iOS Dependency Injection (DI)

Workshop outline

- Theory talk (~45 minutes).
- Break.
- Guided app refactor (~ 2 hours).
- Wrap-up (5 minutes).

Goals

- **Understanding** of DI principles and benefits.
- **Experience** adding manual DI to MVC/MVVM.
- **Awareness** of the costs/benefits of DI frameworks.

I want this workshop to change how you write code.

Talk

What is a dependency?

When a class C uses functionality from a type D *to perform its own functions*, then D is called a **dependency** of C.

C is called a **consumer** of D.

Why do we use dependencies?

- To share logic and keep our code **DRY**.
- To model logical abstractions, **minimizing cognitive load**.

Consumer/dependency example

```
class FriendlyTime {
   // ^^^^^^^^^^^^ Consumer
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
        // ^^^^^^^^^^^^^^^^         Dependencies          ^^^^^^
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

Consumer/dependency example

```
class FriendlyTime {
   // ^^^^^^^^^^^^ Consumer
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
        // ^^^^^^^^^^^^^^^^         Dependencies          ^^^^^^
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

Consumer/dependency example

```
class FriendlyTime {
   // ^^^^^^^^^^^^ Consumer
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
        // ^^^^^^^^^^^^^^^^         Dependencies          ^^^^^^
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

Consumer/dependency example

```
class FriendlyTime {
   // ^^^^^^^^^^^^ Consumer
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
        // ^^^^^^^^^^^^^^^^         Dependencies          ^^^^^^
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

iOS consumers

In iOS, **important consumers** include:

- view controllers,
- *view models.*

These classes are the hearts of our apps. Their capabilities include transforming app state into UI state, processing user input, coordinating network requests, and applying business rules. **Testing them is valuable!**

Getting to testing

- **First**: make consumers (unit) testable (*via DI*).
- **Then**: add unit tests.

iOS dependencies

In iOS, **common dependencies** include:

- API clients,
- local storage,
- clocks,
- geocoders,
- user sessions.

iOS consumer/dependency examples

iOS consumer/dependency examples

- A login **view controller** that uses an *API client* to submit user credentials to a backend.

iOS consumer/dependency examples

- A login **view controller** that uses an *API client* to submit user credentials to a backend.
- A choose sandwich **view model** that uses *local storage* to track the last sandwich ordered.

iOS consumer/dependency examples

- A login **view controller** that uses an *API client* to submit user credentials to a backend.
- A choose sandwich **view model** that uses *local storage* to track the last sandwich ordered.
- A choose credit card **view model** that uses a *clock* to determine which cards are expired.

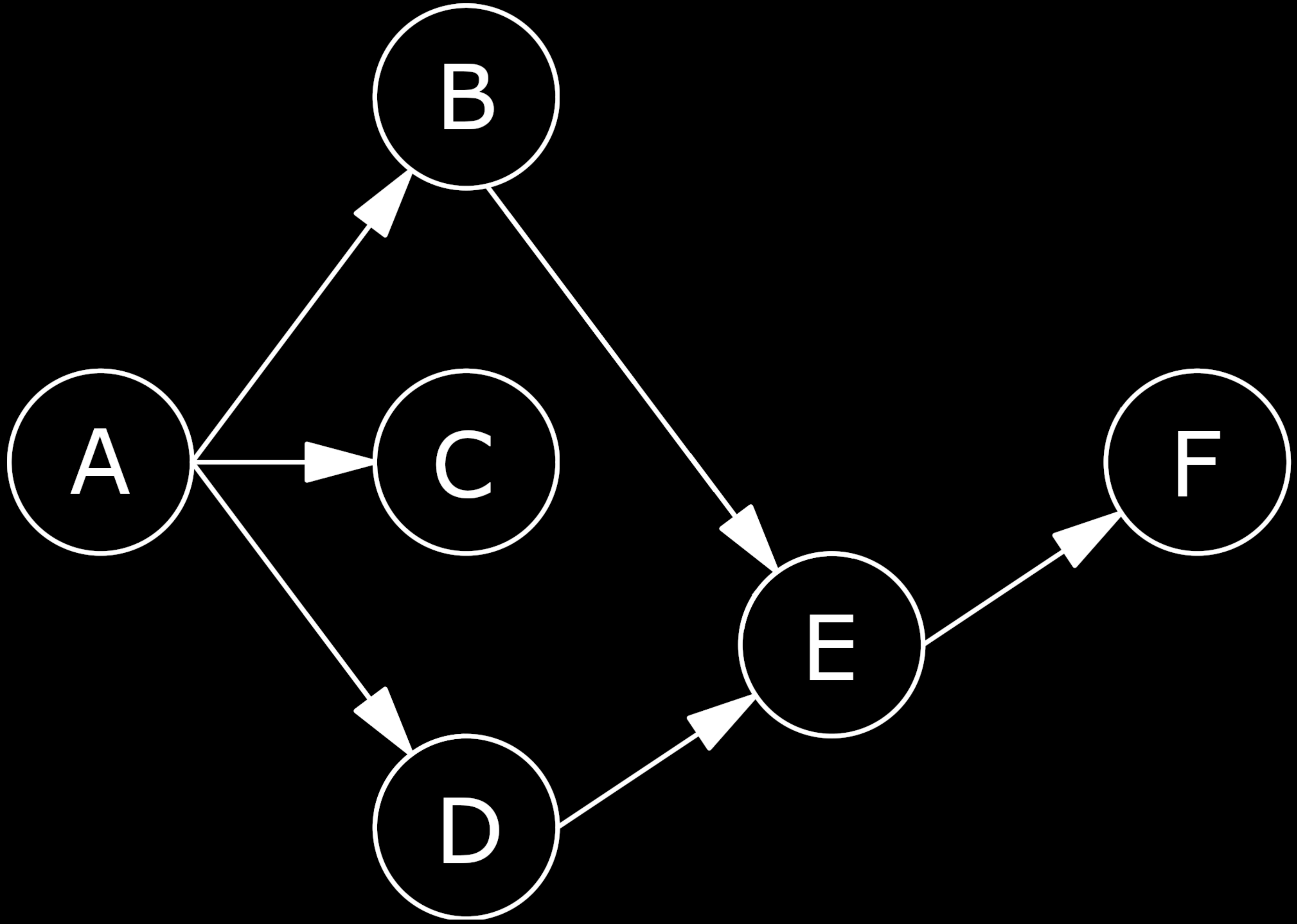
Dependency dependencies

Some classes are **both** consumers and dependencies.

Example: an API client may consume local storage (for caching) **and** be consumed by view models.

We can model all these dependency relationships using a **dependency graph**.

Dependency graph example



Mommy, where do dependencies come from?

Mommy, where do dependencies come from?

- *Consumers* locate their own dependencies (**hard-coded**).

Mommy, where do dependencies come from?

- *Consumers* locate their own dependencies (**hard-coded**).
- *Consumers* ask an external class for their dependencies (**service locator**).

Mommy, where do dependencies come from?

- *Consumers* locate their own dependencies (**hard-coded**).
- *Consumers* ask an external class for their dependencies (**service locator**).
- *An external class* injects a consumer's dependencies via initializers or property mutation (**dependency injection**).

Hard-coded dependencies

```
class FriendlyTime {
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
        // ^^^^^^^^^^^^^^^^    Hard-coded dependencies       ^^^^^^
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

(Be mindful of dependencies added by protocol conformance)

Hard-coded dependencies

```
class FriendlyTime {
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
        // ^^^^^^^^^^^^^^^^    Hard-coded dependencies    ^^^^^^
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

(Be mindful of dependencies added by protocol conformance)

Hard-coding hardships

A consumer with *volatile* dependencies will be **very hard to unit test at all**:

```
func testTimeOfDayMorning() {
  let expected = "Morning"
  let actual = FriendlyTime().timeOfDay()
  // Fails ~70% of the time:
  XCTAssertEqual(expected, actual)
}
```

Hard-coding hardships

A consumer that hard-codes access to *singletons* may have **brittle/slow/lying unit tests** (if state is accidentally shared between tests).

Hard-coding hardships

A consumer's dependencies are **hidden**:

```
// Dependencies on Calendar and Date are invisible:
let friendlyTime = FriendlyTime()
print(friendlyTime.timeOfDay())
```

We can do better

We will *refactor so that*:

- Consumers demand all dependencies via their **initializers**.
- Consumer initializer parameters are all **protocols**.

We can do better

We will *refactor so that*:

- Consumers demand all dependencies via their **initializers**.
- Consumer initializer parameters are all **protocols**.

Outcomes:

- ✅ Production code can supply **real implementations**.
- ✅ Tests can supply **stable fake implementations**.
- ✅ Consumer dependencies are made visible.

Recipe

1. **Create protocols describing *ideal* dependency behaviors.**

Ideal is fuzzy, but desirable properties include:

- names based on ***outcomes***, not implementations,
- ***domain-specific*** names when appropriate,
- a pragmatic balance between ***specificity*** and ***cohesion***.

Recipe

1. Create protocols describing ideal dependency behaviors.
2. **Add protocol instance to consumer initializer.**

Recipe

1. Create protocols describing ideal dependency behaviors.
2. Add protocol instances to consumer initializer.
3. **Use injected instances throughout consumer.**

Recipe

1. Create protocols describing ideal dependency behaviors.
2. Add protocol instances to consumer initializer.
3. Use injected instances throughout consumer.
4. **Create real implementations.**

Recipe

1. Create protocols describing ideal dependency behaviors.
2. Add protocol instances to consumer initializer.
3. Use injected instances throughout consumer.
4. Create real implementations.
5. **Pass real implementations in production.**

Recipe

1. Create protocols describing ideal dependency behaviors.
2. Add protocol instances to consumer initializer.
3. Use injected instances throughout consumer.
4. Create real implementations.
5. Pass real implementations in production.
6. **Pass mock implementations in tests.**

Recap: Before

```swift
class FriendlyTime {
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

1. Create ideal protocols

```
class FriendlyTime {
  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

The `FriendlyTime` class requires a dependency with the ability to provide the **current hour**.

1. Create ideal protocols

```
// Describes the *behavior* our consumer relies on:
protocol IClock {
  var hour: Int { get }
}
```

2. Add initializer parameters

```
class FriendlyTime {
  private let clock: IClock
  init(clock: IClock) { self.clock = clock }

  func timeOfDay() -> String {
    switch Calendar.current.dateComponents([.hour], from: Date()).hour! {
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

3. Use injected instances

```
class FriendlyTime {
  private let clock: IClock
  init(clock: IClock) { self.clock = clock }

  func timeOfDay() -> String {
    switch clock.hour {
    case 6...12:  return "Morning"
    case 13...17: return "Afternoon"
    case 18...21: return "Evening"
    default:      return "Night"
    }
  }
}
```

4. Create real implementations

```
// SystemClock is now one possible supplier of IClock behavior:
class SystemClock: IClock {
  var hour: Int {
    return Calendar.current.dateComponents([.hour], from: Date()).hour!
  }
}
```

5. Pass real implementations in production

Owners of consumers create/locate and inject dependencies:

```
// Initializer injection in production code:
let friendlyTime = FriendlyTime(SystemClock())
print(friendlyTime.timeOfDay())
```

6. Pass mock implementations in tests

```
// Mock clock created for use in tests:
struct StubClock: IClock {
  let hour: Int
}
```

6. Pass mock implementations in tests

```
func testTimeOfDayMorning() {
  let expected = "Morning"
  let stubClock = StubClock(hour: 6)
  let actual = FriendlyTime(clock: stubClock).timeOfDay()
  // Always passes:
  XCTAssertEqual(expected, actual)
}
```

6. Pass mock implementations in tests

```
func testTimeOfDayEvening() {
  let expected = "Evening"
  let stubClock = StubClock(hour: 19)
  let actual = FriendlyTime(clock: stubClock).timeOfDay()
  // Always passes:
  XCTAssertEqual(expected, actual)
}
```

Recipe review

1. Create ideal protocols.
2. Add initializer parameters.
3. Use injected instances.
4. Create real implementations.
5. Pass real implementations in production.
6. Pass mock implementations in tests.

Recipe review

- ✅ Simplest injection technique.
- ✅ Dependency lifetimes controlled using familiar methods.
- ✅ Sufficient for all unit testing needs.
- ✅ Works for fresh code *and* refactors.
- ❌ Repetitive, especially if your dependency graph is deep e.g. `D1(D2(D3(...), ...), ...)`.
- ❌ Insufficient for UI testing.

DI Frameworks

DI frameworks aim to improve on our recipe.

- Dependency graph is described **once**.
- Helper **factory** creates and supplies dependencies.

The details are (much) more complicated, but that's the gist.

Doing DI: Framework injection

- ✅ DRY.
- ✅ Makes dependency graph very explicit.
- ✅ Sufficient for all unit testing needs.
- ✅ Sufficient for all UI testing needs.
- ❌ Increased indirection.
- ❌ Learning curve (for every team member).
- ❌ Longer build times/some performance impact.

I say...

Use a framework if:

- your app needs extensive UI test coverage, or
- your app has a deep dependency graph, or
- your app swaps dependency implementations at runtime.

Otherwise, **prefer manual initializer injection.**

</talk>

Questions?

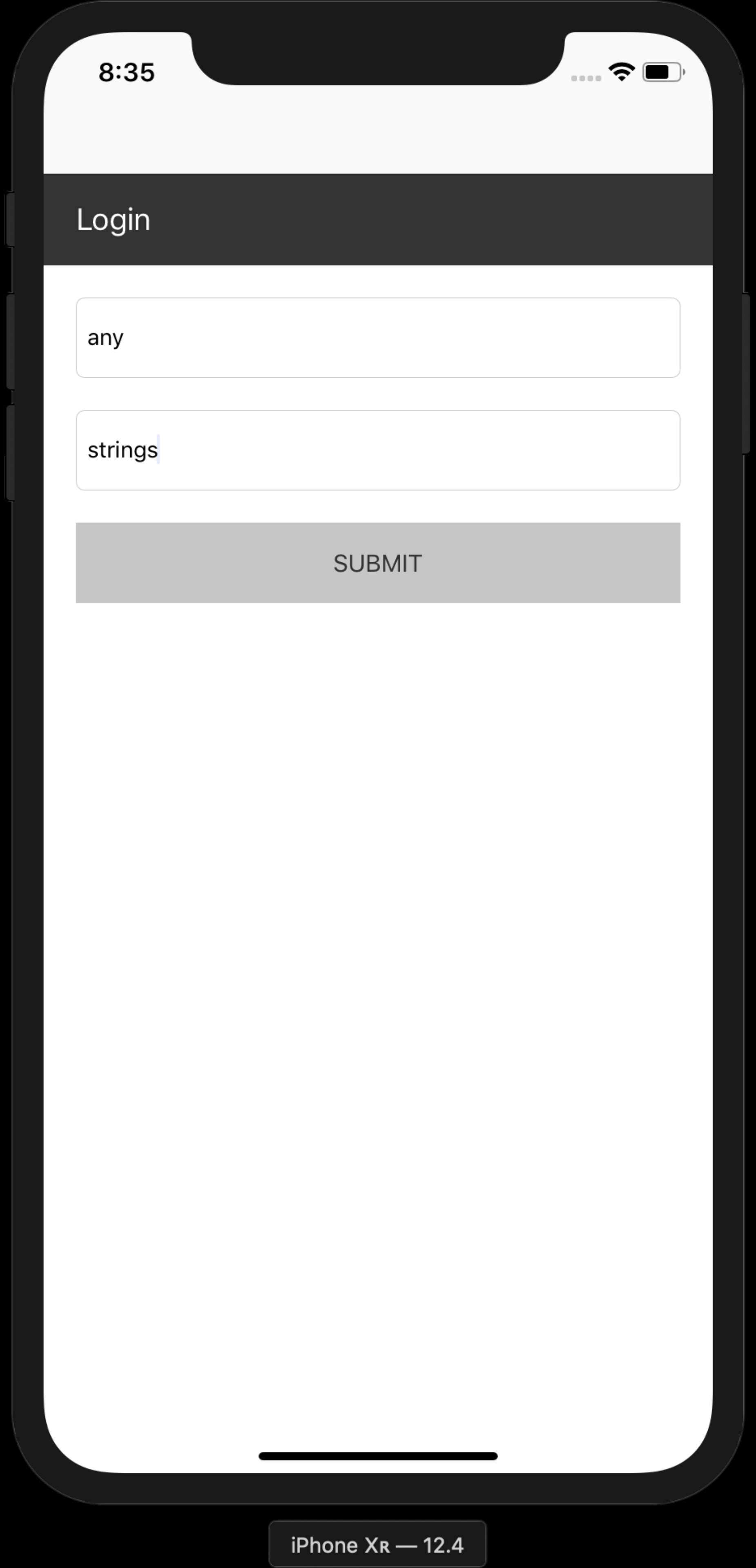
Guided App Refactor

Speedy Subs

Speedy Subs is a small sandwich-ordering app.

Each screen is structured using **MVVM** + delegate.

We will *refactor* each screen *to allow unit testing* via DI.

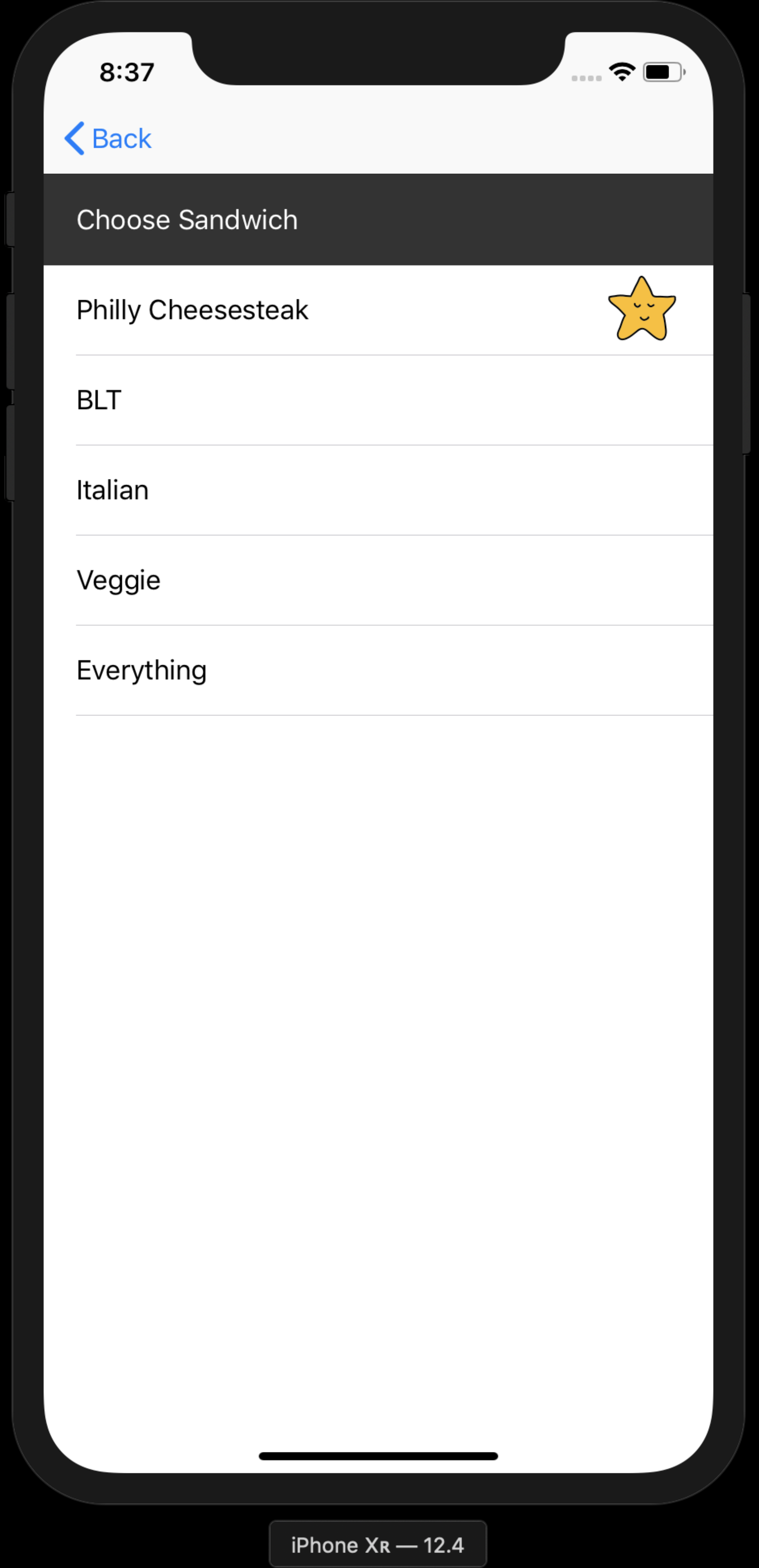


any

strings

Login

- **Username is validated**
- **Password is validated**
- *Login request is made on submit*
- Choose Sandwich screen is launched on success



8:37

Back

Choose Sandwich

Philly Cheesesteak ★

BLT

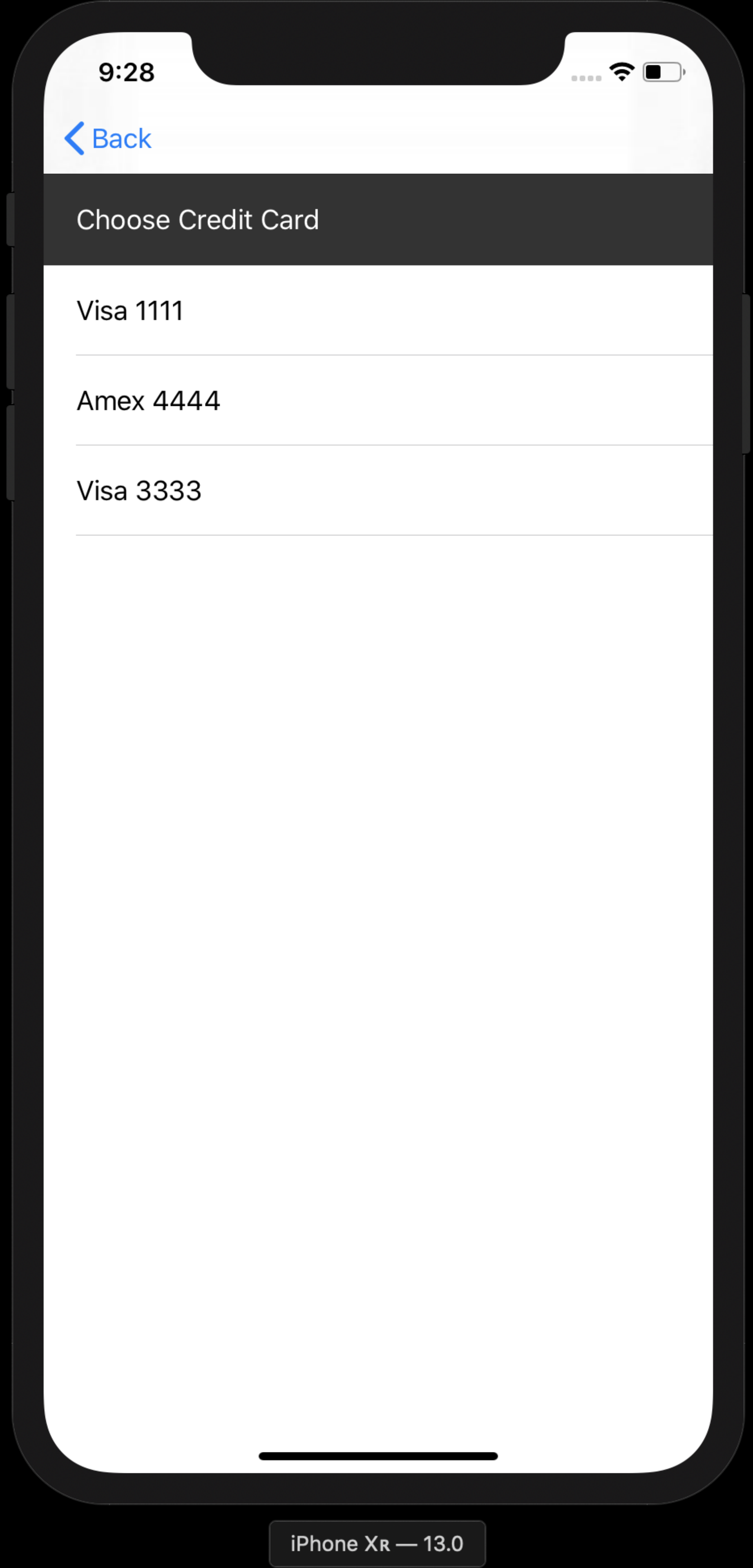
Italian

Veggie

Everything

Choose Sandwich

- *Sandwiches are fetched from network on screen launch*
- **Last-ordered sandwich is listed first**
- **Other sandwiches are listed in order received**
- Choose Credit Card screen is launched on row tap


9:28

Back

Choose Credit Card

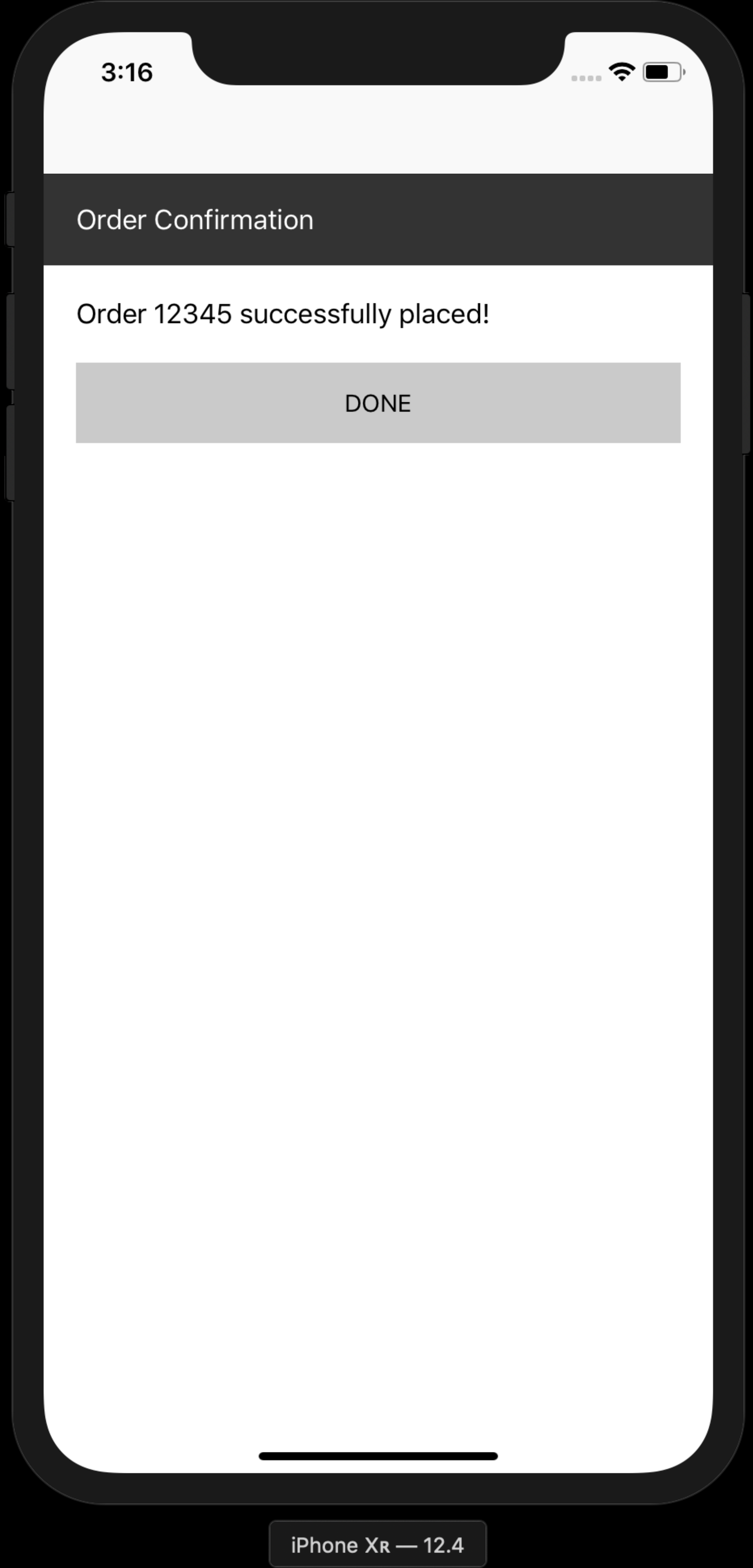
Visa 1111

Amex 4444

Visa 3333

Choose Credit Card

- Credit cards are initially populated from login response
- *Screen implements pull-to-refresh*
- **Only non-expired credit cards are listed**
- *Order is submitted on row tap*
- Confirmation screen is launched on success



3:16

Order Confirmation

Order 12345 successfully placed!

Confirmation

- Done button returns us to the login screen.

Key classes

- `AppDelegate`: entry point & navigation.
- `Session`: holds current customer and order.
- `OrderingApi`: interface to fake backend.

Ready, set, refactor

Wrap-up

DI IRL

- Refactor to MVVM first.
- Follow the recipe!
- Adopt DI incrementally.
- Focus on important/fragile/high-churn areas.

Reflect, Revisit, Repeat, Reinforce

One week from now:

- Re-read slides.
- Refactor `ChooseCardViewModel`.
- (Optional) Write tests for `ChooseCardViewModel`.

Further learning

- (C# book) Dependency Injection Principles, Practices, and Patterns by Steven van Deursen and Mark Seemann
- (Article) Dependency Injection by objc.io
- (Article) Beyond Mock Objects by J. B. Rainsberger