Niko Gröhn

Web Platform for Digital Magazines

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Thesis

11 May, 2015

Author(s)	Niko Gröhn
Title	Web Platform for Digital Magazines
Number of Pages	41 pages
Date	11 May, 2015
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Specialisation option	Software Engineering
Instructor(s)	Erja Nikunen, Principal Lecturer Tatu Dufva, Director of Product

The purpose of this thesis was to document and explain the architectural choices behind an application for reading digital magazines. The application was developed for the company Conmio Oy and is intended to be used as a demo application for sales purposes. It was developed mostly as a prototype for seeing the viability of such application as a web application.

The thesis begins with explaining the initial concept and requirements of the application, while comparing it to some existing applications. The next parts present the technical choices that helped further shape the initial concept. The main flow of control of the application is also shown alongside the development workflows. Latter parts of the text explain how different parts of the application were modularized and how this makes it function more as a platform.

Though this thesis focuses on the web as a platform, parts of it such as the general architecture, could be easily translated to other platforms. This makes this thesis also work as a reference for future work on additional platforms.

The resulting application was a functional prototype that will be used for sales purposes. The now proven prototype also paved way to developing the application on other platforms in the future.

Keywords	Digital magazine, Backbone.js, RequireJS, web-application, single-page application

Tekijä(t) Otsikko Sivumäärä Aika	Niko Gröhn Web-pohjainen alusta digitaalisille lehdille 41 sivua 11.05.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Erja Nikunen, Yliopettaja Tatu Dufva, Director of Product

Työn tavoitteena oli dokumentoida ja selittää arkkitehtuuriset valinnat digitaalisten lehtien lukuun tarkoitetun sovelluksen taustalla. Sovellus kehitettiin yritykselle Conmio Oy. Se kehitettiin pääasiassa prototyyppinä selvitykseksi tämän kaltaisen sovelluksen soveltumisesta web-sovellukseksi.

Teksti alkaa tutustumalla sovelluksen konseptiin sekä vaatimuksiin. Samalla sovellusta verrataan jo olemassaoleviin vastaavanlaisiin sovelluksiin. Seuraavat osat esittävät tekniset valinnat jotka tehtiin alkuperäisten vaatimusten pohjalta, sekä miten näitä teknisiä valintoja käytetään sovelluksessa. Viimeiset osat selittävät miten sovellus modularisoitiin ja kuinka se saatiin toimimaan alustana erilaisille digitaalisille lehdille.

Vaikka työn pääpainona on web-sovelluksen kehitys, on osia tekstistä hyödynnettävissä myös muilla alustoilla. Teksti toimii teknisen katsauksen ohella dokumentaationa mikäli sovellusta jatkokehitetään eri alustoille.

Lopputuloksena työssä kehitettiin sovellus joka osoittautui toimivaksi prototyypiksi. Prototyyppi pääsi myynnin tueksi sekä osoitti mahdolliseksi sovelluksen toteutuksen muille alustoille tulevaisuudessa.

Avainsanat	Digital magazine, Backbone.js, RequireJS, web-application, single-page application

Contents

List of Abbreviations

API	Application Programming Interface
CDN	Content Delivery Network
DOM	Document Object Model is the convention how browser interacts with HTML
HTML	Hypertext Markup Language, markup language to create web pages
JS	Javascript, programming language used in web pages
JSON	Javascript Object Notation, lightweight data-interchange format
MVC	Model-View-Controller, a design pattern
UI	User interface
URL	Uniform Resource Locator
REST	Representational State Transfer, architecture for web services

1 Introduction

This thesis was made as a part of a project to create a new application for reading digital magazines. The aim was to create a highly modular and flexible application for transforming existing news and stories into format applicable to digital magazines.

The application was developed as a research and development project for the company Conmio Oy. Conmio creates largely custom made web and mobile applications for brands and publishers globally. Conmio needed a new product for sales purposes and this application was developed as a proof of concept application for the sales.

This thesis is mostly a documentation for the application architecture and the ideas behind it, without diving too deep into the code. The text focuses on the development of the application for the web platform, as it was chosen as the initial platform for the proof of concept application. The thesis first looks into the initial requirements and the concept of the application, such as different parts it consists of. The latter parts of the text look into technical choices that helped further shape the ideas behind the application.

2 Digital Publishing

Since the introduction of the internet, publishing industries have been in constant change. First came web sites to complement the printed media. A second large change took place with the rise of mobile devices such as iPad, which allowed the opportunity to bring print media into a completely new format.

Digital magazines of today are rarely replicas of their printed counterparts. Instead they are specifically made for the digital medium, making use of the interactivity possibilities delivered by the new medium. Digital magazines don't necessarily compete with print media, but other digital media. They are optimized for the digital screens and offer digital extras such as animations and videos. [1]

2.1 Related Work

Nowadays there are quite many applications that transform content to a more magazine-like content for digital devices. This section takes a look at two of them that have acted as inspiration for this application: Flipboard and Yahoo News Digest. These two applications show two completely different concepts for digital content yet they both are well related to the concept of this application.

2.1.1 Flipboard

Flipboard is a magazine reading application initially built specifically for iPad. The first version of the application was launched on July 2010, a few months after the release of the first iPad. Flipboard's concept was to combine the beauty of print magazines with the web. It takes existing content and news from various media outlets and transforms them into a format suitable for a magazine.

Flipboard was a big hit for the original iPad as it came along in the first third-party iPad applications. Reading news content in a magazine-like experience by swiping sideways was a big hit and the application was named Application of the Year 2010 by Apple. [2]

During the development of the application, Flipboard was only available as native applications for various devices. This changed in February 2015 when Flipboard introduced its web version. The web version offers mostly the same experience as the native applications but optimized for desktop computer usage. In essence, this means changing touch based swipe events to regular scrolling.

Conceptually, Flipboard is quite similar the approach taken in this application. It is likely to be one of the largest inspirations for this application.

2.1.2 Yahoo News Digest

Yahoo News Digest is one of the freshest new ideas for news reading applications on the market. Its aim is to offer the user 10 generated news stories twice a day, in the morning and the afternoon. It attempts to mimic the way people are used to read morning and afternoon newspapers. The articles are curated by selected editors. Therefore the application offers no customization. Every user will receive the same news when the news arrive. The application doesn't offer the user an endless stream of news but a limited set of news and a definitive ending for the news stream. [3]

Yahoo News Digest has a rather different concept than Flipboard but nevertheless it acts as a great inspiration for this application. The intention of Yahoo News Digest is to offer a set amount of relevant news, instead of digital magazines with endless streams of more or less interesting articles. As inspirator, Yahoo News Digest is a catalyst for the flexibility of the application being developed for this thesis. The objective is that this application could be used to create experiences similar to Flipboard and Yahoo News Digest.

3 Project Definition

3.1 Concept

Conmio had an existing solution for building and using digital magazines. This previous solution was highly customizable, but the customization was on a lower level leading to more work when shaping it to a custom magazine. Therefore the need arose to build a more lightweight, yet flexible solution to present these digital magazines. A single application instance should be able to support multiple customized magazines instead of the previous solution where each type of magazine would need its own application instance.

The new application can be split up to three parts. The first is the magazine viewer application, which is the focus on this thesis. It's the part of the application that is publicly visible and is used to read the magazines. The other two parts are a bit separate from the magazine application. First of these is the server side application that is used to parse and format the existing content to a suitable format for the magazine. The last part acts as a glue between the other two parts. This glue is the admin interface, acting between the magazine application and the backend. The admin interface is used to construct the magazine from the pre-formatted content, resulting in a magazine usable by the magazine application.

3.2 Software Design

Once the concept of the application was clear, it was time to make choices regarding the overall design and structure of the application. These choices revolved around whether the application should be made as a native application or as a web application.

3.2.1 Web and Native Applications

Native applications work like traditional desktop PC applications. They are often developed for a single platform such as iOS or Android and they will only run on that single platform. Native applications can easily be installed from the platform's own store. They're also highly integrated to the device and system. They gain access to various system components and sensors such as camera and GPS.

Web applications on the other hand, live in the web. They are accessible by virtually every device that can access the internet. They can't be installed on devices but they can often be bookmarked or sometimes even an application like shortcut can be created. Usually users require the internet to access the application, but there are ways to make them work offline.

The greatest loss for web applications over native applications, is the lack of performance. Native applications can leverage device performance better due to the closer integration with the system. This results in more responsive touch interfaces and generally faster performance. Web application performance is constantly increasing with better performing devices and better browsers.

Especially Google with its Android platform has been constantly thinning the line between web and native applications. Many features that have formerly been available only for native applications, have been incorporated into web applications. Sensors such as GPS and gyroscopes are already available on the web. The most recent upcoming feature is push notifications, which allows web pages to create notifications to the device in the background. For apps that don't necessarily require top notch performance, push notifications have been one of the biggest drivers for creating a native version of the application. [4, 5]

For the prototype application being developed, the web ended up as the winner platform. As this acts as both the initial prototype and demo application, a short development time is essential. The plan is to possibly develop native applications in the future, depending on the success of the prototype. The main reason for picking the web as the platform of choice is the ease of development. Conmio also has some existing solutions and libraries for various features of the application which can further reduce the development time.

3.2.2 Single Page Applications

Traditionally, web pages have relied heavily on the server. Every change in content required a full page refresh, where HTML was rendered on the server and returned to the browser. This results in to the client loading duplicate content on subsequent page loads and loss of responsiveness during the page load.

Single page applications were pioneered in the mid-2000s by applications such as Gmail and Google Maps. After the initial load, single page applications can load the subsequent pages without requiring a full page load. By leveraging Javascript functionality, they can dynamically load the new content, update the URL of the browser and render the new content. Single page applications often make the initial page load a bit longer when compared to traditionally built pages. This is because they often include some sort of Javascript framework or library that is used to handle the following page requests. Unless the application is built as an hybrid application, the first page load will also have to load the initial page content, once it has loaded the application framework. Hybrid applications avoid this problem by supporting both client side and server side rendering of the content or by including the data on the initial page load.

Single page applications have been mostly used for web pages that feel like applications instead of traditional web pages like blogs. Often these applications are meant to be kept open for a long time, so the initial longer page load time doesn't matter. [6]

For the application being developed, single page application is the obvious way to go. The general use case is to keep the page open for a long time while the user scrolls to the articles of the magazine. The frontend should also be decoupled from any backend code, as the backend is meant to only provide JSON data that can be used in both the web application and future native applications. For the this version, it was also decided to not go with the hybrid approach as the benefits of a hybrid application were unimportant for the prototype.

3.2.3 Application Structure

Single page application design was chosen because it fits the customization oriented design of the application itself. The general nature of the application is also quite application like, unlike traditional web pages that require a server to render the content.

With initial load, the client loads the base HTML for the application, along with the core Javascript library and CSS styles. From hereon the core libraries take over and handle loading of the additional content to render a magazine in the client. The core files can take benefit from heavy caching, making subsequent magazine loads super fast. Cached files can be stored almost indefinitely on the client's browser, allowing future page loads to be near instant. When the core of the application is already available, the content is the only thing that needs to be loaded.

The files of the application are all static files. This means that they are generated once during the build phase of the application instead of being generated dynamically. After deployment, no changes are made to these files, so the clients can download and store these files offline until a new version of the application is deployed. This makes hosting of the application cheap and effective as the application files could be deployed to a content delivery network. Content delivery networks (CDN) are worldwide distributed networks of servers mainly serving static files. CDNs are covered in chapter 5 of this thesis.

All of the content that actually changes is loaded dynamically by the core of the application. This is where the server comes into play. The server serves JSON responses to the client that contain the contents of the magazine. Serving raw data from the server keeps the server lightweight and reduces load times of the content from the server, as almost no preprocessing of the content is required in the server. It also allows to use the same server for other platforms as the content format isn't restricted to any platform.

3.3 Interface

The interface is designed with content-first approach and is highly dependable on the content of the magazine. With that in mind, this section shows the general interface design choices and some interface elements that are common to all magazines.

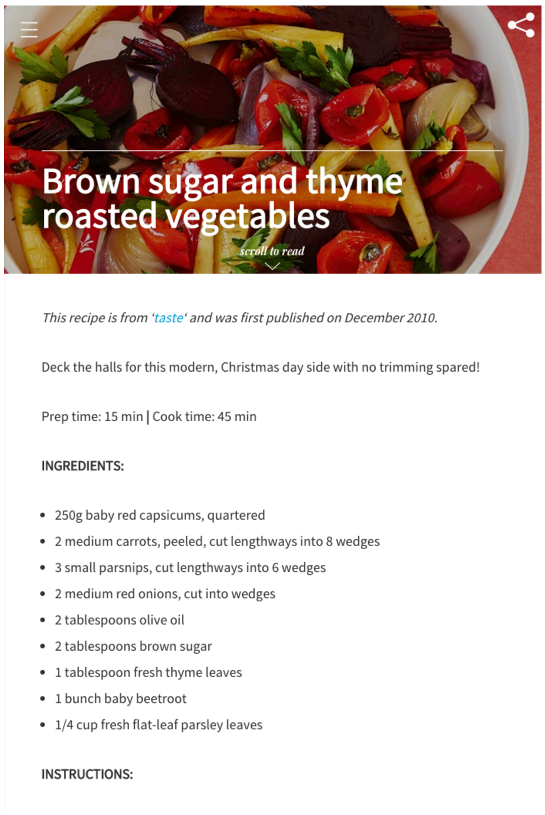
3.3.1 Mobile First Design

The main target devices of the application are mobile devices. As these devices have a highly limited amount of screen real estate, the interface was designed with mobile first methods. In mobile first design, the UI design process begins from the device with the smallest screen real estate. By doing this, the designer can keep focused on the key tasks and elements available, as there isn't room for extra features yet. The designer ends up with a minimal version of the interface to make it usable. When moving to larger screen sizes, new features can be added where they seem necessary, eventually reaching the full desktop size.

The opposite of mobile first design is designing from top to bottom, i.e. building the largest desktop version first and scaling that down to smaller devices. This often causes unnecessary clutter for the smaller screens, due to larger screens typically allowing all kinds of extra elements in the UI. Mobile devices also lack the performance of desktop devices, causing worse performance and slower page load times. [7, 8]

3.3.2 Mobile Interface

To keep the application as simple as possible, it has only one view available. This main view of the application is the page view. This view displays a single page of the magazine, using all available space of the browser window. Example article can be seen in figure 1. The contents of the page are fully customizable by the magazine designer, allowing the page to use the available space as best suited for its contents. The page can be scrolled up and down and for all intents and purposes it acts as a regular browser page.



This recipe is from 'taste' and was first published on December 2010.

Deck the halls for this modern, Christmas day side with no trimming spared!

Prep time: 15 min | Cook time: 45 min

INGREDIENTS:

- 250g baby red capsicums, quartered
- 2 medium carrots, peeled, cut lengthways into 8 wedges
- 3 small parsnips, cut lengthways into 6 wedges
- 2 medium red onions, cut into wedges
- 2 tablespoons olive oil
- 2 tablespoons brown sugar
- 1 tablespoon fresh thyme leaves
- 1 bunch baby beetroot
- 1/4 cup fresh flat-leaf parsley leaves

INSTRUCTIONS:

Figure 1: Example article page

Because the interface doesn't have any buttons for direct navigation, it requires a not so common method of navigating between pages. The navigation makes use of touch events where available. In addition to navigating the current page by scrolling up and down, the user can navigate the magazine by swiping to left and right. Swiping to left reveals the next page. Swiping far enough and releasing causes the magazine to switch to the next page. This kind of navigation is common to users coming from native applications, but it's a rather uncommon navigation method for web pages. Some guidance is required to help the user to understand how the navigation works. Currently this is a tooltip on the first page to direct the user into swiping left.

The second common element on the page is an overlaid button that is used to toggle the navigation. This button floats in one of the four corners of the browser window. It's up to the magazine designer to decide how the button is presented. The button tries to stay out of the user's way by hiding itself when the user scrolls down the page. With this it can avoid blocking textual content while the user is reading it. The button reveals itself again when the user scrolls up or navigates to the next page.

Navigation menu is the third element common to all magazines. It's usually hidden from the user's sight, unless the screen is so large that we have enough empty space to make use of. The navigation menu has two different kind of layouts, depending on the device size.

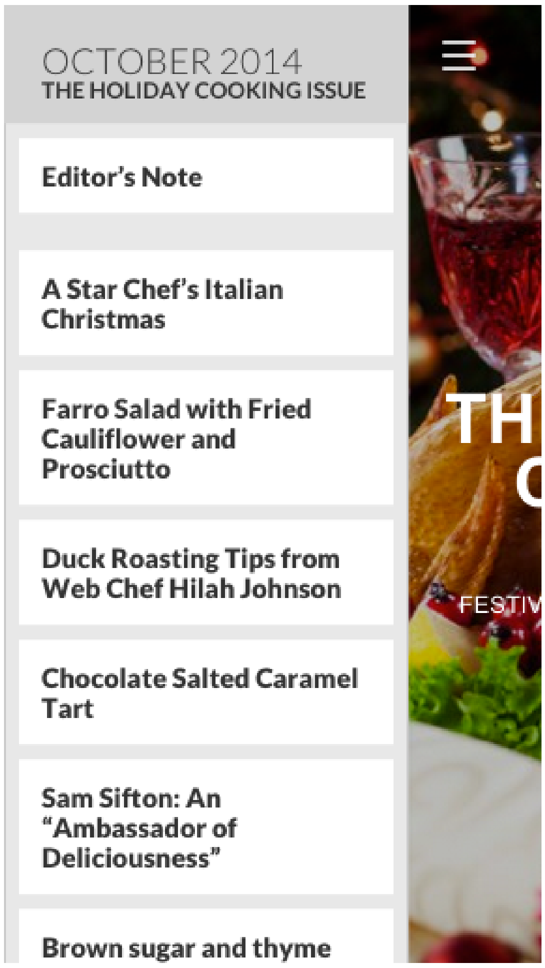

OCTOBER 2014
THE HOLIDAY COOKING ISSUE
Editor's Note
A Star Chef's Italian Christmas
Farro Salad with Fried Cauliflower and Prosciutto
Duck Roasting Tips from Web Chef Hilah Johnson
Chocolate Salted Caramel Tart
Sam Sifton: An "Ambassador of Deliciousness"
Brown sugar and thyme
TH
C
FESTIV

Figure 2: Mobile navigation menu.

For the smallest devices the menu is a simple off canvas menu as seen in figure 2. Off canvas menus are hidden on either the left or right side of the page and they slide into view, either overlaying the content or pushing the content partly off the screen. The content of the menu is a simple table of contents for the magazine, showing the title of the article. Pressing any of the articles causes the first page of that article to jump into the view. The menu is toggled off by pressing the partly visible article on the right.

The second navigation layout on figure 3 is designed for tablets. The situation on tablets isn't much different from mobile devices. The space is still limited and the content needs to be the focus of the application. The menu is similarly an off canvas menu here, pushing the content offscreen when the menu is opened. The only difference between the menus of tablet and mobile is that the tablet menu has preview images available. A slightly bigger screen allows us to improve the visuals of the application as the additional images don't make the menu too cramped.

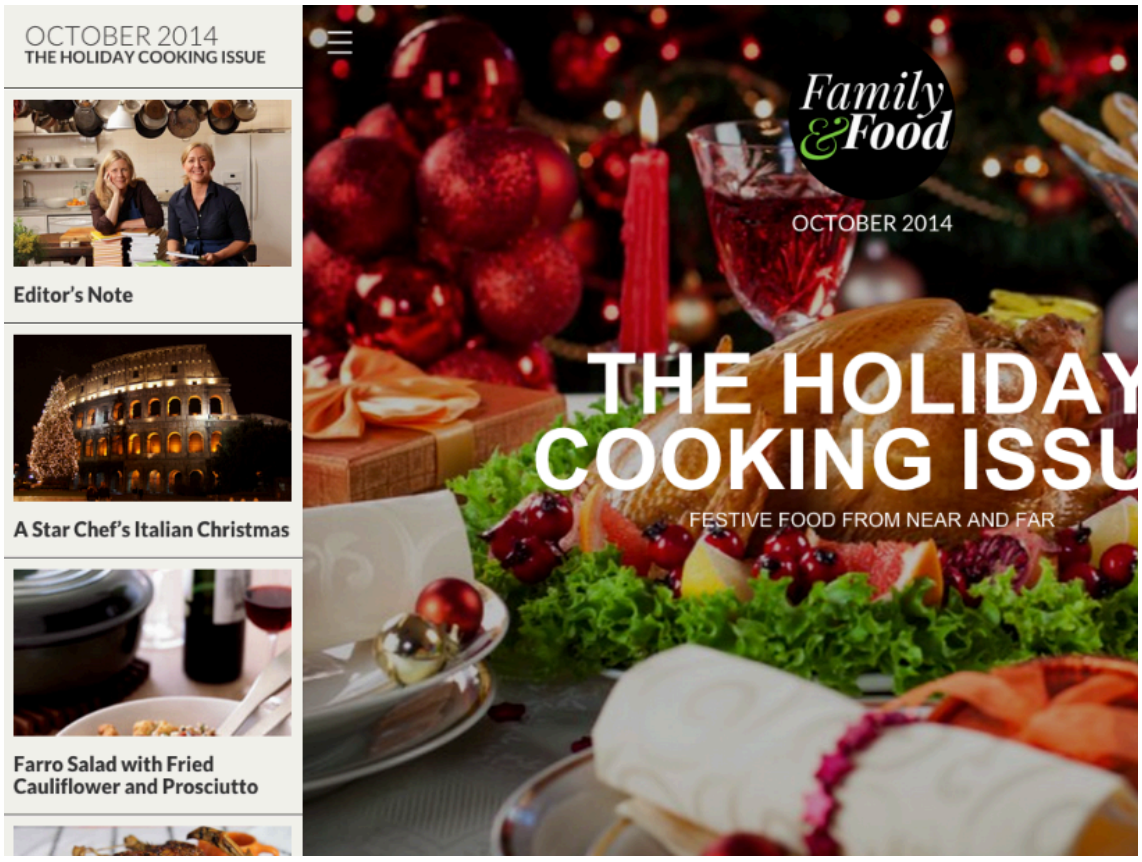

OCTOBER 2014
THE HOLIDAY COOKING ISSUE
Editor's Note
A Star Chef's Italian Christmas
Farro Salad with Fried Cauliflower and Prosciutto
Family &Food
OCTOBER 2014
THE HOLIDAY COOKING ISSU
FESTIVE FOOD FROM NEAR AND FAR

Figure 3: Tablet navigation menu.

3.3.3 Desktop Interface

The main usage of the application will be on mobile devices, but the application was designed to work with desktop browsers too. Desktop web browsers don't have touch events. Instead they rely on mouse and keyboard usage. The swiping gesture could be done with a mouse as dragging from side to side, but that's rather uncommon usage for mouse. A more common navigation method for the mouse would be buttons on each side of the screen. As desktops also have larger screens, the sides have empty spaces so the navigational buttons won't overlay the textual content. Accessibility by keyboard usage was also kept in mind. The pages can also be switched with the left and right arrow keys on a keyboard.

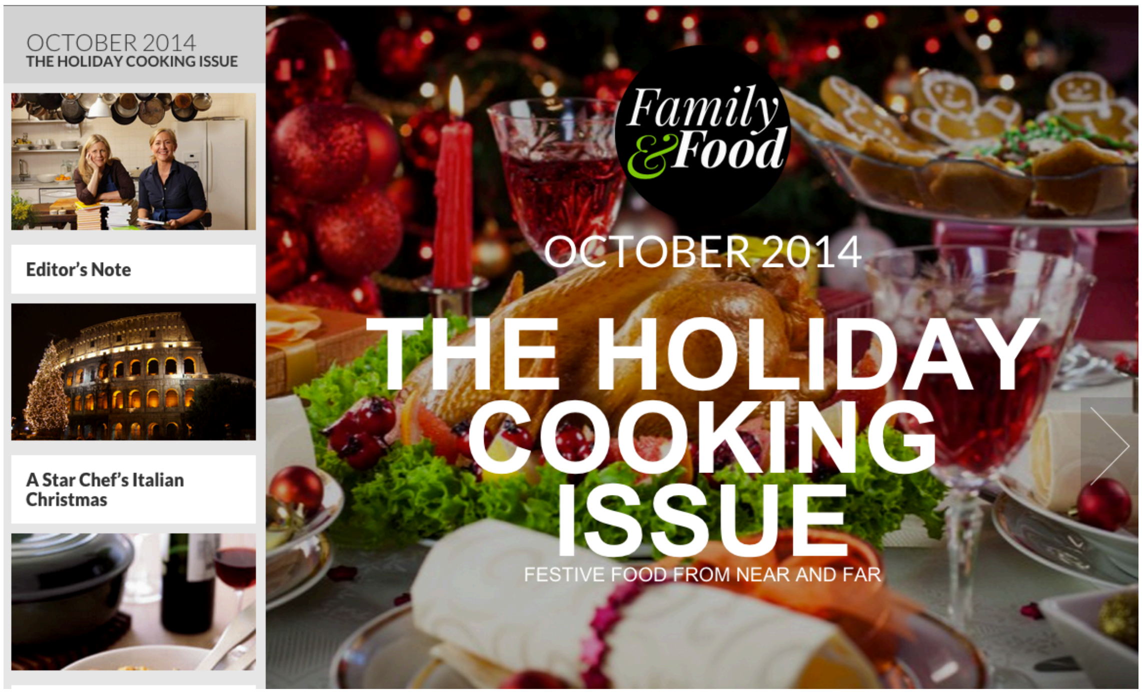

OCTOBER 2014
THE HOLIDAY COOKING ISSUE
Editor's Note
A Star Chef's Italian Christmas
Family &Food
OCTOBER 2014
THE HOLIDAY COOKING ISSUE
FESTIVE FOOD FROM NEAR AND FAR

Figure 4: Desktop navigation menu.

The biggest difference between mobile and desktop layouts is the navigation menu. Desktop browser windows are much wider than those of their mobile counterparts, so the interface can make use of all that empty space as the page contents won't require all that space. To make use of that extra space, the navigational menu is always visible on the left side of the screen as seen on figure 4. This allows the user to always see their position in the magazine. The visuals of the desktop menu are no different from the tablet version. Naturally, the desktop doesn't have the overlaid button to toggle the navigational menu.

3.4 Application Resources

The application can't function solely on its own. To display the content it needs other software to generate the content for the magazines and articles. The application doesn't care how these are generated. The only thing that matters is the format of the content. The content, which we will refer as feed from hereon, is formatted as JSON. JSON (JavaScript Object Notation) is a lightweight data-interchange format. It has strong roots within Javascript standards and is generally easy to use and read. [9]

The feed is loaded either from a REST API or a static JSON file available from an URL.

3.4.1 Content Feed

Listing 1 depicts an example article from the feed. The *id* and *content* properties are mandatory fields of the feed. The *id* is used mainly for constructing URLs that link to a certain page within the magazine. The *content* property generally has data that comes from the client. It's also what the templates and PageViews use for rendering.

```
{
    "id": 1,
    "__template": "demo/article",
    "content": {
        "body": [
            {
                "text": "Lorem ipsum dolor sit amet,
                   consectetuer adipiscing elit.",
                "type": "lead"
            },
            {
                "text": "Sample image",
                "type": "image",
                "url": "http://example.com/image.jpgg"
            },
            {
                "text": "Sed posuere interdum sem. Quisque
                   ligula eros ullamcorper quis, lacinia
                   quis facilisis sed sapien. ",
                "type": "paragraph"
            },
            {
                "text": "Mauris varius diam vitae arcu. Sed
                   arcu lectus auctor vitae, consectetuer et
                    venenatis eget velit.",
                "type": "paragraph"
            }
        ],
        "description": "Lorem ipsum dolor sit amet,
           consectetuer adipiscing elit.",
        "image": {
            "src": "http://example.com/image.jpg"
        },
        "meta": {
            "author": "Lorem Ipsum",
            "origin": "Generated sample"
        },
        "title": "Sample article"
    }
}
```

Listing 1: Sample article from feed

The *__template* property is what defines the page module to use for this page's content. The value directly translates to the module path, i.e. *demo/article* would resolve to a path of `http://example.com/pages/demo/article/` which would contain the files related to the page module. This field is optional and default module will be used if it's missing from the feed.

The data inside the *content* property doesn't have any mandatory rules. It's completely up to the creator of the page template to decide what properties the feed should contain. Though for common text based articles, it's recommended that a common schema is used. All templates that present the same type of content should follow some common schema. If the developer follows these recommendations, it's easy to change the template without requiring any changes to the feed.

3.4.2 Content Feed Creation

The magazine application is agnostic of how the feeds are created, but the concept of the application includes a general idea for how the feeds should be implemented. The feeds are created using an entirely separate application which this section will briefly cover.

The example content creator features two kinds articles. A custom article and another that's directly linked to the client's own feed, a regular article. From the end user's point of view, both article types have only one single common field that is configurable – the template. Other mandatory fields exist for both types, but they are configurable only in custom articles. In regular articles these are fileld automatically.

With custom articles, the user can create any kind of feed for the template to use. Usually custom articles are used for special pages such as cover and alike. Although the user can enter anything into the feed, the feed should be usable by the page. To ease this, templates can provide a JSON schema that is used to determine what content the user can put into the feed for this template. Templates used by custom articles typically require quite specific content, making it difficult to switch the template of a custom article once the feed for it is created.

The regular articles are more restricted than custom articles. They are always linked to external third party content feeds. i.e. if the third party feed changes, the article changes in the magazine. The contents of the magazine mostly consist of articles of this type. These articles aren't customizable by the user of the content creator. Before the user can access these articles in the interface, a feed parser has to be created. The feed parsers read the client feeds and parse the data into a format suitable for the templates. If the resulting format follows the guidelines for articles, the end user can now add this article into magazine using any template that also follows the feed format guidelines. The parser is solely responsible on the format of the JSON feed for each article generated by it.

Among the articles, the user can also set magazine-wide configurations. The mandatory configurations are magazine title and a slug that is used in URL to access the magazine. With these, custom plugins are also enabled in the magazine configurations. Each plugin has their own configurations. Plugins support similar JSON schemas as custom articles so forms for configuring the plugins can be generated for each plugin.

Once the required configurations are created, the magazine can be published making it available by knowing either the magazine id or the slug. The slug is a more readable version for the magazine id for use in readable URLs.

4 Architecture

This section dives into the structure of the application and the tools used to build it. A single-page application approach was decided for the application so the technical choices had to be based on that decision.

4.1 Frameworks and Libraries

The amount of Javascript libraries and frameworks today is overwhelming. New libraries appear almost every day, saturating the market. For this project it was decided that a modern and proven to work library with sufficient existing userbase should be used instead of the most cutting-edge library.

The main contestants for the application framework were AngularJS and Backbone.js due to their wide usage and production stability. Conmio had some experience with using both of these frameworks. AngularJS and Backbone.js are quite different from each other, both great at different use cases. The simplicity of Backbone seemed to suit this application better and the features of AngularJS didn't bring anything useful to the table. Conmio also has some existing libraries that would be beneficial for this application and those libraries should work with the chosen framework. These libraries proved too hard to implement with AngularJS, which eventually caused the Backbone.js to be chosen as the framework of choice.

4.1.1 Javascript Framework

Backbone.js is a lightweight Javascript framework for building MVC applications. MVC is an architectural design pattern that encourages separation of concerns. Traditional applications that follow MVC separate the application into three different main parts: models, views and controllers. Models represent the data of the application. They are often either a single object or a collection of objects. Views control the visual representation of of the models. A model gets attached to a view which displays the model's data to the user. The view might also be able to change the model data. Controller then works as a link between the user and the application. It receives input from the user and directs the messages to appropriate views.

When MVC gets applied to web servers, the user input is limited to the single request that the server gets. On servers the controller acts between the models and views, attaching the model to the view once it has the data and then responding to the request with the rendered view.

Backbone follows the MVC pattern as can be seen on figure 5, but there are some differences that are common to most single page applications that follow MVC. Backbone has both models and collections of models to hold the data. It also has views that handle the presentation of the data inside the models. But Backbone does not include controllers as a separate entity. Its controller functionality is included in the views. Controller's task is to take user input, create models and attach models to a view. Backbone utilizes events for user input. As these events are usually attached to the DOM and the views represent DOM elements, it makes sense to combine the functionality of views and controllers.

Backbone's main selling point is giving structure to the application. It offers a set of data-structuring tools like models and collections and some UI features like views and routing. As Backbone's name implies, it acts only as a backbone of the application. For example, the views don't include any DOM manipulation tools. Backbone gives freedom and flexibility to use the given tools as the user best sees fit. It has dependencies on two of the most widely used libraries in the web: jQuery and underscore.js. jQuery provides

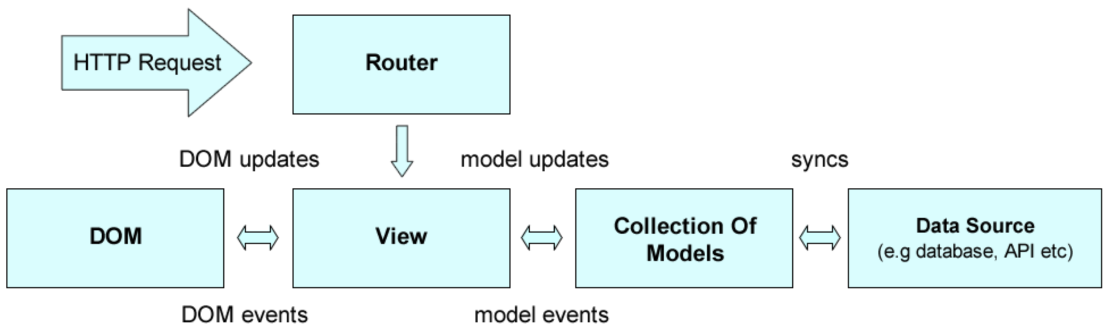


Figure 5: Backbone.js lifecycle

extremely feature-rich set of tools for the most common actions done in web applications such as navigation within the DOM and Ajax requests. Underscore provides functional programming helpers mostly for data manipulation with commonly used methods such as map and filter for array manipulation.

Backbone heavily utilizes events for both user input and communication between views and models. Backbone has two types of events for these tasks: regular DOM events and the internal Event API. The DOM events are created through jQuery and follow the Javascript standars, allowing easy communication between Backbone and other libraries. The DOM events are also used when the user causes click events or other input events.

Backbone's flexiblity and the ability to give structure to the application without enforcing any strict rules on how to use it were the main choices why it was chosen for this application. As Backbone is fairly unopinionated on how it's used, it's rather simple to use it along other libraries that have no relation with Backbone. One such library was Conmio's internally developed library for creating swipeable carousel views. The library does DOM manipulation on it's own and needs just DOM elements to render. As Backbone's views are just DOM elements, making them work together proved to be fairly trivial.

Backbone doesn't even enforce the usage of the dependencies it has, so the developer is free to use other libraries for their own code. For example, the templating method of underscore.js could be replaced with full fledged templating engine such as Moustache or Handlebars. [10, 11]

4.1.2 Templating Engine

Views are an important element of Javascript applications and they need a way to present themselves to the user. In a regular web site with minimal Javascript enhancements it's often enough to keep the HTML markup changes in code. For any larger application that deals with a lot of views, such as single page application, it's recommended to separate the HTML markup from the code.

Templating engine libraries allow you to define your templates in HTML markup with some additional syntax included by the library. That additional syntax consists of methods to include variables from code to the template. The engine then takes a set of data, often represented as JSON, alongside the template and compiles them together. The end result should be usable HTML markup with the variables from template replaced with values from the supplied data.

Backbone.js includes underscore.js, a Javascript library which one feature is micro templating. It's simple and well functioning and its functionality doesn't differ that much from heavier libraries. On the other hand, its feature set includes all functionality of Javascript. This might sound like a good thing, easing inclusion of any kind of logic in the template markup and it's likely to work well with small applications. On larger applications, it's often recommended to use logicless templates. Logicless templates enforce separation of logic and the template markup, keeping the templates clean. Logicless templates are very strict on the features accessible by developer in the template markup. They do include simple helpers such as if and each clauses and support custom helper functions that can be defined in code elsewhere.

Handlebars is one logicless Javascript templating engine. It's an extension on another logicless engine Moustache. It's one of the most widely used templating engines alongside Backbone.js and they are often used together.

```
{{#page}}
    <h1>{{ article.title }}</h1>
    <article>
```

```
        {{#each article.content }}
            {{ paragraph this }}
        {{/each}}
    </article>
{{/page}}
```

Listing 2: Sample article template using Handlebars

Handlebars syntax uses braces for marking variables and helpers in the template markup. It includes a variety of often used helpers such as *if* and *each*. These are called block helpers. Block helpers are expressed by a block that has a beginning and end as we can see on the *page* and *each* helpers in listing 2. The context of the variables in block is changed to the variables given to the block helper. Helpers don't necessarily need to be in the form of blocks. In listing 2 we find the *paragraph* helper in the *each* block. This helper takes the current value of the above iterator and renders a paragraph from it. Logic can be introduced into the templates with custom helpers. Custom helpers are defined in code outside of the template, by using the Handlebars API method *registerHelper*.

In listing 2 we have two custom helpers. The block helper *page* and regular helper *paragraph*. The *page* helper is used to mark the contents of the block as a single page in the magazine. If multiples of this block is in template, the article would span multiple pages. This could be wrapped in a *each* block to dynamically determine the amount of pages. The *paragraph* helper takes a single value from the content array, determines the type of it and outputs HTML depending on the content type. These helpers present functionality that is used by all articles. The functionality is fully abstracted from the templates, making the creation of templates only a matter of building the markup.

4.1.3 CSS Preprocessors

CSS preprocesssors are a powerful way to extend the capabilities of CSS. Sass and Less are two examples of such tools. Conmio has so far used Less as the CSS preprocesssor of choice. For this project Sass was chosen to test its capabilities.

Sass is both a scripting language and a CSS preprocessor. It's a language that extends

CSS and provides new methods such as nested rules, variables and mixins. It provides two different syntaxes for the scripting, older and newer one. The older one is called SASS and differs by quite a lot from regular CSS, using indentation and whitespace instead of braces. The newer syntax on the other hand, is fully compatible with regular CSS, meaning that all valid CSS is valid syntax for Sass. This more recent syntax is called SCSS for Sassy CSS. Neither SASS nor SCSS are valid CSS thus they require compilation to valid CSS.[12]

4.2 Module Loading

A good practice in any larger application is to modularize the code. In modular code, the application's functionality is split up in multiple decoupled modules, sub-programs someone might say. Every module should handle their own dependencies. This allows easier maintainability for the application and allows developer to easily see how changes to one part of the application affect another

Unfortunately, neither Javascript nor Backbone bring means to handle the loading of such modules. Javascript has design patterns such as module pattern and object literal pattern to help code modularization. Due to the nature of Javascript, each of module exists in the global namespace, leading to possible naming collisions. Javascript wasn't initially designed to work in a modular way like this. These patterns also don't supply methods to handle dependencies between modules, leaving every module reliant on the global namespace and making dependency management difficult. This is a fine method to modularize smaller scale web applications with limited amounts of Javascript.

For larger scale Javascript applications like single page applications, this way of modularization is not adequate for application maintainability. For these kinds of applications tools and libraries have been created to help the cause. Most popular ones are AMD (asynchronous module definitions) and Common.js. Module loaders use various methods like injections of HTML script tags to make browser initiate file load. Some module loaders don't support dynamic loading at all and only offer tools to bundle the files during deployment. [13]

4.2.1 Specification

AMD (asynchronous module definitions) is a format to help handle the loading of modular Javascript code asynchronously in the browser. AMD is only a specification for this method of module loading and doesn't include an implementation of it. AMD comes from wanting to ease the dependency handling of Javascript applications. As the name implies, it's also able to handle the dependencies asynchronously during runtime, leading to the application being able to determine dependencies during runtime. [14]

The AMD api specifies a single method called *define*. This method is used to define a module that can be loaded. As all modules are encapsulated within this method, single file can have multiple modules. The method takes up to three arguments: module id and dependencies which are both optional, and the factory which is a function that should be executed to instantiante the module. The factory function should export the module value. The dependencies of a module are typically just file paths that point to the file that holds the module.

Another method in the AMD specification is *require*. It is used to load the defined modules. This method takes just two arguments: the dependencies to load and a function to call once the dependencies are loaded. Typically this method is only used at top-level to bootstrap the application but it can also be used inside modules to dynamically load dependencies by getting the module name from a variable.[15]

```
define(
  'myModule',
  [ 'modules/foo', 'modules/bar' ],
  function( foo, bar ) {
    var myModule = {
      helloWorld: function() {
        return 'Hello world';
      }
    };

    return myModule;
  }
);

require(
  ['myModule'],
  function( myModule ) {
    console.log(myModule.helloWorld());
  }
);
```

Listing 3: AMD module example

The listing 3 shows an example program that has one one module created with the *define* call. This module has two external modules that are located in files *module/foo.js* and *module/bar.js* relative to this file. The module exports a simple object that has one method available. The listing also shows the *require* call which loads the module that is defined above and calls the method exported by the module.

4.2.2 Library

Require.js is a Javascript library that implements the AMD API specifications. It's optimized for browser usage but has versions that function in other environments such as Node. Require.js injects the dependencies as script tags into the DOM, leaving browser to handle the file loading. With this, it also supports Javascript files that do not conform to the AMD format. Therefore it is trivial to use Javascript files that weren't created to be used with module loaders. This is one essential feature of Require.js for this project, as the application requires usage of libraries that weren't created with module loading tools in mind. [16]

Although Require.js is able to handle the dependencies asynchronously as per the AMD specification, it also includes an optimization tool for prepackaging the code. This tool creates the dependency tree for the application outside the browser and then collects the dependencies, packaging them into a single file. Loading multiple separate files in browser is fairly expensive task and can lead to increased load times. The optimizer eliminates these issues by producing a single file that contains all of the dependencies. Even if the optimizer is used, the application can have dependencies fetched asynchronously during runtime. By using variables in the dependency path, the optimizer is unable to optimize it, leaving the dependency to be loaded during runtime. [17]

4.3 Structure

The core of the application is split up to three different Backbone views: magazine, article and pages. Along these there are the magazine and article models and a collection for article models. All of these are created by using the Backbone.js components, wrapping each file as AMD to be used with Require.js. Figure 6 presents the general structure and the relationships between the different classes of the application. This section goes into more depth on some of these classes.

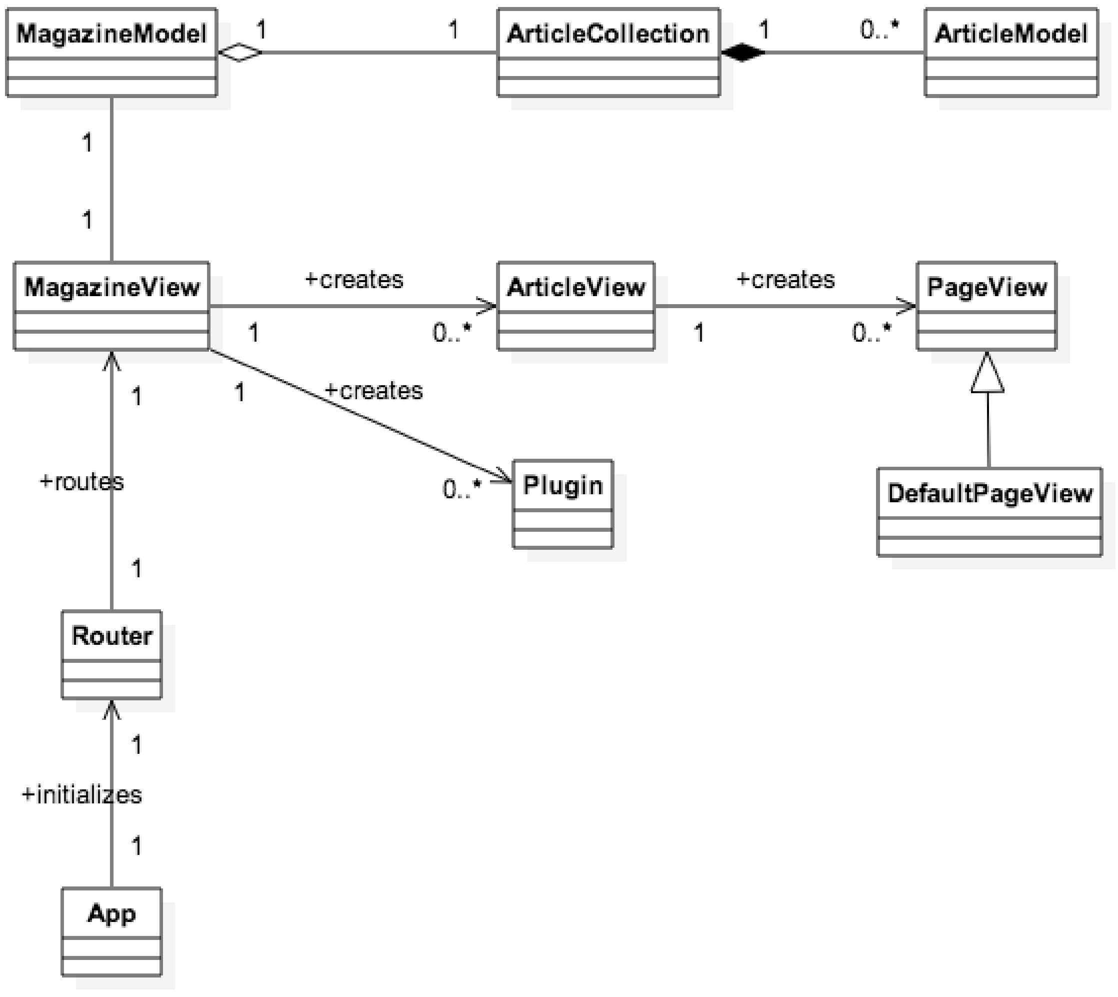


Figure 6: Class diagram of the application structure.

4.3.1 Router

The router is a regular Backbone router. The router's task is to direct the application to the correct view based on the URL inputted by the user. This application has only one view to render, so the router is rather simple. Only two routes are available and they both direct the user to the same view. The first route is an URL with just the id of the magazine, directing the application to render the first page of the magazine. The second route should hold a page number or other identication for a particular page in the magazine. The router directs the view to display this page first.

4.3.2 Models

The application holds two models in it. These are the ArticleModel and MagazineModel. The models don't contain any custom logic and they mainly act as container for the data fetched from the magazine and article APIs. The models are responsible of fetching the data from the correct URL, given the resource id.

The MagazineModel holds information about the name of the magazine, along with a global configuration for the magazine. MagazineModel also includes ArticleCollection that contains all ArticleModels that belong to this magazine. The ArticleCollection is fetched separately and inserted into the MagazineModel once the magazine has been found.

Articles, within the context of this application, are any kind of content that can be presented as one or more pages. The cover page of the magazine counts as a single article, as do regular text articles. ArticleModels present the articles of the magazine. A magazine has multiple articles, which are contained in the ArticleCollection. The API resource endpoint for articles is expected to return a list of articles. When the collection is populated, each of these articles in the list gets inserted into a new ArticleModel in the collection. Each ArticleModel holds the data that should be displayed on the pages. Usually this is at least page title, one or more images and text content. The content's vary for each article and it's the job of the view to find the correct data from the model. Every ArticleModel is also configured to hold the information on which page module should be used to render the article as pages.

4.3.3 MagazineView

MagazineView is the core and the main view of the application. It is what the Router will initialize once it has determined that the URL is in acceptable format. The MagazineView has quite many tasks to perform. It's mainly responsible for creating and populating the data models of the application and feeding the data to ArticleView before presenting the content to the user. MagazineView also creates and controls the carousel, displaying it to the user when the pages are ready.

The MagazineView also works as an event aggregator of the application. It is the glue between the carousel and PageViews, listening to various events from the carousel and forwading them to the related PageView.

4.3.4 ArticleView

ArticleView is the second most important view of the application. A new ArticleView is created for each ArticleModel in ArticleCollection. The ArticleView doesn't act as a regular view as it doesn't actually have anything to display on it's own. Instead it acts as a parent and container for PageViews. Its task is to find the page module resources related to an article and load those. It then has to create the PageView and return that view back to the MagazineView.

4.3.5 PageView

The PageView is a simple Backbone.js view and the only view in this application that has dynamically created content. By default it only renders given HTML into actual DOM elements. The default PageView is quite barebones as this view is designed to be extended to implement custom articles.

For a simple news story without any kind of interactivity, the default PageView with custom template will likely be enough. The alternative is to create a custom PageView that extends the default. Custom PageView's use is to add and implement interactivity on the page. The extended view can make use of events such as ones that trigger when user enters the page or leaves the page. Custom events can also be added for sending forms and pressing buttons on the page. Any user interaction that is possible in browsers can be defined as events here.

The PageViews exist as separate entities from the core application. Other than the default one, the PageViews aren't in the application dependency tree as they are loaded dynamically during runtime. Each PageView is a separate bundle that contains files required for the template to function and display properly. Usually these files are the view, template and stylesheet.

4.3.6 Carousel Library

Carousel is Conmio's internally developed library for building carousel views. Carousel in this case is a set of different elements where only one is visible at a time. The other elements are on the left and right side of the visible element, accessible through user interaction.

It's developed mainly with mobile devices in mind so it fits this application rather well as it is. Most of the application was built around the Carousel library, instead of making the

Carousel work with the application. The Carousel is able to create full screen carousels, which act as the main interface for the user. All other code mostly acts as means to create a DOM element for the carousel to render as a page. Navigation between pages is handled by the Carousel library.

The Carousel is managed by the MagazineView, which creates the Carousel and adds the pages to it. You could think of the Carousel as an extension to the MagazineView. Carousel library also has built in events for communication between the Carousel and other views of the application.

4.4 Flow of Control

This section looks into how the application works. Rendering the magazine is the core workflow of the application and the location of most of the logic of the application. The next subsection looks into how the magazines get rendered to the user. Figure 7 shows this same flow as a sequence diagram.

4.4.1 Magazine Rendering

When the MagazineView gets initialized, its first task is to populate the MagazineModel and ArticleCollection from the server API. The correct resources are identified by the magazine id from the browser URL. The server will respond with JSON data that is used to populate the models. Once the application has the data and the models are populated, MagazineView will initialize the Carousel. The Carousel requires HTML elements to render as pages of the carousel. This task is delegated to the ArticleView. Each ArticleModel within the ArticleCollection will get its own ArticleView.

The ArticleView gets ArticleModel from the parent MagazineView. The model holds information of the name of the page module that should be used to render this article as

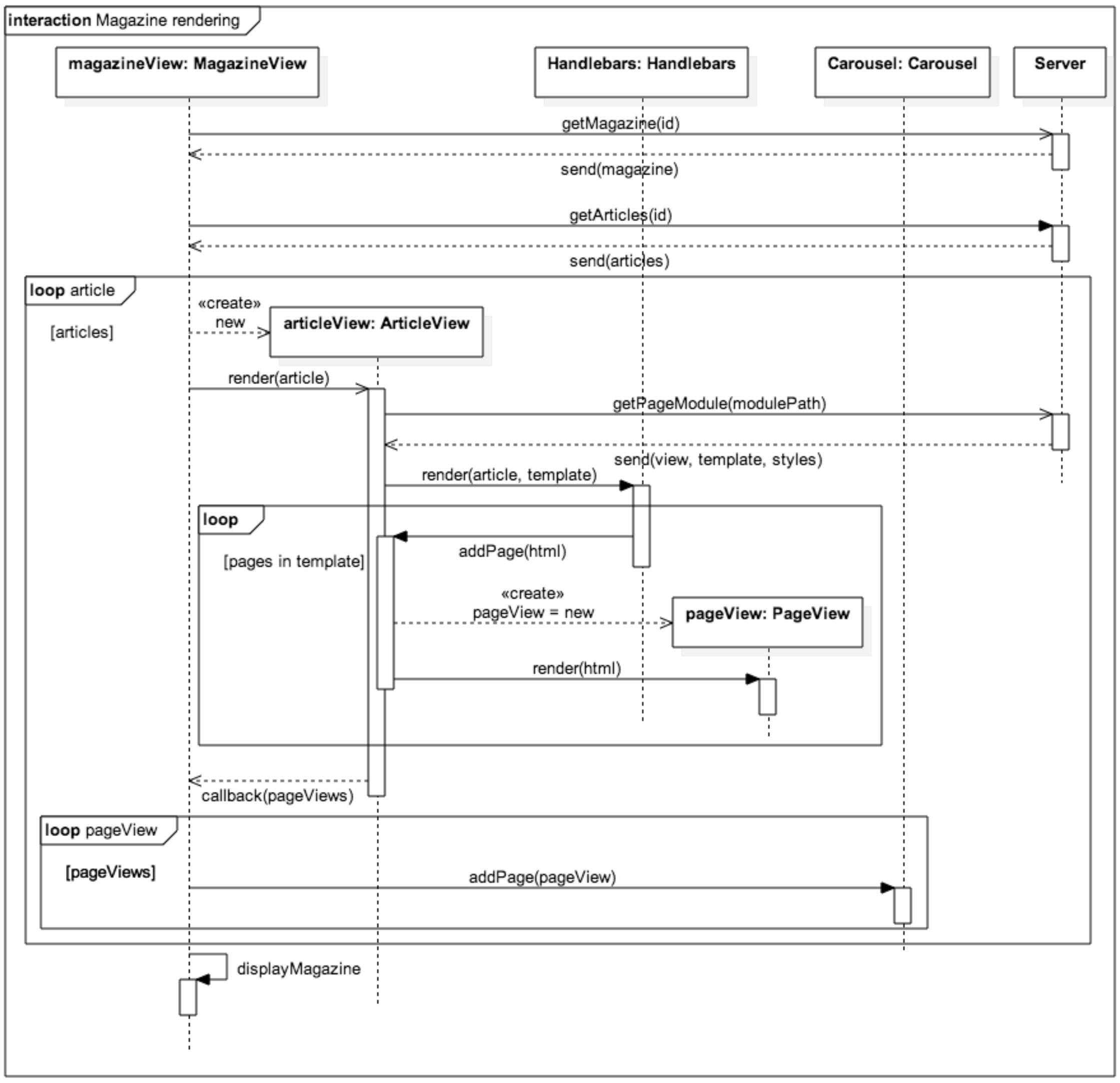


Figure 7: Sequence diagram of the magazine rendering flow.

pages. The name of the page module is used to determine the path to load the module files from. The files have standard names such as `view.js` and `styles.css`, so the view can easily determine which resources to load. All files are optional and default ones will be used if custom ones are not loaded.

Once the ArticleView has the page module files, it can compile the template through Handlebars with data from the ArticleModel. The template will determine how many pages the article will be presented as. For each page in the template, Handlebars will return the page compiled as HTML string. This string is then given to the PageView which will create HTML element of the string and attach itself to the element. The ArticleView can hold as many PageViews as the template has pages.

Once all pages of the ArticleView are rendered, the view will notify MagazineView that the PageViews are ready. MagazineView then retrieves the PageViews and sets each as a new element in the Carousel. The MagazineView continues this, rendering all articles as pages.

The browser URL contains the page index parameter, which tells which page of the magazine should be displayed first. Once the MagazineView reaches this page index while adding the pages to the Carousel, it will immediately display the page to the user. Until now the user has been looking at a splash screen so it's important to get the actual content visible as soon as possible.

Once all pages are rendered, the user can begin interacting with the magazine by changing the pages. From hereon all actions on the application are events based on user input, whether it's user clicking a button or changing a page.

5 Platform

This chapter takes a look at the features that make the application easily customizable and able to function as a platform for multiple different magazines that both look and behave differently.

5.1 Modularization

The application was designed to be easily customizable and extensible. To achieve this, a majority of the application was designed as separate decoupled modules that are not included in the core application. These modules can be split into two different types which are pages and plugins.

The idea of modules is to keep the core application file size as small as possible. All modules are loaded dynamically during runtime. Which modules get loaded, depends on the magazine and article configurations. A single magazine isn't likely to have more than two or three page modules and plugin modules. The dynamic loading of modules does bring some additional overhead to the application startup. The alternative would be to include all page and plugin modules within the core bundle but this would result into a large amount of unnecessary code for a single magazine. The dynamic loading makes the application quite flexible, as the core can be cached almost indefinitely and different magazines can utilize the same core.

During the build, modules are optimized in the same way as the main application, except they are each their own package separate from the main application. All modules have their own dependencies and can introduce new third party libraries to the package.

5.1.1 Pages

Pages are modules that are attached to articles. Generally the pages consist of three different files: Backbone view, Handlebars template and Sass stylesheet.

Pages take the most benefits for modularization. A single magazine isn't likely to have more than a few different page modules. Most of the time this is either two or three templates. But with multiple different magazines, all having different templates, the need for multiple page modules increases sharply. At this point it isn't feasible to bundle these in the main application. If these were all bundled with the core application, the file would include quite many useless pages for the current magazine leading to unnecessarily long load times.

Listing 4 shows an example Backbone view for the custom page. This view has two dependencies: the default page view and Facebook API library. The default page view is used as the basis for the custom view. The custom view extends the default view, gaining all methods and properties of the default view. The most important one is the list of events. The parent view attaches method calls to various existing events, such as the `onPageChange` in the above example. The Facebook API library is a third party library that's installed separately, but the dependency can be loaded in the same way as our own modules.

```
define([ 'views/page', 'vendor/facebook' ],
  function( DefaultPage, FB ) {
    var CustomPage = DefaultPage.extend({
      // Extend parent view events with custom event
      'events': _.extend({
          'click .facebook-share': 'onFacebookShare'
      }, DefaultPage.prototype.events),

      // Cache elements after template is rendered
      afterRender: function () {
        this.$header = this.$('.header');
      },

      // Trigger animation when user enters page
      onPageChange: function () {
        this.$header.addClass('play-animation');
      },

      // Share current page to facebook
      onFacebookShare: function() {
        FB.ui({
          method: 'share',
          url: window.location.href
        })
      }
    }, {
      handlebarsHelpers: {

      }
    });

    return CustomPage;
  }
);
```

Listing 4: Simple page view

The task of the Backbone view is to introduce interactivity to the page. Most of the time this is achieved by attaching the view to various global events of the magazine or custom DOM events attached to the elements on the page. The magazine offers a wide range of useful events for pages to use such as: `pageLeave`, `pageEnter` and `magazineReady`. For example, the `pageEnter` could be used to trigger animations when the user navigates to the page, avoiding animations running in the background. An example of this can be seen in listing 4. These events are automatically attached to a function with a predefined name, in case of the example, the `onPageChange`. The developer doesn't need to explicitly define these events as they are always available. The DOM events have to be defined in the `events` property of the view. They can be attached to any element on the page. The code in listing 4 attaches a click event on an element with class `facebook-share`. When the button is clicked, we trigger a Facebook sharing popup. The events are a simple yet powerful way to introduce interaction to the page.

Another important feature of the view is to introduce logic to the template. As Handlebars is a logicless templating engine, we need a place where to define this logic. Handlebars features helpers, which are custom functions that can be used inside the template. The view object has a special property `handlebarsHelpers` where the developer can define custom helpers that will be made available in the template. These helpers will only be available in the template attached to this view, although some global ones are available. One of these is the `page` block helper. This one is used to mark multiple pages in simple template. It's required on the template on every page.

Page modules also usually include the Handlebars template that defines the HTML markup of the view. A Sass file can also be included. By using Require.js plugins, these different files can even be included in the dependency tree leading to just a single file as the module output.

5.1.2 Plugins

Initially the application had designs for various additional UI elements such as page indicators that float above the page at the top. These features didn't fit some magazines so they had to be made optional. The preference was to keep the core of the application as lean and uncluttered as possible, meaning that it wouldn't have any optional features in the main application. The optional features should be loaded separately as sometimes they could even introduce third party libraries.

Plugins are modules which can affect entire magazine instead of just single pages. The magazine feed includes a section for a list of plugins. Each listed plugin gets loaded during the application startup with user defined configurations from the feed.

Generally, the plugins are allowed to do anything to the magazine. At its simplest, a plugin can be just a new UI element that floats above the page view. Plugins can also affect the entire flow of the magazine. They can have access to the order of the pages in magazine and manipulate it. For example, a plugin for advertisements could inject new full page advertisements to the magazine at certain page change intervals. Plugins can also be used to block certain devices such as desktop or mobile from accessing the magazine at all.

5.2 Continuous Integration

As we have previously determined, the entire application consists only of static files. This makes the requirements for deploying the application to production rather relaxed.

Some years ago the deployment of static websites didn't require much concern. Everything was written to be browser compatible from the beginning and the files could be uploaded to the server by using FTP. But as the browsers themselves have improved, the applications have grown larger and more complex. Tools such as Require.js and Sass have been developed to ease the development of web applications. But on the downside these tools often require a step of preprocessing before the application can be deployed to the server for browser usage.

To help handle these preprocessing tasks, task runners and build systems were made. They are able to do the preprocessing tasks that the most common tools and libraries require. They help with tasks such as running file preprocessors, optimizing files for browser and running unit tests. One of the latest and potent task runners available is Gulp.js

5.2.1 Build System

Gulp.js is a streaming build system, made for automating development tasks. Gulp is written on Node.js which is a server side platform for Javascript. Therefore Gulp also uses Javascript to write the build tasks. Gulp uses streams to handle files. The tasks take a stream of files and run that stream through various plugins that manipulate the files inside the stream. At the end of the stream pipeline the output gets written as a file to a chosen directory.

```
var gulp = require('gulp');
var concat = require('gulp-concat')
var uglify = require('gulp-uglify')

// Concatenate all javascript files to single file
gulp.task('compile', function() {
  return gulp.src('src/**/*.js')
    .pipe( concat({ fileName: 'script.js' } )
    .pipe( uglify() )
    .pipe( gulp.dest('build') );
});
```

Listing 5: Gulp.js task example

Listing 5 presents a simple task for pulling all Javascript files from a directory and concatenating those into a single file to the build directory. For manipulating the streams Gulp requires plugins. In the same example the gulp-concat plugin is used. This plugin takes multiple files from the stream and outputs them as a single file back into the stream. Another one sampled is the gulp-uglify plugin which is used to optimize the file size of the scripts. These plugins are installed from npm, the node package manager. While custom plugins can be created to execute any possible task, the registry already holds plugins for the most common use cases and libraries.

Another very popular task runner is Grunt.js. All tasks can be done with either task runner and the choice between these task runners comes down to personal preference. The biggest difference between these two is that Gulp prefers code over configuration, which leads to simpler and more flexible tasks. The choice of Gulp over Grunt for this project was merely the will to try something new. [18]

5.2.2 Build Tasks

The application makes heavy use of Gulp to run various different tasks to optimize the application for production usage. These tasks can be split up to three main groups based on the file type: HTML, Sass and Javascript.

HTML optimization includes minifying the main HTML file of the application. Minification in this case removes all unnecessary whitespace and new lines from the file. The other job of the HTML task is to precompile the Handlebars templates. Handlebars precompilation creates Javascript files of the from the Handlebars markup removing the need to compile these during runtime. This reduces the time required to render the templates to the browser when using the application.

Sass compilation is the simplest of the tasks, but it's also the most important in the sense that the application is unusable without running this task. While the other tasks just optimize files, the Sass syntax is unusable by the browser as is. The Sass files have to be compiled through the Sass compiler which is an external Ruby application. To ease the stylesheet development, an additional watcher task exists that watches the Sass files for changes and automatically executes the compilation task, refreshing the browser when it finishes.

The Javascript optimization task acts as a wrapper for the RequireJS optimizer. All it does is get the Javascript file that bootstraps the application and run the optimizer with that file. RequireJS doesn't play well with Gulp as it was not intended to be used with streams. Therefore we can only supply the path to the file where the optimizer should start. The RequireJS optimizer will then build the dependency tree from that file and combine the files into a single file with all required code. Javascript files are also optimized to have the smallest possible file size.

5.2.3 Optimization

Javascript optimization doesn't just remove whitespace. It can also parse through the code and optimize code structures to smaller counterparts. Often variable names are renamed to single letter names while making sure that the code is still able to run.

These optimizations have no effect on how fast the code runs. They only reduce the size of the produced file. On websites intended for mobile devices, small file sizes are important to lessen the time to load the site on slower networks. Bundling multiple files into a single file might seem to contradict with making files as small as possible. But in fact establishing multiple connections to load multiple files is even more expensive task for the browser to execute.

The ability of RequireJS to load files dynamically during runtime makes it possible for the application to work without optimizing Javascript files. This makes the development fast as the developer doesn't have to wait for the optimizer to run while developing. It also makes it easy to see the benefits of optimization as presented in table 1.

Table 1: Comparison of application file sizes

Type	Javascript	CSS	Handlebars	Total
Unoptimized	1014KB	76.1KB	19.9KB	1.2MB
Optimized	127KB	7.6KB	0KB	256KB

5.2.4 Deployment

Since the application is all static files and the build and optimization tasks are already included, the deployment is simple. To serve static files from a server only a simple HTTP server is required. It doesn't need to support PHP, Java, Node.js or any other server application. Content Delivery Networks (CDN) are a great way to serve static sites and applications. CDNs consist of servers distributed over large areas, often globally. They serve only static content such as text and images so the servers area cheap to deploy globally. The main benefit of them is allowing the client to choose the geographically closest server to load the content from. Due to their high-availability and -performance they are used to deliver a large fraction of internet content these days. [19]

Before the application can be deployed to a CDN it has to be built. The build tasks for deployment are the same ones that are used during development and are bundled with the application code. After running the deployment tasks, the deployer only has to pick the produced files from the correct location and push them to the CDN. All of this could be done manually but a continuous integration (CI) server to automate the above process is recommended.

6 Results

The aim of this project was to create a web application for consuming existing digital content in a magazine format on mobile devices. The resulting application fulfilled the requirements and was a functional proof of concept. For now, the application is likely to be used as a demo application for the sales team of the company.

The biggest hurdles of the development and the end result was the lack of performance on web usage on all but the highest end of mobile devices. Faster development time, easy distribution and high availability are among the biggest benefits of developing web applications. Unfortunately the lack of performance on lower end devices diminishes these benefits, making native applications more suitable for such animation heavy applications in the current scheme of things.

During the end of the development, new libraries and frameworks have risen, the most promising one being React.js. Building the application from the ground using the more recent utilities could have potentially brought great performance improvements for the application. For a wide range production usage, a revisit of the used technologies and libraries could be beneficial. The web is also moving forward at a great speed and the performance issues could be resolved by just waiting for better hardware and more efficient browsers.

During the end of the proof of concept application's development, work began on native application prototypes. The native applications were developed for both iOS and Android. They were able to utilize the exact same content feeds as the web application, making same magazines available for all platforms. The native applications were much more efficient than the web counterpart. They weren't, however, able to quite reach the levels of flexibility, customization and ease of development of new templates. A possible middle ground would be a hybrid application, handling all animations natively yet using web views for rendering the templates. Therefore it remains to be seen which is the most suitable platform for this application.

7 Summary

The purpose of this thesis was to document the technical and architectural choices behind an application to build and read digital magazines. This application would be both a prototype and a demo application for the sales of Conmio Oy.

The first chapters looked into existing applications, such as Flipboard and Yahoo News Digest, that acted as inspiration for this application. The chapters also took a look at the application concept and design decisions that shaped the later technical choices and created the root for the application. Once the concepts and requirements were clear, the following chapters looked into the technical choices and how those libraries and tools were used to construct the application. The technical choices also helped to shape the concept of the application further. The application was developed with proven to work Javascript libraries such as RequireJS, Backbone.js and Handlebars. The technical choices like RequireJS encouraged to build the application in a modular way, making it able to function as a platform for different kinds of digital magazines. The ease of development was also a very important part of the application development. Build system Gulp.js was used to make tasks to ease the development and optimize the application for production usage. Eventually these tasks would be used when deploying the application on a server. The results concluded with thoughts on continuing the future development as a native application.

References

1 Silva DS. The Future of Digital Magazine Publishing; 2012. [Online]. Available from: http://elpub.scix.net/data/works/att/109_elpub2012.content.pdf [Accesssed 1.4.2015].

2 MacManus R. How Flipboard Was Created & its Plans Beyond iPad; 2010. [Online]. Available from: http://readwrite.com/2010/10/06/how_flipboard_was_created_its_plans_beyond_ipad [Accesssed 1.4.2015].

3 Newton C. Yahoo's sleek News Digest app swims against the stream; 2014. [Online]. Available from: http://www.theverge.com/2014/1/7/5284300/yahoos-sleek-news-digest-app-swims-against-the-stream [Accesssed 1.4.2015].

4 Insights N. Web Apps vs. Native Apps: Fight… Fight… Fight!; 2014. [Online]. Available from: https://medium.com/netxtra-insights/web-apps-vs-native-apps-fight-fight-fight-c43e55b31649 [Accesssed 1.3.2015].

5 Marohnic V. Would push notifications for HTML5 apps kill native apps?; 2013. [Online]. Available from: http://thenextweb.com/dd/2013/02/09/would-push-notifications-for-html5-apps-kill-native-apps/ [Accesssed 1.3.2015].

6 Puplus DF. Rise of the SPA; 2013. [Online]. Available from: https://medium.com/@dpup/rise-of-the-spa-fb44da86dc1f [Accesssed 1.3.2015].

7 Wroblewski L. Mobile First; 2009. [Online]. Available from: http://www.lukew.com/ff/entry.asp?933 [Accesssed 1.3.2015].

8 Mobile First. Zurb Foundation;. [Online]. Available from: http://zurb.com/word/mobile-first [Accesssed 1.2.2015].

9 Introducing JSON;. [Online]. Available from: http://json.org/ [Accesssed 1.3.2015].

10 Osmani A. Developing Backbone.js Applications. O'Reilly Media; 2013.

11 Backbone.js; 2014. [Online]. Available from: http://backbonejs.org/ [Accesssed 1.3.2015].

12 Sass; 2014. [Online]. Available from: https://github.com/sass/sass [Accesssed 1.3.2015].

13 Osmani A. Writing Modular Javascript with AMD, CommonJS & ES Harmony; 2012. [Online]. Available from: http://addyosmani.com/writing-modular-js/ [Accesssed 1.4.2015].

14 Why AMD?;. [Online]. Available from: http://requirejs.org/docs/whyamd.html [Accesssed 1.4.2015].

15 AMD;. [Online]. Available from: https://github.com/amdjs/amdjs-api/blob/master/AMD.md [Accesssed 1.4.2015].

16 RequireJS; 2015. [Online]. Available from: https://github.com/jrburke/requirejs [Accesssed 1.4.2015].

17 RequireJS Optimizer; 2014. [Online]. Available from: http://requirejs.org/docs/optimization.html [Accesssed 1.3.2015].

18 Gulp.js – The Streaming Build System; 2014. [Online]. Available from: http://slides.com/contra/gulp#/ [Accesssed 1.2.2015].

19 Support R. What is a CDN?; 2015. [Online]. Available from: http://www.rackspace.com/knowledge_center/article/what-is-a-cdn [Accesssed 1.4.2015].