# SWIFTUI PERFORMANCE CHECKLIST

```
Improved Practice: Ternary Operator
 Bad Practice: Excessive Branching
                                                           -> keep the same view identify but change its
 -> every time the condition changes SwiftUI will
    recreate new views -> poor performance
                                                           properties
                                                           -> less calls to view body property
 -> loss of state
                                                           -> animations work between different states
struct TreatView: View {
                                                           struct TreatView: View {
   var treat: Treat
                                                             var treat: Treat
   var body: some View {
                                                             var body: some View {
     // Branching based on treat.isExpired
                                                                Text(treat.isExpired ? "Expired Treat" :
     if treat.isExpired {
                                                                                          "Fresh Treat")
        return Text("Expired Treat")
                                                                }
     } else {
                                                             }
        return Text("Fresh Treat")
                                                           }
     }
  }
}
                                                           Improved Practice: Inert Modifers
Bad Practice: Excessive Branching
                                                           -> keep the same view identify but change its
                                                              properties
struct ExpirationView: View {
                                                           -> inert modifiers have cases that don't change
  var date: Date
                                                              the view appearance
  var body: some View {
     // Branching based on date
                                                           struct ExpirationView: View {
     if date < .now {
                                                             var date: Date
        Text("Expired Treat")
                                                             var body: some View {
    } else {
                                                                Text("Treat Expired")
        EmptyView()
                                                                   .opacity(t date < .now ? 1 : 0)
     }
                                                                  // Inert modifier instead of branching
  }
                                                             }
}
                                                           }
Bad Practice: Unnecessary AnyView Usage
                                                           Improved Practice: @ViewBuilder
-> loss of structural identity
                                                           -> return multiple views from a closure with
-> SwiftUI can not efficiently update UI and will
                                                              @ViewBuilder
   do more redraws
                                                           struct ContentView: View {
struct ContentView: View {
                                                             @State private var showText = true
   @State private var showText = true
                                                             var body: some View {
   var body: some View {
                                                                . . . .
                                                              }
   }
                                                            @ViewBuilder
   var body: some View {
                                                             var body: some View {
     if showText {
                                                               if showText {
      return AnyView(Text("Hello, SwiftUI"))
                                                                 return Text("Hello, SwiftUI")
    } else {
                                                               } else {
      return AnyView(Image(systemName: "swift"))
                                                                 return Image(systemName: "swift")
     }
                                                               }
  }
                                                             }
                                                           }
}
```

## **VIEW IDENTITY**

Your data is so important that SwiftUI has a set of data-driven constructs that use the identity of your data as a

- Foreach
  confirmationDialog() / alert()
- List, Table / OutlineGroup

<pre>Bad Practice: Dynamic Identifiers -&gt; change identity of views struct Pet: Identifiable {    var name: String    var id: UUID { UUID() } // dynamic identifier } ForEach(pets) { pet in    PetView(pet: pet) }</pre>	<pre>Improved Practice: Stable Identifiers -&gt; never change during run time struct Pet: Identifiable {    var name: String    let id = UUID() // Stable identifier } ForEach(pets) { pet in    PetView(pet: pet) }</pre>
<pre>Bad Practice: Non-Unique Identifiers -&gt; multiple views with the same identity struct Pet: Identifiable {    var name: String    var id: String { name }     // uniqueness might not be guaranteed if     multiple pets with same name are used } ForEach(pets) { pet in    PetView(pet: pet) }</pre>	<pre>Improved Practice: Unique Identifiers -&gt; all views can be uniquely identified struct Pet: Identifiable {     var name: String     let id = UUID() // unique identifier } ForEach(pets) { pet in     PetView(pet: pet) }</pre>
<pre>Bad Practice: Dynamic and Non-Unique Identifiers -&gt; multiple views with the same identity @State var colors = [Color.red, Color.green,</pre>	<pre>Improved Practice: Unique Identifiers - &gt; all views can be uniquely identified struct ColorData: Identifiable {    var color: Color    let id = UUID() // stable and unique identifier } @State var colorData = [ColorData(color: .red,         ColorData(color: .green)] ForEach(colorData) {    ColorPicker("Color",         selection: \$0.color) }</pre>

© 2024 <u>swiftyplace.com</u>

#### **OPTIMIZE LIST AND TABLE PERFORMANCE**

Bad Practice: Inefficient identifier generation and variable view counts. Impact: Slow list and table updates. Improvement: Use constant view counts per data element. Avoid using AnyView and conditional views inside ForEach.



## MINIMIZE UNNECESSARY VIEW UPDATES

SwiftUI updates views based on changes to their dependencies. To minimize unnecessary updates, carefully consider the dependencies of your views and ensure that only the necessary dependencies are included. Here are different types of view dependencies:

<u>Here are different types of view dependencies:</u> - @State,@Binding, @StateObject, @EnvironmentObject

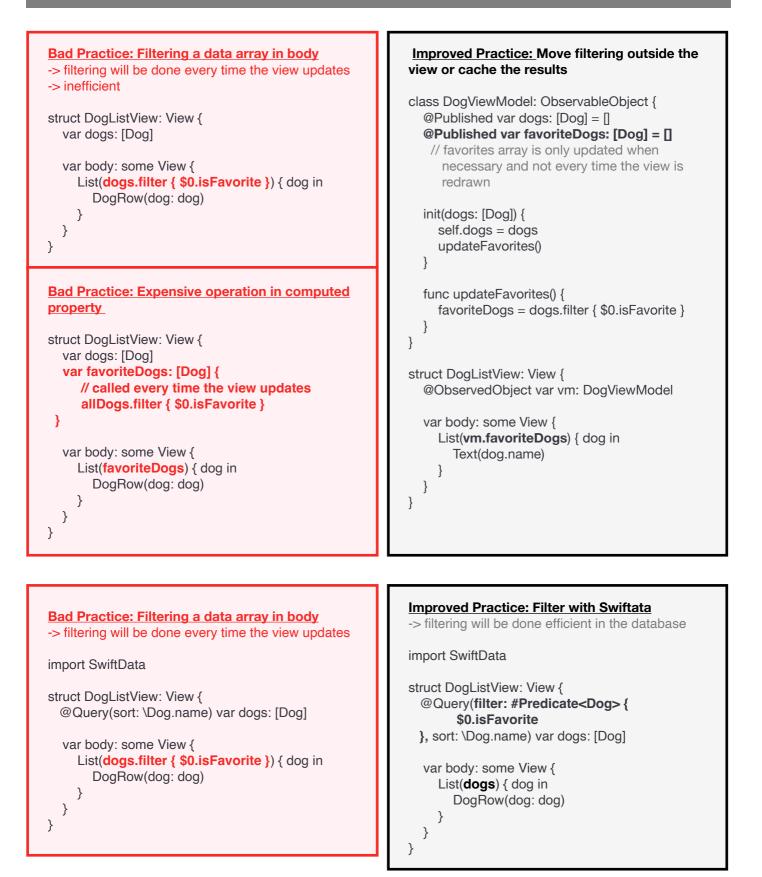
- view properties

<pre>Bad Practice: Unnecessary Dependency -&gt; view depends on a large data structure but only uses a small part of it struct Dog {     var name: String     let imageName: String } struct DogView: View {     let dog: Dog     // view updates when dog changes     var body: some View {         Image(dog.imageName))     } }</pre>	<pre>Improved Practice: Reduce Dependencies -&gt; reduces the view's dependencies, leading to fewer updates struct Dog {    var name: String    let imageName: String } struct DogView: View {    let dogImageName: String    // the only dependency that is used in this view    var body: some View {     Image(dogImageName))    } }</pre>
	<pre>Improved Practice: New Observation Feature -&gt; reduces the view's updates efficiently @Observable class Dog {     var name: String     let imageName: String } struct DogView: View {     let dog: Dog // only updates when property         imagName changes     var body: some View {         Image(dog.imageName))     } }</pre>
<pre>Bad Practice: Unnecessary Dependency -&gt; view depends on a large data set struct ContentView: View {     @EnvironmentObject var vm: ViewModel     // view updates when any property in view     model changes     var body: some View {         Text("Main Content")      } }</pre>	<pre>Improved Practice: Remove Dependencies -&gt; one use view models in subviews that need it struct ContentView: View { // remove dependency to view model var body: some View {  } } struct RemoveDogView: View { @EnvironmentObject var vm: ViewModel var body: some View { Button("Delete") { vm.delete() } } }</pre>

### AVOID UNNECESSARY RECOMPUTATIONS IN BODY

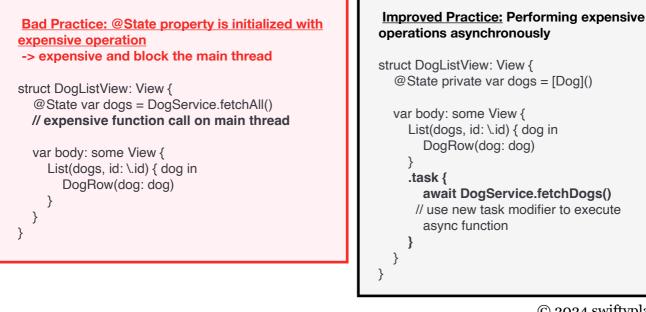
The body should be as lightweight as possible because SwiftUI may call it frequently during the lifecycle of your view. Expensive operations within body can lead to performance issues like slow rendering and unresponsive user interfaces.

**Impact**: Slow updates and poor app responsiveness. **Improvement**: Move expensive operations out of body. Use asynchronous data fetching and cache results.



#### **EXPENSIVE DYNAMIC PROPERTY INSTANTIATION**

Dynamic properties in SwiftUI, like @State or @EnvironmentObject, are powerful tools for managing app state. However, if not used carefully, they can lead to expensive updates.



© 2024 swiftyplace.com

This is a summary of Best Practices from WWDC Talks

- WWDC21 <u>Demystify SwiftUI</u>
- WWDC23 Demystify SwiftUI Performance