# Golden Gate iPad Architecture

Golden Gate iPad Architecture 1

1 Introduction 2

2 Overall code structure 2

2.1 Directory structure 2

2.2 Component graph 3

3 Backend 3

3.1 RESTEasy and VimondAPI 3

3.2 VimondStore 4

4 Prefetching data source 4

4.1 Initialization 4

4.2 Usage 4

4.3 Clearing the cached data 5

5 User interface 5

5.1 Content cells 5

5.2 Grid views 6

5.3 Reusable view controller base classes 6

6 Navigation handling 6

6.1 Node structure 6

6.2 Navigating programmatically 7

6.3 Caching 7

6.4 Helpers 7

7 Localization 8

7.1 Localization of .xib files 8

7.2 Localization of backend responses 8

8 Usage tracking 8

8.1 SDK 8

8.2 What is being tracked? 8

8.3 How to expand tracking? 8

8.3.1 Expanding the events tracked 8

8.3.2 Adding more tracking systems 9

8.4 Asking the user for permission to track 9

9 External libraries 9

9.1 GMGridView 9

9.1.1 Usage 9

9.1.2 Download 9

9.2 TestFlight 9

9.2.1 Usage 9

9.2.2 Download 9

9.3 OCMock 10

9.3.1 Usage 10

9.3.2 Download 10

# 1 Introduction

This document contains a summary of the architectural decisions made in the Golden Gate iPad app (hereafter referred to as “the app”).

The application is developed with iOS5 compatibility in mind.

# 2 Overall code structure

## 2.1 Directory structure

|  |  |  |
| --- | --- | --- |
| Directory path | Contents | Dependencies |
| /GoldenGateTests | Unit Tests. | VimondAPI  RestEASY  Model |
| /Frameworks | External frameworks. | None |
| /Util/ | Utility classes that for legacy reasons just stay in this folder. Can be moved to /GoldenGate/Utils | None |
| /GoldenGate/Controller/ | All view controllers and their .xib files | KIT  Model  View  Utils  Util |
| /GoldenGate/KIT/ | Knowit toolkit classes. More on KIT in a later chapter. |  |
| /GoldenGate/Model/ | Application specific model objects used throughout the project. | VimondAPI |
| /GoldenGate/Resources/ | Images and fonts and strings. | None |
| /GoldenGate/RESTEasy | A framework that replicates some of the functionality of the Java based RESTEasy framework. | None |
| /GoldenGate/Utils | Assorted helper classes | None |
| /GoldenGate/View | UIView subclasses and their .xib files. | Model |
| /GoldenGate/VimondAPI | Vimond backend specific code whose main task is to communicate with the Vimond web api. | Model  RESTEasy |

## 

## 2.2 Component graph



# 3 Backend

## 3.1 RESTEasy and VimondAPI

The RESTEasy framework is used to build HTTP requests and handle their responses based on easy-to-make templates.

The classes found in VimondAPI/Services implement most of the methods available from Vimond web api and produce DataTransferObjects their responses.

Unit tests for these classes can be found in the GoldenGateTests folder.

For examples of how to extend the API, please see any Client class found in GoldenGate/VimondAPI/Services.

The gist of it is:

* Create a ClientMethod object using **RestClient createMethod** and an appropriate method template.
* Create a ClientArguments object
* Fill that argument object with the data correct parameters.
* Call the RestClient execute:arguments:error method.

## 3.2 VimondStore

The VimondStore class is a singleton that provides a façade towards the back-end itself.

The “Store” objects available from the VimondStore are used to bridge the gap between the VimondAPI and the classes found in the Model directory. These Model objects are then used to populate the UI.

# 4 Prefetching data source

The class **PrefetchingDataSource** is instrumental in lazy loading of the content in the grids and tables used in the app.

It provides a cached array-like structure where elements are loaded “page by page” as they are needed. The idea is also that adjacent pages are to be loaded when an element is requested so that scrolling remains a smooth experience.

## 4.1 Initialization

A **PrefetchingDataSource** needs to be initialized with an object implementing the **DataFetching** protocol.

The **DataFetching** object is responsible for delivering objects for a given page index and size. This is specifically tailored to work against the Vimond API that delivers data in paged chunks.

**DataFetching** objects usually perform a call to the Vimond API via the **VimondStore** class.

## 4.2 Usage

Use **fetchDataObjectAtIndex:successHandler:errorHandler** to get the data object at a given object.

This method will first check if there is any cached data for the page that the requested index belongs to. Example: Index 15 will belong to object index 3 on page index 1 in a data source with 12 objects pr page.

If no cached data can be found the needed page is requested from the DataFetching object. The data source will also preload any adjacent pages for smooth transitions during paging. The number of adjacent pages loaded is controlled by the **pagesToPrefetchOnEachSide** property.

## 

## 4.3 Clearing the cached data

There are multiple ways that the cached data in a **PrefetchingDataSource** can be cleared.

* The cached data is very naïvely thrown away once the amount of total objects reported from the **DataFetching** object has changed.
* Before fetching any data the **DataFetching** object is always asked if the data should be reloaded.
* Explicit calls to **resetDataSource**

# 5 User interface

The user interface is to a fairly large extent made up of modular re-usable view classes that are composited using interface builder.

The UIView subclasses in this app either customize their appearance programmatically or by inserting views defined in .xib files.

By using the identity manager in the interface builder one can change the class of the view being loaded.

## 5.1 Content cells

The most prevalent UI element in the app is the content cell (subclasses of CellView).

There are currently three types of content cells:

* VideoCellView
* ChannelCellView
* FeaturedContentCellView

The appearance of the cells will vary depending on the cell’s given CellSize.

Each CellSize will map to a .xib file name suffix.

Example:

CellSizeLarge maps to VideoCellViewLarge.xib, ChannelCellViewLarge.xib and FeaturedContentCellViewLarge.xib

## 5.2 Grid views

Grid views are made easy to populate with the **GridViewController** class.

A **GridViewController** needs two things to work.

* A **PrefetchingDataSource** to define what data it should show.
* A **CellViewFactory** that will create a view for each data object the data source delivers.
  + Customize the factory to specify the class and size of the view to be used.

## 5.3 Reusable view controller base classes

Many of the views in the app share the same look at the same time as they show different kinds of data.

By subclassing one of these classes you can easily register a PrefetchingDataSource and a CellViewFactory that will be used to populate the grids.

Currently, these re-usable base classes are:

* DoublePagedGridViewController
* SingeGridViewController

# 6 Navigation handling

Navigation in the app is based on a tree-like structure of NavAction objects.

The name itself might be a might be a bit misleading. A better name could be NavNode.

## 6.1 Node structure

The figure that follows depicts a subsection of the navigation tree.



**Blue nodes** are objects from a subclass of **ViewAction**

**Square white nodes** are **FilterAction** objects.

ViewAction nodes are visible in the navigation table.

FilterAction objects are visible in the segmented controller in the navigation bar.

Each **FilterAction** has a **BaseNavViewController** class that is to be used when displaying the data for its parent **ViewAction**.

## 6.2 Navigating programmatically

**NavActionExecutor** is the class responsible for handling most navigation in the app.

By using the **executeNavAction:fromViewController:** the **NavActionExecutor** will push and pop and create the appropriate view controllers to the navigation controller of the given view controller.

When navigating directly to a **FilterAction** the appropriate view controller is instantiated with the **FilterAction** as navAction. The view controller usually uses the information found in the parent **ViewAction** of the **FilterAction** to populate its view.

When navigating to a **ViewAction** the child **FilterAction** with the same name as the last used **FilterAction** is chosen. If there is no last used filter then the first child will be chosen.

## 6.3 Caching

View controllers created by the **NavActionExecutor** will be cached on a pr. **FilterAction** basis.

This will also only happen if the **FilterAction** itself allows caching.

Disabling caching on a **FilterAction** can be good a good thing to do for content that may change during application runtime like “My Channels” and “Favorite Videos”.

User testing will tell if the caching is too aggressive.

The **kMaxCachedViewControllers** define in **NavActionExecutor** controls the maximum number of cached view controllers. Once the limit is reached one of the previously cached items will be discarded from the cache.

## 6.4 Helpers

There are currently two helpers for navigation:

* CategoryNavigator – Navigates directly to a category with the given identifier
* FeaturedContentNavigator – Navigates to the content pointed to by a FeaturedContent object. (ie. Featured Content Panels). What happens depends on what kind of content it’s pointing to. It can navigate to a channel, video and category.

# 7 Localization

The NSLocalizedString macro is used for localizing strings programmatically, but there are some special aspects of the localization that needs mentioning.

## 7.1 Localization of .xib files

The KITViewLocalizer object in the GGBaseViewController does localization of strings in .xib files. The KITViewLocalizer traverses the view-hierarchy and substitutes any views containing text-strings ending with LKey with their respective values from the Localizable.strings file.

Note that this only works on UIButtons and UIView classes that respond to the **text** and **setText** selectors.

## 7.2 Localization of backend responses

The metadata delivered from the Vimond web-api sometimes contain strings for multiple languages. Currently the app only uses the string marked with language code “\*”.

In the future, if multiple languages are to be supported, **valueForCurrentLanguage** in the **RestMetadata** class must be changed to fetch the current application language.

# 8 Usage tracking

## 8.1 SDK

We will use the Google Analytics(GA) v.2.0 SDK for iPad. It’s currently in a beta state, but for the best possible future proofing we should use it anyway. Google has a reputation for quality Beta releases.

## 8.2 What is being tracked?

* Screen views
* Like asset
* Favorite asset
* Like channel
* Asset playback/preview
* Playback duration
* Playback failure

## 8.3 How to expand tracking?

### 8.3.1 Expanding the events tracked

To expand tracking system add code the following places:

* Create a new entry in the **GGUsageEvent** enum.
* Register an event template in **GGUsageTracker registerEventTemplates**
  + Check **UsageEventTemplate** for more details.
* Add a method in **GGUsageTracker** named **–(void)track(…)** that calls either **trackEvent:eventData** or **trackTiming:eventData** with the enum created in step 1.

### 8.3.2 Adding more tracking systems

If you would like to add another tracking system create a class implementing the **UsageTracking** protocol and call **registerUsageTracker** in **GGUsageTracker init**

The tracker does not have to do anything useful in tracking methods that are not supported, but it has to implement the method and return YES.

## 8.4 Asking the user for permission to track

According to GA SDK policy we are required to allow the user to opt out of being tracked.

Currently, we do this by asking the user upon initial usage of the app.

<https://developers.google.com/analytics/devguides/collection/protocol/policy>

# 9 External libraries

All external libraries can be found in the /Frameworks/ directory.

## 9.1 GMGridView

### 9.1.1 Usage

**GMGridView** is used in conjunction with **GridViewController** for all grids in the app.

It provides a 3rd party implementation of what is known as Apple’s **UICollectionView** that is available in iOS6. The GG-app is required to be compatible with iOS5 and can therefore not use Apple’s implementation.

### 9.1.2 Download

<https://github.com/gmoledina/GMGridView>

## 9.2 TestFlight

### 9.2.1 Usage

The **TestFlight SDK** provides various functionality that eases testing and test-deployment of the app.

The SDK is currently only used in the app to catch uncaught exceptions and log usage.

In the future it might be used to track usage during testing together with **GoogleAnalytics**

### 9.2.2 Download

<https://www.testflightapp.com/sdk/>

## 9.3 OCMock

### 9.3.1 Usage

OCMock is used by our unit tests to deliver mock responses to calls to the Vimond API

Read more about mock objects on: <http://www.mockobjects.com/>

### 9.3.2 Download

<http://ocmock.org/>

## 9.4 Google Analytics

### 9.3.1 Usage

Google Analytics is used for tracking user and system events throughout the app.  
See chapter 8 for more details.

### 9.3.2 Download

<https://developers.google.com/analytics/devguides/collection/ios/resources>