SwiftUI Navigation & URL Routing

SwiftUI Navigation & URL Routing

Shayan Ali

Senior iOS developer at Comgy GmbH

GitHub: https://github.com/shayan18
Email: syed.shayan18@gmail.com

What is navigation?

What is navigation?

A change of mode in the application.

What is a “change of mode”?

What is a “change of mode”?

It’s when a piece of state goes from ***not existing to existing****, or the opposite,* ***existing to not existing****.*

Navigation APIs

Sheets

```
func sheet<Item, Content>(
    item: Binding<Item?>,
      content: (Item) ->
          Content
    ) -> some View
```

Sheets

```
func sheet<Item, Content>(
      item: Binding<Item?>,
        content: (Item) ->
              Content
        ) -> some View
```

Sheets

```
func sheet<Item, Content>(
    item: Binding<Item?>,
      content: (Item) ->
          Content
    ) -> some View
```

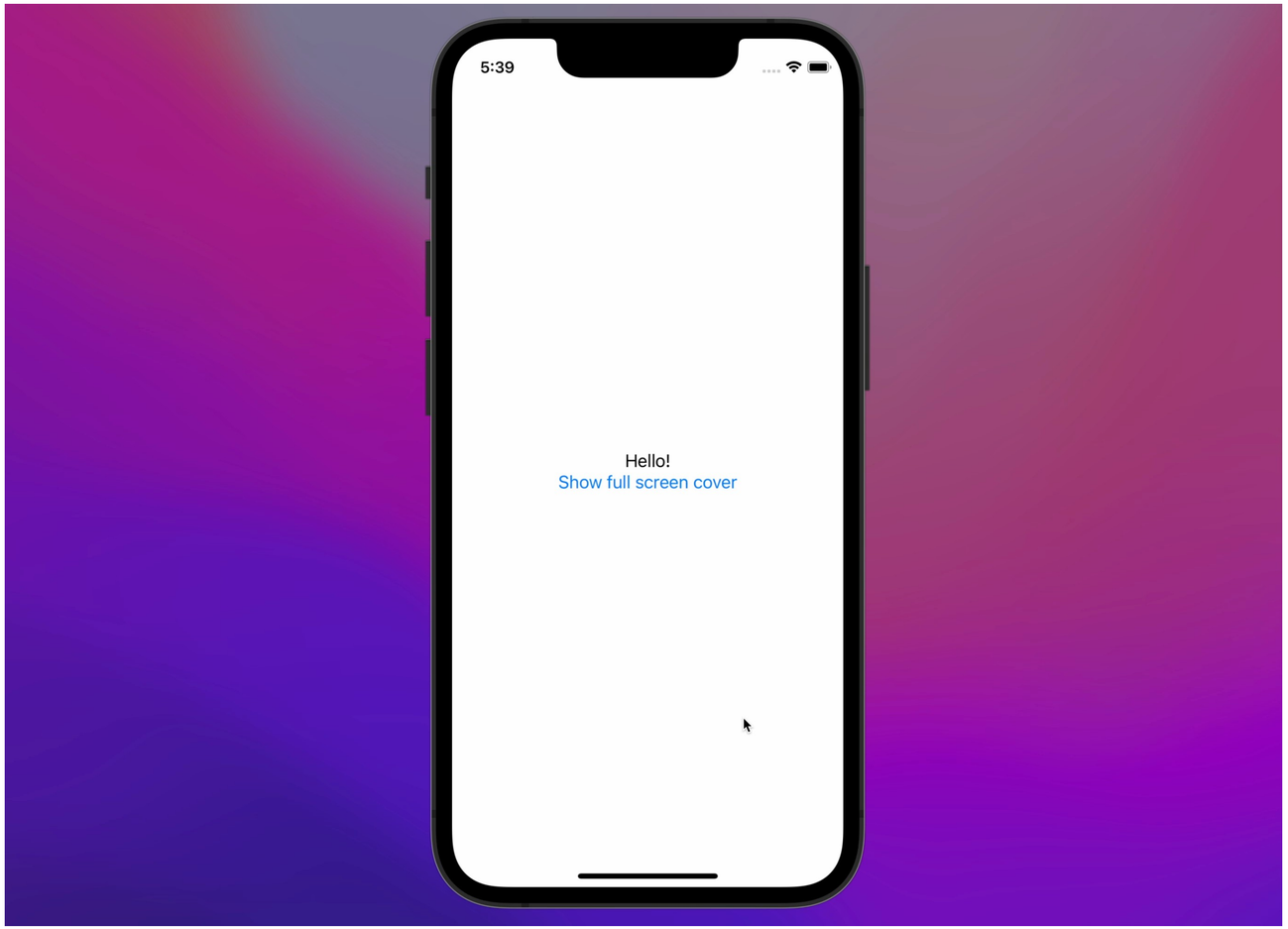
Sheets

```
func sheet<Item, Content>(
    item: Binding<Item?>,
      content: (Item) ->
          Content
    ) -> some View
```

Drill down navigation

Sheets

Full screen covers



Sheets

```
struct ProductView: View {

  @State var products: [Product]
  @State var editProduct: Product?

  var body: some View {
    List {

      ForEach(self.products) { product in

        Button("Edit") { self.editProduct = product }
      }
    }
    .sheet(item: $editProduct) { product in
      EditProductView(product: product)
    }
  }
}
```

Sheets

```
struct ProductView: View {

  @State var products: [Product]
  @State var editProduct: Product?

  var body: some View {
    List {

      ForEach(self.products) { product in

        Button("Edit") { self.editProduct = product }
      }
    }
    .sheet(item: $editProduct) { product in
      EditProductView(product: product)
    }
  }
}
```

Sheets

```
struct ProductView: View {

  @State var products: [Product]
  @State var editProduct: Product?

  var body: some View {
    List {

      ForEach(self.products) { product in

       Button("Edit") { self.editProduct = product }
      }
    }
    .sheet(item: $editProduct) { product in
      EditProductView(product: product)
    }
  }
}
```

Sheets

```
struct ProductView: View {

  @State var products: [Product]
  @State var editProduct: Product?

  var body: some View {
    List {

      ForEach(self.products) { product in

       Button("Edit") { self.editProduct = product }
      }
    }
    .sheet(item: $editProduct) { product in
      EditProductView(product: product)
    }
  }
}
```

```
func fullScreenCover<Item,
    Content>( item: Binding<Item?
    >,
   content: (Item) -> Content
) -> some View
```

```
func popover<Item, Content>(item:
    Binding<Item?>, content: (Item)
    -> Content
) -> some View
```

```
func sheet<Item, Content>(  item:
      Binding<Item?>,  content:
     (Item) -> Content) -> some
                  View
```

```
func bottomMenu<Item, Content>(
        item: Binding<Item?>,
    content: (Item) -> Content)
            -> some View
```

```
func sheet<Item,
    Content>(  isPresented:
    Binding<Item?>,  content:
    @escaping () -> Content
) -> some View

 func fullScreenCover<Item, Content>(
       isPresented: Binding<Item?>,
       content: @escaping (Item) ->
                  Content
            ) -> some View

func popover<Item,
    Content>(  isPresented:
    Binding<Item?>,  content:
    @escaping (Item) -> Content
) -> some View
```

Deep-linking Out of the Box

Step 1

Define the model

```
class Model: ObservableObject
   {   @Published var sheet: SheetModel?
}

class SheetModel: ObservableObject
   { @Published var popoverValue: Int?
}
```

Step 1

Define the model

```
class Model: ObservableObject
  {  @Published var sheet: SheetModel?
}

class SheetModel: ObservableObject
  { @Published var popoverValue: Int?
}
```

Step 1

Define the model

```
class Model: ObservableObject
  {  @Published var sheet: SheetModel?
}

class SheetModel: ObservableObject
  { @Published var popoverValue: Int?
}
```

Step 2

Define the views

```
struct ContentView: View {
 @ObservedObject var model: Model

 var body: some View {
  Button("Show sheet") { self.model.sheet = SheetModel() }
   .sheet(item: self.$model.sheet) { sheetModel in
    SheetView(model: sheetModel)
```

Step 2

Define the views

```
struct ContentView: View {
 @ObservedObject var model: Model

 var body: some View {
  Button("Show sheet") { self.model.sheet = SheetModel() }
    .sheet(item: self.$model.sheet) { sheetModel in
     SheetView(model: sheetModel)
```

Step 2

Define the views

```
struct ContentView: View {
 @ObservedObject var model: Model

 var body: some View {
  Button("Show sheet") { self.model.sheet = SheetModel() }
   .sheet(item: self.$model.sheet) { sheetModel in
    SheetView(model: sheetModel)
```

Step 2

Define the views

```
struct ContentView: View {
 @ObservedObject var model: Model

 var body: some View {
  Button("Show sheet") { self.model.sheet = SheetModel() }
    .sheet(item: self.$model.sheet) { sheetModel in
     SheetView(model: sheetModel)
```

Step 2

Define the views

```
struct SheetView: View {
  @ObservedObject var model: SheetModel

 var body: some View {  Button("Show
   popover") {
    self.model.popoverValue = .random(in: 1...1_000)
  }
  .popover(item: self.$model.popoverValue) { value in

                PopoverView(count: value)
     }
   }
  }
```

Step 3

Construct state

```
ContentView(model:
   Model(sheet:SheetModel(popoverValue: 50))
```

Deep linking Demo

Navigation links

Problems with current Navigation APIs?

```
NavigationLink(
 destination: () -> Destination,
 label: () -> Label)
```

Fire & Forget

```
NavigationLink(
 isActive: Binding<Bool>,
 destination: () -> Destination,
 label: () -> Label)
```

Manual work of binding Bool

```
NavigationLink( tag: Hashable,
 selection: Binding<Hashable?>,
 destination: () -> Destination,
 label: () -> Label)
```

Limited use cases

Problems with current Navigation APIs?

```
NavigationLink(
   tag:Hashable,
  selection: Binding<Hashable?>,
  destination: () -> Destination,
  label: () -> Label
)
```

Problems with current Navigation APIs?

```
NavigationLink(
   tag:Hashable,
  selection: Binding<Hashable?>,
  destination: () -> Destination,
  label: () -> Label
)
```

Problems with current Navigation APIs?

```
NavigationLink(
  tag:Hashable,
  selection: Binding<Hashable?>,
  destination: () -> Destination,
  label: () -> Label
)
```

Problems with current Navigation APIs?

```
NavigationLink(
  tag:Hashable,
  selection: Binding<Hashable?>,
  destination: () -> Destination,
  label: () -> Label
)
```

```
struct ContentView: View {
 @State var users: [User]
 @State var editingUserID: User.ID?

 var body: some View { List {
   ForEach(self.users) { user in
    NavigationLink(
     tag: user.id,
     selection: self.$editingUserID
    ) {
     EditUserView(userID: user.id)
    } label: {  Text("Edit user")
    }}}}}
```

```
struct ContentView: View {
 @State var users: [User]
 @State var editingUserID: User.ID?

 var body: some View { List {
   ForEach(self.users) { user in
    NavigationLink(
     tag: user.id,
     selection: self.$editingUserID
    ) {
     EditUserView(userID: user.id)
    } label: {  Text("Edit user")
    }}}}}
```

```
struct ContentView: View {
 @State var users: [User]
 @State var editingUserID: User.ID?

 var body: some View { List {
   ForEach(self.users) { user in
     NavigationLink(
      tag: user.id,
      selection: self.$editingUserID
     ) {
      EditUserView(userID: user.id)
     } label: {  Text("Edit user")
     }}}}}
```

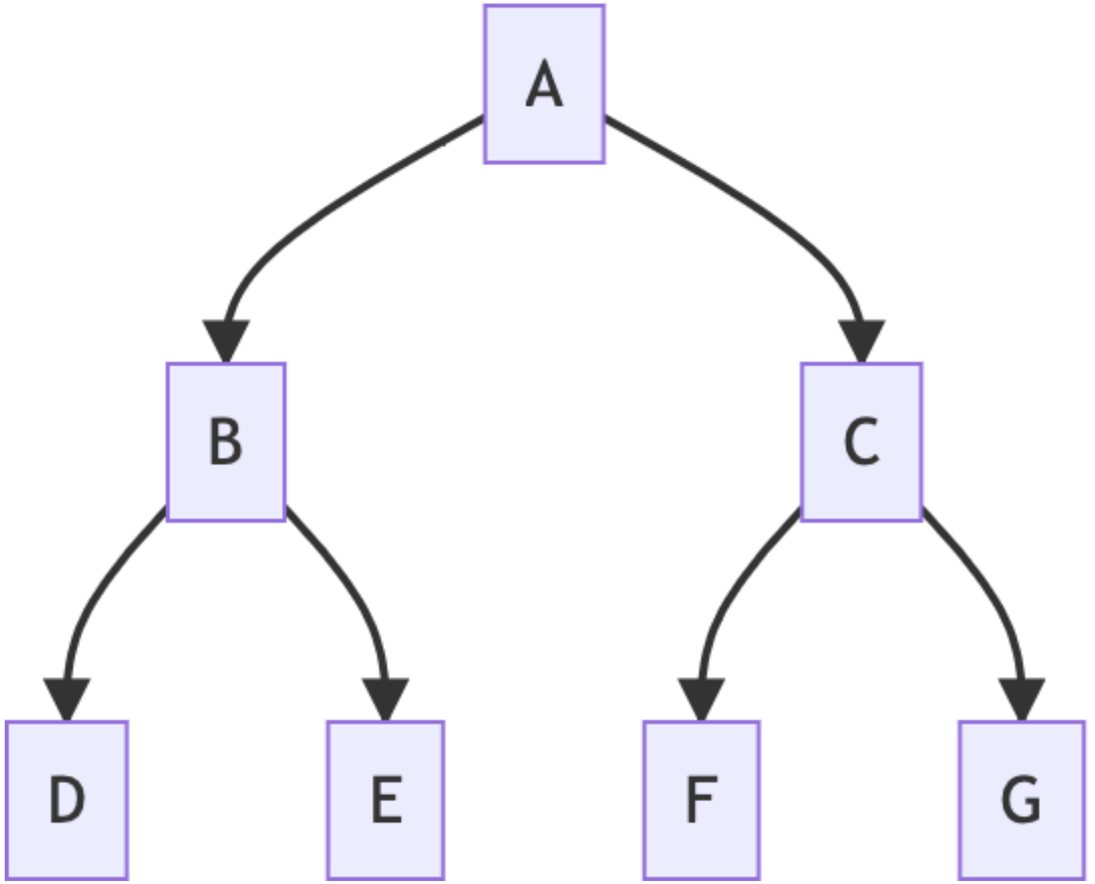
```
struct ContentView: View {
 @State var users: [User]
 @State var editingUserID: User.ID?

 var body: some View { List {
   ForEach(self.users) { user in
     NavigationLink(
      tag: user.id,
      selection: self.$editingUserID
     ) {
      EditUserView(userID: user.id)
     } label: {  Text("Edit user")
     }}}}}
```

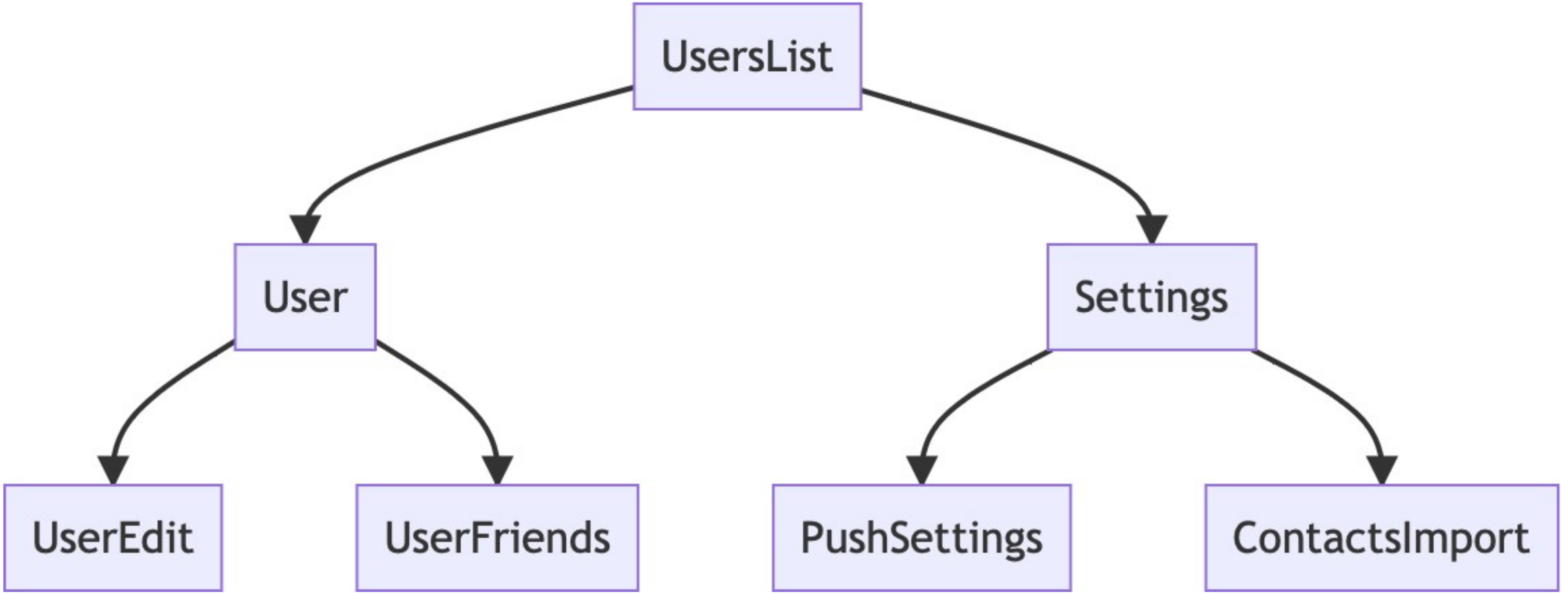
Demo with Problems

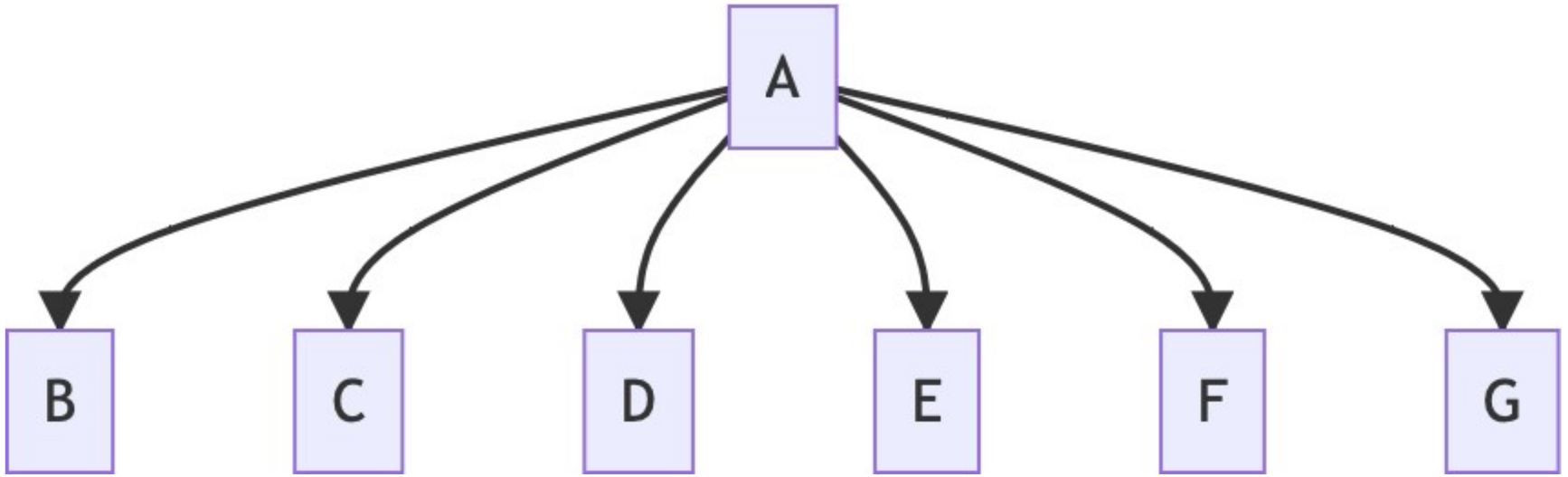
```
NavigationLink(
 destination: () -> Destination,
 label: () -> Label)

func sheet<Content>(  item:
   Binding<Item?>,
  content: @escaping (Item) -> Content) -> some
  View

func
   fullScreenCover<Content>( it
   em: Binding<Item?>,
  content: @escaping (Item) -> Content) -> some
  View
```

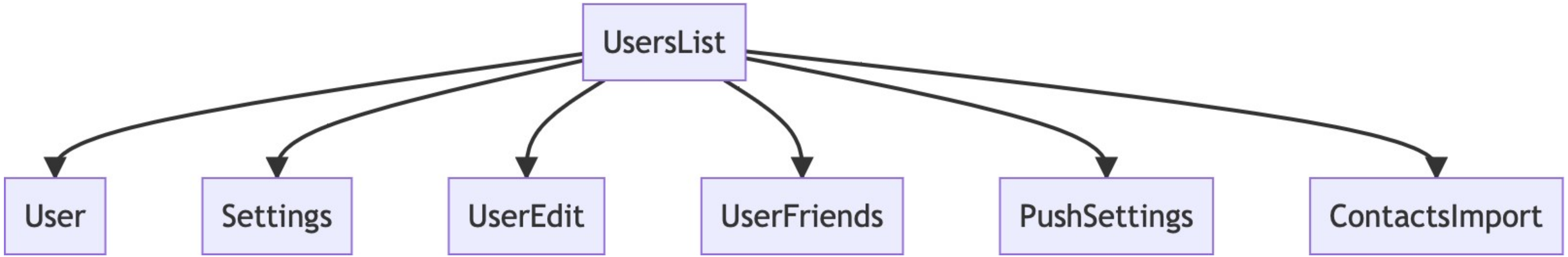
```
NavigationLink(
 destination: () -> Destination,
 label: () -> Label)

func sheet<Content>(  item:
   Binding<Item?>,
  content: @escaping (Item) -> Content) -> some View

func
   fullScreenCover<Content>( item
   : Binding<Item?>,
  content: @escaping (Item) -> Content) -> some View
```

Destination coupling


A
B
C
D
E
F
G


UsersList
User
Settings
UserEdit
UserFriends
PushSettings
ContactsImport


A
B
C
D
E
F
G


UsersList
User
Settings
UserEdit
UserFriends
PushSettings
ContactsImport

Navigation stacks

Decoupling navigation: Data

```
NavigationLink( value: Hashable?,
        label: () -> Label)
```

```
NavigationLink(value: userId) {
        Text(“Edit user”)
                }
```

Decoupling navigation: Data

```
NavigationLink( value: Hashable?,
        label: () -> Label)
```

```
NavigationLink(value: userId) {
        Text(“Edit user”)
                }
```

Decoupling navigation: Data

```
NavigationLink( value: Hashable?,
        label: () -> Label)
```

```
NavigationLink(value: userId) {
        Text(“Edit user”)
                }
```

Decoupling navigation: Interpretation

```
func navigationDestination<Data: Hashable>(
  for: Data.Type,
 destination: (Data) -> Destination
) -> some View
```

```
.navigationDestination(for: User.ID.self) { userID in
   EditUserView(id: userID)
}
```

Decoupling navigation: Interpretation

```
func navigationDestination<Data: Hashable>(
    for: Data.Type,
  destination: (Data) -> Destination
) -> some View


 .navigationDestination(for: User.ID.self) { userID in
     EditUserView(id: userID)
 }
```

Decoupling navigation: Interpretation

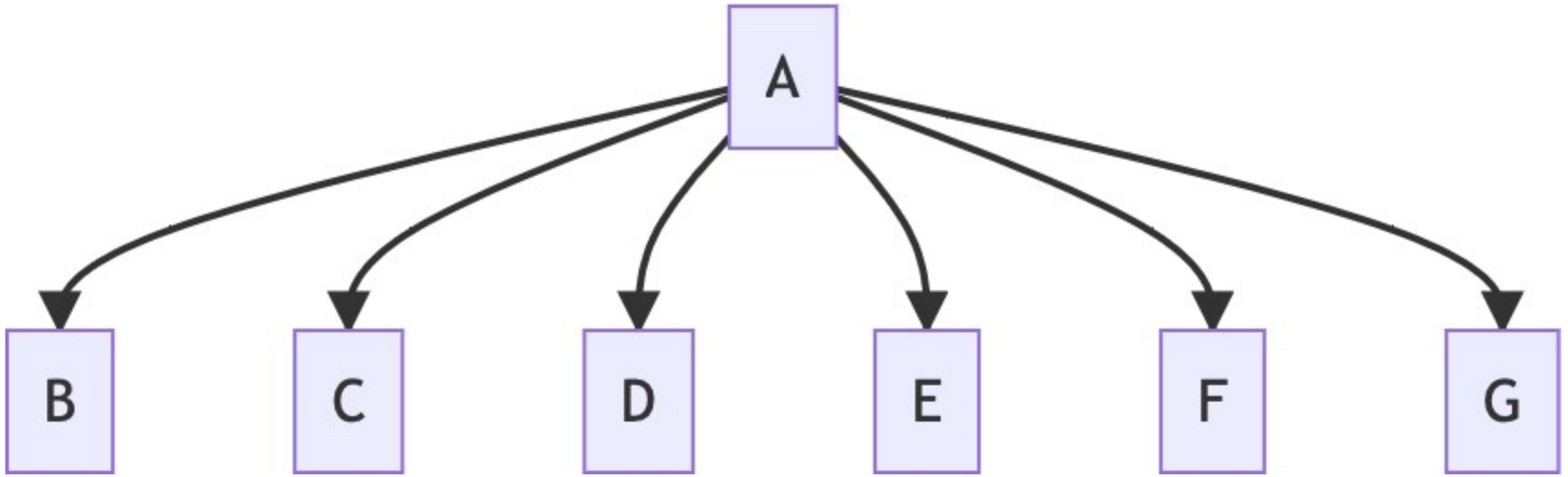
```
func navigationDestination<Data: Hashable>(
  for: Data.Type,
  destination: (Data) -> Destination
) -> some View
```

```
.navigationDestination(for: User.ID.self) { userID in
   EditUserView(id: userID)
}
```

Decoupling navigation: Interpretation

```
func navigationDestination<Data: Hashable>(
  for: Data.Type,
 destination: (Data) -> Destination
) -> some View
```

```
.navigationDestination(for: User.ID.self) { userID in
   EditUserView(id: userID)
}
```


A
B
C
D
E
F
G

Fire and forget

```
NavigationLink("Edit user", value: user.id)

.navigationDestination(for: User.ID.self { userID
  in EditUserView(id: userID)
}
```

State-driven NavigationStack

```
NavigationStack(
 path: Binding<Data>,
 root: () -> Root
)
where
 Data: MutableCollection
    & RandomAccessCollection
     & RangeReplaceableCollection,
 Data.Element: Hashable
```

```
NavigationStack(
path: Binding<NavigationPath>,

Root: () Root
)
```

```
NavigationStack(
  path: Binding<[Element]>,
  root: () -> Root
)
where Element: Hashable
```

```
NavigationStack(
  path: Binding<NavigationPath>,
  root: () -> Root
  )
```

```
NavigationStack(
  path: Binding<[Element]>,
  root: () -> Root

)
where Element: Hashable
```

```
NavigationStack(
  path: Binding<NavigationPath>,
  root: () -> Root
  )
```

Pros and cons of the two initializers

Binding<[Element]>

Pros

Strongly typed elements

Full access to collection API

Instant testability

Codable state restoration

Binding<[Element]>

Cons

Some light coupling with `NavigationLink`

Single point of handling destinations

Binding<NavigationPath>

Pros

Extremely easy to get started with

Maximum decoupling

Binding<NavigationPath>

Cons

Not testable

Not inspectable

Codability has runtime crashes

Which initializer to use?

Demo

URL routing

URL routing

```
/screenA -> ScreenA
/screenB -> ScreenB
/screenC -> ScreenC
```

```
/screenC/sheet    -> ScreenC w/ sheet open
/screenC/sheet/42 -> ScreenC w/ sheet and popover
open
```

```
/screenA/screenB/screenC/sheet/
                  42
```

URL routing

```
/screenA -> ScreenA

/screenB -> ScreenB

/screenC -> ScreenC


/screenC/sheet    -> ScreenC with sheet open

/screenC/sheet/42 -> ScreenC with sheet and popover
open

/screenA/screenB/screenC/sheet/
                  42
```

URL routing

/screenA -> ScreenA

/screenB -> ScreenB

/screenC -> ScreenC

/screenC/sheet -> ScreenC with sheet open

/screenC/sheet/42 -> ScreenC with sheet and popover open

/screenA/screenB/screenC/sheet/42

URL routing

```
/screenA -> ScreenA
/screenB -> ScreenB
/screenC -> ScreenC


/screenC/sheet   -> ScreenC with sheet open
/screenC/sheet/42 -> ScreenC with sheet and popover
open
```

/screenA/screenB/screenC/sheet/42

github.com/pointfreeco/
swift-url-routing

```
enum Destination {
  // /screenA
  case screenA
  // /screenB
  case screenB
  // /screenC/:sheet
  case screenC(destination: ScreenCDestination? =
  nil)
}

// /sheet/:int?

enum ScreenCDestination {
   case sheet(popoverValue: Int? Nil)
}
```

import URLRouting

```
// /sheet/:int?
struct ScreenCRouter: Parser {
 var body: some Parser<URLRequestData, ScreenCDestination>
   { Parse(ScreenCDestination.sheet(popoverValue:)) {
    Path {
     "sheet"
     Optionally { Int.parser() }
    }
    }}}
```

import URLRouting

```
// /sheet/:int?
struct ScreenCRouter: Parser {
 var body: some Parser<URLRequestData, ScreenCDestination>
   { Parse(ScreenCDestination.sheet(popoverValue:)) {
    Path {
     "sheet"
     Optionally { Int.parser() }
    }
    }}}
```

import URLRouting

```
// /sheet/:int?
struct ScreenCRouter: Parser {
  var body: some Parser<URLRequestData, ScreenCDestination>{
  Parse(ScreenCDestination.sheet(popoverValue:)) {

    Path {
      "sheet"
      Optionally { Int.parser() }
    }
    }}}
```

URLRouting

```
// /sheet/:int?
struct ScreenCRouter: Parser {
  var body: some Parser<URLRequestData, ScreenCDestination>{
  Parse(ScreenCDestination.sheet(popoverValue:)) {

     Path {
       "sheet"
       Optionally { Int.parser() }
     }
     }}}
```

URLRouting

```
// /sheet/:int?
struct ScreenCRouter: Parser {
  var body: some Parser<URLRequestData, ScreenCDestination>{
  Parse(ScreenCDestination.sheet(popoverValue:)) {

    Path {
      "sheet"
      Optionally { Int.parser() }
    }
    }}}
```

```
import URLRouting

struct DestinationRouter: Parser {
 var body: some Parser<URLRequestData, Destination> {
 OneOf {
   // screenA
   Parse(Destination.screenA) {
    Path { "screenA" }
   }
   // screenB
   Parse(Destination.screenB) {
    Path { "screenB" }
   }
   // screenC/:sheet
   Parse(Destination.screenC(destination:)) {
    Path { "screenC" }
    Optionally { ScreenCRouter() }
    }
    }
```

```
import URLRouting

struct DestinationRouter: Parser {
 var body: some Parser<URLRequestData, Destination> {
 OneOf {
   // screenA
   Parse(Destination.screenA) {
    Path { "screenA" }
   }
   // screenB
   Parse(Destination.screenB) {
    Path { "screenB" }
   }
   // screenC/:sheet
   Parse(Destination.screenC(destination:)) {
    Path { "screenC" }
    Optionally { ScreenCRouter() }
    }
    }
```

```
import URLRouting

struct DestinationRouter: Parser {
 var body: some Parser<URLRequestData, Destination> {
 OneOf {
   // screenA
   Parse(Destination.screenA) {
    Path { "screenA" }
   }
   // screenB
   Parse(Destination.screenB) {
    Path { "screenB" }
   }
   // screenC/:sheet
   Parse(Destination.screenC(destination:)) {
    Path { "screenC" }
    Optionally { ScreenCRouter() }
    }
    }
```

```
import URLRouting

struct DestinationRouter: Parser {
 var body: some Parser<URLRequestData, Destination> {
 OneOf {
   // screenA
   Parse(Destination.screenA) {
    Path { "screenA" }
   }
   // screenB
   Parse(Destination.screenB) {
    Path { "screenB" }
   }
   // screenC/:sheet
   Parse(Destination.screenC(destination:)) {
    Path { "screenC" }
    Optionally { ScreenCRouter() }
    }
    }
```

```
import URLRouting

struct DestinationRouter: Parser {
 var body: some Parser<URLRequestData, Destination> {
 OneOf {
   // screenA
   Parse(Destination.screenA) {
    Path { "screenA" }
   }
   // screenB
   Parse(Destination.screenB) {
    Path { "screenB" }
   }
   // screenC/:sheet
   Parse(Destination.screenC(destination:)) {
    Path { "screenC" }
    Optionally { ScreenCRouter() }
    }
    }
```

```
import URLRouting

struct DestinationRouter: Parser {
 var body: some Parser<URLRequestData, Destination> {
 OneOf {
   // screenA
   Parse(Destination.screenA) {
    Path { "screenA" }
   }
   // screenB
   Parse(Destination.screenB) {
    Path { "screenB" }
   }
   // screenC/:sheet
   Parse(Destination.screenC(destination:)) {
    Path { "screenC" }
    Optionally { ScreenCRouter() }
    }
    }
```

Question?

Thanks ^_^