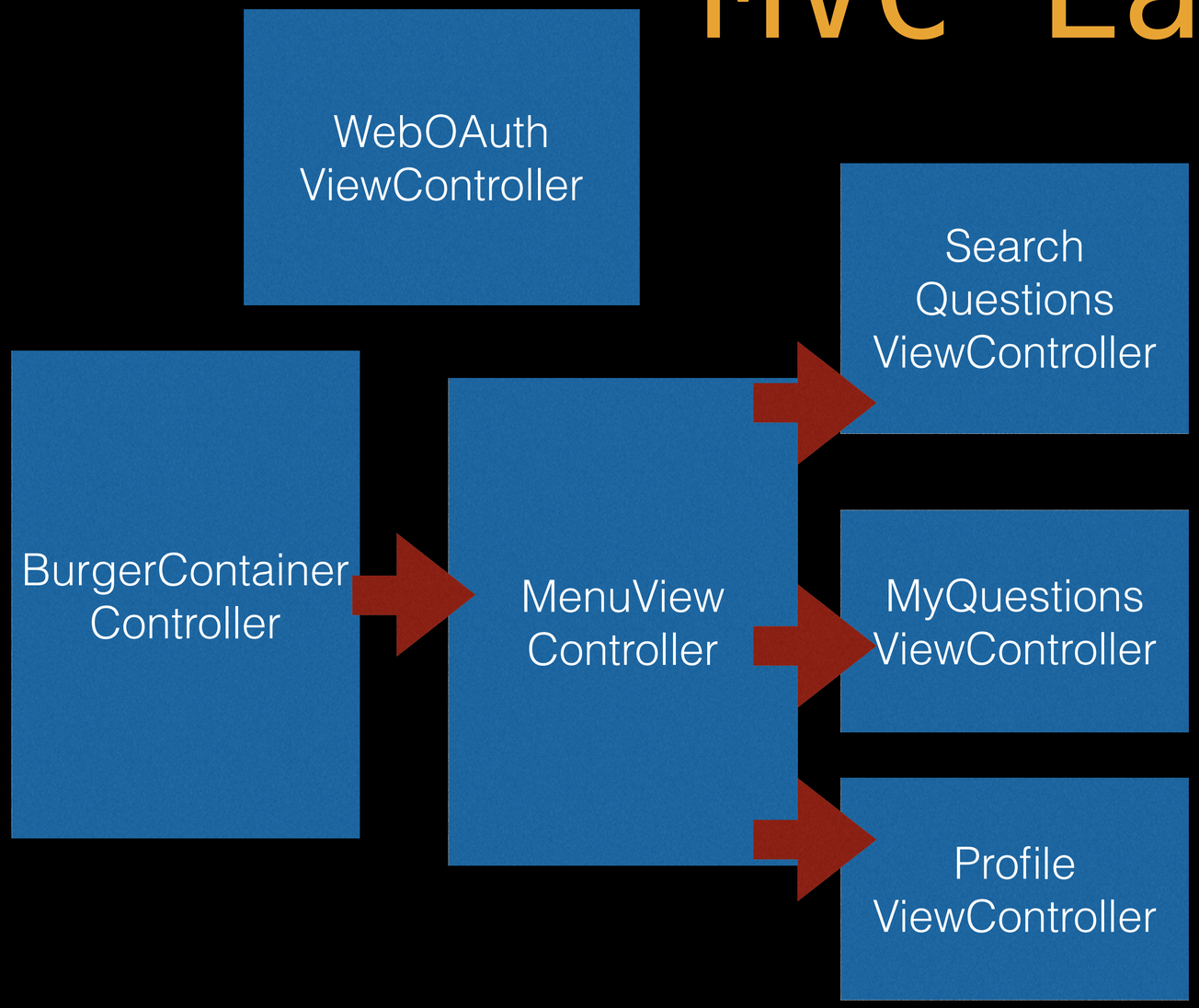
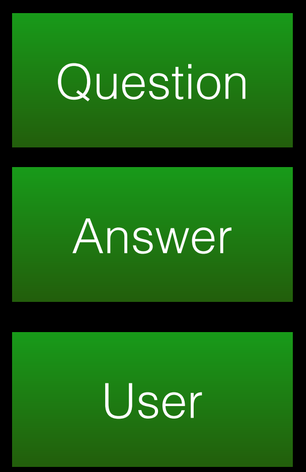
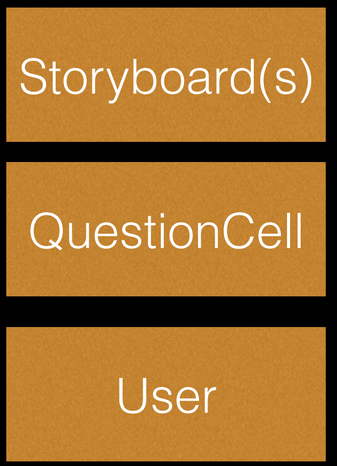
iOS Dev Accelerator Week 7 Day 1

- Custom Container View Controller
- OAuth with WebViews
- Lazy Loading in Objective-C
- UIDynamics Kit & Collection Views

MVC Layout


WebOAuth ViewController
BurgerContainer Controller
MenuView Controller
Search Questions ViewController
MyQuestions ViewController
Profile ViewController
Controllers


Question
Answer
User
Models

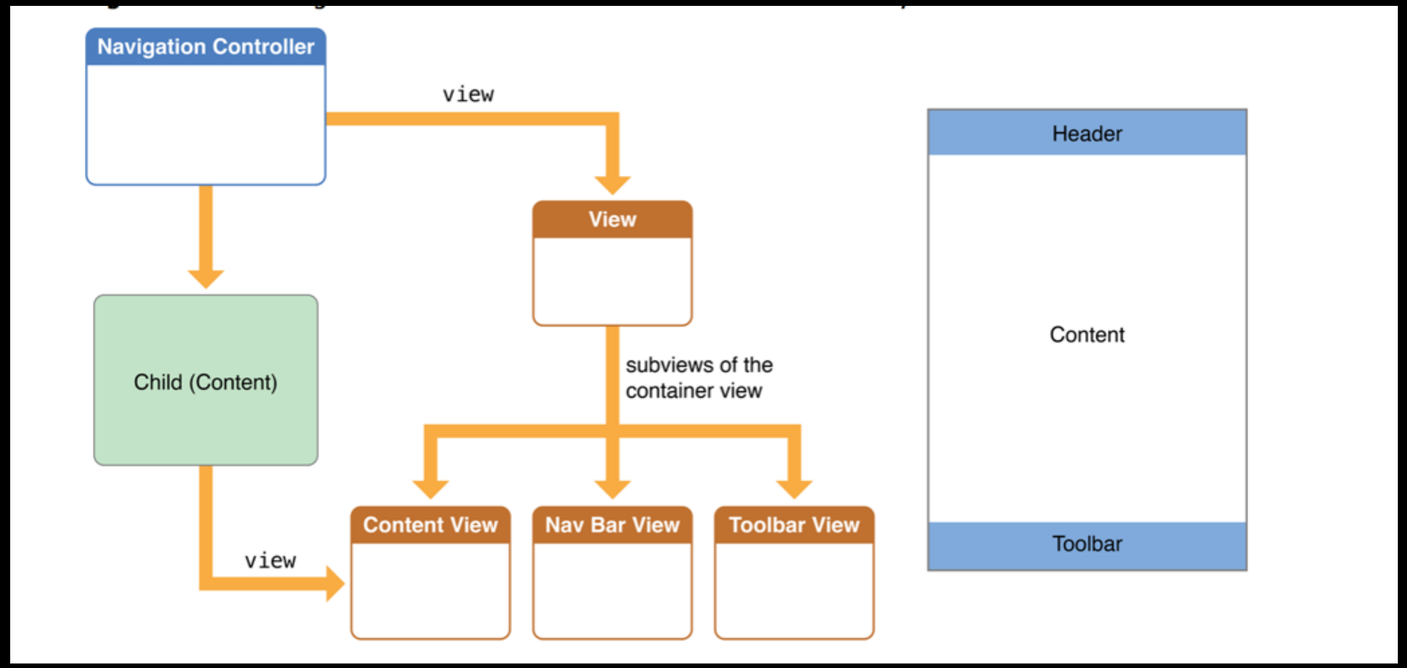

Storyboard(s)
QuestionCell
User
Views

Container View Controllers

- All of the view controllers you have created so far are considered ‘Content View Controllers’
- Container View Controllers are similar to Content View Controllers, except they manage parent-child relationships between the Container VC, the parent, and its Content VC(s), the children.
- The Container VC is in charge of displaying its children VC’s views.

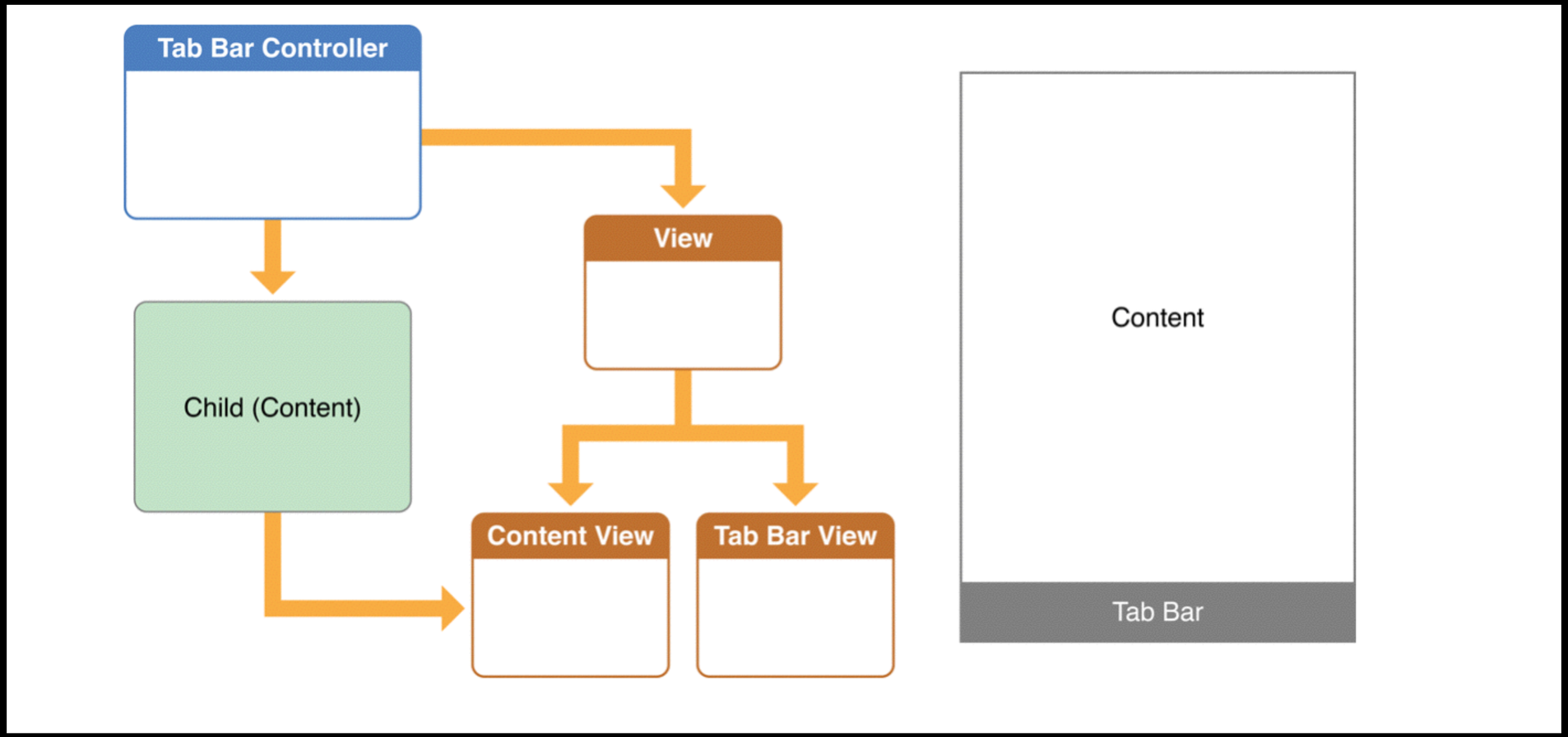
Container View Controllers

- An example of a Container View Controller you have used is the Navigation Controller:



Container View Controllers

- Same Idea with a Tab Bar Controller:



Container View Controllers

- “The goal of implementing a container is to be able to add another view controller’s view as a subview in your container’s view hierarchy”
- When a new VC is added on screen, you can ensure the right events (viewDidLoad, willAppear,etc) are given to all VC’s by associating the new VC as a child of the container VC.

Getting your child on screen

```
func setupChildVC() {
    //create a new VC
    var childVC = UIViewController()
    //tell this VC that we are adding a child VC
    self.addChildViewController(childVC)
    //set the child VC's view's frame, in this case it will
        completely cover the container VC
    childVC.view.frame = self.view.frame
    //add the child view to the parent view
    self.view.addSubview(childVC.view)
    //notify the child vc he is now on screen
    childVC.didMoveToParentViewController(self)
}
```

Taking the child off screen

```
childVC.willMoveToParentViewController(nil)
childVC.view.removeFromSuperview()
childVC.removeFromParentViewController()
```

Transitioning your Child VCs

There is also an optional method you can use for cool transitions

```
//being process of removing old one and adding new one
childVC.willMoveToParentViewController(nil)
self.addChildViewController(nextChildVC)

//set the new child's frame to be off screen to the left, and create an end frame
    that is off the screen to the right
nextChildVC.view.frame = CGRectMake(-700, 0, self.view.frame.width, self.view.
    frame.height)
var endFrame = CGRectMake(700, 0, childVC.view.frame.width, childVC.view.frame.
    height)

//call the transition method on our container view controller
self.transitionFromViewController(childVC, toViewController: nextChildVC,
    duration: 0.5, options: UIViewAnimationOptions.TransitionCrossDissolve,
    animations: { () -> Void in
    //give our new child a frame equal to our containers view, putting him fully
        on screen. set our old child's view to the end frame so it slides off to
        the right
    nextChildVC.view.frame = self.view.frame
    childVC.view.frame = endFrame

}) { (success) -> Void in

    //finish the operation
    childVC.removeFromParentViewController()
    nextChildVC.didMoveToParentViewController(self)

}
```

Demo

OAuth with Webviews

OAuth with Webviews

- When we first learned OAuth, we learned the workflow that actually takes the user completely out of our app and into the web browser.
- Once the the user gives permission for our app, the service provider redirects the user back to our app via the redirect URL.
- This works great, but some web API's don't support mobile app style redirect URL's (appname:\\parameters), which means they they aren't able to redirect the user back to the mobile app.

OAuth with Webviews

- We can get around this by never leaving our app in the first place
- Instead of relying on Safari to show the user the web page that is generated by the service provider to grant permission for our app, we can use a web view.

WebView+Oauth workflow

1. Instantiate a WKWebView and become its delegate
2. Create your initial OAuth URL, which for most web API's will include your clientID and redirect URL
3. implement webView:decidePolicyForNavigationAction:decisionHandler:

webView:decidePolicyForNavigationAction:decisionHandler

- This method is part of the WKNavigationDelegate, and is called anytime the web view is about to load a URL
- We will use this method to listen for each request the web view loads, until the request is loaded that contains the oauth token.
- This method is essentially taking the place of the openURL: method in the app delegate we did for the first oauth workflow.

Special considerations for stack apps

- Make sure the redirect URI you pass in with your initial oauth url is “https://stackexchange.com/oauth/login_success”
- Enable Desktop OAuth Redirect URI, which allows non web apps like ours to participate in the OAuth process.
- No need to have the client secret or key in your app at all, yay!
- And there is no 2nd post call we have to make after our initial request. Instead of giving us an intermediate ‘request code’, they gave us the full oauth token after the first request.

Demo

Lazy Loading in Objective-C

Lazy Loading

- During our swift month we learned lazy loading by lazily downloading images only when they were actually needed, instead of downloading them all at once with our initial API request
- You can also apply this same principle to your properties. Leave them uninstantiated until they are actually needed
- We know a property is needed when its getter is called
- This is much more common practice in objective-C simply because we can’t give our properties values inline with their declaration
- You can also do this in swift by simply marking your properties as lazy

Lazy Loading workflow

- Override the getter of any property you want to make lazy and you basically just do 2 things:
    1. first check if the underlying i-var is not nil. If it is, that means the property has already been instantiated, and simply return it.
    2. setup the i-var and return it (this part of the getter will only run if step 1 evaluated to false)

Lazy Loading workflow

1

2

```
- (NSManagedObjectModel *)managedObjectModel {
    // The managed object model for the application. It is a fatal error for the
      application not to be able to find and load its model.
    if (_managedObjectModel != nil) {
        return _managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"HotelManager"
        withExtension:@"momd"];
    _managedObjectModel = [[NSManagedObjectModel alloc] initWithContentsOfURL:
        modelURL];

    return _managedObjectModel;
}
```

It is not uncommon to see ALL properties done this way

Demo

UIDynamics

UIDynamics Kit

- UIDynamics Kit was introduced with iOS 7.
- It is intended to give developers a way to improve the user experience of your app by incorporating real world behaviors and characteristics (gravity, collisions, etc) into your apps animations.
- UIViews, and any other objects conforming to UIDynamicItem protocol are supported.
- Dynamic behaviors become active when you add them to an animator object, which is simply an instance of UIDynamicAnimator class.

Supported Behaviors

- **UIAttachmentBehavior:** specifies a connection between two dynamic items, or between an item and a point. When one item movies, the other moves. Has damping and oscillation to make it look naturally springy
- **UICollisionBehavior:** lets dynamic items participate in collisions with each other and with specified boundaries.
- **UIGravityBehavior:** specifies a gravity vector for dynamic items.
- **UIPushBehavior:** specifies a continuous or instantaneous force vector for its dynamic items.
- **UISnapBehavior:** specifies a snap point for dynamic Item. The item snaps to a point with a configured effect.

UIDynamicAnimator

- “A dynamic animator provides physics related capabilities and animations for dynamic items”
- To use dynamics,configure one or more dynamic behaviors and then add those to the dynamic animator.
- UIDynamicAnimator has special initializers. You choose the one according to what you want to do achieve:
  - **initWithReferenceView:** Use this if you just want to animate some UIViews (or subclasses of UIViews)
  - **initWithCollectionViewLayout:** Use this if you are applying dynamics to a collection view (really cool and not too hard)
  - **init:** Use this if you want to apply dynamics to custom objects you created that conform to UIDynamicItems protocol

UIDynamicAnimator Internal Workflow

- A dynamic animator works with dynamic items in these steps:
    1. Before adding an item to a behavior, you specify the items starting state (position, rotation, bounds — this will already be done for you)
    2. You then add the behavior to the animator, and the animator takes over. It updates the item's position and rotation as the animation proceeds.
    3. You can programmatically update and items state in the midst of an animation, after which the animator takes back control.

UIDynamicAnimator With Regular Views Workflow

1. Get your interface laid out, with your views added to a super view.
2. Create a property for the UIDynamicAnimator, and then instantiate it using the init with the reference view. Pass in your view controllers view.
3. Create and configure behaviors.
4. Add behaviors to the animator.

Demo

Creating Boundaries

- To place a boundary with UIDynamics Kit, you need to create an instance of UICollisionBehavior
- You can use its init that takes in an array of items. These are the items that need to respect the boundary.
- You then set translatesReferenceBoundsIntoBoundary to true
- Finally add the behavior to the animator.

Demo

Dynamics and collection view layout

- Dynamics Kit was designed from the beginning to work pretty seamlessly with collection view's and their layout objects.
- UIDynamicAnimator has its own special init that takes in an instance of UICollectionViewLayout or its subclass UICollectionViewFlowLayout.
- Lets review collection view layout a bit

UICollectionViewLayout

- Computes layout attributes as needed for:
- CollectionView Cells
- CollectionView Supplementary Views
- Decoration Views

UICollectionViewLayout

- Every collection view uses a layout object to determine where each view it manages should be placed and behave on screen.
- Apple provides a concrete subclass of UICollectionViewLayout called UICollectionViewFlowLayout that gives us a line based layout that we can use right out of the box.
- A collection view's layout is highly customizable. When you want to create a custom layout, you first need to determine if it is suitable for you to subclass flow layout (less work), or create a brand new subclass of UICollectionViewLayout(more work).

UICollectionViewLayoutAttributes

- Manages the following layout-related attributes for a given item in a collection view:
  - Position
  - Size
  - Opacity
  - zIndex (overlapping cells, above or below)
  - Transforms
- One attribute instance per view!

UICollectionViewFlowLayout

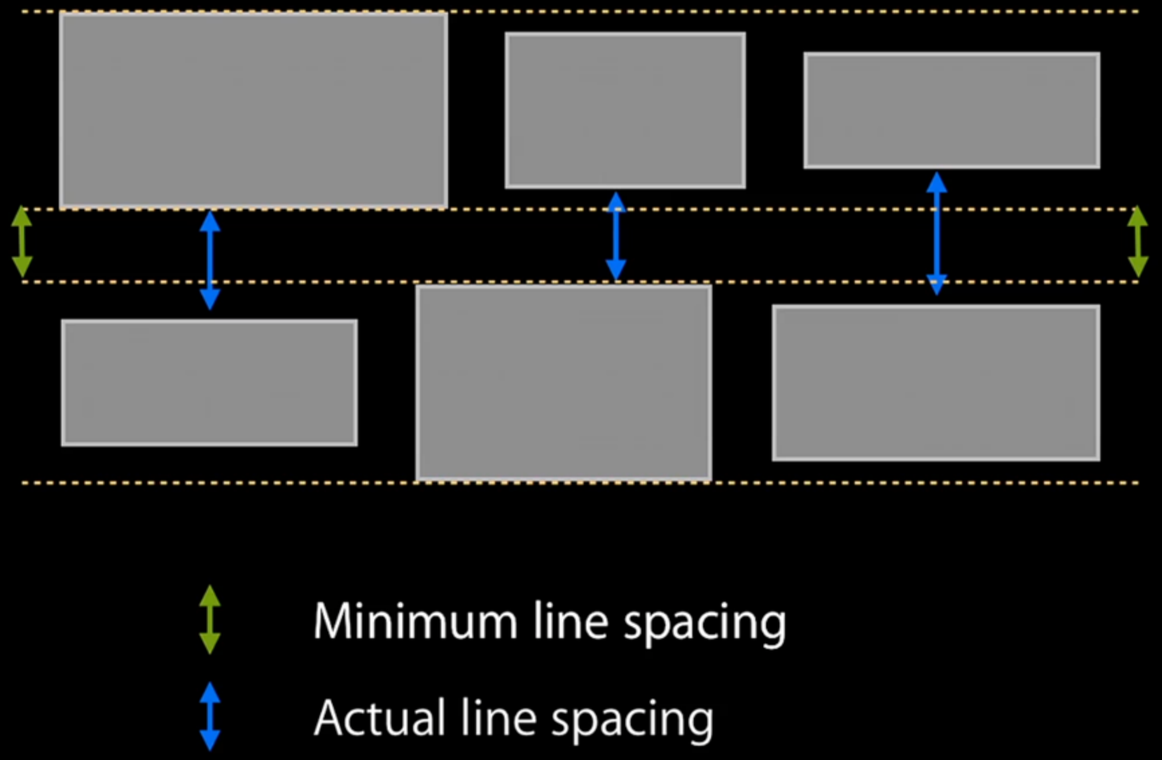
- Flow layout is a line-oriented layout. The layout object places cells on a linear path and fits as many cells as it can along the line.When the line runs out of room, it creates a new line and continues the process.
- Can be configured as a grid or as a group of lines.
- Out of the box, it has lots of things you can customize:
  - Item Size
  - Line Spacing and Inter Cell spacing
  - Scrolling direction
  - Header and footer size
  - Section Inset
- And you customize each of those things either globally with a single property, or through a delegate

Item Size

- The item size for each cell can be set globally by setting the itemSize property on your flow layout.
- Or if you want different size per item, you can do it through the delegate method collectionView:layout:sizeForItemAtIndexPath()

Line Spacing

- You can set a minimum line spacing, either globally or through the delegate:



Inter-item Spacing

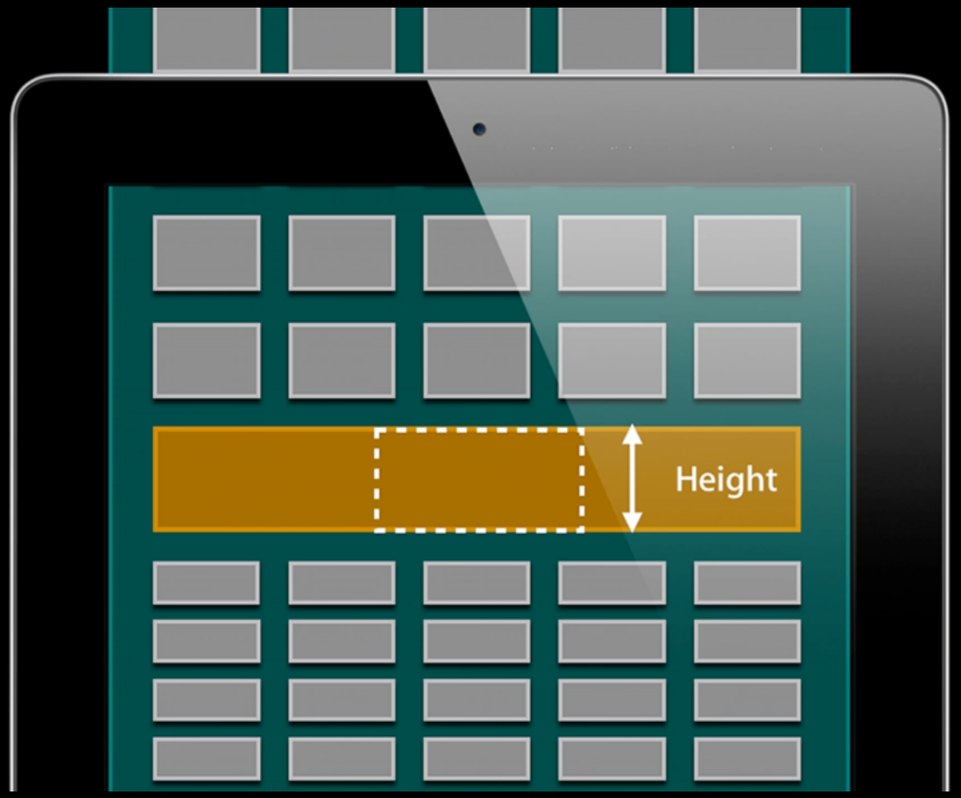
- Same with spacing between individual items:

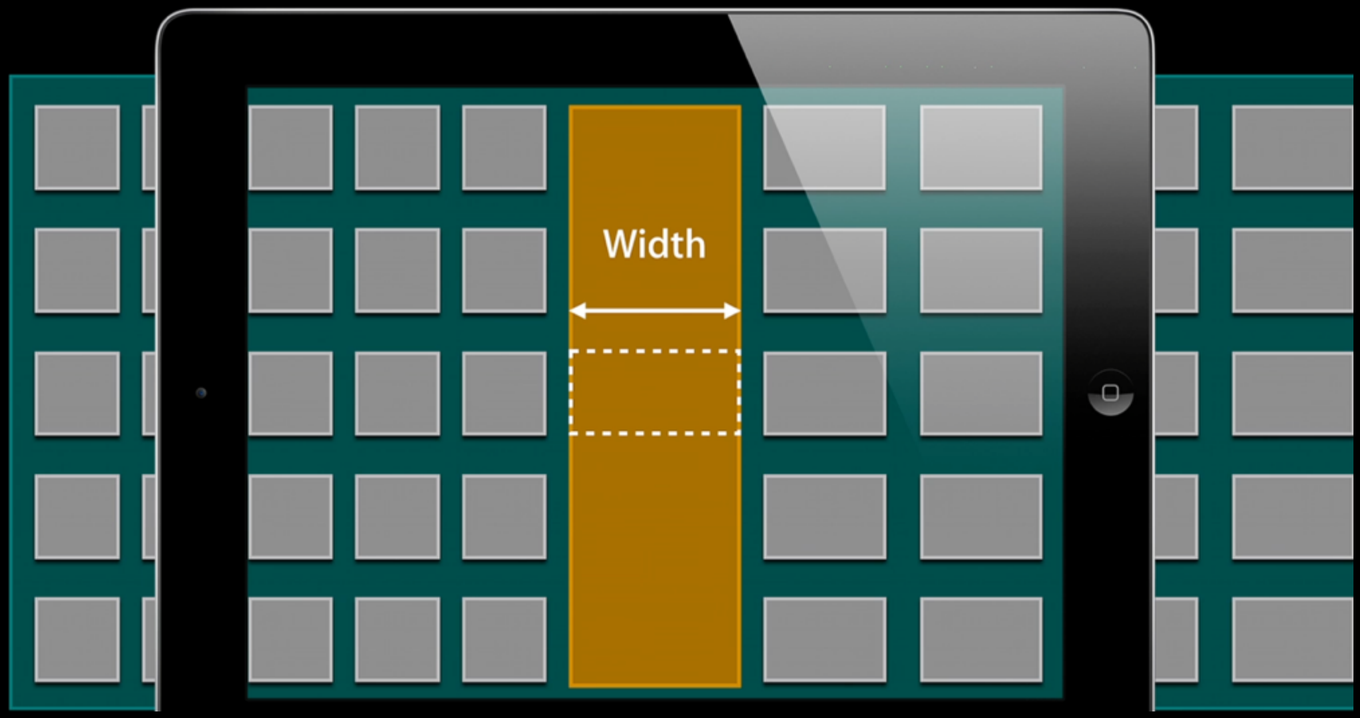
Actual interitem spacing

Minimum interitem spacing

Scrolling Direction

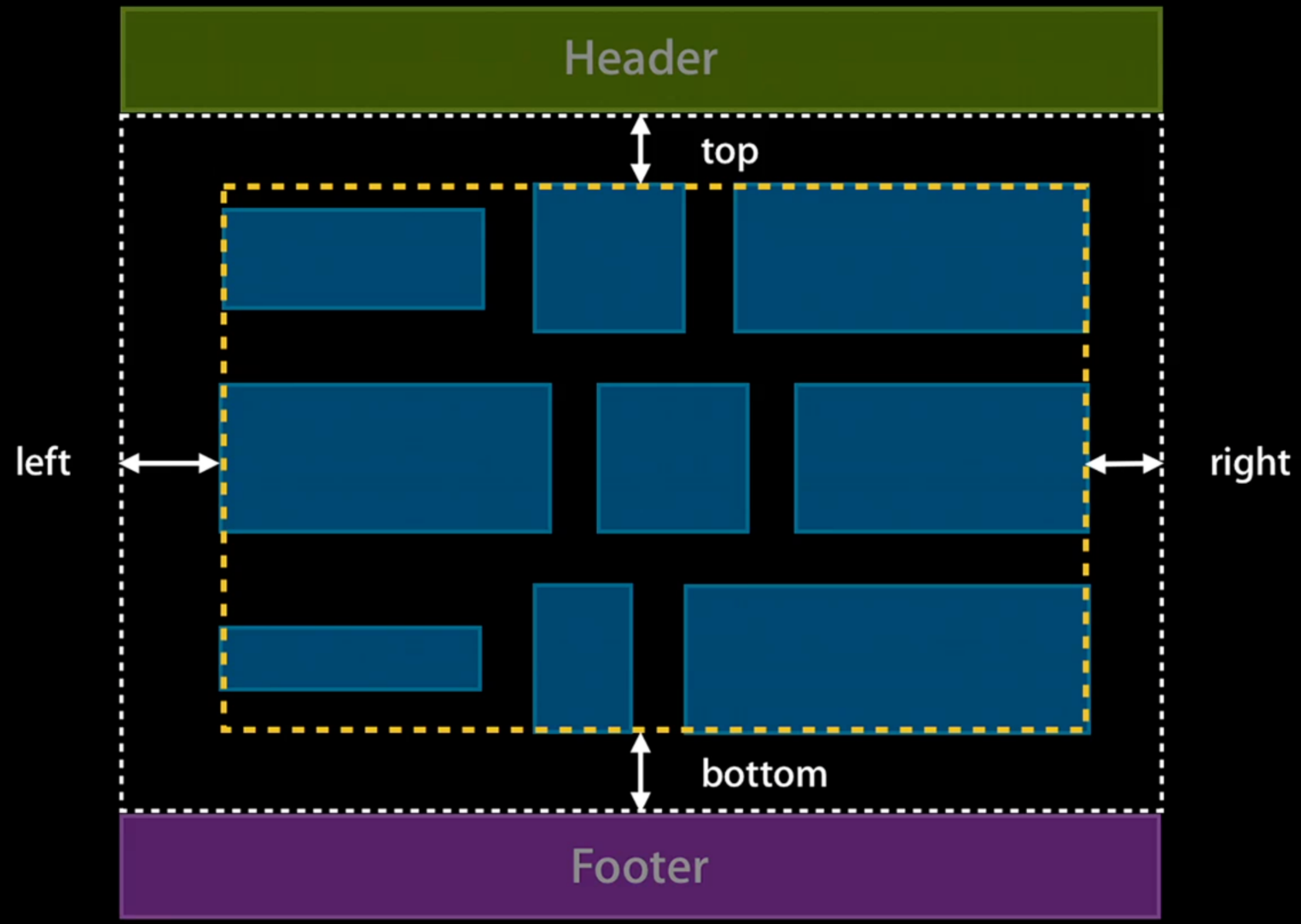
- The scroll direction of your flow layout can defines the base behavior of your entire flow layout
- Dictates the dimensions of the header and footer views:





Section Insets

```
inset = UIEdgeInsetsMake(top, left, bottom, right)
```



Changing the layout

- When you want your layout to change, you need to invalidate your layout.
- You can call invalidateLayout to trigger a layout update.
- You can use performBatchUpdates:completion: and anything you change inside the update block will invalidate the layout AND cause awesome animations.
- Whenever the bounds of the collection view changes, the layout is invalidated (rotation, scrolling)

Demo

When to go custom?

- If you are constantly changing the location of all the cells.
- Basically, if your collection view doesn't resemble a grid, its time to go custom.

Required overrides on UICollectionViewLayout

- collectionViewContentSize: Returns the width and height of the collection view's contents. **This is the entire size of the collection view's content, not just what is visible**.
- layoutAttributesForElementsInRect: Returns the layout information for the cells and views that intersect the specified rectangle. **In order for the collection view to know which attribute goes to cells or views, you must specify the elementCategory on the attribute (cell, supplementary view, decoration view) This is constantly called, every time time the user scrolls the collection view. Yikes.**
- layoutAttributeForItemAtIndexPath: Use this method to provide layout information for your collection view's cells. Do not use this method for supplementary or decoration views.
- layoutAttributesForSupplementaryViewOfKind:atIndexPath: & layoutAttributesForDecorationViewWithReuseIdentifier:atIndexPath: same as above but for supplementary views

UICollectionViewLayout Order of Operations

1. prepareLayout
2. collectionViewContentSize
3. layoutAttributeForElementsInRect (which will probably call layoutAttributesForIndexPath)

If the layout is invalidated, prepareLayout is called and this cycle is repeated.

Dynamics and collection view layout

- In order take advanced of Dynamics and Collection view Layouts, the first thing you will want to do is create a subclass of UICollectionViewFlowLayout
- You then need to implement prepareLayout.
- This method tells the layout object it needs to update its layout. It is called the first time the collection view is presented, and anytime the layout is invalidated.

`prepareLayout()`

- In this method, you need to create behaviors for all the items on screen.
- The trick is, you are only allowed to create and add the behaviors if they aren't already there. Otherwise, you will get weird behaviors and crashes.

Demo

`layoutAttributes`

- The next methods you need to implement on your custom layout `object is layoutAttributesForElementsInRect:` and layoutAttributesForItemAtIndexPath:
- Luckily, the dynamic animator has methods we can use to make both of these methods one liners.

Demo

`shouldInvalidateLayoutBoundsChange:`

- This is the big one.
- This method is called whenever the bounds of the collection view changes (ie during scrolling)
- In this method, it is up to us to update the animator with the new locations of the items, in order to get our animations to look cool.

Demo