Fast Mobile Discovery Workshops

Patterns and approaches to radically accelerate mobile app build time

# Introduction

## Why?

Most of our customers are on mobile. Intuit Mint/Turbo, Simple, Wealthfront and other competitors are growing their market share with sophisticated mobile apps. We must grow our ability to rapidly experiment, discover, and build on mobile platforms.

## Vision

Our objective is to make our mobile app code fun, fast to work in, as well as easy and simple to understand. We should be able to build highly capable mobile apps with many screens and API connections in just a few days for customer discovery.

Mobile app development should be all about rate of delivery, but we should still be able to have confidence in the stability and architecture of our code.

## Approach

1. None of this code goes to Production. ***Ever***. Not even copied and pasted! If we do build something useful for delivery, we would literally have to type it all in again.
2. Many patterns and approaches taught here are not approved for delivery in production code. Or even recommended for production/delivery. Any patterns or approaches for delivery must be reviewed and approved as a team outside of this workshop.
3. NO FEAR! Let’s try new things and have fun with it in a safe space.

|  |  |
| --- | --- |
| **IS** | **IS NOT** |
| Speed as the primary concern  Mindset  Approach  Discovery  Patterns  Techniques  Playing with “Parkinson’s Law”  “Ceramics class” quantity over quality | Production quality  Delivery  Accepted patterns  Forced thinking  Perfect solutions  Recommended solutions |

# Getting Started

Assets and source code

You’ll want to pull down the sample assets and code at <https://github.com/lampo/mobile-discovery-csharp>

### Diff folders on your file system

Using a file system folder diff between different workshop folders can be extremely helpful. This is often a much better way to see differences than using GitHUB.

* Windows users: <https://winmerge.org/?lang=en> (Ask Kevin if you need help)
* Mac users: diff folders using Xcode's FileMerge utility: <https://www.macworld.com/article/1049584/cmpfldr.html>

Turn off identical files for the best delta view.

Also, cleaning out the bin and obj folders when you diff across Xamarin is a huge help. The “Clean Bin” extension for Visual Studio is excellent for this!

### Work with friends!

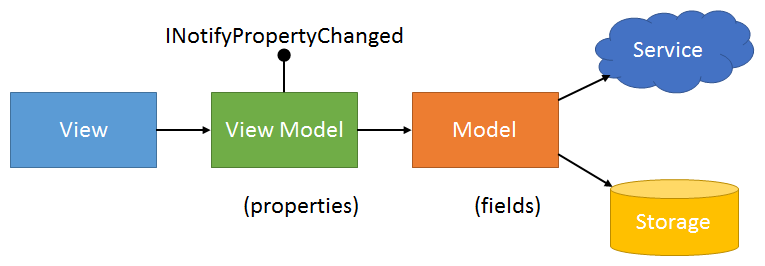
Feel free to pair up or work in a group! Just make sure you are getting LOTS of practice typing this in, building code, and you are getting to deep fluency. Don’t let the group do this work for you!

### Code coverage tools

* Visual Studio: Use the “Fine Code Coverage” extension in VS
* Rider: <https://www.jetbrains.com/help/rider/Cover_Unit_Tests_(Basic_Scenario).html>

### Architectural overview

We’ll be using Model-View-ViewModel architecture for our app. We’ll dive in on this concept in detail as we go through our workshop together, but here is a high-level overview.



# Workshop 1: Placeholder screens

Let’s explore how styling can be fun and quick in a safe place!

For this workshop, we will be building out “placeholder screens” to support the UI in EveryDollar’s onboarding flow.

“Placeholder screens” are very fast build outs to explore page styling. They should present all mobile controls in interactive, styled pages you can navigate through. The quality of the code is not a concern at this point. The objective of placeholder screens is determining style feasibility and exploring how to build the app UI.

With practice, you can build fairly complex, interactive, mobile UX workflows in 1 hour or less.

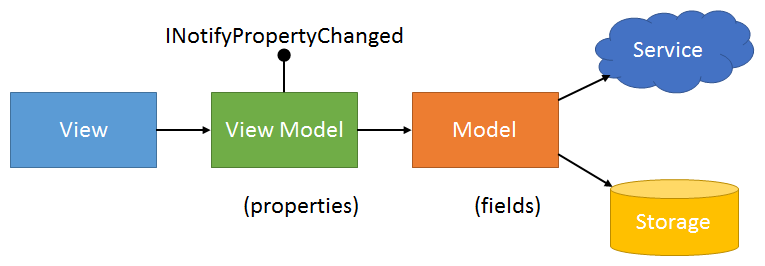
You can run the onboarding flow on EveryDollar by deleting all budgets for an account and restarting the mobile app. Screenshots of this flow are also available.

Pull down and look through the screenshots in the [mobile discovery repo](https://github.com/lampo/mobile-discovery-csharp) Assets folder at <https://github.com/lampo/mobile-discovery-csharp/tree/master/Assets/OnboardingScreens>. You’ll quickly notice that several the screens are very similar:

* The Goals and Status pages both use the same layout for check box titled icons
* A repeating menu page showing the section the user is on and completed
* Each budget group has a similar page for budget item entry: income, housing, transportation, food, personal, giving, debt
* A section intro and section completed page

To study these screens, we will undertake a styling exercise, building several the key screens as individual styling templates for further study.

Architecturally, we will only be working only in the view layer of our project:



### Key concepts in this workshop

* Building fast in a new app: don’t force yourself to work in a big, complex production app
* Use hot reload to see changes in real time
* “Placeholder” screens: rapidly created screens with color, styles, buttons, controls, entries, etc. that you can navigate through, interact with. However, they are do not connect to data, APIs, storage, or models
* Speed > Quality: It is fine if the code is repeated and not pretty. We’ll have time to revisit.
* “Ceramics Class”: doing this repeatedly will increase speed and quality.
* Not using MVVM…yet: in this phase, feel free to throw some temporary code into a page to populate a collection with mock data. No models or view models required!

### Assignment: Xam Visual Challenge

* Read and review <https://devblogs.microsoft.com/xamarin/visual-challenge-conquered/>
* Pull down and look through a few of the example projects at <https://github.com/davidortinau/VisualChallenge/pulls>. The PayPal, AirBNB, and LinkedIn are good starting points to see others building “placeholder screens”
* Give yourself 1 hour to build out the UI-only placeholder screen from any non-Ramsey app you like!
* Repeat to build at least 2 Xamarin Challenge Apps in 2 hours!

### Assignment: E$ Placeholder screens

* Setup a fast discovery project
  + Create a new Xamarin Forms app
  + Add the drawables at <https://github.com/lampo/mobile-discovery-csharp/tree/master/Assets/AndroidAssets/drawable> to the Android project’s Resources/drawables folder
  + Copy both <https://github.com/lampo/mobile-discovery-csharp/blob/master/Assets/OnboardingResourceDictionary.xaml> and <https://github.com/lampo/mobile-discovery-csharp/blob/master/Assets/OnboardingResourceDictionary.xaml.cs> to your main Xamarin project folder, and add the two files to get color resources
    - Review the XAML. It is small and very helpful to have resource files for color and styles commonly used in the app.
  + Update App.Xaml to add the resource dictionary above:

<?xml version="1.0" encoding="utf-8" ?>

<Application

x:Class="Onboarding.App"

xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary Source="OnboardingResourceDictionary.xaml" />

</ResourceDictionary>

</Application.Resources>

</Application>

* + Also, bring in some helpful converters to the Xamarin project. Copy in the Converters folder from the repo at <https://github.com/lampo/mobile-discovery-csharp/tree/master/Assets/Converters>
  + Get the app running on Android device or emulator with hot reload enabled (devices are usually faster!)
  + Right click on the Shared Project (Onboarding) and select Manage Nuget Packages. Download the following package: PropertyChanged.Fody.
* Get the screenshots showing at exactly the size of the device/emulator right next to each other
* In XAML, quickly build out the following screens with styling only using Hot Reload:
  + Intro (first screen)
  + Goals
  + Menu
  + Income
* Feel free to explore data models in the code behind of the pages to mock collections
* Use NavigationPage to quickly connect pages together
* Do not worry if the XAML code gets long here. We will study view abstractions in the next section, and this can be great way to indict areas of long code for reusable pattern study.
* You can see completed placeholder examples at <https://github.com/lampo/mobile-discovery-csharp/tree/master/1-Styling/1-PlaceholderScreens/Onboarding>

# Workshop 2: Reusable views

Now that we have some basic “placeholder” screens up and running in a test app, we will explore ways to optimize our UI styling work.

Once more, review all the screenshots in the [mobile discovery repo](https://github.com/lampo/mobile-discovery-csharp) Assets folder at <https://github.com/lampo/mobile-discovery-csharp/tree/master/Assets/OnboardingScreens>. In the E$ onboarding flow, the pages have high similarity:

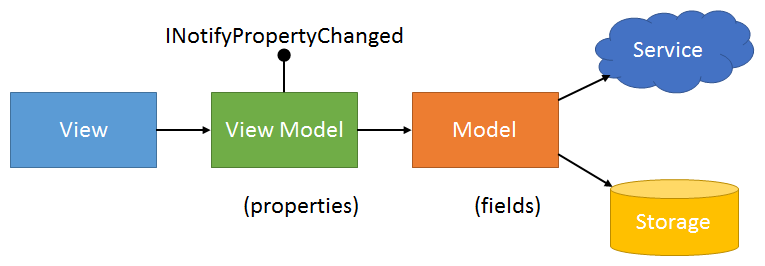
* All pages have a continue button (or similar) with the same shape and size at the bottom
* All pages have a [X] close button in the upper left
* Many pages have a budget summary widget showing at the top

This speaks to the likelihood of a useful pattern or abstraction being able to accelerate many of our pages.

In addition, pages for budget groups are highly similar. They follow an intro, budget group + item entry, and outro pattern for each major section. We will explore patterns for budget group and budget item.

Finally, the goals and status page are nearly identical. A view for the titled checkbox icons on each page will be very helpful here.

Architecturally, like the last workshop, we will be working only in the view layer of our project:



### Key concepts in this workshop

* With dozens of screens, building each page repeatedly would be very painful. Let’s look for ways to create reusable views!
* Taking time to thoughtfully consider reusable views can be a HUGE time accelerator later, so it is great to consider in discovery
* Be willing to try more than one approach. You will likely have to build out different reusable patterns several times to get this right. It took several times in this project!
* Study the framework deeply to know its tricks, secrets, and corners that will help/hinder reusable views

### Assignment: reusable one button content view

* Study the amazing, incredibly powerful, and poorly documented ContentProperty in Xamarin Forms
  + <https://stackoverflow.com/questions/38225716/how-to-set-my-custom-classs-member-views-property-from-xaml-xamarin-forms>
  + <https://docs.microsoft.com/en-us/dotnet/api/xamarin.forms.contentpropertyattribute?view=xamarin-forms>
  + <https://docs.microsoft.com/en-us/dotnet/api/xamarin.forms.contentview?view=xamarin-forms>
* Setup a new project on your own, separate from this workshop, and explore inserting different dynamic content into a view using a [ContentProperty] attribute.
* Make sure you have a strong understanding of this concept before you proceed – it is a key accelerator to Xamarin Forms.
* Load up your code from the placeholder screens assignment, or use the reference code at <https://github.com/lampo/mobile-discovery-csharp/tree/master/1-Styling/1-PlaceholderScreens/Onboarding>
* Create a new Xamarin ContentView in the main Xamarin project’s Views folder called “OneButtonContentView”.
* Give “OneButtonContentView” a ContentProperty attribute and matching property that allows insertion of dynamic XAML content.
* Experiment and try out different patterns for getting a continue button on the bottom and a close button on the top of any content pushed into OneButtonContentView
* Get all placeholder screens to use the OneButtonContentView.
* Reference <https://github.com/lampo/mobile-discovery-csharp/tree/master/1-Styling/2-ReusableViews/Onboarding> if you get stuck or need ideas.

### Assignment: reusable titled icons in goals page

* Using your code from the assignment above, play with breaking out the goals icons into a new view.
* This is the code in the ItemTemplate in your list of goals on the GoalsPage
* If you get stuck or need a reference, see the TitledIconView and GoalsPage in <https://github.com/lampo/mobile-discovery-csharp/tree/master/1-Styling/2-ReusableViews/Onboarding>

### Assignment: reusable budget group and item views

* Using your code from the assignment above, try breaking out budget group and budget item views for the income page.
* If you get stuck or need a reference, see <https://github.com/lampo/mobile-discovery-csharp/tree/master/1-Styling/2-ReusableViews/Onboarding>

### Assignment: reusable UI review

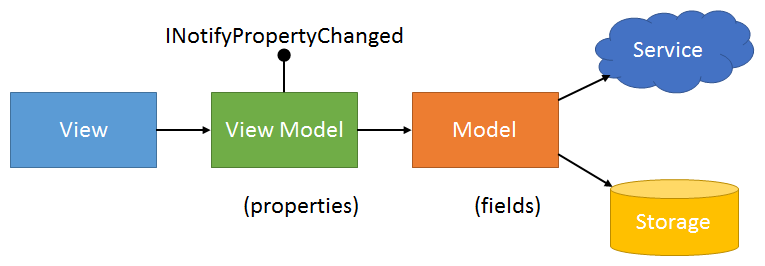
* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/1-Styling/1-PlaceholderScreens/Onboarding> and the completed reusable views project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/1-Styling/2-ReusableViews/Onboarding>
* Pay special attention to the reduction in XAML code as you diff each page. The reusable views have already reduced the page XAMLs by hundreds of lines of code. Imagine the savings across 20+ pages!

# Workshop 3: Modeling

How much time should we be spending in design during discovery? Often quite a bit!

Given the large number of screens, layouts, user state responses, decision trees, API sync, and persistent data needs of the E$ onboarding flow we are targeting in our workshop, we should be spending a LOT of time modelling.

For this section, we will be focusing heavily on the Model area of our app in the diagram below:



### Key concepts in this workshop

* Deep focus on building models
* Models are built specifically to serve the customer experience in the app
* Models are NOT built to serve the API. API considerations are not a first-order concern at this point.
* Models are NOT built to serve the Xamarin Forms framework. Xamarin Forms integration is not a first-order concern at this point.
* View models will work themselves out later. Always start with models.
* Using a helper library to keep our property change simpler (Fody.PropertyChanged)
* Reaching [100% test coverage](#_Code_coverage_tools) for all models without interfaces

### Assignment: research M-V-VM concepts

Spend some time with giving these architecture background documents your time and focus:

* What is M-V-VM? <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel>
* There is “stateful” VM and “stateless” VM. For this workshop, we will be using “stateless”.
  + We are using stateless VMs: <http://www.xamlpatterns.com/stateless-view-models>
  + We are NOT using stateful VMs: <http://www.xamlpatterns.com/stateful-view-model>
* Where should state live when building with “stateless” view models? In views or models! <https://medium.com/livefront/stop-putting-state-in-your-view-models-aa27d0a68b39>
* Study PropertyChanged.Fody attributes, specifically DependsOn, AlsoNotifyFor, and DoNotNotify at <https://github.com/Fody/PropertyChanged/wiki/Attributes>
* Extra credit: What is a Presenter? <https://martinfowler.com/eaaDev/PresentationModel.html>

### Assignment: More M-V-VM concepts

* If you have not already done so, take time, read, and “walk” through the actions in [M-V-VM Sample Project](https://lampogroup.sharepoint.com/:w:/r/sites/RamseyPlus/Shared%20Documents/Global/Engineering/Training%20%26%20Growth/TrainingMaterials/M-V-VM%20project%20study.docx?d=w1b0726eb72044a5a9dddabd4c600203d&csf=1&web=1&e=AnwEIG).
* Spend some time here. These are important concepts for this workshop and client architectures in general.

### Assignment: Add model projects and test projects to the solutions

* Create a .Net standard Models project in your Xamarin solutions. For example, in the Onboarding reference project, these are called Onboarding.Models.
* Add a corresponding XUnit test project for Models.
* When you get done, the solution and project structure should match that in <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/1-BasicModels/Onboarding>

### Assignment: titled icon model

* Don’t code yet – just think. Put pen to paper before you code and consider what model could represent each icon in Goals and Status. Consider the title, the icon, and the selection state.
* Think through how the classes would layout, what they would store, how changes would be communicated in and out, etc.
* Go test-centric! Build out basic titled icon model in the test project focusing on behaviors. In this case, the core behavior is selection
* Implement propertychanged on your model and reference the PropertyChanged.Fody NuGet on your model project
* Use concrete models for testing – no interfaces
* Get to 100% test coverage on your model using [these tools](#_Code_coverage_tools)
* Compare your resultant model to the one in <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/1-BasicModels/Onboarding>

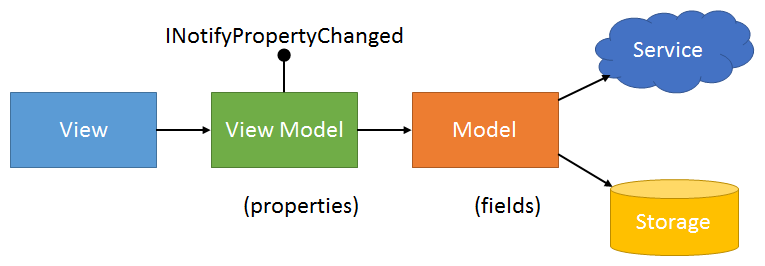
### Assignment: budget group and budget item basic model

* Don’t code yet – just think. Put pen to paper before you code and consider what model could represent a very basic budget group and budget item collection. Consider the names, amounts, but ignore styling/presentation aspects for now.
* Think through how the classes would layout, what they would store, how changes would be communicated in and out, etc.
* Consider also a budget item type as you put pen to paper.
* Go test-centric! Build out basic budget item and budget group model in the test project focusing on behaviors. In this case, the core behavior is selection
* Budget groups are static in the onboarding experience, so you’ll only need to implement propertychanged + Fody on BudgetItem
* Use concrete models for testing – no interfaces
* Get to 100% test coverage on your model using [these tools](#_Code_coverage_tools)
* Compare your resultant model to the one in <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/1-BasicModels/Onboarding>

# Workshop 4: Basic Models + Basic ViewModels

Let’s keep diving in on models that serve our customer experience!

We’ll also start to consider some basic ViewModels as part of this workshop.



### Key concepts in this workshop

* Stateless ViewModels
* ViewModels that are largely “model-centric” and are completely unaware of underlying views
* Using a helper library to keep our ViewModels stateless (Fody.PropertyChanged)
* Reaching [100% test coverage](#_Code_coverage_tools) for all models and view models without interfaces
* Deep focus on building models
* Models are built specifically to serve the customer experience in the app
* Models are NOT built to serve the API. We’ll take on some very prelim API considerations, but limit the impact on our models.
* Models + ViewModels are NOT built to serve the Xamarin Forms framework. Xamarin Forms integration is still not a first-order concern at this point, but we will take a har look at styling.

### Assignment: research M-V-VM concepts

* You may want to revisit [this assignment](#_Assignment:_research_M-V-VM) as you work this section, even if you’ve completed it already.

### Assignment: add more to budget group model

* Don’t code yet – just think. Put pen to paper before you code and consider additional budget group aspects to serve the UI experience. What color theme should each budget group have? Icon? Header text? Special behaviors. There are a lot! Go through the screen shots many times to make sure you have note missed any.
* Think through how the classes would layout, what they would store, how changes would be communicated in and out, etc.
* Try adding the UI properties directly to budget group model. Pretty big, right? And not related to API budget group syncing, right?
* Try moving the UI properties in the budget group model to a new container, a BudgetGroupPresentation. You can see a reference for how this can be done in the Models project within <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/2-BasicModelsII/Onboarding>
* Consider a budget group total amount that is summed from dynamic budget items. Go test-centric! build out the tests for the model and notification first, then implement in Models.
* Get to 100% test coverage on your model using [these tools](#_Code_coverage_tools)
* Once you finish, or if you get stuck, review the budget group model and tests at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/2-BasicModelsII/Onboarding> and see how collection changed and property changes for items in the collection can be handled

### Assignment: View Model projects and test projects

* Create a .Net standard ViewModels project in your Xamarin solutions. For example, in the Onboarding reference project, these are called Onboarding.ViewModels.
* Add a corresponding XUnit test project for ViewModels.
* When you get done, the structure should match that in <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/1-BasicModels/Onboarding>

### Assignment: budget item view model

* Add PropertyChanged.Fody to the ViewModel project.
* You’ll want some handy extensions. Steal them from:
  + <https://github.com/lampo/mobile-discovery-csharp/blob/master/5-MoreViewInt/3-CompletionActions/Onboarding.Models/Extensions.cs>
* Plan and build out a budget item view model. Force it to be stateless, no interfaces, and try going test-centric.
* It should be very simple getter and setters, with Property Changed passing through from the model. Consider this last part carefully!
* Get to 100% test coverage on your changes using [these tools](#_Code_coverage_tools)
* Once you finish, or if you get stuck, review the finished view model and tests at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/2-BasicModelsII/Onboarding>

### Assignment: budget group view model

* Plan and build out a budget group view model. Force it to be stateless, no interfaces, and try going test-centric. Try having it unify presentation specifics along with the data specific (name, amounts). That is the “presentation” role of the view model.
* Consider carefully how to have its listeners dynamically add and remove view models for budget items when they are added/removed from the model. This is an important part to stateless view models you’ll want to invest time and thought into!
* Get to 100% test coverage on your changes using [these tools](#_Code_coverage_tools)
* Once you finish, or if you get stuck, review the finished view model and tests at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/2-BasicModelsII/Onboarding>

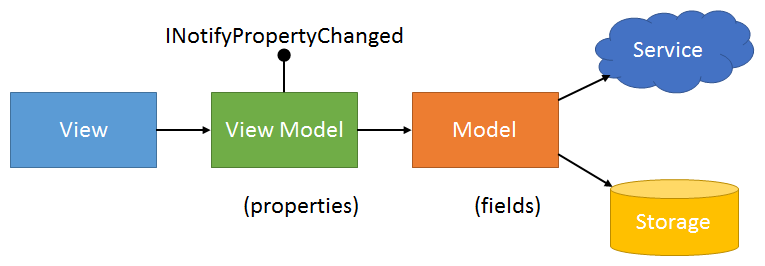
### Assignment: review so far

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/1-BasicModels/Onboarding> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/2-BasicModelsII/Onboarding>

# Workshop 5: More Models + ViewModels

Even more models and viewmodels are needed to serve our customer experience!

We’ll finish out models, add view models, and do our first view connections.



### Key concepts in this workshop

* Stateless ViewModels
* Models are built specifically to serve the customer experience in the app
* ViewModels that are largely “model-centric” and are completely unaware of underlying views
* Retaining [100% test coverage](#_Code_coverage_tools) for all models and view models without interfaces

### Assignment: research M-V-VM concepts

* You may want to revisit [this assignment](#_Assignment:_research_M-V-VM) as you work this section, even if you’ve completed it already.

### Assignment: budget group presentation decoupling

* If you are like me, you may have built the budget group with BudgetGroupPresentation inside it.
* They are better off decoupled. It has been removed and the viewmodel updated to use the two separate classes in the project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding>. Implement this separation of concerns in your code if you have not already.
* Stay at 100% test coverage in models and view models!

### Assignment: budget model

* A budget model to contain all budget groups is needed. Map out on paper how this would look, and build one using test-centric principles and no interfaces.
* A BudgetState enumerator will be useful for over, under, or on-budget experiences, so plan this, too.
* One other consideration is the “Basic Expenses” section, which contains Housing, Transportation, Food, and Personal. Consider options to solve this and experiment.
* Another consideration is an overall income remaining in the budget can be calculated and kept up to date. Have some fun trying things out to solve this.
* A final consideration is building this budget for testing. There are a lot of groups and budget items! Consider doing this with a programmatic builder. Feel free to steal the reference builder in the Models/Builders folder at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding/Onboarding.Models/Builders>
* Stay at 100% test coverage in models and view models!
* A reference for this model and tests can be found at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding>

### Assignment: titled icon view model

* Build out a titled icon view model to present the backing model built in prior workshops
* Go test-centric, building in tests
* Stay at 100% test coverage in models and view models!
* A reference for this with tests can be found at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding>

### Assignment: onboarding profile model

* A “highest level” model with the budget, goals, and status would be helpful across the app. And would allow us to quickly serialize to save/resume if we chose to.
* We will build this and call it our “onboarding profile”
* To make this easier, a TitledIconList class was added as well in the reference project.
* And, to make this “more easier”, a builder is available in the Models/Builders folder at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding/Onboarding.Models/Builders>
* A reference for this with tests can be found at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding>
* Stay at 100% test coverage in models and view models!

### Assignment: viewmodels for titled icon, onboarding profile, budget

* View models for these stay light, simple, and easy using the stateless pattern
* Build your own, staying test-centric
* Get to 100% test coverage on your changes using [these tools](#_Code_coverage_tools)
* Explore the reference set in <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding>

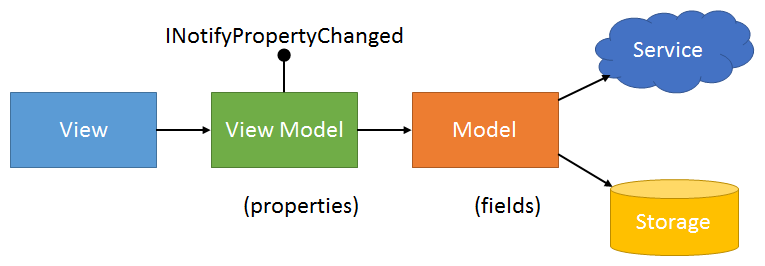
### Assignment: review all models and view models

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/2-BasicModelsII/Onboarding> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding>
* Can these simple, light view models built in a very “model-ish” way represent everything our views need? Let’s go find out!

# Workshop 6: first model-viewmodel-view connection

Alright, we spent a ton of time modelling. Built a lot of view models. Now how does all this connect in app?

Let’s get a screen up and running with the view models we have!



### Key concepts in this workshop

* Scrappy, basic dependency injection
* Starting to use view models and models
* No changes to view models required to accommodate multiple views and pages!

### Assignment: add Microsoft Dependency Injection

* We are going to use Microsoft’s dependency injection service.
* We will be putting many app-wide singletons and transients into Micosoft’s DI system.
* <KEVIN RANT>
  + Microsoft calls these “ServiceCollection” and “ServiceProvider”.
  + They should be named “DependencyCollection” and “DependencyProvider”.
  + They can and often do contain app-wide singleton image collections, sound libraries, config data, models, viewmodels, etc. that are not “services”.
  + So, although called “Services” in many of our projects, in this project, we will use “Dependencies”
* </KEVIN RANT>
* Add the Microsoft.Extensions.DependencyInjection nuget to the onboarding project
* Create a static class called “Dependencies” in main Xamarin Forms project
* Use the content for the class from <https://github.com/lampo/mobile-discovery-csharp/blob/master/3-ViewIntegration/1-Goals/Onboarding/Onboarding/Dependencies.cs>
  + IMPORTANT: this is built for discovery. It is scrappy. It is NOT for production delivery. More on dependencies and service spin up in a future workshop – but not this one 😊
* Update App.Xaml.cs to init the dependencies.

### Assignment: goals page

* Delete all the hacky backing model code from the GoalsPage code behind.
* Setup goals page to have an onboarding profile view model instance passed in to the constructor
* Set the binding content to the onboarding profile view model that was passed in
* From the income page, pass in the onboarding profile view model from Dependencies.
* The XAML should be 90% there or more – try it out.
* Get the button disable working from the view model
* If you get stuck or need help, reference <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/1-Goals>

### Assignment: status page

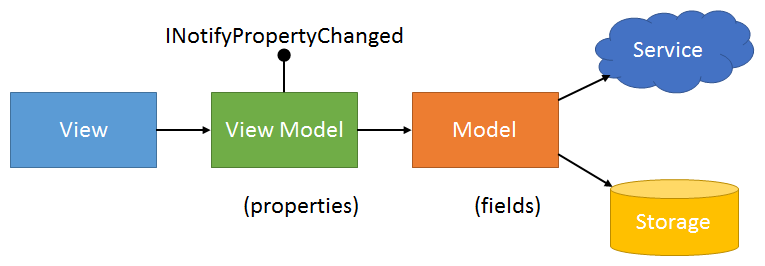
* Copy the goals page and make a status page
* Update the binding and code behind to use Status in the view model
* Make goals page call status page on continue button click
* A little copy and pastey, but reuse levels are high, so we are moving in the correct direction
* If you get stuck or need help, reference <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/1-Goals>

### Assignment: review all changes

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/2-Modeling/3-MoreModels/Onboarding> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/1-Goals>

# Workshop 7: income model-viewmodel-view connection

Let’s get the budget group stack running in a view!



### Key concepts in this workshop

* No changes to view models required to accommodate multiple views and pages!

### Assignment: income page

* Our first chance to use a budget group and budget item model+VM!
* Like we did with goals, drop into the income code behind
* Forcibly remove all those fake placeholder models!
* Have it take a budget group view model in its constructor
* You’ll need to add a continue button click action.
* THAT IS IT! If you get stuck or need help, see <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/2-Income>

### Assignment: WAIT WHAT THAT’S IT?

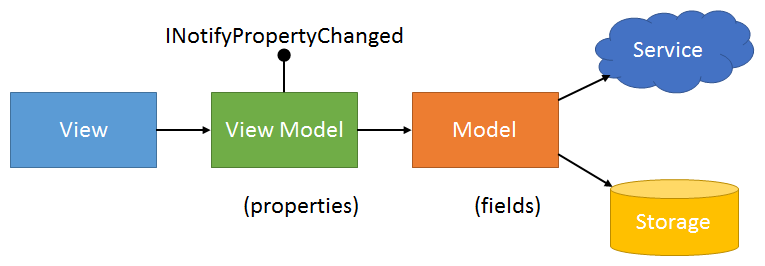
* Review the page and view updates in this workshop and the last
* Notice that no view model or model changes were needed, and how light/easy the page+view connection is to the stateless view models.
* Let’s keep going!!!

### Assignment: review all changes

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/1-Goals> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/2-Income>

# Workshop 8: a bunch of views and pages from one viewmodel

Let’s get nutty and see how many scrumptious views can run from our budget group view model!



### Key concepts in this workshop

* No changes to view models required to accommodate multiple views and pages!

### Assignment: a few android effects and controls

* We’ll grab some Android controls to tint images, change app status bar color, etc.
* Go to the AndroidAssets folder at <https://github.com/lampo/mobile-discovery-csharp/tree/master/Assets/AndroidAssets>
* Copy over the MainActivity.cs, the Controls, and Effects folder to the Android project
* You’ll also need controls and effect in the main Xamarin project. You can grab those two folders from here: <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/Onboarding/Onboarding>
* Review them for fun. Or insomnia.

### Assignment: budget item view improved entries

* Use the Controls for currency and text entry in the BudgetItemView.
* Reference the finished view in <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/Onboarding/Onboarding>

### Assignment: reusable section intro page

* So, we got a lotta section intro pages. Reusable view time!
* Build your own view or steal the SectionIntroView from <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/Onboarding/Onboarding/Views>
* Note this is using the TintImage effect to color the image on each page for reuse.

### Assignment: income intro page

* Copy the IncomePage and rename it to IncomeIntroPage
* Replace the page content with:

<ContentPage.Content>

<views:OneButtonContentView x:Name="contentView" ContinueButtonColor="{Binding ColorKey, Converter={converters:ColorKeyConverter}}">

<views:OneButtonContentView.MainContent>

<views:SectionIntroView />

</views:OneButtonContentView.MainContent>

</views:OneButtonContentView>

</ContentPage.Content>

* Code behind is very basic as well:

using Onboarding.ViewModels;

using Xamarin.Forms;

using Xamarin.Forms.Xaml;

namespace Onboarding.Pages

{

[XamlCompilation(XamlCompilationOptions.Compile)]

public partial class IncomeIntroPage : ContentPage

{

public IncomeIntroPage(BudgetGroupViewModel budgetGroup)

{

InitializeComponent();

BindingContext = budgetGroup;

contentView.ContinueButtonAction = async () => await Navigation.PushAsync(new IncomePage(budgetGroup));

}

}

}

* Have menu page navigate to this income intro page
* DONE! If you get stuck or need help, see <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome>

### Assignment: test reuse on more pages

* Build at least one more section intro page and one more budget group page. Should be just a copy and paste of the exiting income pages, changing navigation and making sure the correct view model is passed in.
* You can see a BasicExpensesIntroPage and TransportationPage in <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome>
* Already feeling the power of reusable viewmodel and views

### Assignment: flexible one button content view

* To make one button content view handle dynamic styling, it needs updates!
* Updates to OneButtonContentView and a complete project using this and all techniques up to this point are at <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/Onboarding/Onboarding>

### Assignment: review all changes

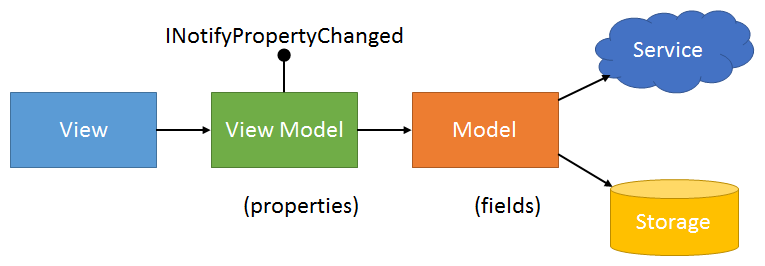
* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/2-Income> and the assignments from this workshop in [https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/](https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/Onboarding/Onboarding)

# Workshop 9: Cheating your way through API integration

Enough view integration for now. Let’s dive in on APIs!

We will really want to abstract our app and models from the API. We’ll start with that interface and work on the “service” side initially.

That means we will be in “Services” in the diagram below:



### Key concepts in this workshop

* Postman is your friend! Use it to walk, study, and hack APIs
* Consider API interface contracts carefully
* Use facades and restructuring to cheat even more, making API integration easier
* Fast programmatic net tests radically outperform mobile app builds. Cheat by using them!
* Cheat the API using autogen code from Postman
* Build remote/API calls in isolation from the rest of the mobile app
* Use facades and restructuring to cheat even more, making API integration easier
* Fast programmatic net tests radically outperform mobile app builds. Cheat by using them!

### Assignment: Postman API walking/hacking

* Even the best made APIs have documentation gaps, behavioral quirks, and response payload variation that can be tricky to connect to and to debug in client code.
* So don’t! Instead, do that work in Postman whenever possible.
* Don’t just trust API docs or current implementations. Make the calls yourself in Postman! Even if you have a reference Postman collection, it is often highly valuable to carefully walk and study an APIs calls and responses to get deep understanding.
* Channel your inner hacker! Using hyperactive (<https://github.com/lampo/everydollar-hyperactive>) and the “F12” debug Network tab in Chrome for tracing, build a new Postman workspace and collection that does the following:
  + signs into E$ API
  + gets all budgets
  + gets a specific budget
  + creates a budget
  + sets a budget item amount
  + creates a new budget item
  + deletes a budget
* If you get stuck, ask an “old EveryDollar engineer” like Steven J, Jeremy S, or Kevin H. for help
* Spend a good deal of time reviewing the responses and the formats that come back from these calls
* A complete collection and environment set is available at <https://github.com/lampo/mobile-discovery-csharp/tree/master/Assets/Postman> for you to compare your results to.

### Assignment: API interface contract design

* Start with the code from your last assignment: <https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/Onboarding/Onboarding>
* Now that we’ve had a chance to deeply study the E$ API and its responses, let’s think through an interface contract for the service that will connect this data into our app models
* We will call this service “Remote Budget”. This service will need a clear, mockable interface for our Models to use and test against.
* Let’s think through the calls that this Remote Budget service will make.
* What would be the smallest, simplest set of calls that a client would want to make to the remote budget API? To support onboarding, we will only need the following API calls:
  + Get Budgets
  + Get Current Budgets
  + Create New Budget
  + Create Budget Item
  + Update Budget Item
* For testing and just for fun, let’s also add these calls
  + Delete Budget
  + Delete Budget Item
* That’s it – that’s the skeleton of our interface contract!
* Let’s add a new .Net Standard project called “Onboarding.RemoteBudget” and add an interface file called “IRemoteBudgetCalls”
* Add all the calls above to the IRemoteBudgetCalls.
  + Consider carefully what arguments you would want to send and receive from each call.
  + It’s fine if it does not compile at this point, we are just modelling the interface contract.
  + This will be much easier with an empty shell “Budget” class inside this project to use as an argument.
  + Make all the calls asynchronous (they all return Task or Task<object>).
* Check your finished result against the example in <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/1-RemoteBudget>

### Assignment: API facing models made easier with cheating (API Restructuring)

* To make working with the API easier, think out a set of models just to contain API information.
* A Budget, BudgetGroup, BudgetItem, and BudgetItemType would be the minimal set here
* However, this is MUCH harder to build out with a few simple classes due \_embedded tags, the URN encoded identifiers, and the USD currency subvalue.
  + In a brand new test solution and project, build out a set of models that can serialize from the E$ API JSON “as-is”.
  + DO NOT MAKE API CALLS JUST YET!
    - This is best done in a unit test project that uses NewtonSoft to deserialize JSON responses you copy in manually from Postman
  + Try to populate ID on all of these deserialized classes
  + Try to get amounts populated as well on budget items.
  + Notice how much work the \_embedded tag , URN id encoding, and amounts brings. Notice the class swell and complexity.
* Let’s force the API to work in a way that HELPS client code rather than adds complexity: API restructuring!
* Add the “Cleaner.cs” file from the Remote Budget project in <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/1-RemoteBudget> to your test solution with the \_embedded models above.
* Use Cleaner.CleanJson() on the E$ API the JSON responses you copied in manually from Postman in your unit test code and Deserialize the cleaned JSON.
* Look at the response JOSN after cleaning using breakpoints, and, if in Visual Studio, the JSON visualizer. IDs! No embedded! Currency amounts that are easy to read!
* Note that the models can be simply and completely reduced to only a very readable Budget, BudgetGroup, BudgetItem, and BudgetItemType.
* Compare your final API models to those in the Remote Budget project in <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/1-RemoteBudget>
* Bring in your final models to the main Onboarding solution.

### Assignment: Postman code stealing and net tests

* Let’s cheat our way to successful, rapid API integration in code using Postman and fast programmatic testing
* Fast programmatic testing for API integration is MUCH faster and MUCH easier to debug than API testing in the mobile app. Try to do as little API work in the slow mobile app as possible!
* Accelerate your work by adding a XUnit testing project to your onboarding solution called “Onboarding.RemoteBudget.TestNet”. As we will run slow unstable network tests, we give the the “TestNet” appendix for clarity.
* Create an empty RemoteBudgetCalls class in the Remote Budget project that implements the IRemoteBudgetCalls interface from above. All calls should throw a not implemented exception for now.
* Create a corresponding test file in the TestNet project calls RemoteBudgetCallsTestNet.cs
* Write a super simple “GetBudgets” API call test in this file. It should look something like this:

[Fact]

public async Task GetBudgets\_ExpectSuccess()

{

var remoteBudgetCalls = new RemoteBudgetCalls();

var budgets = await remoteBudgetCalls.GetBudgets();

budgets.Should().NotBeEmpty();

}

* O.K., you ready to cheat like crazy? Good. Go to Postman and run the “Get Budgets” API call there successfully.
* Click the “Code” text in the upper right of the call in Postman (under the SAVE button).
* Select “C# - RestSharp”
* Copy!
* Paste into your RemoteBudgetCalls:GetBudgets() method!
* Reflect on this for a moment. All the REST setup, all the headers, all the payloads, etc. on every API call we make can autogen from Postman!
* Debug the test with a debug breakpoint on the Console.WriteLine(response) line, making sure you get valid content back
  + If not, sign in on Postman and copy in a new code set with a new JWT
* This can save you a TON of time versus trying to tune content type, parameters, etc. in slow code.

### Assignment – Deserialize the Get Budgets response

* Okay, so the test still fails and we are not done.
* First, go async. Change the client.Execute to be client.ExecuteAsync.
* Second, add a quick status code check after the ExecuteAsync

// check status code

if (response.StatusCode != HttpStatusCode.OK)

{

throw new Exception($"Unexpected response from get budgets call: {response.StatusCode} - {response.Content}");

}

* Finally, after the status code check, add some Cleaner help and deserialization to build out populated API models from the API response:

// check status code

if (response.StatusCode != HttpStatusCode.OK)

{

throw new Exception($"Unexpected response from get budgets call: {response.StatusCode} - {response.Content}");

}

try

{

var json = Cleaner.CleanJson(response.Content);

var jsonObject = JObject.Parse(json);

var budgets = JsonConvert.DeserializeObject<List<Budget>>(jsonObject["budgets"].ToString());

return budgets;

}

catch (Exception e)

{

throw new Exception($"Unable to parse response from Get Budgets: {response.Content}", e);

}

* Get the net test unit test to pass!
  + You may have to sign in and copy in updated JWT tokens from Postman if this takes a while. And it usually does!
  + Make sure you have at least one budget. Make one in Postman or hyperactive if needed.

### Assignment: sign in using AccessControlManager and RestSharp clients

* Add AccessControlManager nuget to the project. It’s a LAMPO nuget, ask friends if you need help
* Set up RemoteBudgetCalls to take in an access control manager and rest client at time of construction like this:

public class RemoteBudgetCalls : IRemoteBudgetCalls

{

readonly IAccessControlManager accessControlManager;

readonly IRestClient restClient;

public RemoteBudgetCalls(IAccessControlManager accessControlManager, IRestClient restClient)

{

this.accessControlManager = accessControlManager ?? throw new ArgumentNullException(nameof(accessControlManager));

this.restClient = restClient ?? throw new ArgumentNullException(nameof(restClient));

}

…

}

* Steal, I mean, leverage the testing helpers in RemoteTesting.cs in the Remote Budget project in <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/1-RemoteBudget>
* Setup your test class with a constructor that runs before each test like this:

public class RemoteBudgetCallsTestNet

{

readonly IAccessControlManager accessControlManager;

readonly IRestClient restClient;

public RemoteBudgetCallsTestNet()

{

// Sign in for every test

accessControlManager = RemoteTesting.SignInForTesting();

restClient = new RestClient($"https://api.everydollar.com/");

}

…

}

* Update your test to use the access control and rest client like this:

[Fact]

public async Task GetBudgets\_ExpectSuccess()

{

var remoteBudgetCalls = new RemoteBudgetCalls(accessControlManager, restClient);

var budgets = await remoteBudgetCalls.GetBudgets();

budgets.Should().NotBeEmpty();

}

* Get the net test unit test to pass!
  + Make sure you have at least one budget. Make one in Postman or hyperactive if needed.

### Assignment – Get all calls working

* Okay, by now you see the pattern we will follow to implement all the API calls.
  + Build a basic net test for the call
  + Use postman to autogen code
  + Paste the code into the method and get it to run in a debug test, tracing to make sure we got a valid response.
  + Add a status code check
  + Use the Cleaner to clean the response JSON
  + Deserialize into an object if appropriate and return it
* NOTE: It can be very helpful to implement delete budget early and use it to clear all budgets in the constructor before each test.
* Compare your completed implementation to the reference at <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/1-RemoteBudget>

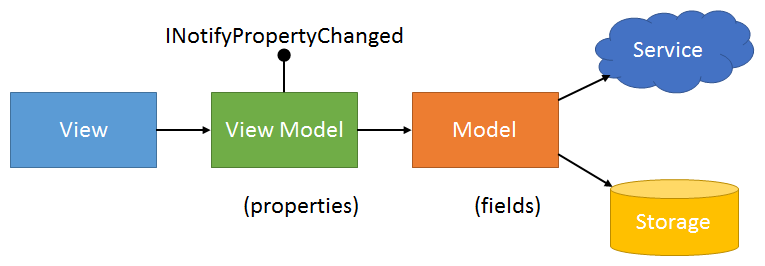
### Assignment: review all changes

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at [https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/](https://github.com/lampo/mobile-discovery-csharp/tree/master/3-ViewIntegration/3-MoreIncome/Onboarding/Onboarding) and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/1-RemoteBudget>

# Workshop 10: Syncing service data to app models

Great, we now have a complete set of API calls built rapidly for discovery. Let’s dive in on connecting Remote Budget service and data to our app models!

Also, let’s work hard to “protect” our app models. We’ll spend a lot of time building and thinking through this buffer.



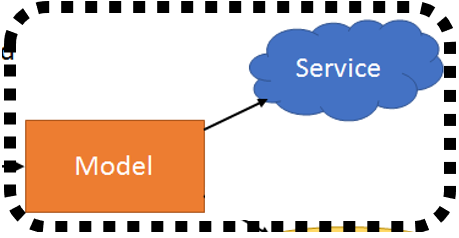
### Key concepts in this workshop

### Shape the API to the customer experience – NOT THE OTHER WAY AROUND!

* Minimize contagion and spread of remote interfaces into app models
* “Slow” integration tests are almost always an order of magnitude faster than manually building and testing in a mobile app
* Be familiar with options for concretes in testing, such as NSubstitute’s Substitute.ForPartsOf
* 100% test coverage held for all models

### Assignment: Build a sandbag bunker to repeal the zombie contagion aka “Sync”

* Integrations in software are tricky. Remote integrations even more so.
* The biasing of app models and architectures toward remote APIs and services is often in dangerous opposition to optimal user experience.
* So, keep the service and remote API at “arm’s length”. Use interfaces and code isolation to protect app models. Winning here means our models are changed little to none.
* In our code base, let’s create a new code module called “Sync” to handle connecting to our service and remote APIs:



Sync

IRemoteBudgetCalls

* Sync will be a small, isolated, protected code set that will have the single responsibity of connecting app models to remote apis
* Create a new folder Onboarding.Models project called “Sync”. It will be the anti-zombie bunker!

### Assignment: TDD remote budget sync and budget group sync

* Let’s start with connecting to remote budget. To stay decoupled and make mock testing easier, Sync will ONLY be allowed to access and reference the IRemoteBudgetCalls interface.
* For now, all IRemoteBudgetCalls will be mocked.
* To make this easier, consider a test helper like this to simulate a partial remote budget coming back from IRemoteBudgetCalls:

public static RemoteBudget.Budget CreateTestRemoteBudget()

{

return new RemoteBudget.Budget("test", new List<RemoteBudgetGroup>()

{

new RemoteBudgetGroup("Income", "Income", new List<RemoteBudgetItem>()

{

new RemoteBudgetItem("test", RemoteBudgetItemType.Income, "Paycheck 1", 0),

new RemoteBudgetItem("test", RemoteBudgetItemType.Income, "Paycheck 2", 0),

}),

new RemoteBudgetGroup("Housing", "Housing", new List<RemoteBudgetItem>(){

new RemoteBudgetItem("test", RemoteBudgetItemType.Expense, "Mortgage/Rent", 0),

new RemoteBudgetItem("test", RemoteBudgetItemType.Expense, "Water", 0),

new RemoteBudgetItem("test", RemoteBudgetItemType.Expense, "Natural Gas", 0),

new RemoteBudgetItem("test", RemoteBudgetItemType.Expense, "Electricity", 0),

new RemoteBudgetItem("test", RemoteBudgetItemType.Expense, "Cable", 0),

new RemoteBudgetItem("test", RemoteBudgetItemType.Expense, "Trash", 0),

}),

new RemoteBudgetGroup("Transportation", "Transportation", null),

new RemoteBudgetGroup("Food", "Food", null),

new RemoteBudgetGroup("Personal", "Personal", null),

new RemoteBudgetGroup("Giving", "Giving", null),

new RemoteBudgetGroup("Debt", "Debt", null),

});

}

* Look up and research NSubstitute’s Substitute.ForPartsOf before the next step
* Stay TDD and, using test-first development, build out a BudgetSync class with a single method that takes a local app budget model and sends it into remote budget.
  + It is likely easier to do this if you create a BudgetGroupSync class, and focus on the high-level BudgetSync just mapping all the onboarding groups out to the budget group sync class.
  + HINT: to make mocking BudgetGroupSync easier, consider building it out very basic initially with one public virtual SyncBudgetGroup method that can be replaced using Substitute.ForPartsOf
* Continue to stay TDD and build out BudgetGroupSync
* Consider how you want to handle items from onboarding that are modified, added, and unchanged. Have fun attacking this TDD and coding tests first, implementation second!
* Get to 100% test coverage on your changes using [these tools](#_Code_coverage_tools)
* EXTRA CREDIT: consider a dynamic property change notified status text that is updated as different budget groups are synced.
* Reference code is available to compare against your finished product at <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/2-Sync>

### Assignment: Integration test

* We will build one integration test to validate this entire pathway. Integration tests are slow, hard to maintain, and should be rare. However, they are often invaluable and catch failure unit tests cannot.
* Not that slow integration tests are almost always an order of magnitude faster than manually building and testing in a mobile app!
* Add an Xunit project called Onboarding.Models.TestIntegration to the solution. Move it into the MVVM solution folder.
* Note the explicit TestIntegration ending here - don’t blend big, slow integration tests into your unit test projects!
* Create a BudgetSyncTestIntegration class in the TestIntegration project
* Write a test or two in this file that signs in and syncs budget over the network. Have fun, try a few different scenarios, and think about how you would want to validate before trying this in the app!
* Reference code is available to compare against your finished product at <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/2-Sync>

### Assignment: Goals status sync

* We also know that we need to send the user’s selected goals and status up into our Ramsey cloud. Let’s create a basic template that will make this easy to implement in the future.
* In Sync, create a new class called `GoalsStatusSync`
* Give it two public virtual methods that take a TitledIconList: LogSelectedGoals and LogSelectedStatus
* The implementation in these two methods is going to be very draft. Just convert the list to a comma separated string using the GetSelectedIconIdsAsCommaSeparatedList method in TitledIconList and write it out to the console.
* Get to 100% test coverage on your changes using [these tools](#_Code_coverage_tools).
  + NOTE: your tests are not able to do behavioral validation…YET! Just make sure all the methods can be called and accessible properly to test runs.
* Check out the reference example in <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/2-Sync> to see an example implementation
* We’ve got a mockable basic start that we can call and debug to see if selections arrive properly. This will let us move forward, and we can add functionality to send to the cloud later.

### Assignment: Add sync to dependencies collections

* Time to get all the dependencies for the API into our Dependencies collection. To make this easy, update the Init() and RegisterServices() methods to take in an IAccessControlManager.
* For now, inside RegisterServices, we can create a local rest client hard coded to Prod and the appropriate sync singletons. Like this:

// TODO: test in integration test, and access all elements

static void RegisterServices(IServiceCollection collection, IAccessControlManager accessControlManager)

{

// Services

var restClient = new RestClient("https://api.everydollar.com/"); // TODO: support Test and QA environments

// Models

var budget = BudgetBuilder.Build();

var profile = OnboardingProfileBuilder.Build(budget);

// Services

collection.AddSingleton(accessControlManager);

collection.AddSingleton<IRestClient>(restClient);

collection.AddSingleton<IRemoteBudgetCalls, RemoteBudgetCalls>();

collection.AddSingleton<BudgetGroupSync>();

collection.AddSingleton<BudgetSync>();

collection.AddSingleton<GoalsStatusSync>();

// Models

collection.AddSingleton(budget);

collection.AddSingleton(profile);

// View models

collection.AddTransient<BudgetViewModel>();

collection.AddTransient<OnboardingProfileViewModel>();

}

* This is a very fast draft of services bring-up. We will revisit this in more detail later.
* Note we are not using these in the app yet. We will do that as we build out the completion final summary screen in a future set of assignments

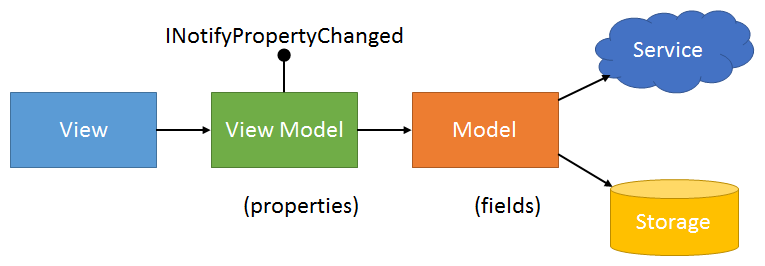
### Assignment: review all changes

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/1-RemoteBudget> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/2-Sync>

# Workshop 11: Budget Summary Widget

A budget summary widget is shown at the top of many screens. It dynamically updates progress bars indicating spending, amounts, and text based on user inputs as they change budget item amounts.

As this involves data that persists through several views, we will want to start with modelling to consider app model classes that may support this. Then we will design a presenter view model and connect our screens last.



### Key concepts in this workshop

* Treating graphical complexity and persistence as a “first class citizen” of our app models
* Modelling first, view models and views last
* Staying at 100% test coverage in all models and view models

### Assignment: Mapping visual needs to app models using ratios

* Take a minute to walk the screenshots, paying special attention to when the budget summary widget shows at the top of various screens. Note what it shows, and when it changes.
* To build this budget summary, adding a class to models to calculate budget group ratios will be a tremendous help. It can carry state across many views, and quickly calculate percentage ratios for each group.
* Now let’s do it! Stay TDD, with a focus on building behavioral tests. We want all the displayed spending groups to automatically calculate their percentage ratio of total spending each time a budget item is changed. Create an empty BudgetRatios class, a corresponding test file, and go!
* Don’t code yet – just think. Think through how to expose a ratio for each budget group in the budget. Consider how it should listen for budget item amount property changes in each budget group.
* TDD build a constructor that takes a budget instance to the constructor with a null check
* Build out behavioral tests that describe the behavioral expectations.
* Build out code to makes the tests pass!
* Get to 100% code coverage
* Review the reference example if you get stuck, or when you finish <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/1-SummaryWidget>

### Assignment: summary model for more visuals in app models

* To continue to support budget summary widget visuals, we will add a second model: Budget Summary.
* This will house Booleans for is on budget/over budget/under budget as well as a budget delta
* In addition, it will house the widths and offsets of each group’s colored bar
* Put pen to paper! Think through how this might work, and how you would model it.
  + Hint: this is much easier if it has a budget AND budget ratios instance inside!
  + Hint: width will need to come from the view -> view model layers. Try giving it a Width property with an easy to validate starting default value, like 100, that can be set by view models.
* Stay TDD! Try to build out test behaviors first and code implementations after tests.
* Get to 100% code coverage
* Stuck? Peek at the models in <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/1-SummaryWidget>

### Assignment: budget summary view model

* Great, we now have models to back the visuals. Let’s get started on the presenter layer view model
* We will only need one BudgetSummaryView model. Note that eventually our final completion screens will use this to (the last under/over/on budget screen before you arrive on the main budget page in the screen shots).
* Good news – this is going to be very easy! It is just going to pass through the ratios, widths, and heights from the budget summary. The only command up from views will be setting width.
* Build out a basic TDD test with an empty class, and implement the presenter. Make sure to test width getting from the VM up to the model and back
* Get to 100% code coverage
* Stuck? Peek at the models in <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/1-SummaryWidget>

### Assignment: dependency integration + view integrations

* Now that we have the view model and models for this widget, let’s get the view layer setup. First, we need to add budget ratio, summary models and the VM to the DI collection:

static void RegisterServices(IServiceCollection collection)

{

var budget = BudgetBuilder.Build();

var profile = OnboardingProfileBuilder.Build(budget);

// Models

collection.AddSingleton(budget);

collection.AddSingleton<BudgetRatios>();

collection.AddSingleton<BudgetSummary>();

collection.AddSingleton(profile);

// View models

collection.AddSingleton<BudgetViewModel>();

collection.AddSingleton<BudgetSummaryViewModel>();

collection.AddSingleton<OnboardingProfileViewModel>();

}

* For convenience, let’s also add a static access to the view model:

public static BudgetSummaryViewModel BudgetSummaryViewModel => ServiceProvider.GetService<BudgetSummaryViewModel>();

* Not beautiful, but good enough for discovery. We’ll revisit the entire DI collection and how to build it in a future workshop.
* Now, let’s build out a BudgetSummaryView. Create an empty content view in the Views folder called BudgetSummaryView. Update the OneButtonContentView to replace the “Budget Summary Here” label with this new view, like this:

<views:BudgetSummaryView HorizontalOptions="CenterAndExpand" VerticalOptions="Center" />

* Now, let’s get the app up and running, and get hot reload on the Housing Page that will have a budget summary on it. Then we can make changes and see them quickly.
* Make sure the XAML for housing page sets up the OneButtonContentView to have IsBudgetSummaryVisible=”true”
* Build out budget summary view “hard coded”, no view model yet. A grid with box views of various hard coded widths and colors for the progress bar works well. Put that whole grid in a frame to get rounded corners.
* Then add three hard coded label layouts for th text under the bar that indicates on budget, over budget, under budget. Give each one a visibility attribute. turn visibility on and off manually to style. Later, we will toggle visibility on each based on the VM’s budget Booleans.
* Great, now it is in, hard coded, looks about right. It is hacky, but, for now, in the view code-behind set binding context to Dependencies.BudgetSummaryViewModel in the BudgetSummaryView constructor. Also, now is a good time to assign a width to the view model. You’ll end up with something like this:

using Xamarin.Forms;

using Xamarin.Forms.Xaml;

namespace Onboarding.Views

{

[XamlCompilation(XamlCompilationOptions.Compile)]

public partial class BudgetSummaryView : ContentView

{

public BudgetSummaryView()

{

InitializeComponent();

// TODO: Find better pattern, using DI instead of service locator

BindingContext = Dependencies.BudgetSummaryViewModel;

Dependencies.BudgetSummaryViewModel.Width = 230; // TODO: get dynamically after view loads

}

}

}

* Great, now we have a budget summary view in our HousingPage! Make sure to it is visible in the TransportationPage, too. Navigate through the income-housing-transportation flow and make sure carries and displays changes properly.

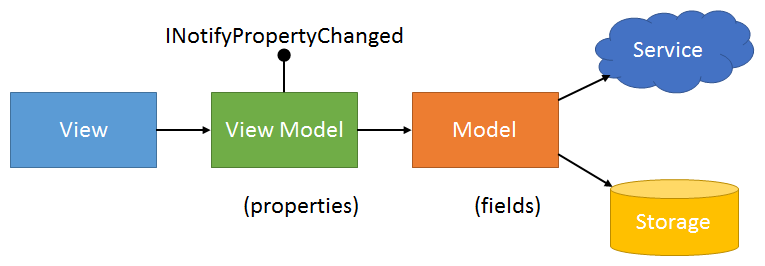
### Assignment: review all changes

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/4-ApiIntegration/2-Sync> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/1-SummaryWidget>
* IMPORTANT NOTE! There are some Android effect changes and OneButtonContentView changes in this tree to give views lifecycle events similar to OnAppearing and OnDisappearing on Pages. It is not my code - stolen from StackOverflow. Works well enough, and might be useful in production. Should probably use the same OnAppearing/OnDissappearing names and method signature as pages if take to production.

# Workshop 12: Reusable menu page

NOT FINISHED BELOW THIS POINT!!!

TBD



### Key concepts in this workshop

* TBD

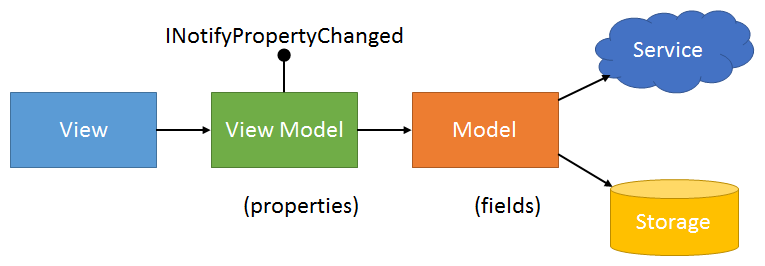
### Assignment: TBD

### Assignment: review all changes

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/1-SummaryWidget> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/2-MenuPage>

# Workshop 13: Completion actions

TBD



### Key concepts in this workshop

* TBD

### Assignment: TBD

### Assignment: review all changes

* Using a [folder diff](#_Diff_folders_on), walk the differences between the placeholder screens project at <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/2-MenuPage> and the assignments from this workshop in <https://github.com/lampo/mobile-discovery-csharp/tree/master/5-MoreViewInt/3-CompletionActions>

TODO: integrate from or reference content in [Fast Mobile Discovery.pptx](https://lampogroup.sharepoint.com/:p:/r/sites/RamseyPlus/Shared%20Documents/Global/Engineering/Mobile%20Development/Fast%20Mobile%20Discovery.pptx?d=we4d3f8c0f5e442d6863d38516adca3e9&csf=1&web=1&e=IfGPvH)

TODO: Budget main app integration

TODO: DI Collection “service” integration testing

NEVER USING THE AUTOINSTANTATE BS – TOO HARD TO DEBUG

Require callers to provide the freaking instance, and validate

TODO: API/service discovery – probably separate document