/after views for meaningful transitions.

When to choose this

- Use transitions when you want specific enter/exit animations for rows (e.g., slide in from left, fade out). If you need custom transitions, avoid List and use LazyVStack in a ScrollView for consistent behavior.

3. `withAnimation { ... }` — Explicitly animate the mutation

Wrap state changes in withAnimation to animate any view changes caused by that mutation. This gives explicit, per-action control and allows different animations for different actions.

Example

```
@State private var items = ["A","B","C"]

var body: some View {
    List {
        ForEach(items, id: \.self) { item in
            Text(item)
        }
        .onDelete { indexSet in
            withAnimation(.spring()) {
                items.remove(atOffsets: indexSet)
            }
        }
    }
}
```

Pros

- Explicit and local: you control exactly which mutations animate and which animation to use.
- Easy to give different animations for different user actions (e.g., spring for insert, linear for reorder).
- Works inside List for common actions like deletion/move when done on the data model.

Cons

- Requires you remember to wrap each mutation you want animated.

- Can be verbose if many different places mutate the same state.
- You still need stable identity for good results.

When to choose this

- Use withAnimation when you want explicit, action-level control (e.g., different animations for insert vs delete, or only animate certain user-driven operations). It's also the safest way to ensure a particular mutation is animated.

Extra tips and common pitfalls

- Identity matters: Mark items Identifiable or use id: so SwiftUI can match items before/after changes. Without stable ids, you'll get no meaningful animation or weird replacements.
- List limitations: List is backed by platform-native table views and may ignore custom transitions or some SwiftUI animation behaviors. For fully custom row animations, prefer ScrollView + LazyVStack.
- For coordinated moves between different containers or views, look at matchedGeometryEffect.

Summary guidance

- Want a quick, global animation for all list changes? Use .animation(:value:).
- Need custom enter/exit animations for rows? Use .transition(...) on the rows — with LazyVStack rather than List.
- Need explicit per-operation control (different animations for different actions)? Use withAnimation { ... }.

If you want, paste your actual List code and tell me whether you want simple deletions animated or custom transitions — I'll show the exact change.

How to animate changes to a Row's state changing

Use List Stable IDs for SwiftUI to determine when items are added, removed, or moved in a list. But a stable id does not include state changes within a row of a list. For example, let's say you have an image in each row that when a button is pressed, you want to animate the changing of that image.

Do not include this Image state in the List Stable IDs. For in-place visual changes you must animate the change itself:

- Wrap the property change in withAnimation { ... }, or
- Use .animation(:value:) on a view that depends on that property, or

- Use transitions/matchedGeometryEffect for more complex behaviors.

Example: toggling an image in a row — correct approach

- Model with stable id:

```
struct Item: Identifiable {
  let id = UUID()
  var title: String
  var isStarred: Bool
}
```

- Toggle with animation (row view or parent):

```
// Parent view
@State private var items: [Item] = [...]

List {
  ForEach($items) { $item in
    HStack {
      Text(item.title)
      Spacer()
      Button {
        withAnimation(.spring()) {
          item.isStarred.toggle()   // animate the state change
        }
      } label: {
        Image(systemName: item.isStarred ? "star.fill" : "star")
          .foregroundColor(item.isStarred ? .yellow : .gray)
          .scaleEffect(item.isStarred ? 1.2 : 1.0) // animate this
          .animation(.easeInOut, value: item.isStarred) // animate the image
change
      }
    }
  }
}
```

Notes:

- We did not change the Stable id when toggling isStarred. SwiftUI keeps the same row view and animates the image change.
- Using ForEach($items) gives you bindings into the array elements so you can mutate them in-place.

What about MVVM where the array gets updated async in ViewModel?

Hard to animate a list when it's updated in a `Task{}` inside of a ViewModel.

There are a few options you have when you can't simply wrap the list in `withAnimation{}`.

1. use `.animation(_:value:)`

```
// ViewModel
@MainActor
class VM: ObservableObject {
    @Published var items: [ListItem] = []
}

// View
struct ContentView: View {
    @StateObject var vm = VM()

    var body: some View {
        List {
            ForEach(vm.items) { item in
                Text(item.title)
            }
        }
        .animation(.default, value: vm.items) // animate when vm.items
changes
    }
}
```

2. Wrap the Published update in withAnimation in the ViewModel

```
@MainActor
class VM: ObservableObject {
    @Published var items: [ListItem] = []

    private var cancellables = Set<AnyCancellable>()

    init(dbPublisher: AnyPublisher<[DBModel], Never>) {
        dbPublisher
            .map { $0.map { ListItem(id: $0.id, title: $0.title) } } //
preserve DB id
            .sink { [weak self] newItems in
                // Option A: animate assignments from the VM side
```

```
                withAnimation(.spring()) {
                    self?.items = newItems
                }
            }
            .store(in: &cancellables)
    }
}
```

3. Keep local copy in the View and animate changes there

```
@State private var localItems: [ListItem] = []

.onChange(of: viewModel.publishedList) { new in
    withAnimation {
        localItems = new.map { ListItem(id: $0.id, title: $0.title) } //
keep id stable
    }
}
```