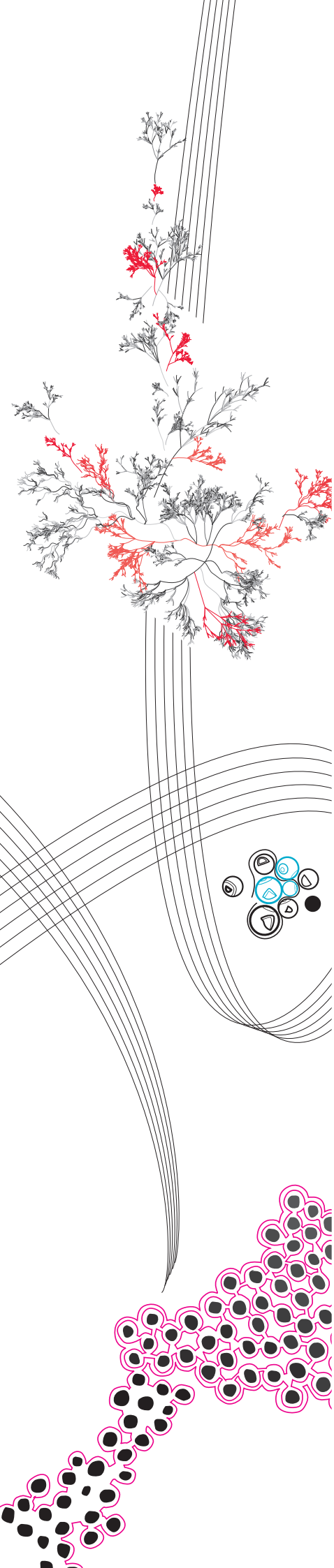

December, 2016

MASTER THESIS

# ON-DEMAND APP DEVELOPMENT

Thom Ritterfeld

**Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)**
**Chair**

Exam committee:
L. Ferreira Pires
A. Rensink

Documentnumber
s1509101 — 1.1

# Preface

This thesis is submitted as partial fulfilment for the Computer Science Master's Degree of the author. It contains work done from May, 2016 to December, 2016. The supervisors on the project have been L. Ferreira Pires and A. Rensink.

The idea for this research topic evolved after years of app development. I developed multiple apps for agencies, start-ups and corporations. At these companies, I worked on a wide area of apps, such as news, social media and commercial apps. During development of the apps I used different languages and technologies across all platforms. Despite the differences, I noticed that I faced the same problems and patterns again and again, especially in the case of updating apps in a timely manner. This motivated me to find a solution and write this thesis.

# Table of Contents

# Chapter 1

# Introduction

This chapter is structured as follows: Section 1.1 gives the motivation for this thesis. Section 1.2 describes the research objectives. Section 1.3 describes the approach to be followed to achieve the objectives. Section 1.4 presents the structure of the report.

## 1.1 Motivation

Applications (apps) are everywhere around and with you, think about Pokémon Go, WhatsApp or CNN (see Figure 1.1). They are all trying to make connecting and engaging with digital content easier than ever before. Apps are usually downloaded ones and changed to provide extra functionalty or another experience, like new icons in Pokémon Go, not only texting but also calling contacts in WhatsApp or watching video clips next to the news items inside the CNN app.

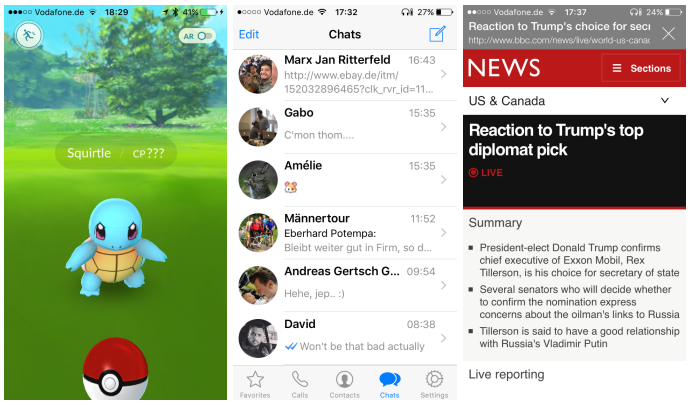


Figure 1.1: Pokémon Go, WhatsApp and CNN

Nowadays, almost all mobile devices have powerful processors and provide fast Internet access. These two properties have led to new possibilities for building apps. Many apps give information about fast-changing content, such as news festivals and catalogues. Therefore, there is high demand for apps that allow to be updated and expanded even quicker and smoother. However, most frameworks that offer tools for the development of apps, still require numerous

steps and a great deal of experience to develop an app[1].

Many companies design their apps to use web page solutions by means of HTML interface elements because it is easier, faster and cheaper to change a web page. This is due to the fact that the web is flexible, changeable and able to grow according to business needs without changing and building apps for multiple platform every time. The downside of using web pages is that they do not work offline and do not provide a user experience that is as good as native interface elements[2]. This is the reason why the web page based solutions do not qualify as a good alternative, in terms of user experience and offline usage[3].

After time, functionality and needs of an app change because of the impact in technology, business or society. The changes that need to be applied then are so called updates. Primarily, it takes a significant amount of time (at least a few days) to apply updates to an app, and even longer for the update to reach all users. Moreover, some users do not update their phone or their apps at all, which means a great deal of legacy maintenance is required. Research has shown that among users who do not allow automatic updates, 18.5% rarely update their mobile apps and 45.6% update less than once a month[4].
The consequences are, firstly, that users are not getting the latest features and experience a company wants to provide. Secondly, the company loses time and is not able to adequately react to changes on the market. Lastly, developers need to change an app in a very short time frame, which makes the app more liable to bugs and unstable features[5].

Previous research[6] has shown that some solutions allow apps to be changed at runtime. The paper referenced here describes a remote Model View Controller (MVC) architecture, which synchronises the user interactions of the app between all client devices.
This solution appears to perfectly fulfil the needs of updating an app dynamically, as it provides an architecture that allows the user interface to be changed at runtime. However, the solution also has disadvantages:

- **No offline usage is possible**
  The user always needs an Internet connection to interact with the app, which is not always available on a mobile device. The app should be able to work at least with the previously fetched user interfaces and content.
- **There is a centralised controller on a server**
  This means that all user interactions with the app are handled by the remote server. The server load could be very high if many people are using the app at the same time, which slows the solution down. It is therefore hardly scalable.

To process of updating an app is either slow and complicated, or provides a bad user experience. Remote MVC could be a solution, but, as of today, it is not scalable and requires an Internet connection on the target device in order to work. Thus, neither of these solutions is suitable to update an app dynamically.

## 1.2 Objectives

This research aims to address the two drawbacks of remote MVC, so to facilitate dynamic updating of the app and improve the user experience.

The main objective of this thesis can therefore be described as follows:
*To create a solution allowing decentralised and offline usage of a mobile app using the principles of the remote MVC pattern.*

To achieve this objective, a framework must be developed and then used by a prototype app. This methodology helps to achieve the main research objective. During development of the framework the following questions are considered:

**RQ1. What are possible approaches to decentralise the user interaction of the remote MVC pattern?**
A new software architecture must be designed which makes it possible to run the remote MVC pattern on the platform-specific MVC architecture[6], without the need for an active Internet connection. This allows the app to be used in a decentralised way, since it only requests data for changes and not for making the user interactions functional.

**RQ2. What are possible approaches to make the remote MVC pattern work offline?**
The new software architecture should also work without an Internet connection. This could be done by storing the dynamic app locally and use the stored data without being connected.

**RQ3. How can a framework be developed that allows the app to be changed dynamically when there is a connection, but still supports offline usage?**
A new architecture is needed which should be capable of running remote MVC locally, while no longer requiring neither an Internet connection nor a central remote controller.

## 1.3 Approach

The main objective of this research will be resolved by the following approach. The approach is mainly based on creating a prototype which demonstrates the capabilities to solve the problems of the main objective.

**To find a solution to the main objective of this research, the following steps are taken:**

1. Gather background information about the approaches to build apps and the MVC patterns, its offline usability and decentralisation.
2. Identify requirements to be able to define an architecture that is capable of giving an answer to the main objective. Which additional requirements give a solution to the main objective and which existing requirements need to persist.
3. Define an architecture to use for developing the framework. The architecture should be able meet all the defined requirements.
4. Implement a prototype that demonstrates the framework. The prototype proves that the framework works and provides the ability to compare the approach against other app development approaches.
5. Verify whether the framework complies with the requirements by performing a change scenario. A change in the code base will simulate an app update and allow to compare the framework before and after the update against other approaches.

## 1.4 Structure

The structure of the thesis is as follows:

**Chapter 2 - Background** Describes background information of app development and previous work on the remote MVC pattern.

**Chapter 3 - Requirements** Lists the requirements in order to define an architecture, and be able to measure if the framework meets the needs.

**Chapter 4 - Design architecture** The architecture that is used to build the framework. The architecture gives a solution to the objectives RQ1, RQ2 and RQ3.

**Chapter 5 - Prototype** Reports on the implementation of the prototype.

**Chapter 6 - Validation** Validates the correctness of the prototype

**Chapter 7 - Reflection** Concludes this report and suggests topics for further research.

# Chapter 2

# Background

This chapter justifies this research. It presents background information about basic concepts in app development, current approaches in app development, composition of an app, server communication and the update process.

## 2.1 Basic concepts in app development

This section describes the basic components needed for develop an app using current approaches.

### 2.1.1 Operating system

The mobile Operating System (OS) typically starts up when a device powers on, and fills the screen with icons. Operating systems manage hardware and connectivity, and are responsible for loading and managing apps and their processes, memory and storage. Most operating systems are also tied to specific processors and hardware.

### 2.1.2 Frameworks

An application framework is a collection of software that provides a structure to support the development of an app. A framework can acts as the basic skeleton for building an application. The intention of designing application frameworks is to lessen the general issues faced during the development of applications. This is accomplished by the use of code that can be shared between different modules of an application. Application frameworks are used not only to build the graphical user interface (GUI), but also in other areas to store data or provide algorithms for recurring calculations. Frameworks give the ability for developers to develop apps faster and share the same modules of software with other apps and developers in an easy way.

### 2.1.3 Libraries

A library is basically a framework but provides a lot of functionality. Most of the time a library is used as starting point to develop an app. It provides general classes and structures to define the app. For building iOS apps for example, CoreFoundation and Cocoa are used by default. These

sample libraries contain for example classes for an NSString to store texts, UIButton interface element to create and represent buttons on the screen.

### 2.1.4 Programming language

A programming language is used to write the code of a program. This code is interpreted by a compiler, and the compiler constructs multiple binaries for all devices using the libraries of the platform. In the end, this binary runs on the mobile phone, by which the OS knows how to set up the environment and resources that are needed to run the binary. In other words, an app is essentially a collection of binary code.

### 2.1.5 Development environment

The development environment is a collection of procedures and tools to develop, test and debug an app. All platforms provide an IDE, simulators to simulate the app on the local machine and libraries with interface objects, which allow developers to develop and test apps.

### 2.1.6 Platforms

A platform in the context of this paper refers to the tooling the vendor provides to develop apps for a mobile operating system. It contains a development environment, programming language and libraries. The main vendors of mobile operating systems are Apple, Google, Microsoft and Samsung. These all provide hardware and software to build and consume apps. The three main operating systems are iOS, Android and Windows Phone. Each operating system uses a different development environment and programming languages. The vendors also provide a way for developers to distribute the apps they build, via app stores.

## 2.2 Development approaches

There are several approaches to building apps. Each approach has its own benefits in terms of time to develop, complexity and performance. All approaches share a layered architecture principle: they run on top of the operating system of the operating system (see Figure 2.1). The number of layers in a software architectures increase the complexity and decreases the performance, but abstraction allows faster and easier development approaches[7]. For a developer, it is important to make the right decision to achieve the proper balance between complexity and performance (for all decisions see Table 2.1).

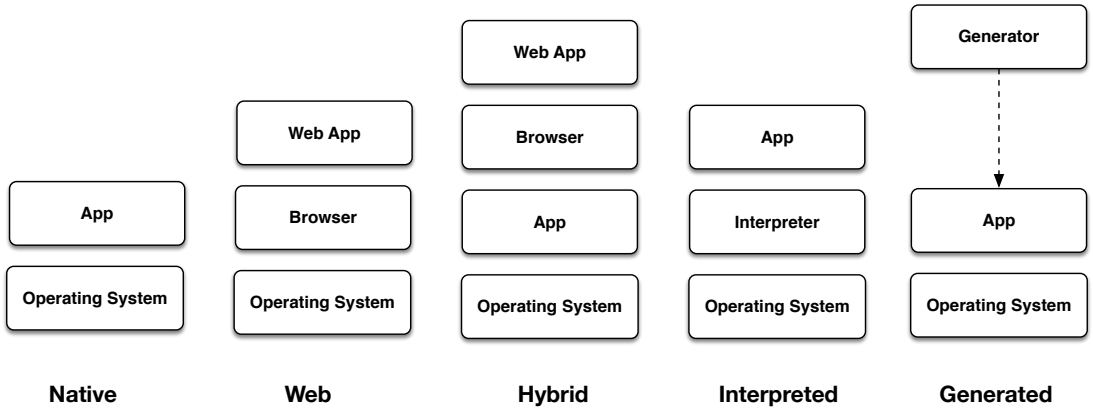


Figure 2.1: Difference in architectures between approaches [7].

### 2.2.1 Native

The native approach performs best by far. This is because it is directly coupled with the operating system and the libraries the platform provides. However, the development costs are high, the time-to-market is long, update processes are complex and the code of the app is not reusable across different platforms because the use of different programming languages[8][9].
The native app runs directly on top of the operating system. The written code uses the frameworks and programming language the platform supports, and is optimised for the device; in addition, the layout rendering is done using the native objects of the platform. Because of this, an app developed with this approach performs best[10][7].
The most popular frameworks are those provided by the operating system; these are CoreFoundation for iOS[11], Application Framework for Android[12] and the .NET Compact Framework for Windows Phone[13].

### 2.2.2 Web

Web apps are browser-based applications in which the software is downloaded from the web[9]. The web approach is easy to update, easy to maintain and functional on all platforms. It also has a very short time-to-market without added costs which makes it flexible compared to other approaches. Every platform has a browser that allows the user to access the app.
Web apps are based on widespread Internet technologies such as HTML and JavaScript. The main disadvantage of web apps is the limited access to the underlying device's hardware and data. Another problem is the extra time needed to render the web pages and the extra cost needed to download the web page from the Internet[8][10].
Examples of the most popular frameworks to implement web apps include JQuery Mobile[14], Sencha Touch[15] and JQTouch[16].

### 2.2.3 Hybrid

The hybrid approach is a combination of the native and web approaches. It basically runs a browser inside a native app. This approach offers the same benefits as using the web approach.

Another benefit is that apps are able to access the hardware this way. However, the downside of this approach is that there is an extra layer which prohibits the app from running smoothly and providing a native experience[8][9].
Hybrid apps are neither purely native neither purely web-based. Not purely native comes from the layout rendering that is done via HTML in the web browser instead of using the native language and objects of the platform, whereas not purely web-based results from the lack of support of some browser-specific features, which are now accomplished using native wrappers. An example of the most popular container for creating hybrid mobile apps is PhoneGap[17]. Most of the time a web framework as described in the previous section is used in addition to a hybrid framework to draw the user interface.

### 2.2.4 Interpreted

The interpreted approach runs another interpreter app inside a native app. This means that the approach is more flexible than the native approach. The interpreter app is capable of reading another language as well as bridging to the native language. This enables free choice of the programming language and a native experience for the interface. The downside of this approach is again poor performance and only being able to share the logic across platforms, not the user interface.
In interpreted apps, native code is automatically generated to implement the user interface. The end users interact with platform-specific native user interface components, while the application logic is implemented independently using several technologies and languages, such as Java, Ruby, XML etc[9]. Because the application logic runs in its own environment and only the user interface runs natively, it requires more resources and does not performing as good as a native app.
An example of one of the most popular software development environments for creating interpreted apps (as well as hybrid apps) is Appcelerator Titanium Mobile[18].

### 2.2.5 Generated

The generated approach is similar to the native approach, since it generates an app using a set of instructions. Most of the time this approach is used by model-driven techniques[19] in which the developer focuses on describing the problem domain in a model[9]. The model is able to generate apps for different platforms. Because the generated apps are native apps, they have good performance as well.
Generated apps achieve high overall performance because they are essentially native apps, where all the layout rendering is done using the native objects of the platform[20].
A popular example of software development environment for creating generated apps is Applaus[21].

| | Native | Web | Hybrid | Interpreted | Generated |
|---|---|---|---|---|---|
| **Available in app store** | Yes | No | Yes, not guaranteed | Yes | Yes |
| **Common technologies** | Yes | Yes | Yes | Yes | No |
| **Hardware access** | Full | Limited | Limited | Limited | Full |
| **Look & feel** | Native | Simulated | Simulated | Native | Native |
| **Performance** | High | Low | Medium | Medium | High |
| **Cross platform** | No | Yes | Yes | Yes | Yes |
| **Offline** | Yes | No | Yes | Yes | Yes |
| **Development cost** | High | Low | Medium | Medium | High |
| **Code re-usability** | Low | High | Medium | Medium | Medium |
| **Security** | High | Low | Medium | Medium | High |
| **Potential users** | One platform | All | Multiple | Multiple | Multiple |
| **Quality UX** | High | Okay | Good | Good | Native |
| **Ease of updating** | Complex | Easy | Medium | Medium | Complex |
| **Time-to-market** | Long | Medium | Short | Short | Long |

Table 2.1: Overview of differences between the approaches combined[9][22][23].

## 2.3 Composition of an app

This section describes the common composition of an app and a newly developed alternative. These are defined as programming patterns which means they can be implemented by all approaches.

### 2.3.1 Model-View-Controller (MVC)

The Model-View-Controller (MVC) pattern, is widely used in all approaches. This pattern is used to separate different components in an application, namely the models, views and controllers (see Figure 2.2). By separating these components, the developer is able to develop an app because it represents the logical way of how an application is built. In particular, the models represent the data, views represent the graphical representation, and controllers act like glue and allow the models and views to communicate with each other. In this way the pattern defines the separation between components inside an app and makes it easier to understand the structure of the app. Apps using the MVC pattern are more easily extend able than other applications where the separations of concerns is different. The reason for this is that many objects tend to be more reusable and the class interfaces are better defined, as the same button views for example are probably reused in other views inside the same app[24][25].
Although this pattern could work for all platforms, there are slight differences. Every platform uses its own kind of MVC composition, that works better with the provided language and libraries. This is particular one of the main reasons why cross-platform frameworks have difficulties to share the same code across platforms[26].

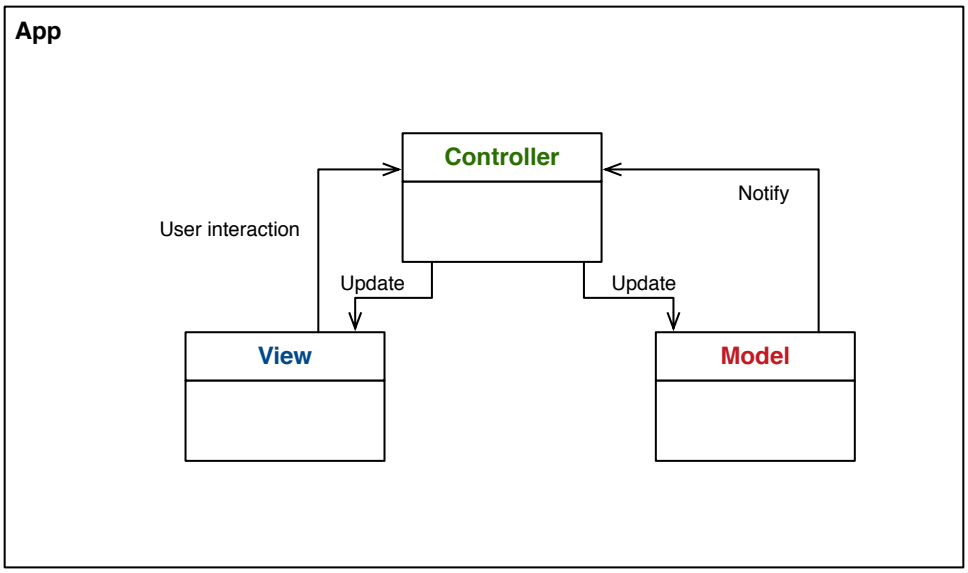


Figure 2.2: The MVC model of iOS [26].

### 2.3.2 Remote MVC

In a remote MVC pattern[27], the views are exposed to other devices. This approach allows to share the user interface of a MVC pattern, to compatible apps shared with other devices that have the ability to adapt the user interface to their specific look and feel. To accomplish this, the remote user interface and the application state are synchronised using a web-based event-driven system. This system uses events that are exposed during user interactions in the interface, the views receive these interactions. The views syncronise the interactions with a server and other clients receive these to reflect the same changes on their interfaces (see Figure 2.3).
This shows that the remote MVC pattern is capable of synchronising interfaces, which is close to our main objective. The problem is that it does not allow to change the controllers of the MVC pattern, which makes offline usage impossible. Because the solution needs to constantly synchronise the interactions with the server it is also not scalable.

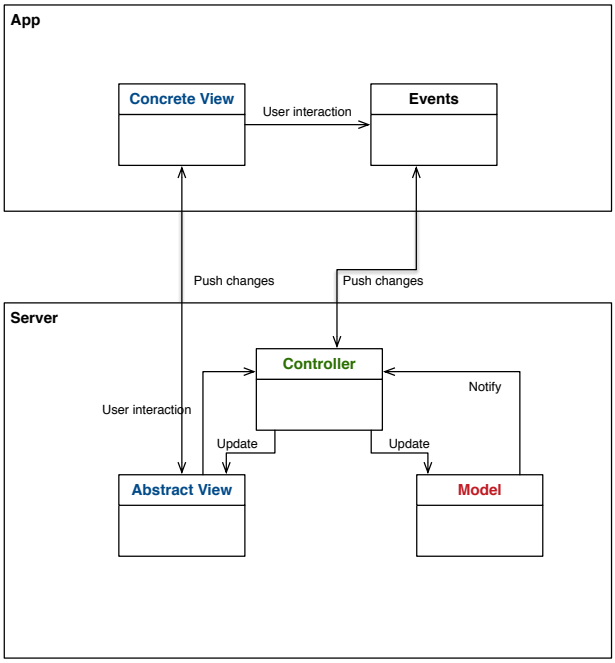


Figure 2.3: Remote Model-View-Controller[27].

### Offline usage

The main difference between a web page and an app is that an app is able to work offline. Although there are apps which are content based, they still have a desired behaviour and allow to interact with the interface and previously fetched content. This is a significant difference between a website and an app. In figure 2.4 an sample is shown where the device is in Airplane mode, which means not connected to the Internet. Apps allow offline interaction, whereas a website without connection is not loading at all.

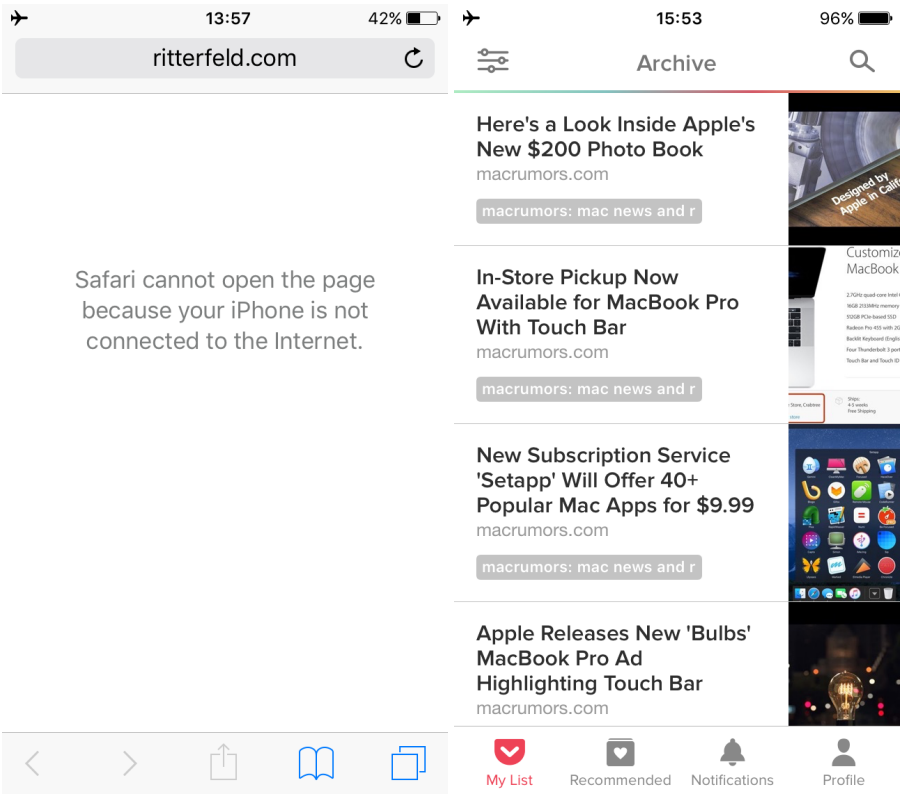


Figure 2.4: Web app(left) versus an app(right) without Internet connection.

### Decentralisation

The main goal of the remote MVC pattern (see Section 2.3.2), is to synchronise multiple devices to represent and replicate the same user interface and actions on all devices at the same time. By sending actions, waiting for the response and changing the interface.
The same scenario happens for websites, where the user requests a website in the browser, the server sends a response, and another interface is shown.
This is in contrast to mobile apps which are used and in control of the user, only the user is able to control the app on one device, and remains in control even when the app is used in areas with poor connectivity. Which can be a problem as for decentralised techniques an active server connection is required to use the app.

### Differences

The differences between the MVC pattern (see Section 2.3.1) and the remote MVC pattern (see Section 2.3.2), are the following:

- **Moving the MVC pattern to a server.** In the research they replicate the MVC pattern on a device that acts like a server.
- **Replicating the view part on remote devices.** All clients push the user interactions on the implemented abstract view to the server.

- **Keeping the view part in sync using event-based technologies.** The clients are fetching user interactions from the server, and check if they need to display another concrete view or update the current concrete view.

## 2.4 Server communication

Apps that connect to web services are widely available. Think about news, weather, stocks and chat applications. Most of these apps are connected to web services using RESTful application program interface (API)s and there main purpose is to deliver content to an end user, which consumes the content using an app on his mobile phone.

### 2.4.1 RESTful

In Representational state transfer (REST) everything is about resources. A resource describes an object and can contain sub resources or sub collection of resources. Because of this flexibility REST is popular and used by many web services. All RESTful services are based upon Hypertext Transfer Protocol (HTTP) and use resource identifiers, in form of URLs to gather content.
HTTP is a high level network protocol which offers already a lot useful functionality. For example, resources can be managed in a CRUD(Create, Read, Update, Delete) way, HTTP provides the same kind of methods (POST, GET, PUT, DELETE). Because of this and other properties like authentication HTTP fits nicely to the main idea of REST.
Resource identifiers are another important property of REST. Most of the time the name of the resource also acts as an identifier. For example, to get all contacts of an Address Book that are stored in a REST service, you just need to do a GET (Read) on the resource itself ”contacts”, the URL could be something like *http://example.com/contacts*. To create a contact POST is called on *http://example.com/contacts*, to edit contact with id 1, PUT *http://example.com/contact/1*, to destroy contact number 1, DELETE *http://example.com/contact/1*. HTTP and resource identifiers together define already the interactions with the Restful web service. The content type is free to choose; the most common content type for apps is JSON, but also others like XML or YAML can be used.
In contrast to other web services like SOAP it is not possible to understand what content to expect, and what resources can be called. This means that good documentation is necessary; as the clients are tightly coupled with the response of the server. Making changes is difficult because of this tight coupling, if the REST services changes, and clients are not updated at the same time, they break without the proper use of versions[28].

### 2.4.2 JSON

Most of the time the content type for REST services is represented in a format, called JavaScript Object Notation (JSON). JSON data can represent models in an abstract way using definitions of text, numbers, dictionary and lists. It also is in comparison with Extensible Markup Language (XML) or Simple Object Access Protocol (SOAP) envelopes, smaller and lightweight, which are two important properties for limited bandwidth on mobile phones[29].
The developer needs to define the same abstract representation inside an app, parsing the JSON files makes the app able to represent the abstract definitions on the screen. To achieve this a developer needs to specify which parts of the models are represented on which part of the screen, and he also defines whether the user is able to interact with these. An app is built

out of multiple of these definitions.
Because the JSON and the app is tight so closely, both the server and client need to be aware of the data they receive and accept. This also means that the developed app always needs to be aware of the content of the JSON files, to be able to represent and understand the requested content. Usually the app is not able to dynamically adapt changes without also changing the code and so the binary.

### 2.4.3 System overview

In the most basic setup an app on a phone communicates with an server to gather content (see Figure 2.5). Usually Restful APIs make use of HTTP request and responses. One server is able to serve the data to multiple mobile phones running the particular app.

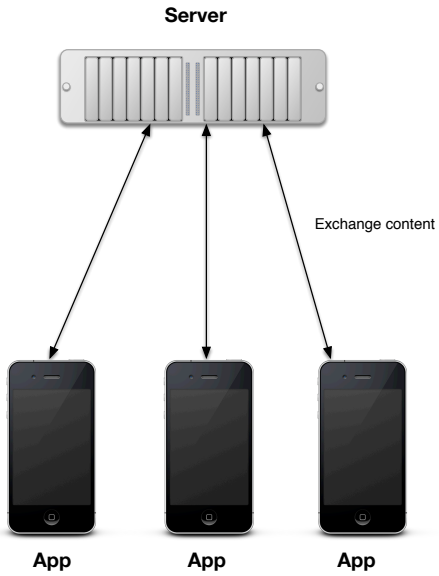


Figure 2.5: System overview

The following subsystem can be identified in the system overview:

- Server, which serves content for the app e.g. JSON files and images.
- App, runs on a mobile phone and is capable of parsing JSON content into objects to represent content on a screen.

## 2.5 Updating

With the exception of the web approach, every approach needs to have a binary to be executed on the phone. This binary can be distributed via the stores of each platform. Any change in code requires a new binary build. To be able to reach all existing users the build needs to be uploaded to the store. The new build is shown as an update in the stores, and the existing users

are able to download the new build, which updates the app.

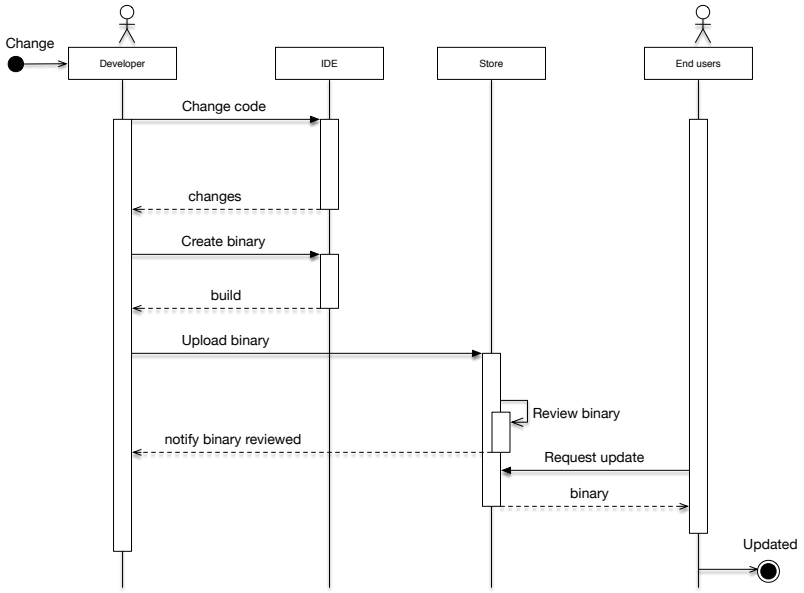


Figure 2.6: Steps involved in updating an app.

As seen in the sequence diagram (see Figure 2.6) binary update involves multiple steps before the update appears in the store[1]. First the developer gets a change, this change needs to be replicated in the code, the developer needs to create a new binary, the developer needs to upload the binary to the store, the store may needs to review the binary, the binary is processed by the store and available to the end user. When the end user is connected and allows updating the app or updates the app manually the version will be downloaded to the device. The process of updating an app in this way is quite time-consuming as involves many steps for the developers. Additionally, the update does not reach all users at the same time, since each user is able to control when he wants to update an app. In this way, the changes practically reach users later because of all the necessary steps after changing the code[23].

# Chapter 3

# Identifying requirements

This chapter describes the requirements the newly developed framework must fulfil. These requirements are gathered by identifying the needs of developing and updating an app. Existing requirements and improvements are differentiated, all of which need to work with the newly developed framework. The purpose of this chapter is to specify the requirements of the solution.

## 3.1 Prioritisation

The requirements are defined using the MoSCoW technique. The technique helps to prioritise requirements; the word MoSCoW stands for Must, Should, Could and Won't. The reasons for MoSCoW are that the requirements are prioritised in an explicit and easy way to communicate than for example High, Very High, Low priorities[30]. Must stands for requirements which guarantee if a project meets the goals, shoulds are trivial but not vital, could are nice to haves, and wont's are not implemented at all so to say out of focus.

## 3.2 Overview

This section gives an overview of the stakeholders of the system which are gathered from an perspective of the process of developing and distributing an app with current approaches. The use case model (see Figure 3.1) shows the system and users that are important in this process.

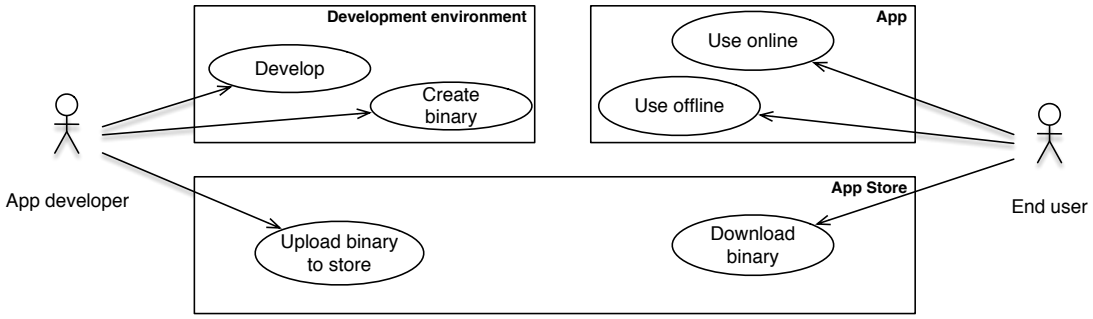


Figure 3.1: Use case model

### 3.2.1 Stakeholders

- End users - install, update an app and using the app online as offline.
- App developers - create and maintain the apps using the tools the platform provides.

### 3.2.2 Systems

- Development environment - the tools are used by the app developer to develop the app and create a binary.
- App - the app itself, the user is able to interact with the functionality that is developed by the developer.
- App Store - the app store is used to distribute the binaries to the end user. The end users are able to download these binaries as apps. Also the updates are binaries and uploaded and accessible via the store.

## 3.3 Requirements

This section describes the requirements of the system. These are divided into two sections: the first discusses the requirements that already exist in current the current native development approach, while the second describes the additional requirements, that should be possible in the future.

### 3.3.1 Existing requirements

**R-1: End users must be able to download and install the app via the stores.**
The end users of the resulting app should be able to download and install the app via the stores of the users mobile phone. This means that the developer creates a binary and submits it to the store of a specific platform.

**R-2: End users should be able to use the app online and offline.**
The app should persist the user interface between launches independently if the app is connected to Internet; this allows the end user to use the app also without an active

connection. Interactions that do not require an active network connection should work just fine.

**R-3: End users should have native user experience.**
The app should present the user interface using native interface elements, this provides the user a native feeling of the app with direct interaction. All interface elements should respond directly in a similar manner as a native implementation.

### 3.3.2 Additional requirements

**R-4: The app developer must be able to make changes to the app without creating a new binary.**
The approach being developed needs to allow change the functionality and layout of the app on the fly. This means that if a developer changes code on a central location like a server, it will automatically reflect all the changes to the clients. This principle is also used in web development where new changes on the website are also visible to all end users.

**R-5: When connected to the Internet, end users always could have access to the latest changes.**
When the app is connected to the Internet it is possible to connect to external services, to validate if it is running the latest version, and downloads a new version if possible. This means that the users is always using the latest version possible.

# Chapter 4

# Design architecture

This chapter describes the system architecture. The architecture is presented in multiple parts; first there is a discussion about current implementations, current and new system architectures, solutions compared. The other chapters present the new architecture with its class diagram and system flows.

## 4.1 Discussion

As the remote MVC pattern already seems to achieve the main objective (see Section 1.2), as it allows developers to change the app without a need to update the binary. In this pattern not only the models are represented in an abstract way on the server, but also the views. The relative small controller inside the binary communicates the user interaction with the server to know which view to display in the next step.
Nevertheless, the solution has some drawbacks, as it does not work offline because it requires an active connection. For mobile apps, this is not a good solution for building dynamic apps. It is not scalable, since the server needs to maintain open network connections for every client who connects; this is very expensive in terms of the number of clients per server. Apps are mainly used in circumstances with poor network coverage, which also makes it difficult on the client side to maintain an open connection.
Apps implementing the remote MVC pattern can present the user interfaces across all devices and remotely synchronise them using server side technology. The approach is centralised because the server is maintaining all the events of the user interface. The user interface must be updated with the response from a network which requires dealing with the mobile app, this does not allow to distribute the app on millions of devices as an open connection needs to be maintained.
A solution for this would be to decentralise the user interface events and models, views, controllers from the server. In this way, the clients will only interact with the server to check if there is a new version of a particular view, controller or model available. This goal can be accomplished by using comparing version numbers or calculate hashes over the previously downloaded files[31].
To gather new content a connection is always required. But the connection is not required to be persistent and allow offline usage without an connection. In the remote MVC pattern the app is only usable when there is an active connection to the server. To allow offline usage of the app it is required to have all the necessary files on the device. A possible solution for this is to

download all files and allow to load the complete app locally.
In the remote MVC pattern, not everything is implemented on the client side but, instead on the server. To make decentralisation possible and offline usability possible, it is necessary to move the interaction model to the client, and download files only if there is a change.

### 4.1.1 Conclusion

A MVC architecture can be replicated in an abstract way. This means that the models, views and controllers are defined on the server, but everything runs inside the client. In this way, it is possible to distribute and use the app without an active Internet connection. This solution does not require an active Internet connection to know which view it needs to display, but executes the user interactions locally.
The proposed system architecture can be seen in figure 4.1. The figure shows the models, views and controllers, all fetched from the server and executed on the client. The figure shows the JSON files on the server, they act as abstract MVC representations, that allow the app to generate instances that act as concrete counterparts, which allow a runtime MVC representation.
The solution could work in the same way as current objects are exchanged on the server using Restful services and JSON as content type (see Section 2.4).

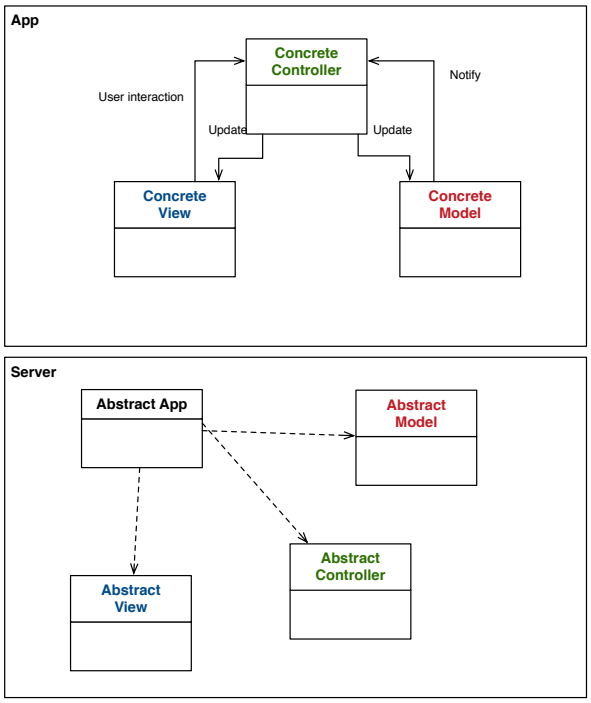


Figure 4.1: Proposed architecture of the solution.

## 4.2 Napton architecture

The newly proposed framework is called Native Adaptive Protocol Translated Object Notation (Napton). This name has been chosen because of the model definitions that are in JSON, which is being translated to the native approach to develop apps.
In the end the framework needs to generate native software components. Generative software usually use metamodels which allow to define an abstract models, that are parsed and converted to concrete models. In existing app development approaches generative approaches are used already, by parsing abstract JSON models into concrete models. Because the technology has already shown that it is widely implemented on mobile devices, it should be possible to be used for generating not only models into runtime objects but other definitions as well, like controllers and views. In perfect harmony, all platforms would define the same pattern as one like in figure 2.2. However, every platform is using its own variation of a MVC pattern. This makes it difficult to define one common MVC structure for all platforms[26]. Because the variations still inherit the same base model, the Napton architecture is based on a form of a basic MVC pattern. The MVC pattern allows to define the abstract models, views and controllers on the server.
Usually the developer needs to specify which classes are used to parse the abstract JSON models into their concrete object counterparts. In Napton reflective programming is used to accomplish this in combination with a decoration pattern[32]. The decorator ensures that the framework is able to generate the platform-specific MVC pattern and native interface objects. This allows the platform specific MVC pattern to delegate the communication of the user interaction to the abstract MVC pattern.
After fetching these abstract models, views and controllers, the framework is able to generate the platform specific concrete counterparts. The JSON files act like metamodels for the Napton generator.
The class diagram (see Appendix A.1) shows the classes that are required on the server and the classes then are implemented in the framework on the app side. Basically every model, view or controller which is defined in a JSON file is an abstraction stored on the server. When the framework fetches these abstract objects it parses them and generates concrete counterparts. The concrete counterparts are able to execute the main flow of the app to start presenting views on the screen. These are the *NaptonObject* subclasses. A NaptonObject instance is able to generate its own native instance. This is possible using the *kind* property which is defined inside every NaptonObject. The *kind* property defines the class which needs to be initialised as native instance. These classes are implemented inside the Napton framework and adopt the NaptonInterfaces. To clarify the generative process, it can be represented inside a metamodel architecture. The metamodel architecture of Napton is shown in figure 4.2. First we see the first abstraction layer(M3) in the case of Napton; the JSON language. The next abstraction layer(M2) are the abstract models defined in JSON files on the server. After the app parsed the(M2) JSON files into abstract NaptonObject instances(M1) it is able to generate the native instances(M0). The native instances represent the runtime instances which represent the app, which is the purpose of the framework.

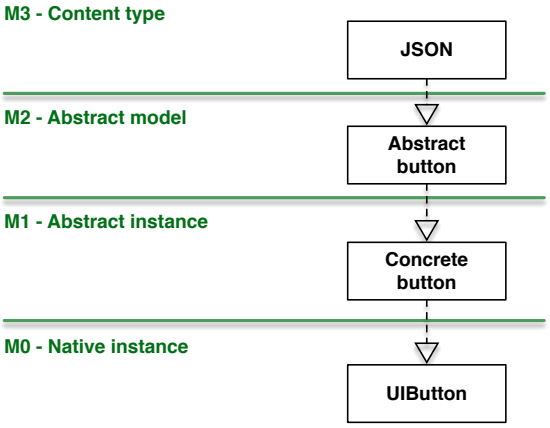


Figure 4.2: Napton metamodel: JSON(M3) to generated instances(M0).

**Definition**

The models, views and controllers are specified on the server side. Napton does not need a specific developed server approach, but needs to be able to request files in a RESTful way, and currently only supports JSON as a content type. In the previous section the class diagram and metamodel of Napton are explained. As seen in figure 4.2, the abstract models are defined in JSON, and as seen in the class diagram these are defined as abstract models on the server. Napton needs specific files which are stored on the server, see the manual in Appendix B to get started. The following files ar necesssary for Napton to create the app:

- *app.json* This file contains information about the abstract app, such as the version number, root_controller and defines where the other resources are located, (model/view/controllers) All resources inside the app are cached and offline available.
- *model.json* The models can be defined like before in JSON. This means that it is fully compatible with existing JSON resources.
- *view.json* The view defines the layout of the elements, subviews and the bindings between the models.
- *controller.json* The controller defines the actions that it can respond to and loads missing resources if necessary. The actions are able to execute functions on controllers.

### 4.2.1 Fetch remote files

Every time the app starts, Napton checks if any new remote files are available. This is checked by means of a version number inside the *app.json*. If the version number differs, new remote files for models, views and controllers are fetched. The most logical approach would be to organise this in a RESTful way, where the models, controllers and views have their own resource identifier */model*, */view*, */controller*[27]. The fetched files are stored for offline in the document storage of the device to allow offline usage. If the version of the app is newer, the framework will replaces previously fetched files.
Napton first tries to connect to the specified server. If this fails, it will use the fetched files from

the previous state. In this way, it is possible to run the app offline as well. Furthermore, if an Internet connection is available, Napton tries to fetch the new models, views and controllers to update the app.

### 4.2.2 Generate models on device

After the remote models are fetched, Napton is able to generate concrete models on the client. These concrete models represent a MVC structure, which is defined inside the fetched files. The concrete models act like a decorator pattern, to allow communication between them above the concrete layer which are the generated objects.
The MVC pattern defined inside the Napton framework is interpreted and able to run as runtime instances. If the app needs to draw or interact with the native controllers provided by the system framework, the Napton instance is able to generate the specific instance using reflective programming. The only requirements for the native instance is that they implement the Napton interface. For the Napton framework, all classes of the platforms library need to implement the Napton interfaces. The reflective technique will call these interfaces to be able to be able to generate the preferred instance.

### 4.2.3 Execute functions of the controllers

The functions that are defined on controllers are predefined functions. This means that the Napton framework is limited to a feature set which is defined by the Napton framework. The functions that are being called need to be defined inside the generated instances.
Like the model generation, the functions are executed on the generated instances of the concrete Napton instances using reflective programming. This makes it possible to access all the native functions provided by the frameworks, which is often a limitation of other cross-platform solutions[9].

## 4.3 System flows

The following sections describes the flow of Napton, based on the architecture. The most important flow is the generator which generates runtime objects from the JSON files objects. The framework first downloads the JSON files, parses them, generates an abstract MVC model and generates objects native interface objects to represent on the screen.
The first section describes this initialisation flow, the second the generation of runtime objects, the third section how a runtime instance is able to gathered arguments from the Napton MVC instances, the last section describes how functions are executed on the Napton MVC instances. These flows together represent the core functionality of the framework.

### 4.3.1 Initialising the app

During the starting up phase of the app, the app checks several things (see Figure 4.3). First it fetches, if possible, the *app.json* from the server, in order to synchronise the previously fetched files with the new remote files.
Initially, the Napton framework requests the *app.json* to discover the resources available on the server. If the version is not equal to the local version, each resource is requested, this can be

a model, view or controller. If there does not exist a local copy or it is outdated, the resource is downloaded and written to a file. The *app.json* also defines a *rootController*, this property is the initial controller where the app starts with. Which is the one which will be returned and its view displayed to the end user.

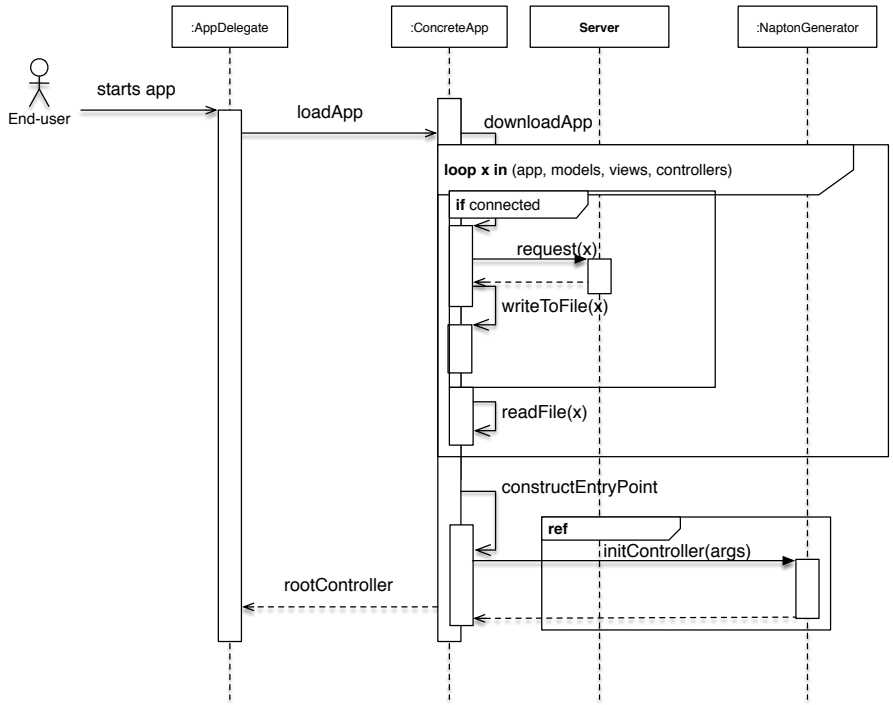


Figure 4.3: Flow initialising the app.

### 4.3.2 Generate concrete instances of the abstract objects

The parser inside Napton is able to convert the fetched files into concrete models. This means that the defined abstract models(M2) are translated into concrete models(M1) that are able to execute and handle code inside the app (see Figure 4.2). Every concrete model is initiated by the kind which is defined inside the abstract JSON model; in this way, the parser knows where to store and how to generate the concrete model.
Every concrete Napton object(M1) has its own generated instance(M0). The generated instance represents the native object of the library of the platform. For example, an Abstract Button is converted into an Concrete Button, which has a generated instance that is a native UIButton on iOS.
The flow in figure 4.4 show how a generated instance(M0) is created, first an object calls the *NaptonGenerator* first, the generator tries to allocate an object given having the given class name. It will call *initWithNaptonObject*, which requires arguments, to know exactly how the instance should look and behave. The parent argument can be an NaptonObject and is needed to execute functions and gather the required function arguments after the object is generated, which is described in the following two sections.

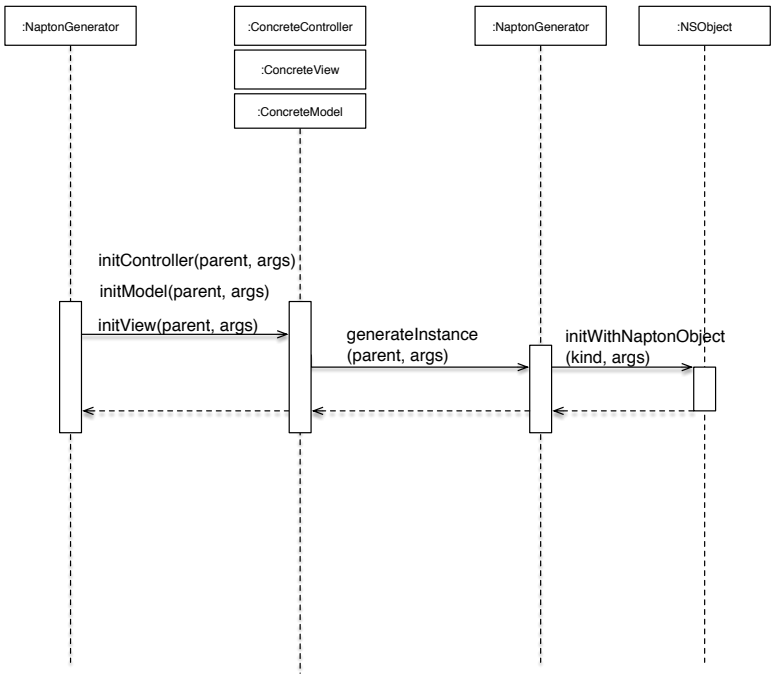


Figure 4.4: Flow generating concrete objects.

### 4.3.3 Getting arguments of the concrete objects

Every function that is initiated needs specific arguments to define the behaviour of the particular function.
Most generic structures are called dictionaries or hashmaps. They allow to store an object to for a given key. The dictionary is used to hold the arguments of the previous metamodel. If the generated instances of the platform request an argument, they can gather the argument through the dictionary of the Napton object, which is one metalevel higher. The generated instance calls every time it needs an argument, the getArg function with a key for the value wants to get (see Figure 4.5). To accomplish this every generated instance has access to *NaptonObject* a parent to call and execute the function of a higher metalevel. The *getArg* returns the first match which can be found.

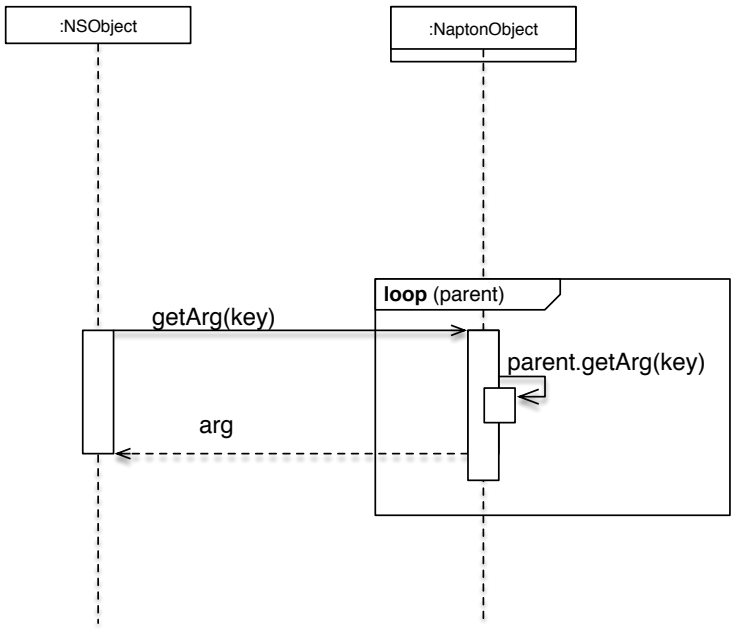


Figure 4.5: Flow getting arguments of the concrete objects.

### 4.3.4 Execute functions of the concrete objects

Executing functions works almost like getting arguments. When a button is pushed, for example, it first sends the user interaction to the concrete Napton view, which checks if it is able to handle the function or passes it to the concrete Napton controller, which is most likely able to respond to it. Every concrete NaptonObject(M1) is able to respond to functions, the Napton function which responds at first defines the scope. If the Napton object is able to respond, it attempts to execute the function on the generated instance. The generated instance obtains the function name and the arguments. The function is then executed using reflective programming techniques as can be seen in figure 4.6.

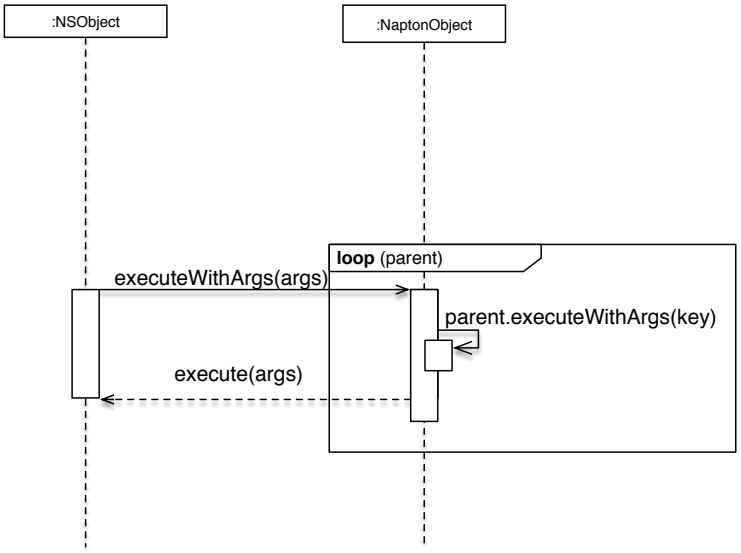


Figure 4.6: Flow executing function of the concrete objects.

# Chapter 5

# Prototype

In this chapter a sample implementation of the web and native approach and the Napton approach described, based on a usecase, and a change scenario.

## 5.1 Usecase

It is necessary to validate whether the framework works as expected, and compare and validate against other approaches. The following case and the prototypes have been developed for this purpose.
Every year the organisation 'Kick-In' of the University Twente is organises an introduction week for the new students. They want to provide an app to the students so that the students can easily see where they need to go. The app should list the events that are organised for students to participate. If a user presses on an list item, he should be able to see the details. Because the organisation is not sure which user interface elements they want to display for each event, they would like to keep this part flexible. For some events, the organisation may want to add pictures after or during the event, or require a form for students to sign-up or win a price. The app should contain at least the two main views: the list view and a view to display the details of a selected event.

## 5.2 Implementation

Based on the use case, an architecture is proposed and implemented for different approaches. The implementations allow the Napton approach to be validated.

### 5.2.1 Architecture

A prototype of the architecture for this use case is displayed in figure 5.1. The architecture shows the MVC pattern, which is implemented by the chosen approaches (web/native/napton). The basic model called *Event* can be seen, which is the abstraction of a single event. Also depicted are the two main controllers, the *EventListControllers* basically connects the view with the list of event models. If a user presses on an event the controller will display an *EventDetailController*, which also has a view and displays all the data for the event.

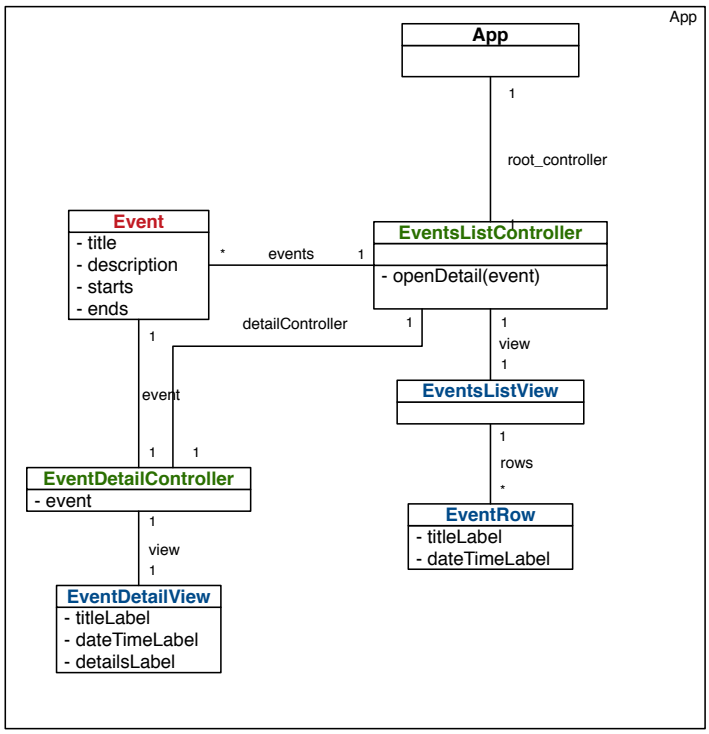


Figure 5.1: Architecture prototype

### 5.2.2 Web approach

The web approach is implemented using the jQuery mobile[14] framework. This fetches the list of events in JavaScript and displays the list using the interface components of the framework (see Figure 5.2), these are defined using HTML and CSS.

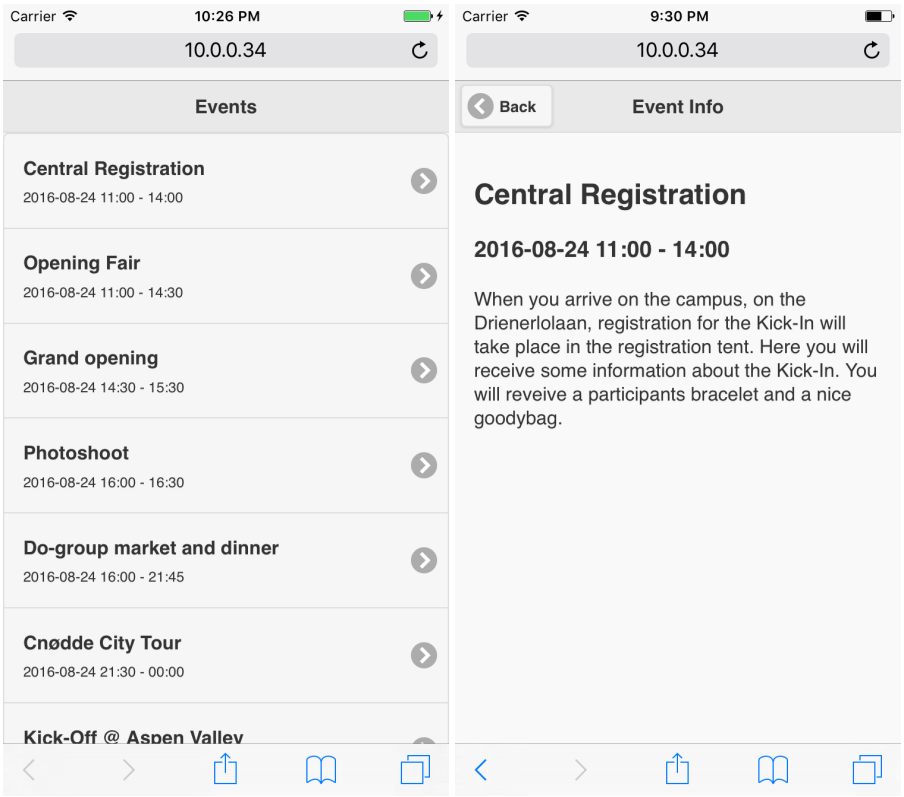


Figure 5.2: Web prototype

### 5.2.3 Native approach

The native approach is implemented using the Cococa Touch[11] framework, provided by the iOS platform. The app also fetches the events, but displays them using the native user interface components delivered by the Cocoa Touch framework. The native user interface elements look and behave like the rest of the OS (see Figure 5.3, compared to the web approach Figure 5.2).

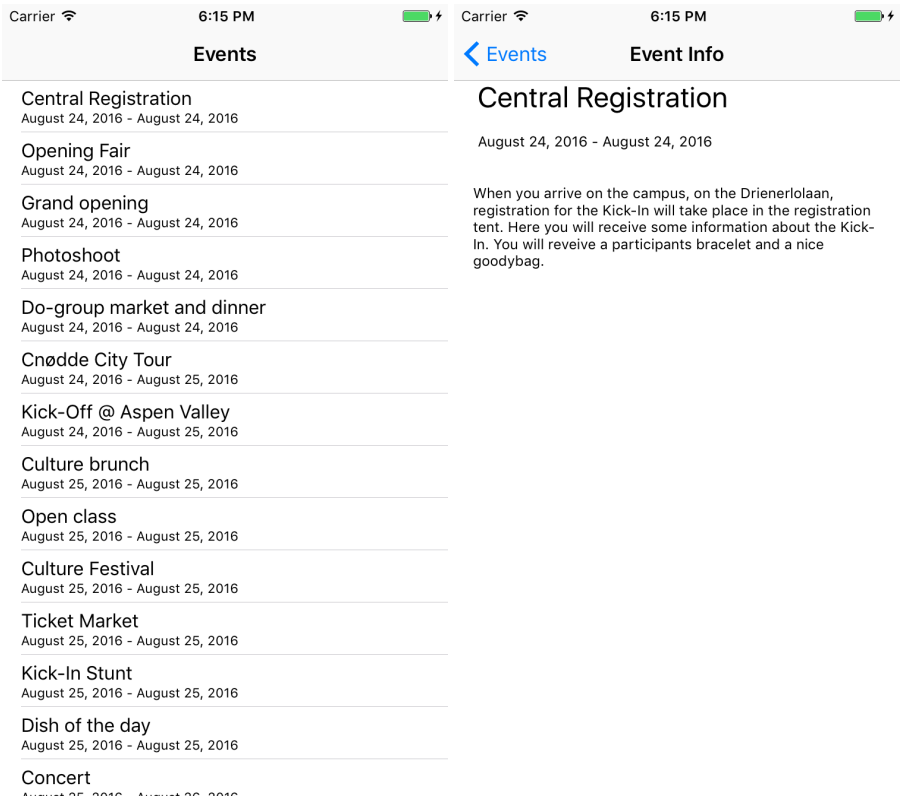


Figure 5.3: Native prototype

### 5.2.4 Napton approach

The Napton approach is mainly written on the server side and requires just an implementation of the Napton framework inside the app binary. The Napton framework is able to use the underlying Cocoa Touch framework to facilitate the creation of native components. By using native user interface elements the experience and look is the same as using a native approach (see Figure 5.4, compared to the similar native approach in Figure 5.3).

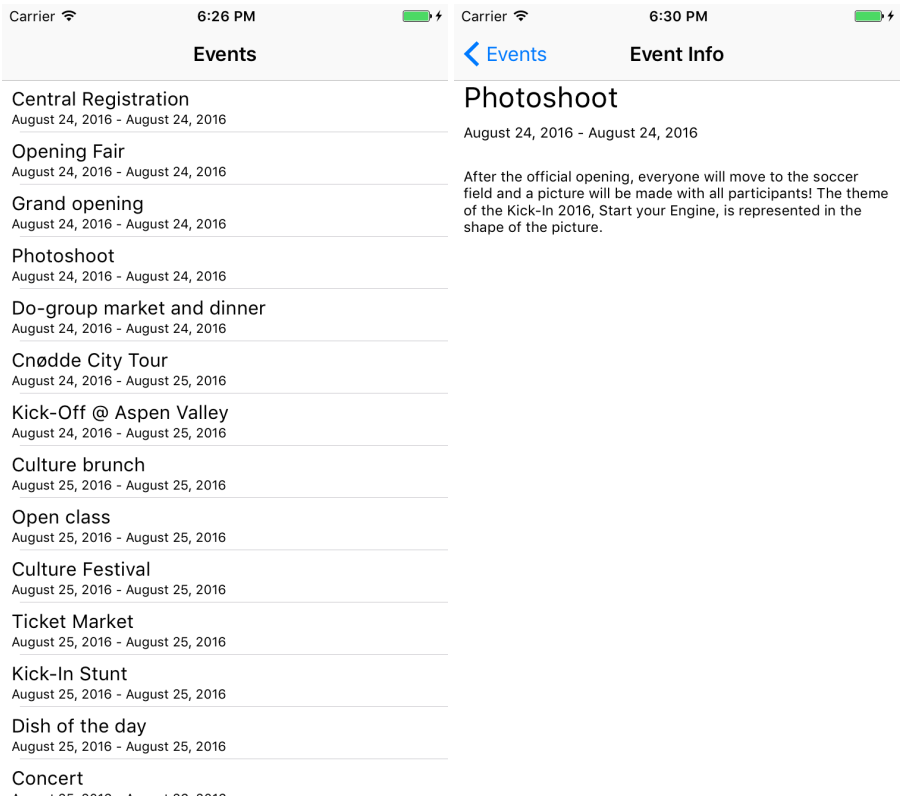


Figure 5.4: Napton prototype

### 5.2.5 Conclusion

For this particular project, the web approach fits very well. This is because it is highly flexible, and permits the changes as described, although it is not offline available and does not provide a great user experience, causing the interface is built using less responsive web technologies and not using the native interface components the platform provides.
The native approach is not a perfect fit either, because it does not allow quick updates; however, it provides suitable performance and allows offline usage.
Because the Napton approach is implemented on the server side, it allows the app to be changed without the need for an update. By using native interface components it also provides a better user interface experience compared to the web.

## 5.3 App update - change scenario

The day before the introduction week of the new students at the University of Twente starts, the “Kick-In” committee has a problem. They have completely forgotten that their photographer had taken pictures for every event to display next to the event details in the app.
A new property must be added to the server, and all clients need to display the image in a new view. The architecture of the client prototype needs to reflect these changes as well, this is done

by adding an *image_url* and an *ImageView* to the architecture (see Figure 5.5). All prototypes have been changed to show the image in the detail screen (see Figure 5.6).

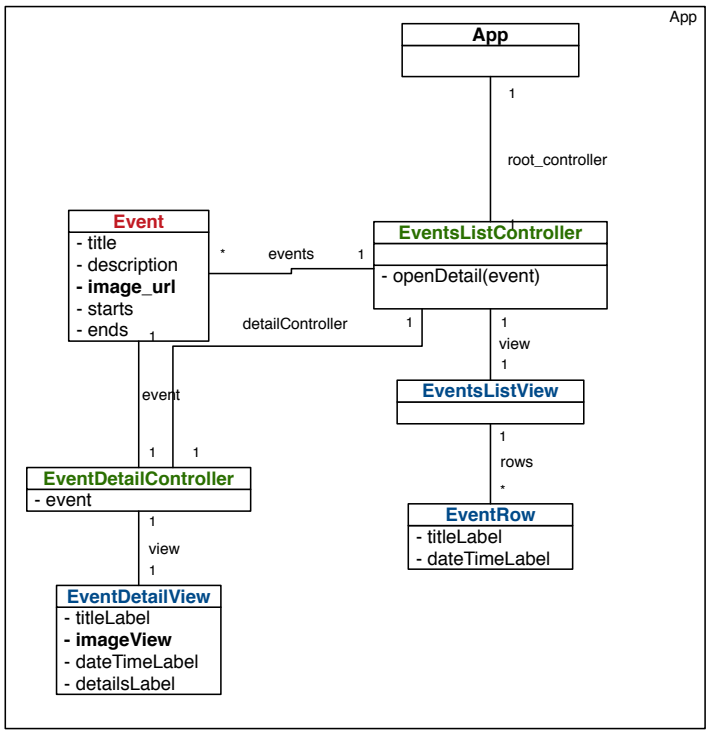


Figure 5.5: Prototype architecture updated according to the change scenario.

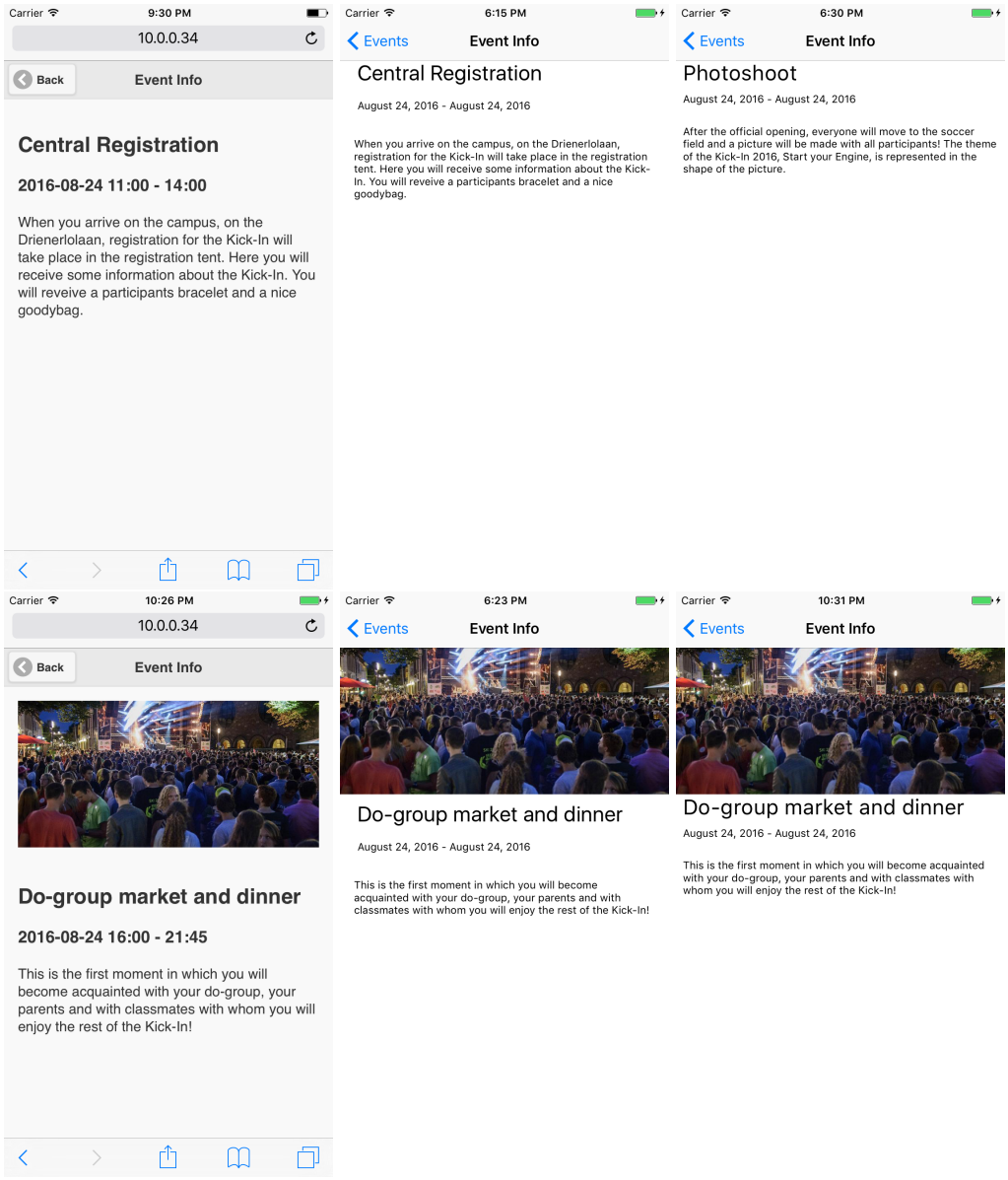


Figure 5.6: Implementation of app before and after the update (web/native/Napton).

### 5.3.1 Conclusion

The changes in code were particularly easier for Napton and the web approach as just changes on the server needed to be made. On the native approach not only the code on the server, but also on the client needs to be changed and requires a new binary to be uploaded to the App Store.

# Chapter 6

# Validation

This chapter validates whether the proposed framework fulfils the requirements stated in chapter 3, and assesses how our framework performs compared to other frameworks. This is accomplished by measuring the performance of the proposed framework against the competition. The requirements are validated using quantitative measurements.
Times and lines of code are used in order to measure performance and complexity. Validation is carried out using a sample application. Included in the measurement process are one web app, one native app and one app which uses the Napton framework.
The apps are the ones developed and described in chapter 5. All apps use the same back-end, listing all the events of the Kick-In introduction week.

## 6.1 Validating requirements

This section validates whether the product - the framework in this case - is able to meet the requirements defined in section 3.3. The requirements validated based on the prototype discussed in chapter 5, and by applying the change of section 5.3 above.

### 6.1.1 Existing requirements

**R-1: End users must be able to download and install the app via the stores.**
The first version of the app needs be created inside the development environment of the platform. The Napton framework must be added to the libraries and initiated. After these steps, the binary can be built and uploaded to the store. The implementation of the app is specified on the server side.

**R-2: End users should be able to use the app online and offline.**
The app is able to work offline because the required files are stored locally. Every time the app starts and is not connected to the Internet and therefore cannot download the remote files, it falls back on the local files. In this way, the end users are able to use the app offline.

**R-3: End users should have native user experience.**
To measure the user experience very exactly multiple methods have been considered, as automated user interface testing, measuring time on launch, or response of buttons. But it is still very difficult to measure the user experience in these ways, as it is influenced

by many factors like: processor speed, networking speed, drawing cycles, memory load, threading, frame rate etc.
To make a fair comparison between the approaches; as they all use different technologies to draw user interface elements and make use of different techniques for threading and drawing the user interface. Because of this, the performance has been measured in different ways. A very important measurement of changes in code is the number of lines in code (LOC). Another very important metric for mobile devices is the size of the app.

**Changes in code**
The number of lines of code (LOC) is often used to measure and predict the development effort and size of a software project. This metric gives a rough indication of the degree of effort put forth by one developer[33][34]. To compare the changes in code of the Napton framework against existing approaches, the number of lines before and after the update were measured.
To measure the number of lines of code, a tool called Cloc[35] was used. Cloc is able to count lines of code of different programming languages in an equal way by ignoring comments and white spaces. After the changes were implemented, the count was repeated in order to see how many lines were added.

| Approach | initial LOC | updated LOC | difference |
|---|---|---|---|
| Native | 382 | 390 | 8 |
| Web | 71 | 82 | 11 |
| Napton | 120 | 123 | 3 |

Table 6.1: Comparing initial app LOC with updated app LOC.

**Download size**
The download size is defined as a measurement which includes all files that are necessarily to run the app, in other words, all the instructions to draw and interact with the user interface. The models requested during runtime on the server are not counted because they are requested on all implemented approaches.
For the initial download size, the elements considered are as follows:

- **Native** The binary size is counted as initial download size.
- **Web** The HTML, JavaScript and CSS files are counted for the web approach.
- **Napton** For Napton the binary size plus the required remote files are counted.

For the updated size:

- **Native** the binary size is used again for native, because users need to download the update via the store.
- **Web** The HTML, JavaScript and CSS files are used again for web.
- **Napton** Only the remote files are counted for Napton, since a new binary is not necessary.

To count the number of bytes of a file, the Unix utility WC[36] was used. WC is able to count the number of bytes of the defined files, and the same utility was used in order to have a fair comparison.

| Approach | Initial size in bytes | Updated size in bytes | Growth |
|---|---|---|---|
| Native | 241.139 | 241.610 | 0.19% |
| Web | 2775 | 3071 | 10.66% |
| Napton | 274.151 | 2513 | -0.92% |

Table 6.2: Comparing initial app download size with updated app download size.

### 6.1.2 Additional requirements

**R-4: The app developer must be able to make changes to the app without creating a new binary.**
Because the whole application is specified inside the server, the developer is able to modify the server-side files and make changes inside the app whenever a view or functionality needs to change. This means that the app developer does not need to carry out all the steps for uploading a binary (see Section 2.5).
A problem like the app change scenario (see Section 5.3) is easy to implement because it only involves changing files on the server. Next to adding the *image_url* JSON property to the events to know which image belongs to the event, just a new ImageView needs to be specified inside the JSON which describes the *EventDetailView*. No other changes are needed, to accomplish the same result as the other approaches, which both need changes on the client side architecture as well.

**R-5: When connected to the Internet, end users always could have access to the latest changes.**
Each time it is connected to the Internet, the app checks if there are any changes to the remote files. If there are any files, the app is always able to adapt its interface and provide the newest features. Because the app is always up-to-date, all end users have the same version and see the same functionality.

## 6.2 Conclusion

The update of the app (see Section 5.3) demonstrates that the framework fulfils the requirements and is able to update the app dynamically. This means that some of the steps to update an app are no longer necessary, and this allows very quick changes to be made for all end users.
The performance measurements conducted show results for Napton that are better than the competitors in terms of less number of lines of code added for the same change, and the smaller download size after a change.
Less changes in the number of lines means that a new feature can be implemented faster, since the developer is able to write more lines in less time, given the complexity of every line is the same[34]. Smaller download sizes to update an app uses less data, and provides a faster overall performance for the end user since less data needs to be processed[37].

# Chapter 7

# Reflection

This chapter gives the final remarks of this research. The chapter gives the general conclusions, answers to the research questions, limitations and recommendations for future work.

## 7.1 General conclusions

In this thesis, a new approach to developing apps was developed. This approach is capable of dynamically updating the whole app without downloading a new binary. This is accomplished by not only parsing models but the whole architecture of an app, which allows the definition of an app on the server. The new approach is combining parts of existing approaches, but does not offer the same functionality, they are not capable of being used offline, Instead, an active server connection was always required in order to interact with the app. Napton combines the benefits of using the web approach with the native approach, by using native user interface elements, which is also used in the generated approach. This makes Napton living between the native, generated and web approach (see Figure 7.1).

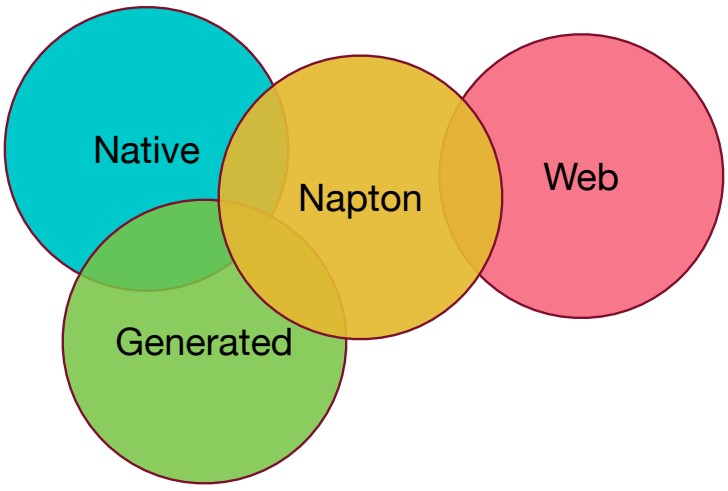


Figure 7.1: Napton approach, sits between the native, generated and web approach.

The Napton framework is my contribution to give another insight to app development. Napton is able to generate an app based on JSON files on the server. This allows developers to develop an app without the need to build and compile code, but rather by writing files server side, like web development. The composition contains models, views and controllers that reflect the MVC development pattern, which is a common composition pattern used to define the user interface and interactions of apps. The framework uses reflective techniques to generated native instances. This technique allows objects to be generated that interact with the arguments and actions of a higher level MVC composition. The remote files are defined in JSON, thus enabling compatibility with all RESTful server implementations, which are already widely used for mobile app development.

Different app prototypes were developed, including an app built using the web approach, another app using the native approach and one using the Napton approach. A real-world case study has been implemented to contain the same functionality on all prototypes. This allows to evaluate the developed framework against other approaches. Using these prototypes, a comprehensive comparison was carried out. The conclusion of this comparison shows that the web approach is the most flexible, while the native approach has the best user experience because of its use of native interface elements. The newly developed Napton approach has both benefits, the use of native interface elements and the flexibility of the web approach by changing the app on the server instead of making changes locally and deploy a new binary. To compare the newly developed framework against the other approaches, a change scenario was done which includes an app update for the prototype apps.

The initial idea of measuring performance of user experience was more difficult than expected. Processor speed, networking speed, drawing cycles, memory load, threading, frame rate are factors that have big impacts on measuring completely different development approaches. Because of this, the performance has been measured in different ways: number of lines of code, and the download sizes of the different prototype apps.
These performance measurements show a smaller footprint of the Napton framework. The prototype using the Napton framework does have a much smaller size in terms of number of lines of code for the same change, as well as the download size after a change. This means that features can be implemented even faster, and an update requires the use of less data for users since they do not need to download the whole app. In addition, the whole process of updating an app is different, as the steps typically needed (building, waiting, uploading, waiting, store review, releasing, updating from the end users) are not necessary anymore. Furthermore, if the end users have the latest remote files, they always have the last version of the app with the newest features available.

Requirements were defined in order to specify the features which the developed framework should fulfil.

## 7.2 Answers to the research questions

The research questions defined for this thesis are described below:

**RQ1. What are possible approaches to decentralise the user interaction of the remote MVC pattern?**
In chapter 2, possible solutions were analysed for decentralising the user interaction. In the design architecture of chapter 4, we present a possible solution. The solution does

not require a server to synchronise and interact with the app, it allows the usage in a decentralised way, where only the user is in control of the app.

**RQ2. What are possible approaches to make the remote MVC pattern work offline?**
In chapter 2, the remote MVC pattern is described, which shows the lack of allow offline usage when the app is being used. The design of the Napton framework in chapter 4 takes into account offline usage. The framework is able to parse local or remote JSON files. Every time the app fetches new files it stores the files locally for offline usage.

**RQ3. How can a framework be developed that allows the app to be changed dynamically when there is a connection, but still supports offline usage?**
The architecture of the framework is described in chapter 4. The architecture is designed to allow offline usage and change the user interface of the app upon connection. The definition of an MVC pattern in JSON enables clients to generate an app on-demand. The JSON files which are fetched from the server, stored locally and refreshed upon a change, allow offline usage without connection in a decentralised way. If there is a connection, the app updates the models, views and controllers and stores the latest version.

## 7.3 Limitations

The Napton approach works good for content based apps. This means apps that provide rich content to the user, like serving news, images, location based info, magazines, blogs or webshops. The apps could also provide some minimal interactions with Restful services, to submit data and request external resources.
The main limitation of the framework is the number of classes that are available inside the framework. The more classes support the NaptonInterface the more functionality is available. By broader adoption and implementation of the NaptonInterface(s) and the principles used by the Napton framework, it could be possible to create utility based apps, which require hardware access, like a compass app, navigation app, or apps communicating via Bluetooth. Also, apps that require more specific requirements as streaming live data, editing media, interacting with local data, requesting data from other protocols as JSON and maintaining complex databases are not feasible to build right now. Although the framework allows easy extension of the framework by implementing the NaptonInterface for custom classes, not all apps are easier and faster to build and dynamically changeable this way, as still specific classes cannot be changed using the JSON arguments Napton provides, but also need to update deeper class based logic.

## 7.4 Future work

The following problems are still unresolved, and could offer new opportunities for future work.

**Cross-platform support**
The implemented solution has theoretically cross-platform support, but there is currently no cross-platform prototype implemented at this moment, which proves the support. A similar framework could be implemented for platforms like Android and Windows Phone. To support the remote Napton files, a platform-specific framework needs to be developed, which supports reflective programming and be able to parse JSON. To do so, a developer just need to adopt the remotes files' MVC pattern to their own platform-specific MVC pattern.

**Translator for platforms**
The Napton framework defines all the concrete classes that can be used by the abstract classes defined in the remote files. Not all native classes are translated at this time. This means that not all library classes can be initiated. It is a very large task to maintain a framework like this, and it will lead to possible incompatibilities in deprecated frameworks of newer operating systems. To allow faster adoption of the Napton framework a translator application could help create the missing concrete classes automatically. Additionally, the translator could generate the documentation and align platform specific classes with equal implementations of other platforms.

**Business logic**
No real business logic can be defined in the current prototype. The apps that can be developed with the prototype at this point are content-driven and do not really need model modifications. Sending and receiving content is possible, however, because it is possible to pass arguments around and to execute network requests. Still, to displaying for example dates, a definition of business logic is necessary. In the current prototype a custom template language was defined which is able to perform basic string concatenations and date formatting. To allow modifications of properties, calculations, sorting, which allows broader definitions of functions in models, a language to define business logic is needed. There is already been worked on solutions to allow logic inside JSON called JsonLogic[38], which looks promising in defining logic without the need of a whole programming language.

**Dynamic runtime class injection**
The Napton framework only allows the use of abstract classes that are defined by the framework. New concrete classes are only accessible when contained inside the binary build, which allows the Napton framework to generate the native instances. Tools already exist to allow quick fixes of bugs, such as Rollout.io[39]. These tools use runtime code injections. A similar solution could allow the framework to be extended dynamically. This permits the extensions of the available classes inside the binary build, and makes it possible to define more abstract classes in the remote JSON files.

# Bibliography


[1] Apple. (). Replacing your app with a new version, [Online]. Available: https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/ReplacingYourAppWithANewVersion.html (visited on 05/20/2016).

[2] P. Gokhale and S. Singh, "Multi-platform strategies, approaches and challenges for developing mobile applications," in *Circuits, systems, communication and information technology applications (cscita), 2014 international conference on*, Apr. 2014, pp. 289–293. DOI: 10.1109/CSCITA.2014.6839274.

[3] P. Smutny, "Mobile development tools and cross-platform solutions," in *Carpathian control conference (iccc), 2012 13th international*, May 2012, pp. 653–656. DOI: 10.1109/CarpathianCC.2012.6228727.

[4] M. Nayebi, B. Adams, and G. Ruhe, "Release practices for mobile apps – what do users and developers think?" In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)*, vol. 1, Mar. 2016, pp. 552–562. DOI: 10.1109/SANER.2016.116.

[5] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *2015 ieee 23rd international requirements engineering conference (re)*, Aug. 2015, pp. 116–125. DOI: 10.1109/RE.2015.7320414.

[6] T. Ritterfeld, "Research topic on-demand app development," May 2016.

[7] C. P. R. Raj and S. B. Tolety, "A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach," in *India conference (indicon), 2012 annual ieee*, Dec. 2012, pp. 625–629. DOI: 10.1109/INDCON.2012.6420693.

[8] M. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *Empirical software engineering and measurement, 2013 acm / ieee international symposium on*, Oct. 2013, pp. 15–24. DOI: 10.1109/ESEM.2013.9.

[9] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th balkan conference in informatics*, ser. BCI '13, Thessaloniki, Greece: ACM, 2013, pp. 213–220, ISBN: 978-1-4503-1851-8. DOI: 10.1145/2490257.2490292. [Online]. Available: http://doi.acm.org/10.1145/2490257.2490292.

[10] A. Charland and B. Leroux, "Mobile application development: Web vs. native," *Commun. acm*, vol. 54, no. 5, pp. 49–53, May 2011, ISSN: 0001-0782. DOI: 10.1145/1941487.1941504. [Online]. Available: http://doi.acm.org/10.1145/1941487.1941504.

[11] Apple. (2000). Cocoa foundation, [Online]. Available: https://developer.apple.com/reference/foundation (visited on 01/06/2016).

[12] Android. (2008). Android application framework, [Online]. Available: https://developer.android.com/guide/index.html (visited on 01/06/2016).

[13] Microsoft. (2008). .net compact framework, [Online]. Available: https://msdn.microsoft.com/en-us/library/f44bbwa1(v=vs.90).aspx (visited on 01/06/2016).

[14] T. jQuery Foundation. (2016). Jquery mobile, [Online]. Available: https://jquerymobile.com (visited on 01/06/2016).

[15] S. Inc. (2016). Sencha touch, [Online]. Available: https://www.sencha.com/products/touch/ (visited on 01/06/2016).

[16] D. Kaneda. (2014). Jqt, [Online]. Available: http://jqtjs.com (visited on 01/06/2016).

[17] A. S. Inc. (2016). Adobe phonegab, [Online]. Available: http://phonegap.com (visited on 01/06/2016).

[18] A. Inc. (2016). Appcelerator, [Online]. Available: http://www.appcelerator.com (visited on 01/06/2016).

[19] M. Usman, M. Z. Iqbal, and M. U. Khan, "A model-driven approach to generate mobile applications for multiple platforms," in *Software engineering conference (apsec), 2014 21st asia-pacific*, vol. 1, Dec. 2014, pp. 111–118. DOI: 10.1109/APSEC.2014.26.

[20] T. A. Majchrzak, J. Ernsting, and H. Kuchen, "Achieving business practicability of model-driven cross-platform apps," *Open journal of information systems (ojis)*, vol. 2, no. 2, pp. 4–15, 2015, ISSN: 2198-9281. [Online]. Available: http://www.ronpub.com/publications/OJIS_2015v2i2n02_Majchrzak.pdf.

[21] P. Friese. (2013). Applaus, [Online]. Available: https://github.com/applause/applause (visited on 01/06/2016).

[22] S. Charkaoui, Z. Adraoui, and E. Benlahmar, "Cross-platform mobile development approaches," in *Information science and technology (cist), 2014 third ieee international colloquium in*, Oct. 2014, pp. 188–191. DOI: 10.1109/CIST.2014.7016616.

[23] I. Dalmasso, S. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," in *Wireless communications and mobile computing conference (iwcmc), 2013 9th international*, Jul. 2013, pp. 323–328. DOI: 10.1109/IWCMC.2013.6583580.

[24] J. Deacon, "Model-view-controller (mvc) architecture," *Online][citado em: 10 de maro de 2006.] http://www. jdl. co. uk/briefings/mvc. pdf*, 2009.

[25] A. Inc. (). Model-view-controller, [Online]. Available: https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html (visited on 10/21/2015).

[26] V. Giedrimas and S. Omanovic, "The impact of mobile architectures on component-based software engineering," in *Information, electronic and electrical engineering (aieee), 2015 ieee 3rd workshop on advances in*, Nov. 2015, pp. 1–6. DOI: 10.1109/AIEEE.2015.7367317.

[27] V. Stirbu, "A restful architecture for adaptive and multi-device application sharing," in *Proceedings of the first international workshop on restful design*, ser. WS-REST '10, Raleigh, North Carolina, USA: ACM, 2010, pp. 62–65, ISBN: 978-1-60558-959-6. DOI: 10.1145/1798354.1798388. [Online]. Available: http://doi.acm.org/10.1145/1798354.1798388.

[28] Oracle. (). What are restful web services? [Online]. Available: http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html (visited on 12/05/2016).

[29] E. T. Bray. (). The javascript object notation (json) data interchange format, [Online]. Available: https://tools.ietf.org/html/rfc7159 (visited on 11/07/2016).

[30] 2. A. B. C. Limited. (). Moscow prioritisation, [Online]. Available: https://www.agilebusiness.org/content/moscow-prioritisation (visited on 12/08/2016).

[31] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov, "Ontology versioning and change detection on the web," in *Knowledge engineering and knowledge management: Ontologies and the semantic web: 13th international conference, ekaw 2002 sigüenza, spain, october 1–4, 2002 proceedings*, A. Gómez-Pérez and V. R. Benjamins, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 197–212, ISBN: 978-3-540-45810-4. DOI: 10.1007/3-540-45810-7_20. [Online]. Available: http://dx.doi.org/10.1007/3-540-45810-7_20.

[32] R. S. Moreira, G. S. Blair, and E. Carrapatoso, "A reflective component-based and architecture aware framework to manage architecture composition," in *Distributed objects and applications, 2001. doa '01. proceedings. 3rd international symposium on*, 2001, pp. 187–196. DOI: 10.1109/DOA.2001.954084.

[33] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: A software science validation," *Ieee transactions on software engineering*, vol. SE-9, no. 6, pp. 639–648, Nov. 1983, ISSN: 0098-5589. DOI: 10.1109/TSE.1983.235271.

[34] J. D. Blackburn, G. D. Scudder, and L. N. Van Wassenhove, "Improving speed and productivity of software development: A global survey of software developers," *Ieee trans. softw. eng.*, vol. 22, no. 12, pp. 875–885, Dec. 1996, ISSN: 0098-5589. DOI: 10.1109/32.553636. [Online]. Available: http://dx.doi.org/10.1109/32.553636.

[35] A. Danial. (). Cloc, [Online]. Available: https://github.com/AlDanial/cloc (visited on 10/16/2016).

[36] FreeBSD. (). Wc - word, line, character, and byte count, [Online]. Available: https://www.freebsd.org/cgi/man.cgi?wc (visited on 08/24/2015).

[37] J. H. Christensen, "Using restful web-services and cloud computing to create next generation mobile applications," in *Proceedings of the 24th acm sigplan conference companion on object oriented programming systems languages and applications*, ser. OOPSLA '09, Orlando, Florida, USA: ACM, 2009, pp. 627–634, ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1639958. [Online]. Available: http://doi.acm.org/10.1145/1639950.1639958.

[38] J. Wadhams. (). Jsonlogic, [Online]. Available: http://jsonlogic.com (visited on 01/08/2016).

[39] Rollout.io. (). Rollout.io, [Online]. Available: https://rollout.io (visited on 10/21/2016).

# Glossary

**Android** Android is a mobile operating system (OS) currently developed by Google, based on the Linux kernel and designed primarily for touchscreen mobile devices.

**binary** The compressed file of all translated sources and files, which is usually stored in a binary format.

**compiler** A compiler is a computer program that translates source code written in a programming language.

**framework** A software framework is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions. Software frameworks may include support programs, compilers, code libraries, tool sets, and application programming interfaces (APIs) that bring together all the different components to enable development of a project or solution.

**iOS** Is the world's most advanced mobile operating system, and it's the foundation of iPhone, iPad, and iPod touch.

**native** Native apps are apps that are build using an approach which takes full advantage of the platform. The ap is developed specifically for one platform, and can take full advantage of all the device features. This delivers the best performance compared to other approaches.

**platform** The platform is the set of tools and devices to enable developers a way to deploy, test and manage mobile apps.

**Windows Phone** Windows Phone (WP) is a family of mobile operating systems developed by Microsoft for smartphones.

# Acronyms

**API** application program interface.

**GUI** graphical user interface.

**HTML** HyperText Markup Language.

**HTTP** Hypertext Transfer Protocol.

**IDE** integrated development environment.

**JSON** JavaScript Object Notation.

**MVC** Model View Controller.

**OS** Operating System.

**REST** Representational state transfer.

**SOAP** Simple Object Access Protocol.

**UX** User Experience.

**XML** Extensible Markup Language.

# Appendix A

# Diagrams

## A.1 Napton class diagram

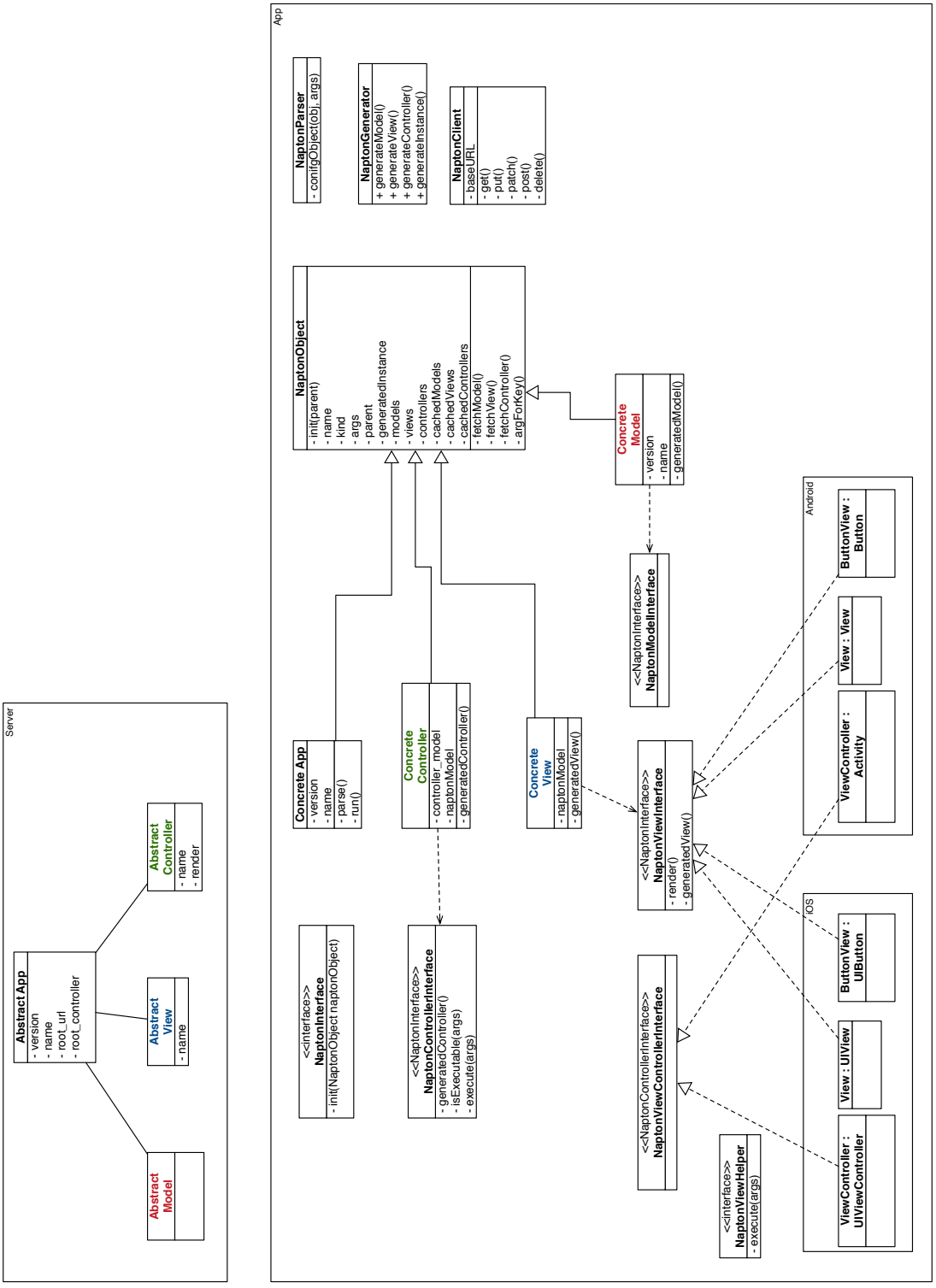




Figure A.1: Napton class diagram

# Appendix B

# Manual

# Napton Manual

Thom Ritterfeld

Version 0.1

# Table of Contents

# Chapter 1

# Getting Started with Napton

Napton is another approach to build apps, especially designed for mobile apps. Nowadays mobile apps are difficult to maintain because the update processes are slow due to long review times, changing technology and users that are not always running the latest versions.
To accomplish these shortcomings websites are generally preferred, but they do not provide the same advantages as native apps.
Napton is a combination of native development combined with the flexibility of web technologies. Many mobile apps nowadays use the JSON format to exchange data between the server. The JSON data contains the content to present on the user interface. Right now this JSON data mostly contains pure data contents about what the app needs to present. Why are we not using JSON as well for business logic and building the user interface?
Napton is a first step into a suitable solution to describe not only the content presented on the screen but the whole structure of an app.
Napton uses an abstract MVC pattern, is defined in JSON and hosted on a server. This allows the mobile device to adopt any user interface or functionality on the fly.
The following steps help to get you started:

## 1.1 Setup a web server to host JSON files

The easiest way to quickly start under macOS is to run this command in the Terminal:

```
sudo apachectl start
```

Check if the server runs by entering: http://localhost in your browser, you should see a message ”It works!”, which confirms that the server is running (see Image 1.1). Files can now be created inside the Apache document root, on macOS the folder usually is */Library/WebServer/Documents*.

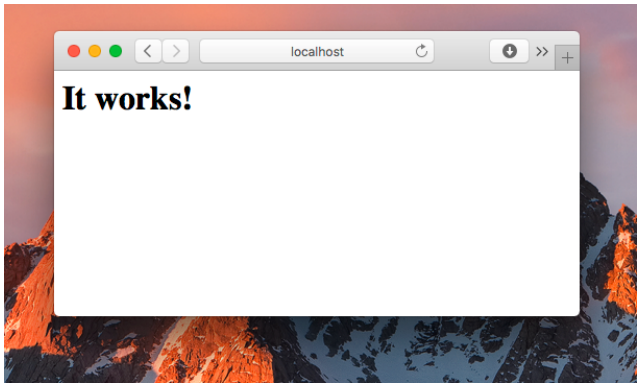


Figure 1.1: Apache runs now

## 1.2 Create a new Xcode Project

Create a new application inside Xcode. Create a new project and choose a Single View Application from the template drawer. Give the app a name which you would like to use (see Image 1.2).

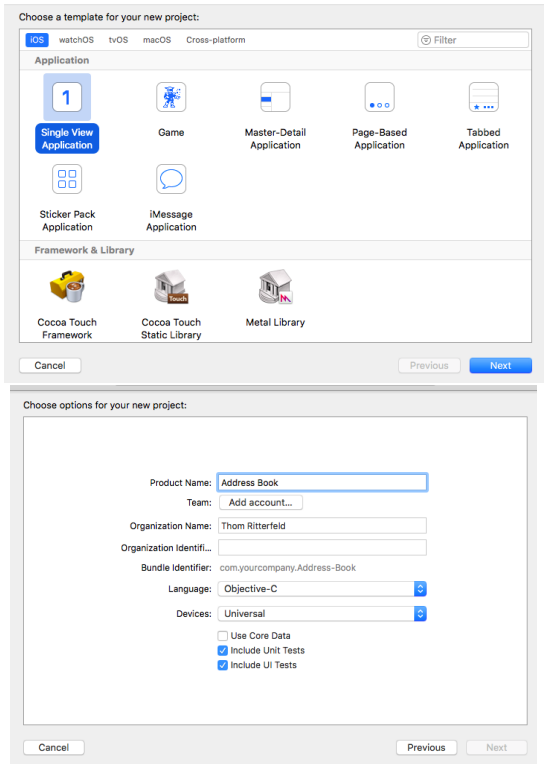


Figure 1.2: Create project

## 1.3 Add the Napton.framework to your Xcode Project

The Napton framework is available here: *http://ritterfeld.com/napton/Napton.framework.zip*. Next add the downloaded framework to your Xcode project by dragging it into your project files on the left side (see Image 1.3).

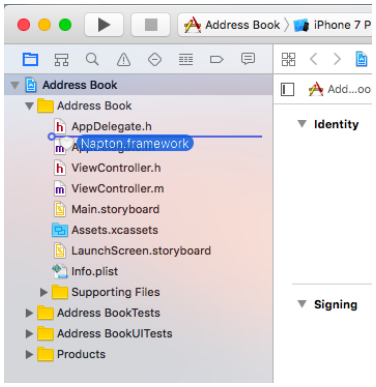


Figure 1.3: Add framework

*Note: make sure that the framework is an Embedded binary in the Project overview (see Image 1.4).*

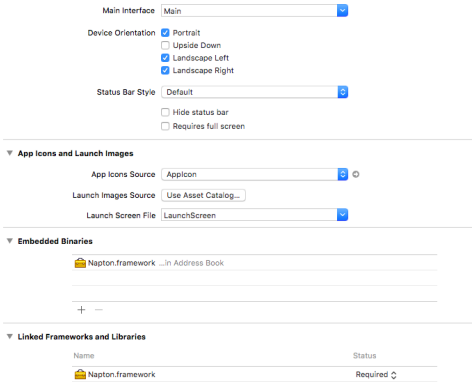


Figure 1.4: Add to embedded binaries

## 1.4 Add handlers to the Napton framework

Change the following lines of your AppDelegate.m:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    [NaptonManager provideAppURL:@"http://<YOUR_IP>/app.json"];
    return YES;
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
    [NaptonManager loadApp];
}
```

## 1.5 What's next?

Everything is done for now to start developing the app. This is done by specifying JSON files on the server. To run successfully an app, controllers, views and models are needed. The next steps will help you to understand how to specify these files and how the framework works. To make understanding the framework easier, the last sections always provides a sample, these samples together describe an Address Book app (see Figure 1.5) and should help to get you up and running faster.

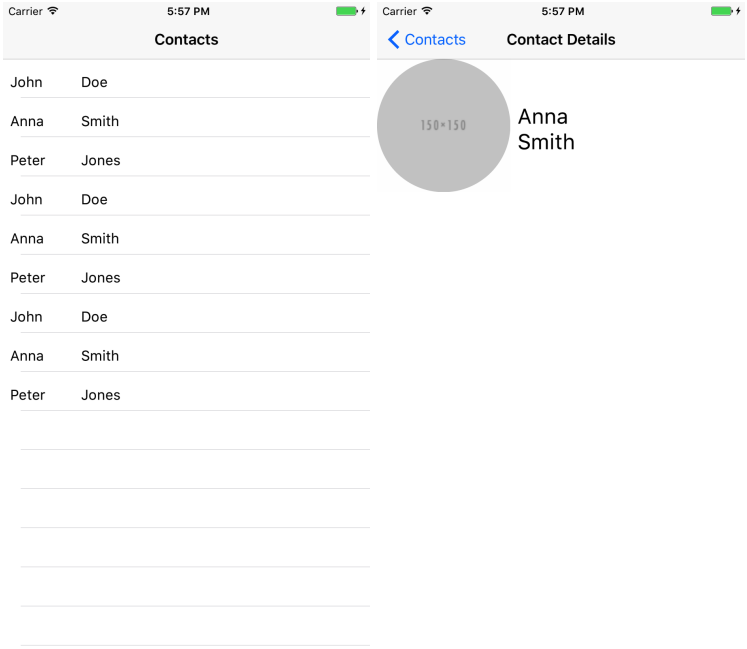


Figure 1.5: Sample Address Book app

# Chapter 2

# Specify the app

This chapter describes the properties that can be defined for the remote app file. This file is the main entry point of the app and is loaded on launch of the app. The URL of section 1.3 should point towards this file. The file basically describes the version number, name of the app, models, views and controllers that are needed upon launch. Every model, view or controller is defined by a key and the value should be an URL pointing towards the JSON specification. The Napton framework is storing these files for offline usage and to be able to launch the app quicker.

## 2.1 Properties

The following properties can be defined inside the app.json:

- **name** The name of the application, this value will be used to determine the application title.
- **version** The version of the application, if this is changed apps will be updated. (during development a the app is always refreshing as long as the DEBUG flag is set).
- **models** A key value JSON object, where the key is the name of the model and the value a string with a path or URL where the model(s) lives.
- **views** A key value JSON object, where the key is the name of the view and the value a string with a path or URL where the view lives.
- **controllers** A key value JSON object, where the key is the name of the controller and the value a string with a path or URL where the controller lives.
- **root_controller** A string with the name of the initial view controller. Make sure the name exists inside the controllers property.

## 2.2 Sample

Create a file on the server called app.json with the following content:

```
{
 "name": "Address Book",
 "version": 1,
 "models": {
   "Contacts": "/contacts"
 },
 "views": {
   "ContactsView": "/views/contacts_view",
   "ContactDetailView": "/views/contact_detail_view",
   "ContactRow": "/views/contact_row"
 },
 "controllers": {
   "ContactsController": "/controllers/contacts_controller",
   "ContactDetailController": "controllers/contact_detail_controller"
 },
 "root_controller": "ContactsController"
}
```

Listing 1: app.json

# Chapter 3

# Specify the models

Models are free to specify, this means there are no requirements or reserved words, and even existing API JSON resources can be used, as long as they confirm to the valid JSON data types.

## Specify objects

An object in JSON describes an unordered collection of values, using the key:value technique.

```
{"firstName":"John", "lastName":"Doe"}
```

## Specify arrays

An array in JSON specifies an ordered collection of values, which can be any valid JSON value. Also objects for example:

```
[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
```

## Specify values

A value in JSON can be a number, string, boolean, null, array or object. These are all supported data types in JSON which is very flexible to describe all required objects inside an application. Values can also be nested for example an object can contain a list with another object inside, for example a company has a list of employees can be described as follows:

```
{
"name": "Haribo"
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
}
```

## 3.1 Sample

Create a file called contacts.json, which represents the contacts of the Address Book, with the following content:

```
[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"},
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"},
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
```

Listing 2: contacts.json

# Chapter 4

# Specify the views

The view describes how an interface element looks like. A basic view has just a background colour, and is able to contain subviews (the contents property is used for this, which is an array containing other view elements). Some specific views display data, like labels, images or allow user input like a button. These views let you allow to define other properties, for these have a look at the API-reference inside chapter 7.

## 4.1 Properties

These JSON properties can be specified on all view objects:

- **name** The name of the view inside your application, this should match in any case you refer to this view. (Make sure the name is equal to the name inside the views property of your app.json)
- **kind** The class which you want to initiate, for example a ButtonView to generate a Button. (by default: View)
- **contents** An array to specify subviews of this view.
- **top** Distance to the top next element inside the view hierarchy.
- **left** Distance to the left next element inside the view hierarchy.
- **right** Distance to the right next element inside the view hierarchy.
- **bottom** Distance to the bottom next element inside the view hierarchy.
- **orientation** Orientation of the contents (”vertical” or ”horizontal”)
- **background_color** Hexadecimal string of the colour of the background. (for black ”#000000”)

## 4.2 Sample

For the sample we need to define three views this time, one for the list, one for the row which is an item of the list, and one for the detail view, the view you will see if you press on a list item. Define all three the JSON files on your webserver:

```
{
        "name": "ContactsView",
        "orientation": "vertical",
        "background_color": "#ffffff",
        "contents":[
                {
                  "kind": "ListView",
                  "items_model": "Contacts",
                  "row_view": "ContactRow",
                  "top": 0,
                  "left": 0,
                  "height": "fill",
                  "actions":[
                          {
                                  "action": "openController",
                                  "controller": "ContactDetailController",
                                  "controller_model": "@",
                          }
                  ]
                },
        ]


}
```

Listing 3: contacts_view.json

```
{
        "name": "ContactRow",
        "orientation": "horizontal",
        "contents":[
                {
                  "kind": "LabelView",
                  "text": "@.firstName",
                  "text_color":"#000000",
                   "text_size":16,
                  "top": 6,
                  "left": 8,
                  "height": 40,
                  "width": 80
                },
                {
                  "kind": "LabelView",
                  "text": "@.lastName",
                    "text_color":"#000000",
                    "text_size":16,
                  "top": 6,
                  "left": 0,
                  "height": 40,
                  "width": 100
                }
        ]


}
```

Listing 4: contact_row.json

```
{
        "name": "ContactDetailView",
        "orientation": "horizontal",
        "background_color": "#ffffff",
        "contents":[
                {
                  "kind": "RoundImageView",
                  "url": "http://placehold.it/150x150",
                  "top": 0,
                  "left": 0,
                  "height": 150,
                  "width": 150
                },
                {"orientation": "vertical",
                  "top": 0,
                  "left": 0,
                  "height": 150,
                  "width": 150,

                  "contents":[
        {
        "kind": "LabelView",
        "text": "@.firstName",
        "text_color":"#000000",
        "text_size": 24,
        "background_color": "#ffffff",
        "top": 50,
        "left": 8,
        "right": 8
        },
        {
        "kind": "LabelView",
        "text": "@.lastName",
        "text_color":"#000000",
        "text_size": 24,
        "background_color": "#ffffff",
        "top": 0,
        "left": 8,
        "right": 8
        },
                    ]
                }
        ]
}
```

Listing 5: contact_detail_view.json

# Chapter 5

# Specify the controllers

In Napton the controller basically owns a view and manages the user interaction. The render property defines the name of the view which needs to be rendered. Optional models, views or controllers can be specified inside the other properties. There exist some other controllers that manage navigation bars for example, for further information see the API-reference inside chapter 7.

## 5.1 Properties

These JSON properties can be specified on all controller objects:

- **name** The name of the controller inside your application, this should match in any case you refer to this controller. (Make sure the name is equal to the name inside the controllers property of your app.json)
- **kind** The kind of controller, by default a ViewController.
- **models** A key value JSON object, where the key is the name of the model and the value a string with a path or URL where the model(s) lives.
- **views** A key value JSON object, where the key is the name of the view and the value a string with a path or URL where the view lives.
- **controllers** A key value JSON object, where the key is the name of the controller and the value a string with a path or URL where the controller lives.
- **title** The title of the controller, this will be displayed in controller and navigational buttons inside the user interface.
- **render** Defines which view to render, refer to a view inside your app.json or inside the views property and matches this particular name.

## 5.2 Sample

Create two controllers on the server called contacts_controller.json and contact_detail_controller.json with the following content:

```
{
        "name": "ContactsController",
        "kind": "NavViewController",
        "render": "ContactsView",
        "title": "Contacts",
        "models": [],
        "views": [],
        "controllers": [],
        "actions": [
                {

                }
        ]
}
```

Listing 6: contacts_controller.json

```
{
        "name": "ContactDetailController",
        "title": "Contact Details",
        "render": "ContactDetailView",
}
```

Listing 7: contact_detail_controller.json

# Chapter 6

# Custom classes

The Napton framework has no support for all classes and options. It only delivers basic functionality of the most used views. If there is more functionality needed still custom classes need to be implemented. Right now this is not possible on the server side. To extend the Napton framework on the client you just need to implement the NaptonInterface. There are different interfaces to choose from, they all need to fit inside a part of the MVC pattern.

## 6.1 Implementing an interface

The following interfaces are available right now to be implemented. All interfaces inherit from the NaptonInterface, and every object is initiated using the following class call:

- **initWithNaptonObject:** The call wants to receive a class instance and passes a NaptonObject to your class. The NaptonObject contains the parsed JSON arguments and context about where the object is inside the hierarchy. This allows your implementation to retrieve arguments and execute actions on another object inside your application.

### 6.1.1 NaptonControllerInterface

- **configWithNaptonModel** This method is called whenever the model is updated. This can be caused because of many reasons most common is that the data has changed. In case you are subclassing a controller you should update the views.
- **actionIsExecutable** This method should return a Boolean if the instance is able to respond to the given selector, upon returning true the method executeWithArgs is called immediately.
- **executeWithArgs** This method is called upon execution of an action. It also contains a dictionary of arguments which are passed from the JSON and contain arguments that can be used for a particular method.

- **generatedController** This method should return the instance, in most cases return self here.

### 6.1.2 NaptonViewControllerInterface

This interface basically inherits all functionality of the NaptonControllerInterface. It only returns a different kind of instance in the generatedController, namely a UIViewController, which is needed for the iOS platform as controllers managing the views are specific ViewControllers. Because of the differentiation on iOS it made sense to create another interface, it may will be extended with other functionality in the future.

### 6.1.3 NaptonViewInterface

A subclass of the NaptonViewInterface always needs to return a UIView instance and implement a render method which prepares the view to be returned.

- **render** Is called every time the view is going to be presented or cached. This is defined by the framework, its just recommended to render the view which will be returned later on.
- **generatedView** Should return an instance of UIView that can be presented on the screen.

### 6.1.4 NaptonViewHelper

A view helper can be used to identify functions inside views. For example parsing the date or representing numbers into currency helpers can be used. Right now the framework only support string representations.

- **executeWithArgs:** The execute with arguments method is executed upon run time and the framework expects to return a string from the helper.

## 6.2 Gathering context

To gather a context of where the object you are creating is inside the composition hierarchy a parent object exists. This parent object is depending on your interface a NaptonModel, NaptonView or NaptonController. And allow you to walk through the app even till the NaptonApp. This makes it possible to get arguments and properties of other views and controllers.

### 6.2.1 Getting parent object

The initWithNaptonObject: is always called upon initiation of a class. The Napton instance which is passed to you brings you a way to gather more information about application data you need, but also arguments of yourself. It is

basically the parsed JSON specification with its arguments. In the next section we explain how we gather arguments from this context.

### 6.2.2 Getting arguments

To gather arguments from a NaptonObject is quite easy. You just need to specify the key and the NaptonObject tries to retrieve the argument if its available. Make sure in your custom classes to have a fall back or maybe display an alert so the developer knows what is missing or wrong. See the following code for a sample. The sample shows how to get the title argument of an NaptonController inside a class which implements the NaptonController interface. The code will print: ”Hello World”, ”My name is Thom”.

```
@implementation CustomController
@synthesize naptonController = _naptonController;

- (id)initWithNaptonObject:(NaptonObject *)naptonObject{
    self = [super init];
    if(self){
        //Set our parent context
        self.naptonController = (NaptonController *)naptonObject;
    }
    return self;
}

- (void)viewDidLoad {
    [super viewDidLoad];
    NSLog(@"title %@", [self.naptonController argForKey:@"title"]);
    NSLog(@"My name is: %@", [self.naptonController argForKey:@"dev_name"]);
}
```

Listing 8: Getting arguments CustomController implementation

```
{
 "name": "MyCustomController",
 "kind": "CustomController",
 "title": "Hello World",
 "dev_name": "Thom"
}
```

Listing 9: Getting arguments Napton controller specification

## 6.3 Sample

This sample shows how a RoundImageView class can be created. The RoundImageView is used inside the ContactDetailView within the views samples in sec-

tion 4.2. Create these files inside your Xcode Project, as they need to be specified inside the app and are not part of the Napton framework.

```
#import <UIKit/UIKit.h>
#import "NaptonViewInterface.h"

@interface RoundImageView : UIImageView<NaptonViewInterface>

@end
```

Listing 10: RoundImageView.h

```
#import "RoundImageView.h"
#import "NaptonModel.h"
#import "NaptonUtils.h"

@implementation RoundImageView
@synthesize naptonView = _naptonView;

- (id)initWithNaptonObject:(NaptonObject *)naptonObject{
    self = [super initWithFrame:CGRectZero];
    if(self){
        self.naptonView = (NaptonView *)naptonObject;
        [self render];
    }
    return self;
}

- (void)render{
    [NaptonUtils styleView:self withNaptonView:self.naptonView];

    [self setContentMode:UIViewContentModeScaleAspectFill];
    [self setClipsToBounds:YES];

    NSString *url = [self.naptonView argForKey:@"url"];
    if(url){
        [self setImage:[UIImage imageWithData:
            [NSData dataWithContentsOfURL:
                            [NSURL URLWithString:url]]]];
    }else{
        [self setImage:nil];
    }
}

- (UIView *)generatedView{
    return self;
}

- (void)configWithNaptonModel:(NaptonModel *)naptonModel{
    [self render];
}

- (void)layoutSubviews{
    [super layoutSubviews];
    [self.layer setCornerRadius:self.frame.size.width/2]; //make it circular
}


@end
```

Listing 11: RoundImageView.m

# Chapter 7

# API-reference

This API reference describes the objects currently available and their properties and actions(methods). There is also a sample included which show samples of the definition in code.

## 7.1 View

### 7.1.1 Properties

- **name** The name of the view inside your application, this should match in any case you refer to this view. (Make sure the name is equal to the name inside the views property of your app.json)
- **kind** The class which you want to initiate, for example a ButtonView to generate a Button. (by default: View)
- **contents** An array to specify subviews of this view.
- **top** Distance to the top next element inside the view hierarchy.
- **left** Distance to the left next element inside the view hierarchy.
- **right** Distance to the right next element inside the view hierarchy.
- **bottom** Distance to the bottom next element inside the view hierarchy.
- **orientation** Orientation of the contents (”vertical” or ”horizontal”)
- **background_color** Hexadecimal string of the colour of the background. (for black ”#000000”)

```
{
  "kind": "View",
  "background_color": "#ffffff",
  "top": 6,
  "left": 8,
  "right": 8,
  "height": 20,
  "contents": [],
  "orientation": "vertical"
}
```

## 7.2 ListView

The ListView basically draws a list with multiple items, and lets the user scroll through these items. The cells can be customised using another view.

### 7.2.1 Properties

- **items_model** The list of models to be used inside the list.
- **row_view** The name of the view which should be used for the rows.
- **actions** An array to define actions which should be executed.

```
{
  "kind": "ListView",
  "items_model": "Events",
  "row_view": "EventRow",
  "height": "fill",
  "actions":[
    {
      "action": "openController",
      "controller": "EventDetailController",
      "controller_model": "@.url",
    }
  ]
}
```

## 7.3 LabelView

The LabelView allows representing texts inside a view. This can be usefull to show texts of objects inisde the user interface.

### 7.3.1 Properties

- **text** The textual content of the label.
- **text_color** Color of the text in a hexadecimal string. (default is ”#000000”)
- **text_font** String with the font-family of the text.
- **text_size** Height of the text in points, default is 16.
- **text_lines** Number of lines the text label in integers, default is 0.

```
{
  "kind": "LabelView",
  "text": "Hello World",
  "text_color":"#000000",
  "text_size": 24,
  "background_color": "#ffffff"
}
```

## 7.4 ButtonView

The buttonView is able to interact with the user and executes actions upon touch.

### 7.4.1 Properties

- **title** The title of the button.
- **actions** An array to define actions which should be executed.

```
{
  "kind": "ButtonView",
  "title": "Show Dialog",
  "actions": [
          {
            "kind": "SystemController",
            "action": "showAlert",
            "message": "Warning!"
        }
  ]
}
```

## 7.5 ImageView

The image view is able to load and display an image for the given URL.

### 7.5.1 Properties

- **url** The url of the image to display.

```
{
  "kind": "ImageView",
  "url": "http://placehold.it/350x150"
}
```

## 7.6 ViewController

The viewcontroller manages a view and is able to present other view controllers on top.

### 7.6.1 Properties

- **title** The name of the controller can be appear inside menu items.
- **openController** Opens another controller with its view modally.

```
{
  "name": "EventDetailController",
  "title": "Event Info",
  "render": "EventDetailView"
}
```

## 7.7 NavViewController

The navigation controller behaves a bit different than a usual ViewController as it manages a Navigationbar and opens new viewcontrollers as pages instead of modally on top.

### 7.7.1 Properties

- **title** The name of the controller can be appear inside the navigation bar.

### 7.7.2 Actions

- **openController** Pushes another controller with its view inside the navigation hierarchy.

```
{
  "name": "EventsController",
  "kind": "NavViewController",
  "render": "EventsView",
  "title": "Events",
  "models": [],
  "views": [],
  "controllers": [],
  "actions": []
}
```

## 7.8 SystemController

The system controller does not manage views, it basically contains basic system actions which can be executed.

### 7.8.1 Actions

- **showAlert** Show an alert, which accepts an *title* and *message* as arguments. The alert is shown on top of all other views.

```
{
  "kind": "SystemController",
  "action": "showAlert",
  "message": "Warning!"
}
```