SwiftUI Navigation & URL Routing

- Brandon Williams
  - brandon@pointfree.co
  - @mbrandonw

SwiftUI Navigation & URL Routing

- Brandon Williams
  - brandon@pointfree.co
  - @mbrandonw
- Stephen Celis
  - stephen@pointfree.co
  - @stephencelis

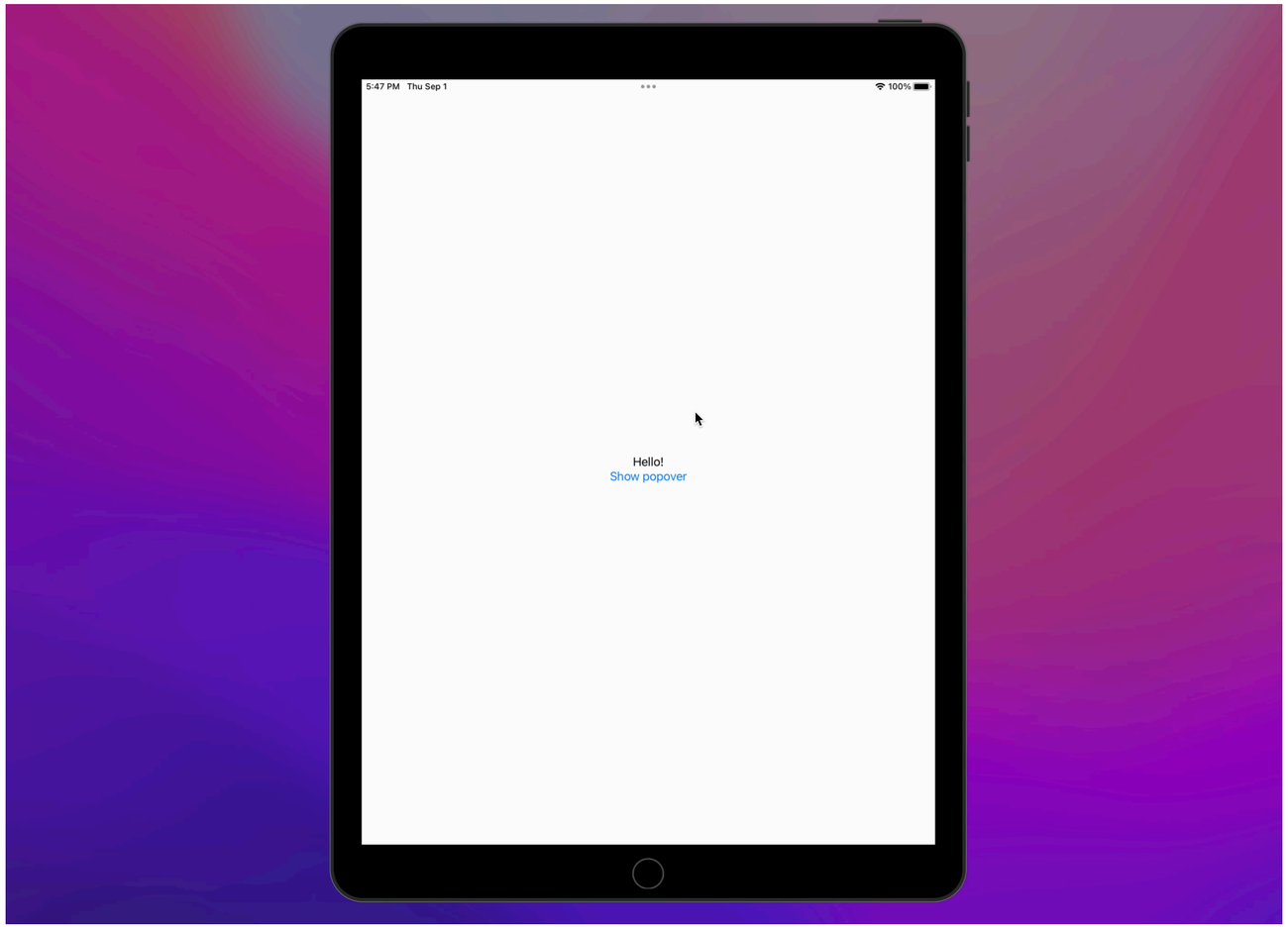
What is navigation?

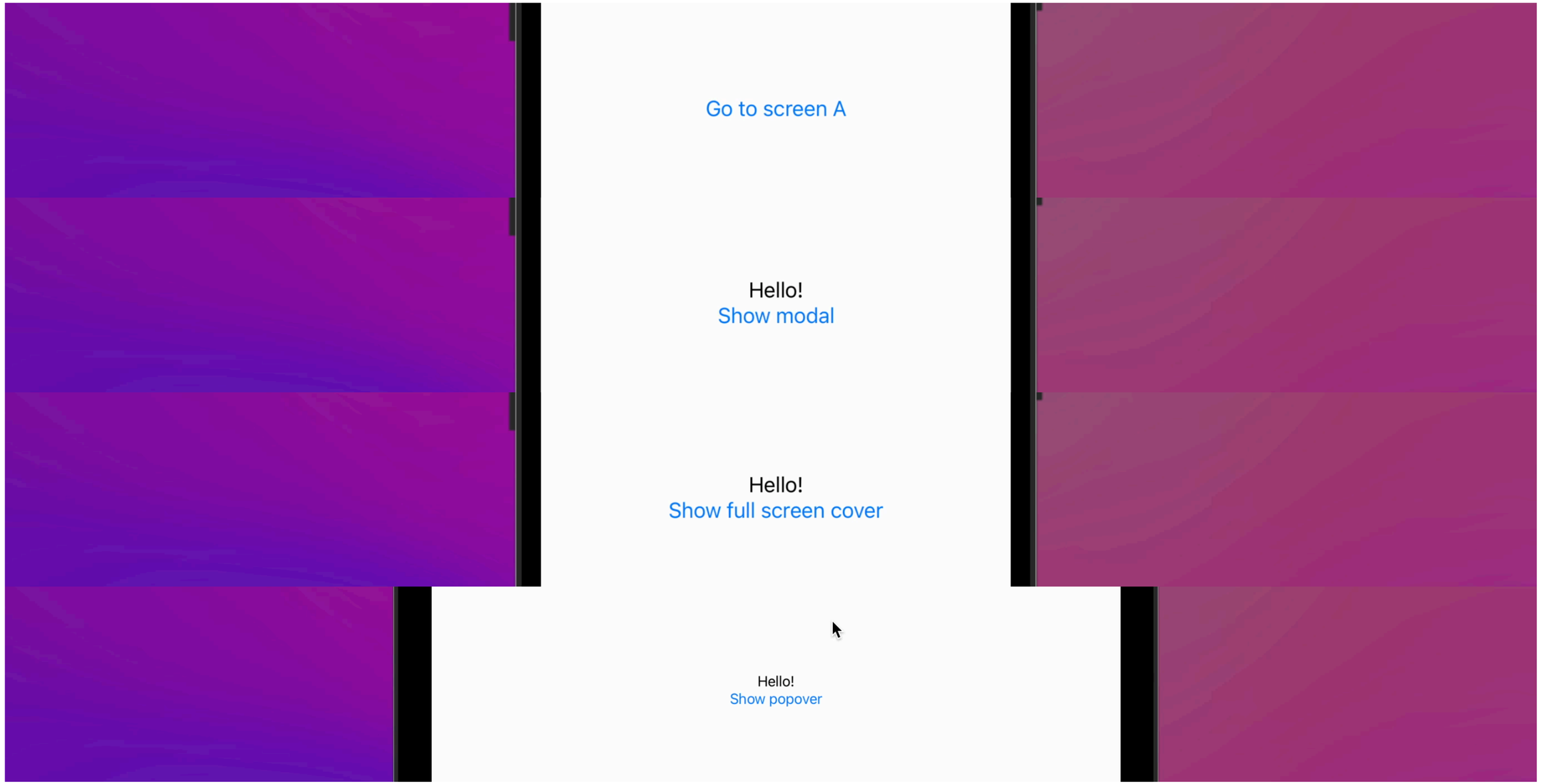
Drill down navigation

Sheets

Full screen covers

Popovers




Go to screen A
Hello!
Show modal
Hello!
Show full screen cover
Hello!
Show popover

What is navigation?

A change of mode in the application.

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
```

Sheets

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

Sheets

```
struct UsersView: View {
  @State var users: [User]
  @State var editUser: User?

  var body: some View {
    List {
      ForEach(self.users) { user in
        Button("Edit") { self.editUser = user }
      }
    }
    .sheet(item: $editUser) { user in
      EditUserView(user: user)
    }
  }
}
```

Sheets

```
struct UsersView: View {
  @State var users: [User]
  @State var editUser: User?

  var body: some View {
    List {
      ForEach(self.users) { user in
        Button("Edit") { self.editUser = user }
      }
    }
    .sheet(item: $editUser) { user in
      EditUserView(user: user)
    }
  }
}
```

Sheets

```
struct UsersView: View {
  @State var users: [User]
  @State var editUser: User?

  var body: some View {
    List {
      ForEach(self.users) { user in
        Button("Edit") { self.editUser = user }
      }
    }
    .sheet(item: $editUser) { user in
      EditUserView(user: user)
    }
  }
}
```

Sheets

```
struct UsersView: View {
  @State var users: [User]
  @State var editUser: User?

  var body: some View {
    List {
      ForEach(self.users) { user in
        Button("Edit") { self.editUser = user }
      }
    }
    .sheet(item: $editUser) { user in
      EditUserView(user: user)
    }
  }
}
```

Sheets

```
Button("Edit") {
  Task {
    let freshUser = try await self.apiClient.fetchUser(user.id)
    self.editUser = freshUser
  }
}
```

```
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

as easy as 1-2-3

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 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 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 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 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 2

Define the views

```
struct PopoverView: View {
  @State var count: Int
  var body: some View {
    HStack {
      Button("-") { self.count -= 1 }
      Text("\(self.count)")
      Button("+") { self.count += 1 }
    }
  }
}
```

Step 2

Define the views

```
struct PopoverView: View {
  @State var count: Int
  var body: some View {
    HStack {
      Button("-") { self.count -= 1 }
      Text("\(self.count)")
      Button("+") { self.count += 1 }
    }
  }
}
```

Step 2

Define the views

```
struct PopoverView: View {
  @State var count: Int
  var body: some View {
    HStack {
      Button("-") { self.count -= 1 }
      Text("\(self.count)")
      Button("+") { self.count += 1 }
    }
  }
}
```

Step 3

Construct state

```
ContentView(
  model: Model(
    sheet: SheetModel(
      popoverValue: 42
    )
  )
)
```

Step 3

Construct state

```
ContentView(
  model: Model(
    sheet: SheetModel(
      popoverValue: 42
    )
  )
)
```

Step 3

Construct state

```
ContentView(
  model: Model(
    sheet: SheetModel(
      popoverValue: 42
    )
  )
)
```

Deep linking

Demo

Sheets, covers, & popovers

Navigation links

Theoretical NavigationLink

```
NavigationLink(
  item: Binding<Item?>,
  destination: (Item) -> Destination,
  label: () -> Label
)
```

Theoretical NavigationLink

```
NavigationLink(
    item: Binding<Item?>,
    destination: (Item) -> Destination,
    label: () -> Label
)
```

Theoretical NavigationLink

```
NavigationLink(
  item: Binding<Item?>,
  destination: (Item) -> Destination,
  label: () -> Label
)
```

Theoretical NavigationLink

```
NavigationLink(
  item: Binding<Item?>,
  action: () -> Void,
  destination: (Item) -> Destination,
  label: () -> Label
)
```

Theoretical NavigationLink

```
NavigationLink(item: self.$model.editUser) {
  self.model.editUser = user
} destination: { user in
  EditUserView(user: user)
} label: {
  Text("Edit user")
}
```

Theoretical NavigationLink

```
NavigationLink(item: self.$model.editUser) {
  self.model.editUser = user
} destination: { user in
  EditUserView(user: user)
} label: {
  Text("Edit user")
}
```

Theoretical NavigationLink

```
NavigationLink(item: self.$model.editUser) {
  self.model.editUser = user
} destination: { user in
  EditUserView(user: user)
} label: {
  Text("Edit user")
}
```

Theoretical NavigationLink

```
NavigationLink(item: self.$model.editUser) {
  self.model.editUser = user
} destination: { user in
  EditUserView(user: user)
} label: {
  Text("Edit user")
}
```

Navigation odd ducks 🦆

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

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

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

Navigation odd ducks 🦆

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

Navigation odd ducks 🦆

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

Navigation odd ducks 🦆

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

Navigation odd ducks 🦆

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

Navigation odd ducks 🦆

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

Navigation odd ducks 🦆

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

Navigation odd ducks 🦆

```
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")
        }
      }
    }
  }
}
```

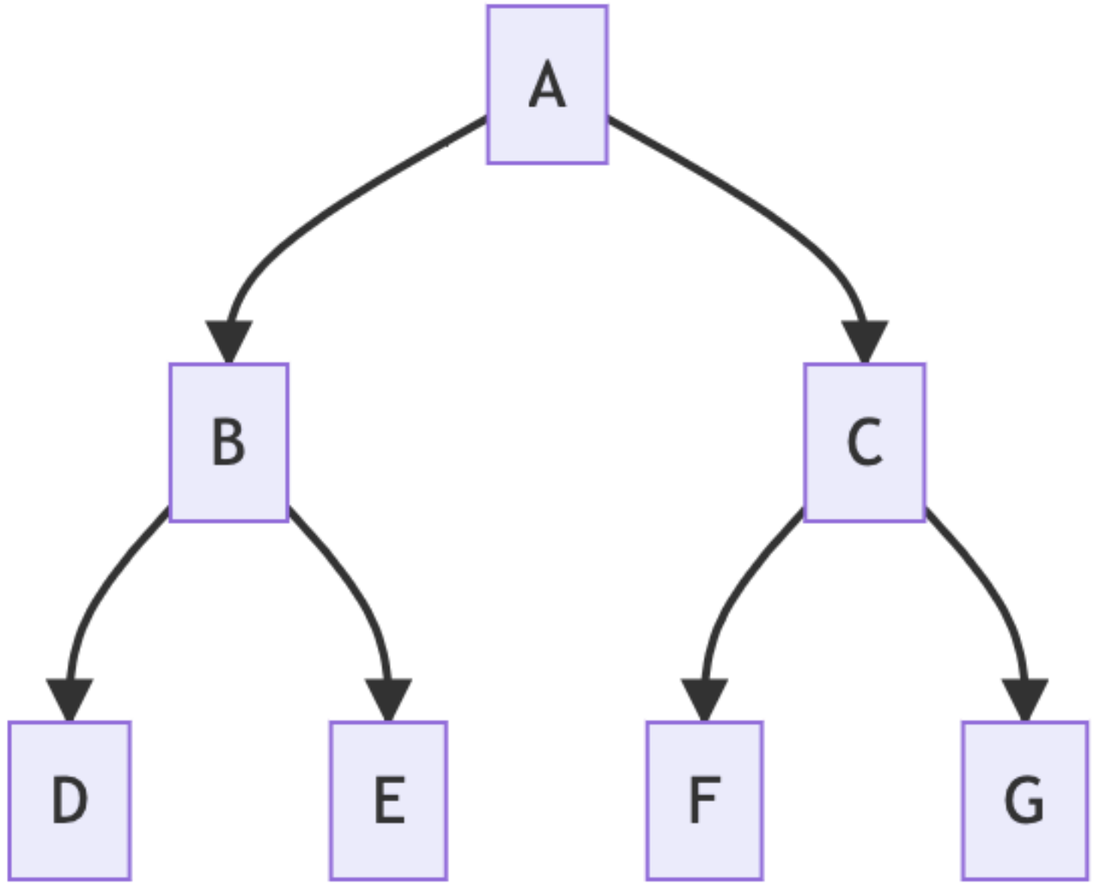
Demo

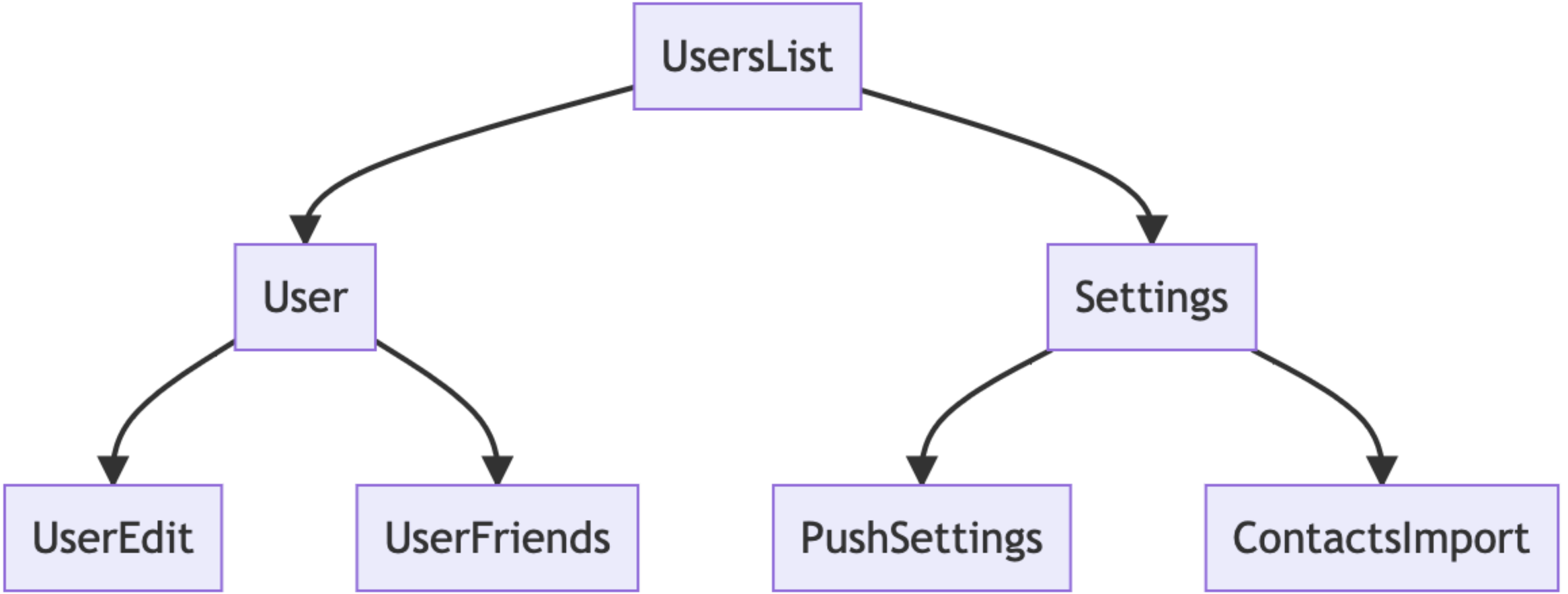
Navigation problems

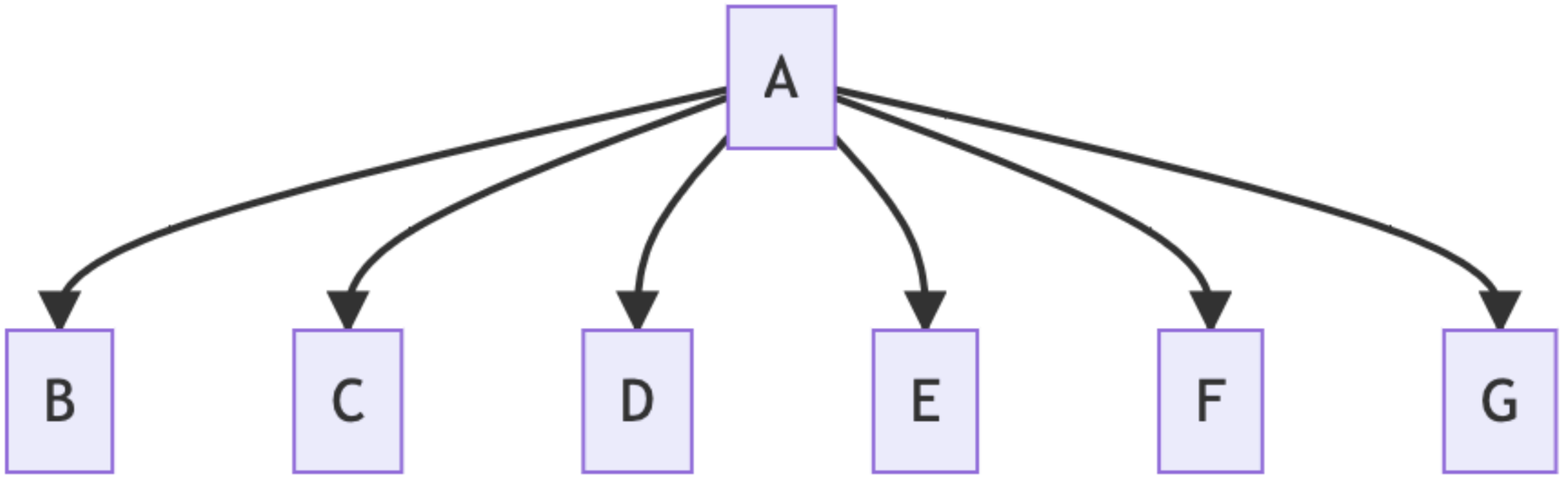
Destination coupling

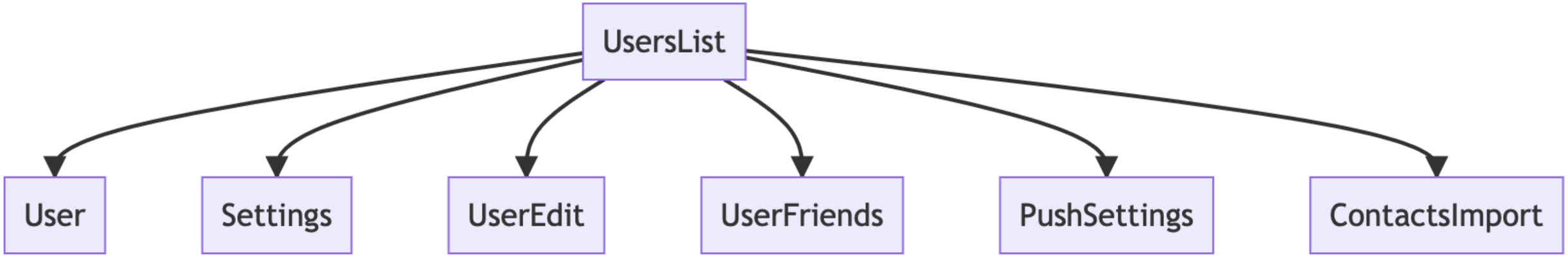
```
NavigationLink(
  destination: () -> Destination,
  label: () -> Label)

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

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

func popover<Content>(
    item: 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

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


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

- `NavigationStack`
- `NavigationLink`
- `.navigationDestination`

Decoupling navigation: Data

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

```
NavigationLink(value: user.id) {
  Text("Edit user")
}
```

Decoupling navigation: Data

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

```
NavigationLink(value: user.id) {
  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)
}
```

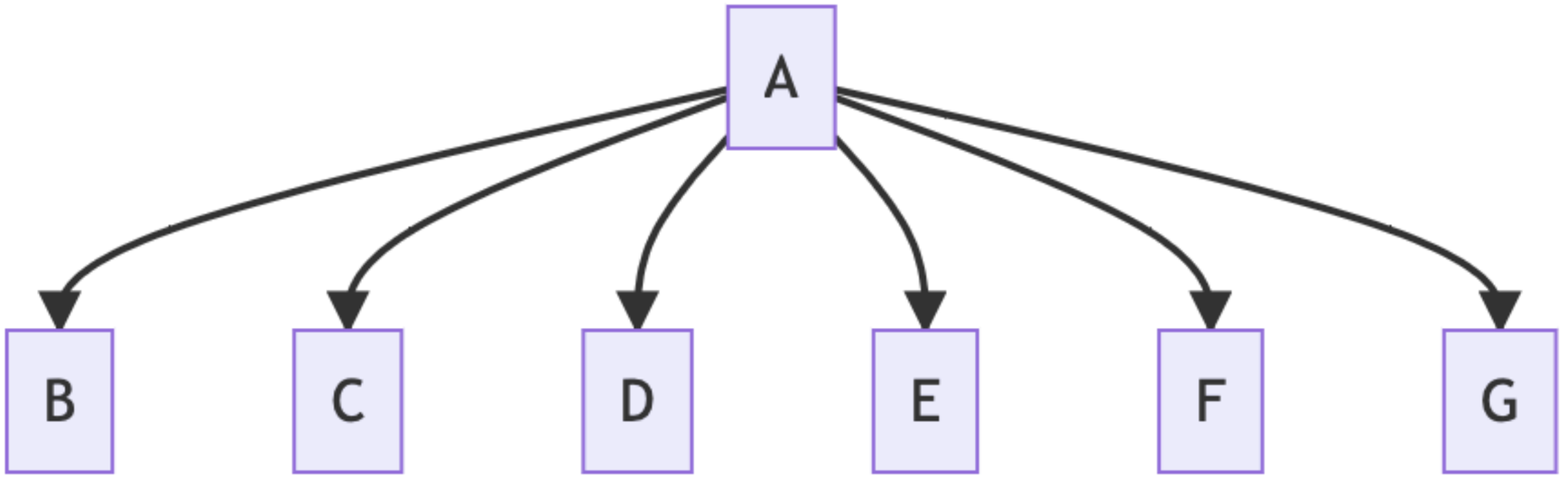
Decoupling navigation: Interpretation

```
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<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
)
```

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

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

```
var path = NavigationPath()
path.append(42)
path.append("Hello")
path.append(false)

var encodedData = try JSONEncoder().encode(path.codable!)

var decodedPath = NavigationPath(
  try JSONDecoder().decode(
    NavigationPath.CodableRepresentation.self,
    encodedData
  )
)
```

```
var path = NavigationPath()
path.append(42)
path.append("Hello")
path.append(false)

var encodedData = try JSONEncoder().encode(path.codable!)

var decodedPath = NavigationPath(
  try JSONDecoder().decode(
    NavigationPath.CodableRepresentation.self,
    encodedData
  )
)
```

```
var path = NavigationPath()
path.append(42)
path.append("Hello")
path.append(false)

var encodedData = try JSONEncoder().encode(path.codable!)

var decodedPath = NavigationPath(
  try JSONDecoder().decode(
    NavigationPath.CodableRepresentation.self,
    encodedData
  )
)
```

Pros and cons of the two initializers

Binding<[Element]>

Pros

Binding<[Element]>

Pros

» Strongly typed elements

Binding<[Element]>

Pros

- Strongly typed elements
- Full access to collection API

Binding<[Element]>

Pros

- Strongly typed elements
- Full access to collection API
- Instant testability

Binding<[Element]>

Pros

- Strongly typed elements
- Full access to collection API
- Instant testability
- Codable state restoration

Binding<[Element]>

Cons

Binding<[Element]>

Cons

» Some light coupling with `NavigationLink`

Binding<[Element]>

Cons

» Some light coupling with `NavigationLink`

» Single point of handling destinations

Binding<NavigationPath>

Pros

Binding<NavigationPath>

Pros

» Extremely easy to get started with

Binding<NavigationPath>

Pros

- Extremely easy to get started with
- Maximum decoupling

Binding<NavigationPath>

Cons

Binding<NavigationPath>

Cons

» Not testable

Binding<NavigationPath>

Cons

- Not testable
- Not inspectable

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 -> stack of screens
```

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 -> stack of screens

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 -> stack of screens
```

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 -> stack of screens
```

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 -> stack of screens
```

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)
}
```

```
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() }
      }
    }
  }
}
```

```
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() }
      }
    }
  }
}
```

```
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() }
      }
    }
  }
}
```

```swift
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


// screenA/screenB/screenA/screenB
struct Router: Parser {
  var body: some Parser<URLRequestData, [Destination]> {
    Many {
      DestinationRouter()
    }
  }
}
```

```
import URLRouting

// screenA/screenB/screenA/screenB
struct Router: Parser {
  var body: some Parser<URLRequestData, [Destination]> {
    Many {
      DestinationRouter()
    }
  }
}
```

URL Routing

```
let router = Router()
try router.match(path: "/screenA") ➡️ [.screenA]
try router.match(path: "/screenB") ➡️ [.screenB]
try router.match(path: "/screenC") ➡️ [.screenC(destination: nil)]

try router.match(path: "/screenC/sheet")
  ➡️ [.screenC(destination: .sheet(popoverValue: nil))]

try router.match(path: "/screenC/sheet/42")
  ➡️ [.screenC(destination: .sheet(popoverValue: 42))]
```

URL Routing

```
let router = Router()
try router.match(path: "/screenA") ➡️ [.screenA]
try router.match(path: "/screenB") ➡️ [.screenB]
try router.match(path: "/screenC") ➡️ [.screenC(destination: nil)]

try router.match(path: "/screenC/sheet")
  ➡️ [.screenC(destination: .sheet(popoverValue: nil))]

try router.match(path: "/screenC/sheet/42")
  ➡️ [.screenC(destination: .sheet(popoverValue: 42))]
```

URL Routing

```
let router = Router()
try router.match(path: "/screenA") ➡️ [.screenA]
try router.match(path: "/screenB") ➡️ [.screenB]
try router.match(path: "/screenC") ➡️ [.screenC(destination: nil)]

try router.match(path: "/screenC/sheet")
  ➡️ [.screenC(destination: .sheet(popoverValue: nil))]

try router.match(path: "/screenC/sheet/42")
  ➡️ [.screenC(destination: .sheet(popoverValue: 42))]
```

URL Routing

```
let router = Router()
try router.match(path: "/screenA") ➡️ [.screenA]
try router.match(path: "/screenB") ➡️ [.screenB]
try router.match(path: "/screenC") ➡️ [.screenC(destination: nil)]

try router.match(path: "/screenC/sheet")
  ➡️ [.screenC(destination: .sheet(popoverValue: nil))]

try router.match(path: "/screenC/sheet/42")
  ➡️ [.screenC(destination: .sheet(popoverValue: 42))]
```

URL Routing

```
let router = Router()
try router.match(path: "/screenA") ➡️ [.screenA]
try router.match(path: "/screenB") ➡️ [.screenB]
try router.match(path: "/screenC") ➡️ [.screenC(destination: nil)]

try router.match(path: "/screenC/sheet")
  ➡️ [.screenC(destination: .sheet(popoverValue: nil))]

try router.match(path: "/screenC/sheet/42")
  ➡️ [.screenC(destination: .sheet(popoverValue: 42))]
```

URL Routing

```
let router = Router()
try router.match(path: "/screenA") ➡️ [.screenA]
try router.match(path: "/screenB") ➡️ [.screenB]
try router.match(path: "/screenC") ➡️ [.screenC(destination: nil)]

try router.match(path: "/screenC/sheet")
  ➡️ [.screenC(destination: .sheet(popoverValue: nil))]

try router.match(path: "/screenC/sheet/42")
  ➡️ [.screenC(destination: .sheet(popoverValue: 42))]
```

URL Routing

```
let router = Router()
try router.match(path: "/screenA") ➡️ [.screenA]
try router.match(path: "/screenB") ➡️ [.screenB]
try router.match(path: "/screenC") ➡️ [.screenC(destination: nil)]

try router.match(path: "/screenC/sheet")
  ➡️ [.screenC(destination: .sheet(popoverValue: nil))]

try router.match(path: "/screenC/sheet/42")
  ➡️ [.screenC(destination: .sheet(popoverValue: 42))]
```

URL Routing

```
.onOpenURL { url in
  do {
    self.model.path = try Router().match(url: url)
  }
  catch {}
}
```

Demo

SwiftUI Navigation & URL Routing

- Brandon Williams
  - brandon@pointfree.co
  - @mbrandonw