Polish-Japanese Institute of Information Technology
Chair of Software Engineering

Master of Science Thesis

Designing a device independent personal task management solution using RESTful API driven interfaces

by
Grzegorz Kaczorek

Supervisor: Mariusz Trzaska Ph. D.

Warsaw, 2011

Contents

Abstract

The thesis discusses the problems of developing multiple clients for applications with a centralised back-end. In particular – the problem of augmenting a web browser interface with multiple clients meant to be deployed on mobile devices.

The thesis describes the state of the art in the field of platform consolidation and device agnostic web applications, concentrating on their deficiencies and inferiority to native client applications.

The thesis then proposes a method of using the concept of a Functionally Extended Application Response generated by the API that and introduces certain design patterns that allow for maximum client logic reuse on multiple platforms.

A prototype of an API, web client and mobile client are all build to determine the feasibility of the aforementioned methodology. Based on the experience of prototype implementation the achieved results are discussed in terms of relevance to solving the problem described in the thesis with special care of defining the parameters for applications that will benefit from the approach presented.

Chapter 1

Introduction

This chapter describes in detail the goals of the thesis and defines concepts that are fundamental to its meaning: centralised API's, mobile clients and the problem of platform fragmentation in the field of application development.

1.1 Goals

The goal of the thesis is to provide a conceptual framework that decreases time and costs for developing an application that benefits from multiple clients.

This goal has to be distinguished from building an alternative to the platform development convergence efforts described in section 2.2.2 as a way to provide a pattern in which they can be used with greater efficiency and not superseded.

To provide a useful framework for concrete tasks, the range of the tasks in question has to be limited. The techniques described here are most useful for one of the most common type of software – a CRUD-activity oriented web application, mostly used by the provided web interface. The application will also have a companion client aimed at mobile devices and utilizing many of their features. The prototype has been described briefly in section 1.2.3.

1.2 Proposed solution

The premise of this thesis is that a lot of client side application logic can be delegated to the server and build a framework on the client side that parses the server responses and respond accordingly. Put in other words there will be an abstraction layer between the server and client that will allow the server to communicate rich messages to the client and the client to process them and act accordingly. The additional layer is called the

Functionally Extended Application Response (FEAR) and is described in detail in section 3.2.

The proposed solution is oriented around some core ideas that are in themselves a well established pattern in software development. The client-server[1] architecture is a venerable player in software engineering, although conceived in times that could not predict the advent of mobile devices. They did however solve similar design problems:

- need of a point of data storage (FTP, HTTP),
- limited access to hardware capable of significant computing tasks (Telnet)

These two problems persist despite today's mobile devices being orders of magnitude faster than early mainframe computers. The need for a central data storage – in terms of limited data storage – is less crucial now – mobile devices are capable of storing large amounts of data (especially in text format).

A new need has arisen however – the need for synchronization. A large part of the populace work on desktop or laptop computers, which are superior to mobile devices on many terms. While it is conceivable that mobile devices will reach the power and raw storage capability of contemporary "workstations", there is little hope of rapidly overcoming the display limitations inherent to small size[2]. There are also two other recognized usability concerns that are frequent in mobile devices: limited connectivity and inferior data entry[3].

Considering these obstacles, the conclusion is that for now, people need to use both types of devices[3]. The segmentation does not mean that some of the performed tasks will not overlap or use the same data. Personal task management and note taking are great examples of location areas where it is comfortable to have a data gathering device on the go, and then use a system with superior display and input devices to organize the data.

This necessitates data synchronization between all these platforms. The problem of two-point synchronization is complicated enough without the introduction of additional nodes [4]. This means that a central data hub is a good way of providing synchronization abilities in a point-to-point fashion. This role is fulfilled by the RESTful API introduced in section 1.2.1. This thesis probes the boundaries of what is feasible to centralize and communicate through FEAR.

[1]It is worth noting that in terms of this dissertation the "server" part is meant in terms of a service, not hardware, ie. a cloud computing service endpoint is by that definition a server.

[2]The yet to prove themselves contestants in the so called "screen-less display" field are the virtual retinal display [1] and bionic contact lens[2].

[3]It is not uncommon for someone to opt for more layers of devices – for example using a powerful mobile phone, a tablet, laptop and desktop computer. The technological gap in terms of computing power and storage space is closing, and the distinguishing factor remains the display size.

This is where the concept of the ultra thin client comes in. Traditionally[4] the term "thin client" was reserved for software that filled the former role in client-server architecture. The "thin" adjective signified that the client did not do all the work by itself, but rather delegated most of it to the server. A brief introduction to the goals of the ultra thin client is covered in section 1.2.2, while an in depth analysis of what signifies an **ultra-thin** client and how if relates to FEARs is done is section 3.2.

1.2.1 Centralised RESTful API

The term API can be viewed in a variety on concepts. Generally, an API is defined as an interface that grants the user access to higher level services. The term API in this thesis is generally meant as the API built as part of the prototype. The distinguishing factors are:

- Web accessibility – it will be accessed by the HTTP protocol on an end-point specified by the URI.
- Language and platform independence – the procedure will be defined by the request URI, header and body and the response will usually contain data that is represented in a generic notation such as JSON or XML.
- Centralised application logic – the data changes, relationships between entities and atomic action consequences will only be calculated on the client side and communicated by FEAR to the client.

1.2.2 Ultra-thin clients

The primary goal of the thesis is to reduce client production time by providing a focal point in which a maximum amount of data and logic will be processed be the API, without the need of engaging the client side. This allows for deployment of multiple client applications for different configurations with the minimum client code needed, which in turn will bring down the costs for the client applications and, maybe even more important, lower the cost of future upgrades of said applications.

That goal alone is already achieved by standards used in the World Wide Web – HTTP servers and hyper-text markup languages coupled with a client in the form of a web browser form a solid combination to build software upon. Both the data and the logic are produced in the form of documents on the server-side and presented in the browser (client). There are standards and best practices that even allow for varying the layout

[4]The term was introduced in 1993 by Oracle employee Tim Negris and popularized by Oracle founder Larry Ellison.

options, sizes and almost any aspect of the presentation layer depending on the size of the display of the devices, supported colour depths and so on.

This ideal separation is not without its cost. Lets enumerate some advantages of a native application written for the Android operating system compared to this web-based application[5]:

- Non-native usability – the interface of the application on the device is different that the one in the browser. Depending on the version of the system, and hypertext standard being used some device interface elements might be different than the user is accustomed to.
- Hardware feature exclusion – the client device hardware (G-sensors, cameras, GPS receivers) are unavailable to the application.
- Exclusion of native input methods – web pages are usually not designed to be operated by touch, nor do they have the capability to respond to platform specific device input actions (such as the "Back","Home" and "Menu" buttons on Android devices).
- Impossible system integration – mobile platforms allow for many special case scenarios for application that are not possible to achieve with server side technology (prominently software run as a system service and home screen "widgets").
- Large data transfer between the HTTP server and the client browser (see section 2.3).

The aforementioned flaws will put any application in the highly competitive mobile application world at a serious disadvantage compared with their native counterparts in terms of user experience. Hence the need for a solution that will allow for the developers all the advantages of a mobile application without the need to replicate client side logic for all target platforms (if it can be avoided).

1.2.3 Prototype

The prototype consists of three parts: a central API utilizing the FEAR pattern and two ultra thin clients (described below). The system is a personal task management solution designed to comply with a generic task management methodology, developed on an as needed basic and whose basic premises and entities are described in section 3.1.2. This particular type of software was chosen because of the following reasons:

- The basic CRUD operations implemented are standard enough so as other application can directly relate to this model.

- The type of application is ideally suited to mobile devices, as its primary role is to be personal and available for the user at any time.
- There are certain functions required of the client, that can utilize native mobile device functionality (e.g.. proximity based tasks, services run on the device to remind the user of the a given task).

1.3 Core concepts

This section defines that mobile devices are and hint at the problem of platform fragmentation.

1.3.1 Mobile clients

This section introduces the reader to the field of software applications that are of special interest during this thesis. Because of the ever increasing impact that the mobile device market is achieving, the thesis is aimed at lowering the costs of developing clients for mobile devices. Mobile devices also suffer from acute platform fragmentation and are hence good candidates for developing several disparate clients. This section describes what said devices are, how can they be categorized and the background information on their development and adoption.

Definition

A mobile client is any client that is run on a **mobile** device. Mobile devices (also called "hand-held devices") are "pocket sized computing[5] devices, typically having a screen display with touch input and/or miniature keyboard"[6].

The traits of mobile devices that are fundamental for their use are[7]:

- designed to be operated outside of a standard office or home setting,
- hardware resources limited – in comparison to a average desktop or laptop computer,
- smaller in size and weight than the aforementioned.

The above characterization is, admittedly, quite subjective and fluid. However - so is the field it describes. As with many other modern disciplines, the state of the art in this one is drastically altered every year, leading to further developments not only in the technology, but most importantly, in the impact it has on the whole market.

[5]Please note that hand-held devices that do not have computational ability (such as radios) are beyond the scope of this thesis.

Multi-purpose devices	Specialised devices
• Mobile computers (laptops, netbooks, netvertibles) • Personal digital assistant / enterprise digital assistant • Mobile phone, smart-phone, feature phone • Tablet computer	• Hand-held game console • Portable media player • Digital still camera (DSC) • Digital video camera (DVC or digital camcorder) • Personal navigation device (PND) • Calculator • Pager

Table 1.1: Categorization of mobile devices with usage specification.

Categorization

Wikipedia lists several classes of mobile devices[6], however the current list (as of the time of writing) is riddled with inaccuracies and semantic chaos, the most obvious one being mixing specialised devices with multi-task ones. Table 1.1 contains a more organised and cross referenced list[8].

The mobile computer category is of particular interest. The term "mobile" is used as a means of distinguishing them from the classic stationary computers. While laptops, netbooks and netvertibles are fine for work and travel, they are not so mobile that most people would consider taking them everywhere they go. Furthermore, the platforms used on this group of devices is usually the same and their stationary counterparts with a lot of platform fragmentation issues (mentioned is section 2.1.1 already solved and clients being a low priority. This is why laptops, netbooks and netvertibles are not be treated as mobile devices in this thesis.

Because of the hight ambiguity in this field in terms of what constitutes a mobile device, this thesis focuses on the commercially available and popular devices that first come to mind when the term "mobile device" is uttered. The primary focus is on phones with computation ability and also tablets (or tablet PCs). These are the devices that are predicted by many to be to be the future of personal computing[9][10][11]. This is the so called "Post-PC" paradigm, distinguished by switching the architecture to the ARM processor (as opposed to the long-established desktop x86 architecture) and using multi-touch capacitive touch screens for a more natural user interface.

History

The two lists in section 1.3.1 present very different device groups. They differ not only in technical specifications, but also usage history. Specialized hand-held devices are well established, driven by the business needs of professionals, that need a portable device to do a very limited amount of tasks. Their goal is to maximize productivity for a given narrow band of tasks which are not performed in the same place. The most obvious examples would include:

- credit card terminals in restaurants,
- portable devices for getting the receiver of a package to sign for it, used by delivery service companies

The world got a glimpse of multi-purpose hand-help tools with the generation of Personal Digital Assistant (PDA) devices. While the combination of limited functionality and less than ergonomic design ultimately proved to be formidable obstacles for the widespread adoption of said devices, they were also proof of concept that a general purpose device for mass markets was possible.

Because the role of the standard PDA were gradually transferred to the well known mobile phone, the market share of the devices dwindled. The growing role of mobile phones eventually sparked an interest into developing them into multi purpose devices. There were many milestones in this adoptions, the most notable of which seems to be the integration of the emerging standard in business communication: e-mail. A Canadian firm called Research In Motion developed an environment to change the pull-based email approach into a push-based one by a system called Blackberry Enterprise Services. This system, coupled with innovative devices setting the standard for comfort, ergonomics positioned RIM into a leader in the market of business phones (the Blackberry brand being almost a synonym of a business oriented phone in many markets).

This however was still limited to mostly the business world. As the cost of the technology went down - producers of standard mobile phones incorporated more and more of the advanced computing features - most notably internet access through a variety of channels.

A few more years had to pass for the technical obstacles to be overcome. One of the universally acknowledged paradigm shifts in the area of mobile device consumer markets was the release of the first iPhone by Apple in Jan 2007. The devices polished design experience led to a new paradigm of touch-based interfaces that can be easily operated by all people regardless of the level of computer experience. The success of the iPhone led to an ever increasing interest of phone manufacturers. In late 2008, the first smartphone with the Android operating system was released – the HTC Dream. This market a different

kind of revolution: because the system was open, there were no licensing fees and both the manufacturers and mobile operators could make changes to the system. This allowed for a wide range of devices to be equipped with this modern operating system, even the low-end models, leading o a proliferation of smartphone usage that continues to this day.

Today – mobile operating system are run on both smartphones and tablet PCs. The major ones along with the market share are presented in section 2.1.2. Whatever the manufacturer or operating system, the fact is that mobile computing is growing rapidly, akin to the growth of the personal PC in the 90's and along with it the need for mobile applications.

1.3.2 The problem of diversity in the area of client platforms

A client is an application running on the target platform and that is used by the target audience. When considering the case presented in the prototype section (1.2.3), such clients are:

- web interface – the main client – due to the fact that web browsers are available for all modern software platforms[6], it is a must-have for any application,
- mobile client(s) targeted at n platforms – due to the rising popularity of mobile devices, the owner of the application will want to provide a client that can increase its usability on its users devices.

As described in section 2.1.2, mobile devices run on variety of platforms, four of which are widespread enough to together make up for over 90%[7] of the market. This would mean that the clients needed to be developer to cover all the main platforms can reach 5 (including the web interface). Without any code-sharing techniques, this would mean every new client substantially increases the amount of work needed to develop and update cross platform support for the application.

The problem of mobile platform fragmentation bears many similarities to the fragmentation of platforms for desktop environments[8] and there are efforts to increase the amount of code that can be shared between mobile clients. The state of the art this field is presented and discussed in section 2.2.2, where it is also explained that those solutions are based on the implementation level of the clients, not the conceptual layer.

[6]Even most mobile devices have web browsers, but these are of varying quality and can offer no guarantee of web standard compliance, much less cutting edge feature support (like HTML5 and CSS3).

[7]EU market share, June 2011. A detailed analysis of market share is made in section 2.1.2.

[8]this has been covered in depth in section 2.1.1

1.4 A short summary of the results achieved, their practical and theoretical significance

The main theme of this thesis is building and API that has as much embed application logic as possible without sacrificing application responsiveness. Because this is such a prevalent problem, there are many existing solution, each of which is explored in the thesis. The proposed alternative solution is one depending on an extended response (FEAR) from the API that can be handled by the client.

Because the problem is such of a complex nature the achieved results and lessons learned (described in details in chapter 6) must be narrowed to the analysed circumstances of building such an application:

- The application is not a one-time-only project
  The concept is implemented with either the continuing extension of the deployed application, in effect saving on coding for all future upgrades, or deploying similar projects in the future using the same technologies, thus getting ready functionality via code reuse. This is discussed at length in section 6.3.

- The application does not have an overly complicated data model.
  The prototype has only a few main entities and about 20 supporting entities with quite clear relations between them (see section 3.1.2). While it is not a simple matter to determine how the rise of complexity would affect the aforementioned system, one can imagine that the amount of changes that would need to be communicated by the FEAR would rise exponentially with the rise of show entities and relationships between them. This however should not be a problem with a decent amount of computational power and communication bandwidth, and it the issue should occur, can be remedied by optimization strategies.

- API connectivity must be assured.
  Because the client rely so heavily on the API, the connection must be available at all times. Is the solution requires frequent work without relying on the API, some of the possible workarounds are mentioned in section 6.4.2. A more detailed account on the effect in listed in section 6.2.2.

- The application can be stateless.
  The application described in this thesis relies on a request-response paradigm for most of its features. Because it's a single user system, the system usually does not even have to poll the server for updates because they are made by the client that is usually being used and hence any changes are triggered by that client. This is in stark contrast to applications that require a near direct connection to the server to

function optimally (chat clients and multiplayer games for example).

Barring the aforementioned provisions and disadvantages of the approach, there are also significant advantages to its implementation. These have been described in section 6.2.1. The most significant of these are:

- Sharing business logic with decreased usual negative effects
  The FEAR approach allows for centralizing logic like data validation and relationships between entities without the increased data transfer and decrease responsiveness of traditional approaches. This allows for more maintainable clients and better control over data in the API.
- Added value of a ready API
  There is substantial value in having a ready, tested and robust API that can be used by willing developers to implement their own tools and augment the existing ones. Building an application on top of an API from the beginning assures that the quality of said is high and the exposed functionality is fully controlled.
- Emergent properties
  Possibilities of implementing complex features with little effort can be uncovered. Section 6.4.2 describes how to leverage the "change" FEAR directive group to assure up-to-date data in clients with very little coding in the clients themselves.

The overall finding is that the FEAR approach is an interesting concept with very direct business applicability. Due to the limited nature of the prototype and this thesis further study is needed both theoretical and practical to better define the parameters of projects that this approach will benefit most significantly.

Despite the provisions for the approach, the potential market reach is quite vast. Sufficed to say that some of the biggest web applications in the world right now - Facebook, Twitter, LinkedIn etc.. could all be implemented using this approach and would gain some if not all of the benefits described in this thesis.

Chapter 2

Problem description

This chapter summarizes the motivation for this thesis and describe the concepts that are relevant to the proper understanding to the problems posed. The first section provides background information about the problem area. The second introduces already existing solutions both conceptual and practical. The last two sections describe in detail the problems with the current prevalent solutions and propose a preliminary plan of action to improve upon those shortcomings.

2.1 Software engineering on multi-platform environments, mobile devices in particular

This section provides the reader with background information regarding the challenges presented to software development by platform fragmentation and introduce the main mobile platforms.

2.1.1 Personal computer platform fragmentation

Software engineering has always faced the problem of target platform heterogeneity. In the beginning software was written in a very low language designed specifically for one particular hardware type. When the need has arisen for these machines to act together and communicate, and it was obvious that the computing revolution was there to stay, engineers have begun to write more abstract, higher level "wrappers" for the lower level languages. These allowed for more code reuse and increased the productivity and scale of software development.

The culmination of these hardware efforts was the dawn of the personal computer (PC) era. Led by Apple Computers and IBM, the hardware for consumer grade computing has been largely standardized, which brought upon massive growth for the industry. Since

the beginning of the PC era to this day, there were always 3 mayor software/hardware ecosystems that counted in the field:

- Apple Inc. – Mac OS
- Microsoft – DOS/Windows
- UNIX-based and UNIX-like (i.e. Linux) operating systems

All of these platforms had supported (and still support) the most popular programming languages. So why the fragmentation? The main reason is that the systems have different modalities – the Mac OS and Windows families were always driven by a graphical user interface, in contrast to the command-line driven DOS, Unix and Linux. To allow the software to communicate with the operating system, vendors shipped a programming API for potential software developers (usually in the most popular programming languages).

The differences if these API were large and driven by the different primary modalities – one could not write software for all these platforms, although because they supported popular programming languages, separate software packages could share the logic of the application, while having to build a front end and communication schema that was tailored to the target system.

Though roughly 30 years have passed since the birth of the PC, the problem of platform fragmentation still persists. Cross-platform languages, virtualization and various frameworks have made it a lot easier for software developers to write cross-platform software, yet there are still problems that are hard to overcome.

No matter what the cross-platform development strategy applied by the developers, there will always be subtle errors in the way the software works, which considerably raises the amount of support the manufacturer has to offer. Another grievance is that the developers are limited to the lowest common denominator in terms of platform capabilities, which puts such applications at a disadvantage to native ones.

There are also subtle differences in user interface conventions that will hinder its usability. Wikipedia provides an example[12]:

> "Applications developed for Mac OS X and GNOME are supposed to place the most important button on the right-hand side of windows and dialogs, whereas Microsoft Windows and KDE have the opposite convention"

The similarities between the PC and post-PC platform fragmentation are worth mentioning because they can give us a glimpse of the probable future situation in the mobile market platform. As described in section 2.1.2, the general platform types are quite similar in the mobile world, as are the problems facing developers. In summary:

- The mobile platform world faces similar challenges that the PC-era encountered in the last 30 years. In those 30 years, the situation has been constantly improving, although the problem has not been completely solved.
- Hardware differences coupled with modality and philosophy[1] disparities between software ecosystems will prevent standardization.
- The possible higher level frameworks, abstracting to the common denominator can be at a disadvantage compared to native platform solutions, both in terms of function and performance.

2.1.2 Main mobile platforms

Mobile operating systems can be categorized into two distinct groups: OEM systems and smart device operating systems. The differences in these two categories revolve around the computational power of the device and the range of devices they support.

OEM systems are currently the more popular category as a whole, mostly due to the popularity of feature phones – low cost mobile devices that are primarily used as phones, but can also perform other common tasks (such as web browsing and email access). The most popular of these is Symbian[2], developed by Nokia. Most other prominent feature phone producers make their own OEM systems tailored to these devices.

Despite their popularity, OEM systems are not a promising area for mobile development – the fragmentation, lack of programmer support and limited device resources prove a formidable obstacle for would-be developers. What is more, there is a steady decline in feature phone sales, with 2,5% feature phone users switching to smart phones every month , the "tipping point" when the amount of users with OEM system will be less than those with a new OS, is predicted to be in June 2012[13]. Most of the feature phone producers have already announced the discontinuation of their proprietary systems in favour of smart device OS.

Operating systems for smart devices are more promising for development. At this time, there are 5 main competing operating systems in this field (in alphabetical order):

Google Android is an operating system backed by the Open Handset Alliance – a joint effort of several companies in a diverse area of fields, from hardware manufacturers to legal services. Android in as open source operating system based on the Linux kernel

[1]Platform philosophy (or guiding principles) are at least (or even more) as important as the technical side. The Apple philosophy of "It just works" stands in stark contrast to the heavy tinkering required in some Linux distributions just to install the system and get it up and running.

[2]There is some controversy on whether Symbian should be classified as an OEM OS – mostly due to the fact that a lot of the devices that were deployed with the systems should be described as smartphones. Two facts however support this opinion: Symbian is used mostly in Nokia phones, and has been discontinued, which conveniently places it in the "legacy" category with other OEM systems.

supporting a very wide range of devices: phones and tablets and even intelligent home appliances.

Android is also the operating system that is the target of one of the clients for the prototype of this thesis (see section 5.1.4).

iOS – used by Apple Inc. in the iPhone and iPad products. Created with ease of use and simplicity in mind, it is supported by a formidable ecosystem of applications (available via Apple AppStore).

Windows Phone, Windows Phone 7 – a product of Microsoft, the latter is also announced by Nokia to be their primary phone operating system

BadaOS/Tizen – BadaOS is a system designed by Samsung Electronics to provide smartphone level support of applications to low level feature phones[3]. Tizen is a next generation operating system currently in development, with first devices planned for Q2 of 2012. The system was based on MeeGo, developed by Intel and Nokia and is planned to be merged with BadaOS.

BlackBerry OS / QNX – developed by and powering the mobile phones of Research In Motion, these systems are especially popular with business clients. While the Blackberry OS is an established, yet diminishing brand, the QNX is a new system developed to power the new wave of touch screen enabled mobile devices.

Market share

Current market share for different operating systems is shown in figures 2.1 and 2.2. The markets described therein are somewhat different in that the EU has, on average, a consumer willing to spend more on electronics. On the other hand, the markets are quite similar because the gap between the UE and rest of world is filled by even bigger consumer markets like the USA, Canada, Australia and Japan. Both markets display some similar tendencies:

- Rapid increases in Android market share, which can be traced to a massive amounts of released hardware in every price range.
- A decrease in Symbian shares. This is to be expected and to accelerate even further, due to the abandonment of the system by manufacturers. This trend is less prevalent in the worldwide scale due to a large role feature-phones play in massive emerging markets with low-price consumer demand (mainly India and China).

[3]After Google Inc. acquired Motorola Mobile in July 2011, Samsung covertly reinvigorated their efforts in this mobile system for fear that the primary developer of the Android platform (Google) would favour Motorola in terms of hardware support. Samsung has not announced any changes in their strategy and officially plan to support the Android OS.

- Custom system are declining – as smartphones gain popularity, manufacturers tend to switch from self made systems to the open source Android platform.
- The BlackberryOS is slowly declining in popularity. This can be traced to the general decreasing satisfaction with Blackberry handsets with consumers worldwide.

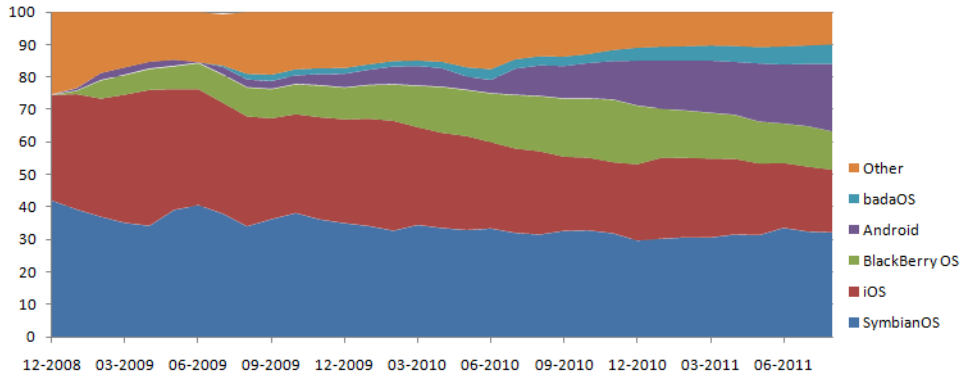

Figure 2.1: Mobile OS market share in time (worldwide). Source: [14]

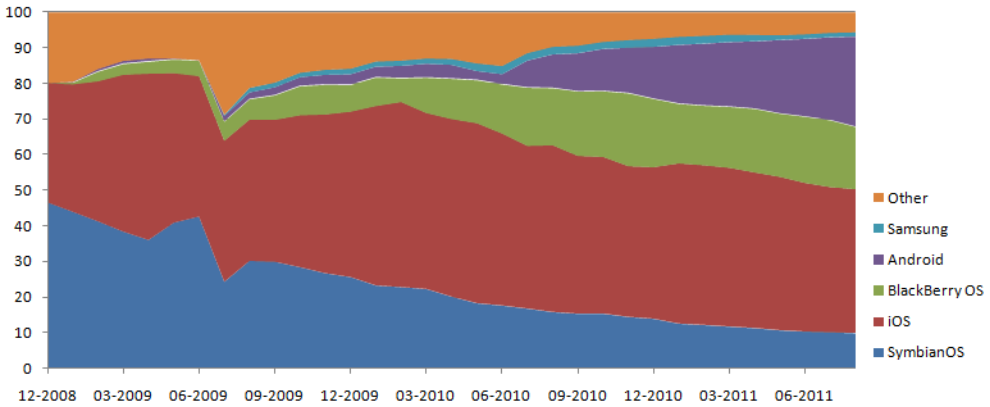

Figure 2.2: Mobile OS market share in time (European Union). Source: [14]

2.2 Current state of the art

This section describes the current trends in dealing with the problem of platform fragmentation. The solutions are divided into groups of similar approaches to the problem and the most significant implementations of these problems are introduced.

2.2.1 Solution segmentation in general

There are a few main solution groups to overcoming the problems inherent to developing clients for disparate software platforms. Below is a summary of those approaches along

with the distinguishing factors and examples of popular frameworks that allow for creation of software that uses them.

2.2.2 Mobile web applications

Building upon the swift rise in popularity of the techniques to be soon proposed in the HTML5 standard, there are several advocates of using standard web technologies to build cross-platform applications and using the native browser to access these applications [15][16][17]. There are already many popular application frameworks that facilitate creating touch optimized browser interfaces.

This is often a good choice when there is need to build a fairly consistent experience between desktops and mobile devices. The applications will, however suffer from all the drawbacks of non-native applications mentioned in section 1.2.2. Depending on the solution, there can be problems with older legacy devices and systems shipped with less-than-perfect default browsers.

The following are three of the more popular frameworks created to aid the developers in providing a better user experience for mobile devices:

- Sencha Touch,
- jQuery Mobile,
- SproutCore.

2.2.3 Multi-OS native application engines

This group of solutions is geared towards using the framework to build several native applications using a single code base. The companies also offer a build platform that allows distribution of application packages without the need to compile them. Due to the relatively small gap in philosophy of mobile device operating systems, these solutions provide for utilizing a large subset of device functionality while keeping the native interface and functionality, which allows them to supersede most limitation listed in section 1.2.2.

Still – the gap between the mobile and desktop settings is apparent and despite that some of these solutions provide the ability to build the applications for desktop environments (i.e. Appcelerator Titanium[18]), in practice the code base has to be versioned manually to bridge these gaps.

From the solutions mentioned below, only Rhomobile Rhodes[4] uses a "non-web"

[4]Rhodes does use web technologies, but they are aimed to be the View of the paradigm, all of the business logic is designed to be written in Ruby.

technology – the Ruby programming language – other frameworks use only a combination of HTML, CSS and JavaScript APIs for accessing the hardware features of the devices, which make them better suited for programmers with a web development background.

- Rhomobile Rhodes,
- Appcelerator Titanium,
- PhoneGap.

2.2.4 Other - Adobe AIR

Adobe AIR (Adobe Integrated Runtime) is a runtime environment from Adobe Inc. that allows for building applications that can the run on any supported platform. Because the resultant application is run in a sandbox, the consistent look is assured on all platforms. This does come at a cost; AIR does not support native client interfaces, but does support device hardware functionality where available (camera, GPS, accelerometers etc.)[19]. The platform was esteemed as one of the best options when the application being built needed a custom interface due to its character – the lack of native UI elements was not a problem then.

The sandbox and custom runtime environment however prove to be quite resource intensive and hence are not recommended for older devices, somewhat limiting their reach.

Also – the platform has lately been assigned a deprecated label. First, Adobe announced that it would be no longer planning on developing the mobile version of Flash Player, which is the internal rendering engine of the AIR platform. In late 2011 Adobe announced that it would be instead working on developing tools and runtime environments for the emerging HTML5 technology stack[20]. Later the company acquired Nitobi, the developer of PhoneGap, mentioned in section 2.2.3. These moves tend to show that AIR will be superseded by another framework based on HTML5 in the near future, however the lack of a specific stance from Adobe might signify that the company is diversifying it's portfolio and will continue to support AIR as a tool in areas that the framework excels at – delivering a consistent, media-rich user experience.

2.2.5 Conclusion

Despite all the efforts and products aimed at providing the developers with cross-platform techniques for creating mobile applications, there are still a lot of unanswered questions

and unsolved problems. Coupled with the dynamic rate at which the market is changing leads some experienced developers to doubt a reasonable solution is possible and focus their efforts at streamlining processes for building applications for all major mobile OSes, or just those that are expected to be the most cost-effective. These sentiments are evident in the following words of Carol Realini[21]:

> "You just get used to it. If you think the world is all about iPhone and Android, just blink and it will be something else. It's going to be a fragmented environment, and it will depend on your application."

Others offer a dissenting opinion, looking to one or more of the aforementioned solutions as the silver bullet that will allow the developer to focus on platform agnostic software development in the future[16][17].

This thesis proposes a solution that is a conceptual one, and hence can work with either of the possible future states, but it is probably more valuable of a tool in a fragmented market and also in situations where deploying additional clients to other than mobile devices is expedient (like non web-based desktop applications).

2.3 The pitfalls of the traditional client-server architecture

The root of the problem with traditional web page interfaces is the round-trip done between the HTTP server and the browser to accommodate every request.

Figure 2.3 is an illustration of the standard process. Because HTTP is a stateless protocol all the communications are self-contained and independent of each other, we will however want to distinguish between the first user request (Rq1) and subsequent ones (Rqn).

The process is initiated the user sends a request, usually with the GET method indicating an action that he wants to accomplish. In the first iteration, this would be accessing the application main page or login page. The server sends back the response, along with 100% of the web page code. Subsequent actions from the user also get that response. Let us assume that the user action requires the code to be changed by n percent, where $0 \leq n \leq 100$. Depending on the application, and the usage profile, n will vary greatly. What that means however is the response body is $(100 - n)$ percent redundant on subsequent requests. This means that the time is takes to send the redundant part to the browser and render it is wasted. Also – the redundant data will lead to increased costs in data transfer.

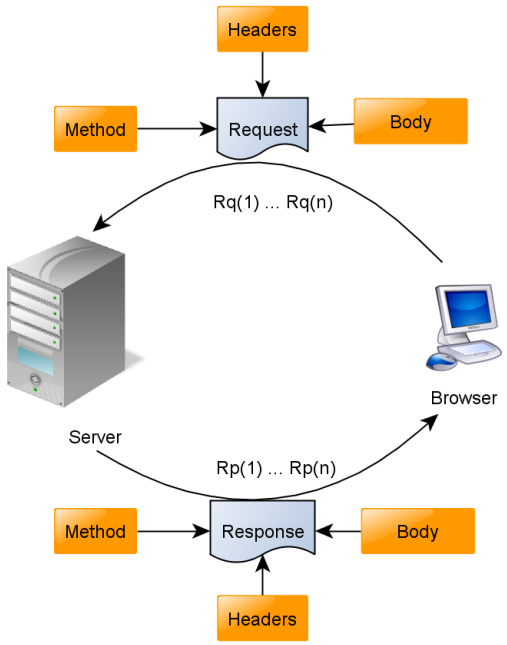

Figure 2.3: Basic browser – server communication during a web based application life cycle

This was the motivation for developing technologies that would limit the amount of redundant response text. The most frequently used technology currently is the use of asynchronous requests that on return are parsed by JavaScript and manipulate the DOM to change the affected areas. This is what's most often referred to as AJAX. Depending on how this is implemented, this can cause a major improvement in response utilization.

The early versions of such implementations usually handled this by replacing a whole container element of the DOM model related with the action that the user did. This facilitates the need to generate the markup on the server side and then replace a whole DOM node with a new one and was an effective technique provided the page elements were not to dependent on each other. Modern JavaScript allows for a much better utilisation ratio because the markup is generated on the client side, and only the updated data elements are sent by the request. Figure 2.4 shows the difference between these two approaches. In the example the response body length for the two approaches were 170 and 92 bytes, which constitutes an almost 50% decrease.

It must be noted that it would be very hard to achieve results that are close to n. This is because even in the most efficient ways of data transfer, there has to be some amount of meta-data to identify the DOM elements that need to be changed.

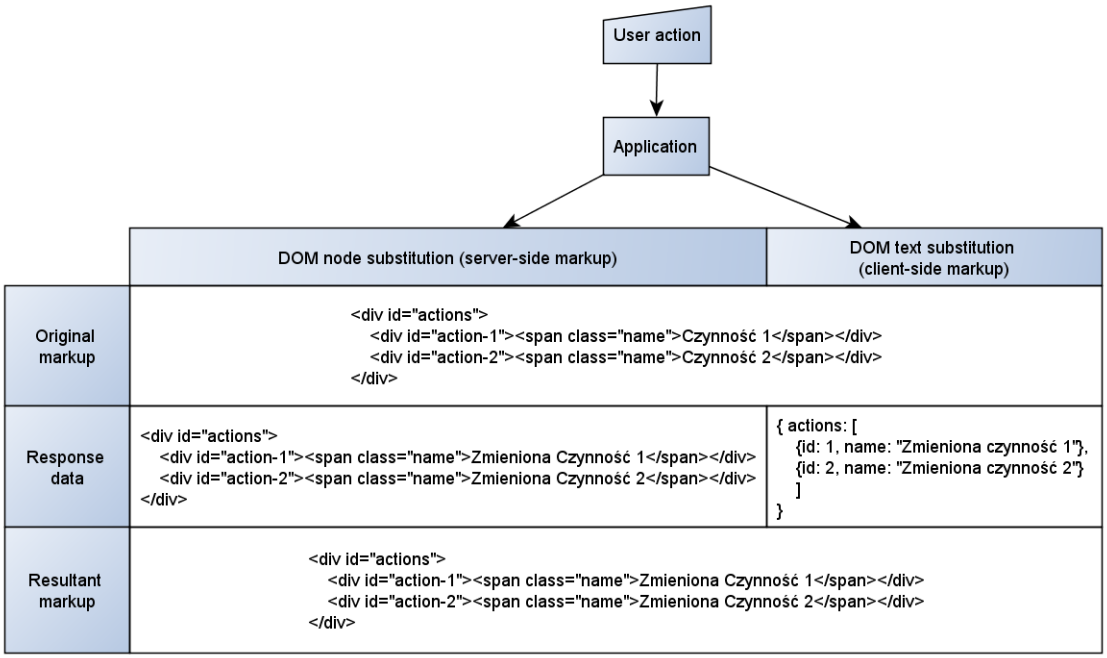

Figure 2.4: DOM node markup substitution vs data substitution.

2.4 Improvements suggested to the presented technologies

The improvements upon the traditional HTTP round-trip model described in the latter part of the last sub chapter go a long way towards making it a more efficient standard for building web-based applications. This thesis focuses on the more efficient data response model as a basis for later implementations.

It does however have its own challenges:

- In order to properly handle the incoming data, there must exists a unique data mapping contract between the server (for embedding meta-data) and the client (for parsing said meta-data, translating it the appropriate DOM node and changing that node). An in depth description of such a problem and an implemented solution is presented in section 6.1.1.
- There are several other 'events', that can happen to a data object besides updating - it can be deleted, closed, locked for editing etc. These events need to be handled and contracted. The basic elements, their significance and process of passing between the API and client are discussed in section 3.2.

Chapter 3

Project premises and philosophy of FEAR

This chapter lists the basic premises of the application prototype, along with a basic architecture entities and goals.

Section 3.2 introduces the Functionally Extended Application Response (FEAR), how can it be used to increase the server side code percentage and the example specification of directives that it can send to a client, along with examples how are they used in the prototype.

3.1 Premises and architecture

This section gives an overview of the application in terms of the elements that it's meant to manage, and the premises on which it is based.

3.1.1 Prototype premises

The following are the basic premises of the project in terms of the resultant prototype as a software product, along with the practical aspects of their implementation and expectations:

- The product consists of three elements: a back-end (API), a web client and a mobile application. The first two are implemented based upon the same application framework (see sections 5.1.2 and 5.1.3) and hence have one code base, but should be treated conceptually as separate entities. The mobile application is completely separate and built using the Android Software Development Kit (described in section 5.1.4).

- Developed by a team of 1 programmer/web developer.
- The primary goal is be to test the assumptions of this thesis.
- There is no project timetable – it is created on an ad-hoc basic as a response to the constraints of the thesis itself.
- Beyond the use of the product as a prototype, it will be developer further as an open-source project that aims to fill specific needs described in other documents[1]. It should be noted that the future of the project, whilst relevant to this thesis as point of reference for future development, will only be considered when explicitly noted.

3.1.2 Application entities

Figure 3.1 shows the relationships of the basic business entities in the system. Below are subsections describing each of these. The descriptions do not contain relationship information unless they go beyond the simple role of containment and categorization.

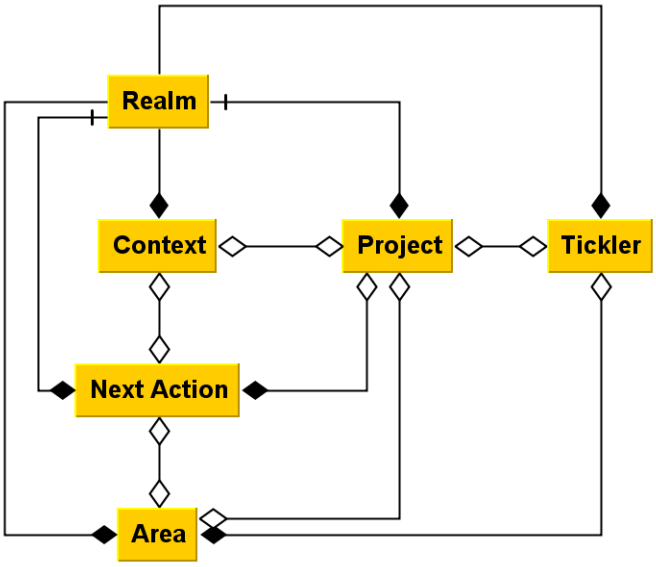

Figure 3.1: Basic entities of the prototype application, with relationships.

Next action (NA)

- the basic element of the system — it represents an action restrained in time, space and complexity. This is the most used element of the system. All other elements exists solely

[1]These were created as a project on another venture and will be released in the form of a project roadmap along with the source code

for the purpose of better management of these “next actions”. NAs can have 3 distinct statuses: “current” ,“waiting for” and “future”. These statuses are used for displaying the type of task that the user wants the see at a given time. The two statuses besides “current” represent an action that cannot be immediately acted upon. The “waiting for” status signifies that an action has been deferred to something beyond the scope of the system (like some other person or occurrence), while the “future” status is given to any task that has any incomplete dependent task.

Task dependence is a concept that allows creating a set of simple, concrete actions leading to the desired outcome, which is one of the primary tenets of the GTD system. All action in this chain have to belong to the same Project. An action can depend on zero/one or many other actions.

Project

This represents an desired outcome, accomplishment of which is predicated on completing two or more actions. Because the outcome can prove elusive even when all of the project NA are accomplished, setting the project as completed requires a separate action from the user.

A project may be marked as “someday maybe” to distinguish it as a low priority goal that is set in the far future. The user should have the ability to review these projects and be reminded to do so on a regular basis.

Tickler

This is an element that should be a reminder for events that should happen on a specific date. One common use of such is to create a tickler for “waiting for” status NAs to remember to follow up on a specific action. One can think of ticklers as Next Actions with an expiration date.

The used interface should treat these elements in a special way. The most important distinction is that the Ticklers that are due on today (and those that are late) should be easily visible to the user, to the point of being annoying. Another feature of the UI is that it should allow for rapid rescheduling of any tickler to some date or a fixed number of days (1, 3, 7, 14 days). There should also be an option to reschedule by a random numer of days.

Ticklers can be recurring, with a set recurrence level (a week, fortnight, month). Upon completion of such a tickler, a next one will come up after the defined amount of time.

Ticklers can be set only to a date, and not a time — they are not meant to be a replacement for a day planner but rather as complementary.

Realm

Structural element that allows for organizing all the other entities mentioned in this sector. It is designed as a global separation tool for areas or our lives that are completely separate in terms of their tasks. The most common example would be the separation of "personal" and "professional" realms.

Realms should be prominently featured in the applications UI and the enabling/disabling of these should show/hide the elements visible on screen to only those listed in the active Realms. This is a tool that will help the user focus on the types of Next Actions what he can/should actually do at this time, while uncluttering the interface.

On the technical side it is worth noting that user interaction can be one of the most intensive events, due to the large amount of elements that need to change state on screen while toggling Realms.

Context

Defines a time, space, resources or other frequently occurring circumstance that facilitate completion of an NA. A list of Actions for calling client would nicely fit into the "@Phone" context. In the personal Realm, it is nice to put things in the "@Drug store" context and bring it up next time that you are near. Coupled with some sort of geolocalization device and list of categorized destinations we could for example have a mobile application that alerts the user that he is within 100 meters of a drug store and he could take care of these tasks.

Area

An important area of activity for the user. This is a totally optional element, but worth exploring as it provides some added value in the system as a whole. The point of the Area is to help the user live a balanced and focused schedule. This is accomplished by first prompting the user to input the Areas of his/hers life that are deemed important. All project and action can then be put into one or more of these Areas. The system can then report the number of tasks completed in each of these Areas and present the NA list with the ones belonging to the most "neglected" Areas at the top.

Areas can also prove useful in a business setting. The classic managerial roles: Planning, organizing, deciding, motivating and controlling can be defined as Areas to make sure that one of these is not forgotten. Further developments may lead to the user assigning priorities to these Areas as well as having the ability to define the level of advice the system provides automatically.

3.2 Functionally Extended Application Response (FEAR)

3.2.1 What is an ultra thin client

The definition of an ultra thin client is simply this: an application that aims to minimize business logic while maintaining a high degree of interface responsiveness.

The first and most important goal is quite clear: minimizing business logic in the client itself, and hence maximizing the amount on the server, is a feature that will help minimize the costs of subsequent client applications and also their further development (upgrading).

This must be amended be a second goal: competitive interface responsiveness. The first goal could be fulfilled with only a browser as a client with all the pitfall described in section 2.3. This was the dominating model of web design since the dawn of the Internet and has since been phased out by techniques that allow for a more responsive interfaces (see section 5.1.3 for a an overview of these techniques).

The reach a point where the client is driven by the server, we should introduce an intermediate layer. Because of the chosen software architecture and protocol (HTTP), the communication from the server will be sent to the client only as a response to said clients action. This response contains, in it's body, the functional directives on what the client has to act upon. This response will be called the Functionally Extended Application Response and is the subject of the next sections.

Figure 3.2 represents the role of FEAR in the request/response pattern of the HTTP protocol. The coloured nodes represent the process of generating, processing and running the functional directives that are described in section 3.2.

3.2.2 Basic rules of FEAR

FEAR pattern/style/philosophy is a concept that entails two actors – an API (server) and a client, where the server assumes a role wherein it is responsible for a part of the clients execution by issuing an extended response. This response is called the Functionally Extended Application Response or **FEAR**. The following are the basic guiding principles of both the *concept* and the actual *extended response* – the FEAR instance.

Concept principles:

- **Opt-in basis** – the client send a response and by an appropriate header signals that it is ready to accept a FEAR.

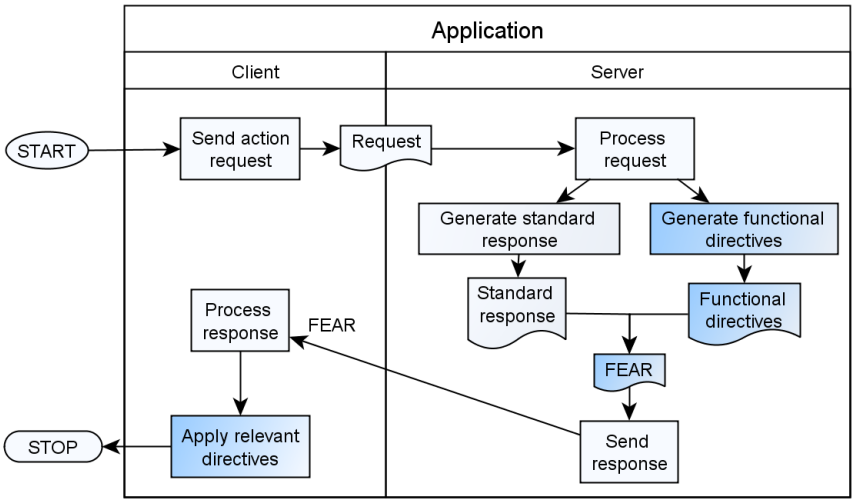

Figure 3.2: The request/response process with a functionally augmented response characteristic.

- **Directive-centric** – a directive is a primary message of FEAR, conveying a specific event, occurrence, command or suggestion.
- **Customizable** – the directives are application domain specific and can be customized between them.
- **Pragmatic** – primary concern is achieving business goals.

FEAR(instance) principles:

- **Is a response** – the FEAR is transported in the body of an API-generated response to a client request, the content type should identify the body as a FEAR
- **Externally described** – FEAR does not contain meta-information, it is described in the headers of the transporting response.
- **Client-interpreted** – a directive provides the client information on *what* happened, the client is responsible for acting on the directive and the API cannot assume anything about the clients actions.
- **JSON encoded** – is encoded in the human readable and easy to parse JSON format.

The strong suites of the style lies within its agility and ability to design software that is more maintainable, more structured and to do so with greater code reuse.

3.2.3 Functional directives in FEAR

When a client makes a request to the client, there should be two possible response types that can be generated by the API. The first is a standard response that would be generated if no functional directives were generated by the API and thus the client should not run the FEAR parsing engine. The content type of this response should be that which is given according to the data (in the prototype, the JSON data mapper is used and hence the content type should reflect that).

The second is a response that transfers one or more functional directives. The possible directives are presented in figure 3.3 and described in some detail in the next subsections. The figure represents a conceptual model of FEAR.

Figure 3.3: The categorization and specification of FEAR directives used in the prototype. (Possible collections of given entities signified by trailing braces).

It must be noted that the structure of the Functionally Extended Application Response presented here is just a suggested structure, not any kind of effort for standardization. It is not only possible, but also recommended to change the structure of the response if the application domain requires it. This is reflected in the directives themselves – some are generic and easy to fit into any application (the *msg* and *change* groups), other are applications specific (i.e. *events* group).

It is also worth noting that a document describing the elements of FEAR is a one sided contract, much like an API service method specification. The API is obliged to provide them, but clients can use or ignore them[2]. Furthermore, any client using the same API can implement and act upon any part of FEAR .

Message (*msg*)

This category of functional directives contains messages that the API generates for the user of the client. They are grouped into three categories that signify different severity of said messages. The directive contains: the actual message content (*msg*), the model name (*model*) and the model ID (*id*). The last two attributes are only set if the given messages relates to any particular model element or collection. That way the client can display the messages in places directly adjacent to the element that they are related to. In special cases each message can also contain a hash table of data (*data*).

The *sticky* and *header* attributes are additional meta-data, the first helps the client determine the intent of the message by signifying whether it should be a permanent information or whether it is ephemeral and can be discarded automatically shortly after showing it to the user. The second attribute, *header*, if it is set, defines an additional title or category for the message to be displayed before it.

Please note that like all directives, these are merely notifications of what, if anything, happened. The actual completion status of a given request is always signified via the response status code (as stated by REST style) and that should be the primary indication of deciding upon client actions that rely on that status.

The messages here relay any kind of communication from the API to the client user. The client can act as a moderator for this communication channel, showing the user messages of only a given type depending on the current modality of the interface. The client can also suppress messages in some situations either ignoring them or saving for later retrieval.

Some examples of situations when an API would return FEAR with a *msg* directive:

- *validation_failed* – data validation for adding a new item failed. Because the

[2]This is in stark contrast with the status code of the response, which should be respected by the client and acted upon accordingly

validation of the model failed on the API side, it should respond with one or more messages describing the type of validation that the item failed and possibly instructions for proper addition. Please note that the client should notice the error in the response status code and keep the yet unsaved data. The errors from the API should have the model name set via the *model* parameter and hence the client can tie the status code and errors relating to the tried model as an indication that the insertion operation failed, and display the resulting error messages near the data that failed validation. The attribute array allows for the API to tie the message to a specific attribute of the model, providing finer control and giving the client the ability to provide more meaningful visual cues to the user.

- *warning* – a similar item already exists. In the example of the prototype, a user could want to insert a Next Action with an identical name to some action that was inserted earlier. This can easily happen due to the user forgetting about already inserting this task. The system can warn the user of the possible duplication that is about to take place so that he can reconsider and maybe remember the former occurrence. The client could present this warning next to the removal button for the newly inserted action.
- *info* – user has completed a set number of actions for the day. This message might pop-up after an action has been set as completed and the threshold of tasks for the day has been completed. This message is not related to any element, but rather describes the global state. Please note that in such a case, the client does not even have to be aware of the existence of such an action counting feature (thus, it does not have to be implemented on the client side, but works from the users point of view).

Change (*change*)

This directive that some changes have occurred that reach beyond the immediate scope of the current client request. This most frequently occurs when elements are changed that are related to element being currently operated upon. Such interdependencies between entities in an application are subject to frequent change and are best addressed on the server level. Please also note that this type of functional directive is a very generic feature bound to turn up in all of but the simplest applications and hence should be always implemented.

The *change* group has directives that directly correlate to the type of change that an entity can be subjected to. All of these have target identifiers where *model* signifies the model name and *id* is the id number of the specified element.

The *update* directive signifies that one or more of the entities attributes have changed as a result of the current request, and the *new_data* array specifies a list of these attributes along with new values (if not set, the assumption should be that the whole entity should be considered as changed). The *del* directive signifies that a particular entity has been deleted, and *new* that one has been created.

An example of such a directive could the deletion of a particular project. Such a request would result in a response that would consist of n *change/del* directives with a *model* of *NextAction* where n is the number of Next Actions that were assigned to the project. The client could remove these tasks from all open views despite not doing anything directly related with deleting tasks.

Special event (*event*)

The special event directive is an application specific domain of directives that allow the API to communicate that a specific circumstance has occurred that the client should know about and be able to properly handle them.

At first glance, this can (and should) be viewed as a mechanism for introducing a new feature and hence a bit contrary to the whole concept of the API-driven client. That is because if there are specific, specialized types of events, the client must know in advance of what a specific event means and what to do with it. But there are cases in which communicating such events will save client side coding. The specific examples of such events based on the prototype give cases studies on their usage.

There is also the possibility for some of the below events to be communicated as common messages (directive type*msg/info*), but there are reasons for considering them as separate entities. See section 3.2.4 for an in-depth discussion of the topic.

The *tickler_activated* special event is a way for the API to communicate to the client that a particular Tickler has reached a state in which it should be highlighted and displayed prominently. An alternative approach to such a notification might be to poll the API every predefined amount of time checking for active ticklers. This technique however saves on data sent, as the request is given only when a user action is performed (signifying that the user is interacting with the client). The important thing to notice is that the API controls when that directive is sent, giving the opportunity to fine tune application behaviour without the need to modify the client.

area_neglected is an event that could signify that a particular Area has reached a threshold of negligence and that something should be done about that. Because the process of calculating the "negligence level" can be a very complex one and very prone to change it is again important to note that the API controls all of said mechanisms and the client only handles the resultant actions that need to be taken.

3.2.4 *msg* or *event*

When considering the two directive groups of FEAR, it should be noted that they can overlap and the decision if a given circumstance is an *event* or a *msg* is not immediately evident.

The primary advantage of the *msg* directives like *info* is it's generic nature and high chance of implementation on the client. Because of that it's basically guaranteed to be processed and communicated. This provides a lot of flexibility for future upgrades and the ability to convey messages about previously unforeseen features to the users without the newest version of the client.

This generic approach however does have a drawback – it has no way on convening data about the message itself, and therefore the client has no way on acting on a specific message, because of the lack of perceived semantic meaning (apart from the less-then-reliable text analysis techniques). The only meta-data the message has is its specific type within the *msg* group.

Would adding meta-data to specific messages help with this? Yes and no. If the message is generic to applications in general, it should be included in the *msg* group with additional meta-data. An example of such a data-augmented message is the *validation* directive. It represents a very common case in which the validation of and added or updated entity failed on the server side. For the client to present the validation failure messages in a meaningful and ergonomic way, it needs information on specific messages relating the attributes. This is clear extended data related with the validation messages.

This is, however, a very common case and should be treated as a standard level message.

To better understand the difference between *msg* and *event* groups, let us analyse the *event* group directives for the prototype in terms of why they are not advisable for the *msg* group.

At first glance – all of the special events seem to be messages – they all convey a certain occurrence within the system itself, whether it is a Tickler or Context that has become active, or the warning of a neglected Area. However, to be useful to the user these messages in themselves need further data. Let us consider the *tickler_activated* directive: when a Tickler becomes active and this would be instead a *msg* with no additional data beyond the text of the message, all the client could do in that situation is display the message (that would probably suggest to the user to use the interface to reach the newly activated tickler). If, however, the directive contains additional data about the type of event that happened, the client can display some familiar icon depicting an active Tickler. So – a *msg* group directive with additional meta-data, not unlike that of the *validation* directive, would serve its purpose.

That is the reason for the other criterion: such messages should be taxonomically separated if their messages resides in the specific application domain. The first reason for this is that such messages can easily be distinguished from the basic lower level messages belonging to the *msg* group. A second reason: *events* do not always have to be messages that should be displayed to the user; they can be events that only the client should respond to and the user does not participate in such a communication.

Another, more distant reason is that such a separation can be used in the future to build a generic FEAR level specification, where the *event* group would be the only name space that is open to modifications by the application.

The summary of the above decision process is illustrated in figure 3.4.

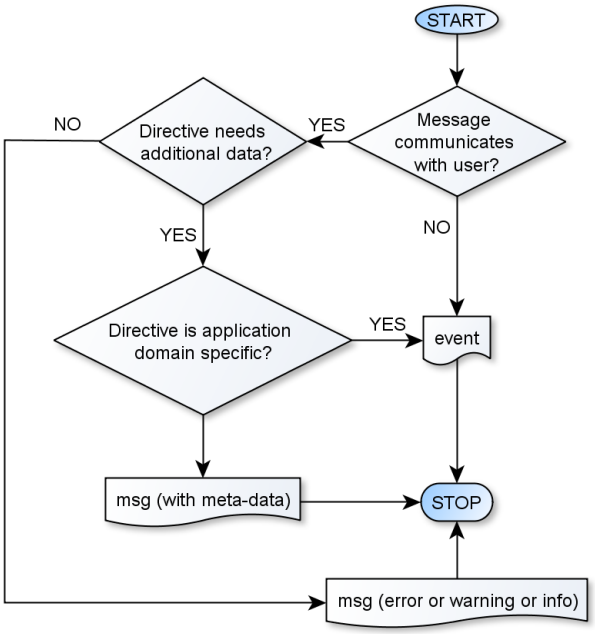

Figure 3.4: Partial directive classification decision process.

3.2.5 Summary

The FEAR philosophy is a pragmatic, simple concept with potential advantages if used correctly and for the tasks that it was supposed to accomplish. This chapter has introduced the basic tenets of FEAR and provided with a skeleton specification of the directives that can be used in the prototype, along with the rationale for including them that can be used as a starting point to building further specifications.

Chapter 4

RESTful APIs

This chapter gives an overview on the crucial architectural style underlying the FEAR pattern – REST – what is it, where does it come from and how it can be leveraged to build APIs. There is also a discussion on the benefits and disadvantages of building a REST API compared to other competing technologies and concepts.

4.1 About REST

This section is be about the REST architecture: what is it, and what it means when an API is "RESTful". There is also a discussion of whether the API implemented as the prototype strictly adheres to the REST guidelines.

4.1.1 What is REST

Representational State Transfer (REST) is an architectural style devised by Roy Fielding in 2000 and formed by applying a set of constraints to elements within the architecture. The basic[1] constraints defined in Fieldings dissertation are [22]:

- **Client − server** architectural style
  The concerns of the client and server are separated, improving the portability of the client and decreasing the complexity of the server components. It also allows the
- **Stateless** Each request from client to server must contain all of the information necessary to understand the request. There should be no need for the service to keep users' sessions; in other words, each request should be independent of others.

[1] The optional "Code-On-Demand" constraint is omitted as not relevant to this thesis.

- **Cache** The underlying infrastructure should support caching, preferably at different levels. Data within the response should be labeled as cacheable or non-cacheable. This allows for increased efficiency and client-perceived performance by limiting the time of a series of interactions.
- **Uniformly accessible** elements Every resource should have a unique, identifying address and can be modified through its representation. Furthermore, messages should include enough meta-information to allow for their processing.
- **Layered System** The system architecture can be comprised of several hierarchical layers, which should be invisible to client so that it just accesses one layer.

4.1.2 Primary characteristics of RESTful web services

Below is a list of "practical" characteristics of an API designed in the REST style.

Resource URIs

All resources should be available via a location specified by the URI. A resource URI in REST loosely translates into Model in the MVC pattern. Resources can also be a specific collection of models of the same type or a specific variant of model. For example, the prototype exposes Next Action models under the ⇒ `/next_actions` URI while also exposing them under the ⇒ `/next_actions_waiting` URI, the second URI points to a list of models that have their status set to 'waiting'. Because of the routing role of the Controller in MVC, it is responsible for generating a response from the URI.

The REST style does not provide any strict guidelines as to how these URI should be constructed, but there is a de facto standard that has emerged amongst the applications implementing the style. The popular convention is shown in table 4.1.2.

The URI can be used as a way if signifying which resource should be operated upon while the request method signifies what should be done with it. The rows in table 4.1.2 signify the type of operation mapped the standard HTTP request methods.

Using HTTP standard status codes

When a client sends a request it expects a response depending on the status of the operation sent. When not using REST, this is often done by sending a response with data about in the body that specifies the status of the request. RESTful APIs use the standard HTTP response code to convey the status of the request. Additional response data can be sent in the body, but it's generally a description of *what* happened rather then *whether* something actually happened (the status of the response).

Resource	**Collection URI, such as** http://app.url/next_actions	**Element URI, such as** http://app.url/next_actions/1
GET	**List** the URIs and perhaps other details of the collection's members	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.
PUT	**Replace** the entire collection with another collection.	**Replace** the addressed member of the collection. If it doesn't exist, this will fail!
POST	**Create** a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	Treat the addressed member as a collection in its own right and **create** a new entry in it.
DELETE	**Delete** the entire collection.	**Delete** the addressed member of the collection.

Table 4.1: Performance at peak F-measure:. Source: [24]

Meta-data in headers

A REST API should be agnostic to the content type that it can serve provided the content can be transmitted by the protocol. Also, single resources can have multiple representation types. In the prototype, some resources can be transmitted as HTML markup or pure data in JSON format. The client should therefore specify the type of content that it accepts and also the type that the response is in. These standard tasks are achieved by standard HTTP request headers, for example by setting a *"Content-type: text/html"* or *"Content-type: application/json"* to the response by the API in response to an analogously set *"Accept"* header. There are several other standard headers that are used in APIs, a reference of which is the HTTP standard (RFC 2616)[23].

Besides standard headers, there are also tendencies to append additional custom headers to requests and responses. These are usually prefixed by *"X-"* and are to be used to convey meta data in the communication between API and client. An example of such a custom header would be to indicate the version of the API to be used by sending a *"X-API-Version: 1.0"* header. The custom header approach is sometimes considered to not be reliable due to the fact that custom headers can be stripped by proxy servers between the client and the API.

4.1.3 Is REST a standard?

Due to the widespread interest that Fieldings dissertation has drawn REST has become a "buzzword". On one hand there are many web services that claim REST compliance despite not conforming to the constraints defined in section 4.1.1. On the other hand,

some people try to "fine tune" the claim made in the dissertation as a logical consequence of said claims. Both of these approaches are chastised by Fielding as violating the spirit of his work[25][26].

There are voices in the discussion that suggest making REST into a standard, to be able to better control which implementation can actually use the acronym and what that constitutes. Fielding himself is opposed to this idea, because "To the extent that such a document is possible for an architectural style, that document is my dissertation [...]" and "you will get is a committee that has some vague notion of what they consider to be design to write down the least common denominator of misinformed "best practice" based upon whatever Microsoft chose to implement in its last release"[25].

These discussions have repercussions for this thesis. Based on the explanations Fielding has given over the years, clarifying the statements from the dissertation, it is apparent that he wants to keep the definition of REST as an architectural style as strict as possible. There are some elements of the API proposed in this thesis that might be contrary to Fieldings vision. Some of the controversial features (or lack thereof) are fixed resource names.

Fixed resource names are not currently supported because the version of the API given does not have a feature that would allow it to instruct clients on how to construct appropriate URIs. Fielding considers this to be a domain-specific standard, equivalent to server-client coupling[26], which he deems to be in a serious violation of one of the base premises: the "Hypermedia as the engine of application state". These features might be implemented to the API in a later iteration, but the Agile methodologies for the project set working software as a priority, and hence will be delayed until there will be practical reason for adhering to the rules devised by Mr Fielding.

That is why, with respect to the opinions of the author of REST principles, the API built as a prototype for the use of this thesis cannot be formally described as RESTful, but rather built in a style that is built upon REST. For simplicity however and due to the fact that at some point in time, the prototype is to be compliant with the above described principles the thesis continues to refer to the API as RESTful.

4.2 Alternative conceptions

This section focuses mostly on the differences between alternatives to the choices presented in this thesis. These are:

- Choosing a non-REST web service API implementations.
- Choosing an alternative back-end strategy, including the client-only strategy.

4.2.1 REST vs SOAP

Most of the advantages that are described in this thesis when talking about a back-end REST API also apply to any web service that can respond with a FEAR message. The most popular standard for building web services besides REST is SOAP. The subsection gives a quick overview of the two and provide reasons for choosing REST in this thesis.

Let us first focus on the shortcomings of REST in comparison to SOAP.

SOAP is a protocol that is an acknowledged standards and has established guidelines for creation of web services. REST is more of a philosophy with a wide range of potential applications, often misunderstood (see section 4.1.3). Due to SOAPs status as a de facto industry standard, and also due to the predictable nature of web services built upon it has a thriving ecosystem of software, tool chains and supporting standards. REST services are more heterogeneous and hence cannot be supported in a similar fashion.

When building an API in REST, the more advanced features of the system are usually designed and implemented onto the system by the programmer using an established pattern[27]. The example for this would be securing web services. SOAP has the advantage of having an open source server that support the WS-Security standard out of the box. With Apache Axis2, SOAP services can be secured in a standard and widely understood way. A REST API could also use the features provided by Axis2, but it would undermine some of its most significant advantages, and would be a path less travelled and hence more cumbersome. A more standard way of securing REST services is by using two-legged OAuth. This however is more of a pattern that an industry standard and hence the programmer is left with implementing it himself or using a third-party library.

Another scenario when SOAP is better than REST in when the service model is oriented around a large amount of complex data types. The WDSL standard and various tool chains allows the developer to easily disclose the available data types and services that operate on them. The client applications can then be developed using tools that generate code from this WDSL, giving the developer a framework to work on. REST does not have such a standard for service data exposure and hence has to be supplemented with documentation.

In this thesis however we are focusing on systems that are quite different from the ones usually built with SOAP. The proposed system:

- has a relatively simple data model
- is meant generally as a private solution and hence does not need enterprise level security
- is meant to be operated chiefly by the web interface and not to act as a standalone web service

The simple data model allow for transfer of data through a lightweight data mapper: JSON, which has very little meta-data overhead. Other data required for operation is sent through the standard protocol headers. This makes for very efficient transfer of data between the server and client, which is very important when designing an API that is not just a web service but also a back end supporting constant communication between server and client.

The development of clients for SOAP services can be cumbersome when doing it on a platform that does not support standard tool chains. Usually when building SOAP clients, the framework code is generated by a tool from the service WDSL and then worked on by a programmer. When the tool chain is missing, the task becomes cumbersome.

A more serious problem could be that not every client development platform has libraries to support sending SOAP requests and parsing the results. Writing a proprietary driver for SOAP due to its complex nature can be a time consuming task, while a JSON parser can be implemented fairly easily (there will probably be no need for that – the JSON web page provides links to JSON parser libraries for all non-niche programming languages[28]).

For all these reasons, the prototype API is built using the REST style and not the SOAP protocol. However, when building the application it is important to consider the particular application and weigh the options and applied to that application in particular.

Table 4.2.1 provides a simple comparison between the two techniques used to build web services.

4.2.2 API vs external RDMS

When considering how to build a central data store, the option of an external database management system is an option. The central storage would be a database management system, such as MySQL, PostgreSQL etc. The clients would all be connecting to the database and operating on it without any intermediate layer.

The first limitation that comes to mind is the lack of and object oriented programming language on the server side. This would mean that all the application logic that cannot or isn't feasible to be implemented in the database would have to be implemented in the clients themselves.

Depending on the RDMS in question, the amount of application logic that can be implemented will vary. For example, while SQLite allows for only a simple amount of programming logic (like "views") while Oracle supports functions, methods and object oriented interfaces that would allow for implementing a large amount of said logic.

The amount of logic implemented in the database itself is inversely proportional to the amount that will later be coded into the clients themselves. This means that every

Feature	REST	SOAP
Type	software architecture style	protocol
Message format	Usually JSON is used as a data mapper.	XML
Transport protocol	Any. Usually HTTP.	HTTP – less frequently SMTP, JMS, AMQP or UDP.
Data overhead	Format dependent – usually lightweight format like JSON with very little overhead	Large – data and procedure calls are encoded by XML. Furthermore, messages have non-optional wrapper XML nodes like Enveloper, Header and others.
Security additions	None / implementation dependant.	WS-Security is an established standard for securing web services. Due to the widespread adoption of SOAP in enterprise scenarios WS-Security is a good option when there is need for end-to-end security (message level security).
Incompatible services models	Stateful service model, complex custom data types.	None.
Service description standard	None.	WDSL.

Table 4.2: Comparison of REST and SOAP

unit of application logic implemented on the server side will save n units of client creation and maintenance time later on where n is the number of clients.

The problem with the aforementioned approach lies in the limit that the database can handle application logic. Because databases are designed to handle data manipulation logic and not application logic as a whole, there are several areas of development that the database cannot implement. This would mostly involve binary file processing, invoking external processes and connecting to external data source. Some good examples for a task management system would be:

- sending emails with reminders about tasks,
- creating reports in binary file formats (PDF, PNG),
- connecting to and external geolocalization web service to get potential places for a task context

Another problem with that approach might be the limited availability for database drivers for all the client platforms. The extent of this problem is determined by the platform and the RDMS.

Therefore, while it would be possible for a limited number of simple approaches when using a fairly mature and feature reach RDMS, this approach is not a reasonable solution for most applications.

Chapter 5

Prototype implementation discussion

This chapter describes the implementational details of the various elements of the prototype and provide the reasoning behind choosing specific technologies, tools, platforms and programming languages.

5.1 Tools, platform and languages used in prototype implementation

The section gives a detailed account of the project managements methodology behind the implementations of the prototype, as well as the implementation tools for all the prototype parts.

5.1.1 Methodology

The decision in terms of software development methodology should be first considered between the two major general groups of software methodologies: heavyweight and lightweight (agile)[29]. Considering the premises of the prototype described in section 3.1.1, we can observe several elements that would suggest that adopting a lightweight methodology would be the proper step[30][31]. Those are:

- Project documentation is redundant because of the combination of the following:
  - the client, project manager and developer roles are all fulfilled by the same person,
  - the implementation phase encompasses a relatively short time period (a few months),

  - to the extent that the prototype needs to be documented, this thesis fulfils that need.

- Requirements for the features, implementation details and priorities are prone to changes depending on the current state of the thesis.
- A working prototype is crucial for evaluating certain claims on a regular basis (which correlates with the frequent, working software as a measure of progress philosophy[32][31]).

After choosing the lightweight methodology group, a reasonable approach would be to review the available methodologies in that group, choose the most appropriate one, and make additions/alterations for project-specific aspects. When browsing and comparing the different methodologies [32][33] it is apparent that almost all of the distinguishing factors between them relate to areas that are irrelevant for this projects. These differences relate to the following areas:

- communication between teams, members and clients,
- responsibility for various aspects of the product,
- organising teams,
- timetable rules.

Considering the above, we assume a methodology that is a simple implementation of the Agile Manifesto[32]. Fine grained techniques and rules on developing the project might be implemented later as the project evolves.

5.1.2 Back-end framework

Application framework - Ruby on Rails

Rails a modern web framework based on the Ruby programming language. Since it's creation in 2003 it gained widespread adoption as a productive, intuitive and comfortable framework. It's main features are:

- strict MVC pattern compliance[1] – due to the multi-platform assumptions for this project, it is very important to be able to provide different views for the same data.

[1]It must be noted that it is possible to violate the MVC pattern by putting business logic in a Controller or View. Rails does not strictly enforce the MVC pattern – it provides a set of tools that allow for comfortably working with the pattern, but also to break it when the need arises

- REST as the underlying pattern – this is the main reason this framework was chosen. Rails uses the MVC pattern where the Models and Controllers are used for back-end development while the Views are used only when there is the need to build a web client. REST is a vast topic and is discussed at length in chapter 4.1.
- Convention over configuration – this is a development pattern that places speed of developing standard components over the pattern of comprehensive configuration. Rails provides "sensible defaults" for file paths, class names, ORM mapping and basically every other feature of the platform. Coupled with a complete set of tools for generating basic elements of the framework, it allows for building standard, working APIs and web applications in just minutes – the developers role is to provide the added value for such an application.
- Created for Agile Development – Rails provides a set of tools that gave it the opinion as the ideal tools for developing with adherence to rules mentioned in the Agile Manifesto, merits of which for this project were mentioned in section 5.1.1.

Database management system - MySQL, PostgreSQL, SQLite3

Development is based on "Active Record" – the Rails ORM hat allows for using the application with a wide range of database systems. The supported databases are MySQL, PostgreSQL and SQLite3, and hence are recommended for maximum stability. Other database systems are also known to work well on Rails.

5.1.3 Front-end framework

The front end for the web application is built using a collection of tools, languages and frameworks.

Rails

The Rails application framework comes with default HTML views that make it easy to develop a "traditional" web application (see section 2.3 for a definition). This feature also allows for multiple Views (with the use of the Controller), that can be used for various uses. In this case, we use the Views feature for embedding Backbones.js templates.

CoffeeScript

CoffeeScript a language that compiles into JavaScript. The goal of the language was to make programming JavaScript more concise, clean and to make it more productive and

akin to that of modern programming languages (it was heavily inspired by Ruby and Python)[34]. The most important features of the language are:

- indentation specific syntax – enforces proper coding style,
- context isolation – helps avoid problems with implicit global variables,
- classes and inheritance – allows for defining class constructs in a more concise manner similar to other programming languages
- loops and comprehensions – implements mechanisms known from Ruby and Python to assure more concise and expressive syntax for iterating over collections

Because the Rails framework from version 3.1 up supports CoffeeScript as a default client-side scripting language, its usage is a natural choice for this project.

Backbone.js

Backbones.js is a JavaScript based framework designed to bring structure into client-side application logic and to enable writing REST API backed single page applications. Due to the differences in usage, this does not directly translate to the MVC model used in the back-end. Below is a summary of the most important classes supplied by backbone, their relation with the MVC pattern and their role in this project:

- **Models** – maps to the Model in MVC. Backbone's philosophy states that this object could contain application logic, validation etc. In this projects, models usually contain only dynamically assigned data (hence there is no predefined data schema, which would force the need for synchronizing the back- and front-end Models), and to used as an element of a Collection (see below). The Model is also responsible for sending proper requests to the assigned URL in case the models need to synchronize state, which gives it the responsibility usually reserved for the Controller in MVC.
- **Collections** – collections of Models in the MVC pattern are usually handled by the Model itself, usually with class-level (as opposed to object-level) methods like *find, findAll* in the Active Record pattern; in backbone, such responsibility lies in Collections. This element, akin to the Model, also has some responsibility of the Controller, because given a proper URL and Model class it will generate a collection of these Models. The Backbone library also provides many utility functions for handling common collection operations (sort, filter, compare).
- **Routers** – maps into the Controller in MVC. It is important to understand the distinction between the role of Router in Backbone, which is to call certain actions based of the path in the URL (be it a proper path or hash-marked string) and

to change the given URL based on set criteria. The role of the Router is **not** to communicate with the back-end API [2], which is handled by Backbone Models and Collections.

- **Views** – map directly to Views, these are usually rendered by Router actions. A View object will usually contain a Model or Collection with the data, and the template name to be rendered by inserting proper data into it. Backbone provides a flexible template system, which can also be augmented by external template engines. On this project, the standard Backbone templates will be used.

The typical flow of application logic for a backbone based application is shown in figure 5.1.

First – the application entry point is reached, at which point the back-end API processes the response depending on the headers. Because we are describing the client application flow, we will assume that the request will accept HTML markup. The back-end generated the HTML markup and embed JavaScript code that will set up the application by first initializing classes and then objects of the Backbone elements: Models, Routers, Views, Collections. The API will then get the necessary data required to display the initial elements, usually in the form of Models and Collection being fed database data.

The generated elements are then ready to begin the client side parsing, the Models and Collection are passed into Views and the main application Router calls the Views render method to display the proper markup to the user. Embed in Views and Routers are the rules for handling user interactions in the work phase.

The "work phase" is the time span between the first application display and the end of the interaction with the application and constitutes a series of user actions. An action in this example could be a simple request confirming a data update. Upon this action, the Model or Collection should interact with the API in order to synchronize the data on both sides. This is accomplished by sending an asynchronous request to the API, with a REST compliant resource URI and appropriate method.

The API processes the request and generates a response with appropriate data. Usually this will contain the status of the operation using a standard HTTP code and the data (if any) that should be updated on the side of the client.

The response data is then processed by the Router, updating the affected Views. The Views might defer some update logic to the Model or Collection when there is something beyond the standard update of markup (generating new elements for example)

.

[2]Which is the reason the Backbone team changed the name of this class – it was named "Controller" in earlier versions of the framework

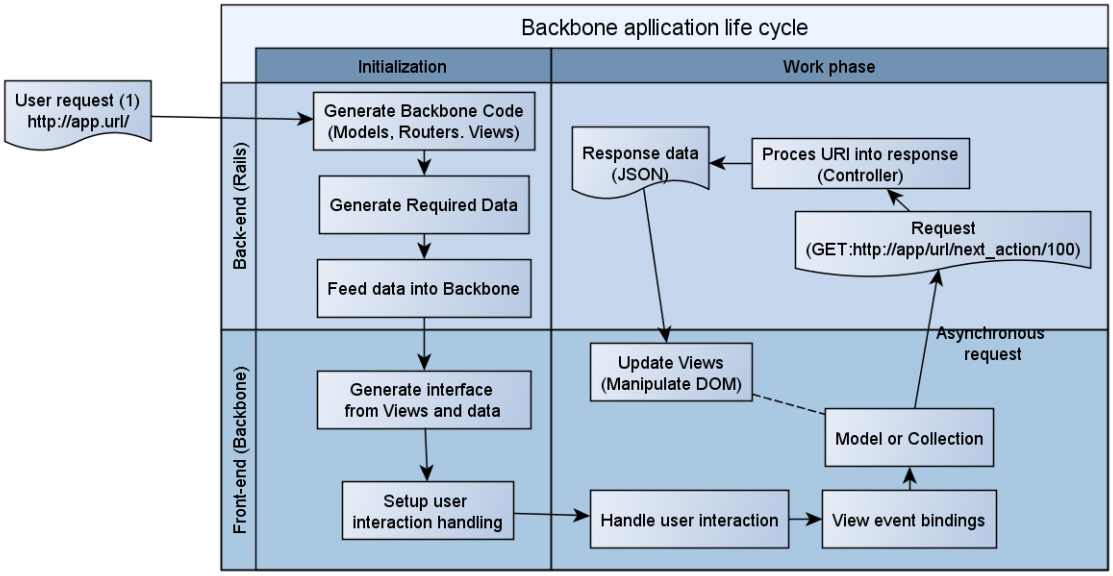

Figure 5.1: Backbone.js based application life cycle

The key innovation in this thesis in the application flow is not in the cycle itself, but in the extent of data that the API returns and how it is parsed by the client side. This is referred to as the Functionally Extended Application Response (FEAR) and is explored in depth in section 3.2.3.

5.1.4 Mobile application – Android OS

The mobile application part of the prototype is built as an application for the Android operating system with the use of the standard Android Software Development Kit. The mobile application is designed for Android version greater than or equal 3.2, which provides the ability to use all modern system features and programming libraries. For an explanation for these choices please refer to section 5.2.3.

The only external library used in the prototype is GSON – a JSON-Java data mapping library for communicating with the API.

The application provides a large subset of the web client functionality, most notably managements of Next Actions, Projects and Ticklers. It is however worth noting that maximizing functionality is not the aim of this client, as it is a companion. A minimal version of the client was considered in the preliminary analysis, one that would allow only limited management of only one entity – the Next Actions. This concept was rightly abandoned due to the possibility of insufficient complexity for discovering certain architecture characteristics.

The mobile application as of yet does not make use of any device specific functionality (camera, GPS, accelerometer etc.).

5.1.5 Examples of mobile and web client interfaces

Figures 5.2 and 5.3 provide sample interface views of the clients compared. The figures do not present an accurate representation of the size of said interfaces, because of the different pixel densities that the devices operate upon and the general modality differences of a web browser and mobile operating system.

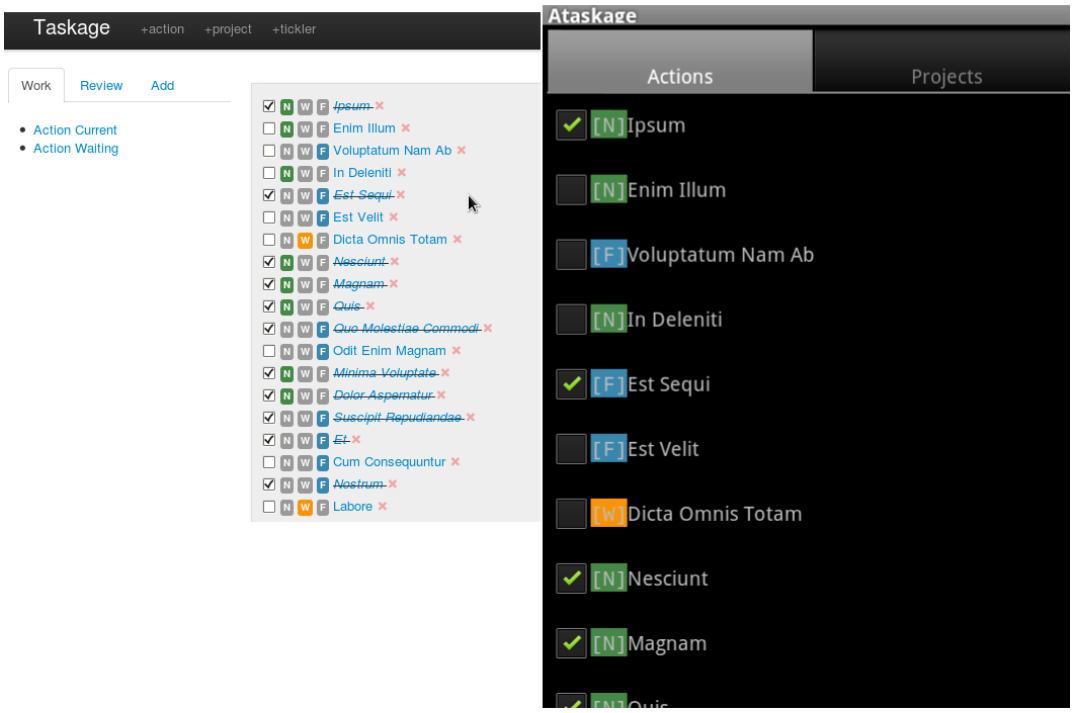

Figure 5.2: Next Action list interface as seen in the web client (left) and mobile client (right).

.

.

5.2 Alternative techniques and technology stacks

This section introduces and briefly describes the alternative implementation options considered and the process in which they where eliminated. The last subsection touches upon the tools used in working with the user interfaces of the web client and if they should be analysed for alternatives.

5.2.1 Back-end implementation alternatives

As alternatives to the solution chosen in the thesis, the following technologies, languages and frameworks were researched:

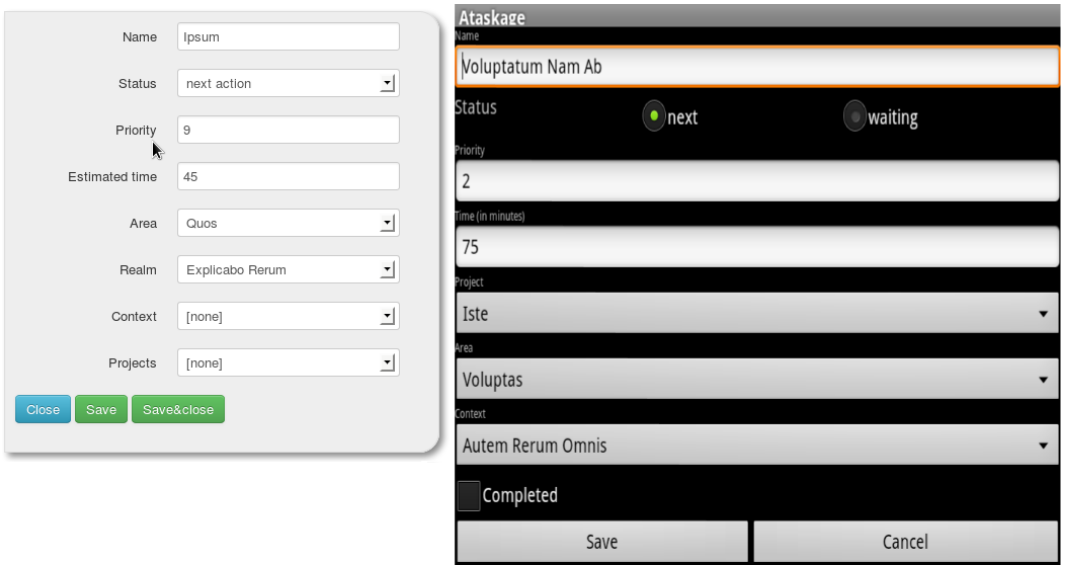

Figure 5.3: Next Action edit interface as seen in the web client (left) and mobile client (right).

- PHP and Zend_Rest or Slim
- C#/C++ and Asp.net – OpenRasta or plain
- Python and Django, Pylons
- Ruby and Sinatra

All of the aforementioned can be compared to the advantages of the Ruby+Rails combination described in section 5.1.2. The below subsections define them and describe them at length. It is worth noting however that the investigation for choosing an implementation technology was not convened in any objective way and was based mostly on discussion on the Internet, with the added filters of personal prejudice and technological curiosity.

Implementation was not crucial to the solutions introduced in this thesis, and although it used as a tool to measure its feasibility and overall introduced results the particular should not lie upon the chosen technologies.

The following are the main themes that were present in Ruby on Rails that promised gains in terms of development time and flexibility.

REST support

One of the premises of the prototype was an REST API back-end. Choosing a framework that has native support and is aimed towards REST based APIs would substantially decrease the development time needed to develop the API. Ruby on Rails does a great

job at this, because from the very beginning it was aimed at creating a combination of a RESTful API back end and a web-enabled client. All generated scaffolding is developed in such a manner that the default web client interface is just a client for the underlying back-end. In the end, most of the view portion provided by the framework was ignored as the user interface was built with Backbone, but the in place mechanisms for connecting models and exposing them as JSON data under a specific URI has proven very helpful.

Standard entity operation support

In any project involving any kind of long-term persistent data storage there will need to be code designed for interfacing the application entities with the given storage engine. The prototype was written with standard relational databases in mind, but also, given its future as an open source project, to be able to run on as many database engines as possible.

Due to the fact that there already exists a variety of ORM libraries for any programming language, it would be suboptimal to not use one of these solution when the needs of the application are not beyond common database use cases. Rails has a built in ORM out-of-the-box which is also connected with other framework features, such as generating scaffolding (Model-View-Controller set).

By using these mechanisms, it is extremely easy and efficient to generate boilerplate code and hence decrease development time.

Test framework

Considering that the project is built as a prototype and also considering its future as an open source project, it is safe to assume that will undergo a lot of refactoring and structural changes in its lifetime.

In the prototype phase, the project has been substantially restructured during its lifetime to accommodate changes that arise in the course of thesis development. Also – some changes were artificially implemented to check whether a given solution to the problem is better.

In the later stages of the project its development will be continued on a "as needed" basis, adding and changing features to better accommodate the needs of the users.

All these changes lead to a lot space for instability and errors in execution. An ideal solution for this is the development of code that tests the usage cases for the application. These tests will then be run on a regular basis to check that the changes made have not had a bad influence of existing code. Unit testing, when used properly, is a widely recognized technique for increasing code quality in projects that are especially susceptible

to changes[35][36][37].

Rails provides a built in testing framework, test generators and support for Continuous Integration tools. All this allows for quickly building, running and integrating tests.

Availability of external libraries

The characteristics of the project suggest, that the application will have to quite versatile in the assortment of tasks it will be required to handle. Even the basic web interface shown in the prototype requires the following features:

- CoffeScript support – because it is a compiled language, the set-up for development with its use would require setting up command line tools that compile files that changed since the last compiling, copy then into the required directories and so on.
- Backbone support – including the Backbone library into the project
- Twitter bootstrap – the web client uses Twitter bootstrap project to have a nicely looking and usable set of components.

The Ruby Gem repository is a directory of plug-ins that are developed by the Ruby community as packages to fulfil a specific function. Because of the vast numbers of these "plug-ins" there is a big chance that the functionality that we seek to implement in the project already has been implemented as a Gem. The three basic features mentioned above were all available as Gems, so that their implementation was limited to a one line statement in the Rails "Gemfile" and, if needed, configuration. This has tremendously limited the amount of time needed for their implementation.

And the features described above are quite simple ones where the time saved is not that impressive. In the future planned functionality will encompass some to all of the following features that can be implemented by Gems:

- PDF, image and chart generation for reports on tasks,
- connection to a geolocalization API (e.g. Open Street Maps),
- user authorization,
- connecting to social media engines.

By utilizing these ready for deployment feature sets, significant programming effort can be saved.

5.2.2 Web client tool alternatives

The tools used to build the web client, particularly Backbone and CoffeScript can obviously be used with other back-ends as well. The tools were chosen mostly on the basis of familiarity and also general consensus that they are solid platforms for web client development. Because CoffeScript is more of a utility than a framework (it only provides a syntax for JavaScript that some feel superior) it an be omitted altogether without any consequences.

There are some JavaScript application frameworks very similar in design to Backbone.js. The most popular are "Spine", "Knockout.JS" and "Batman.js". The differences in these tools mostly exist in what kind of data the project is working with and what kind of support the developer needs. This thesis does not further discuss these tools, as the specifics regarding the implementation of the web client are not that important to the overall results.

5.2.3 Mobile application – Android + Android SDK

The choice of platform was based upon the authors personal preference, knowledge and logistical readiness for developing for that system, the lack of which could generate a lot of attribution errors during the development process. In other words – the pre-existing experience of the author with development on this platform provides a maximum level of variable isolation due to the fact that the FEAR paradigm is the only distinguishing factor between this and previous development projects.

A similar reasoning process led to choosing the Android SDK. The alternative choices were already explained in section 2.2.2 and one of the more popular operating systems, all of the alternatives mentioned there support Android development. However due to the fact that the author has not had any prolonged experience with any of those solutions, the best choice is to use the default development tool kit.

Chapter 6

Lessons learned

This chapter contains the description of the process of testing the proposed solution of the thesis on the developed prototype and it's consequences. During the testing process there were several unforeseen obstacles and problems that had to be overcome. Developing a prototype also led to a clear list of advantages and shortcomings of the proposed solutions.

Section 6.3 focuses on possible consequences of this thesis in the aspect of software that can benefit from the solution described within it.

The last section gives a summary the possible development of this thesis and the proposed solution by describing some of the subject vectors that would benefit from further exploration.

6.1 Obstacles in implementation

This section describes the obstacles that are relative to the solution proposed in this thesis that have been encountered during the implementation of the prototype.

Where applicable it also briefly describes the chosen solutions devised to solve these problems, along with the rationale behind them.

6.1.1 Entity client side updates

FEARs are used to alert the client of changes made in the data that are out of the immediate scope of the current client operation. A good example of this is notifying the client of the changes to the entities made by the latest client request.

In the prototype a next action can be displayed in many places at the same time. The client should only update the view that caused the change in the first place – all other changes are handled by the FEAR parser.

This leads to a problem of multiple entities that need updating. The client using

Backbone.js does have a model/view separation, but JavaScript language limitations and construction pragmatism[1] lead to the entity being instantiated multiply in many forms – usually a few models, collections and representational elements in the form of DOM nodes.

The chosen solution to this problem was the update only the DOM elements relating to the entity. In the prototype every such node was generated from a Model by a View. To know that a particular node relates to a particular entity, data regarding the model and the id is attached to it upon creation by the View.

When a fear response notifies the application of a change in the given entity, the client parser seeks out any DOM elements with the matching model type and ID. Thanks to the embed data about the View that created the node the whole creation process can be repeated: a new View of specified type is created. it is injected with a new Model of specified type and data passed down by FEAR. Lastly, the node is replaced by the result of rendering the View – the new up-to-date DOM node.

This obstacle is less of an issue in the mobile client due to different programming paradigms. Contrary to a web page, where DOM nodes are the primary means to display entities and have to be artificially augmented with programming constructs (as in the above solution), Android interfaces are structured Java objects of specific definable classes that are held in memory when they are displayed and are also easily augmented with arbitrary data.

6.1.2 Error message placement

When an error occurs of the side of the API, there is the natural tendency to try to use the API to display the error in the relevant place – to minimize the client-side code. But such behaviour can lead to overcomplicated data structures that are be of limited use and require a significant amount of coding on the client side.

The question of what is feasible to centralise is a constant theme in this thesis and as in most cases, the answer is not clear.

The case study of the error placement paradigm presents us with an argument against to much centralization and is presented below.

Let us consider a very common occurrence – data validation. When an data model is sent to the API, and the server-side validation fails, presenting an error to the client, what should happen? How do we mark the places that would need to be changed by the API in order to display the corresponding error messages.

[1]It must be noted that a carefully thought out architecture may limit or even eliminate the extent of this problem, but it may come at a cost that would be not worth pursuing.

The answer would probably involve some kind of identification data that would be sent by the client to the API and then said data would be sent back. This round-trip data would allow for loose coupling in terms that the client would be able to "fire-and-forger" the request, trusting that the API would give sufficient data back to re-construct a view in which the validation errors should be displayed.

The potential gain of such an approach would be the ability for the client to handle more logic on it's own, while maintaining consistency with the API. This would be especially useful in that the client could send a request to the API and then act as if the request was a success, without waiting for the response. This would allow for a more responsive design when connectivity in an issue or if response generation takes an unusually long time.

The problem with this approach lies within the framework that needs to be written on the client-side to handle arbitrary generic events sent from the API. The alternative approach is to just wait for the response and wait accordingly. In both the web client and mobile application, the latter is the preferred approach. There are however differences. The web client has the added advantage that a lot of interface elements are displayed at a time and the user is not forced to wait until the request has finished to start interacting with the client. Mobile applications due to screen limitations usually present one view at a time, and hence blocking said view is equal to blocking a whole application. This can be circumvented by a combination of specific platform abilities (the AsyncTask and Intent classes provide the necessary framework in Android).

6.2 Advantages and disadvantages of the chosen solution

This section provides a list of the conceptual and empirical characteristics that define applications developed using the fear solution in terms of their strengths and weaknesses.

6.2.1 Advantages

Tested, ready API for developers

One of the primary reasons for creating an API for a given application is to provide independent developers and clients a way to access the data and services offered in a way of their choosing. This is often a process that is an "add-on" to the original application and is quite costly to the developer. Despite the resources needed for building additional APIs – it is done very often. There are several business reasons for building APIs.

By allowing people to easily augment the application with the functionality they miss, the total feature set of the application increases with every feature added by an external application. By monitoring this process, the original developer can get a good feel of what the original application need to fulfil the needs of the users.

Interest in the service as a way to implement other feature sets can lead to an explosion of following and a major marketing advantage, as was the case with Twitter[38].

If the application API is commercial and the user needs to become a client to use it – such additional applications translate directly into sales.

Also – if there is a platform that the original developer does not want to make a client application for (due to lack of expertise or resources), the API can be an indication to other developers that it will be easy to implement it themselves, thus increasing the possible number of uses for the application. This is especially important for open source projects.

The aforementioned reasons compel companies and organizations to add APIs to their existing products despite the effort needed, not only for development, but also testing and maintenance. In the prototype, because the API is built as a foundation for the base client application, it is already tested and hence needs no additional resources.

Isolation and decoupling

Isolation of specific application modules – decoupling – is a very desirable software metric. Decoupling is a prerequisite for unit testing and allows for entities to be operated on with no knowledge about other entities in the system. The level of decoupling reached by exposing RESTful resources allows for some interesting options.

One ability that comes to mind is the construction of specialized lightweight client applications. In the prototype, we could imagine a super compact client that only works on a small subset of the application ecosystem – the Next Actions (see %ref%). It is perfectly feasible to build such a client that operates on only one resource, ignoring the attributes that are not used [2]. This client would only display current actions and allow the user to mark them as completed.

This is in stark contrast with, for example, sharing the model as part of a library in a programming language. Because the Next Action entity is connected with many other entities, the client would have to at least the definitions of these dependent entities and then work around the built in behaviour that the relationship brings.

In the RESTful API – the client decides what to do with the resources and the relationships between them, with completely ignoring them as a viable options. The web

[2] In order to further save bandwidth, it is possible for the API to generate a stripped down version of the resource when the client dispatches a specific accepted content type, i.e. 'x-json-next_action-min'

client in the prototype uses all of the entities in the system, but it is possible to build a client with a completely arbitrary set of these entities without modifying the API at all.

Decreased data transfer

Because of the reasons specified in section 2.3 the data usage generated by the client application is far less than the comparable traditional approaches. Minimizing data usage is crucial in scenarios where transfer speed becomes a bottleneck and the cost of data is prohibitively high.

Because of the way the application is structured, it needs constant communication with the API. Every second spent on the request-response mode can translate into a second of lost responsiveness of the application, which leads to decreased user experience. These repercussions can be mitigated by using appropriate techniques (like asynchronous requests in HTML/JavaScript) but these require additional attention and resources.

Also, with the still high prices of mobile data transfer mobile phone users are starting to monitor the data usage of their applications and deleting those that generate the most costs. This once marginal phenomenon is spreading its influence due to the rising consciousness of users and new built in system tools, like the ones introduced in Android 4. Viewed in that light, a mobile phone application developer can have substantial financial gains by minimizing the data transfer the application generates.

Transparency and interoperability – databases, languages, platforms

Because of the language independent data notation (JSON), the client using the API has only two technical limitations: it must be able to make HTTP requests and parse the standard HTTP response and the JSON body of it. Because these standards have almost total coverage over different programming languages, a client can be built for virtually any known platform.

The other side of this point is that the general concepts outlined in this thesis regarding the API can be implemented on any platform, thus making it a very versatile and robust solution.

Modular – easily extensible

Due to the loose coupling of the particular parts of the system, elements can be added into the API without breaking backwards compatibility on the side of the client.

If we would decide, that the prototype needs to be extended with an entity Person, that can be attached to a Next Action to signify that the action has been delegated to a particular person, let us examine the needed steps. First, the API would have to be

augmented with the necessary model and controller for the new item. Then, the Next Action model would have to be modified to accept a relation with Person. This would amount to a new attribute in the model: 'person_id', that would be null by default. At this point, the new version of the API is ready and can be exported to production.

The web client, oblivious to the existence of the new entity would not be ready for its management, bot nor would it become unusable. This is important because many models of client distribution have considerable delays between user updates. If the change would not be backwards compatible and break the client, the API would have to provide a versioning strategy, which would need additional resources to do. It must be noted that the model presented here does not completely protect against breaking backwards compatibility, but it does limit the occurrence of such and hence lowers the costs associated with them.

6.2.2 Disadvantages

Building client-side frameworks

Let us consider what happens, when the client receives the FEAR. Depending on the content of said response and ability of the client, the range of tasks that must be done by the client will vary up to the range described in section 3.2.3. All such supported tasks must be implemented in the client, which needs additional effort.

Depending on the complexity of the API and the resultant amount of entity data transported by FEAR, this can lead to overhead of code just to parse and process the FEAR.

The prototype described in this thesis is very far from reaching the complexity limits imposed by FEAR, and hence does not provide us with the necessary data for accurately describing the consequences of reaching such limits.

Slow connection issues

Because of the need for constant communication between the API and client, slow and unstable connection are the most serious problems that arise from the fear based solution.

Furthermore – this technical difficulty cannot be solved without undermining all the fundamental benefits that come from the approach described, as any attempt to remedy the situation will invariably lead to adding some functionality to the client and hence increasing the cost of it's development. Some of these are described in section 6.4.2. The positive aspect is that solutions therein described can be applied on a as-needed basis. That is: having a central API and making 3 clients (one web client, one desktop client,

one mobile client) there is probably only need to apply these optimization on the mobile client, because the other two settings are not designed to work without internet access).

Connection issues can lead to the following detrimental user experience elements:

- Low application responsiveness – because the application needs to connect to the API at any action that the user requests, it will the unusable at that time. The client developer should take note of this and design interfaces accordingly (for example sending requests in the background).
- Data loss – if the client sends the given information and the API does not respond due to a temporary loss of connection, the client should save the request to a later time – in another case, the data submitted will be lost leading to user frustration and lack of trust.
- Decreased client location coverage – The client will not be able to function in places bereft of data coverage. This can be an important factor. In an example for the usage of a hypothetical mobile client for the prototype API, we could imagine someone wanting to work with the application while travelling on the underground railway. Because there is no mobile coverage there, the user will not be able to do that.

6.3 Potential applications

This section explores the repercussions of this thesis by defining a set of characteristics that facilitate a proper implementation of the fear based solution. These characteristics are grouped into those relating to the application model (the data, relationships, use cases) and the technical details (i.e. resources, platforms).

Repetitive development

The very philosophy of FEAR presupposes that application development is not a one-time unique event. It it were, then all of its superlatives could be achieved only with using a RESTful API. Worse still - there would be additional effort required to implements some of the FEAR elements that might not be used on the particular application being developed, even in the "standard" set of response directives. Consequently, there are two ways in which the fear response can save on development time: repeated application development and continuous upgrades.

The first of these applies in a situation in which the developer builds more than one application in a language that was used in an earlier project. The time saved comes from

the fact, that if the developer followed good coding style, the part of the application related to parsing FEAR and reacting to the more standard events can be reused. In a hypothetical scenario of building another prototype application for this thesis, the following files from the code appended to this thesis could be reused:

- `/taskage/app/assets/javascript/fear.js.coffee` – could be used in most web clients
- `/ataskage/src/model/Fear.java` – can be used in Android applications.

This would give the developer some standard functionality, like handling validation, only for the cost of embedding the objects into the requirements of the given projects. This of course assumes that the given project would adhere to other conceptual and technical compatibility requirements from this chapter.

The other option for the optimal usage of the fear pattern has already been voiced in this thesis. It is the assumption that the application as a whole evolves with time. In this case, the approach can become especially valuable for building mobile and desktop clients, because of the inherent uncertainty of the update procedures of these clients. If appropriate care is taken, the application can be ready for a certain set of changes in functionality that will arise of the side of the API. This becomes more important when considering the fast face of development of today's businesses (and the popularity of the Minimal Viable Product approach) and also considering that the basic mechanisms of fear can allow for a certain level of communication with the user (messages, validation failure due to outdated software etc.).

Stateless communication

Because of protocol limitations, the application should mostly rely on pull based communication. REST is a stateless architecture style and was not designed to handle direct, sustained communication between the two tiers.

This means that a large portion of applications relying on small but frequent data transfer that has no discernible primary direction (i.e. when the server should initiate the communication). Such applications include multiplayer games, chats, collaborative editing tools and any application that has to synchronize the states between several clients in a manner as rapid as possible.

However, due to the proposed WebSockets standard [39], if such components are only a small part of the application as a whole, nothing stands in the way of combining the two technologies to use the strong suited of both. There is even the possibility of sending a fear body over WebSockets, but that would entail problems ot its own.

6.4 Further development possibilities

6.4.1 Clients for other platforms

A natural consequence of this thesis would be to develop more clients targeted at both mobile and desktop devices. The implementation of said applications could give rise to problems and solutions not encountered in the clients implemented as part of the prototype and hence increase the understanding in which the FEAR method is a viable design pattern for applications.

6.4.2 Overcoming the disadvantages

Offline work and impaired connectivity

There are several techniques that can be applied to the client application that increases the user experience in cases of inferior API connectivity.

Augmenting client-side logic is the most obvious one. This leads to hard-coding the logic that the application should receive though a FEAR. It is the simplest but also most costly due to neutralising the benefits of such a technique. This is advised only when the client in question is limited scope and needs to be able to cope with limited connectivity.

Creating a request queue is a technique that involves dispatching requests into as queue and then to the API. When there is no connection with the API, the requests are not sent – they wait for the connection to be re-initialized. In the meantime – the client runs is "offline mode" with the user being able to interact with a subset of the functionality (or even all of it). Of course, the changes that are usually handled by the fear are not processed and hence the client can have a visible state that is not up-to-date with the one that it should have.

The request queue technique can also lead to data loss when working on the same data from different clients. Example – the user changes an action name on a mobile client, which has lost connection. The request is saved in the queue and the user forgets about it. The user then changes the action name again in a web client. Sometimes after this, the mobile client regains connectivity and sends the older requests. If care is not taken by the API to take the request time into account, the user will end up with "old", which will surely be disturbing. This is represented in the figure 6.1.

Also the queue technique is not appropriate for all applications. In the client has a lot complex, intertwined elements, it will become very confusing for the user if he will be able to interact with the application but not get desired results (inconsistent data). In these cases it might be better to just notify the user that connectivity has been lost and

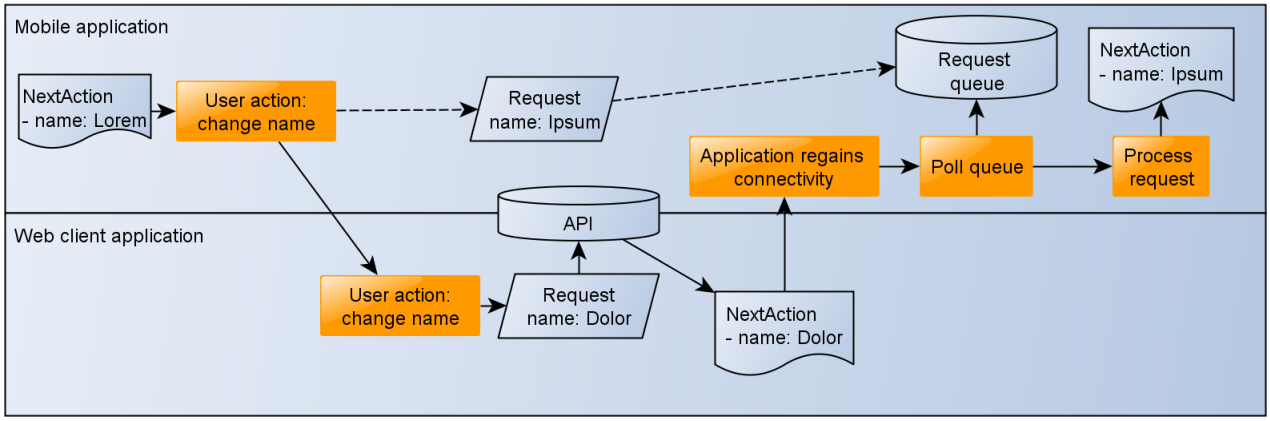

Figure 6.1: Possible data corruption in data synchronization.

the client is not usable.

Synchronization – client side state management

It is possible, it not expected, that the data on the API side can be updated by multiple clients at a time. In the case of the prototype application such a case might be rare, because the system is a *personal* task management solution. In this system, the only way to do that is to use the web client and mobile application at the same time, which is an artificial situation due to the fact that they fulfil the same function and the user should have no need for that sort of behaviour.

But if we imagine a shared environment, in which several users can share tasks between each other, the scenario becomes obvious. In this case, we would need to develop a system that informs the client of the changes made to the underlying data. The traditional way is just to set up the client to poll the API with a request in set time intervals. This makes certain that the latest data is requested regardless of user interaction. Another way would be to "piggy-back" the change information with the usual user requests, when the client sends any request to the API, it automatically attaches said information as part of the response. Because the responses are standardized in the FEAR pattern, adding information would not break any response parsing logic on the client-side.

Please note that the described changes would require a minuscule[3] amount of changes on the client side with the first approach, and absolutely no changes with the second. This is because the *change* directive group, when implemented, should handle all the possible changes in data.

For the above mechanisms to work, there needs to be a way for the API to determine what particular client is issuing a request, when did it last issue a request and what

[3]Can be done in a few lines of JavaScript in the case of the prototype web client.

datasets have changed since then. This can be achieved in multiple ways. In the case of the prototype a possible approach would be:

- **in the API** - make sure that all crud operation on the database regarding the traceable entities are saved in a transaction log
- **in each client** - generate a unique token during start-up, save that token as the client session id.
- For each request from the client –
  - **poll the transaction log** for elements that have a creation date that is larger that the last request time for a given client, then transform these events into proper FEAR directives
  - **save** the time of request along with the client id,
  - **output a FEAR**

The little amount of effort needed to implement this shows the advantages of the FEAR approach.

Bibliography

[1] A. Strak, "Retinal display technology for future mobile applications,"

[2] W. Lin, A. Dezieck, and M. Aufrecht, "Visions of bionic lenses: Foresight for the future,"

[3] D. Zhang and B. Adipat, "Challenges, methodologies and issues in the usability testing of mobile applications," *International Journal of Human-Computer Interaction*, vol. 3, no. 18, pp. 293–308, 2005.

[4] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," *Information Theory, IEEE Transactions on*, vol. 49, no. 9, pp. 2213–2218, 2003.

[5] A. Charland and B. Lerous, "Mobile application development: Web vs. native.," *Communications of the ACM*, vol. 54, no. 5, pp. 49 – 53, 2011.

[6] Wikipedia, "Mobile devices." [http://en.wikipedia.org/wiki/Mobile_devices].

[7] O. Buyukkokten, O. Kaljuvee, H. Garcia-Molina, A. Paepcke, and T. Winograd, *Efficient Web browsing on handheld devices using page and form summarization.* ACM Transactions on Information Systems, 2002.

[8] M. Miłosz, *Informatyka Gospodarcza: 8. Systemy mobilne*, vol. 4, p. 211. C.H. Beck, 2010.

[9] J.-S. Lee, S.-H. Yoo, and S.-J. Kwak, "Consumers' preferences for the attributes of post-PC: results of a contingent ranking study.," *Applied Economics*, vol. 38, no. 19, pp. 2327 – 2334, 2006.

[10] R. Goldsborough, "Mobile devices on rise, but death of PC is greatly exaggerated.," *Community College Week*, vol. 23, no. 13, p. 17, 2011.

[11] R. Goldsborough, "Is the PC really dying?.," *Tech Directions*, vol. 70, no. 8, p. 14, 2011.

[12] "Cross-platform." [http://en.wikipedia.org/wiki/Cross-platform].

[13] "Smartphone vs. feature phone 'tipping point' coming." [http://www.itpro.co.uk/634816/smartphone-vs-feature-phone-tipping-point-coming].

[14] Statcounter.com, "Statcounter.com." [http://gl.statcounter.com].

[15] R. Padley, "HTML5 - bridging the mobile platform gap: mobile technologies in scholarly communication.," *Serials*, vol. 24, pp. 32 – 39, 2011.

[16] S. J. Vaughan-Nichols, "Will HTML5 restandardize the web?.," *Computer*, vol. 43, no. 4, pp. 13 – 15, 2010.

[17] B. Korkmaz, R. Lee, and I. Park, "How new internet standards will finally deliver a mobile revolution.," *McKinsey Quarterly*, no. 3, pp. 47 – 53, 2011.

[18] M. Obcena, "Rich cross-platform desktop applications using open-source titanium.," *Linux Journal*, no. 185, pp. 54 – 59, 2009.

[19] "Adobe air – features." [http://www.adobe.com/products/air/features.html].

[20] "Flash to focus on pc browsing and mobile apps." [http://blogs.adobe.com/conversations/2011/11/flash-focus.html].

[21] M. Creeger, "ACM CTO roundtable on mobile devices in the enterprise.," in *Communications of the ACM*, vol. 54, pp. 45 – 53, 2011.

[22] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*.
PhD thesis, University of California, Irvine.

[23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616: Hypertext transfer protocol–HTTP/1.1, june 1999," *Status: Standards Track*, 1999.

[24] "Representational state transfer." [http://en.wikipedia.org/wiki/Representational_state_transfer].

[25] R. T. Fielding, "REST APIs must be hypertext-driven." [http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven].

[26] R. T. Fielding, "RFC for REST?." [http://tech.groups.yahoo.com/group/rest-discuss/message/6735].

[27] J. Udell, "Two approaches to services.," *InfoWorld*, vol. 27, no. 11, p. 35, 2005.

[28] "Json." [http://www.json.org].

[29] "Software development methodology." [http://en.wikipedia.org/wiki/Software_development_metho

[30] "Toward agile: An integrated analysis of quantitative and qualitative field data on software development agility.," *MIS Quarterly*, vol. 34, no. 1, pp. 87 – 114, 2010.

[31] D. Mishra and A. Mishra, "Complex software project development: agile methods adoption.," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 8, pp. 549 – 564, 2011.

[32] M. Fowler and J. Highsmith, "The agile manifesto," *Software Development*, vol. 9, no. 8, pp. 28–35, 2001.

[33] U. A. K. Asif Irshad Khan, Rizwan Jameel Qurashi, "A comprehensive study of commonly practiced heavy and light weight software methodologies," *IJCSI International Journal of Computer Science Issues*, vol. 8, no. 4, pp. 441–550, 2011.

[34] "Coffeescript." [http://coffeescript.org/].

[35] G. Myers, C. Sandler, and T. Badgett, *The art of software testing.* Wiley, 2011.

[36] J. Link and P. Fröhlich, *Unit testing in Java: how tests drive the code.* Morgan Kaufmann Pub, 2003.

[37] M. Levesque, "A metamodel of unit testing for object-oriented programming languages," *Arxiv preprint arXiv:0912.3583*, 2009.

[38] K. Makice, *Twitter API: Up and running.* O'Reilly Media, 2009.

[39] I. Fette, A. Melnikov, Google Inc., and Isode Ltd., "Request for comments: 6455," *Status: Standards Track*, 2011.

List of Figures

List of Tables