Peter McIntyre

Professor | Seneca College | Toronto Canada

DPS923 MAP523 notes - Wed Mar 23 2016

Core Data fetch requests. Web service fetch request, saved to Core Data store.

Classes that describe real-world objects

Before working more with Core Data, we will briefly review how to design and write a cu

that describes a real-world object.

Entity classes use *properties* for data/state storage, and *methods* f languages and frameworks.

In our apps, we work with objects and collections of objects, simila languages and frameworks.

What about persistence? How do we persist the object graph? Wel techniques, archiving, and the Core Data framework, which we stuc

> We should note that small and simple object graphs can be p list". However, this kind of storage is perhaps more suited to, settings, and is usually inadequate for persisting the app's da objects, do NOT use a plist.

Reintroduction to Core Data

Core Data is an object design, management, and persistence framework.

It helps you manage the lifecycle of objects and object graphs.

How do I get started?

Use a project template.

In the course code examples, you will see a project named **ClassesV2**.

Make a copy of that, and you will have all you need to get started.

> How do I make a copy?
>
> The Readme.txt file in the ClassesV2 project helps you do this.

What's in the "ClassesV2" project template?

The project template has all the pieces you need.

It is *nicely organized*.

The project template includes the **Core Data 'stack'**, which provides the necessary objects, and a factory/builder.

A **store initializer** is included, enabling you to create startup data for the app when launched for the

first time.

Its **Model** class is configured for Core Data, and has examples of properties of methods.

And it has a *table view controller*, and a standard *view controller*, which can use Core Data objects.

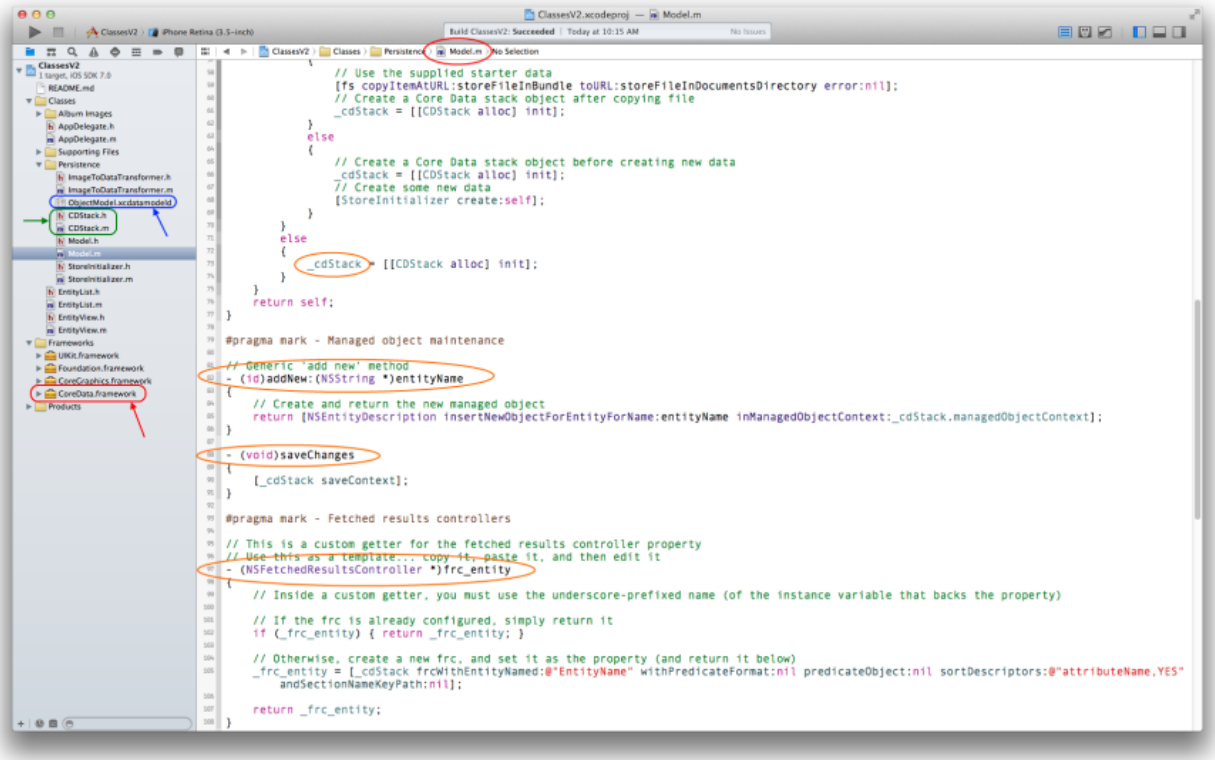
Oh, and it also has a WebServiceRequest class, which we will use soon.

Where do I see Core Data in a project?

The following screen shot shows some of the Model class in the ClassesV2 project template. (Click to view it in a new tab/window.)

Look at the following notable items:

- The CDStack class
- The ObjectModel file, which is the Core Data (object) model created by the model editor
- Properties and methods in Model.m, shown in the code editor



How do I design my entity objects?

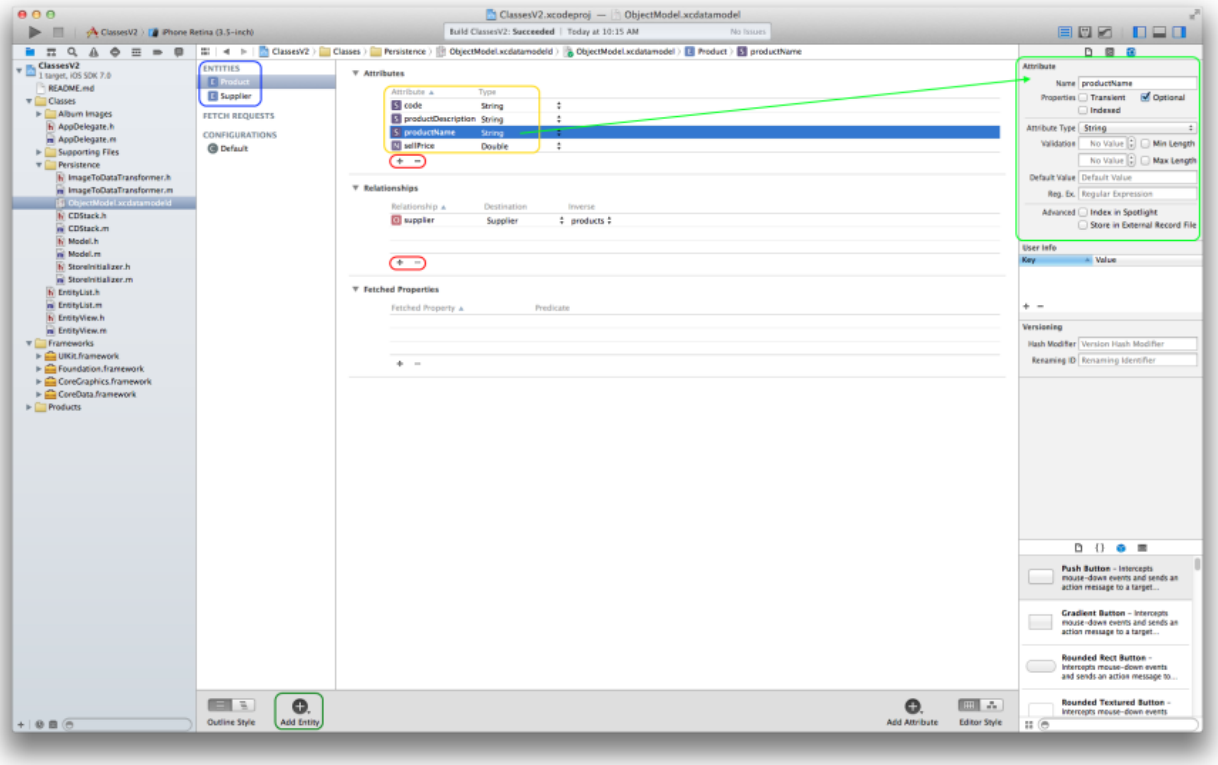
Use the Core Data model editor.

The screen shot below shows an example where two entities were designed. (Click to view it in a

new tab/window.)

Look at the following notable items:

- Add Entity control
- List of entities that have been designed
- Add (+) and Remove (-) properties controls
- For a selected/highlighted entity, a list of attributes and relationships
- For a selected/highlighted entity, available settings in the Data Model Inspector (in the right-side Utility area)



Naming conventions

Entity names begin with an upper-case letter. Multi-word names use camel-casing.

Property names - attribute, relationship - begin with a lower-case letter. Multi-word names use camel-casing.

Do NOT use “description” for the name of an attribute.

> Why?
>
> It’s documented in the Core Data Programming Guide, and in the NSPropertyDescription class reference document. In summary, DO NOT use these names for properties:
>
> - description

- class
- entity
- objectID
- self

A “to-one” relationship property name is singular. For example, “supplier”.

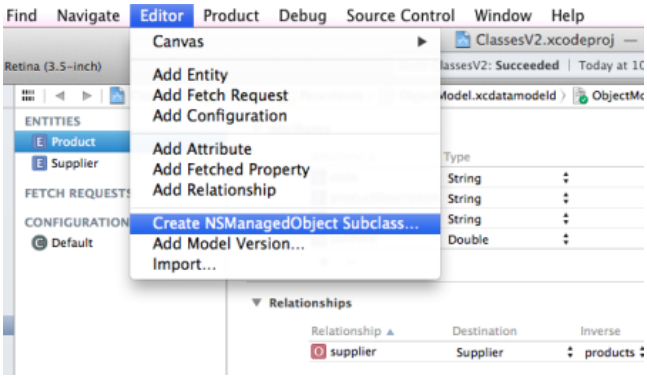
A “to-many” relationship property name is plural. For example, “products”.

After designing an entity, use Xcode to generate a custom class

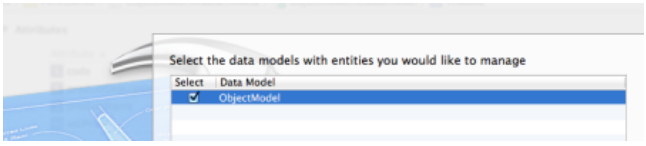
After designing an entity, and adding and configuring its properties, use Xcode to generate a custom class. Here’s how:

1. Display the Core Data model editor. Select one or more entities.

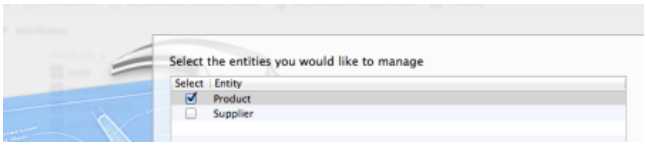
2. On the Editor menu, choose “Create NSManagedObject Subclass…”. Answer the dialogs appropriately.



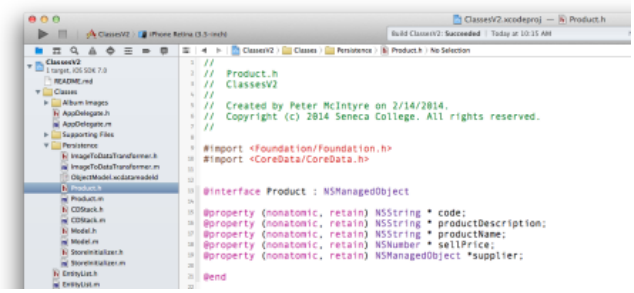
Then…



Then…



The result will be a class that is much more pleasurable to use when writing code.



What is a "managed object"?

"NSManagedObject is a generic class that implements all the basic behavior required of a Core Data model object." (From the NSManagedObject class reference document.)

NSManagedObject inherits from NSObject.

In addition, we normally will use Xcode to create a *custom class* for an entity, which inherits from NSManagedObject. For example, if we created a "Person" entity, and generated a custom class, the inheritance hierarchy would look like this:

NSObject > NSManagedObject > Person

What is the "managed object context"?

"An instance of NSManagedObjectContext represents a single "object space" or scratch pad in an application. Its primary responsibility is to manage a collection of managed objects. The context is a powerful object with a central role in the life-cycle of managed objects, with responsibilities from life-cycle management … to validation, inverse relationship handling, and undo/redo." (From the NSManagedObjectContext class reference document.)

Think of it as an in-memory "scratch pad" or temporary "work area" that you use.

What 'management' tasks can I perform on my objects?

All the tasks you would expect:

Fetch: Get all, or get one, or get some filtered, or get a scalar value (e.g. the number of objects). Done with a "fetch request", introduced below.

Add: Add new object.

Edit: Edit an existing object.

Remove: Remove an existing object.

These tasks are implemented as methods in the Model class.

Where is the data (object graph) persisted?

In your app's "Documents" directory.

The Core Data stack manages access to the store file. We don't have to worry about it.

The data format of the store file is *private*, and is **NOT** important to us. We repeat, it is **NOT** important to us. Understood?

What is a "fetch request"?

"An instance of NSFetchRequest describes search criteria used to retrieve data from a persistent store." (From the NSFetchRequest class documentation document.)

What "*search criteria*"?

- Name of entity being searched
- If required, a predicate (logical conditions that constrain a search)
- If required, sort descriptors

What's a "fetched results controller"?

A *wonderful* and *awesome* object.

"You use a fetched results controller to efficiently manage the results returned from a Core Data fetch request to provide data for a UITableView object." (From the NSFetchedResultsController class reference document.)

Are you planning to use a table view? Then you will want to 'bind' it to a fetched results controller. The result? Happiness.

If we have only one entity in our project, we create one fetched results controller.

If we have multiple entities in our project, we create a fetched results controller for each entity.

They are created in our Model class, as properties, with custom getters.

How do I…

How do I *define an entity object*?
Use the Core Data model editor.
Add an entity, and then add and configure properties.
Finally, use Xcode to generate a custom class for the entity.

How do I *write code to manage the entity*?
Do most of your work in the Model class.
Create a property for the entity's fetched results controller.
Then write methods for other fetch requests, and for handling object creation, modification, and removal.

How do I *perform searches and handle results*?
You can use the fetch request object in the fetched results controller if you plan to 'bind' the results to a table view.
Alternatively, you can use a fetch request object to do so.
In either case, results are available as an NSArray of zero-or-more objects.

How do I *add, edit, and remove objects*?
As noted above, write methods that handle object creation, modification, and removal in the Model class.
For 'add', create and configure a new object.
For 'edit', fetch the object. Then change its property values.
For 'remove', fetch the object. Then ask the context to remove it.
Always "save changes".

Can I study a code example?

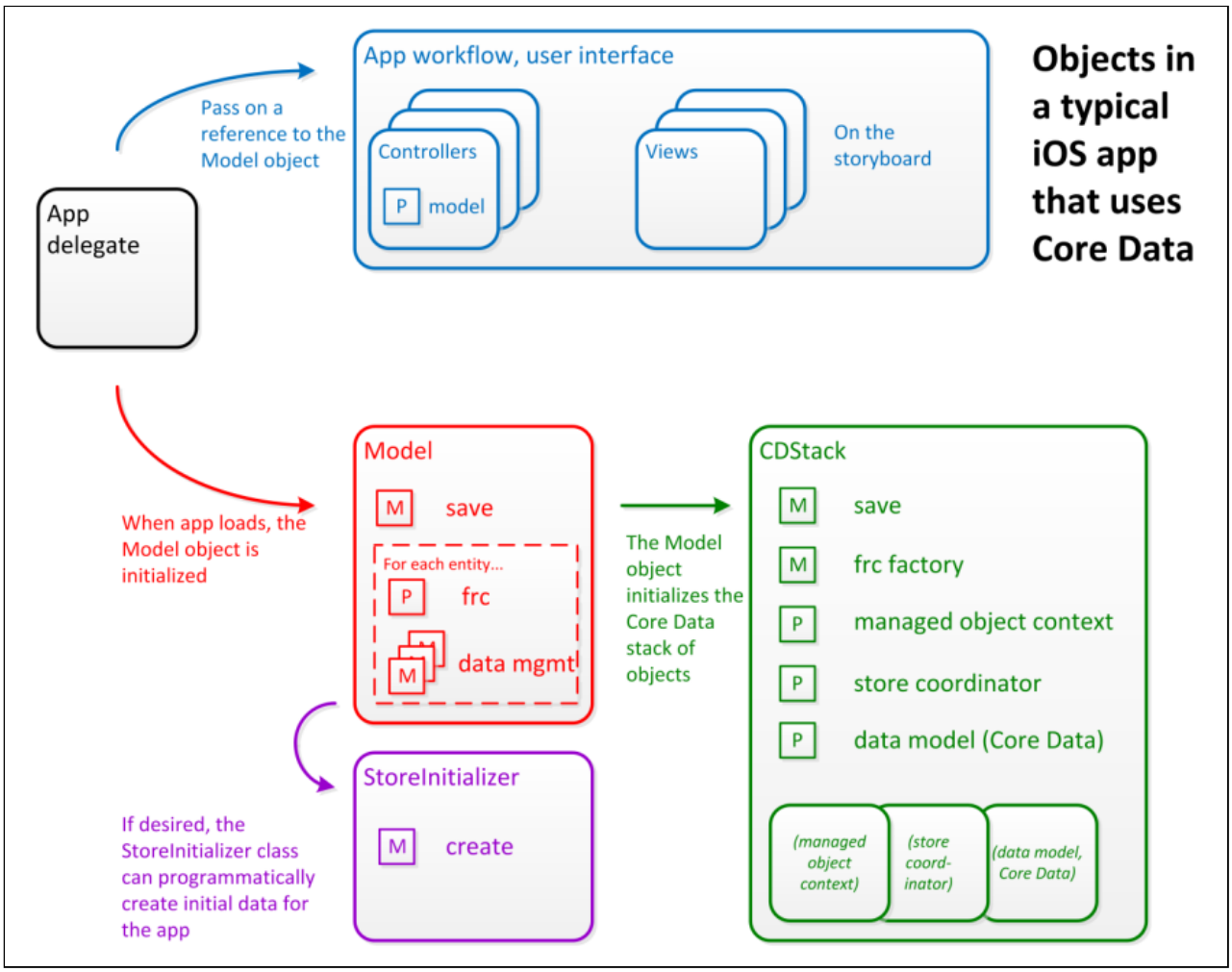
Yes.

The **CanadaAdd** code example (in the GitHub repository) is a good example of an app that works with one entity.

You *should attempt* to re-create this example, using your own copy of the ClassesV2 template. Strongly recommended.

Show me a diagram of the objects in an iOS app that uses Core Data

In the style that we have been using in class, here is a diagram. (Click to open it full-size in a new tab/window.)



The 'add/edit item' pattern

At this point in time, you have some knowledge of and experience with:

- navigation-based app style
- table view
- model object
- persistent store, using Core Data

What if you want to *add new items* to your app?

New items can be added from a number of different sources:

- a data entry view that you create as part of the app
- a select list with items fetched from the network
- on-devices sources, including the camera

How is this done? Using the *add/edit item* pattern. A central feature of this pattern is the *modal view* concept.

What is a modal view? It is a view "that provides self-contained functionality in the context of the current task or workflow." Continuing from the Apple *iOS Human Interface Guidelines* document:

> A modal view:
>
> - Can occupy the entire screen, the entire area of a parent view (such as a popover), or a portion of the screen
> - Contains the text and controls that are necessary to complete the task
> - Usually displays a button that completes the task and dismisses the view and a Cancel button that abandons the task and dismisses the view
>
> Use a modal view when you need to offer the ability to accomplish a self-contained task related to your app's primary function. A modal view is especially appropriate for a multistep subtask that requires UI elements that don't belong in the main app UI all the time.

Modal views are often used in iOS apps for these situations:

- The user is completing a fill-in form that has user interface controls (an "add item" pattern)
- The user must select one or more items from a lengthy list (a "select item(s)" pattern)

Here is the design and coding approach:

1. Create a controller
2. Add code that declares a *protocol* (with at least one method), and add a *delegate* property to the controller
3. Add a new scene to the storyboard (and embed it in a nav controller)
4. Alternate between the scene and controller code to build the user interface and handle events
5. In the controller, add code (to validate and) package the user input, and call the delegate method
6. In the presenting controller, adopt the protocol, code the segue, and implement the delegate method(s)

A protocol is a source code module that declares properties and methods for a task. Then, a class in your app can 'adopt' the protocol, and provide an implementation for the protocol's members.

Discussion in The Swift Programming Language guide

Discussion in the Cocoa Core Competencies document

You can follow along by studying the “CanadaAdd” code example. It implements the “add item” pattern.

You can also watch these videos, which cover the same content.

Part 1:

Add item pattern (part 1)

Part 2:

Add item pattern (part 2)

Create a controller

Add a new controller. It will be a subclass of UIViewController. In the code example, the controller name is “ProvinceEdit”.

Add code that declares a protocol, and add a delegate property to the controller

Edit the ProvinceEdit controller’s source code file.

At the bottom of the source code file (and BELOW the controller’ class closing brace), declare a delegate, and its (required) method:

```
protocol EditItemDelegate {
    func editItemController(controller: AnyObject, didEditItem ite
}
```

Notice that this protocol has only one method. You can declare as many methods as you need.

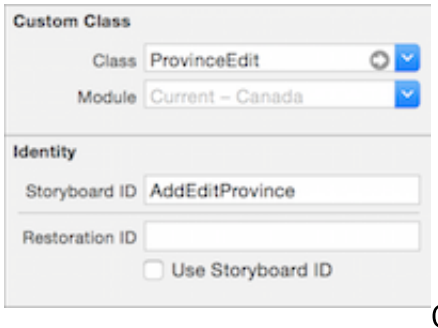
The implementation code - the tasks that the method actually performs - is written in the class that adopts the protocol.

Next, define a “delegate” property for the controller class.

```
var delegate: EditItemDelegate?
```

Add a new scene to the storyboard (and embed it in a nav controller)

On the storyboard, add a new View Controller scene from the object library. On the Editor menu, embed it in a navigation controller.



On the Identity inspector, set its Custom Class property to be the just-added ProvinceEdit class.

Also, in the "Identity" section (just below the Custom Class section), enter a string for the "Storyboard ID". A suggested name is "AddEditProvince".

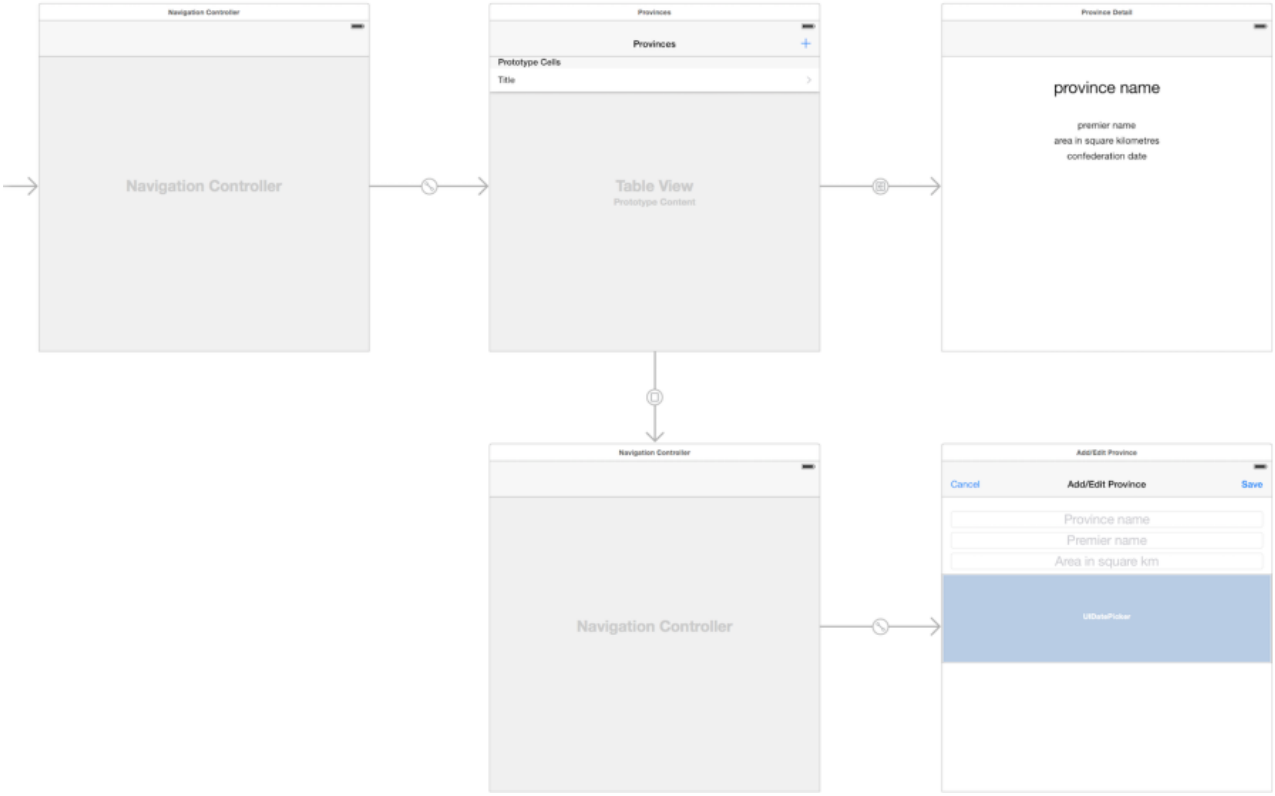
Set the scene's nav bar title.

Add the user interface objects. (For this app, we need three text fields, and a date picker.)

Add two 'bar button item' controls - one for 'Cancel' (left side), and the other for 'Save' (right side).

On the storyboard, create a segue from the presenter scene (ProvinceList) to the new scene. Here's how:

1. Add a 'bar button item' control to the presenter scene's nav bar, right side
2. Control+click-drag from the control to the new scene
3. On the segue popup, choose 'modal', then add an identifier (e.g. 'toProvinceEdit')



.

Alternate between the scene and controller code to build the user interface and handle events

As you have done before, alternate your work between the scene and controller code to build the user interface, and handle events. Add outlets, and actions for 'cancel' and 'save'.

.

In the 'add item' controller, add code that packages the user input, and call the delegate method

Add properties for the model object, and an entity object:

```
1 var model: Model! // will be configured by the presenting controll
2 var detailItem: Province? // not used in this example, but will be
```

In the 'cancel' method, we simply call back into the delegate method, and pass *nil* as an object.

In the 'save' method, your code will package the user input, and call the delegate method, passing along the user input package.

You can decide on the packaging format. If you have a single string or number item, just send that along. If you have multiple items to send, package them into an NSDictionary and/or an NSArray object. In *this* situation, package it into a new 'Province' object.

In the presenting controller, adopt the protocol, code the segue, and implement the delegate method(s)

In the presenting controller's declaration, adopt the new 'add item' protocol.

Code the segue.

Add a method (defined by NSFetchedResultsControllerDelegate) that will respond to changes in the 'fetchedObjects' result set. The method is *controllerDidChangeContent()*.

Then, implement the delegate method(s).

Finally, dismiss the 'add item' controller.

Standalone fetch request

A *fetched results controller* object is used with a table view.

If you are not using a table view, how do you query the on-device data store?

With a *fetch request* object.

> Previously, you have learned that a fetched results controller includes a fetch request property.

You can use a fetch request object to perform a *get-all*, *get-some-filtered*, or *get-one* query. For all of these situations, the results come back as an array that has zero or more objects.

> A get-all query will not need a predicate.
>
> A get-some-filtered or get-one query will need a predicate. Predicate string format examples are fully documented here (although some will not work with a Core Data store).
>
> For any query that will return a collection, you can choose to configure sort behaviour, if you wish.

Follow this guidance to code your first *standalone fetch request*. In this scenario, we are fetching a 'Sport' object that's in the Winter 2015 programming Assignment 1. We are using a *get-one* predicate, so the fetched results will be an array with exactly one object in it.

For best results, in your Model class, create a method that will accept a fetch request argument, execute the fetch request, and deliver the results. (We do this because controllers do not have direct access to the Core Data stack.)

Here's how to create a fetch request object:

```
1 let f = NSFetchRequest(entityName: "Sport")
```

Next, ask yourself whether you need a predicate, or sorting.

In our situation, we do need a predicate. We can configure it like this:

```
1 f.predicate = NSPredicate(format: "hostId == %@", argumentArray:
```

Here's what the 'execute fetch request' in the Manager class looks like:

```
// General-purpose fetch request
func executeFetchRequest(fetchRequest fr: NSFetchRequest) -> [AnyObject] {

    // Prepare an error object
    var error: NSError? = nil

    var results = cdStack.managedObjectContext?.executeFetchRequest(fr, error: &error)

    if let error = error {
        print("Fetch request error")
        results = []
    }

    return results!
}
```

(Winter 2015 content…) Fetch data from web service and save on device

One of the tasks in the winter-term Assignment 1 is to fetch data from a web service, and save the data on the device.

Today's code example, Toronto2015, is in the GitHub code repository. Download it, study it, and use some of its code and principles to enable you to add functionality to your app.

Your professor will guide you through the topic.

(Winter 2015 content…) How to design your coding plan

The code example is fully-commented, and lengthy, mostly because of the comments.

Most of the logic is in the 'launch' controller, in the Launch.swift source code file.

It relies on functionality in the Model class, and in a new 'extension' that's in the Extensions.swift source code file. Learn more about extensions in the Swift Programming Guide section.

What is the coding plan?

Well, it has a number of steps, summarized below:

In the launch controller, check if the data store has any objects in it.

If not, then register for a ‘fetch sports’ notification.

Then send a request for ‘sports’ to the web service, and wait for the results.

When the response comes back, it calls the ‘fetch sports completed’ method.

Next, register for a ‘fetch venues’ notification.

Send a request for ‘venues’ to the web service, and wait.

When the response comes back, it calls the ‘fetch venues completed’ method.

At this point, you have the data.

Add ‘sport’ objects to the device store, in a loop.

Then, add ‘venue’ objects to the device store, in a loop.

While adding each ‘venue’ object, must set the relation to the ‘sport’ object(s).

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Action:

Print

Like

Be the first to like this.

RSS feed

DPS907 Web services dev links

- DPS907 Course Notes
- Web service code examples

BTI420 Web apps dev links

- BTI420 Course Notes
- Web app code examples

DPS923 iOS dev links

- DPS923 Course Notes
- iOS code examples

My recent posts

- Responsive top-of-page logo image with Bootstrap
- Configure Fiddler export to save the message body
- Designing an ASP.NET MVC web app (version 1)
- Adding HTML Forms to ASP.NET MVC web apps
- Data persistence choices for ASP.NET MVC web apps

Past courses etc.

- 2015 Winter BTI420 (web apps)
- 2014 Fall DPS907 (web services)
- 2014 Winter BTI420 (web apps)
- 2014 Winter DPS923 (iOS dev)
- 2013 Fall DPS907 (web services)
- 2013 Summer BTI220 (web client)

Top
Blog at WordPress.com. The INove Theme.