301: App Architecture (Intro)

Slide with title

This talk is about app architecture. So what do we mean by that, what is architecture?

Simply put, it's the organization of your code: what objects do you have, and how do these objects talk to each other.

Slide: popping objects with ? and arrows between them

Let's say you want to make a certain type of app. How do you decide which objects you will need? And once you have these objects, how will you make them communicate?

In this talk, I'm going to show you some architecture ideas in practice, using a basic demo app.

Slide: app

The app is called Bidly and it lets you bid on online auctions, a little like eBay. Unfortunately the app suffers from bad architecture. So in the next hour or so we'll improve the app, step by step.

Slide: table of contents

Here's what we'll cover:

1. First off, I'll go over some of the basic ideas about architecture and how to write good code, to give you an idea of the main principles.
2. The biggest issue with the Bidly app is that the domain model — or data model — is hidden inside the view controllers. We'll fix that in the live demo.
3. A typical architecture problem is that the view controllers do way too much. This is also known as Massive View Controller. In the lab you'll take the networking code out of the view controllers and put it into its own set of classes.
4. And finally, I have a challenge for you where you'll pull everything even further apart, so that each piece of functionality lives in its own object.

Slide: clean architecture diagram

OK, time for some theory.

Why should you bother to invest in a clean architecture for your app?

Picture this – and I'm sure you've been here before: Your code is perfect. You have a design that's neat and clean, and everything makes sense. The source code is beautiful and it should really be framed and hung up in an art gallery somewhere.

Slide: slightly worse architecture diagram

A little while later, your designer or your client asks you to add in a new feature. It doesn't quite fit with the current design, but you can still work it in there with a few simple workarounds.

It would be better if you could revise the design at this point, but you got to ship this thing, right? There's no time for such luxuries.

Slide: bad architecture diagram

Then you add another feature, and another, and so on until the code is a big mess, hanging together by hacks and duct tape, and your original design is nowhere to be found.

This is usually the version that ships – even though you hate it – and you really want to throw away the whole thing and rewrite it from scratch for the next version.

A good architecture can prevent this. It can reduce the impact of changes and the accumulation of what we call "technical debt".

Plus it makes your code easier to understand and reason about, which is a good thing for someone like me, as I'm really not as clever as I look. It helps if the code is clean.

This talk is about finding a better structure for your code that can grow with your app.

Slide: decoupling

The main tool for achieving well-architected code is decoupling.

You want to reduce dependencies between the different parts of your app as much as possible, so that when one part needs to change the other parts remain unaffected.

This is what we call "separating the concerns". Your data model code is independent of the UI; the user interface code is independent of the network API; and so on.

Any changes are then localized to just their own components; if you need to change the

UI, you only have to change the UI code — none of the other code is affected because it's all cleanly separated.

This also makes it much easier to write unit tests, because you can test each part of the app in isolation.

Slide: separation of responsibilities

So how do you do this? Ideally, each object has only a single purpose. This is also known as the "single responsibility principle". As soon as an object starts doing too much, it may be better to split it up into multiple objects.

Decoupling and separating responsibilities is in fact what the main architectural patterns in iOS and OS X are about: that's why we have model-view-controller, delegates, and so on.

The less your objects know about each other, the better.

Slide: MVC

A quick word about MVC. Model-View-Controller is the one architectural pattern everyone knows about, but it gets too much attention, in my opinion.

MVC is only a very small idea: you keep your data — or the model — and your view separate through an intermediary object, the controller. That's all there is to it.

But MVC is not the whole story. There are many other responsibilities in your app that do not fall under M, V, or C.

Here are some of the things a typical app does:

Slide: list of responsibilities

Some of this obviously goes into the M, V or C. But where does networking go, for example? Is it part of the model, the view, or the controller? I would say none of the above.

Networking logic should probably go into a completely new class. But then how do you use that class from the different parts of the app that need to do networking?

That's the question that architecture tries to answer.

In this talk I'm going to give you some suggestions for ways to structure your code in

such a way that it becomes easier to understand, easier to maintain, easier to change, and – hopefully – has fewer bugs.

Don’t take this as dogma. The purpose of this talk is just to give you some new insights, but you can apply them in many different ways. Every project is different, but at least you can let these principles guide you.

OK, let’s get started with the live coding.

Slide: part 2 title