Richard Topchii

Responsive Scrollable Dynamic User Interface on iOS

Bachelor's Thesis
Information Technology

December 2016

MAMK
University of Applied Sciences

DESCRIPTION

MAMK University of Applied Sciences	**Date of the bachelor's thesis** 02.12.2016
Author(s) Richard Topchii	**Degree programme and option** Information Technology

Name of the bachelor's thesis

Responsive Scrollable Dynamic User Interface on iOS

Abstract

The purpose of this thesis was to investigate new ways of programming the user interfaces on iOS, so that they would satisfy both the performance requirements and would be convenient to use for developers. Nowadays the interfaces in mobile apps have become very complex. On the other hand, users expect apps to be performant and reliable. The developers need a way to quickly introduce new changes in the app without the need to change a lot of code. The major companies, for example, the Facebook, the LinkedIn or the VK are designing own frameworks to tackle these challenges.

I will propose the framework for the declarative layout definition which will also have acceptable level of scrolling performance. The thesis contains both the architecture of the framework as well as technical details. The proposed solution will be evaluated by building fully functional social networking iOS app and distributing it using the App Store platform. By looking at the analytics data I will evaluate reliability of the proposed framework.

The results revealed that the proposed solution greatly improves developers' productivity when working with layout. The scrolling performance level is acceptable and very high, but there is still a small room for improvement. The solution work stable on a large scale showing more than 95% of crash-free users in a social networking app.

I consider the proposed framework ready for use in a production environment. However, I recommend improving performance level of the framework. Another direction for the improvement can be the introduction of the advanced interactions, for example, videos that start playing automatically while the user scrolls the feed.

Subject headings, (keywords)

iOS, Objective-C, Apple, performance, social networks, mobile apps

Pages	**Language**	**URN**
87	English	**URN:NBN:fi:amk-2016121820906**

Remarks, notes on appendices

Tutor Reijo Vuohelainen	**Bachelor's thesis assigned by** Mikkeli University of Applied Sciences

1 INTRODUCTION

Social networks are widely used nowadays. Users can access the network in multiple ways: on the Web or with the help of native mobile apps. It is common for social networks to have a user interface which can be scrolled infinitely. This is often referred to as Timeline, Newsfeed or Wall. To make the transition from the Web to mobile apps (and vice versa) smooth, the interfaces should share some similarities. Apple has already set a high mark with built-in iOS apps, so that the users expect to have the same level of responsiveness from any third-party app. In general, this level is easily achieved for simple apps like a contact list, where all the elements representing contacts look exactly the same. However, the layouts used in social networks are much more complex and dynamic. In static layouts all the elements of a collection look similar to each other, if not the same, while in dynamic layouts each item is different. From a developer's perspective these layouts are more challenging to program: the number of edge cases grows a lot with big freedom in arranging the elements of a post (texts, photos and other attachments).

The approach for architecting an iOS app described in this thesis was used in practice while developing VPages app. The application has similar interface to VK and extends its functionality in community management. VK (originally VKontakte) is the largest European online social networking service. It is based in St. Petersburg. It is available in several languages, but is especially popular among the Russian-speaking community. Like other social networks, VK allows users to message each other publicly or privately, to create groups, public pages and events, share and tag images, audio and video, and to play browser-based games. While this feature set may be enough for the general public, social media administrators would like to have more control over their communities or brand pages. VPages app targets the needs of this specific audience.

1.1 Purpose

Newsfeed-like layouts are very common for iOS apps and may have very wide application: from social networks and chats to catalogues and magazines. App user retention depends highly on user experience, and responsiveness is one of its core components. Some users may open an app less frequently or remove it completely, if they are not satisfied with its performance. For the applications showing advertisements, smaller user base means smaller profits. Being able to build complex and responsive interfaces gives more freedom for designers to express themselves. They may create completely new layouts and ways of how users interact with the content, thus making app experience engaging and memorable.

On the other hand, the ability to prototype and fine-tune layout quickly will improve collaboration between designers and developers. Personally, I had experience, when the user interface design has been changed after it has been completely implemented in code. The UI was tested using prototyping tools such as Pixate and Invision. The problem was revealed during the user testing phase: the app interface had steep learning curve, so the users didn't understand how to use it.

Having a framework for building dynamic interfaces will help reduce bugs as some parts of code will be reused in multiple apps. At first, the framework would be exposed to different usage patterns, so the testing would be more thorough. Second, any bug fixed in the framework would be automatically fixed in all apps using that framework.

1.2 Research goals

The goal of this study is to find a solution for implementing responsive native dynamic interfaces on iOS, similar to Timeline. The solution should allow the layout to be defined in a declarative way and to prototype new layouts quickly, without the need to understand implementation details.

The performance goal could be achieved by doing as much computation as possible asynchronously, on a background thread and by using computation resources rationally. The Abstraction goal could be achieved by using specific syntactic constructions, declaring data types which describe layout, such as lists, stacks and grids. These data types would also contain methods and algorithms to calculate layout. Using these data types programmers would be required only to specify the layout they want to achieve, without explicitly working with sizes, dimensions and positions of individual views.

2 USER INTERFACE PROGRAMMING ON IOS

Since the introduction of Swift in 2014 (Swift (programming language) - Wikipedia), it is possible to use one of three programming languages for native iOS development. The other two are Objective-C, the latest version of which appeared in 2006 (Objective-C - Wikipedia) and Objective-C++, a mix of Objective-C and C++.

In practice, if there are no reasons to use C++, the choice is made between Objective-C and Swift. Both Swift and Objective-C may be used in a single project (Swift and Objective-C in the Same Project), independently of the base project language. Among the benefits of Swift are cleaner syntax, type inference and rich functional programming features. However, the choice was made in favor of Objective-C because of the following:

- Swift takes longer to compile, and the compilation time depends on the project size.

- The Swift compiler is not yet mature, and some of its errors may be difficult to trace. In large projects this may lead to a lot of time being spent on finding and correcting pieces of code that cause the compiler to crash (A collection of test cases crashing the Swift compiler). Objective-C compiler is well-tested and crashes less.

- Swift is an evolving language. Its syntax is constantly in change. While there is automatic conversion to new syntax in Xcode, it does not work in some cases. With a large codebase this may lead to extra time being spent on manual conversion.

- CocoaPods (How to Use CocoaPods with Swift), a dependency manager for iOS, can create dependencies only as dynamic frameworks when working with Swift projects. Dynamic frameworks are supported since iOS 8.0. Therefore, it is impossible to use CocoaPods with a Swift project targeting an iOS version prior to 8.0. Without using CocoaPods it is required to copy third-party libraries to the project, which is inconvenient and time-consuming.

With Objective-C as a base project language it is still possible to use Swift third-party dependencies and to support an iOS version prior to 8.0 by copying all the library source code directly to the project.

While Objective-C has been chosen for building VPages, the same concepts can be applied to Swift as well. However, the implementation would be different, as a lot of new features, such as protocol extensions, change the way the framework can be designed.

2.1 UIKit and UIView

The UIKit framework provides the crucial infrastructure needed to construct and manage iOS apps. This framework provides the window and view architecture needed to manage an app's user interface, the event handling infrastructure needed to respond to user input, and the app model needed to drive the main run loop and interact with the system. The UIView class defines a rectangular area on the screen and the interfaces for managing the content in that area. At runtime, a view object handles the rendering of any content in its area and also handles any interactions with that content. Views can embed other views and create sophisticated visual hierarchies. This creates a parent-child relationship between the embedded view (known as the subview) and the parent view doing the embedding (known as the superview) (iOS Developer Library, 2016).

Manipulations to the application's user interface must occur on the main thread. Thus, you the methods of the UIView class should always be called from the code running in the main thread. The only time this may not be strictly necessary is when creating the view object itself but all the other manipulations should occur on the main thread (iOS Developer Library, 2016).

This limitation has no impact for the majority of different scenarios. For example, iOS UITableView is designed in a way to successfully overcome main-thread-only UIView manipulation by reusing its subviews.

2.2 UITableView and UICollectionView

On iOS the Newsfeed interface is usually implemented with the help of UITableView or UICollectionView.

UITableView is designed for displaying and editing hierarchical lists of information. Table view displays information in a form of set of rows and sections with width equal to that of the table view and height different for each row and section. An example of the table view can be seen in the iOS Contacts app.

UICollectionView is used to display an ordered set of data items. The most common use of UICollectionView is similar to iOS default Photos app which uses it to display a list of the user's pictures. Any other layouts of sets of data can be implemented with the help of UICollectionView: grids, stacks, circular layouts, dynamically changing layouts (About iOS Collection Views).

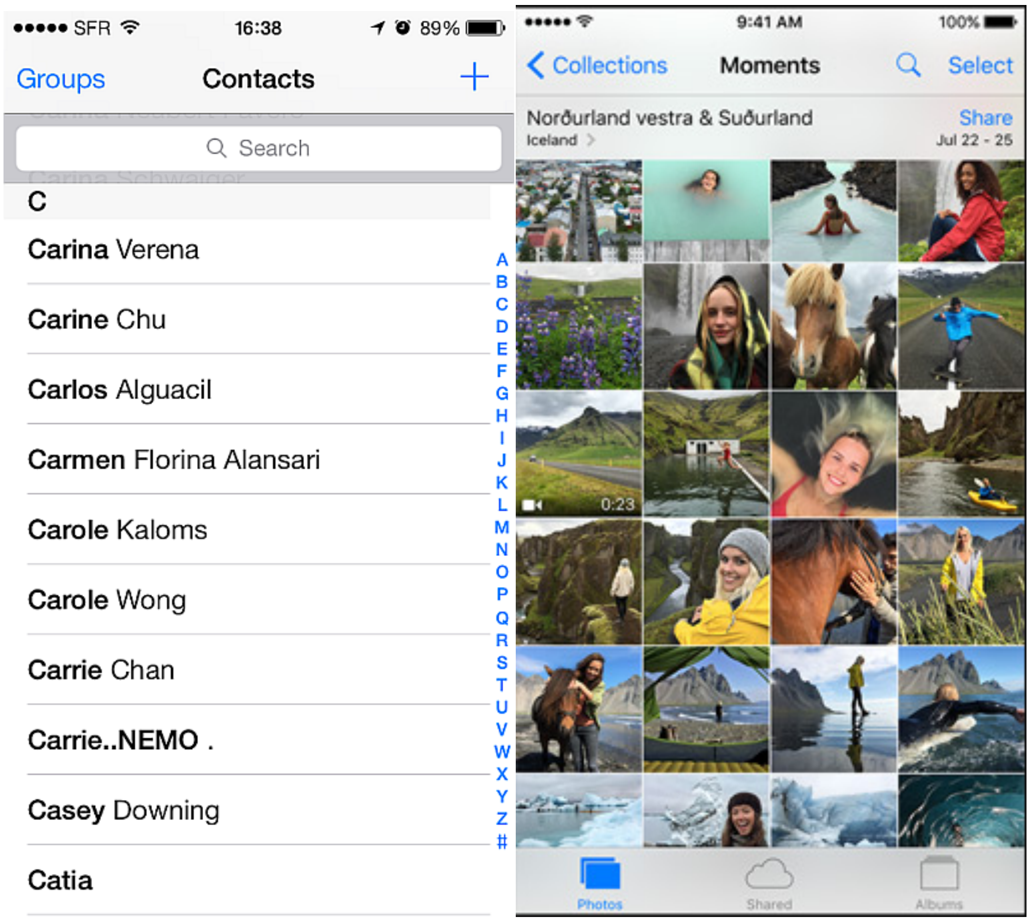


Figure 1. iOS Contacts (on the left) and Photos (on the right) apps

Both UITableView and UICollectionView were designed with high performance in mind and share some similarities in their API. For example, both implement object pool pattern (Object pool pattern): instead of creating the view for each item (or row), the table/collection view operates with a set of subviews just enough to fit the screen. While the user scrolls down, the subviews are updated with content and resized appropriately (Collection View Basics – Reusable Views Improve Performance). Schematic representation of the cell reuse process is shown on Figure 2.

Let's imagine, the user is about to scroll to the area where Cell7 should be placed. In that moment, Cell1 is already beyond the superview bounds and it can be deallocated to free memory. However, this does not happen. Instead, Cell1 changes its content to Cell7 and moves below Cell6. When the user scrolls further, the process repeats. No new subviews are created or added to the view hierarchy. They're simply resized and rearranged on their superview (which is table view). View hierarchy compositing is an expensive operation. By doing it only once (when UITableView instance is created), it is possible to achieve maximum responsiveness even with fastest scrolling speeds.

For tables with a large number of rows (for example, 1000) cell reuse helps reduce memory footprint dramatically. Another reason for implementing this strategy is that just changing the cell's content takes much less time than creating a cell and then building its view hierarchy if some cells share common design.

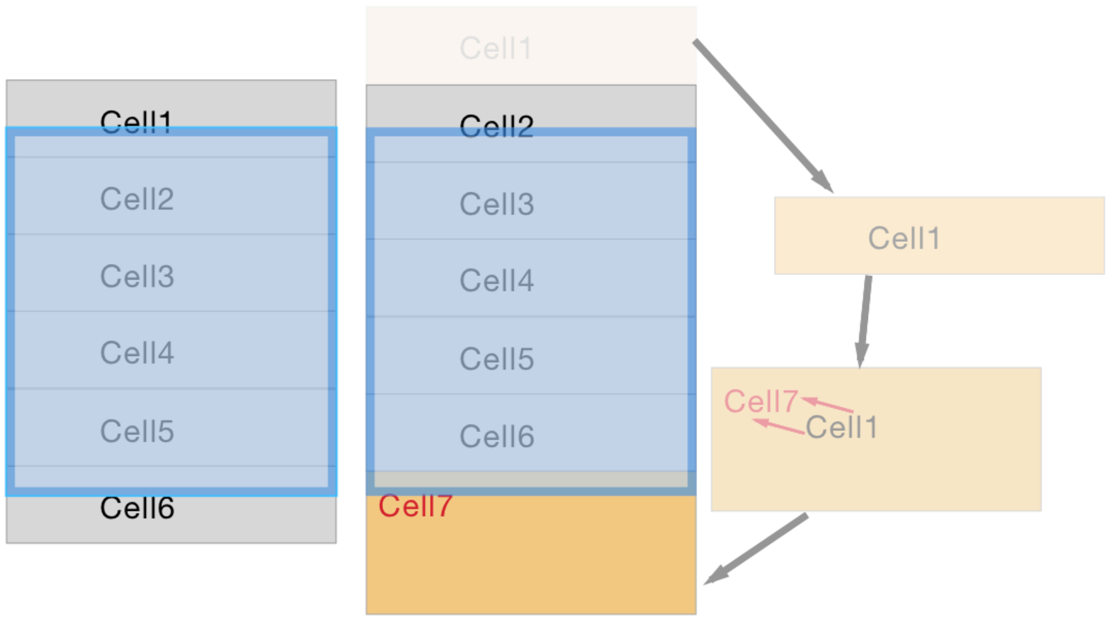


Figure 2. Schematic representation of UITableView reuses its cells

In order to look smooth, all animations should have the frame rate of 60 fps (frames per second), which means that the system has only 16 ms to render each frame. Scrolling is a special case where each screen update is a separate animation. Depending on scrolling velocity the table view must complete at least one new row minimum in 16 ms. However, sometimes the whole screen needs to be updated (when the user scrolls fast or when the table is reloaded). Any delays in updating new cells will lead to dropped frames and choppy animations. Anything less than 60 fps will be noticeable by the user, even if it is 58-59 fps. Therefore, new cell configuration should not only fit in 16 ms timeframe, but take as small time as possible. (WWDC 2012 session 228 "iOS App Performance. Graphics and Animations")

2.3 Delegation pattern in UITableView / UICollectionView

Both UITableView and UICollectionView (called later as the Reusable Views) use delegation pattern to display content. They ask another object, assigned as the *dataSource* which conforms to the UICollectionViewDataSource or UITableViewDatasource protocols, to provide cells to be displayed.

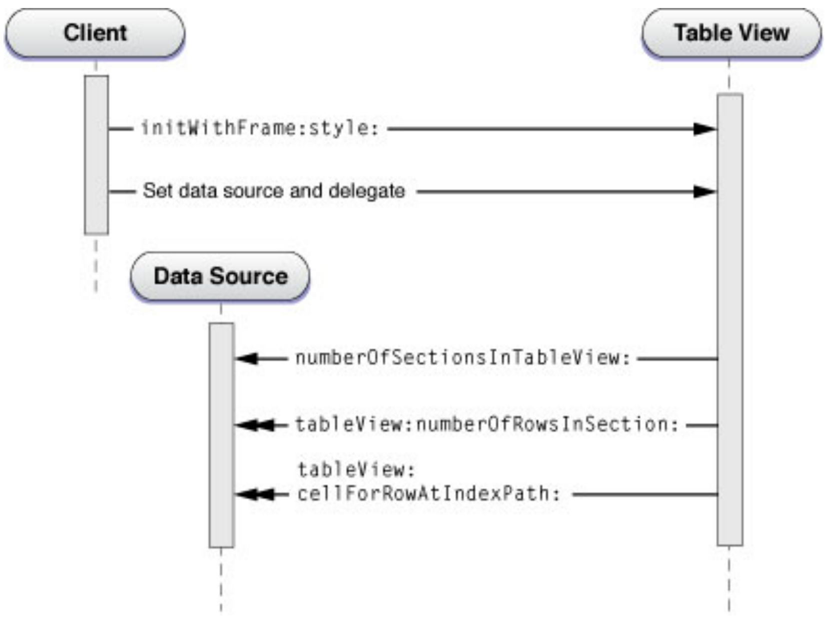

Client
Table View
initWithFrame:style:
Set data source and delegate
Data Source
numberOfSectionsInTableView:
tableView:numberOfRowsInSection:
tableView:
cellForRowAtIndexPath:

Figure 3. UITableView and its Data Source. Source: Apple Developer Portal.

Let's take a look at Figure 3, where the relationship between UITableView and its Data Source is shown. After the Table View has been configured, it asks the Data Source for number of sections in view. In case of non-zero number of sections, the Table View asks the Data Source for number of rows in each section, and then gets only the amount of cells needed to fill the screen. As user scrolls, the Table View asks its Data Source for more cells.

The Data Source should return the cell when asked, but where does it get the cell? It asks the Table View. The role of the Data Source is only to configure the cell, not create it. An example of the process is shown in Figure 4. When the cell is no longer required to be displayed, it enters the so-called reuse pool and could be returned by the Table View when the Data Source asks it for an empty cell.

Here is a step-by-step description of the process:

1. The Table View asks its Data Source for a cell.

2. The Data Source asks the Table View for an empty, unconfigured cell.

3. The Table View returns a free cell from the reuse pool, or creates a new one, if there are no cells.

4. The Data Source configures the cell with corresponding data. This step, for example, can be represented by passing a data model object to the cell.

5. The Data Source returns the configured cell to the Table View.

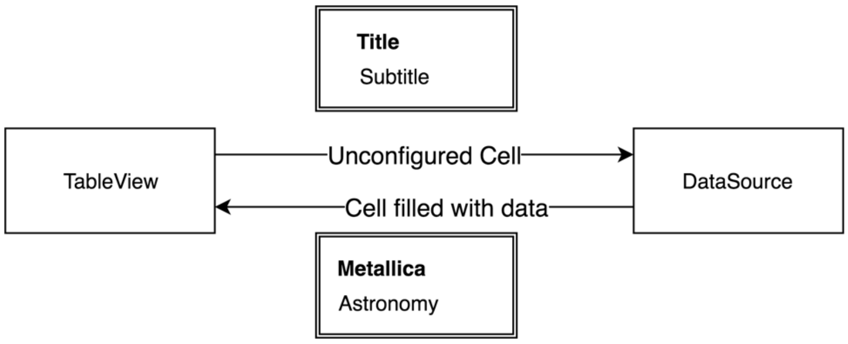


Figure 4. UITableView asking DataSource to configure the cell.

2.4 Challenges of building social networking mobile apps

Compared to the majority of mobile apps which solve only one problem, social networks consist of multiple modules. While some of them are independent of others, the majority of the modules are dependent. For example, showing a user profile requires downloading and processing not only user information, but her last photos, timeline, like/repost/comments statistics. Showing particular posts also requires comments to be shown. Because of rich functionality, social networking apps are much more complex to develop. Improper choice of app architecture may lead to tight binding between modules which leads to difficulties in adding new or changing existing functionality.

Social networks' users are accustomed to big freedom in expressing themselves. They can write texts, attach photos, audio files or documents as well as combinations of different attachments. It is possible to share posts to their own timeline with some comments (repost), saving the original post's authorship.

Such freedom leads to complex user interface structure. While in the majority of apps there is only one kind of user interface layout, in social networks the number of possible interface layouts is large, as the ways of multiple content type and quantities can be combined.

Let's compare Instagram (photo-oriented service) with the VK (general purpose social network) app.

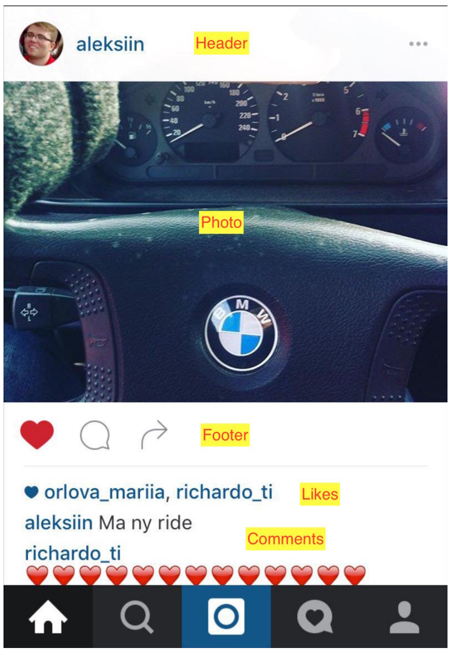


Figure 5. Instagram timeline

All Instagram posts share the same layout which is represented by header, followed by the content (photo/video), footer, likes and comments.

iOS UITableView is built for handling user interfaces exactly like this. By using the same view hierarchy, but they changing its content and size, it is possible to avoid expensive operations of adding or removing subviews which can be done only on main thread. While the Instagram app layout may be implemented in many ways, it is still possible to make it using the same approach as in Contacts app, for example. What is important, is that the layout stays the same all the time, no matter what the content is. This is different from the most social networks, where the layout depends on the content.

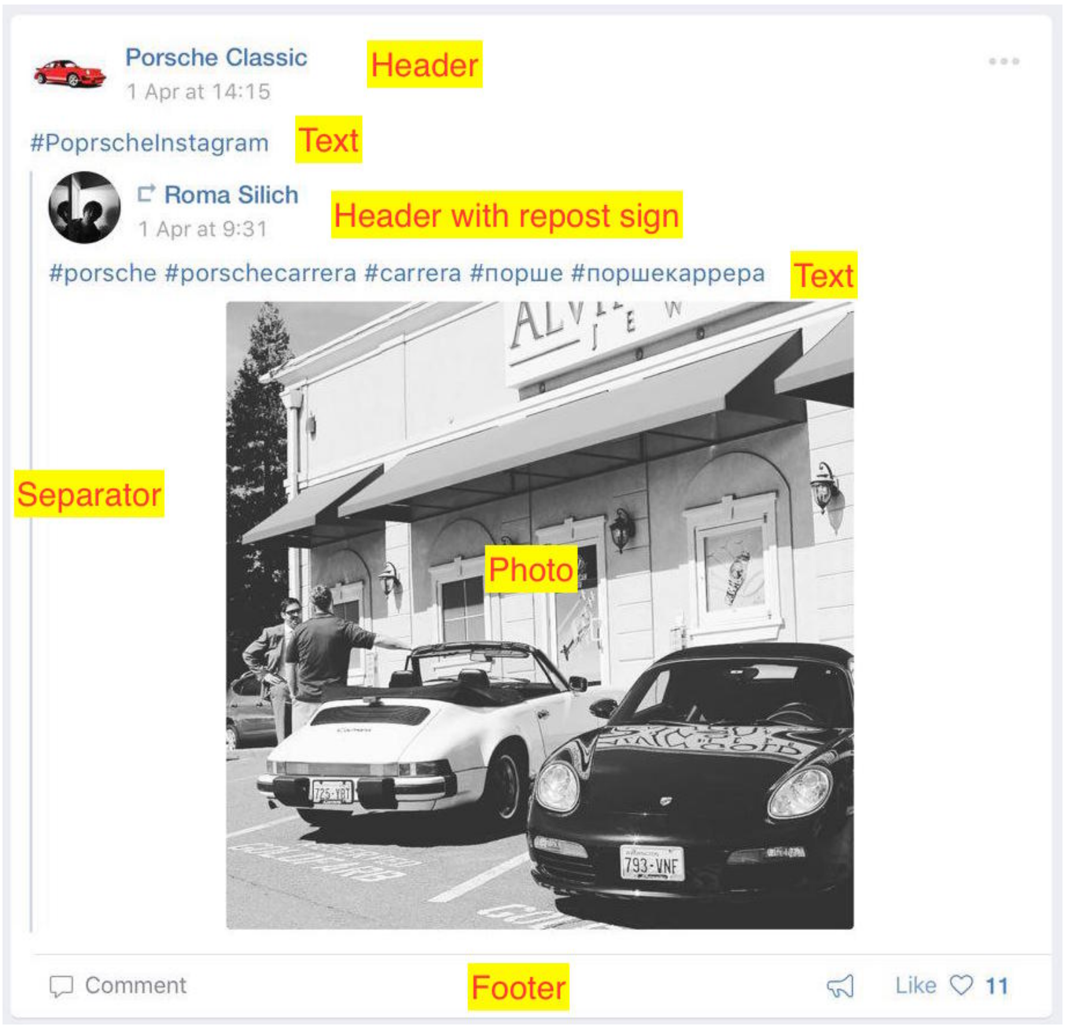


Figure 6. VK for iPad post

This is one of the examples of what VK post may look like. This post has two headers (one header for the original author, the other for the community that shared the post).

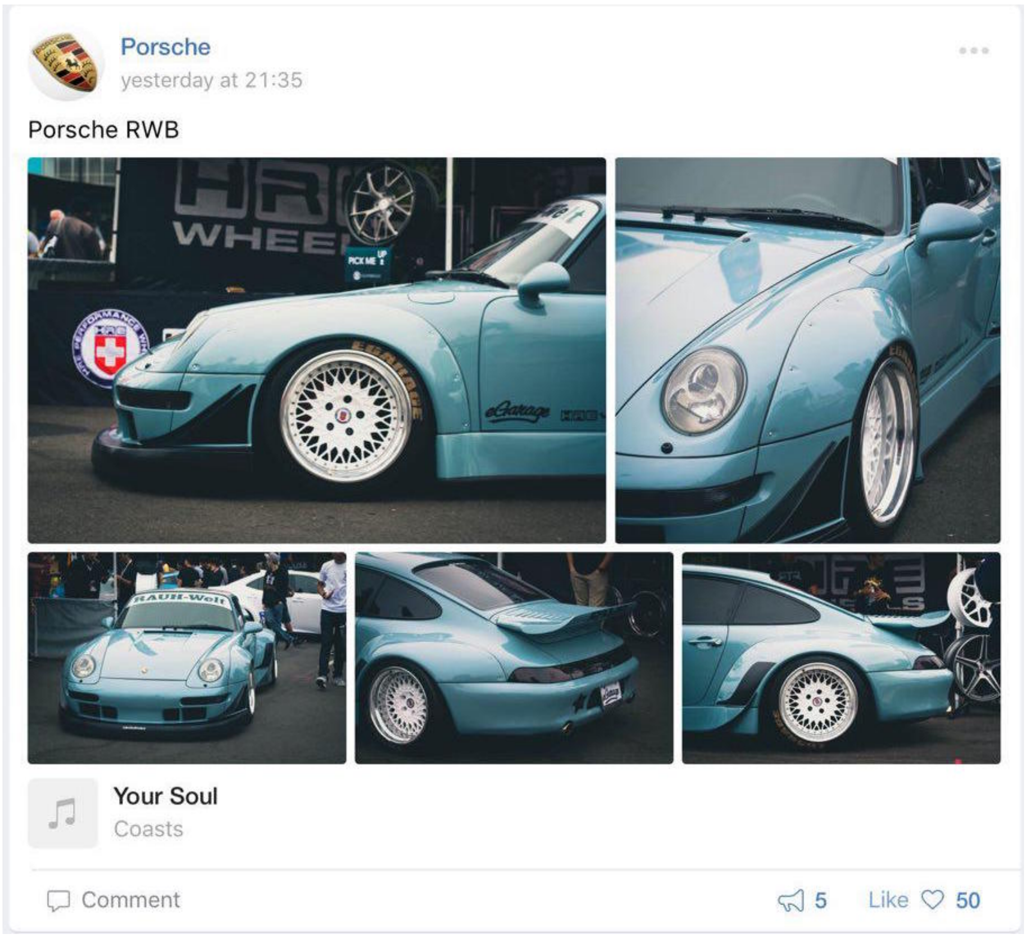


Figure 7. VK for iPad post

Another example of VK post is shown in Figure 7, where multiple images are combined in unique magazine-like layout. The layout depends on the number of images and their aspect ratio to minimize the total cropped area. An audio attachment is shown below the photos.

It is clear that UITableView cell reuse pattern cannot be directly applied in this case. A new cell cannot be composed just by using the old one, as some subviews may be missing. Such varying layouts will require changes in the view hierarchy (adding or removing subviews) while the user scrolls the timeline. If there are too many views are added or removed during one run loop cycle a main thread blocking is likely to occur resulting in UI freeze and frames being dropped.

One solution is to always generate a new view hierarchy, without reusing cells. This would significantly simplify programming, as there would be no intermediate stage (cell with old content) from which a new cell should originate.

However, this disrupts the normal behavior of UITableView, which is designed to reuse cells to achieve fast scrolling speeds. If view hierarchy re-created each time a cell is about to be shown, there is virtually no cell reuse.

The layout of each cell's content also becomes complicated as some elements of the cell may be missing. Another issue is that UITableView, when being reloaded, requests heights of each cell at least once. So, if a table consists of 100 posts, UITableView will iteratively request the height of each of these rows at one time. In simple cases (like Contacts app), where each cell has static height, this is not an issue at all. In complex cases it is impossible to know the height of the cell without performing the layout.

As UITableView requests the height of each cell at least one time (the first time when UITableView is reloaded (to calculate the relative size of the scrolling indicators) and the second time when a cell shows up on the screen (to change the size of the cell), it is required to calculate the heights of all cells and to cache these precalculated heights. In our case when the layout of each cell is unique, this means that not only height, but the whole layout should be at least calculated beforehand and at most, cached. This happens, because it is impossible to get the size of the post without calculating its layout: while there are elements with fixed size (post header, post footer), the size of other elements may depend on the content, for example, text, photos and other attachments.

Another complexity is added by the fact that after the view hierarchy has been composed, there is no information for layout code about subviews that are missing. Therefore, we end up with numerous checks for subviews present, which results in lots of monotonous "if-else" code. This code is difficult to manage and edit.

User interface state inconsistency across the screens is another challenge when building social networking apps. The issue arises when the user, for example, "likes" one post and then opens this post on the screen, different from the one she "liked". The "likes" counter should be updated on all the corresponding screens at the same time. The other case is a user changing her profile picture which should be updated in messages, timeline, feedback and other screens at the same time.

Because of the content varying among different posts a user has the possibility to do a number of different actions from each screen. These actions may lead to:

- Making a request to the server and updating UI on response (e.g. “liking” a post)
- Opening a popover dialog (“share”, showing the users who “liked” the post)
- Changing the layout of the post (“Show more” button)
- Opening profile page, either of a user or a community
- Opening detail view with comments
- Opening messages with the particular user
- Opening a photo viewer or video player
- Some other consequences that are not listed here

In a traditional approach each UIButton is configured with a selector (it is possible to treat it as “method to be called”) and the corresponding target (the instance on which the selector should be called). Usually the target is the cell itself which forwards method invocations by UIButton to the corresponding methods of the cell’s delegate which handles events. However, this approach may become tedious to implement, as it requires defining methods in both UICollectionViewCell / UITableViewCell subclass and the delegate which handles these events.

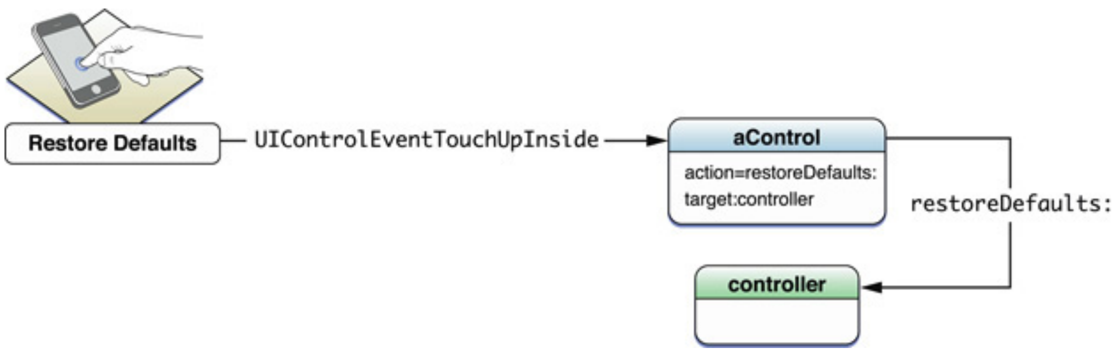


Figure 8. Target-Action design pattern (Apple developer portal)

3 PROPOSED SOLUTION

In this chapter I propose a solution for responsive scrollable user interfaces which will overcome the performance issues of UIKit and will be easier to work with. The solution is based on a few following concepts:

- As much work as possible is being performed asynchronously on a background thread for a chunks of objects, freeing up main thread.
- View hierarchy and layout is being described in a higher-level declarative manner, so that it is easy to change view layout without the need to change the actual layout code. All the view structure can be configured at one place, without editing both initialization and layout code.
- View hierarchy structure and layout is cached in descriptor-like objects instead of being recalculated each time the view is about to show up on a screen.
- As many of the views from a post that goes off-screen are reused for composing a new post minimizing the amount of time spent for initializing new instances of UIView.

3.1 Node as a prototype for UIView

In order to manipulate view hierarchy asynchronously, a thread-safe version of UIView is needed. Let's define the Node, which is NSObject subclass acting as thread-safe UIView prototype. Node stores all the information needed to compose view hierarchy. One node can be added on top of another (parent) node and positioned relatively to it. This is similar to UIKit concept of *subviews* and *superview*. Node occupies much less memory as opposed to UIView. Each UIView has corresponding CALayer, and normally, CALayer has a backing store which is a pixel bitmap. Node does not have any pixel bitmaps and, therefore, occupies only memory needed to store a node instance.

Class declaration of the Node is shown in Code 1. Each node has size and array of child nodes.

```
@interface Node : NSObject
@property (nonatomic, strong) NSArray <NodeChild *>* children;
@property (nonatomic, assign) CGSize size;

- (id)objectForKeyedSubscript:(NSString *)key;
- (void)setObject:(id)object forKeyedSubscript:(NSString *)key;
@end
```

Code 1. Node class declaration

Implementing *objectForKeyedSubscript* and *setObject forKeyedSubscript:* adds object subscripting functionality, so that Node is capable of storing any user information:

```
node[@"imageKey"] = [UIImage new];
UIImage *retrievedObject = node[@"imageKey"];
```

Code 2. Object subcripting in Node

One node class can be prototype for different UIView subclasses: UIImageView (needs an Image), UIButton (needs a target on which invoke action), RenderedTextLabel (needs a RenderedText, will be discussed later).

```
@interface NodeChild : NSObject
@property (nonatomic, assign) CGPoint origin;
@property (nonatomic, strong) Node *node;

- (instancetype)initWithNode:(Node *)node origin:(CGPoint)origin;

+ (instancetype)childWithNode:(Node *)node;
+ (instancetype)childWithNode:(Node *)node atPoint:(CGPoint)point;
@end
```

Code 3. NodeChild class declaration

NodeChild encapsulates Node, adding origin property. Origin defines, where to place top left corner of NodeChild on its parent node. It cannot be defined for Node, as it has no parent node (unless being encapsulated in NodeChild).

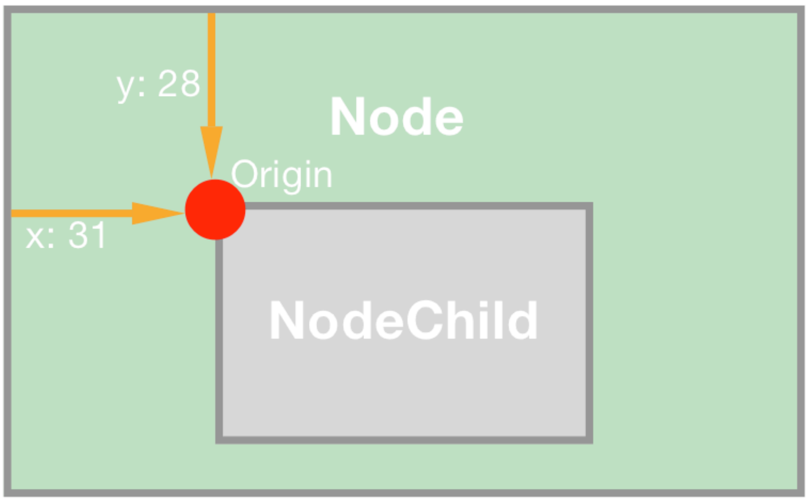


Figure 9. Origin points, where top left corner of NodeChild should be placed in a parent node

It is possible to compose an UIView hierarchy from corresponding Node hierarchy by invoking NodeAddToView function recursively:

```
void NodeAddToView(UIView *host, NSArray <NodeChild *> *nodes) {
for (NodeChild *child in nodes) {
     Node *node = child.node;
     UIView *view = [UIView new];
     view.frame = (CGRect){ .origin = child.origin, .size =
node.size };
    [host addSubview:view];

     NodeAddToView(view, node.children);
    }
}
```

Code 4. Example of composing view hierarchy from node hierarchy

3.2 Components for declarative layout definition

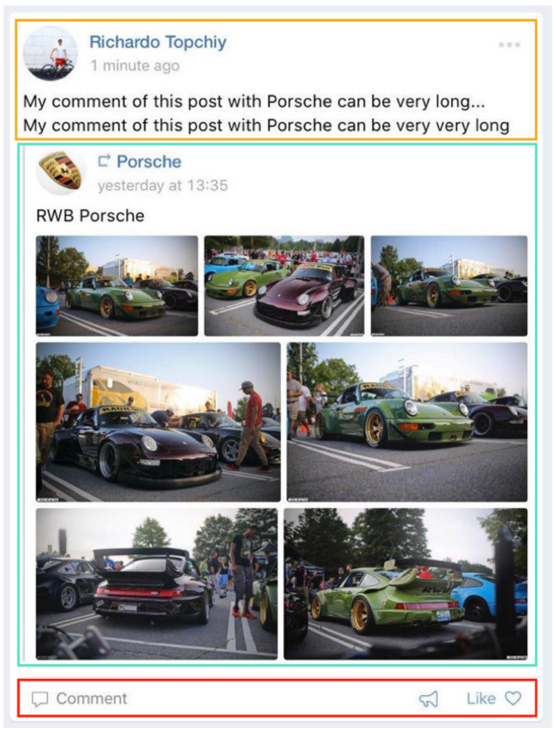
While working with UIKit, the developer has to program layout in an imperative way. View hierarchy should be composed during the initialization stage (preferably, only once) and then laid out each time the system calls *layoutSubviews.* This requires storing subviews as properties, or elements of the array and accessing them later at the layout stage. A simplified version of the code to accomplish this task is shown below. It is important to notice, that both layout calculation and user interface update take place on the main thread in the *layoutSubviews* method.

```
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        self.firstView = [UIView new];
        self.secondView = [UIImageView new];

        [self addSubview:self.firstView];
        [self addSubview:self.secondView];
    }
    return self;
}

- (void)layoutSubviews {
    [super layoutSubviews]
    CGRect frame = CGRectMake(0, 0, 50, 50);
    self.firstView.frame = frame;
    frame.origin.y += 50;
    self.secondView.frame = frame;
}
```

Code 5. Composing view hierarchy and defining layout in iOS

With the help of nodes it is possible to define view structure beforehand and use it at later stage. This solves some of the performance issues, as the layout calculation can be moved to background thread, freeing up the main thread only for interface updates. However, the layout still needs to be calculated manually – the same way as it is done for views. This requires dealing with size and position of each view (or node)

It was pointed out earlier, that VK layout is highly dynamic and working on this level of abstraction is not convenient (while it gives total control of the implementation). When looking closer at the layouts used in social networks it is possible to notice general patterns. While each post is unique, it can be broken down to few independent modules. For example, when a user shares post to his own timeline, the author's content is being placed first (orange) and the shared content follows (green). The post footer always placed last (red). The example of this layout is shown in Figure 10.



My comment of this post with Porsche can be very long...
My comment of this post with Porsche can be very very long

Figure 10. Example of post shared on a user's timeline

The structure of the post is independent of the content. It is enough to know just the size of each element to calculate layout for the whole post. Each next element is being placed after last element in the list.

Component is the class to define such independent module. The purpose of each component is to generate node hierarchy for any size. Component is responsible only for defining own nodes' hierarchy.

```
@interface ComponentContext : NSObject <NSCopying>
@property (nonatomic, assign) ComponentSize size;
@end

@interface Component : NSObject
@property (nonatomic, strong, readonly) id state;

- (instancetype)initWithState:(id)state;
- (__kindof Node *)nodeForState:(id)state context:(ComponentContext
*)context;
@end
```

Code 6. Component and ComponentContext class declarations

Component is a base class from which all other components should be subclassed. There is no need in instantiating an instance of *Component.* Therefore it could be implemented as an abstract class, but there are no abstract classes in Objective-C.

ComponentContext defines everything relevant for the application context. An instance of *ComponentContext* is being passed to each component. In this example *ComponentSize* is the only property found in *ComponentContext.* However, any other information can be shared this way: fonts, styles, app status, delegates responsible for handling events.

ComponentSize is a C-struct which is represented by two *CGSize* values. They are minimum and maximum sizes that define acceptable size range for the node that is being generated by the component. By providing the size range instead of fixed size it is possible to make the layout dynamic and let the elements occupy desired space.

```
typedef struct {
    CGSize min;
    CGSize max;
} ComponentSize;
```

Code 7. ComponentSize C-struct declaration

Given the size range, it is possible to generate the node:

```
- (__kindof Node *)nodeForState:(id)state context:(ComponentContext
*)context {
    Node *node = [Node new];
    node.size = CGSizeWithConstraints(CGSizeZero, context.size);

    node.component = self;
    node.componentContext = context;

    return node;
}
```

Code 8. Component node generation

Code 8 will create a node with zero size and without content. The *state* parameter is passed to the method, but remains unused.

State is any Objective-C object which is needed for the *Component* to generate a node. While *Component* defines layout, *state* defines what should be shown. Each component stores its state after initialization. The class of state is defined by particular component:

```
- (instancetype)initWithState:(id)state {
    self = [super init];
    if (self) {
        _state = state;
    }
    return self;
}
```

Code 9. Component initializer

For example, for *ImageComponent* the state is *UIImage:*

```
@interface ImageComponent : Component
@property (nonatomic, strong, readonly) UIImage *state;
+ (instancetype)image:(UIImage *)image;
@end
```

Code 10. ImageComponent class declaration

The dynamic nature of Objective-C allows downcasting property types in *Component* subclasses providing better type safety. For example, in *ImageComponent* the type of *state* is *UIImage*, not *id* as in the case of its superclass *Component* (see Code 6).

Overridden NodeForState:Context: method of ImageComponent is shown in Code 11. *ImageComponent* rescales the node, if the image size does not fit the provided range. Then it sets the image for a predefined key, so that it is possible to access it at a later stage.

```
static NSString *const kImageComponentNodeImageKey = @"Image";
// ...
- (__kindof Node *)nodeForState:(UIImage *)image
context:(ComponentContext *)context {
    __kindof Node *node = [super nodeForState:image
context:context];
    node.size = CGSizeWithConstraints(image.size, context.size);
    node[kImageComponentNodeImageKey] = image;
    return node;
}
```

Code 11. ImageComponent creates node

3.3 Optimizing node hierarchy to UIView hierarchy translation

It was pointed out earlier that the VK layout is highly dynamic and it will require changes in the view hierarchy of the post during scrolling. One of the solutions was to always build the view hierarchy from scratch. This will impact performance. In fact, this solution has been shown in Code 4 where new *UIView* is for each node.

Reusing some of the views from a previous post to compose a new one can significantly increase performance. When reusing views, two costly operations are being skipped: initializing a view and adding it to view hierarchy. Another way of improving performance is to minimize the total number of views being used. This can be achieved by flattening view hierarchy – eliminating views that are only used to group other subviews together. Grouping eases a developer's work as it helps treat a set of subviews as a single entity. For example, instead of setting each node's position, it would be possible to only set position of the parent (grouping) node. Positions of the children nodes would be adjusted implicitly. When working with nodes, it is still possible to use them for grouping, but when node hierarchy is about to be translated to views, the grouping nodes can be removed. Schematic representation of this process is shown in Figure 11 and the algorithm is shown in Code 12.

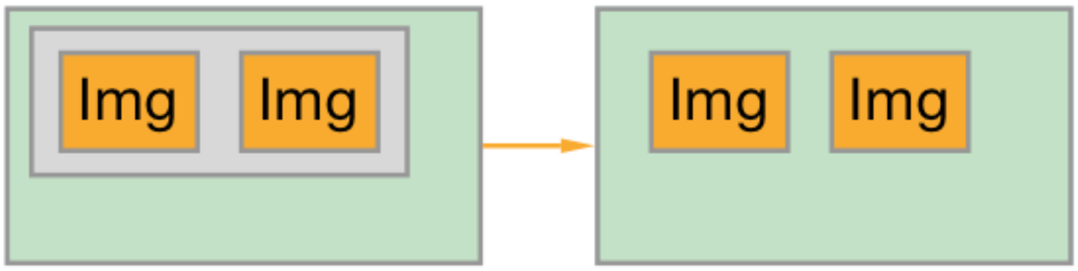


Figure 11. Simplification of node hierarchy

```
static void NodeSimplifyVisitor(Node *node, CGPoint origin, NSMutableArray
<NodeChild *> *acc) {
    if ([node isUtility]) {
        for (NodeChild *child in node.children) {
            NodeSimplifyVisitor(child.node, CGPointAddPoint(origin,
child.origin), acc);
        };
    } else {
        [acc addObject:[NodeChild childWithNode:node atPoint:origin]];
    }
}
```

Code 12. Simplification of node hierarchy

In Figure 11 the middlemost node (gray) does not have any content. It stores only two nodes with images. Such node is used only to group and keep the other nodes together and can be called “grouping” node. Therefore, it can be safely removed with two image nodes added to the parent (green) node. Their new relative positions can be calculated by summing the origin of the utility node (gray) with origins of each of the image nodes.

By recursively applying the function shown in Code 12, it is possible to flatten node hierarchy. This is the last step before it can be translated to views.

3.4 Updating view state from node

While many operations over nodes can be performed disregarding the data type it stores, at some point concrete *UIView* subclasses should be instantiated. For example, *UIImageView* to show image and *UILabel* to show text. Views share the way their position and size can be set. However, the content is updated differently for each of the *UIView* subclasses. *ComponentController* is the class responsible for initializing the corresponding instance of the *UIView* and updating it from the information stored in the node. The architecture of *ComponentController* is similar to *Component:* it acts as an abstract class. The class declaration of *ComponentController* is shown in Code 13.

```
@interface ComponentController : NSObject
@property (nonatomic, weak) __kindof UIView *view;

+ (__kindof UIView *)createView;
- (instancetype)initWithView:(__kindof UIView *)view;

- (void)updateWithNode:(Node *)node;
@end
```

Code 13. ComponentController class declaration

Subclasses override the *createView* and *updateWithNode* methods to match the interfaces of concrete *UIView* subclasses. For example, the implementation of *ImageComponentController* shown in Code 14. It returns *UIImageView* and sets the property *image* during view update process.

```
@implementation ImageComponentController
@dynamic view;

+ (__kindof UIView *)createView {
    return [UIImageView new];
}

- (void)updateWithNode:(Node *)node {
    self.view.image = node[kImageComponentNodeImageKey];
}

@end
```

Code 14. ImageComponentController implementation

3.5 ComponentController life cycle

iOS automatically manages the life cycle of each *UIView*. Superview holds strong reference to each of its subviews. When the superview is deallocating, all of its subviews are also removed from memory. As there is no need for *ComponentController* to exist without the corresponding *UIView,* it would be convenient to bind their life cycles to each other. This will eliminate the need for keeping track of each *ComponentController*, as their life cycle would be fully handled by the system.

This can be achieved by making *UIView* store strong reference to *ComponentController.* The *UIView* class is a part of iOS. Therefore, it is impossible just to add another property to class declaration. However, due to the dynamic nature of Objective-C, it is possible to achieve a desired effect by other means. The *ComponentController* life cycle can be linked to the *UIView* life cycle with the help of associated objects (see Code 14) (Objective-C Associated Objects). By using categories, it is possible to access *ComponentController,* as it would be a regular property of *UIView* (see Code 15).

```
@interface UIView (ComponentController)
@property (nonatomic, strong) ComponentController *_vk_componentController;
@end
```

Code 15. UIView+ComponentController category interface

```
static void const *kComponentControllerKey = &kComponentControllerKey;
@implementation UIView (ComponentController)

- (ComponentController *)_vk_componentController {
    return objc_getAssociatedObject(self, kComponentControllerKey);
}

- (void)set_vk_componentController:(ComponentController *)controller{
    objc_setAssociatedObject(self, kComponentControllerKey, controller,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

@end
```

Code 16. UIView+ComponentController category implementation

3.6 ReusePool

The *ReusePool* class serves two purposes:

- Keeping track of views available for reuse, providing view of a particular class when needed (object pool pattern)
- Removing views that are no longer needed from the hierarchy and passing them to another pool

The second feature of *ReusePool* makes it possible to create the cascades of pools with different performance levels. *ReusePool* is implemented using NSDictionary which stores an array for each of the keys (Figure 12).

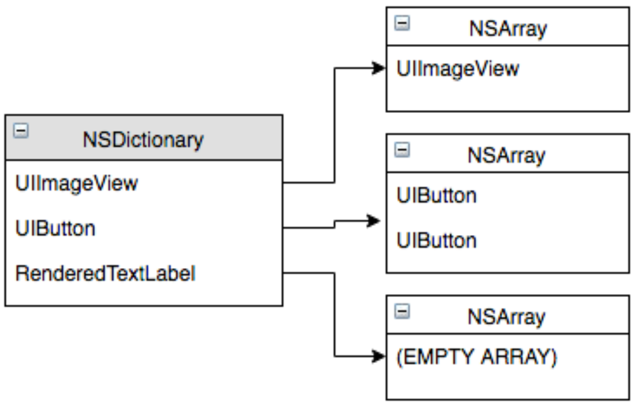


Figure 12. ReusePool architecture

```
@interface ReusePool : NSObject
- (__kindof UIView *)dequeueViewForKey:(NSString *)key;
- (void)setView:(__kindof UIView *)view forKey:(NSString *)key;

- (NSDictionary *)getDictionary;
- (void)addViewsFromDictionary:(NSDictionary *)dictionary;
@end
```

Code 17. ReusePool class declaration

When views are added from one pool to another, they are removed from the superview. This is achieved by iterating through all the items in reuse pool and invoking *removeFromSuperView* on each view (Code 18).

While there may be any number of pools in a cascade, two pools are used in this particular implementation. They are called Hot and Cold respectively. They differ by the state of the views they store:

- Views in Hot pool are already added as a subview.
- Views in Cold pool are initialized, but not added as a subview.

```
- (void)addViewsFromDictionary:(NSDictionary *)dictionary {
    NSArray *keys = [dictionary allKeys];
    for (NSString *key in keys) {
        NSArray *views = [dictionary valueForKey:key];
        for (UIView *view in views) {
            [view removeFromSuperview];
            [self setView:view forKey:key];
        }
    }
}
```

Code 18. addViewsFromDictionary: the method of ReusePool

3.7 Translating node hierarchy to view hierarchy with view reuse

Node hierarchy is translated to view hierarchy with the help of C-function shown in Code 20. It takes three input parameters: host view, where all the other views should be added; the array of *NodeChild* from which view hierarchy should be composed; and Cold reuse pool. The host view may contain some subviews, if it is reused, or it may be empty.

The example usage of the function is shown in Code 19.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:kWallPostCellIdentifier];

Node *node = self.postsNodes[indexPath.row];
NodeAddToViewReuse(cell, @[[NodeChild childWithNode:node]], self.coldPool);

return cell;
}
```

Code 19. Example usage in UITableView

The Cold pool is one per *UIViewController*, and the Hot pool is one per host view. Therefore, Cold pool is created when *UIViewController* is being initialized and Hot pool is created each time *UITableView* requests new cell. After cell is being composed and returned to the TableView,…..

Let’s go through the function step-by-step.

At first, new Hot reuse pool is created and all the subviews from host are added to it. The check for *ComponentController* is important, as there may be some subviews created by the system and incompatible with this framework (as they don’t have corresponding *ComponentController*).

```
ReusePool *hotPool = [ReusePool new];
for (UIView *subview in host.subviews) {
     if (subview._vk_componentController != nil) {
         [hotPool setView:subview
forKey:NSStringFromClass([subview._vk_componentController class])];
    }
}
```

Code 20. Adding subviews to "hot" pool

Then, node hierarchy simplification takes place (described in section 4.3):

```
NSMutableArray *children = [NSMutableArray array];
for (NodeChild *child in nodes) {
    NodeSimplifyVisitor(child.node, child.origin, children);
}
```

Code 21. Node hierarchy simplification

After node simplification, *children* array contains only nodes which need to be translated to views. Next step is to try to get view from any of the pools or create new one:

```
Class controllerClass = [[node.component class] controllerClass];
if (controllerClass) {
    UIView *view = [hotPool
dequeueViewForKey:NSStringFromClass(controllerClass)];
    if (!view) {
        view = [coldPool
dequeueViewForKey:NSStringFromClass(controllerClass)];
        if (!view) {
            view = [controllerClass createView];
         }
     }
```

Code 22. Requesting view from pool or lazily creating a new one

As the views in Hot pool are already added to the hierarchy, it is cheaper to take Hot view. If there are no more views, next step is to try get view from Cold pool. Only if that fails, controller is asked to instantiate view of particular type.

In case of view has been just created, but not reused, it does not yet have *ComponentController*. Code 23 will initialize it if it's missing.

```
ComponentController *controller = view._vk_componentController;
if (!controller) {
    controller = [(ComponentController *)[controllerClass alloc]
initWithView:view];
    }
```

Code 23. Lazy instantiation of ComponentController

When both view and controller are present, it is time to update view with new content and add it to view hierarchy. Then, the function *NodeAddToViewReuse* goes into recursion, with recently added view as a host.

```
[controller updateWithNode:node];
view.frame = (CGRect){ .origin = child.origin, .size = node.size };
[host addSubview:view];
NodeAddToViewReuse(view, node.children, coldPool);
```

Code 24. Updating view with node and composing view hierarchy

After view hierarchy has been composed, unused views are added to the Cold pool and removed from the hierarchy:

```
[coldPool addViewsFromDictionary:[hotPool getDictionary]];
```

Code 25. Adding views to the "cold" pool

Schematic representation of the view reuse process is shown in Figure 13. All of the elements of the old post are used in new post. Few other views are taken from the Cold pool. When Cold pool becomes empty, views are lazily instantiated. In this case no views are being added to the Cold pool, as there are no unused views left in the old post.

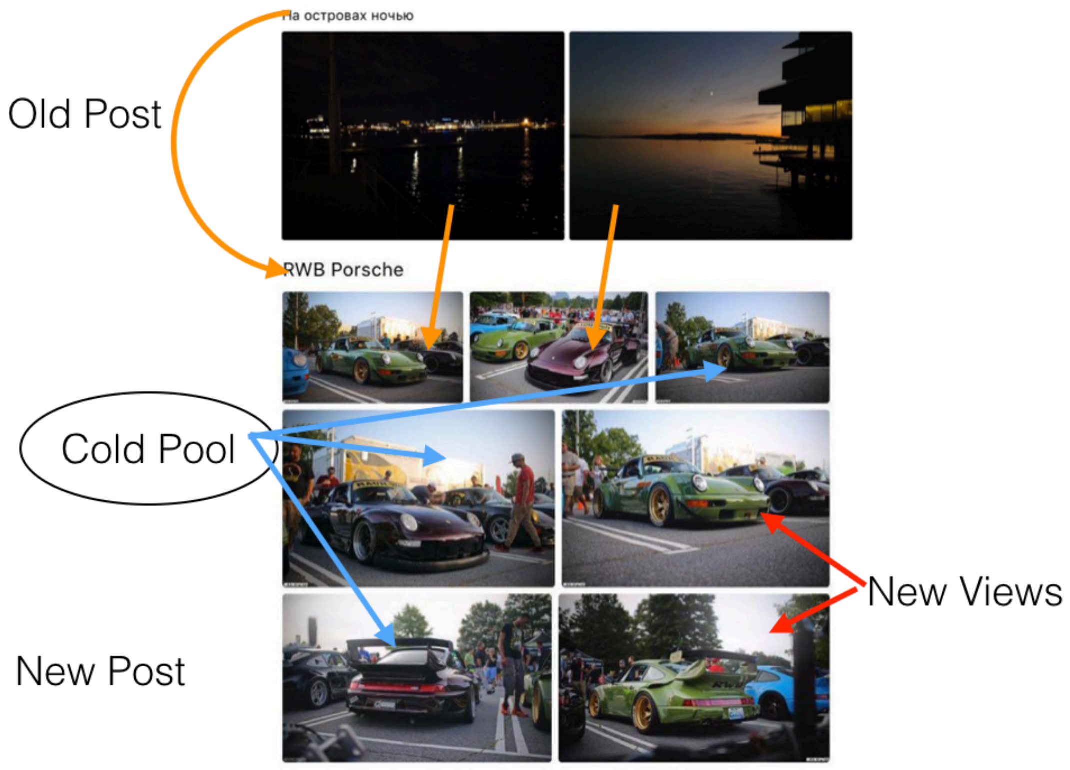


Figure 13. Schematic representation of the view reuse process

```
void NodeAddToViewReuse(UIView *host, NSArray <NodeChild *> *nodes,
ReusePool *coldPool) {
    ReusePool *hotPool = [ReusePool new];

    for (UIView *subview in host.subviews) {
        if (subview._vk_componentController != nil) {
            [hotPool setView:subview
forKey:NSStringFromClass([subview._vk_componentController class])];
        }
    }

    NSMutableArray *children = [NSMutableArray array];
    for (NodeChild *child in nodes) {
        NodeSimplifyVisitor(child.node, child.origin, children);
    }

    for (NodeChild *child in children) {
        Node *node = child.node;

        Class controllerClass = [[node.component class] controllerClass];
        if (controllerClass) {
            UIView *view = [hotPool
dequeueViewForKey:NSStringFromClass(controllerClass)];
            if (!view) {
                view = [coldPool
dequeueViewForKey:NSStringFromClass(controllerClass)];
                if (!view) {
                    view = [controllerClass createView];
                }
            }

            ComponentController *controller = view._vk_componentController;
            if (!controller) {
                controller = [(ComponentController *)[controllerClass
alloc] initWithView:view];
            }
            [controller updateWithNode:node];

            view.frame = (CGRect){ .origin = child.origin, .size =
node.size };

            [host addSubview:view];

            [coldPool addViewsFromDictionary:[hotPool getDictionary]];

            NodeAddToViewReuse(view, node.children, coldPool);
        }
    }
}
```

Code 26. NodeAddToViewReuse function

3.8 Tapable Component

The solution provided in this thesis would be incomplete if there would be no possibility of making any of the elements tapable. This can be done by placing *UIButton* underneath component that should react to taps or other gestures. This design allows even making a set of components tapable as if they were one.

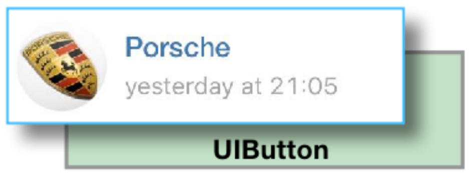


Figure 14. Tapable post header made of a group of components

Let's take a look, how *TapableComponent* works. It's state consists of two objects - *Component* and *eventName* (code 27). The *eventName* is related to the event handling and will be discussed later.

```
@interface TapableComponentState : NSObject
@property (nonatomic) Component *component;
@property (nonatomic) NSString *eventName;

+ (TapableComponentState *)component:(Component *)component
eventName:(NSString *)eventName;
@end
```

Code 27. TapableComponentState class declaration

TapableComponent requests node from the component stored in its state, resizes it and adds onto own node. This flexibility allows making defining complex tapable groups as shown in Figure 14.

```
- (__kindof Node *)nodeForState:(TapableComponentState *)state
context:(ComponentContext *)context {
    Component *component = state.component;
    Node *componentNode = [component nodeForContext:[context copy]];

    Node *node = [super nodeForState:component context:context];
    node.size = CGSizeNormalize(CGSizeWithConstraints(componentNode.size,
context.size));
    node.children = @[ [NodeChild childWithNode:componentNode] ];

    node[kTapableComponentNodeEventNameKey] = state.eventName;

    return node;
}
```

Code 28. TapableComponent nodeForState: context: method implementation

3.9 Event handling

I propose using URL-based event handling to maximize flexibility needed in social networks. *EventName*, shown in code 27, acts as an URL. It uniquely identifies resource to be shown to the user. There are no exact format of the URL, but examples are shown in code 29. Let's take the first string: it specifies *post_id* and *action* (like in this case), the second string just specifies what to open – user's *photos*. Last string has both, *photo_id* and *action*(open "comments" section and allow user to type a comment). This design allows to quickly introduce new services, actions and routes without adding new methods or changing existing classes.

```
post_id:123456/like
photos
photo_id:5555/comment
```

Code 29. URL-based routes example

The URL provides a lot of information about the resource which triggered the action, however it may not be enough to show interactive response for the user. In mobile apps user may interact with the same resource or interface element in many different ways. For example, a button can be tapped or tapped and held. Application response depends on the way user interacts with its content. Knowing frame (the size and the location on the screen) of the element which triggered the event may help in showing dynamic transitions, such as image zooming when presenting a photo viewer, or animations. I consider encapsulating this information along with URL in a separate object shown in Code 30, called *ComponentEvent*.

Some information about the event is being passed in a form of an URL while the other as a properties of the *ComponentEvent*. The reason for such architecture is to separate generic properties, applicable to every *ComponentEvent* from resource-specific ones. For example, as every event is triggered by the user's interaction with concrete interface element, it can have property *frame* or *sourceView*. However, not every element may have unique *id*, for example, therefore, such information should be encapsulated in the URL and handle by specific, resource-related handler. If needed, other properties can be added to the *ComponentEvent* class without the need to refactor other parts of the application. Such a flexibility leads to better developer's productivity and quicker introduction of new features.

To be able to receive events, an object must conform to *ComponentEventHandler* protocol, shown in **Code 31**.

```
@interface ComponentEvent : NSObject
@property (nonatomic) NSString *action;
@property (nonatomic) NSString *path;
@property (nonatomic) UIView *sourceView;

+ (ComponentEvent *)eventWithAction:(NSString *)action;
@end
```

Code 30. ComponentEvent class declaration

```
@protocol ComponentEventHandler
- (void)handleEvent:(ComponentEvent *)event;
@end
```

Code 31. ComponentEventHandler protocol declaration

Using protocols instead of concrete classes helps decouple modules of the framework. The event handler may be implemented in many different ways, the only requirement for it is to conform to the *ComponentEventHandler* protocol. Another level of flexibility originates from the fact that it is possible to use multiple event handlers. Each component may have own instance of class conforming to the *ComponentEventHandler* protocol, which can handle events in a different way from the rest. A particular component accesses its corresponding event handler with the help of the *ComponentContext: ComponentEventHandler* is just a property, as shown in Code 32. The particular implementation of the event handler goes beyond the scope of this thesis, and it is worth mentioning that it *ComponentEventHandler* protocol acts as a connection or loose coupling between two modules: user interface layout on one side and application navigation and routing on the other.

```
@interface ComponentContext : PMObject <NSCopying>
@property (nonatomic, weak) id <ComponentEventHandler> eventHandler;
@end
```

Code 32. Object conforming to ComponentEventHandler protocol is referenced by ComponentContext.

The URL is generated when particular instance of *TapableComponent* is created. Only *TapableComponent* can trigger events, hence only it truly needs an URL for initialization. However, other higher-level components may require part of the URL in their initializer only to pass it later to the corresponding *TapableComponent*. An example of the architecture is shown in Figure 15, its implementation in Code 33 and how the element is rendered on the screen in Figure 16.

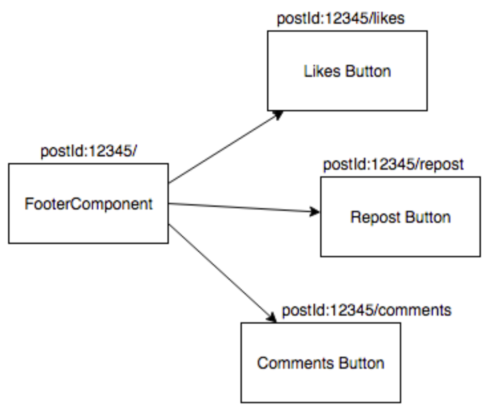


Figure 15. A set of complete URLs is created by adding a base string with unique string of particular button.

```
NSString *baseString = [NSString stringWithFormat:@"post_id:%@/",
[model.post.id stringValue]];

TapableComponentState *likeState = [TapableComponentState
component:[CounterButtonComponent state:likeStateC]
eventName:[baseString stringByAppendingString:@"like"]];

TapableComponentState *commentState = [TapableComponentState
component:[CounterButtonComponent state:commentStateC]
eventName:[baseString stringByAppendingString:@"comment"]];
```

Code 33. Creating a set of URL from base string and button identifier

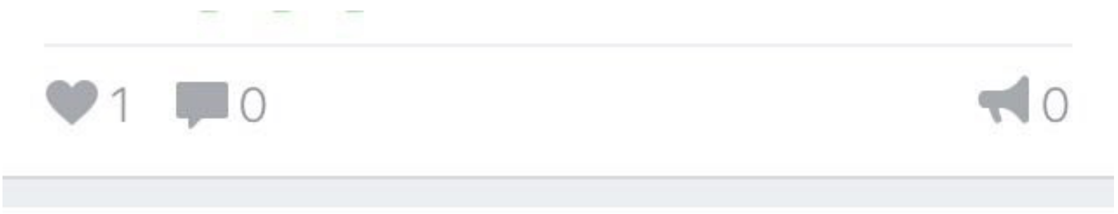


Figure 16. Footer component in the app.

3.10 Text Component

Text component is similar to other components that generate non-utility nodes (the ones that are translated to UIView). But it is different in a way that it uses custom UIView subclass to move text rendering to background thread. The interface is similar to other components and shown in Code 32. Text rendering takes place in *nodeForState: context:* method (Code 33). The text is being constrained only by width, allowing it to grow either infinitely or up to specified number of lines. After rendering, text may be set to *RenderedTextLabel*, a custom *UIView* subclass which is responsible for displaying *RenderedText.*

```
@interface TextComponentState : NSObject
@property (nonatomic, strong) NSAttributedString *string;
@property (nonatomic, assign) NSUInteger numberOfLines;
@property (nonatomic) NSNumber *lines;
@end

@interface TextComponent : Component
@property (nonatomic, strong, readonly) TextComponentState *state;

+ (instancetype)text:(NSAttributedString *)text;
+ (instancetype)text:(NSAttributedString *)text
numberOfLines:(NSUInteger)numberOfLines;
@end
```

Code 34. TextComponentState and TextComponent class declarations

```
- (__kindof Node *)nodeForState:(TextComponentState *)state
context:(ComponentContext *)context {
    Node *node = [super nodeForState:state context:context];

    RenderedText *renderedText = [RenderedText text:state.string
                                              width:context.size.max.width
                                      numberOfLines:state.numberOfLines];

    node.size = CGSizeNormalize(renderedText.size);
    node[kTextComponentNodeRenderedTextKey] = renderedText;
    return node;
}
```

Code 35. TextComponent nodeForState: context: metod

```
@implementation TextComponentController

+ (__kindof UIView *)createView {
    RenderedTextLabel *label = [RenderedTextLabel new];

    return label;
}

- (void)updateWithNode:(Node *)node {
    self.view.text = node[kTextComponentNodeRenderedTextKey];
}

@end
```

Code 36. TextComponentController class implementation

3.11 NetworkImage Component

Images play important role in social networking apps. They may be embedded in app bundle or downloaded from the Internet. *ImageComponent* may be used to display any *UIImage* disregarding its origin. However, the interface of *ImageComponent* is not so convenient, because it accepts only *UIImage* as an input and remote images are represented using URLs. Apart from this, few other differences of working with remote images are:

- They may show placeholder image while remote image is being downloaded.
- *ImageComponent* can infer size of its image from *UIImage* property, but this cannot be done for remote images until they have been downloaded. Therefore, *NetworkImageComponent* should accept setting predefined size or infer it from placeholder image.
- A lot of libraries already implement image downloading, caching and decoding. From developer's perspective, it would be easier to have possibility of choosing any of these libraries and linking them to *NetworkImageComponent.*

Given suggestions mentioned above, the interface for *NetworkImageComponent* shown in Code 35 is not very different from *ImageComponent*. Two properties are being added: *url* and *size.*

Size property is used for layout, when size of the remote image is not yet known. The size of the node is determined by following these rules:

1. Size property has the highest priority. If it is non-zero, this size is set for node.
2. If size is zero and placeholder image is set, then placeholder's size is set for node.
3. If size was not specified and there is no placeholder image, size of the node is zero.

The implementation of this logic is shown in Code 36.

```
@interface NetworkImageComponentState : NSObject
@property (nonatomic) NSURL *url;
@property (nonatomic) UIImage *placeholderImage;
@property (nonatomic) CGSize size;
@end

@interface NetworkImageComponent : Component
@property (nonatomic, strong, readonly) NetworkImageComponentState *state;

+ (instancetype)URL:(NSURL *)URL;
+ (instancetype)string:(NSString *)string placeholderImage:(UIImage
*)image;
@end
```

Code 37. NetworkImageComponentState and NetworkImageComponent class declarations

```
- (__kindof Node *)nodeForState:(NetworkImageComponentState *)state
context:(ComponentContext *)context {
    __kindof Node *node = [super nodeForState:state context:context];

    CGSize size = CGSizeZero;
    if (!CGSizeEqualToSize(self.state.size, CGSizeZero)) {
        size = CGSizeWithConstraints(self.state.size, context.size);
} else if (self.state.placeholderImage) {
        size = CGSizeWithConstraints(self.state.placeholderImage.size,
context.size);
    }

    node.size = size;
    node[kNetworkImageComponentNodeURLKey] = state.url;
    node[kNetworkImageComponentNodePlaceholderImageKey] =
state.placeholderImage;

    return node;
}
```

Code 38. NetworkImageComponent nodeForState: context: method

Image caching libraries on iOS usually implement their interface by extending *UIImageView* functionality. Using these libraries it is possible to set URL directly to image view and let the framework do the rest. I provide examples using Haneke library, however, any other library may be used to accomplish this task.

The interface for setting a remote image with Haneke is shown in Code 37.

```
    // Setting a remote image
    [imageView hnk_setImageFromURL:url];

    // Setting a local image
    [imageView hnk_setImageFromFile:path];

    // Setting an image manually. Requires you to provide a key.
    [imageView hnk_setImage:image withKey:key];
```

Code 39. Setting remote image with Haneke library

NetworkImageComponentController should invoke one of that methods when updating its view. An example of *UIImageView* update implementation is shown in code 38. Method *updateWithNode:* is a customization point, its implementation depends on chosen caching library.

```
- (void)updateWithNode:(Node *)node {
    [imageView hnk_setImageFromURL:node[kNetworkImageComponentNodeURLKey]
placeholder:node[kNetworkImageComponentNodePlaceholderImageKey]];
}
```

Code 40. NetworkImageComponentController updates its UIImageView

3.12 Utility components

Until now, all the components described (ImageComponent, NetworkImageComponent, TextComponent, TapableComponent) had their nodes being translated to views. In the following, these components are called “primary”, as they represent the content displayed. The utility components are different in a way that their nodes are removed during node hierarchy simplification process (4.3). Such nodes are called respectively, the utility nodes. Utility components can also be referred as “secondary” as they represent only layout. Secondary components are made only to simplify layout declaration, they don’t play any role from performance point of view.

As utility components do not display content, but delegate this responsibility to primary components, they often require *Component* instance in their initializer. Utility components working principle is to store other components in their *state* property. When asked to provide node, they delegate this request to the previously stored component and position nodes in a predefined layout. The following section describes different kinds of utility components in detail.

3.13 InsetComponent

InsetComponent (code 39) shows other component's content with some insets which are specified as *UIEdgeInsets.*

```
@interface InsetComponentState : NSObject
@property (nonatomic, strong) Component *component;
@property (nonatomic, assign) UIEdgeInsets insets;
@end

@interface InsetComponent : Component
@property (nonatomic, strong, readonly) InsetComponentState *state;
+ (instancetype)component:(Component *)component
withInsets:(UIEdgeInsets)insets;
@end
```

Code 41. InsetComponentState and InsetComponent class declaration

It's important to note, that any component can be embedded in *InsetComponent,* even another *InsetComponent.* Schematic representation of the node it generates is shown in Figure 17 and the code to generate node is shown in code 40. When *InsetComponent* is asked to return own node, at first, it ask embedded component for node; then, it positions it on own node with respect to insets. After node is being composed, it sets *utility* flag to *YES*, indicating, that this node is used only for layout.

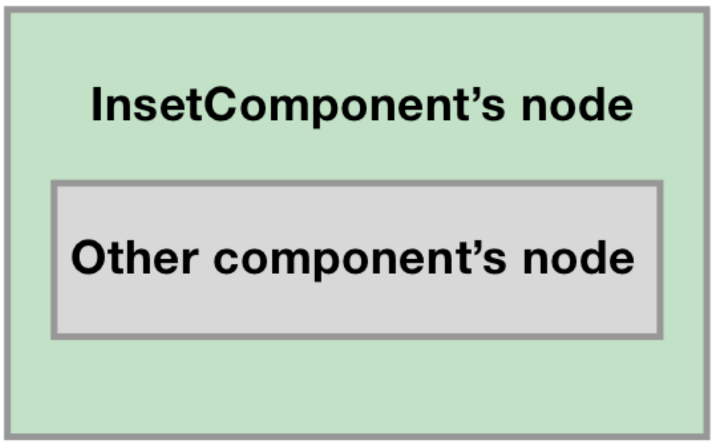


Figure 17. Node generated by InsetComponent

```
- (__kindof Node *)nodeForState:(InsetComponentState *)state
context:(ComponentContext *)context {
    const UIEdgeInsets insets = state.insets;
    const CGFloat hInsets = insets.left + insets.right;
    const CGFloat vInsets = insets.top + insets.bottom;

    ComponentSize childSize = context.size;
    childSize.min = CGSizeInset(childSize.min, hInsets, vInsets);
    childSize.max = CGSizeInset(childSize.max, hInsets, vInsets);

    ComponentContext *childContext = [context copy];
    childContext.size = childSize;

    Node *componentNode = [state.component nodeForContext:childContext];
    const CGSize size = CGSizeInset(componentNode.size, -hInsets, -
vInsets);

    Node *node = [super nodeForState:state context:context];
    node.size = CGSizeWithConstraints(CGSizeNormalize(size), context.size);
    node.children = @[
        [NodeChild childWithNode:componentNode
                         atPoint:CGPointMake(insets.left, insets.top)]
    ];
    node.utility = YES;

    return node;
}
```

Code 42. InsetComponent nodeForState: context: method

3.14 ListComponent

In social networking apps a big part of data is presented to user in a form of lists. List is a collection of items placed in series. Lists can be horizontal or vertical, from left to right or from right to left. Examples of lists have been already shown in this thesis in Figures 10 and 14. Lists can be embedded: one assembled list is an item of another, bigger list. With the help of embedded lists, it is possible to define complex layouts, such as shown in Figure 16.

ListComponent helps define lists in a declarative fashion. It takes generic parameters such as order of items, spacing and direction as input and automatically positions all the elements on the given space. List parameters are shown in Code 45. *ListComponent* does not require subclassing to define custom lists (as shown in Figure 16), instead, it uses composition and accepts any number of Objective-C blocks, defined in Code 44. These blocks are made only for convenience, as *ListComponent* stores all items' components in its state (see Code 46).

An example of ListComponent's syntax is shown in Code 43. The code can be split logically into two parts: the top part specifies list items in given order and the second part (*completion*) specifies list parameters, such as direction, alignment and spacing. Notice, how items can be added conditionally at a single point. This is different from traditional approach shown in Code 5 and described in section 3.1, where programmer had to keep track of all subviews being added and then perform layout at later stage. When using *ListComponent* it is only needed to specify whether to add element to the list or not. Items reordering with *ListComponent* is also simple: to change item's order it is needed to move corresponding code block up or down. Any change in code is being immediately reflected in the layout.

```
Component *nameComponent = [ListComponent
build:^(ListComponentBuilderAddComponentBlock addNameItem) {

    if ([source isLocked]) {
         addNameItem(friendsOnlyImageComponent);
    }

    if ([source isPinned]) {
         addNameItem(pinnedImageComponent);
    }

    } completion:^(ListComponentState *s) {
      s.direction = ListComponentDirectionHorizontal;
      s.verticalAlignment = ListComponentVerticalAlignmentMiddle;
      s.reversed = YES;
      s.interItemSpace = 2.f;
    }];
```

Code 43. Example of layout defined with ListComponent

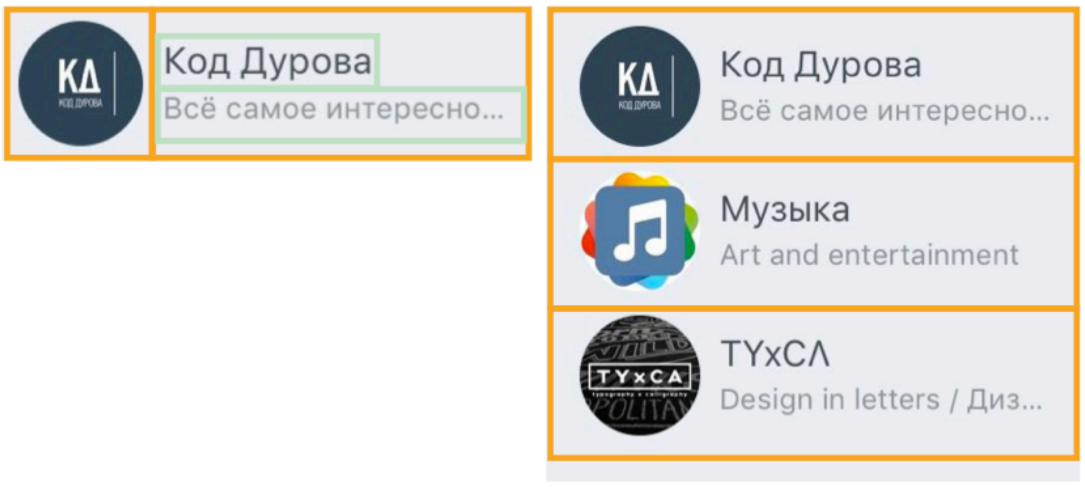


Figure 18. Layout being composed using embedded lists

```
typedef void(^ListComponentBuilderAddComponentBlock)(Component *);
typedef
void(^ListComponentBuilderBlock)(ListComponentBuilderAddComponentBlock);
```

Code 44. Blocks used in ListComponent

```
typedef NS_ENUM(NSInteger, ListComponentDirection) {
    ListComponentDirectionHorizontal,
    ListComponentDirectionVertical
};

typedef NS_ENUM(NSInteger, ListComponentHorizontalAlignment) {
    ListComponentHorizontalAlignmentLeft,
    ListComponentHorizontalAlignmentCenter,
    ListComponentHorizontalAlignmentRight
};

typedef NS_ENUM(NSInteger, ListComponentVerticalAlignment) {
    ListComponentVerticalAlignmentTop,
    ListComponentVerticalAlignmentMiddle,
    ListComponentVerticalAlignmentBottom
};
```

Code 45. List parameter options

```
@interface ListComponentState : NSObject
@property (nonatomic, strong) NSArray <Component *> *components;
@property (nonatomic, assign) ListComponentDirection direction;
@property (nonatomic, assign) ListComponentHorizontalAlignment
horizontalAlignment;
@property (nonatomic, assign) ListComponentVerticalAlignment
verticalAlignment;
@property (nonatomic, assign) BOOL reversed;
@property (nonatomic, assign) CGFloat interItemSpace;
@end
```

Code 46. ListComponentState class declaration

ListComponent is implemented in a similar way to other utility components. While the constructor is a bit different from other components (as it requires writing code generating components in blocks), it is made only for the developer’s convenience and does not alter ListComponent’s architecture. The constructor is shown in Code 47. It adds all the components specified in blocks to its state and then applies parameters, specified in the *completion* block.

```
+ (instancetype)build:(ListComponentBuilderBlock)buildBlock
completion:(void(^)(ListComponentState *s))completion {
    NSMutableArray *components = [NSMutableArray array];
    buildBlock(^(Component *component){
        if (component) {
            [components addObject:component];
        }
    });
    ListComponentState *state = [ListComponentState new];
    state.components = [components copy];

    if (completion) {
        completion(state);
    }

    return [self list:state];
}
```

Code 47. ListComponent convenience constructor

In the *ListComponent* node the generation process can be divided into three stages:

1. Node generation: stored components are asked to provide a node. Nodes are collected in an array.

2. Layout: *origin* property is set for each node.

3. Alignment: each node usually has different size (horizontal and vertical), and therefore may be aligned in several ways: to the left, centered or to the right.

Let’s look at how each of these phases is implemented.

Stage 1: node generation

The essence of this stage is to ask each component for a node and add it to the array for future processing. *NSEnumerator* is used to iterate over the array of components. If the list is reversed, the enumeration starts from the array end. It is possible to change the order of the list just by switching *reversed* flag to YES or NO. The implementation of this logic is shown in Code 48.

```
  NSEnumerationOptions opts = 0;
  const BOOL reversed = state.reversed;
  if (reversed) {
    opts |= NSEnumerationReverse;
  }

  [state.components enumerateObjectsWithOptions:opts
                                  usingBlock:^(Component *_Nonnull
component, NSUInteger idx, BOOL *_Nonnull stop) {
```

Code 48. Setting options for NSEnumerator

The next step is to ask each component for a node. However, there is no need to get node from every component if the list cannot fit them all. By skipping components which will be placed beyond list bounds we may save extra CPU time and memory. This can be achieved by checking if the total size of all nodes still fits list maximum size. Depending on the list type (horizontal or vertical), available extra width or height should be checked, as shown in Code 49.

```
  switch (direction) {
    case ListComponentDirectionHorizontal: {
      if (childMaxSize.width <= space) {
        *stop = YES;
        return;
      }
      break;
    }

    case ListComponentDirectionVertical: {
      if (childMaxSize.height <= space) {
        *stop = YES;
        return;
      }
      break;
    }
  }
```

Code 49. Checking, if the node fits the list

If there is a space left, component is asked for a node. Then, one of the dimensions of node's size (width or height) is subtracted from the maximum space available, as shown in Code 50. It’s worth observing, that the space between two adjacent list items is also subtracted during each iteration.

```
ComponentContext *childContext = [context copy];
childContext.size = ComponentSizeMaxSize(childMaxSize);

Node *componentNode = [component nodeForContext:childContext];

switch (direction) {
  case ListComponentDirectionHorizontal: {
    childMaxSize.width -= space + componentNode.size.width;
    break;
  }

  case ListComponentDirectionVertical: {
    childMaxSize.height -= space + componentNode.size.height;
    break;
  }
}
```

Code 50. Asking stored component for a node and subtracting its dimensions from the size available

This phase is finished with adding a node either to the tail or to the head of the array depending on whether the list is reversed or not. The implementation is shown in Code 51. It’s worth to be mentioned, that the type added to the array is *NodeChild,* not *Node.* During the next stage (layout), nodes have to be positioned on parent node. Therefore, they should be encapsulated into *NodeChild* and have the *origin* property.

```
NodeChild *nodeChild = [NodeChild childWithNode:componentNode];
if (reversed) {
  [children insertObject:nodeChild atIndex:0];
} else {
  [children addObject:nodeChild];
}
```

Code 51. Adding node to the array depending on list direction

Stage 2: layout

At this stage we have an array with properly sized nodes, but their origins are all set to zero. The goal of the layout stage is to place one node after another keeping inter-item spacing between them. Illustratively process is shown in Figure 19 and its programmatic implementation in Code 52. As at the stage of node generation, the process can take place in two different ways, depending on the list direction. The total size of the list is also calculated.

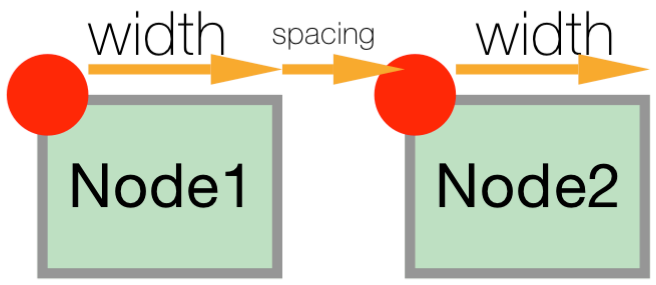


Figure 19. Layout of two nodes in a horizontal list

```
for (NodeChild *child in children) {
  switch (direction) {
    case ListComponentDirectionHorizontal: {
      if (listSize.width > 0) {
        listSize.width += space;
      }

      child.origin = CGPointMake(listSize.width, 0);

      listSize.width += child.node.size.width;
      listSize.height = MAX(listSize.height, child.node.size.height);
      break;
    }

    case ListComponentDirectionVertical: {
      if (listSize.height > 0) {
        listSize.height += space;
      }

      child.origin = CGPointMake(0, listSize.height);

      listSize.height += child.node.size.height;
      listSize.width = MAX(listSize.width, child.node.size.width);
      break;
    }
  }
}
```

Code 52. Implementation of layout stage in code

Stage 3: alignment

While the layout process defines how nodes should be placed in the list’s direction it is not responsible for arranging them in a secondary direction. Code 53 shows, how vertical origin is set to zero for horizontal lists. The goal of the alignment stage is to arrange the list items in the secondary direction and to define where the list should start from. If all of the items are equal in their secondary dimension, there is no need for the alignment stage. However, most of the time, lists are composed of items of different sizes. An example of vertical alignment options is shown in Figure 20.

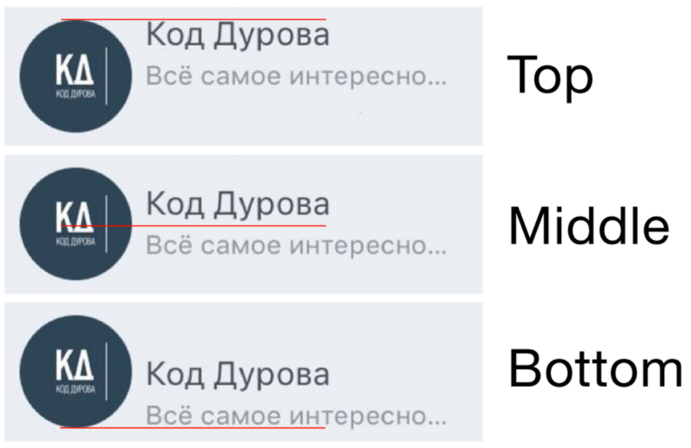


Figure 20. Vertical alignment options with examples of layout in horizontal list

If the size of all list items is smaller than the total space available for the list, there may be different options of how the group of items may be placed. This is defined by alignment in the list direction (e.g. horizontal for horizontal lists) and shown in Figure 19.

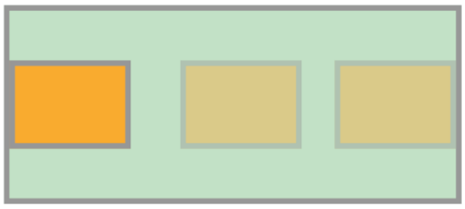


Figure 21. Possible horizontal alignment options in horizontal list

In code, alignment is implemented for both the horizontal and the vertical lists, but for simplicity, I mention the code for horizontal list only. Vertical list alignment code is very similar. The only difference is that height and width are swapped with each other.

Alignment algorithm shown in Code 53 works by calculating the total free space available in horizontal and vertical directions and then by adjusting each item's position according to the specified alignment type.

```
  const CGSize nodeSize = CGSizeWithConstraints(CGSizeNormalize(listSize),
context.size);

  const CGFloat dxTotal = nodeSize.width - listSize.width;
  const CGFloat dyTotal = nodeSize.height - listSize.height;

  const ListComponentHorizontalAlignment hAlign = state.horizontalAlignment;
  const ListComponentVerticalAlignment vAlign = state.verticalAlignment;

  for (NodeChild *child in children) {
    CGPoint origin = child.origin;
    const CGSize childNodeSize = child.node.size;

    switch (direction) {

      case ListComponentDirectionHorizontal:
      {
        switch (hAlign) {

          case ListComponentHorizontalAlignmentLeft:break;
          case ListComponentHorizontalAlignmentCenter: origin.x +=
ceilf(dxTotal/2); break;
          case ListComponentHorizontalAlignmentRight: origin.x +=
ceilf(dyTotal); break;
        }

        const CGFloat dy = nodeSize.height - childNodeSize.height;
        switch (vAlign) {

          case ListComponentVerticalAlignmentTop:break;
          case ListComponentVerticalAlignmentMiddle: origin.y += ceilf(dy/2)
break;
          case ListComponentVerticalAlignmentBottom: origin.y += dy;
break;
        }

        break;
      }
        // Code for vertical list is placed here
```

Code 53. Alignment in horizontal list

3.15 FutureComponent

Nodes, generated for different contexts usually differ in size and appearance, but the order of elements and logical layout stays the same. In some cases, layouts for large and small screen sizes are so different that it would be easier to use a separate component for each size range. However, the exact screen size may be known only at runtime.

Therefore, there should be a solution, how to pick the right component depending on the application context.

FutureComponent implements this behavior by storing a piece of code as an Objective-C block and executing it when the *ComponentContext* is provided. The main idea behind *FutureComponent* is to “look into the future” and be able to design layout for yet unknown size range. Block syntax being used is shown in Code 54: the context is being passed as an input parameter to the block and the *Component* is the output parameter.

Figure 22 and Code 55 illustrate FutureComponent's working principle: when asked to provide a node, at first it executes a stored block to get the component, and then asks the component to generate the node.

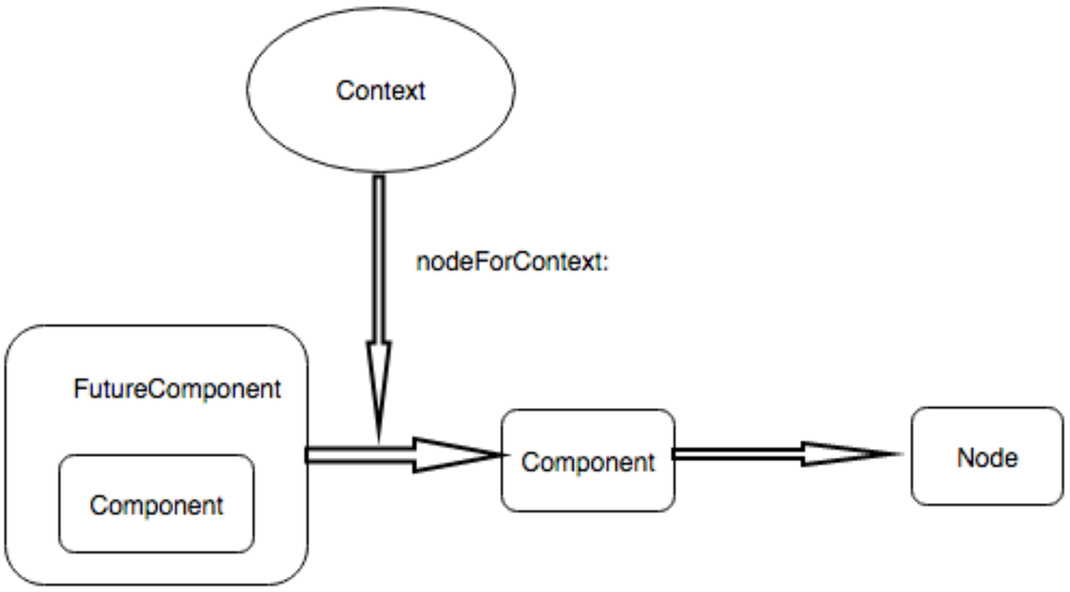


Figure 22. FutureComponent's principle of operation

```
typedef Component *(^FutureComponentBlock)(ComponentContext
*futureContext);

@interface FutureComponent <C: Component *> : Component
@property (nonatomic, strong) FutureComponentBlock state;
+ (instancetype)componentWithContext:(FutureComponentBlock)block;
- (C)componentWithContext:(ComponentContext *)context;
@end
```

Code 54. FutureComponent class declaration

```
- (Node *)nodeForState:(FutureComponentBlock)block
context:(ComponentContext *)context {
    Component *component = block(context);
    return [component nodeForContext:context];
}
```

Code 55. FutureComponent nodeForState: context: method

3.16 Higher-order components

All the components described earlier, primary and secondary, operate with generic concepts, such as images, texts and lists. Models used in social networking apps deal with concrete entities, such as post, message and user profile. Higher-order components fill the gap between generic components describing layout and content and models, describing the data only. They define the logical structure of the elements and usually require a model or a view model to be initialized. Higher-order components do not generate nodes at all, but use other components to compose a layout structure from data.

An example of higher-order component is a *SourceComponent*, shown in Figure 23. A *Source* is a view model, representing the source from which the post originates. No matter whether it is an user, a community, a public page or event, the layout stays the same. The place of the *SourceComponent* in the post is shown in Figure 24.

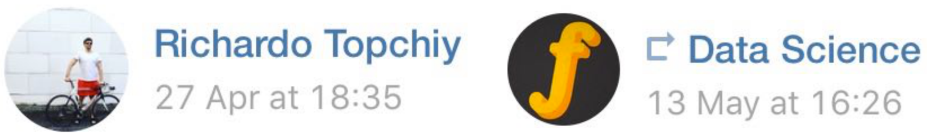


Figure 23. Examples of SourceComponent

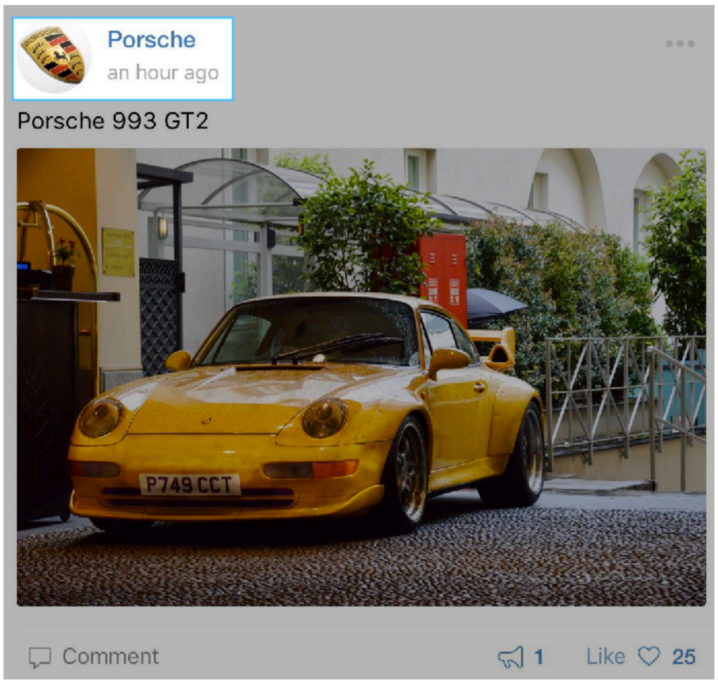


Figure 24. How SourceComponent is placed in the post

When looking at the Figure 23, we may notice that second source has an arrow placed on the left to the title, while the first picture does not have it. The arrow indicates the original author of the content, if the user shared (reposted) that content on her own timeline. All possible additional items in SourceComponent are shown in Figure 25.

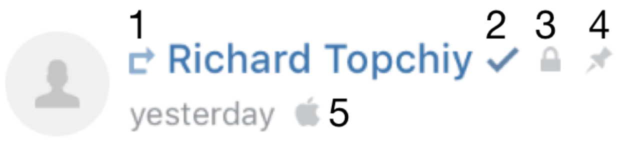


Figure 25. SourceComponent additional items: 1 - repost, 2 - verified, 3 - friends only, 4 - pinned post, 5 - application

Another example of higher-order component is *PhotoGrid.* VK uses an algorithm to arrange images in posts. The algorithm tries all the possible photo combinations and finds the one in which combined cropped area of the images is minimal. An example of a cropped image is shown in Figure 26. Having smaller cropped area gives the user the ability to see more even without the need to open the full picture.

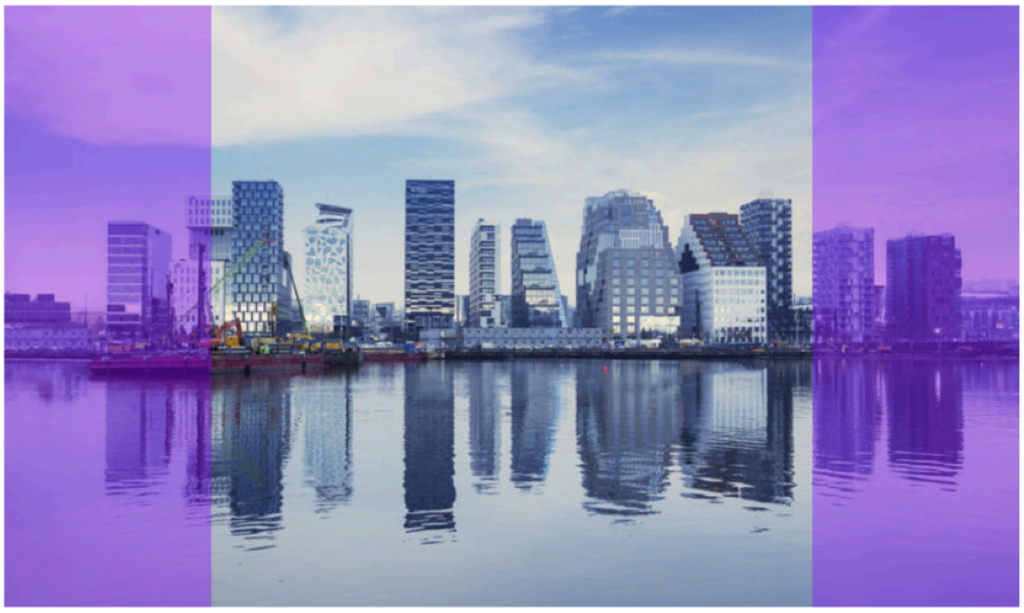


Figure 26. Rectangular image cropped to square. Cropped regions have violet tint. Minimizing cropped area helps user to see more content. Some buildings are not visible on the picture after cropping.

Examples of the layouts produced by the minimal crop area algorithm are shown in Figure 27.

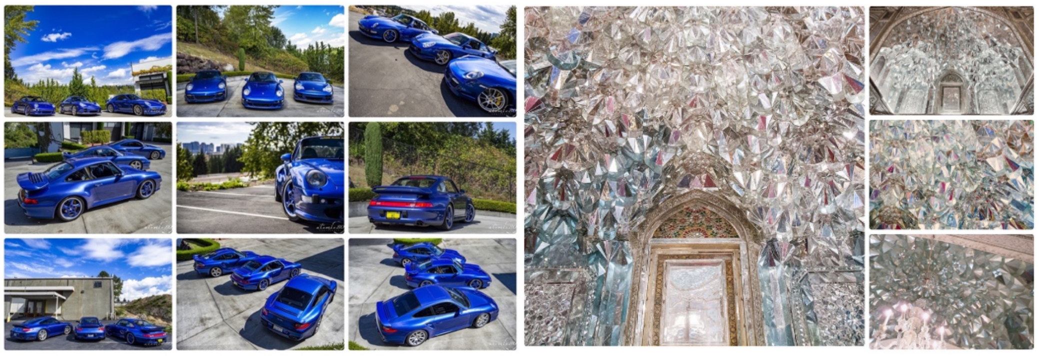


Figure 27. Layouts produced by the minimal crop area algorithm

I would like to focus not on the implementation of such an algorithm, but rather on how it connects together with the rest of the framework. At first, I would like to clarify few other details: all the images should be interactive, so that the user could tap on it triggering an event. Also, the grid should fully occupy one dimension (horizontal in Figure

27), but may be of different size in another dimension, because the aspect ratios of images are different. Therefore, all the elements in the grid should be encapsulated in *TapableComponent.*

```
@class ImageComponent;

@interface PhotoGridComponentState : NSObject
@property (nonatomic) NSArray<Component *> *components;
@property (nonatomic) NSArray <id<ThumbnailDescription>> *thumbnails;
@end

@interface ThumbnailDescription : NSObject <ThumbnailDescription>
@property (nonatomic, assign, readonly) CGFloat ratio;
@property (nonatomic, strong, readonly) NSIndexPath *indexPath;

- (instancetype)initWithRatio:(CGFloat)ratio indexPath:(NSIndexPath
*)indexPath;
+ (instancetype)descriptionWithRatio:(CGFloat)ratio indexPath:(NSIndexPath
*)indexPath;

@end

@interface PhotoGridComponent : Component
@property (nonatomic) PhotoGridComponentState *state;
+ (Component *)components:(NSArray<Component *> *)components
thumbnails:(NSArray <id<ThumbnailDescription>> *)thumbnails;
@end
```

Code 56. PhotoGridComponent class declaration.

In order to calculate image layout, the algorithm needs aspect ratio of each image, shown in Code 56. *ThumbnailDescription* class binds aspect ratio to the particular image by the *indexPath* property. *PhotoGridComponent* takes an array of the other components and an array of *ThumbnailDescription* (i.e. aspect ratios of those components) as an input. The initializer of the *PhotoGridComponent* doesn’t have any references to the *ImageComponent.* Without any modification, it is possible to use the same code to arrange other components, for example, videos.

Layout algorithm is purely abstract and operates with such concepts such as aspect ratio. Component layout system operates with subclasses of *Component (*which acts as a common interface along various components). *PhotoGridComponent* binds abstract layout algorithm to the component layout system, acting as intermediary between the two.

Let's look at how *PhotoGridComponent* creates its node. We already know, it has an array of other components and an array of their aspect ratios. The function *Enumerate-FramesForThumbnails* is the layout algorithm call: it takes an array of aspect ratios, maximal total size to fit the layout and a block (closure) to execute. The closure is called for every element providing its index to reference to some other object outside the closure's scope. In our case, such object is a component to ask for a node, and lay out then according to the algorithm's result. After we get an array of nodes, properly resized, we set them as children on *PhotoGridComponent's* node and return it. The *utility* flag is set to *YES* on the parent node, as it is used only to position other nodes and does not show any content by itself.

```
- (__kindof Node *)nodeForState:(PhotoGridComponentState *)state
context:(ComponentContext *)context {
    CGSize maxSize = CGSizeMake(context.size.max.width,
context.size.max.width);
 NSMutableArray *children = [NSMutableArray new];

    EnumerateFramesForThumbnails(self.state.thumbnails, maxSize,
^(NSUInteger idx, id<ThumbnailDescription> thumbnail, CGRect frame) {
        ComponentContext *childContext = [context copy];
        childContext.size = ComponentSizeMaxSize(frame.size);

        Component *component = [self.state.components objectAtIndex:idx];
        Node *node = [component nodeForContext:childContext];
        node.size = frame.size;
        NodeChild *child = [NodeChild childWithNode:node
atPoint:frame.origin];
        [children addObject:child];
    });

    Node *node = [super nodeForState:state context:context];
    node.children = [children copy];
    NodeChild *last = [children lastObject];
    node.size = CGSizeMake(last.origin.x + last.node.size.width,
last.origin.y + last.node.size.height);
    node.utility = YES;

    return node;
}
```

Code 57. PhotoGridComponent's nodeForState: context: method.

To use *PhotoGridComponent* with data models, it is required to create all the components which will actually display data, as shown in Code 58. At first, the image component is created and then placed into TapableComponent to make it intercept the user's touches. *PhotoGridComponents* is not responsible for the components creation, its only job is layout. Each component is just a building block and has responsibility over a limited scope.

```
NSMutableArray *components = [NSMutableArray new];
for (VKPhoto *photo in photoAttachments) {
    Component *component = [NetworkImageComponent string: photo.photo_604];
    TapableComponentState *state = [TapableComponentState 
component:component eventName:photo.photo_604];
    TapableComponent *button = [[TapableComponent alloc] 
initWithState:state];
    [components addObject:button];
}

if ([photoAttachments count] > 0) {
    buildBlock([PhotoGridComponent components:[components copy] 
thumbnails:[photoAttachments copy]]);
}
```

Code 58. Using PhotoGridComponent in PostComponent.

4 ARCHITECTURE OVERVIEW

In this section I will present the overview of the solution, without going into the details. As shown in Figure 28, the core modules of the framework are the Layout Declaration module and the Node to View Translation module. The former serves as a bridge between the View Models and the Components. A View Model represents information prepared and formatted to be shown. For example, it may include strings as “A day ago” instead of timestamp or date. The purpose of the Layout Declaration module is to make development and editing of the layout easier, less complex and quicker.

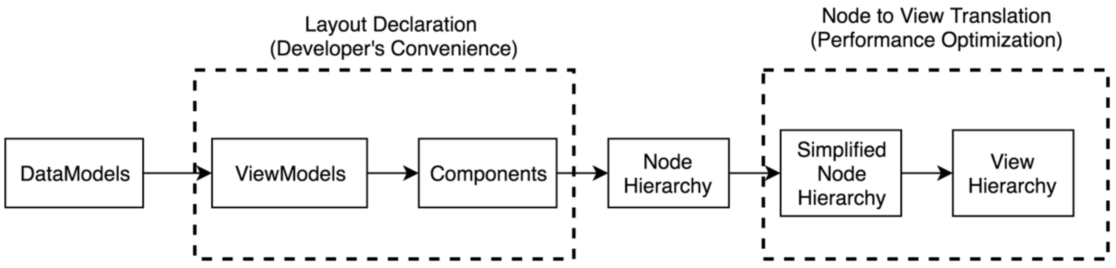


Figure 28. Architecture of the framework.

The Node to View Translation module exists with the purpose of maximizing scrolling performance. It takes simplified node hierarchy as an input and constructs view hierarchy by reusing existing views or creating new ones if necessary. Both modules are weakly coupled with the help of the Node objects. It is possible to completely change one of the modules without any change in another one. Therefore, any improvement in the module responsible for performance becomes available to every part of the app immediately. The rule works as well in the opposite direction: any new layout written using the Components works fast without the need of further fine-tuning and optimization. The only requirement for both modules is that they should exchange information using the Node objects. Any data should be encapsulated inside the Node.

5 VPAGES APP

The proposed solution was used by me together with Artem Konovalov to build a social networking app called VPages. The app is based on VK API and provides additional tools for community management. The app's target audience covers social media managers and community / page administrators. Their daily routine include content creation (posts, photos, videos, etc.), content moderation, advertising, responding to comments and messages on behalf of the community or a brand. Administrators may have one or more communities counting thousands of participants. The Web version of VK has a set of advanced tools aimed at helping administrators do their job, for example, analytics, advertisement settings, user ban settings. These and many other tools are missing from the VK app for iOS, forcing community managers to stick to the PC and making them unable to do any work while on the go. Moreover, a lot of actions don't even require a PC: answering a user's question, or posting a few photos can be done using a phone. There was definitely a place for an app specializing in community management.

To make the transition from VK for iOS and VPages (and vice versa) smooth, we decided to make the user interface resemble VK. The layout, buttons and their functions are the same in both apps, making it easier to learn additional functionality. The app is intended to be used together with VK for iOS: VK for personal communication and VPages for community management. Therefore, all the actions possible with VK for iOS should be possible with VPages, including complex navigation inside the app. For example, the user can tap another user's profile picture, then do it once again and again, and then tap on community header. Now the community page shows up on the screen. Nevertheless, there should be a possibility to go back and the app should keep all the other screens in memory.

The solution described in this thesis has been used to achieve three goals while building VPages:

1. Similarity of interface with VK for iOS: if the original app was built using the same concepts, the interface is easy to replicate. Moreover, new features will be implemented using similar concepts and tools. Therefore, they are more likely to fit the new product.

2. Reducing development complexity by using declarative approach to define layout instead of imperative: the overall level of complexity of the social networking apps is high, and it is a limiting factor for the introduction of new features. Any novelty requires testing not only the new feature, but a regression testing as well. Having a layout system built the way to ensure that there are no regressions will significantly improve development speed, or reduce the number of bugs keeping development speed unchanged. Layout framework that is built around the concept of interchangeable modules that are responsible for only one function make it easier to scale out – make more developers do work in parallel. The framework ensures that modules developed by different programmers will work together. The only rule to obey is to make modules conform to common protocol.

3. Maintaining an acceptable level of performance while respecting goal 2: usually, there is a possibility to fine-tune individual modules of the user interface, thus achieving greater performance. It's clear that by accounting specific constraints of the particular design, it is possible to write code on a lower level, making it perform better than a higher-level code. However, lower-level code is more difficult to write, it takes more time to develop and test. And the most important drawback is that lower-level code is extremely difficult to change or modify if some of the requirements change. It is important to notice, the requirements may change even without a developer's consent: new iOS versions may introduce features requiring the change of the current codebase.

Keeping all the three goals in mind and 2 and 3 in particular (as they are general, not application-specific), I consider that component-based layout system was a good choice for the app of such complexity.

6 EXAMPLES OF USING THE PROPOSED SOLUTION IN THE APP

Designing a framework to improve user experience and the developer's productivity is only one half of the solution. The other half is to use the framework in a way that leverages all its benefits. In this section I provide examples of how the framework is used in the app and the reasoning behind the decisions.

The use of the intermediate objects (nodes) to manipulate layout and not *UIView* subclasses gained a significant performance advantage – a possibility of computing layout asynchronously, i.e. without blocking the main thread. "Asynchronously" here means that the whole layout stage can be offloaded to the background thread, while the main thread is kept as free as possible, which is crucial for the app to be responsive.

Asynchronous layout processing leads to the possibility of doing speculative or *spec* work. Spec work can be described as doing computations, or downloading data that may not be needed later at all. However, if the results of the spec work are needed later at some point, it will improve performance. There would be the possibility to use already precomputed and cached data instead of performing all the computations from the very beginning. It is clear that performance leverage can be taken only if the spec work is done off the main thread. Loading the main thread with spec work will slow down the user interface, as it would compete with higher priority work (such as displaying the interface).

Most of the apps which have an interface similar to Newsfeed use paginated loading. The content is loaded in *pages* - ranges of items, for example, timeline posts in predefined date range. Once user has scrolled through one page, the other starts downloading. The process continues until there are no more content to load. Starting to load next page while user hasn't scrolled through the current one is a kind of speculative work: the network and CPU resources can be wasted, if user decides not to continue scrolling the feed. However, if the user continues to scroll, there would be no delays before new content appears, as it has been already downloaded. Speculative resource downloading improves perceived performance: resources are shown immediately, without network delays. The performance improvement is only perceived, as the amount of fps device outputs stays the same. Spec layout improves real performance, as it frees the main thread, leaving it only to operations related to compositing the view hierarchy. By using both, spec resource download and spec layout, we could provide better user experience. Let’s look, how these two techniques are implemented in code.

The two classes responsible for information loading and display are *Controller* and *Loader.* Every app screen utilizing the component layout system has these two classes. The *Controller* class is a subclass of *UIViewController,* and therefore, its lifecycle is managed by the system. It holds a strong reference to the corresponding *Loader* which is responsible for all the network operations, data preprocessing and spec work. The *Controller* class’ responsibility is to act as a bridge between the user interface (front end) and the services available (back end), and to tie all the components of the app together.

An essential part of *PMWallController* interface is shown in Code 59. It can be initialized either with an *ownerId* (a social network user, a person), or with a *groupId* (a community or a page). *PMWallController* holds strong reference to the *PMWallLoader.* An essential part of PMWallLoader is shown in Code 60.

```
@interface PMWallController : PMBaseVC <PMPostComposerDelegate>

- (instancetype)initWithOwnerId:(NSNumber *)ownerId;
- (instancetype)initWithGroupID:(NSNumber *)groupId;

@property (nonatomic) PMWallLoader *loader;

@end
```

Code 59. Highlights of PMWallController class interface

```
@interface PMWallLoader : NSObject
@property (nonatomic, weak) id <PMWallLoaderDelegate> delegate;
@property (nonatomic) ComponentContext *ctx;
@property (nonatomic) NSNumber *ownerId;
@property (nonatomic) NSNumber *postId;
@property (nonatomic) NSMutableArray<PMCommentModel *> *commentModels;
@property (nonatomic) NSMutableArray<PMPostModel *> *postModels;

- (instancetype)initWithOwnerId:(NSNumber *)ownerId;
- (instancetype)initWithOwnerId:(NSNumber *)ownerId andPostID:(NSNumber *)postID;

- (void)loadNextposts;
- (void)loadNextComments;

- (void)loadBatchPostsAndInfoFor:(PMWallLoaderType)loaderType;
- (void)loadPostForRequestStr:(NSString *)string;
- (void)loadPostAndCommentsForReqStr:(NSString *)reqStr;
- (void)likeComment:(PMCommentModel *)model;
- (void)likePost:(PMPostModel *)model;
- (void)refreshData;

- (void)deleteAllPostsforOwner:(NSNumber *)owner ids:(NSString *)ids;
- (void)deleteBatchIds:(NSArray <NSString *> *)ids forOwner:(NSNumber *)owner
andType:(PMWallLoaderDeleteAllType)type;

-(void)deleteAllCommentsForOwner:(NSNumber *)owner ids:(NSString *)ids;

@end
```

Code 60. Highlights of PMWallLoader class interface.

```
@protocol PMWallLoaderDelegate
@optional
- (void)nodesDidLoad:(NSArray<Node *> *)nodes;
- (void)commentNodesDidLoad:(NSArray<Node *> *)nodes;
- (void)userCommentsDeleted;
- (void)usersPostsDeleted;
- (void)loadingErrorOcurred:(NSError *)error;
@end
```

Code 61. Highlights of PMWallLoaderDelegate protocol.

PMWallLoader acts as a wrapper around the social network API, encapsulating all the network activity and spec work. *PMWallController* asks its loader to perform some sort of activity. When the activity has been done, the controller is notified with the help of one of the protocol methods, shown in Code 61. Schematically their relationship is shown in Figure 29.

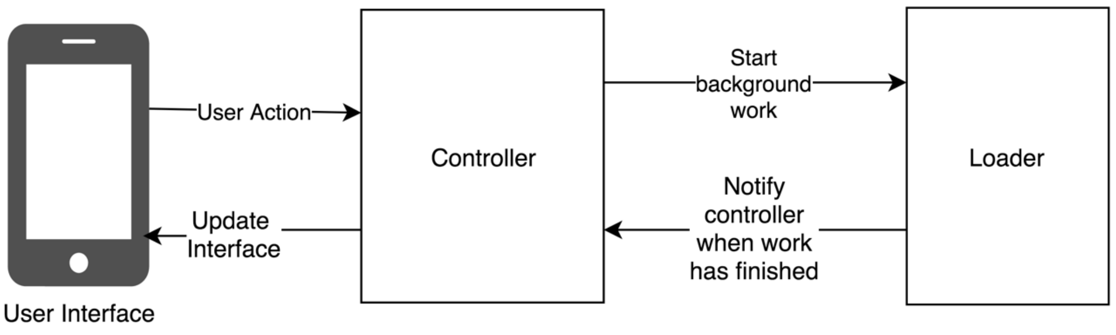


Figure 29. Scheme of Controller, Loader and relations between them.

As discussed earlier, these two classes together implement paginated loading. Let’s look, how they work together to make an impression of infinitely scrollable feed. At first, the Controller notifies the Loader to start preparing resources to show. The Loader keeps track of what has been already downloaded and prepared and requests only the resources needed. To hide the fact that the feed is not continuous, but paginated, the Controller notifies the Loader beforehand, as shown in Code 62. As soon as the Controller has started loading process, it sets the flag *isLoading* to true. This will ensure that only one call to the Loader can be made during a loading cycle. When the user scrolls the feed further, the loader would not be notified, if the next batch hasn’t loaded yet.

```
- (void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell
forRowAtIndexPath:(NSIndexPath *)indexPath {
    NSInteger row = indexPath.section;
    NSInteger preloadTrigger = [self.postsNodes count] - 25;

    if (row >= preloadTrigger && preloadTrigger > 0) {
        if (!isLoading) {
            isLoading = YES;
            [self.loader loadNextposts];
        }
    }
}
```

Code 62. PMWallController notifying PMWallLoader when there are still 25 posts to show

In PMWallLoader, the *loadNextPosts* method call executes another method, specifying the exact range of data to load. Range can be defined by offset (starting number) and length. The Loader keeps all the downloaded data in its state, therefore, can calculate the next range needed without referring to the external objects. If, for some reason *loadPostsWithOffset: count:* has been triggered during load cycle, a check will prevent the Loader from sending two network requests. *LoadingOffsetMarker* is a number, representing current offset being loaded, or *nil* if loader is idle. The checks will not pass only if the Loader has been requested the same range of items again, before the first request finished. If all the checks have passed, the Loader builds and executes network requests. This logic is shown in Code 63.

```
- (void)loadNextposts {
    [self loadpostsWithOffset:self.posts.count count:50];
}

- (void)loadpostsWithOffset:(NSInteger)offset count:(NSInteger)count{
    if (self.loadingOffsetMarker != nil && [self.loadingOffsetMarker integerValue] ==
offset) {
        return;
    }

    self.loadingOffsetMarker = @(offset);
    __weak typeof(self) weakSelf = self;

    VKRequest *request = [self getRequestForPostsWithOffset:offset andCount:count];
    [request executeWithResultBlock:^(VKResponse *response) {
        __strong typeof(weakSelf) strongSelf = weakSelf;
        [strongSelf processResponse:response];
    } errorBlock:^(NSError *error) {
        NSLog(@"Error: %@", error);
        __strong typeof(weakSelf) strongSelf = weakSelf;
        strongSelf.loadingOffsetMarker = nil;
        [strongSelf.delegate loadingErrorOcurred:error];
    }];
}
```

Code 63. PMWallLoader creates and executes network request.

When the server response has been received, it is processed to get view models (linking users' names with posts, for example). Using these view models, components are created and asked for a node. After this process has finished, the loader unlocks itself by setting *loadingOffsetMarker* to *nil* and notifies the delegate using the protocol method. This logic is shown in Code 64.

```
- (void)processResponse:(VKResponse *)response {
    __weak typeof(self) weakSelf = self;
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0), ^{
        __strong typeof(weakSelf) strongSelf = weakSelf;
        PMWallResponse *wallResponse = (PMWallResponse *)response.parsedModel;

        // Additional response processing is performed here

        NSArray<PMPostModel *> *wallPosts = [factory wallPostsWithPosts:wallResponse.posts];

        NSArray *nodes = [self nodesForModels:wallPosts];
        dispatch_async(dispatch_get_main_queue(), ^{
            [strongSelf.delegate nodesDidLoad:[nodes copy]];
            // Unlock the loader
            strongSelf.loadingOffsetMarker = nil;
        });
    });
}

-(NSArray *)nodesForModels:(NSArray <PMPostModel *> *)wallPosts{
    NSMutableArray *nodes = [NSMutableArray array];
    for (PMPostModel *wallPost in wallPosts) {
        Component *component = [PostComponent wallPost:wallPost];
        Node *node = [component nodeForContext:self.ctx];
        [nodes addObject:node];
    }
    return [nodes copy];
}
```

Code 64. Response processing and spec work.

When the delegate (i.e. the Controller) receives nodes, it updates the user interface: stops loading animations (if they have been shown) and reloads *UITableView*, as shown in Code 65.

```
- (void)nodesDidLoad:(NSArray<Node *> *)nodes {
    isLoading = NO;
    if ([nodes count] > 0) {
        [self stopLoadingAnimation];
        [self.postsNodes addObjectsFromArray:nodes];
        [self finalizeReload];
    }
}

- (void)finalizeReload {
    if (self.refreshControl.refreshing) {
        [self.refreshControl endRefreshing];
    }
    [self.tableView reloadData];
}
```

Code 65. PMWallController updates user interface after it received new nodes.

In Code 64 all the processing is offloaded to the background thread using libdispatch methods (*dispatch_async*). The user can interact with the app while the processing is going on. The control returns to the main thread (*dispatch_get_main_queue()*), to notify the delegate, as this call can be used to update the interface. Schematic representation of how the Controller and the Loader work together is shown in Figure 30.

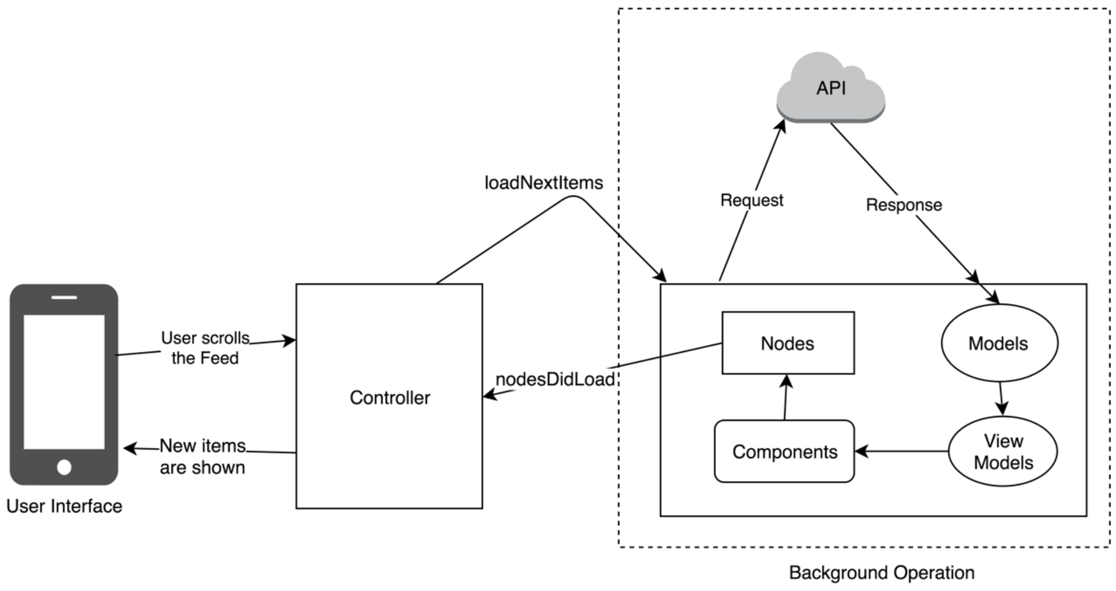
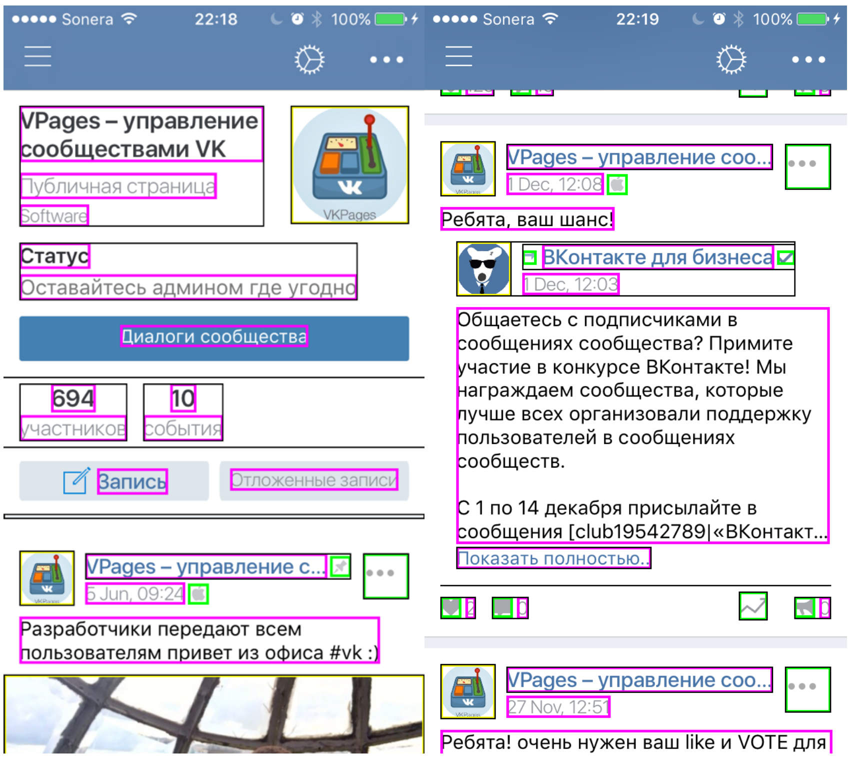


Figure 30. Scheme of continuous paginated loading and asynchronous layout calculation

6.1 Examples of User Interface Layouts

The previous section describes how the proposed solution is used in the app. In this section let's look at the examples of the layouts which can be built using the component layout system. The application code has been modified to highlight different components' role in building the final look. Each primary component has colored border around it's corresponding view. The color of the border depends on the kind of the component. Only primary components can be highlighted this way, as neither secondary nor higher-order components do not translate their layout to view hierarchy directly. Because of this, some higher-level structure is lost on the screenshots. For example, *PhotoGridComponent* is shown just as an array of buttons with images. Let's look at the examples in Figures 31-34. The components can be used not only for Newsfeeds but for example, for chats or comments. The legend shown in Figure 35 links border colors of the view with type of the view and corresponding component. Every view can be colored only in one color. However, if two views are placed on one another, two borders will be visible. For example, black and yellow borders will be used for NetworlImageComponent which can be tapped.


●●●●● Sonera 22:18 100%

VPages – управление сообществами VK

Публичная страница

Software

Статус

Оставайтесь админом где угодно

Диалоги сообщества

694 участников

10 события

Запись

Отложенные записи

VPages – управление с...

5 Jun, 09:24

Разработчики передают всем пользователям привет из офиса #vk :)

●●●●● Sonera 22:19 100%

VPages – управление соо...

1 Dec, 12:08

Ребята, ваш шанс!

ВКонтакте для бизнеса

1 Dec, 12:03

Общаетесь с подписчиками в сообщениях сообщества? Примите участие в конкурсе ВКонтакте! Мы награждаем сообщества, которые лучше всех организовали поддержку пользователей в сообщениях сообществ.

С 1 по 14 декабря присылайте в сообщения [club19542789|«ВКонтакт...

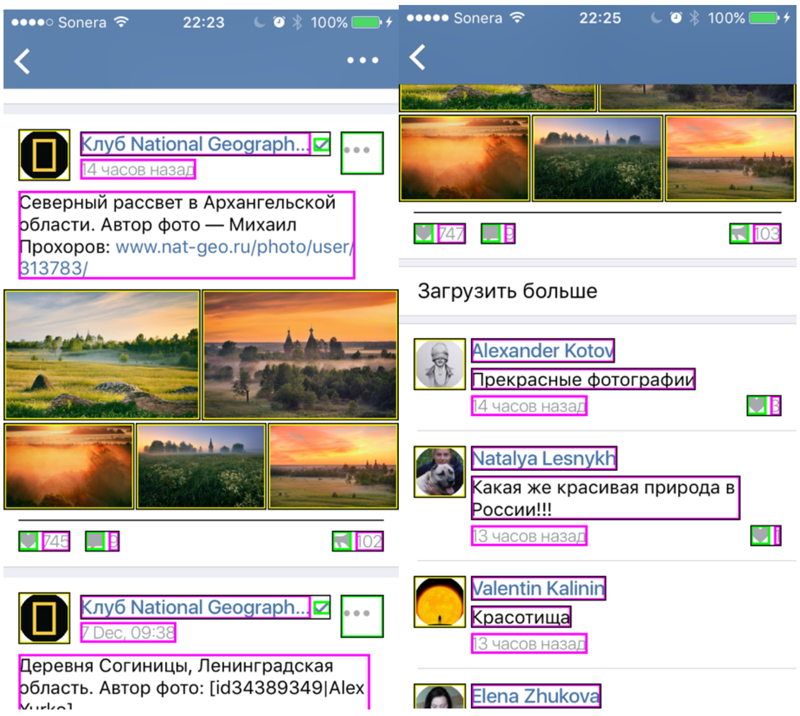
Показать полностью...

VPages – управление соо...

27 Nov, 12:51

Ребята! очень нужен ваш like и VOTE для

Figure 31. Timeline or Newsfeed built using the component layout system.


●●●●○ Sonera 22:23 100%

Клуб National Geograph...

14 часов назад

Северный рассвет в Архангельской области. Автор фото — Михаил Прохоров: www.nat-geo.ru/photo/user/313783/

745 2 102

Клуб National Geograph...

7 Dec, 09:38

Деревня Согиницы, Ленинградская область. Автор фото: [id34389349|Alex

●●●●● Sonera 22:25 100%

747 3 103

Загрузить больше

Alexander Kotov

Прекрасные фотографии

14 часов назад

Natalya Lesnykh

Какая же красивая природа в России!!!

13 часов назад

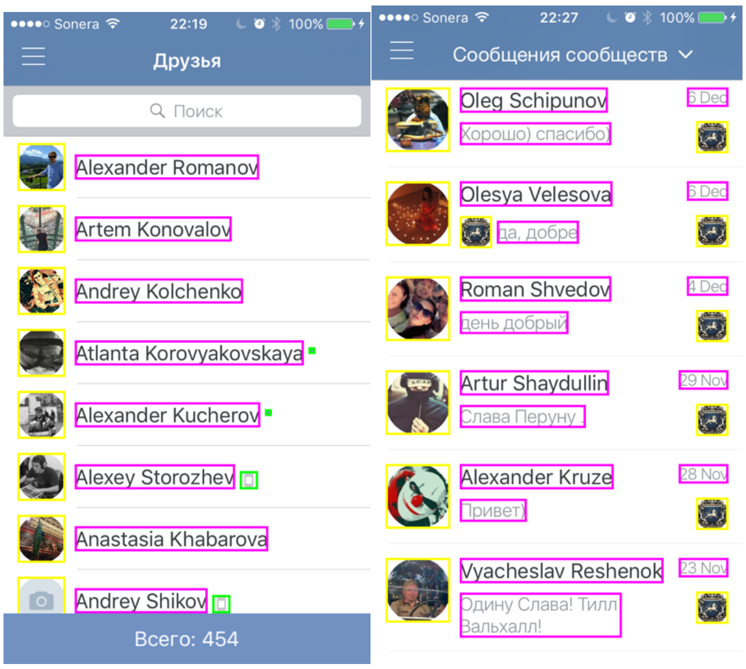
Valentin Kalinin

Красотища

13 часов назад

Elena Zhukova

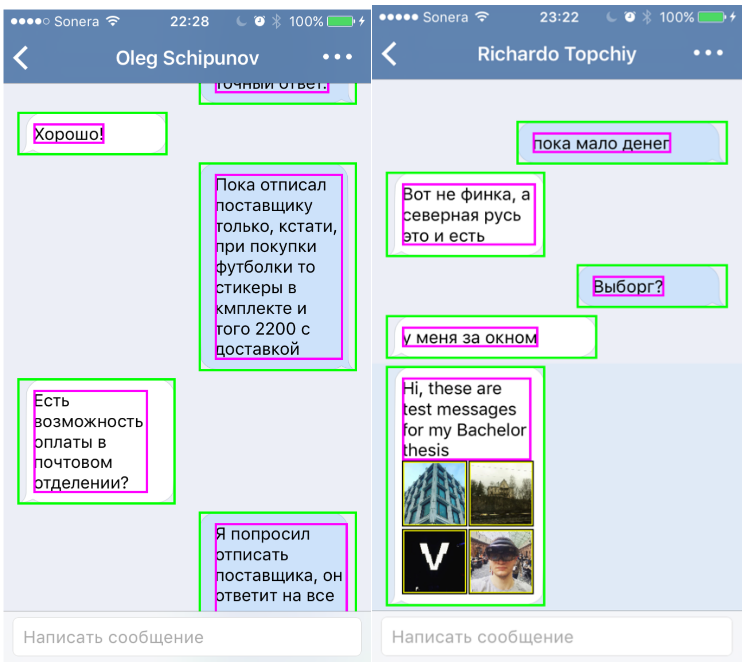
Figure 32. PhotoGrid and comment section.



Alexander Romanov
Artem Konovalov
Andrey Kolchenko
Atlanta Korovyakovskaya
Alexander Kucherov
Alexey Storozhev
Anastasia Khabarova
Andrey Shikov
Всего: 454

Oleg Schipunov 5 Dec
Хорошо) спасибо)
Olesya Velesova 5 Dec
да, добре
Roman Shvedov 4 Dec
день добрый
Artur Shaydullin 29 Nov
Слава Перуну
Alexander Kruze 28 Nov
Привет)
Vyacheslav Reshenok 23 Nov
Одину Слава! Тилл Вальхалл!

Figure 33. Friends and community messages lists.



точный ответ.
Хорошо!
Пока отписал поставщику только, кстати, при покупки футболки то стикеры в кмплекте и того 2200 с доставкой
Есть возможность оплаты в почтовом отделении?
Я попросил отписать поставщика, он ответит на все

пока мало денег
Вот не финка, а северная русь это и есть
Выборг?
у меня за окном
Hi, these are test messages for my Bachelor thesis

Figure 34. Chats.

🟣 RenderedTextView (TextComponent)

🟡 NetworkImageView (NetworkImageComponent)

🟢 ImageView (ImageComponent)

⚫ UIButton (TapableComponent)

Figure 35. Legend which links border colors, views and components together. These colors are used in Figures 30-33.

7 EVALUATION

In this section I will evaluate the proposed solution on whether it was a good choice for the VPages app. The solution had to address two issues: decrease development complexity while maintaining acceptable level of user interface performance. The combination of these two goals is harder to achieve compared to any of the goals itself. Therefore, evaluation would consist of two parts: whether the solution is easier to use than programming user interface in another way and, whether the performance is good enough. The performance of the framework is easy to measure: by building the app with and without the view reuse module. By skipping the view reuse and creating all the views on the main thread, we will simulate app performance without the framework. Comparing frames per second of each result, we would be able to see whether the performance improved. The other goal is more difficult to measure, because there is no definite way to measure developers' performance and ease of use.

User interface performance can be measured in fps, or frames per second, i.e. how many times per second the screen content refreshes. The more fps value is; the smoother animations are. iOS maximum refresh rate is 60 fps. Therefore, any number other than that is subpar performance. Of course, there would be a big difference between 59 fps and 15 fps, but generally, 60 fps is the ultimate goal. One app can perform with different refresh rates on different devices: the performance of the same app may be slower on iPhone 5 compared to performance of the same app on iPhone 7. Newer devices are running better hardware and faster than the devices of previous generations. Therefore, by testing performance on older devices we can ensure that the performance level is at least the same on the faster ones.

I've picked iPhone 5S for performance measurement. It was introduced in 2013 and it is the second slowest device which supports iOS 10. Only iPhone 5 is slower.

Performance can be measured only on the device, as iOS Simulator app runs on the developer's computer, which is usually faster than any of the iOS devices. To conduct measurements, device should be connected to the computer via cable. Xcode has the Instruments tool where numerous measurements can be conducted. The one needed for us is the Core Animation. It measures GPU performance.

Measurement strategy:

1. Use the same set of data across all the measurements.

2. Test performance with view reuse enabled.

3. Test performance with view reuse disabled.

4. Compare the results and find out, whether the view reuse approach helps improving performance.

7.1 Measurements Results

I used official app community page for both tests: https://vk.com/vpagesios. At first, I've tested app with view reuse enabled. The number of frames per second never dropped below 50 and most of the time has been in 55-60 range. It is not the best possible performance, however, it's very good considering complexity of the app. With view reuse disabled I didn't notice any significant difference in performance. The results are shown in Figure 36.

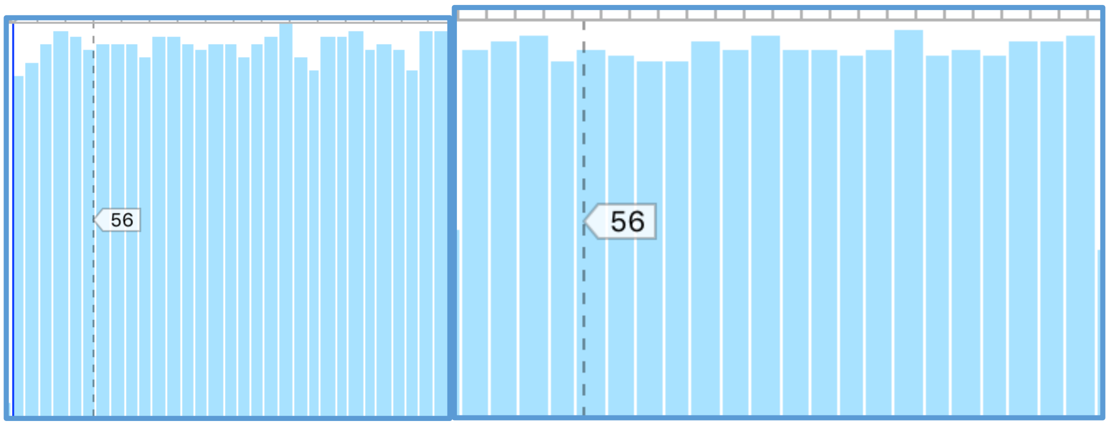


Figure 36. Frames per second with view reuse enabled (left picture) and with view reuse disabled (right picture).

7.2 Results Evaluation

While the performance and developer's convenience has been improved with the framework, there is still a possibility to make it perform even better. One of the performance drawback is due to the way framework uses Cold and Hot reuse pools. During the view reuse process Hot pool is created containing all the subviews of the view it operates with. It means, if the subview is not directly placed on the other view, it is not added to the pool. In our case, if other view is placed on top of the button, for example, as in tapable images, only button component is added to the Hot pool. Instead of desired flat view hierarchy where all the subviews are placed on the host (empty) view, view hierarchy is split onto multiple groups of buttons containing other views. Therefore, most of the time views would be removed and added to a new superview, which limits performance and leads to frame drops. If there were no tapable components, it would be possible just to rearrange subviews to get new layout. In social networks, however, most of the interface elements are interactive and need to respond to user's input. In Figure 37 two cases are shown. If the most of the elements of the lists were built like it is shown on the left picture, the number of frames per second would be closer to 60. However, the bigger part of the content is built like it is shown on the right picture. If we take for example, one of the ImageViews (green color). In case of the right picture, those ImageViews cannot be just moved to a new position. If ImageView is needed for some other task, it will be removed from the button. Then, new ImageView should be added to the view hierarchy again. In worst case, it should not only be added, but initialized. The algorithm cannot just take the page header, for example, and fill it with new content.

Some views may be removed and some views may be added, reducing time available for main-thread operations.

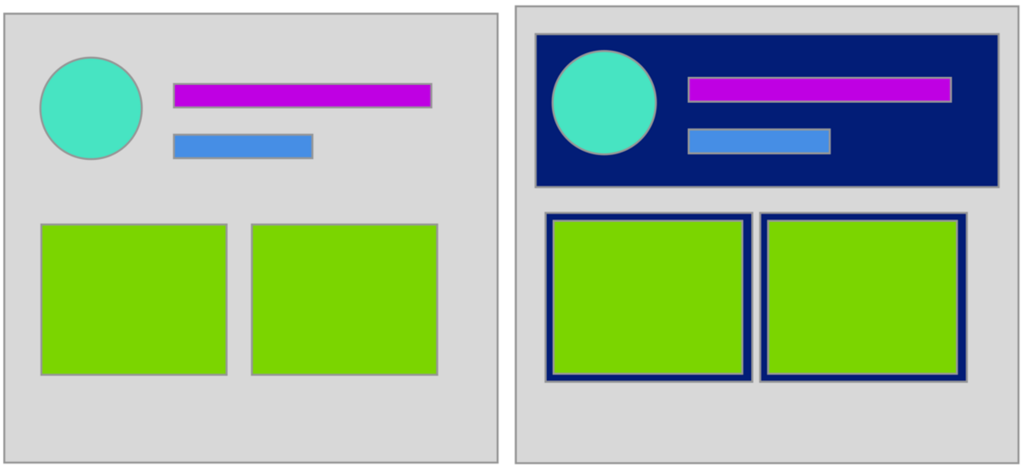


Figure 37. Desired view hierarchy has a flat structure. Because of the tapable components, actual view hierarchy is being split into multiple groups of subviews.

7.3 Development Productivity Evaluation

To evaluate developer's productivity, I must refer to own opinion and experience, because there is no other way to precisely measure, whether development is more productive with the proposed framework or not. This is not the first social networking app in development of which I took part, therefore I can compare between different experiences. In my opinion, layout definition become much easier with the Components framework, as I do not have to explicitly compose view hierarchy by adding or removing subviews. Also, I like that the framework reflects the way designers define layout using relative margins and concepts such as lists and insets. The framework also helps when building adaptive layouts which stretch or shrink depending on the size of the screen.

To fix a layout bug, it is required to open only necessary element of the layout. For example, if some element of the header is positioned incorrectly, most likely, editing the "Header" component will solve the issue. This is not always true when layout is defined in imperative way, or using AutoLayout. Adding one extra element or changing their order can be quite complex: in case of AutoLayout it will require rewriting of the whole layout code. In case of the Components, changing the order of the elements in list will be reflected in layout immediately. Therefore, I consider the Components framework to be a great choice for the Agile development, when requirements change often.

7.4 Stability Evaluation

Application stability is one of the most important factors. If app works unreliably, crashes frequently and contains bugs, users won't be satisfied and will delete the app. Social networking apps are among the most complex, therefore, the framework should work predictably and reliably on a large scale. It is simply impossible to test all the use-cases during the app development, because there are millions of profiles in the social network.

The VPages app was released on 5th of April 2016. The app has embedded analytics library from Fabric.io, which can be used to monitor user base and stability of the app. As for December 2016, the app has about 1500 monthly active users and this number is not growing. Number of daily active users falls in range between 200 and 300. The dynamics of these parameters is shown in Figures 38-39.

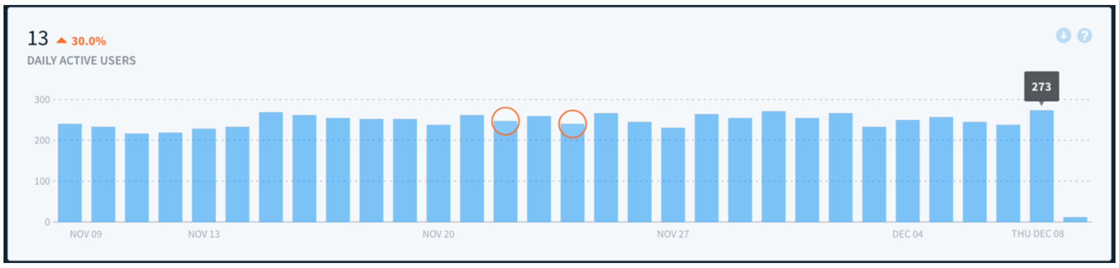


Figure 38. Daily Active Users.

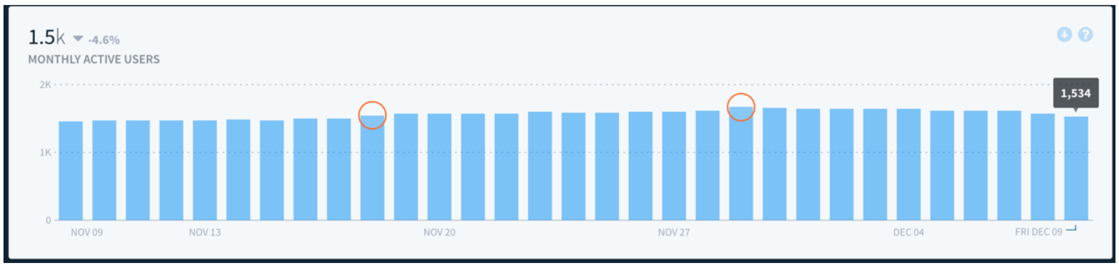


Figure 39. Monthly Active Users.

It is not very big user base, but it may be enough to understand, whether the app works reliably or not. If app didn’t bring value to the users, they would simple delete it. Looking at the percentage of crash-free users, i.e. users who opened the app during the day and it didn’t crash, we can get a better understanding of how reliable framework is. The crash-free users percentage graph is shown in Figure 40. The line almost never drops below 95%, which is a very good result for social networking application.

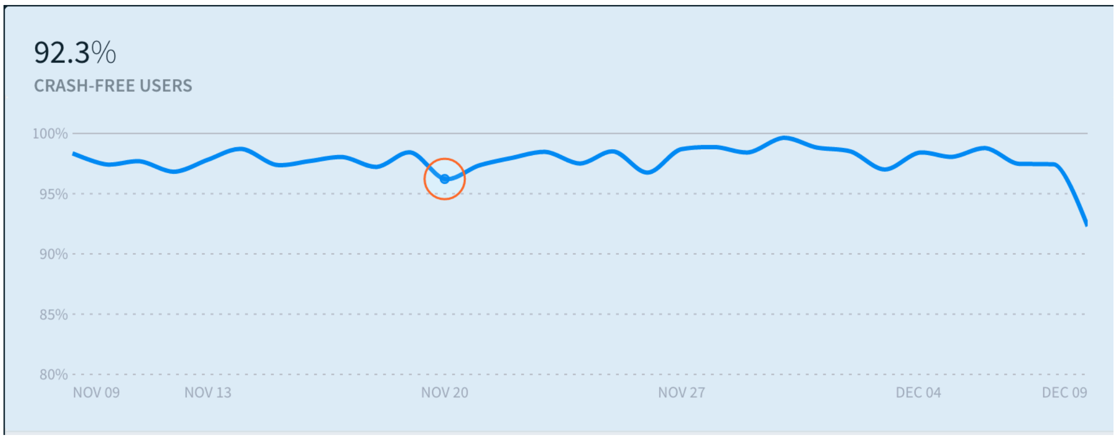


Figure 40. Crash-free users percentage.

While the numbers mentioned above are related to the VPages app, they do not evaluate the quality of the Components framework directly. The reason is that the Components framework contributes only to the one part of the app, while there are many more modules and frameworks. A single failure in any of the modules could result in a crash. Therefore, the percentage of crash-free users should be considered the minimum. As if the app was built using only the Components framework, that percentage would be higher.

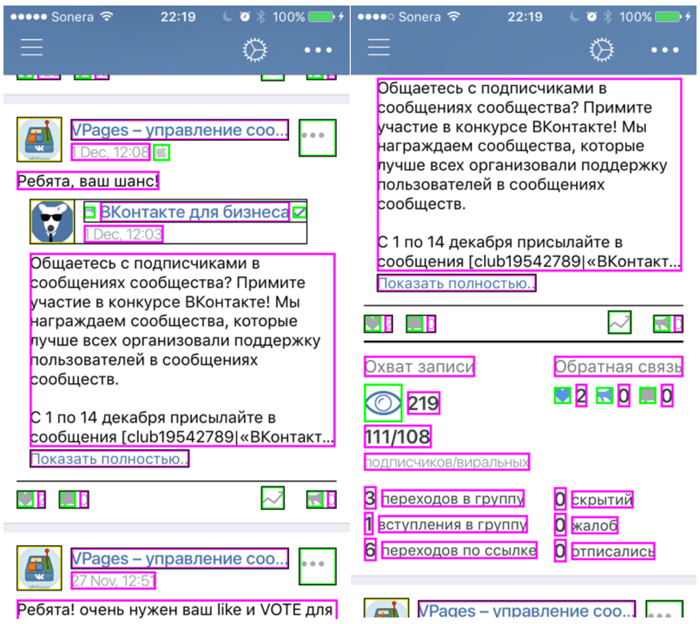
7.5 Users’ Reception

I consider one of the best ways to evaluate the app is to ask its’ users for a feedback. We’ve already got about 700 followers in our official community. Also, we constantly receiving thank you messages and bug reports, which indicates that app is popular. We are also receiving feature requests and requests to port the app to the Android. On App Store all versions of the app have received 35 ratings, average is 4 star.

7.6 Suitability

While the proposed solution greatly increases the developer's productivity and maintains acceptable level of scrolling performance, it is not suitable for every app. In this section I will describe cases where the Components framework may or may not fit.

The Components framework was designed for Newsfeed-like interfaces to achieve highest level of scrolling performance. Because of loose coupling between the Layout Declaration Module (i.e. the Components) and the Node to View Translation module, it makes incorporating any kind of interactivity difficult. For example, if user should be able to drag any of the elements with a touch, the framework may only overcomplicate programming. When the state of the user interface changes, all the Nodes of the post or element are being regenerated. This leads to rebuilding of the view hierarchy. Advanced post analytics is shown using this technique. In Figure 41 on the left image the screen with collapsed analytics section is shown. When user presses analytics button, a new section drops down.



Ребята, ваш шанс!

Общаетесь с подписчиками в сообщениях сообщества? Примите участие в конкурсе ВКонтакте! Мы награждаем сообщества, которые лучше всех организовали поддержку пользователей в сообщениях сообществ.

С 1 по 14 декабря присылайте в сообщения [club19542789|«ВКонтакт...

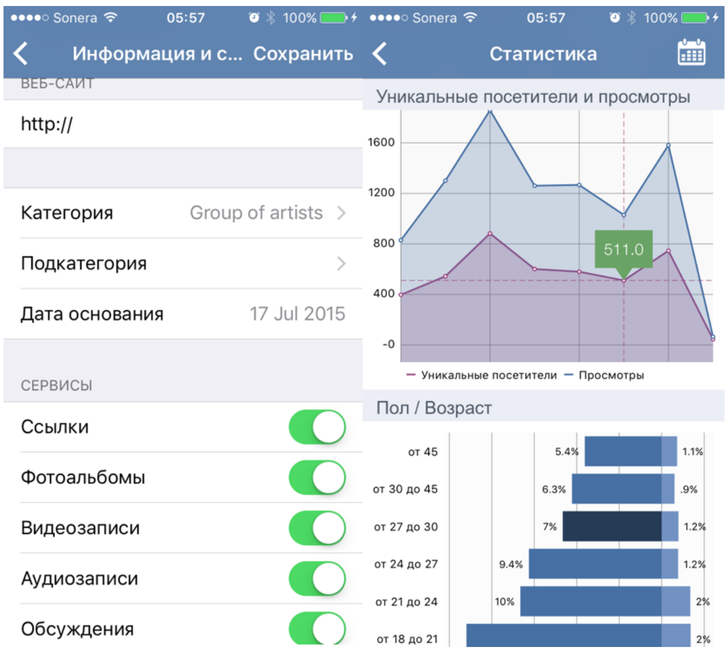
Ребята! очень нужен ваш like и VOTE для

3	переходов в группу	0	скрытий
1	вступления в группу	0	жалоб
6	переходов по ссылке	0	отписались

Figure 41. Presentation of advanced analytics for post.

If app is required to handle more complex interactions other than basic tap, long press or double tap, I would not recommend using the Component framework. I also follow this recommendation myself, therefore some of the screens in VPages are made without the framework. Examples of these screens are shown in Figure 42. The picture on the left is a community settings page. It contains text fields, switches, category selectors and date pickers. All the above are custom iOS views. To add them to the Components framework, the developer would be required to create custom primary component for every view. Then, the developer would have to handle all the interactions via the Component routing system. There is no need in the Components, as the layout of the settings page always stays the same.

The picture on the right shows an example of the community analytics page. Every view is custom and interactive. User can pinch to zoom, tap to see the details and interact with the content in many ways. Moreover, the initialization of each of the chart views takes a lot of time and resources. They cannot be reused in a way as, for example, UIImageView. Therefore, me and Artem decided to use ready chart library, "iOS-charts" .


●●●●○ Sonera 05:57 100%

Информация и с... Сохранить

ВЕБ-САЙТ

http://

Категория Group of artists

Подкатегория

Дата основания 17 Jul 2015

СЕРВИСЫ

Ссылки ✓

Фотоальбомы ✓

Видеозаписи ✓

Аудиозаписи ✓

Обсуждения ✓

Figure 42. Screens which do not use the Components framework.

Another example is related to tracking what part of the content user sees and automatically starting to play videos. This will require loading resources on demand. Views containing videos are extremely resource intensive, therefore, the developer should write a custom feed to optimize for performance.

The Components framework is a solution for one problem. It won't fit all cases. It's good to know, what the tool is capable of and use it only when needed.

8 CONCLUSION AND RECOMMENDATIONS FOR FUTURE RESEARCH

In this thesis, I've proposed the solution for building responsive scrollable interfaces on iOS using native components. Also, I've shown a social networking app based on the solution as an example. The Components framework makes development of complex interfaces a lot easier because of its modular structure. The level of performance is acceptable for most of the apps and can be improved. I recommend using similar approach for building social networking apps, chats or catalogues. However, apps are becoming more and more interactive nowadays which may limit use of the framework.

8.1 Improving Performance with the Help of the UICollectionViewLayout

Right now the performance of the framework is not the best of what it could be. Therefore, there is a room for improvement, even if it is small. As I've described before, the frames per second drops may be the consequence of non-flat view hierarchy. By using UICollectionView and custom UICollectionViewLayout it would be possible to rewrite the Node to View Translation part completely without touching the other parts of the app. The most important feature of the UICollectionView is that its elements may be placed one onto another without the need of adding or removing subviews. All the items are placed onto the UICollectionView once and then, can be moved to the front or put back by modifying their *z-Index* property. Moreover, all the elements can intercept events by default using the UICollectionView delegate methods. This removes the need in TapableComponent.

8.2 Interactivity

In future, animations in the newsfeed would become standard. In fact, they've already used in lots of apps. Another interactive element is video or GIF animation which starts playing automatically. It requires keeping track of where the place which user has scrolled to and pre-loading assets beforehand. Some of the interactivity elements could be added to the framework without making it more complicated to use. However, there should be a line between higher-level declarative frameworks like this and lower-level that aim for maximum performance while giving more control over the layout, such as AsyncDisplayKit from Facebook.

8.3 Event Handling and Routing

I intentionally didn’t mention full description of the solution I’ve used for the event handling and only provided the interface. This interface may be used to host event handling modules of different complexity, depending on the needs for the app. However, it would be good to have some default event handling module, so the developer would not be required to write its own. This will help minimize time needed to start implementing layout in code and will keep developer focused on more important issues, such as application business-logic and bugs.

BIBLIOGRAPHY

Swift (programming language) - Wikipedia
https://en.wikipedia.org/wiki/Swift_(programming_language)
Referred 8.4.2016.

Objective-C – Wikipedia
https://en.wikipedia.org/wiki/Objective-C
Referred 8.4.2016.

Swift and Objective-C in the Same Project
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCo-coaApps/MixandMatch.html
Referred 8.4.2016.

A collection of test cases crashing the Swift compiler.
https://github.com/practicalswift/swift-compiler-crashes
Referred 8.4.2016.

How to Use CocoaPods with Swift
https://www.raywenderlich.com/97014/use-cocoapods-with-swift
Referred 8.4.2016.

About iOS Collection Views
https://developer.apple.com/library/ios/documentation/WindowsViews/Concep-tual/CollectionViewPGforIOS/Introduction/Introduction.html#//ap-ple_ref/doc/uid/TP40012334-CH1-SW1
Referred 8.4.2016.

Object pool pattern
https://en.wikipedia.org/wiki/Object_pool_pattern
Referred 8.4.2016.

Collection View Basics – Reusable Views Improve Performance

https://developer.apple.com/library/ios/documentation/WindowsViews/Conceptual/CollectionViewPGforIOS/CollectionViewBasics/CollectionViewBasics.html#//apple_ref/doc/uid/TP40012334-CH2-SW13
Referred 8.4.2016.

WWDC 2012 session 228 “iOS App Performance. Graphics and Animations”
https://developer.apple.com/videos/play/wwdc2012/238/
Referred 8.4.2016.

Objective-C Associated Objects
http://nshipster.com/associated-objects/
Referred 29.4.2016.