301: App Architecture (Demo)

Quick demo of the app

Let’s take a quick look at the app, so you get some idea of what it’s all about.

Open “1-Starter” project

Open the project from the **1-Starter** folder in Xcode and run it on the simulator.

Run the app

The app is called Bidly and lets you bid on online auctions. Imagine this is some sort of eBay clone.

App: Search for something

Before you can bid on something you first need to search for it.

Go to the Search tab and type something random (it doesn’t matter what). These are all the open auctions. In the terminology of the app, they are the “items”.

This is a typical mobile app that communicates with a server, although the server back-end is entirely faked at the moment. What you see on your screen will be somewhat different from mine, because we all have our own randomized set of data to work with.

App: Item Detail screen

Tap on a item to see the details. Because this is all fake data, the photos don’t make sense; the text is placeholder text. But you get the idea.

An item has a number of bids associated with it; here they are ordered from highest bid to lowest.

App: Star the item

If you want to start watching this item, you tap the star button. From then on, the item appears in the Watch screen (which is the 3rd tab at the bottom).

App: Go to Watch screen

These are all the items you're interested in, usually because you're bidding on them, but you can also watch items without bidding.

App: New Bid screen

To make a new bid you first tap an item and then press the plus button. It communicates with the server and then your bid is made. It appears here in the list.

App: Pull-to-refresh on Activity screen

Bids also appear on the Activity screen. The Activity screen lets you keep an eye on any new bids from competing bidders.

You can pull-to-refresh or just wait a few seconds until the Activity screen polls the server. This is probably the screen you're most interested in as a user because you want to keep track of any bidding activity on the items that you want buy.

So it's a pretty simple app but there's enough going on for us to explore some of these architecture issues.

[2:30]

What is wrong with this app?

Go to Xcode project

Let's have a look at the Xcode project. This app is mostly made up of view controllers. There are a few other source files but they don't really do that much right now.

Slide: 01 - Current Architecture

I have a slide of this that illustrates the current organization of the app.

As you can see, there is a web of dependencies between the view controllers that really shouldn't be there. They are all tangled up.

The reason for this is that each view controller "owns" the particular set of data it needs. Each view controller is a silo, and contains the data for just that task.

These view controllers don't share any data between them, so if something happens in the Activity screen that has an impact on the Watch screen, then they need to access each other's data.

Because of this, each view controller needs to have a direct reference to the other view controllers.

That breaks just about every rule in the book about encapsulation and data hiding, and keeping things decoupled. It leads to duplicate code, confusion about who is responsible for what, and hard to find bugs.

> *Back to Xcode, source code for ActivityViewController*

Let's quickly look at the source code for the ActivityViewController. This view controller does a lot of stuff. Look at that. That's way too much work for a single class. The other view controllers are like this too, so it's a big mess.

I hope you don't write your own apps like this, but from what I've seen on forums and Stack Overflow, this is often how apps do get built. It's view controllers all the way down, and nothing much else, and the view controllers know way too much about each other.

> *Slide: 02 - Improved Architecture*

So the first thing we're going to do is clean up that mess and extract the data model so that the view controllers no longer own any data. Instead, they properly share the data model between them, so that it is no longer scattered throughout the code.

The goal is to end up with an architecture like in the slide, where all the different jobs that the app does are neatly separated.

[4:45]

The domain model

> *Open "2-Demo Starter" projecct*

I want you to close this project and open the project from the **2-Demo Starter** folder. This project already has some of the changes that we need, to save some time.

I'm going to argue that the data model, or domain model as I'll call it from now on, is the most important part of the app.

The “domain” of an app describes the problem that the app solves. It’s not just the data, it also includes all the “domain logic” for the app, or “business logic” as it is sometimes called.

But it *excludes* logic that isn’t directly relevant to the domain, such as formatting dates for display on the screen.

In this case the domain is online auctions. And so the domain model looks like this:

> *Slide: 03 - Domain Model*

The server provides three of these objects – the Item, Bid, and Bidder. An Item is something that’s up for auction, and can have multiple Bids. A Bid is made by a Bidder, who are the users of this system. Pretty simple, right?

Well, there is also the Watchlist. This is not something provided by the server but it belongs only to the mobile app. The Watchlist is simply all the items whose bids you’re keeping an eye on.

This picture right here describes what the app is all about. It doesn’t include anything about the user interface, or the server API, or Core Data, or JSON. It’s just about auctions. The domain for this app is auctions. Anything else is just implementation details.

Think of the domain model as the heart of the app.

Unfortunately, right now, this domain model is completely hidden in the code. Half the stuff, such as the Watchlist, is an array somewhere in a view controller. There is no Watchlist object yet. That’s why all these other view controllers need to have a reference to this WatchViewController, so they can access this array of Item objects. We can do better than that!

What we’re going to do in the next twenty minutes or so is make the domain model explicit, so that it is obvious, and cleanly separated from the rest of the app.

All right? Let’s get going.

[7:00]

Relationship between Bid and Item

Back to Xcode. Look at Bid.swift and Item.swift.

Right now, the Item and Bid classes are simply data containers. They contain the exact same fields as the JSON data we receive from the server.

This makes it a so-called “thin model”. It’s just data, nothing more.

A lot of the domain logic — and by that I mean the algorithms that deal with managing these auctions and bids — is still inside the view controllers. So let’s fatten up the model a little and move as much of that code into these model classes as possible.

Slide: 04 - Item and Bid Relationship

First off, the relationship between an Item and its Bids is one-way only. The Item has an array of Bids, but a Bid doesn’t know who its Item is. If you have a Bid object and you want to find the Item that it belongs to, you have to loop through an array and compare IDs, and so on.

ActivityViewController, networking code

You can see that here in the ActivityViewController. This is the networking code, and this kind of loop is typical.

That’s a lot of messy code that we can pull out of the view controllers, simply by making this a two-way relationship.

1) Connect the Bids to their Items

As you can see here, Bid currently refers to items by ID. This is how the JSON data from the server does it, and that was taken over literally into the model.

But we’re no longer dealing with JSON data here, so it makes more sense to give Bid objects a direct reference to the Item that they belong to.

In **Bid.swift**, remove the `itemID` and `itemName` instance variables.

In their place, add a new instance variable:

```
weak var item: Item!
```

This is a weak variable because a Bid does not own the Item that it belongs to.

In `init(JSON)`, remove the lines that use the old `itemID` and `itemName` variables.

Also in `init(coder)` and `encodeWithCoder()`.

That's it for the Bid class. Now we have to fill in this new `item` variable somewhere, and that happens in the Item class.

In **Item.swift**, there is a helper method `addBid()`. Add the following line to it:

```
bid.item = self
```

This sets up the two-way relationship. After the Bid gets added to the Item, it is given a reference to that Item.

In `init(coder)`, add the following after the call to `super`:

```
// Reconnect the bids with this item object.
for bid in bids {
  bid.item = self
}
```

This code is used to save the list of items to a local datastore, in this case simply using NSCoding and NSKeyedArchiver.

Now new `Bid` objects are always connected to their `Items`.

It makes sense for the JSON data to refer to items by ID – that's often how this works if the service is backed by a relational database – but there's no reason why we can't make that more convenient in our own code.

Just by making this simple change, the model objects are already shaping up to be more of a true domain model.

[10:30]

Taking arrays out of view controllers

As I mentioned, each view controller currently has its own set of data. There is no shared data between them at all. This is why all the view controllers have references to each other.

Slide: 05 - That's Bad, M'Kay

The most important piece of data in this app is the array of items from the WatchViewController. This array contains the auction items that the user is keeping an eye on.

Slide: 06 - ItemDetail to Watch

If the user presses the star button to watch an item, it ends up in this array. When any of the other view controllers need to find an item, they also look into this array, and possibly even modify it.

It's a good idea to turn this array into a model object of its own, so that it is truly shared between those view controllers. We're going to call this new object the Watchlist.

Slide: 07 - Shared Watchlist

This is the improved architecture that we'll end up with. Note how the view controllers don't have any arrows pointing at each other any more. They now only depend on this one shared Watchlist object, not on the other view controllers.

2) Item Detail View Controller

Xcode, Watchlist.swift

I've already added a basic version of Watchlist to the project, so let's have a look at that. It contains a read-only array of Item objects and a few methods that let you add new items, remove items, and so on.

We're going to start with the ItemDetailViewController, because that's where the star button is, and that's how you start watching new items.

Xcode, ItemDetailViewController.swift.

Add a new instance variable to **ItemDetailViewController.swift**:

```
var watchlist: Watchlist!
```

Every view controller that needs to use the Watchlist object gets one of those variables. This technique is called dependency injection.

Slide: 08 - Dependency Injection

I resisted the temptation to make Watchlist a singleton, because singletons have a number of important downsides. In particular, they create a new dependency between the singleton and the object that uses it, while we're trying to *reduce* dependencies.

With dependency injection, the link between the view controller and a particular Watchlist object doesn't exist until you actually run the app.

Show this in the app, from Search screen

When you open the detail screen and you're already watching that item, it should show the star button as selected.

To determine the state for this button, the app needs to check somehow whether the user is currently watching the item. That happens in `viewWillAppear()` in ItemDetailViewController.swift.

Xcode, viewWillAppear()

This is what I mean. This code looks into the WatchViewController's array of items. Again, this is a loop that compares IDs to look for an Item object. That's pretty horrible, so we're going to use the new Watchlist object for that.

Let's throw away all this code and replace it with the single line,

```
watchingItem = watchlist.hasItem(item)
```

That's a lot simpler. This is an example of what I called "domain logic" or "business logic". You're asking a question here, "Is this Item currently being watched?" That's exactly the sort of thing a domain model is supposed to be able to answer.

The same thing goes for `watchToggled()`. This is the action method that gets called when the user taps the star button.

This method violates basically every principle of data hiding! Don't write code like this!

Again there is one of those loops, inside the if-statement. Note here that it's also doing stuff on another view controller, to keep that in sync as well when you stop watching the item.

Replace all that with:

```
watchlist.removeItem(item)
```

And in the else-clause, do:

```
watchlist.addItem(item)
```

That’s makes it much clearer what is going on, right?

Previously, the concept of “the watchlist” was hidden beneath all this cruft that looked at the array in the WatchViewController.

But now that you have a domain model object for it, you can directly express this idea – I’m removing something from the watchlist, I’m adding something to the watchlist – and the code for that is immediately obvious.

This method also uses WatchViewController to sort the list of items, and to save the watchlist to disk. Replace that code with the following:

```
watchlist.sortItems()
watchlist.saveWatchlist()
```

These methods do not exist yet, but you’ll soon add them.

There is even a line that tells this other view controller to reload its table view. You don’t really want to do that.

It’s up to the WatchViewController to notice that the Watchlist object has changed, and if so, reload its own table view. One view controller should not do anything to the views of another view controller. That’s just naughty.

Now you can remove the instance variables that refer to the other view controllers:

```
var activityViewController: ActivityViewController!
var searchViewController: SearchViewController!
var WatchViewController: WatchViewController!
```

These are no longer needed.

To recap, what you’ve done here is change the Item Detail view controller so that it no longer accesses the items array inside WatchViewController, but the new shared Watchlist object. You were able to throw away a lot of code.

You'll do the same for the other view controllers now.

[16:00]

3) Watch View Controller

Xcode: WatchViewController.swift

Let's switch over to the infamous WatchViewController. This screen shows the items that the user is currently keeping track of.

First, we'll get rid of the items array, because that's the whole purpose of this exercise. Remove these view controller references as well.

```
var activityViewController: ActivityViewController!
var searchViewController: SearchViewController!

// This must be public because other view controllers need to access it.

var items = [Item]()
```

Of course, this view controller also needs a reference to the shared Watchlist object.

```
var watchlist: Watchlist!
```

There are now a couple of places in the code that give errors, so let's fix those.

In `prepareForSegue()` we're segueing to the ItemDetailViewController. We just changed the code for that class, so it no longer has any references to these other view controllers. It just needs the Watchlist object.

So remove the lines that pass along the view controller references, and replace it with this:

```
controller.watchlist = watchlist
```

This is dependency injection in action. Because ItemDetailViewController needs to have a reference to a Watchlist, we need to give it that object here, during the segue.

Also, instead of reading from the `items` array, which no longer exists, you now use `watchlist.items`.

```
controller.item = watchlist.items[indexPath.row]
```

Anywhere else that this class tries to use `items` directly, now will use `watchlist.items`. That happens in the table view data source methods:

- `tableView(numberOfRowsInSection)`
- `tableView(cellForRowAtIndexPath)`

There are still some errors in these methods, down here: `loadWatchlist()` and `saveWatchlist()`. This is where the array of items is saved to a local file, so the app can restore this after it quits and starts up again. A bigger app would probably use Core Data for this, but Bidly makes do with a plist file.

The load and save methods still want to use the old items array. The solution is to move these methods to where they belong, inside Watchlist.swift.

Cut the entire `// MARK: Persistence` section out of `WatchViewController` and paste it into **Watchlist.swift**.

Back in **WatchViewController.swift**, in `tableView(commitEditingStyle, forRowAtIndexPath)`, the code still tries to call the old `saveWatchlist()` method. Change this to:

```
watchlist.removeAtIndex(indexPath.row)
watchlist.saveWatchlist()
```

We can also remove `init(coder)`, because it just existed to call `loadWatchlist()`.

If we're not loading the watchlist file here, then what is a good place? I think it makes most sense to load the plist file when the Watchlist object is first created. That happens in AppDelegate when the app starts up, in `didFinishLaunchingWithOptions`.

AppDelegate.swift

It gets created here and then AppDelegate passes it to all the view controllers that need it.

While we're here, we also need to give this Watchlist object to our view controller. Add the following:

```
WatchViewController.watchlist = watchlist
```

Also remove the lines with the errors. And now the Watchlist model object is also shared with this view controller.

Watchlist.swift

We're not quite done yet. In **Watchlist.swift**, call `loadWatchlist()` from the `init()` method, so that the file gets loaded when Watchlist is constructed.

The main principle behind Object-Oriented Programming is that you organize your code into objects that contain both data and functionality. You don't perform operations on those objects, but you ask the objects to perform these operations on themselves.

That's why you moved things that are related to the watchlist, such as loading and saving, into Watchlist.swift. You can also do this with the code that sorts the items.

WatchViewController.swift

If you look at **WatchViewController.swift**, you'll see that it also has a `sortItems()` method. Cut it out of this file and paste it into **Watchlist.swift**.

Slide: 09 - Responsibilities

Now WatchViewController is a lot cleaner and smaller. It is independent of any other view controllers and only uses the Watchlist object to communicate with the rest of the app.

Build and run, and the app should work as before.

Run the app.

[21:00]

4) Observing

I mentioned that view controllers no longer communicate directly, but indirectly through the Watchlist object.

Slide: Observing

If a view controller is interested in something that happens with the watchlist, for example when a new bid is added to an item, then it can observe the Watchlist object somehow and react to any changes, in order to redraw a table view or something.

For example, if another user makes a new bid on one of the items you're watching, and new this bid is received by the app, the table view in the Watch screen should update. This currently doesn't happen if you're already on the Watch screen.

Run app.

So, I built this special option into the Settings tab. If you tap this row, the app will pretend that some other user made a bid on one of the items you're watching. It has a 2-second delay, so you can switch tabs in the mean time.

Let's try that out. After 2 seconds you get a message in the debug pane that a new bid was sent to the server, and then after a couple seconds more, the app receives this new bid from the server.

But nothing changes on the Watch screen. The data changes out from under you but there is nothing to trigger a table view reload with the new data.

It's clear that WatchViewController needs some other way to observe Watchlist, so that it knows when new bids are added.

Watchlist.swift

If you take a look at **Watchlist.swift**, you'll see that it has some code that lets you add so-called observers to the watchlist.

An observer is simply a class that implements the `WatchlistObserver` protocol. This is like giving Watchlist a delegate, except that it can have more than one of these observers.

Now when do we need to notify these observers? A good place is whenever a new bid is added to an item.

Uncomment the following code in `addBid()`:

```
// Notify the observers
for observer in observers {
  observer.watchlist(self, addedBid: bid, toItem: item)
}
```

This simply loops through any registered observers and calls the method from the protocol on them.

WatchViewController.swift

In **WatchViewController.swift**, add `WatchlistObserver` to the class declaration:

```
class WatchViewController: UITableViewController, WatchlistObserver
```

In `viewDidLoad()`, tell the Watchlist we want to observe it, because we want to be notified when new bids are added to any items the user is watching.

```
watchlist.addObserver(self)
```

At the bottom of the file, uncomment the code from the `// MARK: Observing the Data Model` section.

This finds the row in the table, for the item that was updated and reloads that row.

> *Run app*

Let's try adding a new bid again. Keep a close look at [the item]. The number of bidders should go up by one and the highest bid amount should change as well. Did you see that?

What happens is that the ActivityViewController polls the server and receives the new bid. Previously, it would tell the WatchViewController, "Hey there is a new bid, and you should reload your table view."

But now, it only adds this new bid object to the Watchlist. The Watchlist will then notify its observers, including WatchViewController. Then this view controller can decide for itself what to do with this update.

That is a much better way to communicate such updates between view controllers.

This is only one way that you can make your code observe the domain model. Another common approach is KVO, or NSNotificationCenter, or even something like ReactiveCocoa.

There are plenty of choices, but the point is that the view controllers don't talk to each other, only to the model objects.

[26:00]

5) Other domain logic

I treat the domain model as something that I can ask questions of relating to that domain.

Examples of such questions that the current domain model can already answer, are:

- Is this particular item being watched?
- How many bids are there on a particular item?

Examples of some new questions you could add, are:

- What is the difference between the starting bid and the final bid?
- Is someone an aggressive bidder, do they spend a lot of money, how likely are they to win, and so on?
- It could even try to predict how much money you need to spend to win a bid, based on the past behavior of your competitors, and whatever other data you have available.

This is often called the “business logic” of the app. That is the sort of thing that goes into the domain model. I talk about the domain model a lot, but that’s really the core of your app. Everything else, such as the UI, builds on it.

There is still some logic in our view controllers that really belongs to the domain. In this case, the question that gets asked is:

- What is the largest bid amount for a particular item?

You can see that code in WatchViewController, in the `cellForRowAtIndexPath` method. This logic really should go into the domain model, in particular into the Item class.

In **Item.swift**, I already created this method for you, uncomment the code for `highestBidAmount`. This is actually a computed property, and it simply loops through all the bids and finds the one with the highest amount.

In **WatchViewController.swift**, remove this code:

```
var highestBidAmount = 0.0
for bid in item.bids {
  highestBidAmount = max(bid.amount, highestBidAmount)
}
```

Replace the line that sets `cell.highestBidLabel.text` with:

```
if let highestBid = item.highestBid {
  cell.highestBidLabel.text = currencyFormatter.stringFromNumber(highest
Bid.amount)
} else {
  cell.highestBidLabel.text = "-"
}
```

The reason you need to use if-let is that the highestBid property can be nil if there are no bids yet.

Do the same in **SearchViewController.swift**. That has the exact same code, so we're getting rid of some code duplication here as well.

Build and run, and everything should still work OK.

[29:00]

6) That's it!

OK, that concludes the live demo for this talk.

We did a pretty extensive rewrite of the app. The app now has a clear and obvious domain model that is shared by all the view controllers.

We decoupled the domain data and logic from the view controller code, which is a good thing.

The view controllers now only depend on this domain model, not on any of the other view controllers. So we also reduced dependencies.

You are ready to move on to the lab, where you will go a step further and also remove the networking code from the view controllers.

Good luck, and let me know if you get stuck or if you have any other questions!

[30:00]