Allan Fernando Arriaga

JavaScript Web Applications

Building web applications using Asynchronous Module Definition (AMD)

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Bachelor's Thesis

Date 22/05/13

Author Title Number of Pages Date	Allan Fernando Arriaga Building web applications using Asynchronous Module Definition (AMD) 50 pages + 1 appendix 22 May. 13
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Specialisation option	Audio Visual Technologies and Hybrid Media
Instructor(s)	Tatu Dufva, Product Manager, Conmio Oy Kari Salo, Principal Lecturer

Asynchronous Module Definition (AMD) is a design pattern used to build modular, browser-based applications. This document aims to provide an overview about what the pattern solves and why it is important not only for the industry but also for developers and ultimately for users and consumers.

Building modules instead of large applications is an efficient pattern to create a better mobile experience often on high-latency 3G connections. AMD makes possible to load individual modules and manage dependencies. Libraries like RequireJS, enable developers to optimize this process turning modules into ordinary packages.

The purpose of this thesis is to improve a development process previously established and allow developers to create new web services with a common core code structure.

As a result of this study the implementation was successfully included in a project at Conmio and transformed into a grails plugin for a Media Broadcasting client case. Finally, the implementation was used to create many tablet apps for the broadcasting networks that the client owns.

Keywords	javascript, asynchronous, modules, AMD, design patterns, web applications, dependency management, script loader, software engineering, mobile apps

Table of Contents

1 Introduction 5

1.1 About Conmio 6
1.2 Problem Description 6
1.3 Domain and Focus 7
1.4 Team details 7

2 Theoretical Background 8

2.1 Modern Web Applications 8
2.1.1 Source, Platform & Service 10
2.1.2 Native and Hybrid Applications 10
2.2 Think Mobile First 11
2.3 Modular Web Design 12
2.4 The Module Pattern 12
2.5 AMD: Asynchronous Module Definition 14
2.5.1 Before AMD 15
2.5.2 Why AMD? 16

3 Specifications and Project Description 18

3.1 Client Case 18
3.2 Targets 18
3.3 Application Architecture 19
3.3.1 Data models 20
3.3.2 Application structure 21
3.3.3 Application Programming Interface (API) 22
3.4 Toolbox: libraries chosen 23
3.4.1 BackboneJS 24
3.4.2 Backbone.Marionette 24
3.4.3 Underscore and Lodash 25
3.4.4 Handlebars 26
3.4.5 RequireJS 26
3.4.6 Other Libraries 27
3.5 Implementation 28
3.5.1 Configuration and initial set up 29
3.5.2 Basic View hierarchy 34
3.5.3 Creating components 36
3.5.4 Making a grails plugin 41

4 Project Summary 41

4.1 Lessons Learned 42

4.2 Improvements for the future 44

5 Conclusions 48

References 50

6 Screen shots of main views and services 1

1 Introduction

Mobile Web is commonly used to refer to mobile Internet access, the use of browser based Internet services with handheld devices, such as smartphones or tablets. The *mobile web* has grown rapidly to challenge the traditional web browsing in the last few years. However, the web is really one and the same but the device used to browse makes a difference.

In the last few years the growth of different types of mobile devices and platforms has made the mobile web a challenge for developers. The fragmentation of the devices usually pushes new services towards a narrow target audience including only leading manufacturers and latest operating systems. This thesis researches a solution to improve the web based applications written in JavaScript for the modern browser previously done at the company.

Personally, I have been working as a front-end developer and mainly doing JavaScript development for web services in a company that serves mostly huge media publishing houses. It is mainly front-end developer's responsibility to make sure that the services are performing as expected and that the code base is organized and clear for further development to happen.

Web services have been rising lately at a rapid pace such as services that are formed with multiple smaller pieces like widgets or smaller apps. These services can be, for example, the local weather, stock market or sports services that update in real time.

The company studied in this thesis has been looking for new techniques to improve the development process and maintain innovation in their technology to be able to prevail in a rapidly evolving industry. This thesis covers one of these techniques and gives guidance for further development with the same concepts. Asynchronous Module Definition (AMD) pattern has been selected for improving the web development process.

Chapter one covers the general introduction for the thesis. In chapter two, I'll focus on general concepts and the theoretical background which provides the basic technical knowledge for understanding what the thesis is about. Chapter three focuses on the project specifications and the implementation including basic documentation on how the project is structured and how it can be utilized in the future. The final part of the

document includes the project summary conclusions and thoughts on how the implementation can be further improved in the future.

1.1 About Conmio

Conmio is a relatively new Helsinki based IT company established in 2002, and since then, it has developed an innovative solution for multi-channel publishing enabling media companies around the globe to be published on the mobile web. Conmio has been helping new customers discover new possibilities and improving current mobile web services to the latest trends and technologies. Conmio's services and products include media brands, mobile operators, publishing content and systems integration.

Mobile devices are becoming more powerful and web browsers are faster and more efficient, there has been a trend of applications moving away from the traditional native environment world and towards a more adaptable and maintainable web environment or a hybrid method, creating a native wrapper around views and UIs built with standard web technologies. Even though Conmio's strong front is mobile websites, the company has also developed native applications for iOS, Android, Windows, BlackBerry and Symbian devices.

Conmio's client base is wide, including large publishing houses such as The New York Times, MTV3, Boston Globe, Bloomberg. Nowadays, Conmio is a workplace to around 40 professionals and it has headquarters in Helsinki and New York.

1.2 Problem Description

When building large-scale JavaScript applications, there is a high risk of creating a costly and messy implementation which is almost impossible to scale and reuse.

The Module Pattern, specifically Asynchronous Module loaders, provide a solution to this problem. Instead of building large-scale applications, the application can be divided into smaller pieces that are testable, maintainable, scalable and reusable on their own. Ultimately, these pieces are put together to create a fully functional application.

The objective of this document is to highlight the many advantages of Asynchronous Module Definition (AMD) and its philosophy, in other words, to present AMD as the simplest most suitable way of creating digital content publishing services that are made of many decoupled, reusable and adaptable components/modules. Also, the benefits of building components/modules instead of large scale applications.

1.3 Domain and Focus

When we talk about web applications (webapps) development, it is easy to deviate because there are several technologies available to build apps for the web. The domain of this document is Modern Web Applications, often simply called ***web apps*** *or "HTML5 apps".* The main focus is on client-side or browser-based applications involving heavy workload using JavaScript, as it is the main programming language to create the application structure and data modelling.

It can be assumed that our target devices are capable of handling heavy scripting on the client-side, mostly in the tablet category including the latest iterations of widely used devices, i.e. Apple's iPad running iOS 5+.

Server side programming will be referred to but it is not the focus of this document. The technologies used in the back-end are Groovy on Rails (Grails). Grails is a Java based web framework and it is used in Conmio's own product. The backend solution is used to create a RESTful API that can then be used with JavaScript to model data client-side.

1.4 Team details

The development team was formed by six software developers: two server side proficient developers and 4 client side proficient developers. My work was client side oriented and my tasks in this project included: reviewing specifications and wireframes, creating layouts, style implementation, creating templates, data modelling, creating models and collections, implement view logic, architecture design, component design and implementation and finally writing documentation and APIs.

2 Theoretical Background

This chapter presents the main concepts used throughout the document and its abbreviation(s), if any.

Most importantly, one must understand that some, if not all, of these concepts can be taken outside of the web application context and applied in other domains as well. For the sake of clarity we will talk about these concepts within the context of this document.

2.1 Modern Web Applications

The typical abbreviation used to refer to a web application is **Web App**. This is a very difficult concept to define. We all might have our own idea of what a web application is, and we all might be right. A good definition might be:

> An Open Web App is much more than an HTML application. Think of "Open Web Apps" (OWA) as Web-standards-built applications or "Web Run Time" (WebRT) applications.
>
> A WebRT application runs outside a normal browser and can be launched from the desktop (or device equivalent) on any of the popular internet-connected devices. Of course, WebRT applications can be run inside the browser too if the user prefers. [1]

Now, the definition above is a targeted towards a concept that Mozilla is developing and that is what they call Open Web App. Further in that document, they state why they call it *open* but we are going to stay away from that word in this document, the assumption is that web apps are essentially open.

Also, further in that document, there is a clear explanation on what the main difference between a web app and a traditional web site is. For this document, the following list of characteristics are used:

A Web App:

- May run outside a traditional browser window.
- May access a limited amount of hardware features and data.

- Does not require manual download, installation or updating.
- May be used as a stand-alone application by itself.
- Uses web standards and technologies for its architecture and implementation.
- May be launched in the same way as other native apps.
- Is portable and platform agnostic.

The list above gives us a better idea on what a web app is and what problems may be solved by taking this approach. Writing a definition explicitly is tricky because this concept is subjective. The characteristics that describe a web app today may change and be completely different in a few years.

Recently, the World Wide Web Consortium (W3C) has put efforts together and created a working group that focuses on creating standardized specifications to develop web apps, including existing and new Application Programming Interfaces, APIs.

> The W3C Web Applications (WebApps) Working Group is chartered to develop specifications for webapps, including standard APIs for client-side development, and a packaging format for installable webapps. This work will include both documenting existing APIs such as XMLHttpRequest and developing new APIs in order to enable richer web applications. [8]

The consortium is pushing really interesting initiatives that will make web applications a lot more rich and feature full, improving regular web technologies, such as HTML, CSS, SVG, and JavaScript, with all the resources packed in a single file. The functionality that one could expect from a web app may include online and offline mode as well as device functionalities such as access system resources just like a native app would.

The work W3C is putting together enlightens the future of WebApps and it will enable developers around the globe to create, pack and deliver web apps on top of solid standards.

So far, we have only seen the tip of the iceberg regarding the possibilities in the future for web apps. This will affect the way we consume and push out content and our lifestyle, making it possible to access data on demand on anything from anywhere and everywhere.

2.1.1 Source, Platform & Service

Throughout this document I will be referred to as **source**. The source of all information and the virtual space where data lives.

Platform involves software. The operating system (browser) used to render the application and the hardware used to host that operating system. Platform enables communication with source.

Service, is the actual application. In this case, service refers to the application, usually within the platform's software through which the users access the platform.

2.1.2 Native and Hybrid Applications

Hybrid and Native applications are also difficult to define. Margaret Rouse said it well:

> A native application (native app) is an application program that has been developed for use on a particular platform or device." [2] "A hybrid application (hybrid app) is one that combines elements of both native and Web applications. [3]

The concept native and hybrid are used in the document mostly for comparison or reference. Most common characteristics of native applications are listed below.

A Native Application:

- Has source code executable only by a particular platform.
- May use OS's APIs and hardware features.
- Usually, launched from the home screen.
- Does not require an external web interface (browser).
- Requires manual download, installation and updating.
- Usually, is distributed through vendor specific app store.
- Requires vendor specific app store approval
- Often, has a consistent look and feel with that of the platform.

The list gives a rough idea of what native applications are and helps identify a web app from a native app. Personally, I consider hybrid applications to be more abstract so this

term will be used solely to represent the concept of an application that contains elements of both native and web.

There are technologies that enable developers to deploy applications created using standard web technologies onto native platforms. These technologies allow to wrap web app code into an installable native app which then renders the web views.

2.2 Think Mobile First

Before the mobile revolution, web designers and developers had the freedom to design for relatively large screen sizes and techniques as the grid and 960px wide were used to create websites that rendered more or less consistently across different display sizes. Before mobile became big there was much less focus put into performance or optimization for a particular render engine.

Web technologies are improving enabling developers and business to create impressive mobile services that are responsive and work consistently in a wide array of devices. Thinking mobile first, we get rid of the belief that the targeted user will be anchored to a desk or a non-mobile station. We assume the user is mobile by nature and we can deliver an application that adapts and responds to the user's needs. For example, let's say we are designing a video publishing application, we could provide a responsive visual design and an adaptive architecture to deliver a web app that is available to consumers from a mobile phone, a tablet or even on a 52" smart TV.

The concept aims to find the device that represents a middle point from the array of targeted devices, in other words, find the middle of the spectrum considering processing power, screen resolutions and type of user interface. It is more efficient to design and optimize for a device that is somewhere in the middle of the spectrum and then adapt as we go on to more or less powerful devices with larger or smaller screens.

However, it is important to keep in mind that a big screen does not mean the device has a large amount of processing power or even a well known rendering engine for the built-in browser. The device could be a smart TV with a custom implementation of a webkit browser, and limited abilities of navigation and processing power.

2.3 Modular Web Design

Modular Web Design means to create a component library for a web service. Each component can be optimized for user experience and re-used in many ways across the web service. Each component presents a chunk of the web page.

Re-using the common components designers and developers can produce wire-frames, mock-ups and mark-up far more efficiently. When design team works with developers in collaboration, they can improve the quality and make web site development faster. [6]

Often, it is important to think about a web page from the Designer's point of view. This means defining different components on the web page, as well as dividing and breaking down the design into meaningful chunks and look how each component changes under different conditions.

When components or their properties can be re-used in multiple situations, the developer can use them as core modules and influence the whole at once. Components are usually mixed with design patterns since both are combined into a library, a toolbox for designers.

The components represent chunks of web design and code for a web site. Design patterns are used across the components to give a web site a uniform look and feel and components are inspired by design patterns.

2.4 The Module Pattern

The module pattern is the only design pattern that is going to be defined in this document. The purpose is to provide a very simple explanation of what it does and why it was created as well as other relevant information.

The concept of what a design pattern is, in software engineering, is open to interpretation. An earlier publication on design patterns for buildings and construction, from 1977, offers an excellent definition.

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. [4,3]

Design patterns are needed because they provide proven solutions, to a given problem. One of the most common problems that affected large JavaScript applications is *chaos*. The language does not provide an obvious solution to keep the application code organized, comprehensive and maintainable like other languages do.

Over the time JavaScript authors have been experimenting with different alternatives, patterns and conventions to go around this issue aiming to create a practical solution and define the steps one may follow to take this solution into use. In other words, create a design pattern for this particular problem.

The core idea of creating modules in JavaScript is to create isolated pieces of code that work independently of the rest of the application, in other words, isolation and reusability. The pattern offers many advantages and has been used for a few years. It is well know within the development community since Eric Miraglia, from Yahoo's YUI, blogged about it in 2007. [5]

One might be wondering why we would need to use modules when creating large JavaScript applications and the answer is not straightforward. There are several reasons why one would want to implement the principles of the module pattern when designing and planning to build a large JavaScript application. The most common reasons are listed below:

The application would:

- Be scalable
- Be team-driven
- Be localized
- Have coss-instatiation private variables
- Be extensible
- Be deferrable (put up to a later time)
- Avoid global variables and namespacing
- Have implied encapsulation, no more object nesting
- Be able to mange dependencies

Although this list makes sense, why are we building complex applications with pure JavaScript instead of using other, non-procedural, languages like Java or C++? The reason we want to use JavaScript to build web apps is because, at the moment, it is the language that allows us to do so. In other words, we have no other choice – at least non that is a web standard.

A web app needs to have as few calls to the remote server as possible, and all the heavy work happens client-side, right there in the browser. This is necessary so the application truly fulfils the core elements of a web app stated earlier.

JavaScript was not really designed and built to create this type of client-side complexity and we must be careful when designing and strive to find an architecture that makes sense. This is often very difficult to achieve, therefore, we make use of design patterns and specifically of the module pattern. [12]

2.5 AMD: Asynchronous Module Definition

Now that we have defined and understood why we would need to implement the module pattern let’s go into AMD, Asynchronous Module Definition. AMD is a JavaScript API, a standard for defining modules in JavaScript. It provides a simple and adjustable module format with forward compatibility design with browser environment in mind and to accommodate asynchronous loading. AMD enables us to load modules in parallel, as asynchronicity is a property of the browser, browser modules should be asynchronous.[12]

As JavaScript is becoming “the language” to write client-side applications, there is a need to organize the application into several script files. Otherwise it would be just a massive JavaScript file that can scroll forever, almost impossible to maintain and extend by human developers. Then, inspired by the module pattern, script loaders emerged.

Script loaders are JavaScript libraries that enable us to load scripts that contain pieces of code, referred to as a *modules*, and make those available only when needed. This takes a big weight of a developer’s back and makes the application more readable, easier to maintain, faster to debug and most importantly scalable and extensible. All

modules are independent and isolated. Modules communicate with each other not through direct calls but through event agreements. [11]

To this day, the most popular script loaders use AMD's advantages to import smaller blocks of JavaScript in parallel. When the last imported module has been loaded, the importing module may be evaluated.

2.5.1 Before AMD

It is worthwhile mentioning that before AMD emerged, another very popular method was widely used: CommonJS. In contrast with AMD, CommonJS loads modules synchronously and it covers many other topics as well, for example: packages. CommonJS module logic is widely used to declare modules that may work in a web server. [12]

There are clear differences. In Table 1 there is a list of what these two very powerful module loaders feature and how, depending on what the goal is, they can be the most appropriate tool to require, load and define modules. Most of these are differences between both formats, and they are both valid in server and client side. Which approach to use is tightly related to the application architecture and technical specifications, as well as the environment on which it will be developed.

CommonJS:	**AMD:**
Server-first approach	Browser-first approach
Synchronous	Asynchronous
Forward Looking	Support for plugins
Single module type: objects	Supports several module types
Implicitly wrapped modules	Explicitly wrapped modules (define)
Clean slate, no global baggage	Simplifies backward compatibility

Table 1. CommonJS and AMD features and basic characteristics

The main reason why we would use AMD instead of CommonJS is because we would be developing a one page Ajax driven JavaScript application; in other words, we want a module loader that has been thought out for use in a browser-first environment.

2.5.2 Why AMD?

Programming is all about solving problems and after the programmer has understood the problem at hand, the main objective is to determine the steps or actions needed to find a suitable solution. To write efficient programs and specially to solve very complex problems, the most common first action to take is to divide the problem into smaller, easier, problems.

The advantages of *divide and conquer*, are compelling. Dividing a complex problem into smaller problems with a common goal helps programmers to really understand it and to localize issues and possible bugs and encapsulate disasters that may occur in the program.

Creating applications based on modules is no different from that. It helps to understand, encapsulate and test individual pieces of code. Once each piece has been completed and tested, they can be assembled together in harmony and with full control over them. Each module will know nothing about what they are part of but instead, modules use interfaces to interact with their external environment.

In my previous experience developing a single page web app using modern web technologies, we did not have a proper module loader in place and instead another approach was used. References to “modules” done via global variable names that were loaded in another script in a particular order. Modules were processed in server side as well as minification and concatenation.

At that time AMD was gaining popularity and libraries like RequireJS where just maturing into a solid solution. However, time and other constraints did not allow the team to implement any AMD solution properly.

The architecture was working fine, but as the application became more complex and new developers joined the team, there were several problems. The development process became tedious, slow and challenging to debug.

Ugly hacks, redundant code, global space pollution and overall lack of encapsulation in the application were just some of the problems. These could have been resolved or, at the very least, minimized by using the module pattern.

In the example project, we had the chance to start over from ground zero and build a new application this time putting an AMD solution, RequireJS, in place from the start. This enabled the team to work on individual modules and disclose the application into smaller pieces, making the whole development process more enjoyable and less tedious.

Moreover, the team had the opportunity to create components that may be re-used in other projects using similar application architecture and even for this particular client case. The aim is to build an application once for a TV channel and use that same core structure and architecture to create similar applications for the rest of the channels available to the public.

Currently, JavaScript does not offer a built-in module loader and given the versatility of the language there are several ways to modularize a particular piece of code depending on which environment it is going to be executed onto. [11,12]

There are two main solutions that we have explored earlier in the document, AMD and CommonJS. AMD offers clear advantages to use in the browser. A typical AMD module is defined as follows:

```
define(id?, dependencies?, factory);
```

In the call above, **`id`** is an optional identifier of the module being defined; **`dependencies`** is an optional array of dependencies used by de module and **`factory`** is a function that returns the defined module.

The goal here is NOT to sell the idea of using AMD or a module pattern approach, but simply to demonstrate in practice how the process goes and what are the benefits to build an application for modern browsers using an AMD library, in this case RequireJS. From the options available today RequireJS is the most optimal library for the use case presented.

3 Specifications and Project Description

3.1 Client Case

The client is a well known broadcasting company with several channels. In this study, we are focusing on one specific channel, the name Channel 1 will be used for reference.

The ultimate goal with this project is to create a solution that will be deployed for several channels, same feel, structure, functionality and architecture will be used combined with a unique look and theme for each individual channel.

Conmio's strategy is to create a solution for Channel 1 first and then abstract that into a core code base, creating the moving force for all of versions deployed to different servers. The final result is a Grails plugin that will be used to create individual applications and as many iterations as needed.

3.2 Targets

The concept is to create a video driven application that aims to provide a standardized model with alternating elements that may be developed and deployed to various instances of similar projects. Like broadcast media, the idea is to create an application with options for elements that respond to the device and display in use.

This front-end application will have an iPad driven layout, compatible with iOS 4.x and main Android 4.x upwards tablets. The feed structure is built to consider the most used objects of a video-type service, although not all fields are mandatory. Project-specific settings determine which parts of the feed are utilized.

The main goals are:

- Create a web application
- Create a modular architecture
- Use AMD for module loading and dependency management
- Build as many components as possible

- Build one web app, extract the core and use it to build customized iterations

3.3 Application Architecture

Conmio's front-end components are utilizing back-end components via HTTP and accessing directly third-party API's such as social media and advertising APIs. Back-end components are utilizing the client's APIs for data source, which are usually in xml or JSON format.

Figure 1 illustrates the basic application structure. The back-end module parses the data from the content Application Programming Interfaces (APIs) and renders a new RESTful API that the front-end then can process via an HTTP get request. Basically data is requested through an XMLHttpRequest (XHR) object and using JavaScript it is modelled and stored in data objects.

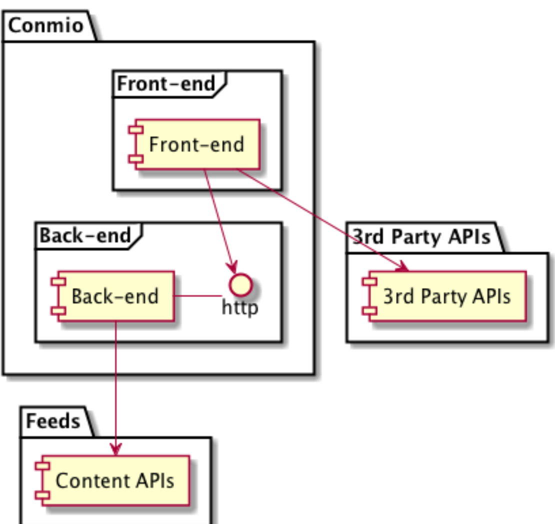

Figure 1. Basic Application Architecture.

Back-end components are created using Grails (Groovy on Rails) and their functionalities will not be described in details, as this is a client-side application we will disregard details of the back-end process for simplicity. Front-end models are described more in details in the Data models subchapter.

3.3.1 Data models

Data models are modelled using JavaScript. A *collection* object represents an array or collection of objects. It can be extended to form collections of models, for example, the *articles* collection object refers to a collection of *article* models.

The *locations* object is based on a locally hosted data file in JSON format. Once again, we create a collection of *location* models. Each of these *location* models store relevant information about a particular location or section in the application. This object is used to better represent the state of the application.

For example, we use a *CurrenLocation* model to know at all time where in the app the user is and then we could bind to the changes of this model to update the UI and inform the user what is the current state of the app. This is particularly useful for navigation.

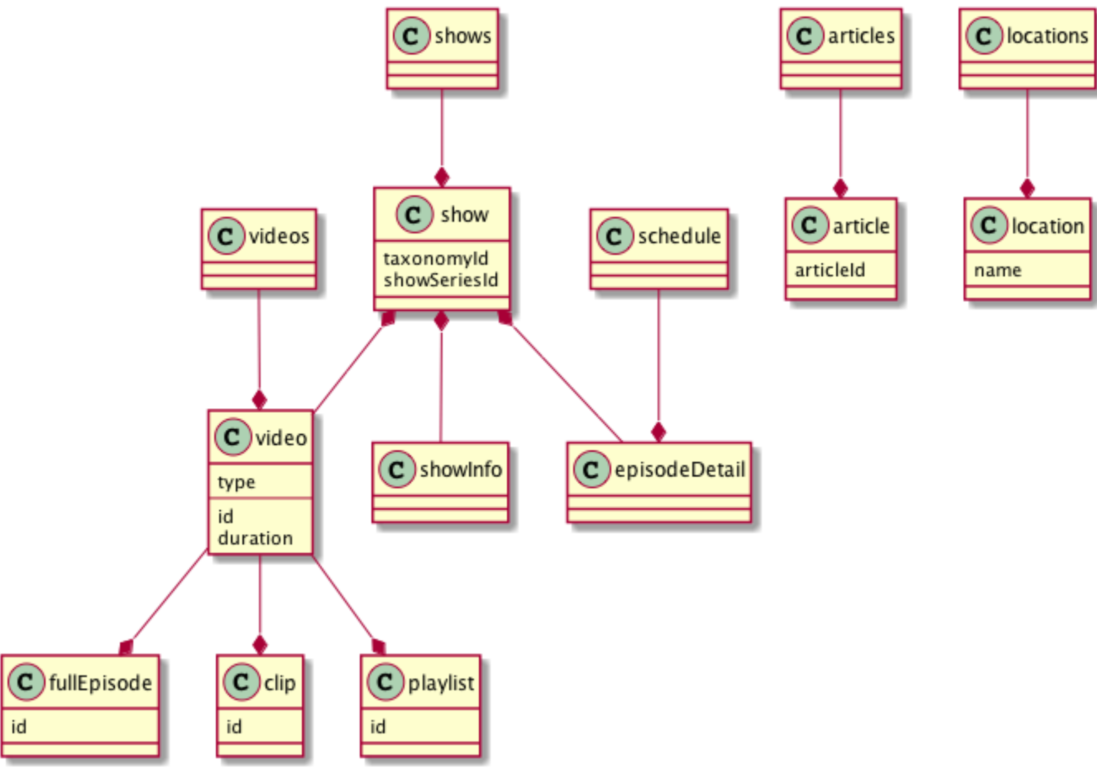

Figure 2. Data models.

The diagram in figure 2 represent JavaScript Objects, usually in JSON format and each of the objects maps to one API call. There are four basic objects: *videos, show, schedule* and *articles*. These objects map to the corresponding API call and will be modeled using the response data.

The application data models have been designed and implemented with a specific type of content in mind, audio-visual content in the form of video clips. The diagram in figure 2, illustrate the relation between the models used in the application. These models should allow expansion and adaptation to similar video based applications for a different channel with the least possible changes, possibly as simple as passing a configuration object to the main app initializer.

3.3.2 Application structure

The application structure in this chapter refers to the front-end part of the application, where most of the processing is happening. The directory and module organization is categorized depending on the type of module being created.

The main groups are Models, Collections, and Views. The basic application directory structure is shown in Figure 3.

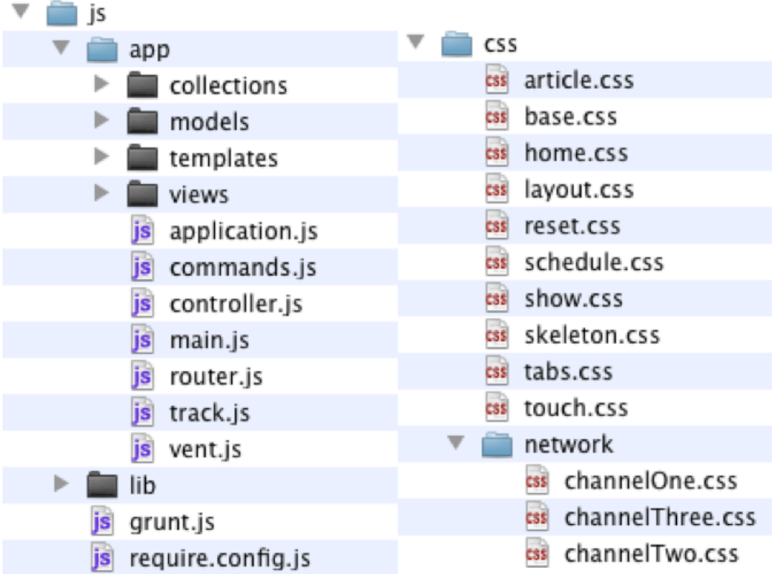

Figure 3. Application Front End directory structure.

The development team chose to organize the application structure based on what is familiar already. Grails framework utilizes a standard Model View Controller (MVC) architecture where the main goal is to keep data (model) and user interface (views) separated and use a controller to communicate and basically control the application. The reason to do this is to have a loosely coupled application structure thus changing the data and business logic (functionality of the application) may be done without affecting the user interface and vice versa.

The modern web application utilizes concepts of traditional software architecture previously used server side in the web or for native platform software. At Conmio, an open source JavaScript library, BackboneJS, is used to transfer some of the main concepts of MVC into the client side. BackboneJS is not truly MVC but it could very well be used as such, which is why the application structure has been organized this way. It improves readability and prevents chaos and helps new developers already familiar with traditional MVC get started.

3.3.3 Application Programming Interface (API)

The APIs are organized in the form of JSON objects. The following there is an example of a typical response to an API call.

Show detail response

```
[
   {
   showId: 123,
   showSeriesId: 456,
   name: "Show name",
   about: "Info about show",
   aboutImageURL: "http://foo.bar",
   latestClipId: 789,
   episodes: [
           {
               startDate: "Jun 03",
               startTime: "4:00pm",
               duration: 30,
               description: "Show description...",
               title: "Show Name",
               number: 1,
               highDef: true,
               parentalRating: "TV-PG",
               cc: true,
```

```
                showTitle: "Show Title”
            }
            {
                startDate: "Jun 03",
                startTime: "4:00pm",
                duration: 30,
                description: "Show description...",
                title: "Show Name",
                number: 1,
                highDef: true,
                parentalRating: "TV-PG",
                cc: true,
                showTitle: "Show Title”
            }
        ]
    }
]
```

Code example by Conmio Development Team

Some local configuration objects are used; on the back end there is module configuration for each of the iterations and together with specific controllers create a unique Grails back-end to serve correct data and styles to each of the channels available.

Each of the APIs response corresponds to a Model or a Collection of models; these are modelled in the client side using a JavaScript framework, BackboneJs. This library will be presented in the Toolbox chapter.

3.4 Toolbox: libraries chosen

When one needs to choose a *toolbox* to build and application or any type of software, for that matter, there are a number of things to consider and choosing the “right tool for the job” is subjective.

In the JavaScript ecosystem there are often millions of frameworks and libraries that solve common problems and it can get really difficult to choose between them. For this particular implementation, the toolbox has been set up by senior developers together with the management considering aspects such as: community involvement, enhancements planned, popularity, documentation and design principles.

The following subchapters provide basic understanding of the libraries chosen for the implementation and how they can benefit this particular type of application.

3.4.1 BackboneJS

BackboneJS is a JavaScript library that emerged on the need of creating structure in large JavaScript applications. The main objective of BackboneJS is to give coherent structure to our application by enabling developers to represent data as objects/models, organize our models into collections and finally hook up those data models to the Document Object Model (DOM) through views.

It is worth mentioning that this is not a traditional Model-View-Controller approach. Ideally, a backbone application would work as the following diagram illustrates.

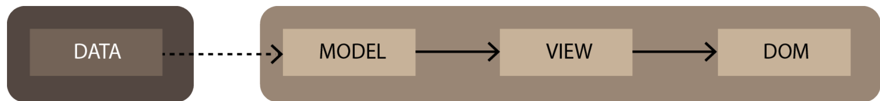

Figure 4. BackboneJS flow.

As we can see in figure 4, the data is synchronized from the server, abstracted and then encapsulated into a model. The model then provides the data to the view and the view is responsible to build the HTML which then, eventually, may be put back into the DOM.

With this flow we can add a very good and maintainable structure to our application and avoid some of the problems that would otherwise be inevitable in this type of application. BackboneJS is also a very minimalistic library with only 800 lines of code.

3.4.2 Backbone.Marionette

Backbone is a great tool, but its minimalistic approach leaves out a few important things, like composite views, for example. This might seem trivial on a sample application where complexity is at a basic level, but a backbone view might evolve into a big mess and we'll end up without any structure. This is what Backbone aims to solve in the first place.

To go around these issues, we can use a library called Marionette, which pretty much adds a layer on top of Backbone that enables us to create composite views, items views, collection views and layouts with region managers.

Marionette enables the non-trivial application developer to increase productivity. Above what Backbone already provides, it simplifies even more the creation of large JavaScript applications and it does so adding a set of design and implementation patterns commonly found in applications that the creator and contributors have been using to build Backbone apps. [7]

Marionette is utilized to make the architecture more modular, reusable, event-driven and, most importantly, more structured. Basically, Marionette, allows us to successfully build a complex application at view level and yet keep it organized, scalable and modular. There are clear benefits of using Marionette vs. plain Backbone and the key points are:

- Scalability with modular architecture driven by events
- Easier to modify and adjust for application specific needs
- Less boilerplate introducing view types that are specialized
- Module oriented architecture, application and modules attach to it
- Visuals may be composed at runtime with a Layout view and regions in it
- View and layout nesting made easy within visual regions.
- Memory management is build-in for views, regions, and layouts
- EventBinder and event cleaning easier and built-in
- Event or Command driven application interfacing
- Flexible, architect may pick to use only what needed [7,195]

3.4.3 Underscore and Lodash

UnderscoreJS is a very useful utility belt that provides a lot of good helper methods that are frequently used in large JavaScript applications and compliment backbone.marionette's functionality. BackboneJS uses UnderscoreJS as a dependency and it has been basically built on top of it. Lodash claims to be a drop in replacement for UnderscoreJS adding AMD support and performance improvements

on `_.each()` and other looping methods. However, the latest version of RequireJS supports loading non-AMD-ready modules such as BackboneJS and UnderscoreJS through a shim configuration.

3.4.4 Handlebars

Conmio's web apps use JavaScript to create dynamic interfaces. DOM manipulation is costly especially when changing huge chunks of the document with each change of the view; that is when, a templating engine can make a difference.

The "logic-less templating" principle works very well in general, but sometimes a bit more than that is needed. HandlebarsJS library offers templates that could be precompiled instead of having to be compiled on the client side. This reduces time dealing with the issues that sometimes occur without logic in the templates.

UnderscoreJS provides a minimalistic templating engine. In this implementation, *Backbone.Marionette.Handlebars* library is utilized, which is a combo that provides a good blend of the HandlebarsJS's goodies with the tenacity of Marionette.

HandlerbarsJS allows us to also use basic looping and logic in the templates, which might be dangerous. It is up to the developer to know when using logic in the templates seems like a good idea and when to avoid it. Most templates should be straight forward and should contain no logic, but at times complexity may not be optimally handled in the view layer of the application. Thus putting simple logic in the template is accepted.

3.4.5 RequireJS

RequireJS is the most popular script loader and dependency manager. This library is simple to use because it follows AMD format. RequireJS API evaluates dependencies eagerly, rather than lazily. In other words, the dependencies are evaluated ahead, i.e., before they are needed, rather than delay the evaluation until they are strictly required. In practice, RequireJS is completely compatible with CommonJS modules, requiring only different wrapping.

To load JavaScript files, just pass their paths to the **`require()`** function, specifying a callback that will be invoked when the dependencies are all loaded. Below, the *application* and *example* modules are passed as arguments to the callback; they do not have to be fetched with the **`require()`** function.

```
require(
   ["path/application", "path/example"],
   function(app, example) {
        // Modules loaded!
   }
);
```

It is not just modules that you can require but also ordinary JavaScript libraries as dependencies [10]. Other libraries will work, but they will not be passed correctly as arguments to the required callback. However, any library that has dependencies is required to use the module format.

To help with optimization, RequireJS encourages you to place your initial script loader in a separate file. The library even provides shorthand to do, simply setting a *data-main* attribute in your script tag as presented below:

```
<script data-main="path/config.js" src="path/require.js"></script>
```

3.4.6 Other Libraries

There were also other libraries used in this application that will not be described in detail. However, it is worthwhile pointing them out and the main purpose for including them in this project.

The secondary libraries used are:

1. jquery: widely used library for JavaScript development, backbone depends on it.
2. juissi.swipe: Conmio’s swiping and scrolling solution mostly used for custom smooth scrolling and building paginable carousels.
3. moment.js: better date handling for JavaScript
4. i18nprecompile: used mostly for internationalization (if needed)
5. json2.js: used to ensure availability of a JSON property in the global object

6. overthrow.js: overflow support, emphasis on native implementations ensuring cross-browser usability.
7. Fastclick.js: instant click for touch events, no more delay
8. Advertising: third party libs used for advertising and tracking

3.5 Implementation

Initially, we set up a Grails back end that parses feeds provided by the client and serves an API that we can then use with our front-end implementation.

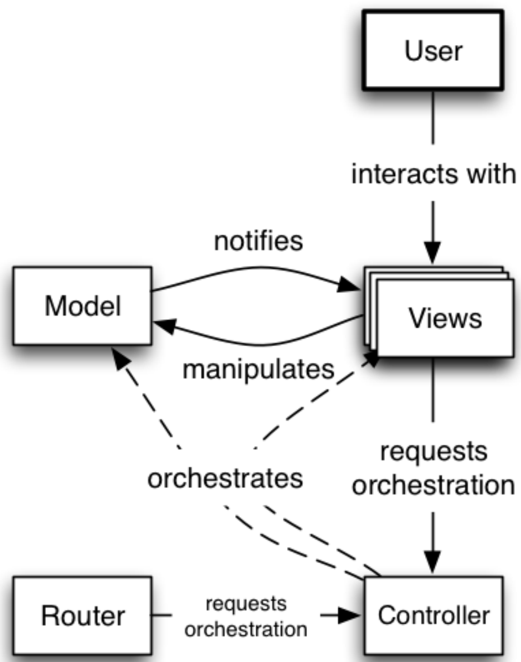

Figure 5. Basic application flow

Each of the API calls provided by our grails application will be mapped to one data model or collection of models. To help us better organize and form these models, we use Backbone framework which enables us to create Models and Collections of models with useful methods. This way we can easily organize our data and follow the documented architecture (see figure 2 for reference).

Figure 5 shows the basic flow for this particular example application, mostly using events and commands to communicate between the different components.

Marionette let us create views, that are used to render pages where users can consume the contents of our data models. HandlebarJS templates help to lay down those views into the browser through HTML tags.

Being a large scale, component driven application, we must follow the module pattern to organize our app. RequireJS will enable us to manage dependencies and modules effectively and effortlessly compelling to AMD style and syntax.

Finally, we use other helper libraries, like jQuery, to manage how our application reacts to user interactions and others, like JuissiSwipe, to enhance the user experience by adding custom swiping support to our web app.

3.5.1 Configuration and initial set up

The mobile (phone) version of this application was already implemented, so we start by creating a new grails view for the tablet implementation and we use Conmio's plug-in DeviceGate to detect the device and serve the correct view.

The grails view for tablet looks like this.

index.gsp

```
<!DOCTYPE html>
<html>
 <head>
   <title>Web App</title>
   <meta name="viewport"
    content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">
   <!-- CSS  -->
   <link rel="stylesheet" href="/tablet/css/reset.css">
   <link rel="stylesheet" href="/tablet/css/touch.css">
   <!-- JS -->
   <script type="text/javascript" src="/tablet/js/lib/require.min.js"></script>
   <script type="text/javascript" src="/tablet/js/require.config.js"></script>
 </head>
 <body>
 </body>
</html>
```

Code example by Conmio Development Team

In this file we load require and the configuration only. That way we minimize the number of script tags in our application. The require configuration will contain almost all of the libraries used, as well as the dependencies that will be loaded initially, to kick start the app.

The require configuration file looks like this:

tablet/js/require.config.js

```
require.config({
    shim: {
        backbone: {
            deps: ['underscore', 'jquery'],
            exports: 'Backbone'
        },
        "freewheel.admanager": {
            deps: ['jquery'],
            exports: "tv.freewheel.SDK"
        },
        "juissi.swipe": {
            deps: ['jquery'],
            exports: 'Conmio'
        },
        "sitecatalyst": {
            exports: "s"
        }
    },
    hbs: {
        disableI18n: true
    },
    paths: {
        jquery                          : "lib/jquery",
        backbone                        : "lib/backbone",
        "backbone.eventbinder"          : "lib/backbone.eventbinder",
        "backbone.wreqr"                : "lib/backbone.wreqr",
        "backbone.babysitter"           : "lib/backbone.babysitter",
        "backbone.marionette"           : "lib/backbone.marionette",
        underscore                      : "lib/underscore ",
        "Handlebars"                    : "lib/handlebars ",
        hbs                             : "lib/hbs ",
        json2                           : "lib/json2",
        i18nprecompile                  : "lib/i18nprecompile",
        "backbone.marionette.handlebars": "lib/backbone.marionette.handlebars",
        "juissi.swipe"                  : "lib/juissi.swipe",
        "moment"                        : "lib/moment.min",
        "freewheel.admanager"           : "lib/freewheel.admanager",
        "overthrow"                     : "lib/overthrow",
        "fastclick"                     : "lib/fastclick",
        "sitecatalyst"                  : "lib/s_code"
    },
    deps: [
        "backbone.marionette.handlebars",
```

```
        "juissi.swipe",
        "overthrow",
        "app/main"
    ]
});
```

Code example by Conmio Development Team

The above shows a typical configuration structure for a web app of this kind. There are some libraries that are not AMD compatible, those may be put into a ***shim*** object like in our config file because backbone is not AMD compatible, so we add its dependencies manually and explicitly define what object it exports, this object will be exported and attached to the window object.

Note the last part of the configuration object is an array of dependencies assigned to the property ***deps***. These will be required and loaded as soon as **`require()`** is defined but the difference is that it will wait for the loader to process the configuration. In other words, it is a way to specify an array of modules to load asynchronously together with the configuration object and without blocking any other calls to **`require()`**.

Next we create the main module and the application module. In the main module we start up the app and do minor work, like adding regions and loading the basic home layout.

app/main.js

```
require(
    [
        "backbone.marionette",
        "app/application",
        "app/router",
        "app/views/baseLayout"
    ],
    function(
        Marionette,
        application,
        Router,
        baseLayout,
        FastClick
        ){
        "use strict";
        //application regions
        application.addRegions({
            baseRegion: "body"
        });
        application.addInitializer(function () {
            //show the base layout and start the router
```

```
            application.baseRegion.show(baseLayout);
            Router.start();
        });
       //finally, start the app
        application.start();
    }
);
```

Code example by Conmio Development Team

Note that we are calling **require()** in this module as we want this to be executed initially and together with the configuration. Within the application module we instantiate a new Marionette.Application object.

app/application.js

```
define(
    "app/application",
    ["backbone.marionette"],
    function (Marionette) {
        "use strict";
        return new Marionette.Application({
            currentShow: {},
            currentVideoId: null
        });
    }
);
```

Code example by Conmio Development Team

Note that this module has been named, which is the first argument in the *define()* call. The reason we explicitly name this module is because, even though, the RequireJS optimization tool can generate the module names automatically, we have not yet experimented using it in Grails resources plugin context. The very least we can do is concatenate and minify all the files, which will cause module information to be removed unless it is explicitly defined.

Otherwise, a very simple module, it returns a new Marionette.Application object with some default configuration properties and its initial values. This is the root of our backbone/marionette application.

Lastly we must handle state of the application, in other words, the URL and its changes. For that we use a Marionette.AppRouter and a corresponding controller. In

this context the controller serves as a mediator between the views and the router. They mostly send events or execute commands when the state of the app changes. The URL state or app state will be handled by the router and this will then call function defined in the controller.

app/router.js

```
define(
    "app/router",
    [
        "backbone.marionette",
        "app/controller"
    ],
    function(Marionette, Controller) {
        "use strict";
        var AppRouter = Marionette.AppRouter.extend({
           controller: Controller,
            appRoutes: {
                // Section fronts
                "": "home",
                ":section": "section",
                // Show page routes
                "show/:showId" : "show",
                "show/:showId /video/:videoid" : "video",
                // Articles
                "article/:articleid" : "articles"
            },
            start: function() {
                Backbone.history.start();
            }
        });

        return new AppRouter();
    }
);
```

Code example by Conmio Development Team

app/controller.js

```
define(
    ["app/vent", "app/commands"],
    function(Vent, commands) {
        return {
            home: function(){
                commands.execute("goToSection", "home");
            },
            section: function(section){
                commands.execute("goToSection", sectionObj);
            },
            show: function(showId){
                Vent.trigger('show:video', showObj);
```

```
                commands.execute("goToSection", "show");
                commands.execute("goToVideo", videoConfigObj);
            },
            video: function(showId, videoId){
                commands.execute("goToSection", "show");
                commands.execute("goToVideo", videoConfigObj);
            },
            articles: function(articleId){
                commands.execute("goToSection", "article");
                commands.execute("goToArticle", articleId);
            }
        }
    }
);
```

Code example by Conmio Development Team

Commands and Vent are part of *backbone.wreqr*. ***Vent*** represents an instance of a *Backbone.EventAggregator* object and ***Commands*** and instance of a *Wreqr.Commands* object. We use these objects to help us better handle the applications state and reflect its changes accordingly. More information about those can be found in the Marionette's documentation in github [9].

The controller executes commands or trigger events on the corresponding objects. The application may handle pre-defined commands by adding a command handler or binding to events dispatched by the controller through the event aggregator.

3.5.2 Basic View hierarchy

Up to this point we have set a basic start up for a project of this kind. Typically we will proceed to create the views and layouts for the different parts of the application following the principle stated below:

A view should not have any knowledge of its parent views. Any view is only allowed to reference its sub views, but never its parents.

This principle is very important and following it will prevent many of the most common issues encountered when nesting views, regions and layouts as the application's complexity evolves.

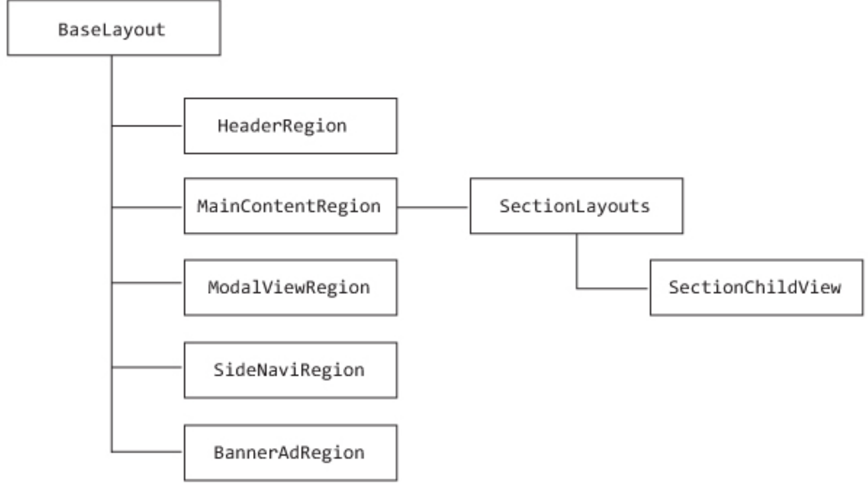

Figure 6. Basic view hierarchy

The following is an example of the home layout code.

app/views/home/homeLayout.js

```
define(
    [
        "backbone.marionette",
        "app/commands",
        "hbs!app/templates/home/homeLayout",
        "app/views/home/myView"
    ],
    function homeLayout(
        Marionette,
        Commands,
        HomeLayoutTemplate,
        myView
        ) {
        "use strict";
        return Marionette.Layout.extend({
            template:{
                type:'handlebars',
                template:HomeLayoutTemplate
            },
            className:'home-page',
            id: 'home-layout',
            regions:{
                regionOne: "#regionOneContainer",
            },
            initialize: function(){
                  //constructor function
            },
            setCollections: function() {
                  //setting the collections for the views
            },
```

```
            onShow: function () {
                //callback for the show event
                //gets called on "show" event
                this.regionOne.show(myView);
            },
        });
    }
);
```

Code example by Conmio Development Team

app/templates/home/homeLayout.hbs

```
<div class="section-front" id="homeLayout">
    <div class="does-not-map-to-a-region"></div>
    <div id="regionOneContainer"></div>
</div>
```

Code example by Conmio Development Team

The base layout is done in a similar fashion. From this code we can assume that the base layout will contain regions for all of the parts stated in figure 6 and views will be shown in their corresponding regions.

Marionette makes it easy to deal with view nesting and it takes care of many of the boiler plate code. For example, there is no need to define a render function for every view, instead there are events and callback for the main functionality of the view. For example, instead of writing a custom render function for every view we can simply define an `onRender` function or bind to the `render` event fired by the marionette view.

3.5.3 Creating components

There are a couple of components that were created for this particular implementation. For the demonstration, one of the components will be disclosed it in detail. The components developed for this app are the following:

- Modal View
- Side Bar View
- Thumbnail View
- Tab View
- Filterable Collection

From these components, the Modal View component will be extracted and explained to further illustrate the idea behind creating components that can be used for different situations if they are abstracted properly.

Modal View Component

The idea of creating a modal view component came out when the development team noticed that the modal window, or modal pop-up overlay was being used for several scenarios within the application. The mission was to abstract the behavior of the modal window and create a component that can be used across the app. The basic functionality of the component is:

- Open the modal view layout from anywhere in the app
- Creates overlay box with opacity on top of the content
- Renders a content container region for any type of view
- Can show a view/layout/app in the content container region
- May be closed by pressing a close GUI button or the overlay
- Should be closed when the application state (URL state) changes

With that functionality laid out, the planning of the component can be executed. Initially, the raw state and functionality of the component is extracted from one of the specifications and UI/UX design documents. Figure 7 shows the raw wireframe, and modal layout with its closing interface as well as the content region described above.

From there on, we sample one of the scenarios, usually the most complex one and create the component based on that. Further abstraction may be required after the component has been used and proven that it can be worked with.

The initial implementation of the component flow is shown in figure 8. User can interact with any given view anywhere within the application context. This will then execute a command passing the view that should be shown in the modal content region together with the command name.

The base layout then adds a command handler that tells the component to render the view and then shows the component layout element within the appropriate region.

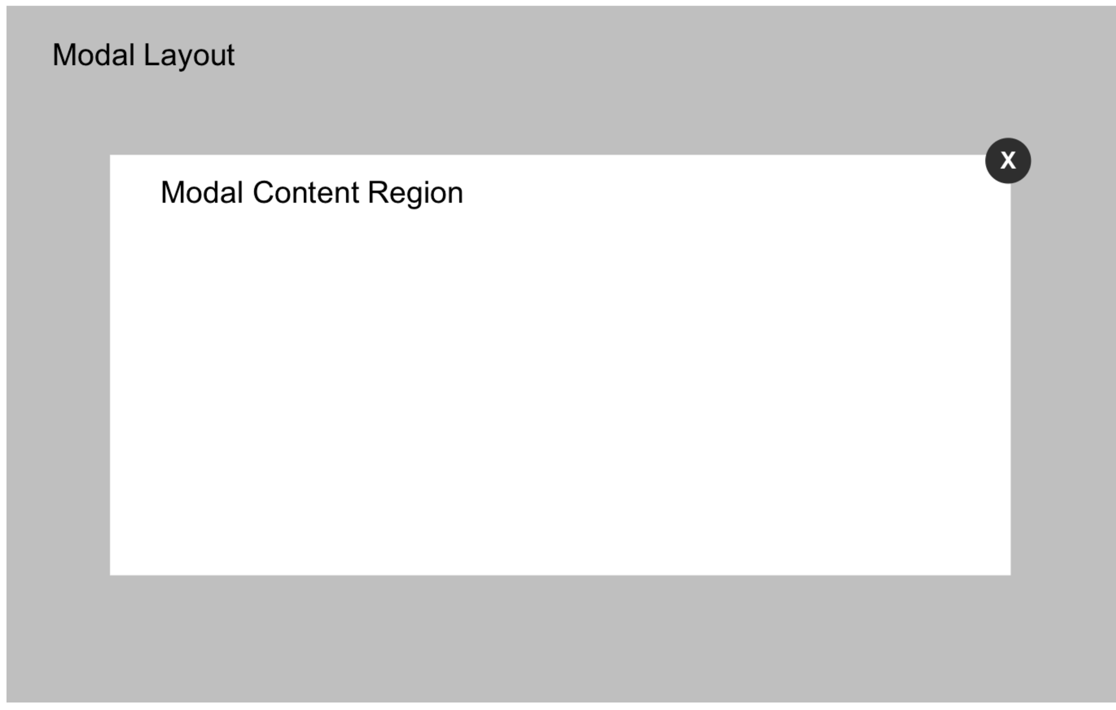

Figure 7. Modal view raw wireframe

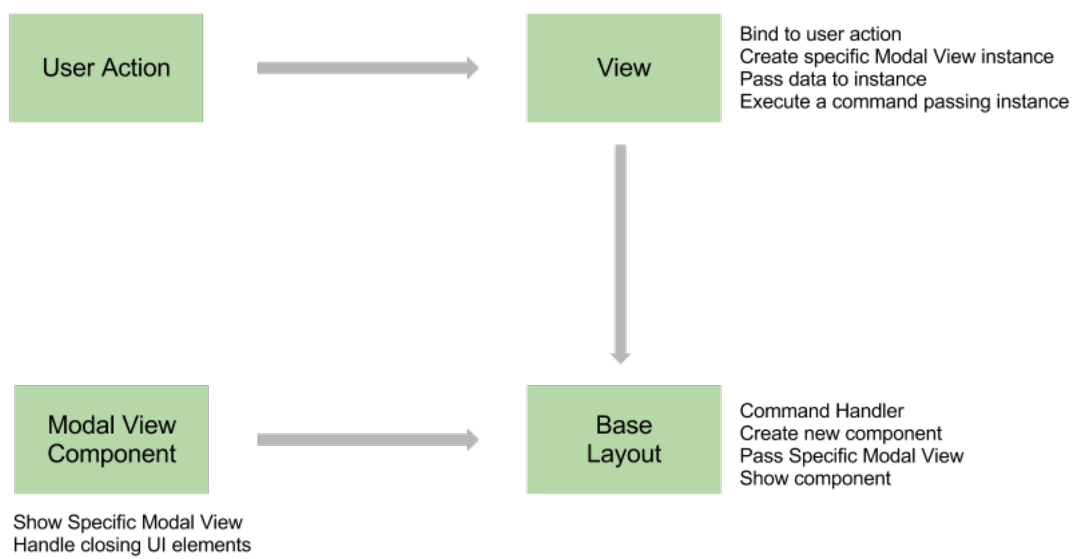

Figure 8. Modal view component flow

The modal view component module is very simple. It contains the basic functionality described above as well as the appropriate template and mark up so it can be styled accordingly.

app/views/ModalView.js

```
define(
    [
        "backbone.marionette",
        "hbs!app/templates/modalLayout"
    ],
    function(Marionette, Template){
        "use strict";
        return Marionette.Layout.extend({
            template: Template,
            className: "modal-popup-container",
            attributes: {
                id: "modalView"
            },
            regions: {
                contentRegion: "#modalContentRegion"
            },
            triggers: {
                "click .modal-close-btn": "closeView",
                "click .modal-overlay": "closeView"
            },
            initialize: function(){
                //close component on route change
                Backbone.history.on("route", this.close, this);
            },
            onRender: function(){
                //show the view set on instatiation
                this.contentRegion.show(this.options.contentView);
            },
            onShow: function(){
                this.on("closeView", this.close);
            }
        });
    }
);
```

Code example by Conmio Development Team

app/templates/modalLayout.hbs

```
<div class="modal-overlay"></div>
<div class="modal-content-container">
    <button class="modal-close-btn"></button>
    <div id="modalContentRegion"></div>
</div>
```

Code example by Conmio Development Team

The command handler to show the modal within the base layout is straightforward. We receive a Backbone view when the command gets executed and pass that view into our base layout, so it can be shown in the appropriate region for the modal view.

This command handler has to be in the same module where we instantiate the base layout. For our example app, this is within the *baseLayout* module.

app/views/baseLayout.js

```
Commands.addHandler("component:modalView", function(view){
      baseLayout.showModalComponent(view);
});
```

Code example by Conmio Development Team

BaseLayout, represents an instance of the baseLayout view object. Within the base layout view code we create a method called ***showModalComponent*** and the view is passed on from the command execution somewhere else in the application. Also, a handler must be added to show the *About Show* modal view within the modal content region executing the component command and passing a view as an argument.

app/views/baseLayout.js

```
showModalComponent: function(view){
      var modalView = new ModalView({
          contentView: view
      });
      this.modalViewRegion.show(modalView);
}
```

Code example by Conmio Development Team

app/views/show/showLayout.js

```
showAbout: function(clickEvent){
      clickEvent.preventDefault();
      //create a new instance of a specific modal view
      var aboutShowModalView = new AboutShowModalView({
          contentType: "about",
          showId: this.model.get("showId")}
      });
      //execute command and pass the view intance
      Commands.execute("component:modalView", aboutShowModalView);
}
```

Code example by Conmio Development Team

Note that when we instantiate a new *ModalView*, we only have to set the content view property to a view instance that we want to show within the modal content region. This component could be further abstracted and it could take more configuration options when creating a new instance. For example, a custom template could be set. For this example application this is enough as all the modals have the same basic template for all use cases.

3.5.4 Making a grails plugin

Further on in our component base modular application development, the next step is to create a grails plugin that can be maintained and developed further as well as include versioning and release notes. Once the first implementation has been completed and it is stable, the core functionality will be extracted to create the core grails plugin.

The core plugin will be used to create three different services and more later. Those tablet applications will be serving three different networks (channels) that the broadcasting company owns and they are all video based services with an article carousel and a schedule page.

Different grails applications are created for each of the services and they all contain the core plugin. Basically what is different and special for each application, is the visual style and content which is unique for each of the services built on top of the grails core plugin.

4 Project Summary

The purpose of this thesis was to improve a development process previously established and allow developers to create new web services with a common core code structure.

This implementation was successfully included in a project at Conmio and transformed into a grails plugin for our Media Broadcasting client case. Finally, the implementation was used to create different tablet sites for the broadcasting networks that the client owns.

The new implementation, made the creation of the additional services much faster compared to the traditional method of creating separate code for each service. The implementation also makes maintenance and error correction easier since all the core elements for the services are centralized.

The first service took around two months to complete including all fixes done for an iPad target device and after that it was moved to the core plugin and successfully used in three different Networks (channels).

The first, channel 1, service was used as a template for the implementation. After the core was complete, the development team was able to create the two other services in less than a week.

4.1 Lessons Learned

Many of the problems encountered in the development process of this application were minor and leaning towards specification changes. At some point the specification changed so much that it was like trying to hit a moving target. Deadlines were short ahead and a new team was formed rapidly when more expertize was needed.

If I look back objectively, the implementation of AMD modules and configuration of require was really not a problem and there is nothing I would change significantly about it. AMD modules work, and they work well. Getting started with RequireJS is really not that hard and creating new modules is a very simple process.

The most challenging part was to understand how the AMD really works. Asynchronicity as a concept is not easy to grasp and AMD really embraces it. It is worthwhile for a developer doing AMD modules to understand that modules are asynchronous. A module might not be ready at a certain point in time and often we must use tools like *promises* and *deferred* objects to help us better write more efficient and understandable algorithms.

Most importantly, a developer working in this type of projects should have a good understanding on how browsers really work. To write efficient code a developer should also understand what is loaded, when and how often it is loaded.

Communication is the key

AMD modules were a natural next step in my learning curve and I had not too much trouble understanding it. When coming from a more traditional JavaScript background, not writing entire apps in JavaScript, it may be overwhelming. There are several new concepts to learn before getting started, which makes documentation a crucial for a project like this.

During the process we learned that it is definitely worthwhile to spend time planning and truly understanding the lay of the land. In other words, make sure that the developer understands the problem and expectations, in the same way as the client, project manager and designers. Also there should be clarity on the limitations or flaws and those need to be communicated early enough and to every party involved.

Improving communication throughout the team not only creates a better sense of understanding and assists to see the objective clearly, but it enhances transparency and productivity within the project. As a consequence, the project flow and implementation process improve when there is a natural distribution of information and knowledge across the development team.

Scrolling and swiping for web apps

Using Marionette and Backbone for this type of applications really makes many tasks simple and saves a lot of time and boilerplate code. The library tends to manipulate the Document Object Model (DOM) quite a lot and takes care of appending, removing, emptying, attaching and detaching DOM nodes from the tree dynamically. This is excellent, and it is something we used to do manually when writting raw backbone views more than a year ago.

Libraries like Juissi.Swipe, Conmio’s solution for scrolling and swiping in the browser, also manipulate the DOM and must attach and detach elements on the fly to ensure performance and a smooth swiping experience. This is often creating conflicts between them. At some point, it became a battle for the DOM, which is the “boss” over elements in the DOM tree.

Custom touch support could be a real challenge and we must take advantage of the browser's support as much as possible. For this project **Overthrow.js** was introduced and utilized as a fall back to make sure native browser's scrolling is enabled when possible. Juissi.Swipe is utilized only when we need pagination or carousels that are complex including both scrolling and swipe-able containers as well as pagination support.

4.2 Improvements for the future

The implementation improved the previous development process, however, there is always room to improve. There are some aspects of the development process that I learned during the implementation stage.

Following there are suggestions that may improve the quality of the development process in future projects and, from a technical perspective, may bring significant impact in the application performance.

Implement LESS and the RequireJS plugin

One should implement less CSS pre-complier as a more appropriate solution for theme/look handling for each of the different services using the core plugin. Adding Less will show significant benefits as we could take advantage of all of the great tools and utilities that it brings to the table. Variables and mixins are some examples of what may be used to generate different themes and style services a lot faster. It will simplify configuration for new services in the future.

The RequireJS LESS plugin allow loading LESS dependencies in the same way as RequireCSS. LESS can then be parsed and built into the script layers naturally as part of the RequireJS workflow. This could be at least used with the components that were created for this project.

Below there is a list of pros and cons which points out the strengths and weaknesses of this project:

Pros

- Easier to manage CSS classes
- Minifies into just one file
- Only one browser request is needed
- No `<link>` needed, the layer is included as a `<script>`
- The page can be displayed while the layer is still loading asynchronously
- Manage templates and their styles together

Cons

- One JavaScript error may leave the entire page unstyled
- Users with disabled JavaScript will not be able to view any styles
- Slightly slower than loading styles directly
- Difficult to catch styling bugs if used incorrectly

Better directory structure

Templates should be included within the module's directory; for example, instead of having a templates directory, the templates will be included in the specific directory within the views directory as shown below:

```
app/
    • models/
    • collections/
    • views/
        ▪ baseLayout.js
        ▪ baseLayout.tmpl.hbs
    • home/
        ▪ homeLayout.js
        ▪ homeLayout.tmpl.hbs
    • show/
        ▪ showLayout.js
        ▪ showLayout.tmpl.hbs
        ▪ showItemView.js
        ▪ showItemView.tmpl.hbs
    • components/
        ▪ modalView.js
        ▪ modalView.tmpl.hbs
```

RequireJS configuration

In the require config file, there are a few changes that may improve the application performance. Using ***map*** and ***paths*** to normalize base modules, at least hbs (RequireJS Handlebars Plugin) uses its own definitions for the same dependencies that other modules use. Using *map* removes the need to duplicate these modules and *paths* definitions reduce the need to prefix core dependencies with *lib/*.

Some libraries also provide a complete bundle, like Marionette newest release includes the Backbone.Wreqr and Backbone.BabySitter prerequisites built in. This could be beneficial and will make our config file a bit cleaner.

Explicitly named modules

Even though explicitly naming modules makes them less portable, we should consider that, we have not yet experimented with using the optimization tool in Grails resources plugin context too much.

The very least we can do is concatenate and minify all the files, which will cause module information to be removed unless it is explicitly defined. It could be a good practice to explicitly name modules or at least the main modules to avoid problems or errors during the optimization process.

Custom Marionette.View for pagination modules

Swiping was definitely a challenge and it was definitely something the team spent some time dealing with. The idea of creating a custom Marionette.View came up as we started to see a pattern.

Creating a custom view will enable developers to create swipeable components faster and easier. The custom view may include the hook and make pagination available locally within the module to avoid having to do it manually. This will add consistency and reduce the amount of code needed to implement pre-defined type of views.

The following code snippet serves as an example of how we could use a custom *JuissiView* to create an article carousel as a Marionette view. This could significantly

simplify building this type of carousels with scrollable containers and we do need to create this for almost every project.

Regardless of what content it renders, some kind of carousel with swiping functionality and scrollable containers within the swipe-able ones, is featured in this type of applications. Therefore, it would make sense to have a custom view in place.

An article carousel module example, using custom Marionette view including the necessary Juissi.Swipe hooks.

```
define(
    "article-carousel",
    [
        "backbone.marionette.juissi",
        "hbs!app/templates/aTemplate"
    ],
    function(Marionette, Template){
        "use strict";
        return Marionette.JuissiView.extend({
            template: Template,
            className: "swipeable-carousel",
            id: "articleCarousel",
            scrollableChildren: this.$el.children()
            paginationContainer: this.$el
        });
    }
);
```

Improve modules and dependency management

There is no doubt that AMD modules already provide proper encapsulation. However, in this application, there are some objects that have been available everywhere, which in theory defeats the purposes of defining AMD modules in the first place.

These objects are available everywhere, without a need to explicitly require them as dependencies, for example, jQuery or Juissi.Swipe objects are available for every new module created within the application context.

Many errors can rise from having these objects available globally and this may violate the principle stated earlier in the document. A view should not be aware of its parent view but only of its child views. Having access to the jQuery object anywhere may create confusion and one could be tempted to manipulate HTML elements from an

incorrect place, in other words, manipulation of the parent view’s HTML elements from within a child view.

This goes against what we are trying to accomplish here and should be prevented by loading libraries as AMD modules for the ones that allows us to do it. In case of in-house libraries we must make them AMD compatible so they can be loaded appropriately.

Testing

There is currently no testing for the JavaScript application in place. Writing basic unit tests and functional tests should be part of any application to ensure quality of the code pushed out to production.

Automated tests should be written for, at the very least, the main functionality of the site. Manual testing was done for the application as part of the quality assurance process but it is well known that automated test brings benefits to the development process and it also enhances the manual testing significantly.

5 Conclusions

Today, Conmio is able to provide an effective solution to distribute digital media for a Client that has a significant amount of content to be published daily. Conmio does that through several channels or networks consistently with a common architecture core keeping their brand and identity throughout all services. This implementation has proven that using AMD modules was a good choice to create an initial implementation and then re-use that to consequently create several iterations for all the different distribution channels to optimize efficiency.

This strategy is not a new one compared to what Conmio has provided for other customer cases in the past. For one client Conmio delivered a framework on which the client was able to successfully deploy over 100 tablet sites.

However, this time, the main objective was to create a loosely coupled modular application, with as many components as possible and ultimately re-use as much code as possible.

The development team successfully created this application using JavaScript basically from scratch using AMD modules to make the application more optimal for development. In other words, the application code base is now not only easier to understand, develop and improve, but it is a lot more granular and modular which makes feature addition and bug fixing a lot more effective. The main advantages of using AMD, for the example application, are:

- Increased development enjoyment
- Faster development velocity
- Easier debugging, expansions and code analysis.
- Better app performance
- Reduced amount of sent requests
- Better asynchronicity, on demand loading
- Clear organization and structure
- Simpler architecture
- Better encapsulation
- Less code written

Over the past few years I have worked in most of the large JavaScript projects that Conmio has developed. This is by far the best way to deal with dependencies and take advantage of the asynchronicity of the browser. There is definitely a new level of confidence when writing code. Everything is a module and errors are within the module and they most likely will not lick or pollute any other modules. The module being created may break but that does not necessarily mean that the whole would break.

The application developed for this client case performs significantly faster than other project we have developed. There are several tools that we are using here to make performance better but the main AMD has definitely made a difference. Therefore, we can require the module only when we need it and its dependencies are loaded accordingly. This impacts the application performance greatly not to mention the organization and structure are comprehensive and more simple.

References

1. Various authors. Mozilla Developer Network (MDN). FAQ's about apps [online]. Last update: MarkGriffin, October 10, 2012 1:44:38 PM. URL: https://developer.mozilla.org/en-US/docs/Apps/FAQs/About_apps Accessed: November 4, 2012

2. Margaret Rouse. Native application (native app) [online]. Search Software Quality. Last update: June, 2011. URL: http://searchsoftwarequality.techtarget.com/definition/ native-application-native-app Accessed: November 4, 2012

3. Margaret Rouse. Hybrid application (hybrid app) [online]. Search Software Quality. Last update: July, 2011. URL: http://searchsoftwarequality.techtarget.com/definition/ hybrid-application-hybrid-app Accessed: November 4, 2012

4. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. USA and Canada: Addison-Wesley Professional Computing Series; 1995.

5. Miraglia Eric, A JavaScript Module Pattern [online]. YUI Blog; July 12th, 2007. URL: http://www.yuiblog.com/blog/2007/06/12/module-pattern/ Accessed: January 4th, 2013.

6. Curtis Nathan. Modular Web Design: Creating Reusable Components for User Experience Design and Documentation. Berkeley, CA: New Riders; 2010.

7. Osmani, Addy. Developing Backbone.js Applications. O'Reilly Media Inc. Ebook Pre-release; April 2012. Last updated: April 13, 2013. Estimated publishing date: USA, May 15, 2013. URL: http://my.safaribooksonline.com/ Accessed: April 12, 2013.

8. W3C WEB APPLICATIONS (WEBAPPS) WORKING GROUP. [online]. http://www.w3.org/2008/webapps/ Last Accessed: 10 May. 2013.

9. Marionettejs.org 2013. Marionette.js - A scalable and composite application architecture for Backbone.js. Website. http://marionettejs.org/ Last read: 10 May. 2013.

10. Requirejs.org 2013. RequireJS - JavaScript file and module loader. Website. http://requirejs.org/ Last read: 10 May. 2013.

11. Zakas, Nicholas. Maintainable Javascript. Sebastopol, CA: O'Rielly Media Inc; May 24, 2012. p. 67-76

12. Osmani, Addy. Learning JavaScript Design Patterns. O'Reilly Media Inc; August 20, 2013. p. 26-37:141-164

6 Screen shots of main views and services

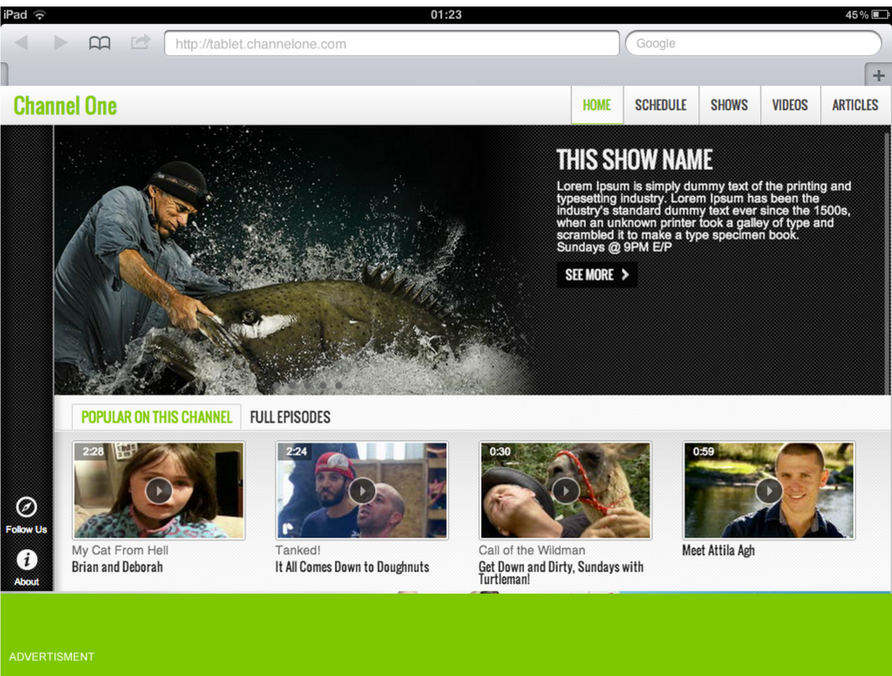

Figure 9. Channel one, initial implementation

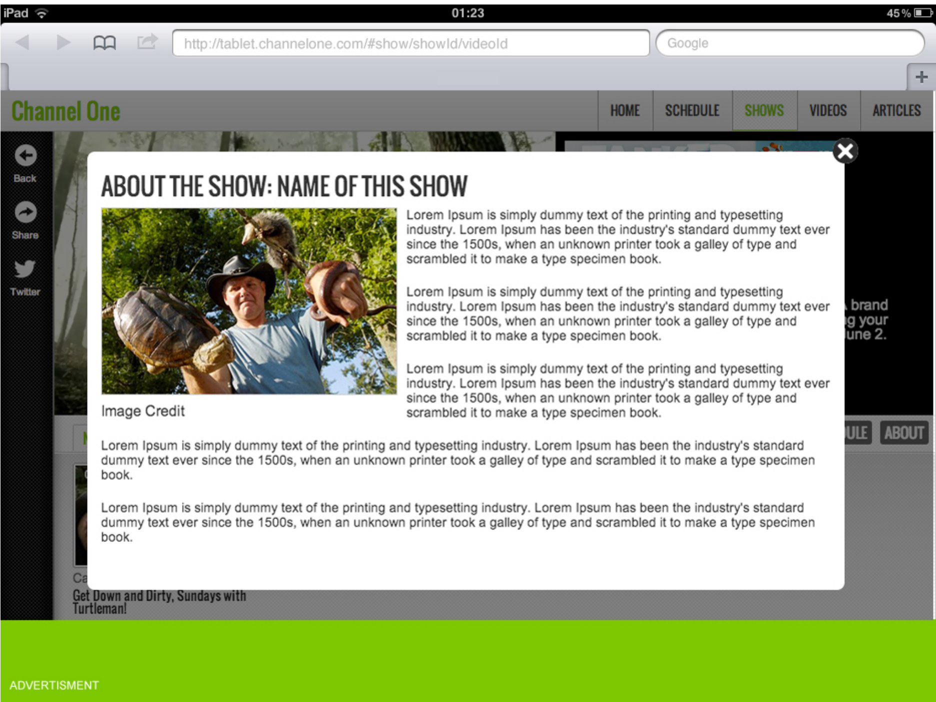

Figure 10. Channel one, modal view component used to show the *About Show* information modal pop-up

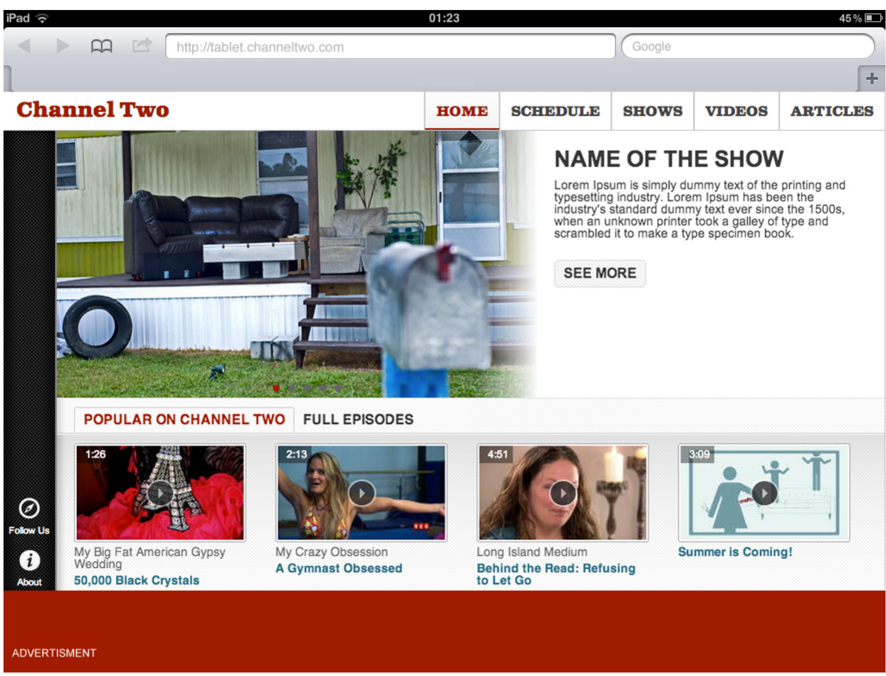

Figure 11. Channel two, created based on the core grails plugin

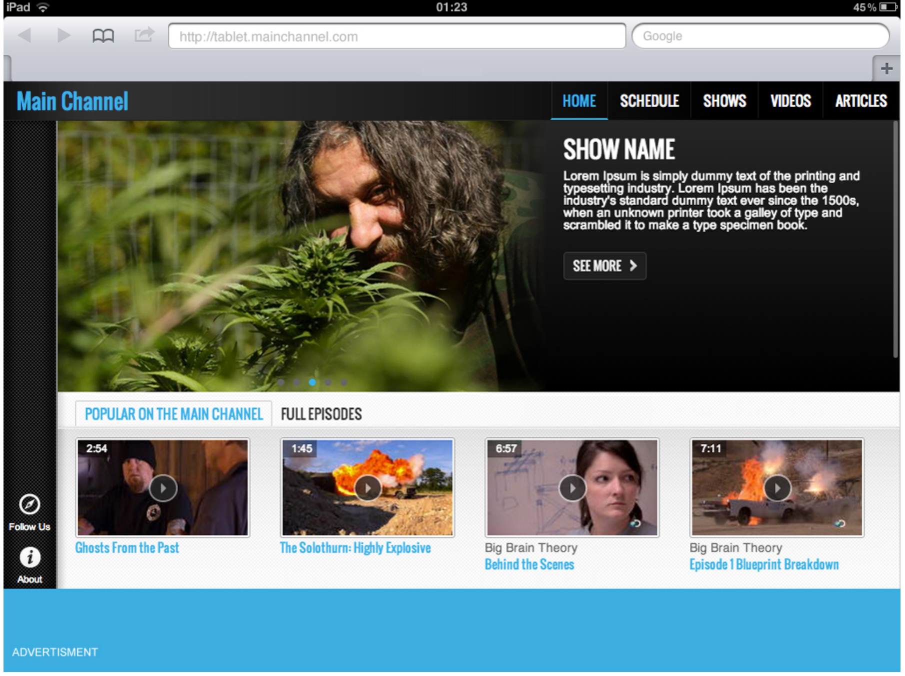

Figure 12. Main channel, created based on the core plugin