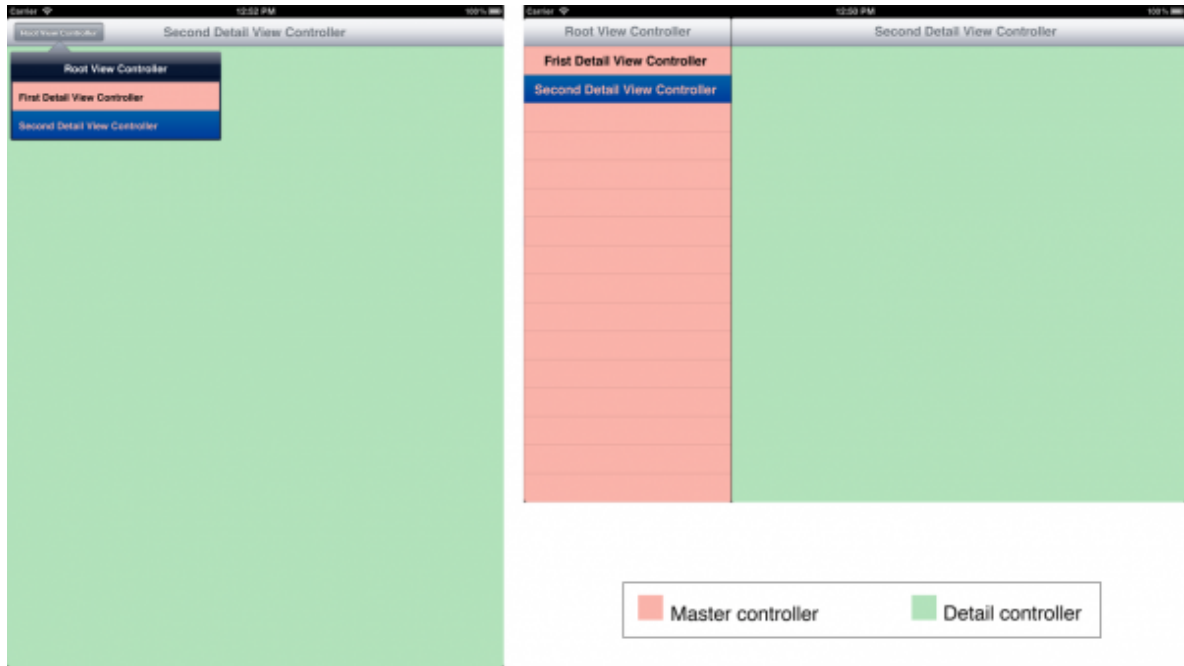


스터디 3주차

Segue

사전적으로 Segue(세그웨이)는 '하나에서 다른 것으로 부드럽게 넘어가다'라는 뜻을 가지고 있다. 앱으로 말하면 화면 전환을 뜻한다. 아이폰에서 어떤 버튼을 눌렀을 때 다른 화면으로 넘어가는 것이 바로 세그웨이이다.



아이패드앱 또는 아이폰/아이패드를 동시에 지원하는 유니버설 앱을 개발할 경우 위와 같이 화면 분할을 지원할 수 있다. UISplitViewController를 이용하여 구현하게 되는데 위에서 빨간색이 마스터뷰가 되며 녹색이 디테일뷰에 해당된다. 뷰를 로드할 때 이러한 두개의 영역에 로드할 수 있다.

Segue의 종류

1. show

화면에 보여지고 있는 마스터 또는 디테일 영역에 뷰를 로드한다. 마스터와 디테일 영역 모두 화면에 보여지고 있을 경우 로드되는 새로운 콘텐츠 뷰는 디테일 영역의 네비게이션 스택에 푸시된다. 마스터와 디테일 영역중 하나만 보여지고 있을 경우 현재 뷰컨트롤러 스택의 최상단에 푸시된다. 새로 나타나는 화면은 오른쪽에서 왼쪽으로 이동하며 보여지게 된다. 네비게이션바에 Back 버튼이 생기게 되고 아이패드, 아이폰 모든 디바이스에서 똑같이 작용한다.

2. show detail

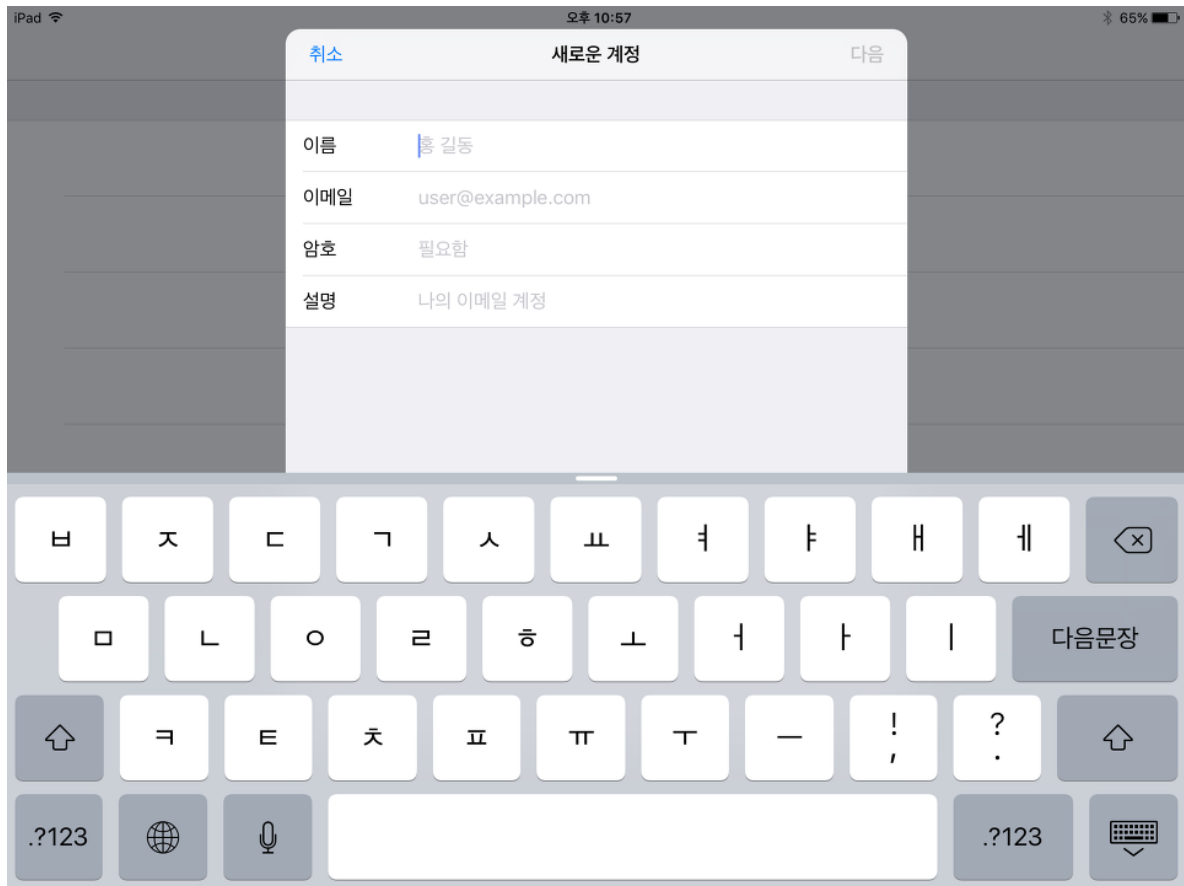
show와 매우 비슷하지만 푸시가 아닌 교체(replace)된다는 점이 크게 다르다. 마스터와 디테일 영역 모두 화면에 보여지고 있을 경우 로드되는 뷰는 디테일 영역을 교체하게 되며 둘중 하나만 보여지고 있을 경우 현재 뷰컨트롤러 스택의 최상단 뷰를 교체하게 된다.

ex) 아이패드에서 가로화면으로 이메일을 보고 있을 때, 이메일 내용을 클릭하면 디테일 뷰의 내

용이 바뀌게 된다.

3. present modally

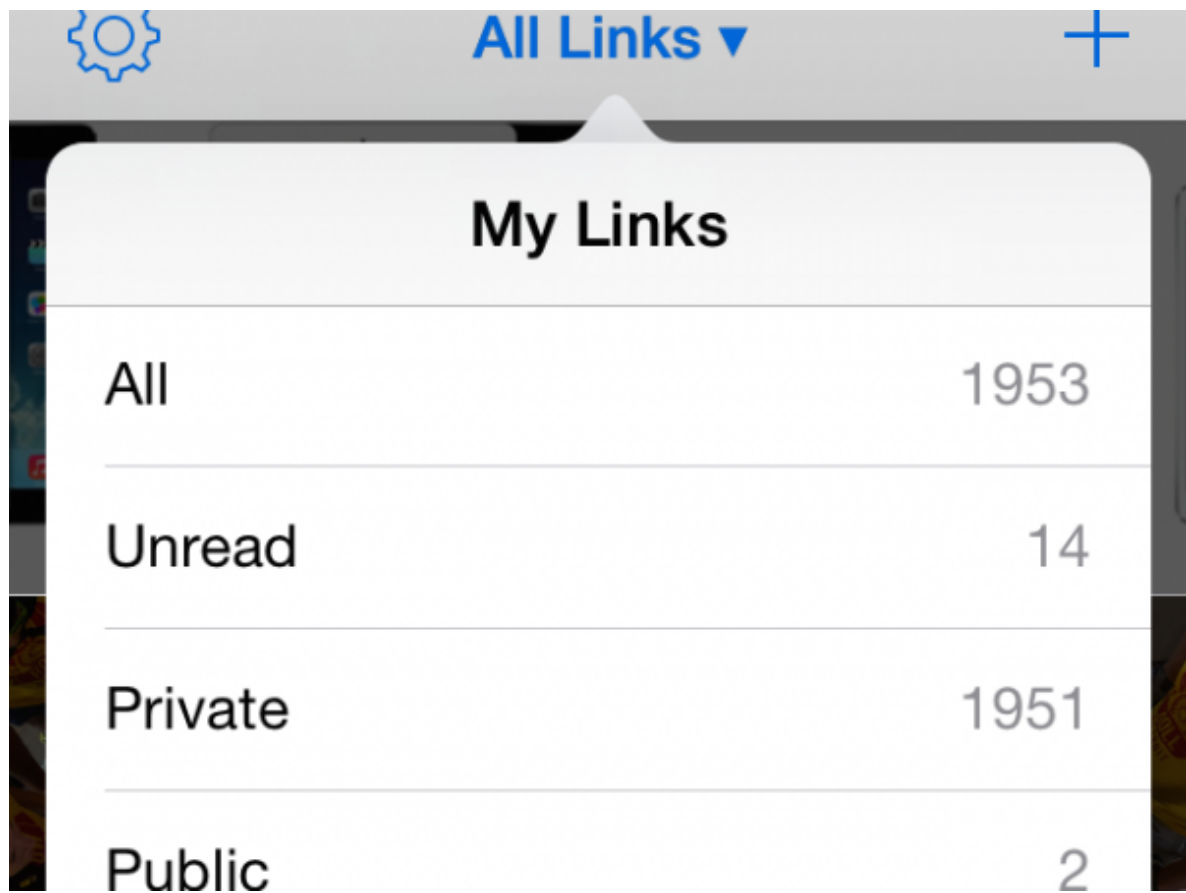
옵션에 따라 여러 다른 방법으로 기존 뷰컨트롤러를 덮는 새로운 뷰컨트롤러를 나타나게 한다. 주로 아래에서 위로 나타나는 뷰 컨트롤러를 나타나게 하기 위해 사용된다. 아이폰에서는 새로 나타난 뷰가 화면 대부분을 덮는다. 아이패드에서는 화면보다 작은 뷰가 나타나면서 배경이 흐릿하게 보여지게 된다.



4. popover presentation

현재 보여지고 있는 뷰 위에 앵커를 가진 팝업 형태로 콘텐츠 뷰를 로드한다.

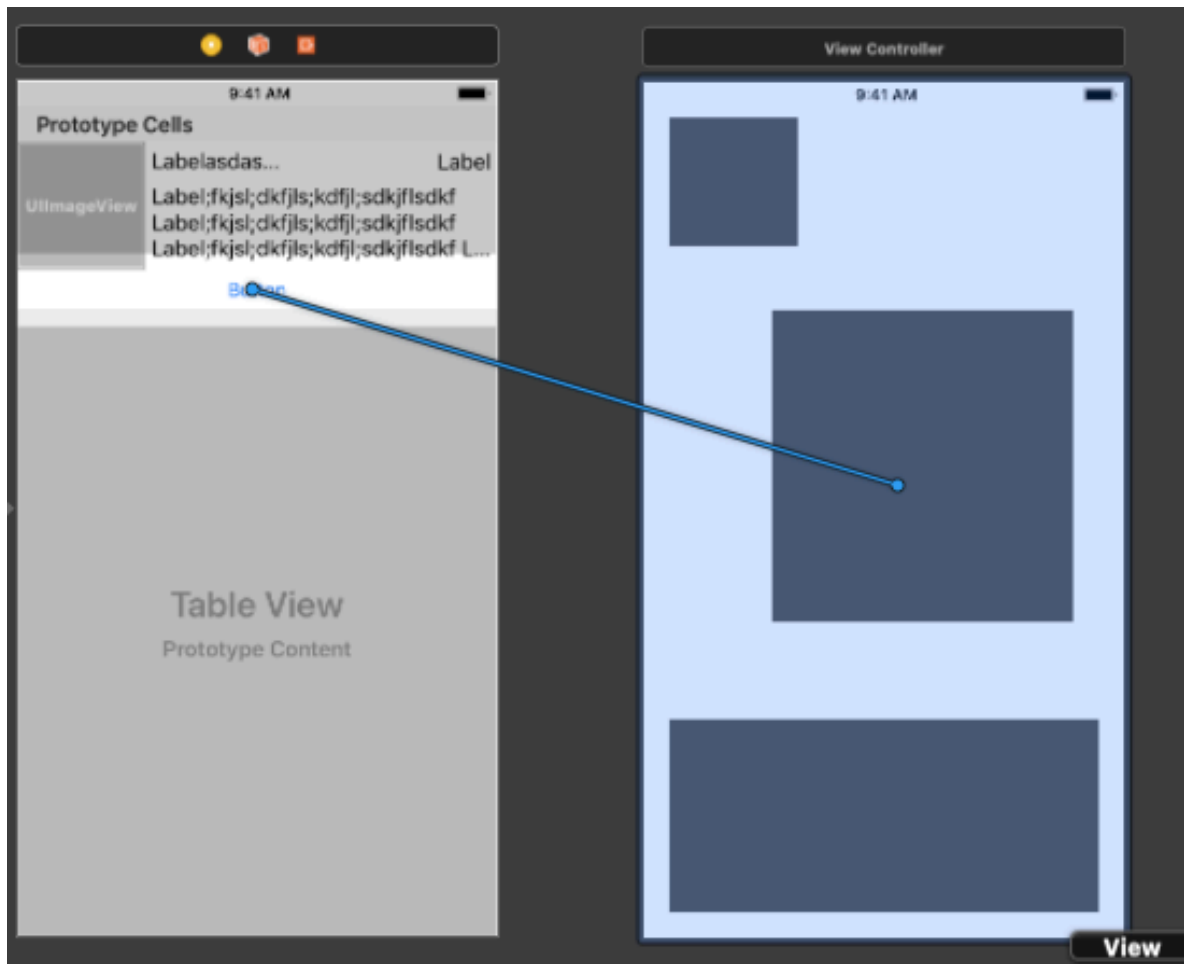
UIPopoverArrowDirection 옵션을 사용하여 창에 붙어있는 엣지의 방향을 설정 할 수 있다. 새로 띄운 뷰의 바깥영역을 터치하면 새로 띄운 뷰가 사라지게 된다. 아이폰에서는 모달뷰처럼 작동하게 된다.

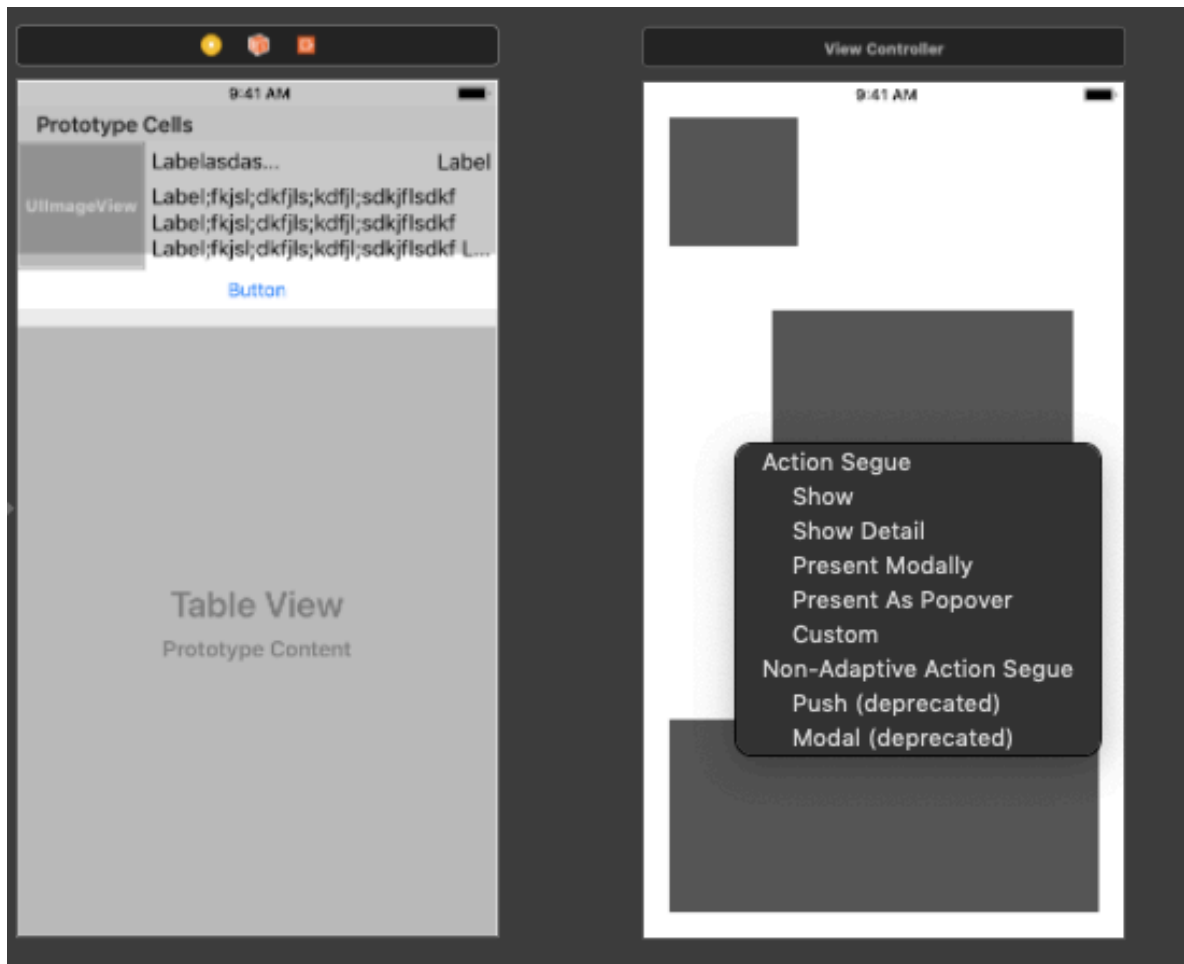


My Links	
All	1953
Unread	14
Private	1951
Public	2

5. custom

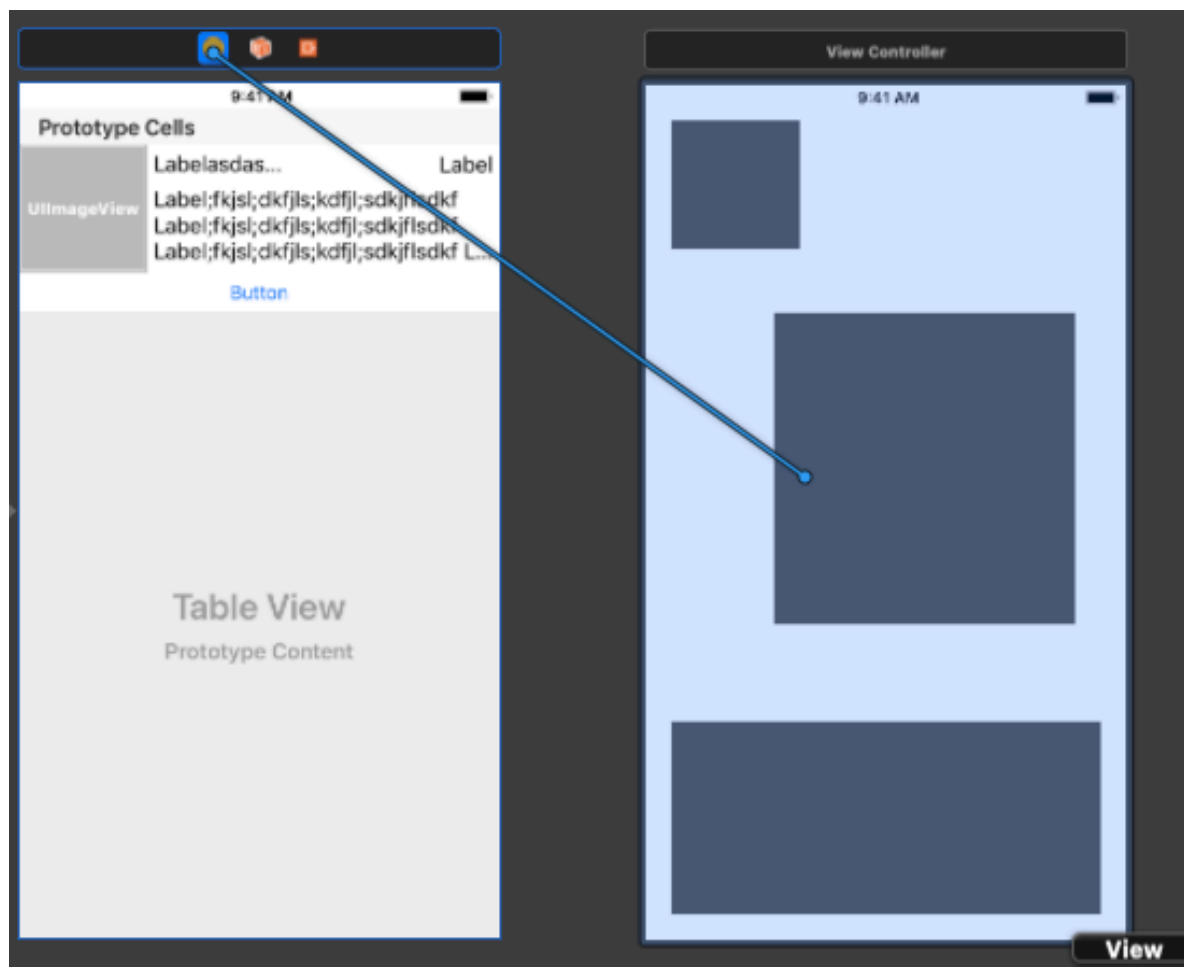
개발자가 임의로 지정한 동작을 수행 할 수 있다.

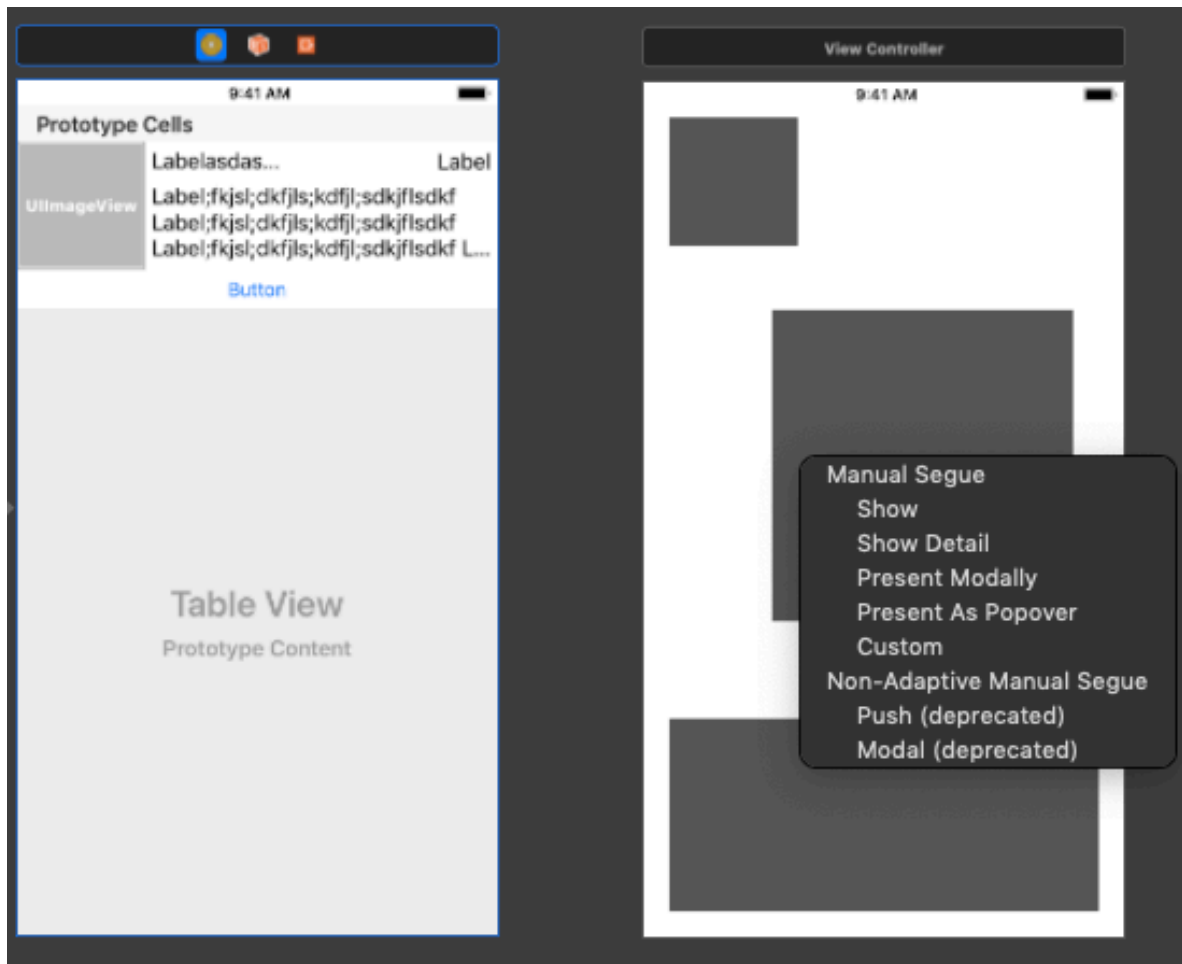




임의의 버튼을 컨트롤키와 함께 오른쪽 뷰까지 마우스 왼쪽 버튼 드래그를 해주면 다음과 같이 어떻게 뷰를 띄울것인지를 선택하는 팝업이 뜬다.

위의 방법의 경우 버튼을 통해 새로운 뷰로 이동하는 방법을 매우 쉽게 구현할 수 있지만 무언가 처리를 한 뒤에 이동하고 싶다면 코드를 이용한 Manual Segue 라고 불리는 시작 트리거가 없는 Segue를 이용하면 된다.





추가된 Segue를 선택한 뒤에 인스펙터를 확인해 보면 Storyboard Segue 탭에 Identifier를 지정 할 수 있다. 여기서 지정해 두면 코드상에서 이 Segue를 언제든지 수행할 수 있게 된다.

타입캐스팅

스위프트의 타입캐스팅은 `is`와 `as` 연산자를 이용해서 인스턴스의 타입을 확인하거나 자신을 다른 타입의 인스턴스인양 행세할 수 있는 방법으로 사용할 수 있다.
참조 타입에서 주로 사용된다.

ex)

```
print(coffee is Coffee)    //true
print(coffee is Americano) //false
print(myCoffee is coffee)  //true
```

`is` 연산자 외에도 메타 타입(Meta type) 타입을 이용하는 방법

타입의 타입을 뜻하며 타입 자체가 하나의 타입으로 또 표현할 수 있다는 것이다.

클래스, 구조체, 열거형 뒤에 `.Type`을 붙이면 메타 타입을 나타낸다.

프로토콜 타입의 메타 타입은 `.Protocol`이라고 붙여주면 된다.

ex)

```
SomeClass라는 클래스의 메타 타입은 SomeClass.Type
SomeProtocol의 메타 타입은 SomeProtocol.Protocol
```

```
let intType: Int.Type = Int.self
```

```
var someType: Any.Type
```

```
someType = intType  
print(someType)    //Int
```

.self 표현은 값 뒤에 써주면 그 값 자신을, 타입 이름 뒤에 써주면 타입을 표현하는 값을 반환한다.

만약 프로그램 실행 중에 인스턴스의 타입을 표현한 값을 알아보고자 한다면 type(of:) 함수를 사용한다.

```
print(type(of: coffee) == Coffee.self)    //true
```

다운캐스팅 : 클래스의 상속 모식도에서 자식클래스보다 더 상위에 있는 부모클래스의 타입을 자식클래스의 타입으로 캐스팅하는 것

타입캐스트 연산자에는 실패의 여지가 있기 때문에 as? 와 as! 두 가지가 있다.

as? 연산자는 다운캐스팅이 성공할 경우 옵셔널 타입으로 인스턴스를 반환하며 실패할 경우 nil 을 반환한다.

as! 연산자는 실패할 경우 런타임 오류가 발생하며 성공할 경우 반환타입은 옵셔널 타입이 아니다.

ex)

```
if let actingOne: Americano = coffee as? Americano
```

“만약 coffee가 참조하는 인스턴스가 Americano 타입의 인스턴스라면 actingOne이라는 임시 상수에 할당하라”

프로토콜

프로토콜은 특정 역할을 하기 위한 메소드, 프로퍼티, 기타 요구사항 들의 청사진을 정의 한다. 구조체, 클래스, 열거형은 프로토콜을 채택(adopted)해서 특정 기능을 실행하기 위한 프로토콜의 요구사항을 실제로 구현할 수 있다.

어떤 프로토콜의 요구사항을 모두 따르는 타입은 ‘해당 프로토콜을 준수한다(conform)’고 표현 한다.

프로토콜은 정의를 하고 제시를 할 뿐이지 스스로 기능을 구현하지는 않는다.

– 프로토콜 정의

```
protocol 프로토콜 이름 {  
    프로토콜 정의  
}
```

```
struct SomeStruct: AProtocol, AnotherProtocol {  
    //구조체 정의  
}
```

만약 클래스가 다른 클래스를 상속받는다면


```
class SomeClass: SuperClass, AProtocol, AnotherProtocol {
    //클래스 정의
}
```

- 프로퍼티 요구

프로토콜은 자신을 채택한 타입이 어떤 프로퍼티를 구현해야 하는지 요구할 수 있다. 어떤 프로퍼티(연산, 저장)인지는 신경 쓰지 않는다. 이름과 타입만 맞도록 구현해주면 된다. 다만 프로퍼티를 읽기 전용으로 할지 혹은 읽고 쓰기가 모두 가능하게 할지는 프로토콜이 정해야 한다.

읽고 쓰기 요구 -> 구현: 읽기 전용(X)

읽기 전용 요구 -> 구현: 읽기 쓰기(O) (읽기 포함해서 어떤 식으로든 가능)

프로토콜의 프로퍼티 요구사항은 항상 var 키워드를 사용한 변수 프로퍼티로 정의한다. 읽기 쓰기 모두 가능한 프로퍼티는 프로퍼티 정의 뒤에 { get set }, 읽기 전용 프로퍼티는 { get } 이라고 명시한다.

ex)

```
protocol SomeProtocol {
    var settableProperty: String { get set }
    var notNeedToBeSettableProperty { get }
}
```

타입 프로퍼티를 요구하려면 static 키워드를 사용한다.

클래스의 타입 프로퍼티에는 상속 가능한 타입 프로퍼티인 class 타입 프로퍼티와 상속 불가능한 static 타입 프로퍼티가 있지만 따로 구분하지 않고 모두 static 키워드를 사용하면 된다.

- 메서드 요구

프로토콜은 특정 인스턴스 메서드나 타입 메서드를 요구할 수도 있다. 프로토콜이 요구할 메서드는 프로토콜 정의에서 작성하며 메서드의 실제 구현부인 중괄호 부분은 제외하고 메서드 이름, 매개변수, 반환타입 등만 작성하며 가변 매개변수도 허용한다.

프로토콜의 메서드 요구에서는 매개변수 기본값을 지정할 수 없다. 타입 메서드는 마찬가지로 static 키워드를 명시한다. 실제 구현할 때는 static 키워드나 class 키워드 어느 쪽을 사용해도 무방하다.

ex)

```
protocol Receiveable {
    func received(data: Any, from: Sendable)
}
```

```
protocol Sendable {
    var from: Sendable { get }
    var to: Receiveable? { get }
```

```
    func send(data: Any)
```

```
    static func isSendableInstance(_ instance: Any) -> Bool
```

```
}
```

```
class Message: Sendable, Receiveable {  
    var from: Sendable {  
        return self  
    }  
  
    var to: Receiveable?  
  
    func send(data: Any) {  
        guard let receiver: Receiveable = self.to else {  
            print("Message has no receiver")  
            return  
        }  
  
        receiver.received(data: data, from: self.from)  
    }  
  
    func received(data: Any, from: Sendable) {  
        print("Message received \(data) from \(from)")  
    }  
  
    class func isSendableInstance(_ instance: Any) -> Bool {  
        if let sendableInstance: Sendable = instance as? Sendable {  
            return sendableInstance.to != nil  
        }  
        return false  
    }  
}
```

```
let myPhoneMessage: Message = Message()  
let tourPhoneMessage: Message = Message()
```

```
myPhoneMessage.send(data: "Hello") //Message has no receiver
```

```
myPhoneMessage.to = yourPhoneMessage  
myPhoneMessage.send(data: "Hello") //Message received Hello from  
Message
```

```
Message.isSendableInstance(myPhoneMessage) //true
```

– 가변 메서드 요구

값 타입(구조체, 열거형)의 인스턴스 메서드에서 자신 내부의 값을 변경하고자 할 때는 메서드의 func 키워드 앞에 mutating 키워드를 적어야 한다. 프로토콜에서도 인스턴스 내부의 값을 변경하는 메서드를 요구하려면 mutating 을 명시 해야한다. 클래스 구현에서는 mutating 키워드를 쓰지 않아도 된다.

즉, 만약 프로토콜에서 가변 메서드를 요구하지 않는다면, 값 타입의 인스턴스 내부 값을 변경하는 mutating 메서드는 구현이 불가능하다.

ex)

```
protocol Resettable {  
    mutating func reset()  
}
```

– 이니셜라이저 요구

프로토콜은 특정한 이니셜라이저를 요구할 수 있는데 마찬가지로 정의하지만 구현은 하지 않는다.

구조체는 상속할 수 없기 때문에 이니셜라이저 요구에 대해 크게 신경쓸 필요가 없지만 클래스의 경우 required 식별자를 붙인 요구 이니셜라이저로 구현해야 한다. 이는 곧 상속받는 클래스에 해당 이니셜라이저를 모두 구현해야 한다는 뜻이다.

ex)

```
protocol Named {  
    var name: String { get }  
  
    init(name: String)  
}
```

```
class Person: Named {  
    var name: String  
  
    required init(name: String) {  
        self.name = name  
    }  
}
```

클래스 자체가 상속받을 수 없는 final 클래스라면 required 식별자는 불필요하다.

프로토콜을 준수하면서 상속을 받은 클래스의 경우 override와 required 식별자를 순서 상관없이 모두 표기해야 한다.

ex)

```
class MiddleSchool: School, Named {  
    required override init(name: String) {  
        super.init(name: name)  
    }  
}
```

실패 가능한 이니셜라이저를 요구할 수도 있다. 이 프로토콜을 준수하는 타입은 해당 이니셜라이저를 구현할 때 실패 가능한 이니셜라이저로 구현해도, 일반적인 이니셜라이저로 구현해도 무방하다.

ex)

```
protocol Named {  
    var name: String { get }  
  
    init?(name: String)
```

```
}
```

프로토콜도 상속이 가능하며 상속 리스트에 class 키워드를 추가해 프로토콜이 클래스 타입에만 채택될 수 있도록 제한할 수도 있다.

ex)

```
protocol ClassOnlyProtocol: class, Readable, Writable{
    //요구 사항
}
```

- 프로토콜 조합과 프로토콜 준수 확인

하나의 매개변수가 프로토콜 둘 이상을 요구할 수도 있다.

클래스 & 프로토콜 조합에서 클래스 타입은 한 타입만 조합할 수 있다.(구조체나 열거형 타입은 조합할 수 없다)

ex)

```
var someVariable: Car & Truck & Aged //클래스 2개, 오류
```

```
func celebrateBirthday(to celebrator: Named & Aged) {
    print("Happy birthday \(celebrator.name)!! Now you are \(celebrator.age)"
}
```

is 와 as 연산자를 통해 대상이 프로토콜을 준수하는 확인할 수도 있고, 특정 프로토콜로 캐스팅 할 수 있다.

ex)

```
protocol Named {
    var name: String { get }
}
```

```
struct Person: Named {
    var name: String
}
```

```
let sample: Person
print(sample is Named) //true
```

```
if let castedInstance: Named = sample as? Named {
    print("\(castedInstance) is Named")
} //Persone is Named
```

- 프로토콜의 선택적 요구

@objc(Objective-C코드에서 사용할 수 있도록 만드는 역할) 속성이 부여된 프로토콜의 요구 사항 중 일부를 선택적 요구사항으로 지정할 수 있다. 클래스에서만 채택할 수 있고 열거형이나 구조체 등에서는 아예 채택할 수 없다.

익스텐션

구조체, 클래스, 열거형, 프로토콜 타입에 새로운 기능을 추가할 수 있다. 추가할 수 있는 기능은 다음과 같다.

- 연산 타입 프로퍼티 / 연산 인스턴스 프로퍼티
- 타입 메서드 / 인스턴스 메서드
- 이니셜라이저
- 서브스크립트
- 중첩 타입
- 특정 프로토콜을 준수할 수 있도록 기능 추가

익스텐션은 타입에 새로운 기능을 추가할 수는 있지만, 기존에 존재하는 기능을 재정의할 수는 없다.

	상속	익스텐션
확장	수직 확장	수평 확장
사용	클래스 타입에서만 사용	클래스, 구조체, 프로토콜, 제네릭 등 모든 타입에서 사용
재정의	가능	불가능

-익스텐션 문법

```
extension 확장할 타입 이름 {
    //타입에 추가될 새로운 기능 구현
}
```

기존에 존재하는 타입이 추가로 다른 프로토콜을 채택할 수 있도록 확장할 수도 있다.

```
extension 확장할 타입 이름: 프로토콜1, 프로토콜2, 프로토콜3 {
    //프로토콜 요구사항 구현
}
```

익스텐션을 통해 연산 프로퍼티 추가할 수 있다.

```
ex)
extension Int {
    var isEven: Bool {
        return self % 2 == 0
    }

    var isOdd: Bool {
        return self % 2 == 1
    }
}
```

```
print(1.isEven) //false
print(2.isEven) //true
```

static 키워드를 사용하여 타입 연산 프로퍼티도 추가할 수 있다.
익스텐션으로 저장프로퍼티와 프로퍼티 감시자는 추가할 수 없다.

익스텐션을 통해 메서드를 추가할 수 있다.

```
ex)
extension Int {
```

```

    func multiply(by n: Int) -> Int {
        return self * n
    }
}
print(3.multiply(by: 2))    //6

```

타입의 정의 부분에 이니셜라이저를 추가하지 않더라도 익스텐션을 통해 이니셜라이저를 추가할 수 있다.

익스텐션으로 클래스 타입에 편의 이니셜라이저는 추가할 수 있지만, 지정 이니셜라이저는 추가할 수 없다.

ex)

```

class Person {
    var name: String

    init(name: String) {
        self.name = name
    }
}

extension Person {
    convenience init() {
        self.init(name: "Unknown")
    }
}

let someone: Person = Person()
print(someone.name)    //Unknown

```

제네릭

제네릭을 사용하면 어떤 타입에도 유연하게 대응할 수 있고 재사용하기도 쉽고, 코드의 중복을 줄일 수 있다.

제네릭이 필요한 타입 또는 메서드의 이름 뒤의 <> 사이에 제네릭을 위한 타입 매개변수를 써준다.

ex)

```

func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA: T = a
    a = b
    b = temporaryA
}

```

- 제네릭 함수

제네릭 함수는 실제 타입 이름(Int, String 등)을 써주는 대신에 플레이스 홀더(위 함수에서는 T)를 사용한다. 플레이스홀더(T)는 타입의 종류를 알려주지 않지만 말 그대로 어떤 타입이라는 것은 알려준다. 즉, 매개변수로 플레이스홀더 타입이 T인 두 매개변수가 있으므로, 두 매개변수는 같은 타입이라는 것을 알 수 있다. T의 실제 타입은 함수가 호출되는 그 순간 결정된다. 타입 매개변수는 플레이스 홀더 타입의 이름을 지정하고 명시하는 역할을 하며, 함수의 이름 뒤 <>안쪽에

위치 한다.의미있는 이름으로 타입 매개변수의 이름을 지정해주거나 관용적으로 T, U, V로 표현 한다.

- 제네릭 타입

제네릭 타입을 구현하면 사용자 정의 타입인 구조체, 클래스, 열거형 등이 어떤 타입과도 연관되어 동작할 수 있다.

ex)

```
struct Stack<Element> {  
    var items = [Element]()  
    mutating func push(_ item: Element) {  
        items.append(item)  
    }  
    mutating func pop() -> Element {  
        return items.removeLast()  
    }  
}
```

```
var doubleStack: Stack<Double> = Stack<Double>()  
var stringStack: Stack<String> = Stack<String>()  
var anyStack: Stack<Any> = Stack<Any>()
```

익스텐션을 통해 제네릭을 확장한다면 익스텐션 정의에 타입 매개변수를 명시하지 않아야 한다. 대신 원래의 제네릭 정의에 명시한 타입 매개변수를 사용할 수 있다.

ex)

```
extension Stack {  
    var topElement: Element? {  
        return self.items.last  
    }  
}
```

- 제네릭에 타입제약은 클래스 타입 또는 프로토콜로만 줄 수 있다.

열거형, 구조체 등의 타입은 타입 제약의 타입으로 사용할 수 없다.

ex)

```
func swapTwoValues<T: BinaryInteger>(_ a: inout T, _ b: inout T) {  
    //함수 구현  
}  
func Stack<Element: Hashable> {  
    //구조체 구현  
}
```

여러 제약을 추가하고 싶다면

```
func swapTwoValues<T: BinaryInteger>(_ a: inout T, _ b: inout T) where T: FloatingPoint, T: Equatable {  
    //함수 구현  
}
```

- 프로토콜의 연관 타입

연관타입은 프로토콜에서 사용할 수 있는 플레이스 홀더 이름인데, 타입 매개변수의 그 역할을 프로토콜에서 실행할 수 있도록 만들어진 기능이다.