Introduction to SwiftUI
Data Management

Girish Lukka

Topics

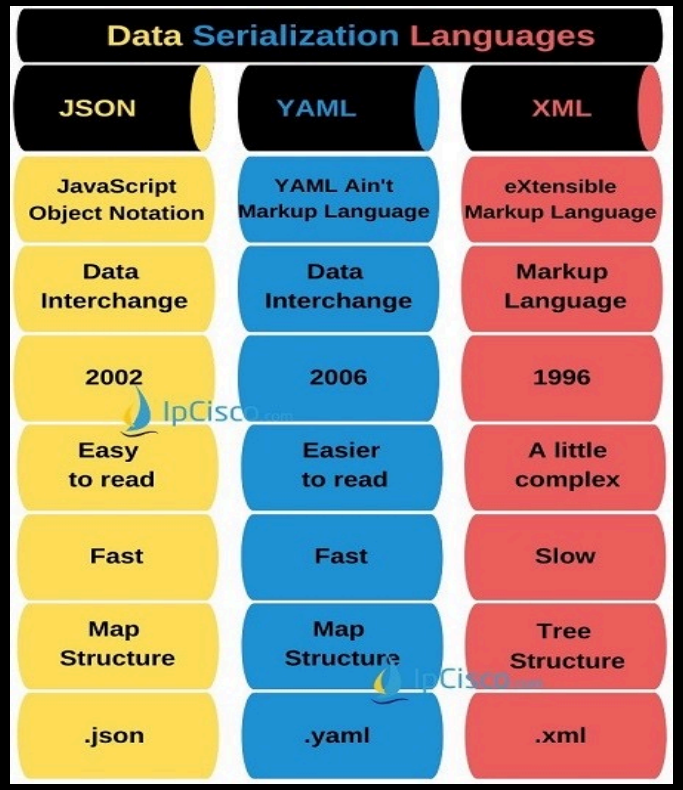
- App Bundle
- **Introduction to json**
- Mapping json object to swift struct
- Data persistence

History

- Web 1.0 – SGML, HTML
- Web1.1 – XML, HTML XHTML
- XML provided a technology to store, communicate, and validate any kind of data, in a form that applications on any platform can easily read and process.
- The delivery of pre-rendered HTML pages to the browser was not scalable - Web1.2 AJAX
  - HTML (or XHTML) and CSS for presentation.
  - The Document Object Model (DOM) for dynamic display of, and interaction with, data.
  - XML for the interchange of data, and XSLT for its manipulation.
  - The XMLHttpRequest object for asynchronous communication.
  - JavaScript to bring these technologies together.

History

- Web 2.0 – Single-page Applications
- Emergence of json as "most popular data interchange format.”
- In a nutshell:
- Json is lightweight, faster and compact
- Xml is more versatile, has presentation capability and is more secure.
- Good summary of the both – Comparison



Data interchange technology

JSON (JavaScript Object Notation): JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is often used for exchanging data between web applications and APIs. JSON is based on a subset of the JavaScript programming language and can be used with many programming languages, including Java, Python, and Swift.

XML (Extensible Markup Language): XML is a markup language that is used to structure and store data in a human-readable and machine-readable format. XML is often used for exchanging data between web services or between applications. XML is highly customizable, and can be used to describe complex data structures

Data interchange technology

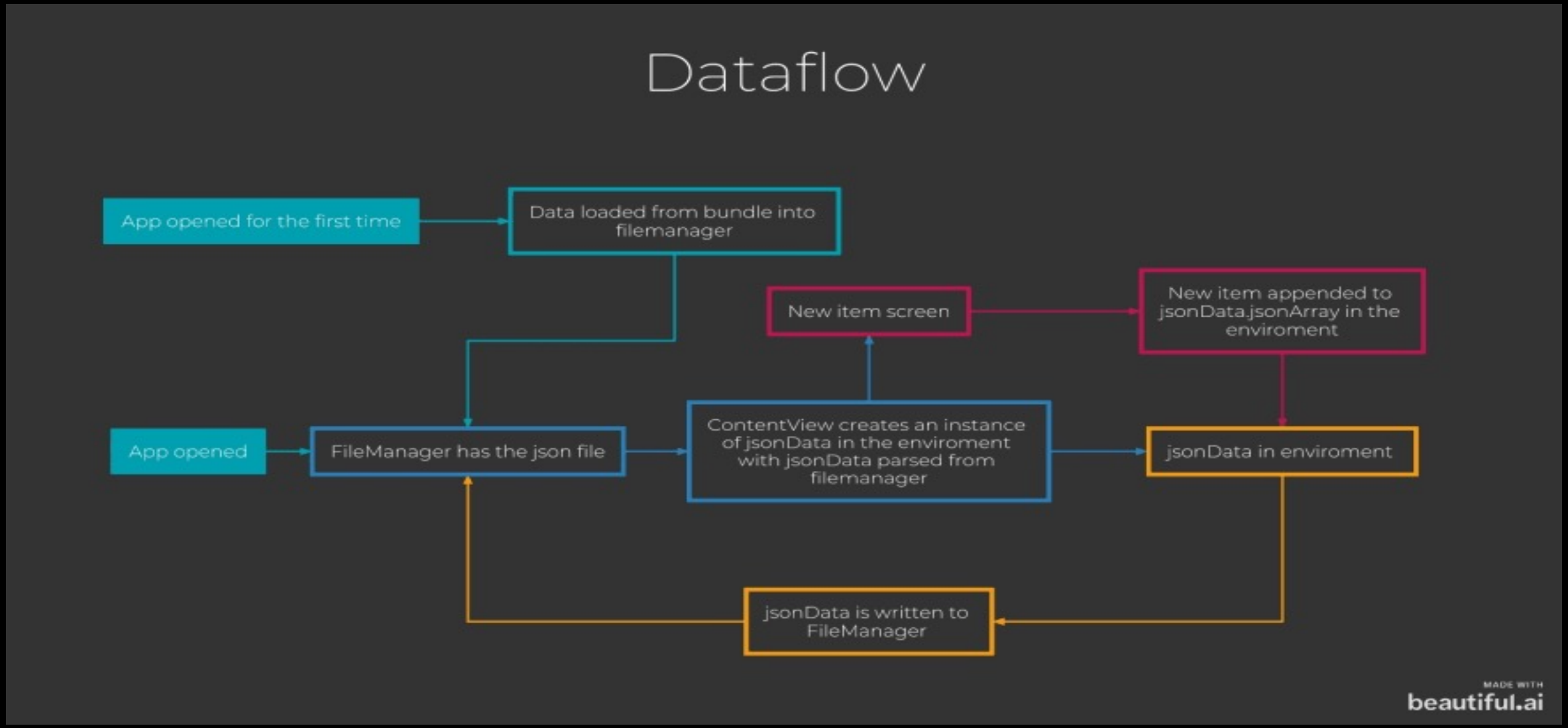
CSV (Comma Separated Values): CSV is a simple data interchange format that is used to store tabular data in a plain text format. CSV files consist of rows of data, with each row separated by a newline character, and each column separated by a comma. CSV files can be easily imported and exported by spreadsheet software such as Excel or Google Sheets.

GraphQL: GraphQL is a query language and runtime for APIs that is designed to enable efficient and flexible data interchange between clients and servers. GraphQL allows clients to specify the exact data they need and reduces the amount of network traffic required to fetch data. GraphQL can be used with many programming languages and frameworks, including React, Node.js, and Ruby on Rails.

JSON – most popular

1. Lightweight: JSON is a lightweight format that uses a simple structure of key-value pairs to represent data. This makes it easy to read and parse, and it also reduces the size of the data that needs to be transferred over the network.
2. Easy to read and write: JSON is designed to be human-readable and easy to write. This makes it easier for developers to work with JSON data, and it also makes it easier to debug and troubleshoot.
3. Easy to parse: JSON is easy for machines to parse, which means that it can be easily converted into objects or data structures in a programming language. This makes it easier to work with JSON data in applications and systems.
4. Wide support: JSON is supported by most programming languages and platforms, including JavaScript, Python, Ruby, Java, and .NET. This makes it easy to use JSON in a variety of contexts, from web development to mobile app development to server-side applications.
5. Versatility: JSON is a versatile format that can be used for a wide range of applications, from web APIs to data storage to configuration files. This makes it a popular choice for many developers and organizations.

Dataflowdata



Dataflowdata – using Bundle

- The Bundle framework provides a simple and consistent way to load resources from the app bundle and make them available to the rest of the application. When an app is built, Xcode creates a bundle that contains all the resources that are needed to run the app, including images, sounds, and data files. The Bundle framework provides APIs for accessing these resources and loading them into memory.

- Code demo - PizzaApp

What is JSON?

JSON stands for **JavaScript Object Notation**.

JavaScript started out as a new language in 1995 as a way for internet browsers to run code.

When calling services to get data over the internet there needed to be a way to express objects as text.

Example:

```
{
    “field1”: “value1”,
    “field2”: “value2”
}
```

Notation means “a system of writing used to represent numbers, amounts, or elements in something”.
Such as shown here by representing data in a structured way with text.

JSON encloses data with **braces**. The braces usually specify a single object which we would recognize as a class or struct.

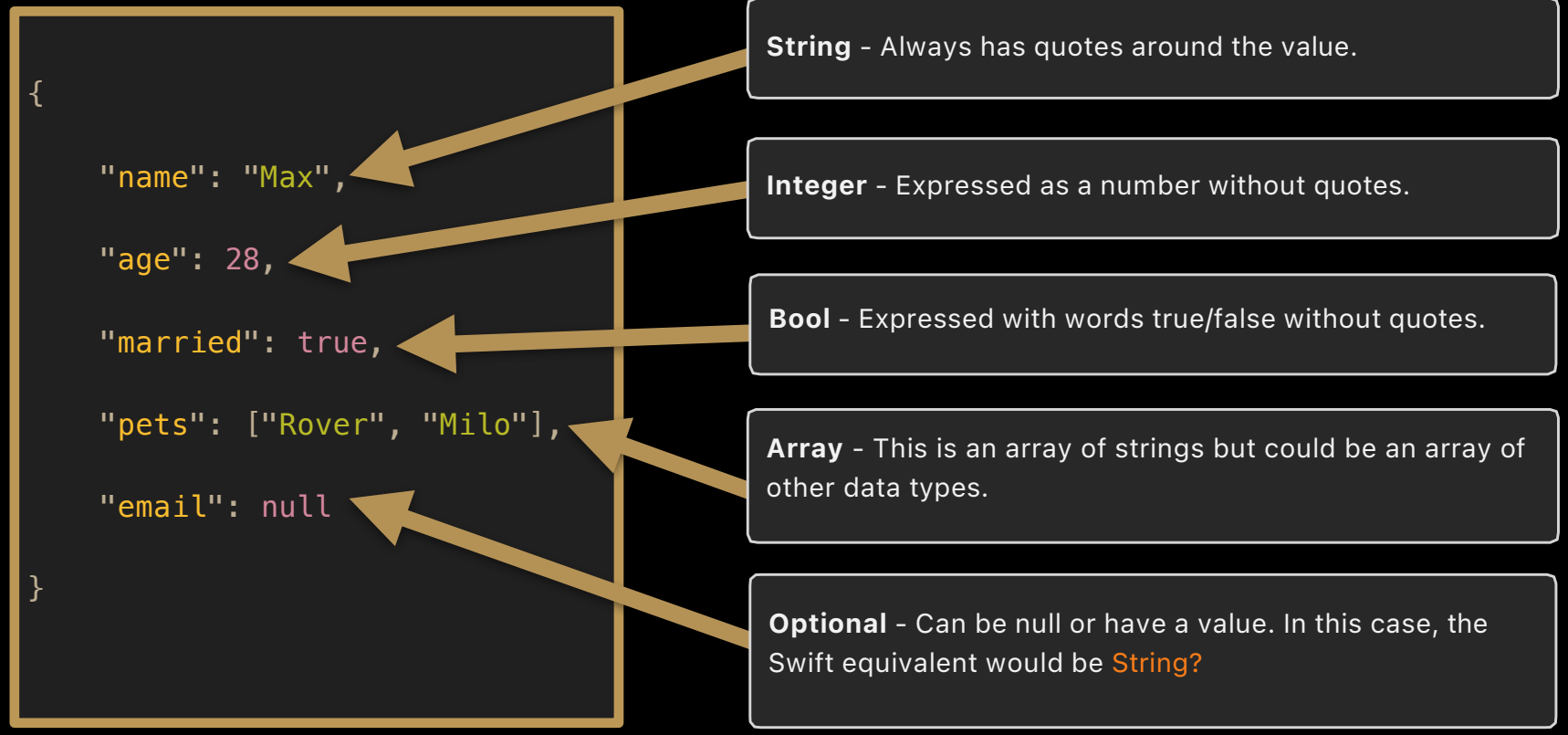
Data is identified with a field or property name.

```
{
    “firstName”:“Joe”,
    “lastName”: “Smith”
}
```

The value comes after the field/ property name and a colon.

Data Types

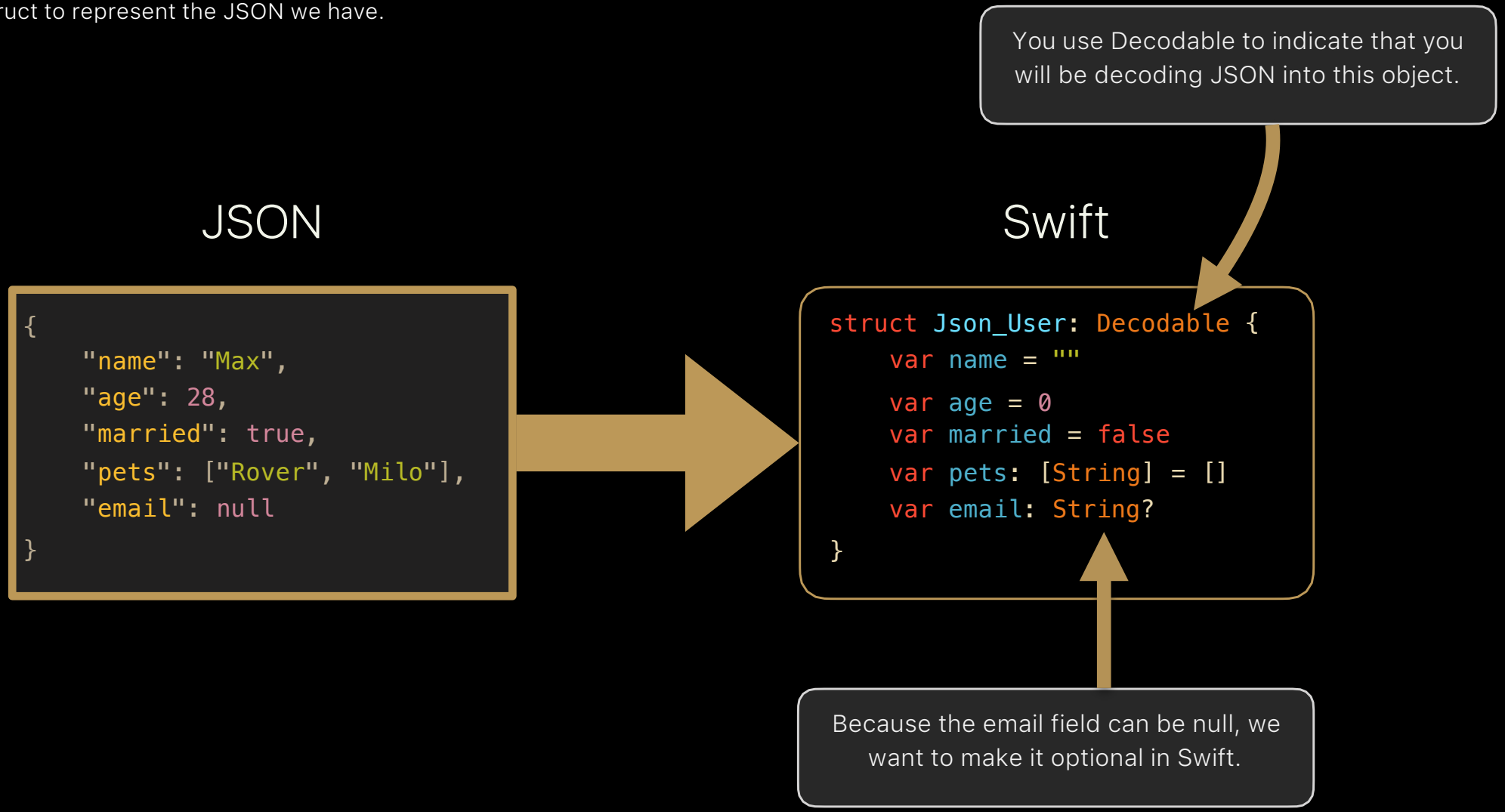
Here are some ways JSON expresses different data types.



The goal is for you to be able to look at the JSON you are receiving and create a matching struct or class to easily move this data into by using “decoding”. JSON is the “coded” part. When you **decode**, you transfer values from JSON to a Swift object.

Decodable

Here is the struct to represent the JSON we have.



Decoding (Observable Object)

Decoding JSON into a Swift object will most likely happen in your observable object. Here is an example of how this might happen.

```
var jsonString = """
{
    "name": "Max",
    "age": 28,
    "married": true,
    "pets": ["Rover", "Milo"],
    "email": null
}
"""

class Json_DecodingOO: ObservableObject
    { @Published var user = Json_User()
    @Published var jsonError: Error?

    func fetch() {
        let jsonData = jsonString.data(using: .utf8)!

        let decoder = JSONDecoder()

        do {
            user = try decoder.decode(Json_User.self, from: jsonData)
        } catch {
            jsonError = error
        }
    }
}
```

First, you want to convert your JSON to data if it's not already in this format.

What is utf8? See next page.

You will also need a JSONDecoder to do the work of transferring the data from JSON to your object.

This is where the actual decoding happens. You specify the destination type and the JSON data to decode.

Sometimes errors can happen if you don't set up the destination type correctly to match the JSON.

What is UTF-8?

UTF stands for “Unicode Transformation Format”.

What is “Unicode”?
Unicode is an international standard of representing characters, letters and even emoji’s using a specified code. (This code will eventually be converted into ones and zeroes.)

Example:

Character	Unicode	Binary
A	U+0041	01010101 00101011 00110000 00110000 00110100 00110001
Ä	U+00C4	01010101 00101011 00110000 00110000 01000011 00110100
😃	U+1F603	01010101 00101011 00110001 01000110 00110110 00110000 00110011

Binary is what is used by the computer (processor).

8

Why is it utf-**8** though?
The 8 specifies how “bits” (1 or 0) will be used.
If you look at the Binary column, you will notice there are groups of 8 bits.

You convert your JSON string into Unicode first before you can decode it into a struct or class:

```
let jsonData = jsonString.data(using: .utf8)!
```

Json structure


```
[
  {
    "id": "1010",
    "name": "girish",
    "mark": "100",
    "status": "active",
    "pic": "image/002.jpg"
  },
  {
    "id": "1020",
    "name": "mario",
    "mark": 60,
    "status": "active",
    "pic": "image/003.jpg"
  },
  {
    "id": "1030",
    "name": "andy",
    "mark": 70,
    "status": "active",
    "pic": "image/004.jpg"
  }
]
```


A collection of name/value pairs. Different programming languages support this data structure in different names. Like object, record, struct, dictionary, hash table, keyed list, or associative array.
It can also be nested

```
let jsonData =  """
{
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874",
      }
    }

""".data(using: .utf8)
```

Json parsing

- Json keys are usually written as snake_case
- Swift is camelCase
- Map json object to a swift object

Coding Keys

Coding keys are necessary when:

- A Swift variable/property name must not start with a number. If a key does start with a number you have to specify a compatible CodingKey to be able to decode the key at all.
- You want to use a different property name.
- You want to exclude keys from being decoded for example an id property which is not in the JSON and is initialized with an UUID constant.

Flattening a json - demo

```
let jsonData =  """
{
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874",
      }
    }

""".data(using: .utf8)
```

```
struct User: Decodable {
    var id: Int?
    var name: String = ""
    var userName: String = ""
    var email: String = ""
    var street: String = ""
    var suite: String = ""
    var city: String = ""
    var zipCode: String = ""

    //coding keys
    private enum UserKeys: String, CodingKey{
        case id
        case name
        case userName = "username"
        case email
        case address
    }

    private enum AddressKeys: String, CodingKey{
        case street
        case suite
        case city
        case zipCode = "zipcode"

    }
```

Flattening a json

```
    init(from decoder: Decoder) throws {
        if let userContainer = try? decoder.container(keyedBy: UserKeys.self){
            self.id = try userContainer.decode(Int.self, forKey: .id)
            self.name = try userContainer.decode(String.self, forKey: .name)
            self.userName = try userContainer.decode(String.self, forKey: .userName)
            self.email = try userContainer.decode(String.self, forKey: .email)

            if let addressContainer = try?userContainer.nestedContainer(keyedBy: AddressKeys.self, forKey: .address){
                self.street = try addressContainer.decode(String.self, forKey: .street)
                self.suite  = try addressContainer.decode(String.self, forKey: .suite)
                self.city = try addressContainer.decode(String.self, forKey: .city)
                self.zipCode = try addressContainer.decode(String.self, forKey: .zipCode)

            }
        }
    }
}
if let user = try? JSONDecoder().decode(User.self, from: jsonData!){
    print(user.name)
    print(user.city)
    print(user.email)
    print(user)

}
```

Using a nested scheme - demo

```
let jsonData =  """
{
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874",
      }
    }

""".data(using: .utf8)
```

```
struct User: Decodable {

    let id: Int
    let name: String
    let username: String
    let email: String
    let address: Address


    struct Address: Codable {


        let street: String
        let suite: String
        let city: String
        let zipcode: String

    }


}
```

Mapping guidelines

- When first encountering mapping json objects to swift, the key issues are:
- When coding keys are not needed and why
- When to use coding keys without init()
- When to use coding keys and init()
- Types of init() – object creation directly, object creating from json data and mapping swift object to json

Mapping – init() functions

```swift
struct Person: Codable {
    var name: String
    var age: Int
    var address: String

    enum CodingKeys: String, CodingKey {
        case name
        case age
        case address = "home_address"
    }

    init(name: String, age: Int, address: String) {
        self.name = name
        self.age = age
        self.address = address
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        name = try container.decode(String.self, forKey: .name)
        age = try container.decode(Int.self, forKey: .age)
        address = try container.decode(String.self, forKey: .address)
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(name, forKey: .name)
        try container.encode(age, forKey: .age)
        try container.encode(address, forKey: .address)
    }
}
```

Mapping – no coding keys nor init() why?

```swift
import Foundation

let jsonData =  """
{
  "name": "John Smith",
  "age": 32,
  "isEmployed": true
}
""".data(using: .utf8)

struct Person: Decodable {
    let name: String
    let age: Int
    let isEmployed: Bool
}
if let person = try? JSONDecoder().decode(Person.self, from: jsonData!){
    print(person.name)
    print(person.age)
    print(person)
}

```

```
John Smith
32
Person(name: "John Smith", age: 32, isEmployed: true)
```

Mapping – coding keys but no init() why?

```swift
import Foundation

let jsonData =  """
{
  "first_name": "John",
  "last_name": "Doe",
  "age": 30
}

""".data(using: .utf8)

struct Person: Decodable {
    let firstName: String
    let lastName: String
    let age: Int

    enum CodingKeys: String, CodingKey {
        case firstName = "first_name"
        case lastName = "last_name"
        case age
    }
}
if let person = try? JSONDecoder().decode(Person.self, from: jsonData!){
    print(person.firstName)
    print(person.lastName)
    print(person)

}
```

```
John
Doe
Person(firstName: "John", lastName: "Doe", age: 30)
```

Mapping – flattening json with nested coding keys and init(from:)

```swift
import UIKit

let jsonData =  """
{
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874",
      }
    }

""".data(using: .utf8)

//User flat model

struct User: Decodable {
    var id: Int?
    var name: String = ""
    var userName: String = ""
    var email: String = ""
    var street: String = ""
    var suite: String = ""
    var city: String = ""
    var zipCode: String = ""

```

```swift
    //coding keys
    private enum UserKeys: String, CodingKey{
        case id
        case name
        case userName = "username"
        case email
        case address
    }

    private enum AddressKeys: String, CodingKey{
        case street
        case suite
        case city
        case zipCode = "zipcode"

    }

```

Mapping – flattening json with nested coding keys and init(from:), continued

```
48      init(from decoder: Decoder) throws {
49          if let userContainer = try? decoder.container(keyedBy: UserKeys.self){
50              self.id = try userContainer.decode(Int.self, forKey: .id)
51              self.name = try userContainer.decode(String.self, forKey: .name)
52              self.userName = try userContainer.decode(String.self, forKey: .userName)
53              self.email = try userContainer.decode(String.self, forKey: .email)
54
55              if let addressContainer = try?userContainer.nestedContainer(keyedBy: AddressKeys.self,
                    forKey: .address){
56                  self.street = try addressContainer.decode(String.self, forKey: .street)
57                  self.suite  = try addressContainer.decode(String.self, forKey: .suite)
58                  self.city = try addressContainer.decode(String.self, forKey: .city)
59                  self.zipCode = try addressContainer.decode(String.self, forKey: .zipCode)
60
61              }
62          }
63      }
64  }
65  if let user = try? JSONDecoder().decode(User.self, from: jsonData!){
66      print(user.name)
67      print(user.city)
68      print(user.email)
69      print(user)
70  }
```

```
Leanne Graham
Gwenborough
Sincere@april.biz
User(id: Optional(1), name: "Leanne Graham", userName: "Bret", email: "Sincere@april.biz", street: "Kulas Light", su
"Gwenborough", zipCode: "92998-3874")
```

Mapping – nested json to nested swift no keys nor init() – why?

```swift
let jsonData =  """
{
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874",
       }
    }
""".data(using: .utf8)
// change username to userName and you need coding keys!
struct User: Decodable {
    let id: Int
    let name: String
    let username: String
    let email: String
    let address: Address

    struct Address: Codable {
        let street: String
        let suite: String
        let city: String
        let zipcode: String

    }
}
if let user = try? JSONDecoder().decode(User.self, from: jsonData!){
    print(user.name)
    print(user.address.city)
    print(user.email)
    print(user)
}
```

```
Leanne Graham
Gwenborough
Sincere@april.biz
User(id: 1, name: "Leanne Graham", username: "Bret", email: "Sincere@april.biz", address: __lldb_expr_20
556", city: "Gwenborough", zipcode: "92998-3874"))
```

Mapping – nested json to nested swift with keys but no init() why?

```swift
let jsonData =  """
{
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874"
    }
}
""".data(using: .utf8)

struct User: Decodable {

    let id: Int
    let name: String
    let userName: String
    let email: String
    let address: Address

    struct Address: Codable {
        let street: String
        let suite: String
        let city: String
        let zipcode: String
    }
    private enum CodingKeys: String, CodingKey{
        case id
        case name
        case userName = "username"
        case email
        case address
    }
}
```

```swift
do {
    let user = try JSONDecoder().decode(User.self, from: jsonData!)
    print(user.name)
    print(user.address.city)
    print(user.email)
    print(user)
} catch {
    print("Error decoding JSON: \(error)")
}
```

```
Leanne Graham
Gwenborough
Sincere@april.biz
User(id: 1, name: "Leanne Graham", userName: "Bret", email: "Sincere@april.biz", address: __lldb_expr_203.User.Address(street: "Kula
zipcode: "92998-3874"))
```

Mapping – nested json to nested swift no keys nor init() – why?

```swift
let jsonData =  """
{
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874",
       }
    }
""".data(using: .utf8)
// change username to userName and you need coding keys!
struct User: Decodable {
    let id: Int
    let name: String
    let username: String
    let email: String
    let address: Address

    struct Address: Codable {
        let street: String
        let suite: String
        let city: String
        let zipcode: String

    }
}
if let user = try? JSONDecoder().decode(User.self, from: jsonData!){
    print(user.name)
    print(user.address.city)
    print(user.email)
    print(user)
}
```

```
Leanne Graham
Gwenborough
Sincere@april.biz
User(id: 1, name: "Leanne Graham", username: "Bret", email: "Sincere@april.biz", address: __lldb_expr_20
556", city: "Gwenborough", zipcode: "92998-3874"))
```

PizzaStore - demo

```
[

    {
            "name": "Chicken Sizzler",
            "ingredients": "Green Chillies, Jalapeños, Red Onions, Chicken",
            "imageName": "chicken_sizzler",
            "thumbnailName": "thumbnail_chicken_sizzler",
            "type": "meat"
        },
    {
            "name": "Beef Sizzler",
            "ingredients": "Green Chillies, Jalapeños, Red Onions, Seasoned Minced Beef",
            "imageName": "beef_sizzler",
            "thumbnailName": "thumbnail_beef_sizzler",
            "type": "meat"
        },
    {
        "name": "Meat Feast",
        "ingredients": "Green Chillies, Jalapeños, Red
        "imageName": "meat_feast",
        "thumbnailName": "thumbnail_meat_feast",
        "type": "meat"
    },

    {
            "name": "Vegetable Supreme",
            "ingredients": "Mushrooms, Mixed Peppers,
            "imageName": "veg_supreme",
            "thumbnailName": "thumbnail_veg_supreme",
            "type": "vegetarian"
        },
        {
            "name": "Veggie Sizzler",
            "ingredients": "Green Chillies, Jalapeños, Mixed Peppers, Red Onions",
            "imageName": "veg_sizzler",
            "thumbnailName": "thumbnail_veg_sizzler",
            "type": "vegetarian"
        }
]
```

```
import Foundation


struct Pizza: Identifiable, Decodable, Encodable{
    var id = UUID()
    var name: String
    var ingredients: String
    var imageName: String
    var thumbnailName: String
    var type: String
}

```

Loading json file – from user defaults or file

```swift
//  Created by GirishALukka on 14/02/2023.
//

import Foundation
class PizzaStore: ObservableObject {
    @Published var pizzas: [Pizza] = []
    //{
//        didSet {
//            savePizzasToUserDefaults()
//        }
    //}

    init() {
        let userDefaults = UserDefaults.standard

        // Try to read the pizzas data from user defaults
        if let pizzasData = userDefaults.data(forKey: "pizzas"),
           let pizzas = try? JSONDecoder().decode([Pizza].self, from: pizzasData) {
            self.pizzas = pizzas
        }
        // If pizzas data is not available in user defaults, load it from the bundled pizzas.json file
        else if let url = Bundle.main.url(forResource: "pizzas", withExtension: "json") {
            do {
                let data = try Data(contentsOf: url)
                let decoder = JSONDecoder()
                self.pizzas = try decoder.decode([Pizza].self, from: data)
                // Save the loaded pizzas data to user defaults
                let pizzasData = try JSONEncoder().encode(self.pizzas)
                userDefaults.set(pizzasData, forKey: "pizzas")
            } catch {
                print("Error decoding pizza data: \(error)")
            }
        }
    }

```

Saving and deleting

```swift
    // Function to save the pizza data to user defaults
    func savePizzasToUserDefaults() {
        let userDefaults = UserDefaults.standard
        do {
            let pizzasData = try JSONEncoder().encode(self.pizzas)
            userDefaults.set(pizzasData, forKey: "pizzas")
        } catch {
            print("Error encoding pizza data: \(error)")
        }
    }

    func deletePizza(_ pizza: Pizza) {
        if let index = pizzas.firstIndex(where: { $0.id == pizza.id }) {
            pizzas.remove(at: index)
            savePizzasToUserDefaults()
        }
    }
}
```

ContentView – tab navigation and filter by type

```swift
import SwiftUI

struct ContentView: View {
    @EnvironmentObject var pizzaStore: PizzaStore
    @State private var isPresentingAddPizzaModal = false
    @State private var newPizza = Pizza(name: "", ingredients: "", imageName: "", thumbnailName: "", type: "")

    var body: some View {

        NavigationStack{
            TabView {
                PizzaList(pizzas: pizzaStore.pizzas)
                    .tabItem {
                        Label("All Pizzas", systemImage: "cart")
                    }

                PizzaList(pizzas: pizzaStore.pizzas.filter { $0.type == "meat" })
                    .tabItem {
                        Label("Meat Pizzas", systemImage: "circle")
                    }

                PizzaList(pizzas: pizzaStore.pizzas.filter { $0.type == "vegetarian" })
                    .tabItem {
                        Label("Vegetarian Pizzas", systemImage: "leaf")
                    }
            }
            .navigationBarItems(trailing:
                                    Button(action: {
                isPresentingAddPizzaModal = true
            }) {
                Image(systemName: "plus")
            }
            )
            .sheet(isPresented: $isPresentingAddPizzaModal) {
                AddPizzaView(pizza: $newPizza, isPresented: $isPresentingAddPizzaModal)
                    .environmentObject(pizzaStore)
            }
        }
    }
}
```

PizzaListView

```swift
//
//  Created by GirishALukka on 14/02/2023.
//

import SwiftUI

struct PizzaList: View {
    let pizzas: [Pizza]

    var body: some View {

        NavigationView{
            List(pizzas) { pizza in
                NavigationLink(destination: PizzaDetailView(pizza: pizza)) {
                    HStack {
                        Image(pizza.thumbnailName)
                            .resizable()
                            .frame(width: 50, height: 50)
                        Text(pizza.name)

                    }
                }
            }
            .navigationTitle("Pizza Menu")
        }
    }
}
```

PizzaListView

```swift
//
//  Created by GirishALukka on 14/02/2023.
//

import SwiftUI

struct PizzaList: View {
    let pizzas: [Pizza]

    var body: some View {

        NavigationView{
            List(pizzas) { pizza in
                NavigationLink(destination: PizzaDetailView(pizza: pizza)) {
                    HStack {
                        Image(pizza.thumbnailName)
                            .resizable()
                            .frame(width: 50, height: 50)
                        Text(pizza.name)

                    }
                }
            }
            .navigationTitle("Pizza Menu")
        }
    }
}
```

AddPizzaView

```swift
import SwiftUI

struct AddPizzaView: View {
    @Environment(\.presentationMode) private var presentationMode
    @EnvironmentObject var pizzaStore: PizzaStore
    @Binding var pizza: Pizza
    @Binding var isPresented: Bool

    var body: some View {
        NavigationView {
            Form {
                Section(header: Text("Name")) {
                    TextField("Pizza Name", text: $pizza.name)
                }
                Section(header: Text("Ingredients")) {
                    TextField("Ingredients (comma separated)", text: $pizza.ingredients)
                }
                Section(header: Text("Image Names")) {
                    TextField("Image Name", text: $pizza.imageName)
                    TextField("Thumbnail Name", text: $pizza.thumbnailName)
                }
                Section(header: Text("Type")) {
                    Picker(selection: $pizza.type, label: Text("Type")) {
                        Text("Meat").tag("meat")
                        Text("Vegetarian").tag("vegetarian")
                    }
                    .pickerStyle(SegmentedPickerStyle())
                }
                Section {
                    Button(action: {
                        pizzaStore.pizzas.append(pizza)
                        presentationMode.wrappedValue.dismiss()
                        pizzaStore.savePizzasToUserDefaults()
     // save each time new one is added as alternative to didSet in class
                    }) {
                        Text("Add Pizza")
                    }
                    .disabled(pizza.name.isEmpty || pizza.ingredients.isEmpty || pizza.imageName.isEmpty || pizza.thumbnailName.isEmpty)
                }
            }
```

AddPizzaView - continued

```
40                      pizzaStore.savePizzasToUserDefaults()
41      // save each time new one is added as alternative to didSet in class
42                  }) {
43                      Text("Add Pizza")
44                  }
45                  .disabled(pizza.name.isEmpty || pizza.ingredients.isEmpty || pizza.imageName.isEmpty || pizza.thumbnailName.isEmpty)
46              }
47          }
48          .navigationBarTitle(Text("Add Pizza"), displayMode: .inline)
49          .autocapitalization(.none)
50          .navigationBarItems(leading: Button(action: {
51              presentationMode.wrappedValue.dismiss()
52          }) {
53              Text("Cancel")
54          })
55      }
56  }
57  }
```

PizzaStoreApp Screens