SWIFTUI PERFORMANCE CHECKLIST

Bad Practice: Excessive Branching
-> every time the condition changes SwiftUI will recreate new views -> poor performance
-> loss of state

```
struct TreatView: View {
    var treat: Treat
    var body: some View {
        // Branching based on treat.isExpired
        if treat.isExpired {
            return Text("Expired Treat")
        } else {
            return Text("Fresh Treat")
        }
    }
}
```

Improved Practice: Ternary Operator
-> keep the same view identify but change its properties
-> less calls to view body property
-> animations work between different states

```
struct TreatView: View {
    var treat: Treat
    var body: some View {
        Text(treat.isExpired ? "Expired Treat" :
                               "Fresh Treat")
    }
}
```

Bad Practice: Excessive Branching

```
struct ExpirationView: View {
    var date: Date
    var body: some View {
        // Branching based on date
        if date < .now {
            Text("Expired Treat")
        } else {
            EmptyView()
        }
    }
}
```

Improved Practice: Inert Modifers
-> keep the same view identify but change its properties
-> inert modifiers have cases that don't change the view appearance

```
struct ExpirationView: View {
    var date: Date
    var body: some View {
        Text("Treat Expired")
            .opacity(t date < .now ? 1 : 0)
            // Inert modifier instead of branching
    }
}
```

Bad Practice: Unnecessary AnyView Usage
-> loss of structural identity
-> SwiftUI can not efficiently update UI and will do more redraws

```
struct ContentView: View {
    @State private var showText = true

    var body: some View {
        ....
    }

    var body: some View {
        if showText {
            return AnyView(Text("Hello, SwiftUI"))
        } else {
            return AnyView(Image(systemName: "swift"))
        }
    }
}
```

Improved Practice: @ViewBuilder
-> return multiple views from a closure with @ViewBuilder

```
struct ContentView: View {
    @State private var showText = true

    var body: some View {
        ....
    }

    @ViewBuilder
    var body: some View {
        if showText {
            return Text("Hello, SwiftUI")
        } else {
            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 form of explicit identity for your views.

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

Bad Practice: Dynamic Identifiers
-> change identity of views

```
struct Pet: Identifiable {
    var name: String
    var id: UUID { UUID() }   // dynamic identifier
}

ForEach(pets) { pet in
    PetView(pet: pet)
}
```

Improved Practice: Stable Identifiers
-> never change during run time

```
struct Pet: Identifiable {
    var name: String
    let id = UUID()  // Stable identifier
}

ForEach(pets) { pet in
    PetView(pet: pet)
}
```

Bad Practice: Non-Unique Identifiers
-> 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)
}
```

Improved Practice: Unique Identifiers
-> all views can be uniquely identified

```
struct Pet: Identifiable {
    var name: String
    let id = UUID()  // unique identifier
}

ForEach(pets) { pet in
    PetView(pet: pet)
}
```

Bad Practice: Dynamic and Non-Unique Identifiers
-> multiple views with the same identity

```
@State var colors = [Color.red, Color.green,
                     Color.blue]
 // dynamically can change the colors which are
    used as identifiers

ForEach(colors,  id: \.self) {
     // might have multiple views with
        same color identifier
   ColorPicker("Color", selection: $0)
}
```

Improved Practice: Unique Identifiers -
> 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)
}
```

OPTIMIZE LIST AND TABLE PERFORMANCE

B**ad 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.

Bad Practice: Using Conditional Views Inside ForEach
-> SwiftUI has to evaluate each condition to determine the number of rows

```
ForEach(dogs) { dog in
    if dog.isFavorite {
        DogRow(dog: dog)
    }
}
```

Improved Practice: Filter outside the ForEach
-> List now has a constant number of views per element, improving performance.

```
ForEach(viewModel.favoriteDogs) { dog in
    DogRow(dog: dog)
}
```

Bad Practice: Complex Identifier Generation
Using complex logic to generate identifiers for list rows can slow down your app

```
List(dogs, id: \.self) { dog in
    // If the Dog struct is complex, using it directly
     as an identifier can be expensive.
    DogRow(dog: dog)
}
```

Improved Practice: Simple Identifiers
Use simple, unique properties for identifiers, such as an id field, to speed up the list's performance.

```
List(dogs, id: \.id) { dog in
    // Identifiers are now cheap to generate, leading
       to faster load and update times.
    DogRow(dog: dog)
}
```

Bad Practice: Nested ForEach
List has to retrieve identifiers for all nested ForEach and the total number of rows

```
List {
    ForEach(toyCategories) { category in
        Text(category.name).bold()
        ForEach(category.dogs) { dog in
            DogRow(dog: dog)
        }
    }
}
```

Improved Practice: Use Dynamic Sections
Use Nested ForEach together with Section
-> SwiftUI optimizes dynamic sections,

```
List {
    ForEach(toyCategories) { category in
        Section(header: Text(category.name)) {
            ForEach(category.dogs) { dog in
                DogRow(dog: dog)
            }
        }
    }
}
```

Bad Practice: AnyView in List/ForEach
-> cannot determine the number of rows from the view structural hierarchy
-> SwiftUI creates all views to retrieve row identifiers

```
ForEach(dogs) { dog in
    AnyView(DogRow(dog: dog))
    // SwiftUI does not know how many rows
        are shown for each dog
}
```

Improved Practice: Avoid Type-Erasing Views
Keep the view types explicit within ForEach to allow SwiftUI to optimize view updates.

```
List {
    ForEach(dogs) { dog in
        DogRow(dog: dog)
    }
}
```

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:
- @State,@Binding, @StateObject, @EnvironmentObject
- view properties

Bad Practice: Unnecessary Dependency
-> 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))
    }
}
```

Improved Practice: Reduce Dependencies
-> 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))
    }
}
```

Improved Practice: New Observation Feature
-> 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))
    }
}
```

Bad Practice: Unnecessary Dependency
-> 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")
      …
    }
}
```

Improved Practice: Remove Dependencies
-> 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()
      }
    }
}
```

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.

Bad Practice: Filtering a data array in body
-> filtering will be done every time the view updates
-> inefficient

```
struct DogListView: View {
    var dogs: [Dog]

    var body: some View {
        List(dogs.filter { $0.isFavorite }) { dog in
            DogRow(dog: dog)
        }
    }
}
```

Bad Practice: Expensive operation in computed property

```
struct DogListView: View {
    var dogs: [Dog]
    var favoriteDogs: [Dog] {
        // called every time the view updates
        allDogs.filter { $0.isFavorite }
    }

    var body: some View {
        List(favoriteDogs) { dog in
            DogRow(dog: dog)
        }
    }
}
```

Improved Practice: Move filtering outside the view or cache the results

```
class DogViewModel: ObservableObject {
    @Published var dogs: [Dog] = []
    @Published var favoriteDogs: [Dog] = []
      // favorites array is only updated when
         necessary and not every time the view is
         redrawn

    init(dogs: [Dog]) {
        self.dogs = dogs
        updateFavorites()
    }

    func updateFavorites() {
        favoriteDogs = dogs.filter { $0.isFavorite }
    }
}

struct DogListView: View {
    @ObservedObject var vm: DogViewModel

    var body: some View {
        List(vm.favoriteDogs) { dog in
            Text(dog.name)
        }
    }
}
```

Bad Practice: Filtering a data array in body
-> filtering will be done every time the view updates

```
import SwiftData

struct DogListView: View {
   @Query(sort: \Dog.name) var dogs: [Dog]

    var body: some View {
        List(dogs.filter { $0.isFavorite }) { dog in
            DogRow(dog: dog)
        }
    }
}
```

Improved Practice: Filter with Swiftata
-> filtering will be done efficient in the database

```
import SwiftData

struct DogListView: View {
   @Query(filter: #Predicate<Dog> {
            $0.isFavorite
   }, sort: \Dog.name) var dogs: [Dog]

    var body: some View {
        List(dogs) { dog in
            DogRow(dog: dog)
        }
    }
}
```

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.

Bad Practice: @State property is initialized with expensive operation
-> expensive and block the main thread

```
struct DogListView: View {
    @State var dogs = DogService.fetchAll()
    // expensive function call on main thread

    var body: some View {
        List(dogs, id: \.id) { dog in
            DogRow(dog: dog)
        }
    }
}
```

Improved Practice: Performing expensive operations asynchronously

```
struct DogListView: View {
    @State private var dogs = [Dog]()

    var body: some View {
        List(dogs, id: \.id) { dog in
            DogRow(dog: dog)
        }
        .task {
            await DogService.fetchDogs()
          // use new task modifier to execute
            async function
        }
    }
}
```


This is a summary of Best Practices from WWDC Talks

- WWDC21 Demystify SwiftUI
- WWDC23 Demystify SwiftUI Performance