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".

Bidly 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 (that's the 1st tab). 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 app 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.

What is wrong with this app?

Go to Xcode project

Let's have a look at the Xcode project. As you can see, 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.

Everything happens in the view controllers. All the data is in the view controllers, and so is all the logic.

Because of this, these view controllers all have references to each other, because they need to use each other's data. There is a web of dependencies between them that really shouldn't be there.

For example, the Activity screen receives new bids from the server every couple of seconds. When that happens, it has to find the corresponding item in the array from the Watch view controller.

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 bugs that are very hard to fix.

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. 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 all very focused on the view controllers.

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, but it's inside one well-defined area.

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.

The domain model

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 fundamental problem that the app is supposed to solve. In the case of Bidly the domain is online auctions. And so the domain model looks like this:

Slide: 03 - Domain Model

An Item is something that's up for auction, and it can have multiple Bids. These two objects are provided by the server.

There is also the Watchlist. This is not provided by the server but it belongs only to the mobile app. The Watchlist is simply all the items that you want to keep an eye on (what you saw in the 3rd tab of the app, after you press the star button).

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. This, what you see right here, is the thing that really matters.

The domain model isn't just the data, it also includes all the "domain logic" for the app, or "business logic" as it is sometimes called. But it *excludes* any logic that isn't directly relevant to the domain, such as formatting dates for display on the screen, or performing network requests.

Slide: 04 - Domain Logic

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

For example, I can ask the Watchlist, is this particular item being watched?

Or I can ask an Item, how many bids do you have? Or what the highest bid that you have? Or what is the difference between the final bid amount and the starting bid.

Now, Bidly is very simple app right now, but you can imagine a version of this app that has an analysis feature that looks at all the auctions and tells you how likely you are to win any of them. It could predict how much money you'd need to spend to win a bid, it can look at the past behavior of any other bidders to see how competitive they are, and so on.

All the answers to these questions also go into the domain model. So that includes any code and algorithms that deal with managing these auctions and bids.

Think of the domain model as the heart of the app. Everything else, such as the UI, builds on it. If you get this part right, then you're already halfway there.

Slide: 05 - Domain is Hidden

The problem with Bidly is that it *didn't* get this part right… This domain model is completely hidden in the code. It's not obvious what is part of the domain and what isn't. The Watchlist is an array somewhere in a view controller. There is no Watchlist object yet. And each view controller has its own small piece of the domain logic.

We can do better than that, so what we're going to do in the next twenty minutes or so is is extract everything that is related to the domain from the view controllers, and move as much of that code into these model classes as possible. That will make the domain model obvious and cleanly separated from the rest of the app.

OK, that's the theory for now, so let's put this into practice.

Taking arrays out of view controllers

Run app, search, open detail screen

If you're using this app and you want to start watching an item, you tap the star button here.

What currently happens is that this view controller, ItemDetailViewController, takes this Item object and places it inside an array from this other view controller. That's this one, from the 3rd tab. So you have one view controller talking directly to another view controller. "That's bad, m'kay?"

Go into detail screen for item you're already watching

When you open the detail screen and you're already watching the item, the star button should appear selected and you can now "unwatch" the item. Determining the state of this button is done again by looking into that array of this other view controller.

It's a good idea to turn to take that array out of the view controller and turn it into a model object of its own, so can we can share it between all the different view controllers. And of course that is going to be the Watchlist object I showed you earlier.

Slide: 06 - Shared Watchlist

This is the improved architecture that we'll end up with. Notice 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.

And all the domain model stuff is now in an area of its own; it's cleanly separated from the view controllers.

1) Item Detail View Controller

Open “2-Demo Starter” project

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.

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 our improvements 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. (If you’re curious, the name for this technique is dependency injection.)

Now, what shall we use this Watchlist object for? Remember how I showed you that the star button should be selected or not, depending on whether you’re already watching the item? Determining the state for this button in `viewWillAppear()` in ItemDetailViewController.swift.

Xcode, viewWillAppear()

This code is typical for this app. It uses a for-loop to look into the array of another view controller, to see whether it has the Item object in question. That’s pretty horrible, and these loops appear all over the place, 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 and clean architecture! It physically hurts me to see code like this – even though I'm the one who wrote it.)

Again there is one of those loops, inside the if-statement, and it's very similar to the one we just removed. That's why I said there was a lot of duplicate code.

Note here that it's also doing stuff on another view controller, and that happens because each view controller has its own set of data, so they need to be kept in sync every time you make a change to one of them.

Replace all that with:

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

And in the else-clause, do:

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

Now we're speaking the language of the domain model. That's makes it much clearer what is going on, right?

Previously, the concept of "the watchlist" was hidden beneath all this cruft with the for-loops and looking into someone else's array. That lead to a lot of duplicate code. And not just that – you also couldn't tell at a glance what that code was doing.

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 talked to the WatchViewController to sort the list of items, and to save the array to disk. We're going to tell the Watchlist to do that from now on.

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

These methods do not exist yet, but you'll add them in a moment.

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 – that something's been added to it or removed from it – 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. Now the ItemDetailViewController no longer knows anything about the other view controllers. And that's a good thing.

To recap, what you've done here is change the Item Detail view controller so that it no longer uses the items array inside WatchViewController, but the new shared Watchlist object. As a result, you were able to throw away a lot of code and it made the meaning of your program a lot clearer.

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

2) 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 (the 3rd tab).

The whole purpose of this exercise is to get rid of the items array, so let's do that first.

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

var items = [Item]()
```

Remove these view controller references as well because we don't want this view controller to know anything about the other view controllers:

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

Instead, we'll give this view controller 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 that we just fixed. That class 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. Of course, the ItemDetailViewController needs to have a reference to a Watchlist, we need to give it that object here, during the segue.

```
controller.watchlist = watchlist
```

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 was saved to a local file, so when the user quits the app and then later starts it up again, all his data is still there.

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. We can write `watchlist.items` here because this is a readonly array, it has a private setter.

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.

What that means is that this logic for loading and saving really belongs in Watchlist.swift.

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

Call `loadWatchlist()` from the `init()` method, so that the file gets loaded when Watchlist is constructed.

That begs the question, where does the Watchlist get created anyway? 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, they were for when we still gave all the view controllers references to each other.

And now the Watchlist model object is also shared with this view controller.

WatchViewController.swift

Now we can go back to **WatchViewController.swift** and clean up the remaining errors.

First, we can remove `init(coder)`, because it just existed to call `loadWatchlist()`, which no longer happens in this file.

In `tableView(commitEditingStyle, forRowAtIndexPath)`, the code still tries to call the old `saveWatchlist()` method. This is for swipe-to-delete. Change this to:

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

There's one more error, up there. This view controller also has a `sortItems()` method. That no longer belongs here. Cut it out of this file and paste it into **Watchlist.swift**.

Now everything that has to do with managing the watchlist sits in Watchlist.swift

Slide: 07 - Responsibilities

The view controller is a lot cleaner and smaller. It only does view controller tasks, such as managing the table view and handling the segue to the Detail screen.

All the tasks related to managing the actual domain data now live in Watchlist. So that's

a much better separation of responsibilities.

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

Run the app.

3) Observing

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

Slide: 08 - Observing

If a view controller is interested in something that happens with the watchlist, then it can observe the Watchlist object somehow and react to any changes.

For example, another user can make a new bid on one of the items you're watching. So the Activity screen polls the server and a few seconds later it receives the new bid.

Previously, it would directly tell the WatchViewController, "Hey there is a new bid, and you should reload your table view." But now it no longer does that and it just updates the Watchlist object.

What should happen is that the Watch screen is notified of this new bid so that it can put the new data into its table view. But this currently doesn't happen, because the Watch view controller is not observing the Watchlist object yet for such events.

Run app. Show item on watch screen.

I'll show you this in the app. Here we have an item that we're watching and it has X bidders and the highest amount is Y. Whenever another user makes a new bid, these two things should automatically change.

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 short delay, so you can switch tabs in the mean time.)

So I'm going to click that. 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. What should have happened is that these

labels got updated. If we switch to the Activity screen, then the bid is there, and if we switch back then it's also updated here.

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. Right now this has just this one method, `watchlist-addedBid-toItem`.

This is like giving Watchlist a delegate, except that it can have more than one of these observers.

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.

```
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?

Now it's no longer the ActivityViewController that tells the Watch screen to reload its

table view. That message now comes from the Watchlist, when it is notifying its observers about this event. 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. You can also use 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.

4) Other domain logic

At this point, we're almost done. We've moved all the domain data into objects of their own, but 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 that calculates the highest bid amount really should go into the domain model, in particular into the Item class.

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

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

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, and returns it. This returns an optional, because the Item may not have any bids yet.

Back in **WatchViewController.swift**, 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.

So the code in `cellForRowAtIndexPath` no longer has the domain logic; it just puts text into the labels.

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.

So that's another example of code that is found in a view controller but really shouldn't be.

5) That's it!

OK, that concludes the live demo for this talk.

We did a pretty extensive rewrite of the app. So what we did was, 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!