Matias Oksa

WEB API DEVELOPMENT AND INTEGRATION FOR MICROSERVICE FUNCTIONALITY IN WEB APPLICATIONS

UNIVERSITY OF JYVÄSKYLÄ
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS
2016

ABSTRACT

Oksa, Matias

Web API development and integration for microservice functionality in web applications

Jyväskylä: University of Jyväskylä, 2016, 77 p.

Information Systems, Master's Thesis

Supervisor(s): Semenov, Alexander

This paper presents a model for web application programming interface (API) that uses the microservice architecture to enable and support end-user feature development. The study follows the design-science paradigm of information systems research. Participatory design stands as one ISR-based theory and definitions of user types are derived from that. Web application definitions and web application development methods are also explored to some extent. The study will also present an implementation of the presented web API design that is then analyzed and reviewed in comparison with other similar API design patterns. Implementation consists of application extensions, web API and microservice based feature extensions, written in Javascript using Node.js runtime environment and the host application uses ASP.NET MVC. The implementation reveals benefits of microservice architecture regarding scalability, extensibility and utilization of the user-base in application feature development. Extending an existing application also emphasizes the importance of communication protocol specifications and related knowledge.

Keywords: Web API development, Participative design, Design science, web applications, microservice architecture

Index of Tables

Table 1: Design-Science Research Guidelines (von Alan, March, Park, Ram, 2004)....12
Table 2: Actor types in the context of this thesis....16
Table 3: Construct implementation requirements....32

Illustration Index

Illustration 1: Extended monolithic layering....26
Illustration 2: User interface layer microservices....27
Illustration 3: Pure microservices layering....28
Illustration 4: Creato.red first level deck view....34
Illustration 5: Question deck view....35
Illustration 6: Question element contents....36

TABLE OF CONTENTS

ABSTRACT 2

TABLE OF CONTENTS 4

1 INTRODUCTION 6
- 1.1 Motivation 8
- 1.2 Research question and focus 9
- 1.3 Expected results 9

2 RESEARCH METHODS 11
- 2.1 Design science research 11

3 BACKGROUND THEORY 14
- 3.1 Participatory design 14
- 3.2 Users and actors 16
- 3.3 Microservice architecture 17

4 TECHNICAL ASPECTS 20
- 4.1 Web applications 20
- 4.2 Web API 21
- 4.3 Web application development 23

5 MODEL AND IMPLEMENTATION 24
- 5.1 Planning 24
  - 5.1.1 Functional requirements of the host application 24
  - 5.1.2 Microservice architecture layering options 25
  - 5.1.3 Microservice granularity 29
  - 5.1.4 Internal communication definitions 29
  - 5.1.5 External communication definitions 30
  - 5.1.6 End-user development support 30
  - 5.1.7 Microservice extension support 31
  - 5.1.8 Lasting compatibility 32
  - 5.1.9 Construct implementation model 32
- 5.2 Development 34
  - 5.2.1 Host application: creato.red 34
  - 5.2.2 Host application technologies 37
  - 5.2.3 Implementation technology decisions 37
  - 5.2.4 Construct model-based implementation exploration 38
  - 5.2.5 Implementation-phase decisions 39

6 RESULTS......41
6.1 Implementation construct review......41
6.1.1 Implementation construct review on the planning model......41
6.1.2 Planning model construct comparison......41
6.1.3 Django......42
6.1.4 WordPress......42
6.2 Construct analysis......43
6.2.1 Implementation issue analysis......43
6.2.2 Implementation success analysis......44

7 DISCUSSION......45
7.1 Discussion of findings......45
7.2 Implications and limitations of the study......46
7.3 Future research options......46
7.4 Conclusion......47

LIST OF REFERENCES......48

APPENDIX......50

1 INTRODUCTION

Outsourcing the development of a in-organization software is complicated more often than not, as the development effort requires degrees of extensive knowledge of the business field. Initializing such projects includes many potential causes of issues along the development timeline and, not excluding project agreement related issues, also early artifacts such as software feature specifications tend to end up being very intricate. While the specification process is alleviated by methodologies such as Requirements Engineering, even the need for a methodology that defines the process of generating the specifications for development speaks volumes of the complexity of the issue. This elaborate specification is necessary in order to level the vision for the software results for both the developer and the customer organization. Despite the quality effort put in the specifications and reiteration along the development timeline, customer dissatisfaction may still arise from the implementations. Regarding in-organization software development, an Internet connection is also ubiquitous in today's work environment and hardly any work is without some level of networking, as interaction with external data is most of the time, if not always, necessary.

From this point of view, web applications offer tantalizing characteristics relating to tool development in business circles. These characteristics include wide array of devices suitable for the use of the web applications, including the growing amount of mobile devices that enable the use of this type of work tools outside of the physical premises of the workplace. Typical development model of these in-organization tool applications usually follows a monolithic software architecture, where the application features are contained within a single application. Microservices architecture is another software architecture that is arising from the monolithic application development practice. The microservice architecture defines the application structure as multiple self-contained single-task applications that communicate between each other in a service-like manner in order to offer the intended application features. This intercommunication opens possibilities for extending the main application with features that may be

developed by external developers such as the customer organization as well as the end-users of the application. This also leads to the research question of this study and if the microservices development is ”anyone's game”, we're here to explore the ”plays”, so to allegorize.

Research question of the study is as follows: **How to enable further microservices based web application development with Web API design and implementation?** Web Application Program Interface (API) is a central concept within the microservices architecture, as the microservices communicate using the communication methods defined by the API. Several aspects of the human side of the issue are also explored to improve the holistic view on the subject, such as ”why would end-user development support work”, ”what are the assumed benefits of the support” and ”what conditions does the use of microservices architecture require”. To answer these questions, a microservice-centric Web API model is designed and implemented, in order to further evaluate the architecture's strengths and weaknesses in end-user development context. The implementation of the modeled Web API is evaluated based on the expected improvements as defined in the design model as well as similar, existing implementations of the API architecture. The base web application is used as comparison target in the explored case. Viability and overall usefulness is examined in comparison with the similar implementations. The design is done with end-user development in mind and the implementation should reveal actual as well as unforeseen differences between the implementation and the design and also between the implementation and similar architectural decisions. The differences occur as decisions are made in the implementation step to comply with technical requirements of the application. This is especially prevalent in the implementation case of the thesis, where an existing web application serves as a base for the improvements. Technologies used in the development have requirements of their own and while the design isn't changed, certain features are implemented in specific ways dictated by the technologies. In this thesis, the implementation is developed using Node.js Javascript runtime environment and the base application for the new extensions uses ASP.NET MVC for the server-side application, MongoDB as database application and React for the client-side application. The technologies are further presented in the chapters 5.2.2 and 5.2.3.

The rest of the thesis is organized as follows: Chapter two contains a description for the research method used in the study, namely design science research. Chapter three contains background theories relating to the research field, participatory design, actor model, microservice architecture. Chapter four presents technological details relating to the study, namely web applications, web API and web application development. Chapter five contains design science related presentation of problem relevance and two constructs or "artifacts" of the study, the planning construct and the implementation construct. Success of the implementation is reviewed and analyzed in chapter

six, "Results". Findings of the study are discussed in chapter seven, as well as limitations and implications of the study and future research options. Thesis is concluded also in chapter seven. Appendix contains most relevant excerpts of the implementation construct in a source code format.

1.1 Motivation

Business-oriented tool development present following generalizable issues:

- The end product doesn't match the use cases or needs of the actual users of the software
- Beneficial innovations regarding functional requirements may arise during the use of the application
- Stability of (web) applications in continuous development can be an issue especially in applications with monolithic architecture

These issues may relate to difference of views towards the software application between the extended development team including the requirements engineering team as compared to the views of the content consumer actors regarding the software application. The issue of unmatched use cases and needs is more prominent in cases where ready-made packaged software solutions are used for specific tasks beyond their initial task orientation, especially if the software is repurposed to the other task by the customer organization. This issue may also arise in cases where there are multiple detached groups connected to a software project, hindering the communication of the project goals between the developers and the end-user group. This study, as explained in the next chapter, aims to utilize microservice architecture and web API design to offer solutions to the aforementioned issues, arguing that the end-user development possibilities offered by microservices architecture can be a viable option to steer the application development towards a more efficient use of a software application.

The last issue regarding stability arises from the monolithic web applications, especially in continuous development, as faulty implementations, or oversights in testing may result in errors that may cause the whole application to collapse. Instability in one feature affects the stability of others, as the features are interconnected by design. Simply put, if one part of the application doesn't work, the whole application doesn't work. The microservice architecture may offer a solution to this issue, as the faulty implementations affect only a part of the application and a properly working fallback system may retain some overall stability within the software application. While these measures can also diminish the need for proper testing suites and test driven development, simultaneously decreasing the need for this type of lateral

development, testing suites are still recommended to automate testing for central actions within the application and this testing documentation can be also used to generate the API help documentation to an extent. These generalizable issues are also recognisable in the web application case and related development presented in greater detail in chapter 5.2.

1.2 Research question and focus

Research question for the study is **How to enable further microservices based web application development with Web API design and implementation?** Subquestions include ”under which conditions web API can be used as a platform for microservices” and ”what is the architecture for the API that would enable efficient microservices development”. Focus of the study is placed mainly on the development of the platform as well as the technology decision and characteristics. Lesser focus is also placed on planning and specification evolution steps and review and analysis for completeness sake from the point of view of general project examination.

1.3 Expected results

Expected results of the study include the constructs, developed microservice architecture, platform specifications, planning methodology as well as results and methods used to review and analyze the platform construct. Business-oriented results includes documentation of unexpected differences between planning stage and finished feature, as decisions are to be made in development step which might arise from infeasible or further developed specifications.

Implementation of microservice-based web API may be used to further determine the necessary changes to the characteristics and requirements of the initial interface model, as well as the improvements that may be applied to the implementation in order to be more successful. From the problem-solving approach of the design-science research paradigm, examination of the construct may give answers to questions besides the main research question, such as ”how to leverage knowledge of the user base in feature development” as well as ”how to improve the feedback-design-implementation cycle of feature requests to be more responsive to user needs” and on the other hand, general questions reversed from the construct, such as ”what kind of problems does this implementation/construct solve”. These questions or problems, as designed by the paradigm, arise from personal experience as a web application user and developer. Concrete example relating to the case web application was that the users found the action of creating multiple documents one after another

tedious. The solution in that case was to implement a keyboard shortcut that allowed the user to create documents without clicking graphical user interface elements, latter being the part of the action that was quickly dismissed as the tiresome aspect. Here, the lowest common denominator question found in the aforementioned questions is that of "how can I perform this action better". The same question from the point of view of the developer is likewise "how could the user perform this action better". The aforementioned questions are also applicable to real-world business problems, where the answers are transformable into business-oriented results that may change or improve the view on user base as a resource.

One of the distinctive limitations of the study is that the software product is successful in gaining an user-base and enough users interested in end-user development in the first place. This can be understood so that the web API implementation benefits are only realizable with larger user-bases and not necessarily beneficial otherwise. This limitation, while worth some consideration, may be averted by the initial user-agnostic nature of the API. Even with modest size of user-base and lesser chance of participative design interaction, the web API offers greater interoperability mechanisms for the software product development. While the interoperation is to be defined in the web API architecture in similar manner as before, the definitions also offer structure for new feature implementations. User-base generation and success in the market are acknowledged to be greater pressing questions regarding software product success and the business-oriented results, but as such they are dismissed as ”besides the point” regarding this study.

Results of the thesis include a model for the API and the microservices and an implementation of the API and a microservice. The model and implementation development are explored in chapter 5. The defined characteristics or requirements of the model are presented in chapter 5.1.9. The implementation, which is related to a host application presented in chapter 5.2., can be described as application extension, where the implementation offers the model-based API functionality and microservice-support for the host application. Chapter 5.2.5 presents some exemplary differences between the model and the implementation, as case-specific requirements are applied to the implementation. The implementation related results are further examined in chapter 6 with comparison between the implementation, the model and similar, popular application frameworks. The findings and implications are examined in chapter 7, business-implications including benefits of the user-base as a development resource and microservice-architecture use in web application development.

2 RESEARCH METHODS

Research method of the study is constructive research where the main construct is the developed microservices platform. Supporting constructs include results and the method of planning used for the platform, as well as a construct for review and analysis for the platform. Design science paradigm, such as the one explored by von Alan, March, Park and Ram (2004) is also utilized in the research method.

2.1 Design science research

Design science paradigm used in this study is one of the two main paradigms of information systems (IS) research (von Alan, March, Park, Ram, 2004). The paradigm is fundamentally a problem-solving paradigm, that seeks to innovate and create artifacts that benefit the information systems use and design. As such, the paradigm clearly fits the targets of this study, one of which is applicable and actionable artifacts regarding the design of web APIs and their connectivity to web applications. Design science aims to create and evaluate IT artifacts intended to solve identified organizational problems (von Alan, March, Park, Ram, 2004).

Identified organizational problems related to the API development and the business point-of-view may include questions such as ”how to achieve faster feature delivery to requesting customers” as well as ”how to enable the user base as a source of feature design and development”. While these are not applied to this thesis as research questions, these merely illustrate examples of business-beneficial targets and effects of the web API framework. As web applications based on Web APIs tend to be somewhat open for creative thinking from the point of view of feature development, business perspective works as one of the main drivers of the feature design process. Depending on the importance of a web application in the workflow and processes of a company, the qualitative and central characteristics of the web application may

have indirect connections to a wide array of organizational elements. Affected elements on the service provider side may focus on business strategy and organizational structure, whereas the client organization effects are mostly on information technology strategy and information systems infrastructure.

Table 1: Design-Science Research Guidelines (von Alan, March, Park, Ram, 2004)

Guideline	**Description**
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations and/or design methodologies
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences

Von Alan et al (2004) present guidelines for design-science research as seen in table 1. These guidelines are applicable as steps to achieve well structured design-science research. Examining this study through the guidelines: The Web API design initially fits the model and method types of artifact requirement, and the exemplary implementation those of a construct and an instantiation.

Problems relating to the study targets are mostly relevant for organizations that develop or use web applications in their workflows. The Web API model and the implementation of this study are evaluated by comparison to the base application in the implementation case of the thesis. The implementation and the related design model are also examined in comparison

to similar existing constructs to evaluate the viability and general usefulness factors of the implementation. Research contributions are reviewed based on the construct. Research rigor expects comparison between the implementation and the design model as well as comparison between the implementation and the existing models, although novelty is as well one of the optimal goals of the construct. Design as a Search Process guideline is accomplished with the implementation development, as it follows the model to "satisfy laws in the problem environment" and "utilizing available means" to comply with the case goals. Communication of Research is achieved with publications of the study as virtual documents and physical copies following the university thesis publication process.

Guidelines applied in chapters:

- Introduction: Problem Relevance
- Background theory, Techical aspects: closest to Design as a Search Process
- Model and implementation: Design as an Artefact, Design Evaluation, Research Rigor
- Results, Discussion: Design Evaluation, Research Contributions

3 BACKGROUND THEORY

Development of a proof-of-concept web application in a Jyväskylä University research project has a Web Application Program Interface (API) as a planned feature. The proof-of-concept web application is a question-based note taking tool for companies to use in generating meeting documentation as an example. In this chapter, related theoretical bases and concepts are examined in a bottom-up manner, starting with more baseline concepts and continuing further from there.

3.1 Participatory design

Participatory design represents an approach ”towards computer systems design in which the people destined to use the system play a critical role in designing it” (Schuler, Namioka, 1993). More characteristics of participatory design are relayed from prior research. Schuler and Namioka (1993) align traditional design with the aims to automate the skills of human workers with computerization, whereas the participatory design aims to improve the tools needed by the human workers. Participatory design assumes that the workers or users know best what is needed to improve their work performance, assigning the role of expert to the user and software designers are seen as technical consultants. Participatory design promotes users' perceptions and feelings of the technology to the level of importance with the users' ability to achieve their objectives using the technology (Schuler, Namioka, 1993). Computer-based applications are seen in the context of a workplace as processes rather than as products within the participatory design point of view.

While participatory design does not necessitate the use of experts in the design and development process, they are certainly useful. However, participatory design according to Schuler and Namioka (1993) promotes active participation in the search for the optimal or better designs and as such, the

users are in a critical role for the improvement of the designs and resulting software. Assumptions regarding participatory design and this study are if the users know best how to improve their work performance, advanced users may be able to improve their work performance relating to the use of the application given enough support. While the basic operation of the application should not require excessive knowledge of the inner workings of the application, the option to change and improve the application may prove to be greatly beneficial. This has ties to behavioral science research in that question that is posed as to ”why would people do development for free” is essentially behavior research. Hars and Ou (2001) explore the motivations of people developing open source software and these motivations include internal factors, such as intrinsic motivation and altruism, as well as external factors, such as future rewards and personal needs.

Intrinsic motivation consists of motivation by the feeling of competence, satisfaction and fulfillment that arises from writing programs. This is closely related to personal needs as a motivation factor, and as such may lead the development towards certain goals aside from ”basic user needs” as noted by the researchers. This intrinsic motivation can be assumably reinforced by support of end-user development, so that aspiring feature developers might spend less time examining and studying the development environment, i.e. the development environment ”just works”. Altruism in this context is the opposite motivation factor, in which the programmer might see usefulness for others in a certain functionality and decides to implement it fully or to an extent. This type of motivation factor may be reinforced by open issue tracking or feature request board, which might have the additional benefit of redirecting original developer attention towards user needs as well, besides inspiring end-user developers.

External motivational factors include revenue from related products and services, human capital, self-marketing and peer recognition within the future rewards category, as well as personal needs as a category. Revenue from related products and services is related to crowdsourcing and bounty hunting within the software development context. Human capital consists of programmers intent to expand their skill base, as freedom to select tasks leads to freedom of selecting enticing learning experiences. Self-marketing is a motivational factor where the motivation arises from the competence and skill demonstration that is possible in open source development. Peer recognition is tied to the latter, where the motivating factor is desire for fame and esteem.

While not diving in depth into the behavioral side of development, this stands to back the assumption that given enough support, participatory design has means to succeed in its approach to the software development. The support however is necessary for this type of interaction to emerge, where the most simplistic definition of benefit or value is faster feature implementation.

Regarding web application development, this may also redefine the role of the original development team, as the focus is shifted from feature development more to the support tasks, so that the feature developers on the end-user side may achieve their goals better. Beneficial side-effects of the participative design interaction may also include improved work performance on the end-user side and improved relationships between the developer company and the customer companies and between the developers and the end-users of the application, as the users are included in the work tool development.

3.2 Users and actors

This chapter examines the concept of users within the context of the project. The definition of user is used synonymously with end user, that is, going by the full definition by Merriam-Webster (2016), ”the ultimate consumer of a finished product”. Simply put, ”user” shall refer to the person actually using the product for its intended purpose. However, there are user-like entities that may provide worthwhile to note and these entities shall be referred to as ”actors”. Four distinctive actor types are clearly recognisable in the software project context and they are defined in Table 1.

Table 2: Actor types in the context of this thesis

Actor type	**Functionality source**	**Functionality consumer**	**Design source**
Developer	Yes	No	Yes
Content producer	Yes	Yes	Yes
Content consumer	No	Yes	No
Managerial actor	No	No	Yes

Actors within this context can be roughly grouped into four distinct categories, namely developers, content producers, content consumers and managerial actors. Actors with ”functionality source” characteristic develop functionalities in an application that may be then used by actors with ”functionality consumer” characteristic. ”Design source” characteristic relates to the need for particular functionality in an application that does not include the development of said functionality but may lead to a functionality request or commission.

Developer actors consist of the developers of the host application and they possess the largest technical knowledge and development effort to the application. Difference between content producer and consumer actors is related to the extent of participation towards an software application. Depending on the intensity of the relationship between the main developers

and the content consumers and content producers, which may vary from weak intensity (large developer company, multiple customer companies) to strong intensity (small developer company or important customer), the amount of visibility for the developers to the actual use targets of the software product is also affected. This may lead to differences in the vision for the software product and as such, may need adjustments from the relationship management to keep the software product development on the right track regarding to the customer needs.

Content producer actors are users that are actively participating in the development of the software. Titlestad, Staring and Braa (2009) explore closely related term ”boundary spanners”, that can be seen as functionality producer actor within this context, where the boundary spanners affect the boundaries of the software functionality by expanding them into specialized areas by applying area specific expertise to provide or improve functionality of the software for that particular area. In the traditional definition of software development, this type of actor is largely non-existent due to the proprietary or closed source format of the software often prohibiting further software end user development.

Content consumer actors make use of the content and functionality available to them in order to accomplish tasks unrelated to the software development. In a sense, they approach the use of an application from the point of view of ”what is it designed to do” instead of ”how it can be designed to perform better or improved”. The further development and improvement of the software is left to the actual developers with some extent of interaction between the end-user and developer groups via feedback and improvement requests.

Customer organisations existence in this definition also includes the notion of managerial-level actor that may commission development of functionality either within the organisations content producer actors or the developer actors, depending on the relationship intensity and availability of options regarding further software development. Lack of end-user development support within a software product leaves customer organisations to rely solely upon the development efforts of the host application developer organisation. In that situation, if it so happens that the relationship intensity between these two organisations is also weak, need for essential changes within an application may lead the customer organisation to seek software solutions elsewhere.

3.3 Microservice architecture

Microservices are applications that communicate with a host application (such as the web application here) in order to implement external features for specific

results using available data from the host application. Microservices are small, autonomous services that work together (Newman, 2015). Danado, Davies, Ricca and Fensel (2010) define microservices as small, sharply focused applications with their own graphical user interface. The researchers also introduce the notion of user as a feature or information provider as well as a consumer relating to the development and use of the microservices. The study explores few approaches to the microservice creation from user point of view as well.

Namiot and Sneps-Sneppe (2014d) define microservice architecture as ”an approach to developing an application as a set of indepedent services.” While the researchers argue that the term ”microservices architecture” is new, one origin concept for the architecture is also mentioned. Uckelmann, Harrison and Michahelles (2011) explore architecture requirements for future of Internet of Things and the proposed system is that then acknowledged as the microservices architecture by Namiot and Sneps-Sneppe. This study by Uckelmann et al. ties the microservices architecture origins into the designs relating to Internet of Things (IoT) and Machine-to-machine (M2M) topics. As these topics are also interests in the project motivating this research paper, the use of microservices architecture can in that sense be seen fitting as well. The IoT and M2M aspects of the project are considered somewhat essential in that they, when used creatively and properly, may simplify complex processess relating to data handling The proposed architecture as a platform base for future Internet of Things defines a multitude of related characteristics, such as ”Extended static data support”, ”integration of dynamic data”, ”integration of an actuator interface” and the like. Most of these characteristics arise from examination of one ”EPCglobal Network” that is introduced to be one of the larger industrial de-facto standard systems relating to the use of Internet of Things within the industry scope.

Monolithic architecture is another, somewhat opposing architecture relating to web service development. Web services built on monolithic architecture often deliver presentation, logic and data access layers as a package (Alpers, Becker, Oberweis, Schuster, 2015). These layers can and often are split into separate elements in microservices architecture. The microservices characteristics vary from API feature extensions to task-oriented applications and could be made available for the user-base to generate value for the user-base as a whole. Namiot and Sneps-Sneppes (2014c) explore concerns relating to microservices architecture, as the architecture adds another layer of complexity to the application for the host application developers as well as a testing hindrance. These examples are a few of the challenges that should be addressed in the planning phase of the development project and explored in the study as well.

Microservices are one of the emerging directions in web-applications that enable users as feature developers for the web applications. This enables the host application developers to integrate features, offer extended functionality and steer development of the host application. How the users can be guided to solve their own issues using the API by developing microservices to cater to their needs and how does the use of the API affect the development targets for the web application are two examples of the interesting questions related to the area. Initial designs for the microservice features include the development and uses of mobile phone applications in order to upload audio recordings from meetings as an example, as well as connecting the web application to other microservice platforms (such as ifttt.com for example) for various intricate tasks and purposes.

Beside microservices, central concepts of the study include the microservice platform model introduced in the study and examination of implementation, where these microservices can be developed to a certain extent by and delivered to user groups using the platform. The delivery and development aspects of the platform could introduce virtually faster feature delivery for customers, as these characteristics of the platform would involve the customers in the feature developement optimization and integration processes to variable extents. Customized features are one of the main driving forces of software development projects in business organizations.

Potvin, Nabaee, Labeau, Nguyen and Cheriet (2015) lightly touch on the subject of microservices in a completely different environment, namely using voice-over-IP technologies as a substitute for current telecommunications delivery. Sneps-Sneppe and Namiot (2014) also introduce a microservice based architecture for telecommunication applications. These studies may as well be reviewed for insight and versatility exploration of the microservice architecture. Another M2M related study by Sneps-Sneppe and Namiot (2012) offer further considerations in the M2M direction of the software project, as it combines ETSI API framework with M2M elements and explores M2M related characteristics that may be utilized in planning.

The implementation of the web API would enable the use of the web application as a microservices platform. The microservice layer would reside on top the API layer and the application should initially offer microservice repositories for host application developers, individual users and company users. Microservices are one option for web application development which may enable users as source of features for web applications.

4 TECHNICAL ASPECTS

This chapter consists of more technology oriented details of the study, exploration of the topics Web applications, Web API and web application development.

4.1 Web applications

Web applications can be distinguished from web sites for the advanced interaction options they offer. Web application enables the user to interact with and change the content, making the web site dynamic. This distinction is quite light, as the definition of a web service may change based on the view point or type of the user using the service. While a content consumer may see a Wikipedia page or a blog site as a web site following the previous definition, this view may change into web application for the content producer, as e.g. Wikipedia and blogging hosts may offer various web applications to produce and otherwise interact with the content. The term web application may be also used to refer to any type of application, in-browser or otherwise, that use the internet connection for communication with other applications, such as servers and the like. In this study and more generally however, the main definition of the term is that these applications work within a web browser.

In this sense, the web application definition also has ties to the Software as a Service (SaaS) business model, as with the SaaS applications are often delivered as in-browser applications to the customer base. The National Institute of Standards and Technology defines Software as a Service service model so, that the end user can access the application provider's applications through a thin client (web browser) or with another application. SaaS business model is also loosely coupled with subscription based revenue streams.

Common denominator for these in-browser web applications is that they use a database on the server side to store user data. Using this denominator it

can also be argued that dynamic web sites that use web page based viewports, such as e-commerce applications may also be viewed as web applications. Most notable difference between this type of simple web application and an advanced one, such as the in-browser text document editor by Google, is the use of asynchronous communication technologies. These technologies enable the web application to communicate with the server end asynchronously or ”behind the scenes”, as the use of the application is not confined to the action-reply chain of typical web sites. Asynchronous Javascript and XML or ”AJAX” is one of the more prominent implementations of the asynchronous communication technologies. This implementation uses browser-side Javascript functionality to communicate with the server application using HTTP methods and Javascript Object Notation (JSON) or Extensible Markup Language (XML) to pass the data around.

Graphical user interface may also differ from that of typical web sites and in most cases also is distinctively different, often imitating typical desktop applications in grander sense such as feel or user experience, but also in smaller sense such as graphical thematic design regarding button appearance, for example, as the web browser defines a default appearance for visual elements from which the application may differ intentionally to offer a consistent look-and-feel for the user interface of the application. This often has connections to the mental image research regarding application appearance, as user expectations largely dictate the graphical design decisions relating to the application development. Often the most direct approach is taken in that the application looks graphically and is recognisable to the task it is performing. Examples of these include Google Drive document editor and Spotify web player, for example.

4.2 Web API

Web API or web application programming interface is an entity that defines the server functionality that is open for various external applications to make use of. These open end points are often defined by Unique Resource Identifiers, that are used to identify requested resources and options for expected response data format among other things. Web applications communicate with these end points to offer various methods of interaction with the server resources or content. The most basic set of these endpoints follow the database action set pattern that mainly consists of database row insertion, search, deletion and update actions, as the server application relies heavily upon the existence of a database application. This functionality offers the most primitive tools for the web application or indirectly the user to interact with the database and the contents, namely creating database entries, reading them, updating them and

deleting them. This functionality can and often is extended to include parsed creation, where the user input is combined with a database document template to generate more complete database entries, reading entries in various formats such as JSON or XML based on the request, and recursive update or recursive deletion, where content depending on or linked to another database entry is also updated or deleted.

One of the hypotheses of this study is that this Web API entity can be developed in such a way that the development of applications can be more fluid and that the API is more open in its approach to web application development support. As the API stands as an abstraction layer between the user-side web applications and the server-side backend server software, especially development of the web applications benefit from well documented web API functions, often referred to as the API documentation or just API on websites that offer API functionality. The intended goal of API documentation is to enable or easen the off-sourcing of web applications, but more often than not, the documentation is either only partially complete or outdated. This may mostly be caused by lack of development guidelines or general disregard for documentation needs. However, the documentation generates value for the web application developers, as a well-documented API besides well structured API functionalities can support the web application development exceptionally well.

Maleshkova, Pedrinaci and Domingue (2010) explore Web API development and feature discovery using SWEET. The study also offers introductions to general technologies within Web API architecture design and as such may prove useful in further examination of the technological aspects in the introductions part of the study.

Design behind the API should be such that it allows for more flexible use of the API in Machine-to-machine (M2M) and Internet of Things (IOT) related development and use of the application, as well as offering intuitive microservices development environment. Microservices, IOT and M2M should be explored as intended targets for results of the use of the API. Multiple referred studies (Namiot, Sneps-Sneppe, 2014ab., Krylovskiy, Jahn, Patti, 2015.) explore the topics of IoT and M2M, that may offer insight into the grander scale of design in the development project within the research paper. API design related factors such as authentication, communication, API feature promotion and integration to and from the host application are to be explored within the study.

4.3 Web application development

One of the more prominent web application development patterns is model-view-controller, MVC. The original model-view-controller design pattern is from 1979 (Reenskaug, 2007). Since then, the design pattern has been adopted by web application development. This pattern consists of three entities that interact with each other on the server-side. Model in this pattern refers to the types of data handled by the application and the definitions for the contents of these data objects. View defines what content of the models is offered to the user and the web application on the users side to represent the data. Controller defines the controls that the user has regards to the data manipulation and within the web application design pattern, the Controller entity is directly related to the web API that the server application offers. Most beneficial features of this development pattern are clearly defined responsibilities and simplistic break-down of the server application components. Most programming languages offer at least some support for web server development, but web server and web application development is programming language-wise dominated by PHP and ASP.NET/C#, as determined by W3Techs (2016).

On the users' web application or client-side, relevant knowledge is that of the View models and their properties as well as the Controller or web API defined end points that are necessary to interact with the data. Model can be seen as an internal part of the server-side application, thus mostly invisible for the client application. In-browser applications are largely developed in Javascript due to the ubiquitous support of said scripting language in the variety of web browsers. As a sidenote, Javascript is an implementation of ECMAScript scripting language family, other siblings including JScript by Microsoft and ActionScript by Adobe Systems.

Something worth of note are also the issues regarding the prominence of the MVC development pattern. The recognition of the most used design patterns is difficult in this case, as most recognised development patterns are essentially MVC based, apparent hybrids combining unspecified development patterns with MVC or that the other patterns may be transformed fully or to a large extent to MVC based format. This is again mostly related to the simple presentation form of the MVC, as the pattern entities are recognisable with sufficient accuracy in most development patterns.

5 MODEL AND IMPLEMENTATION

5.1 Planning

This subchapter consists of planning process, planning results, application specifications and basis for construct review and analysis. The contents of the subchapter are on a more abstract level of the microservices applications to retain applicability in implementations beside the scenario implementation presented in the next chapter. The contents relate to the ”Design as an artefact” guideline of the design science paradigm as well as the ”Design evaluation” guideline, as the implementation is evaluated partly on the construct implementation model as presented in the chapter 5.1.9.

5.1.1 Functional requirements of the host application

Functional requirements in requirements engineering are definitions for what the software application should do or what must be implemented in order for the user to accomplish their tasks (Wiegers, Beatty, 2013). Functional requirements of the host application in the context of the case application are largely determined by initial functional requirements based on the purpose of the software application in question. The development status of the application affects the functional requirements analysis. New software application projects may define the initial requirement set for the functional requirements whereas applications that are extended to enable microservices and Web API define at least some of these functional requirements already.

Spiral development models, such as The Spiral Model (Boehm, 1988) and the various Agile-based software development models (incl. Cohen, Lindvall, Costa, 2003) feature a feedback loop where these functional requirements are adjusted to accommodate the shifting development targets and then implemented in another development iteration.

Functional requirements design based on the software project type can be categorized as follows:

- New software application projects
    - Define functional requirements based on results of the requirements engineering
- Expanding software application projects
    - Functional requirements are already known.

Both of these general categories of software projects targeting web application development using the model-view-controller design pattern should offer requirements for basic database operations (create, read, update, delete) for all ”models” or data entity types that are handled via the application and stored in a database. This database operations requirement layer acts as a proposal for the very lowest level of interaction for the API. Any higher level interaction can be categorized into data aggregation and group of higher level interaction options includes view-models or data elements visible to the users, as these view-models often intentionally reduce the amount of data visible to the user, thus aggregating the data entity instances. This transformation between models and view-models may prove useful for data security purposes. The host application developers can use this layer of abstraction to reduce the visibility of vulnerabilities or other possibly security compromising information that is delivered to the user.

Web applications, especially those in public internet context, should be developed with data security concerns in mind. User-visible data entities in any web applications include metadata or information about the data. Especially without the use of viewmodels this metadata can also be unnecessary overhead for regular use of the application. These data aggregation options may also require additional API layer elements or ”end-points”, as is the case with search feature as an example, where the search results view is effectively data aggregation. Development of the API should keep in mind the following central expected features: basic data handling actions and data aggregation support for user-visible data elements (view-models) and user interface features such as searching.

The functional requirements of the host application are explored in the following subchapters and presented in a table format in chapter 5.1.9.

5.1.2 Microservice architecture layering options

Within the context of basic features, feature microservices and feature extension microservices, the term ”feature” relates to the implementations of the functional requirements of the host application.

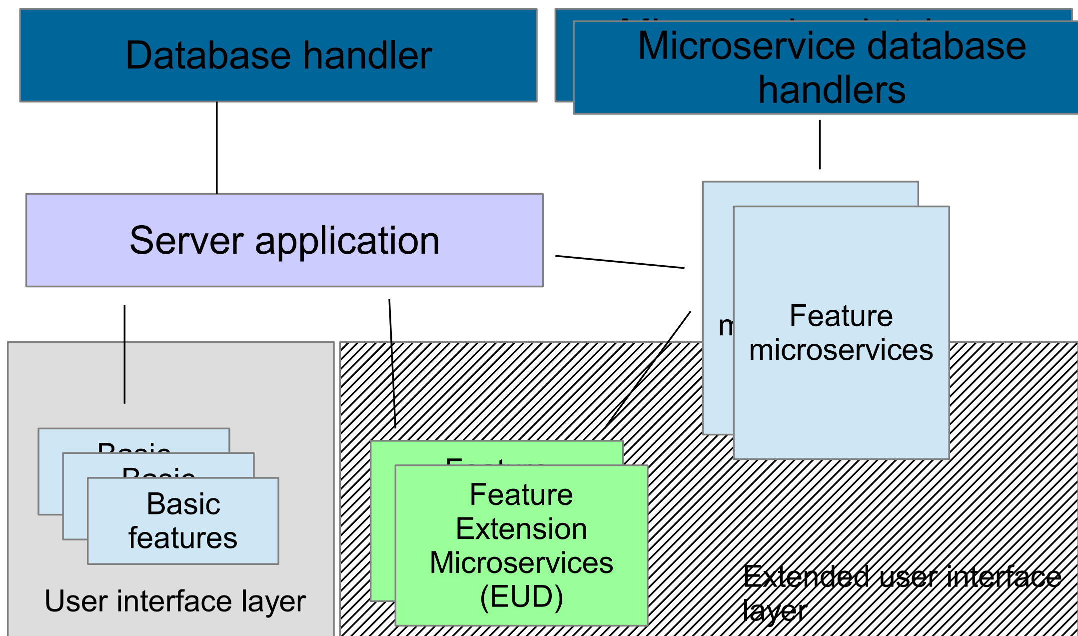

Illustration 1: Extended monolithic layering

Figure 1 depicts a layering option where the microservices are located on the user interface layer. The microservices, despite being otherwise complete implementations for functional requirements, are connected to single back-end application instance that handles all of the microservices needs for database interaction. The user interface in this layering option relies heavily on the stability of the single back-end application instance and the application is likely less stable by comparison to the other microservices layering options.

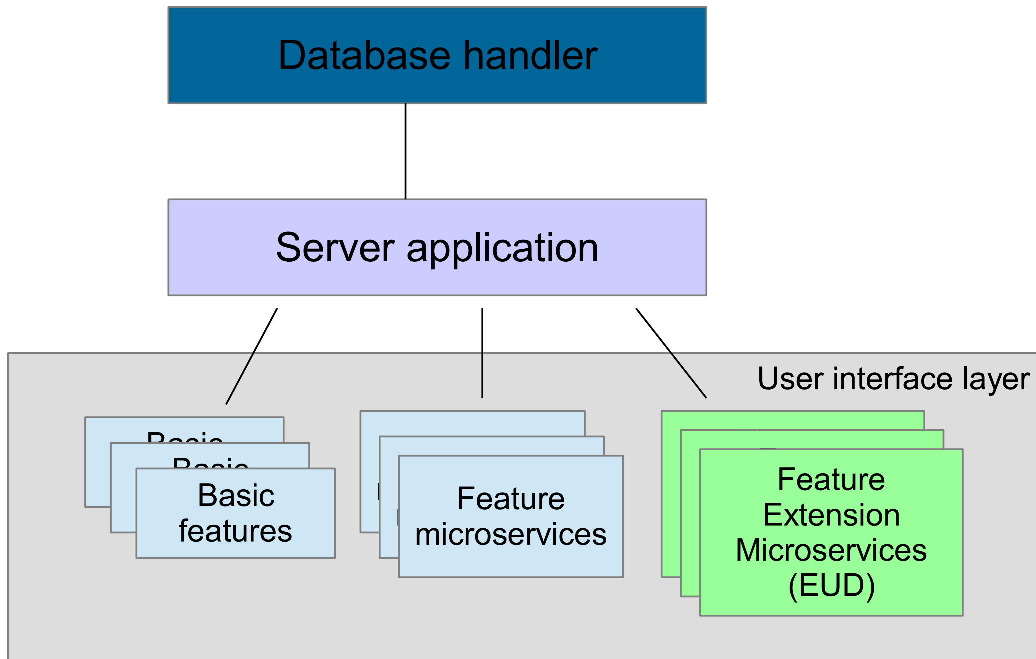

Illustration 2: User interface layer microservices

Figure 2 depicts a hybrid between monolithic and microservices architectures, where the microservices are connected to the monolithic back-end application or dedicated database handler of their own or both of these two. This layering option may be enticing in large transformations of existing web applications to microservices architecture, as the alternative may in that case very well be complete rewrite of the whole software application. This option provides structure to further development, as new features may be developed as microservices while the architectural transformation is in place.

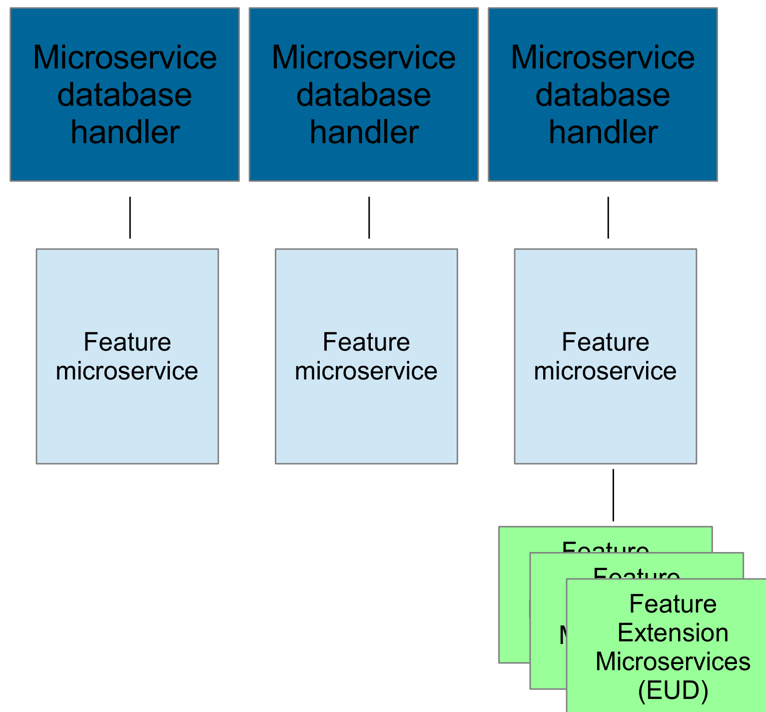

Illustration 3: Pure microservices layering

Figure 3 depicts pure microservices layering, where the whole software application consists of stand-alone microservices and their database handler instances. This layering option and architecture within is robust, scalable and extendable by design. One issue within this layering option is the possibility of ”data pockets” or effectively hidden data, in a situation where microservices handling similar data entities use different database handlers as a result of continued development and use of microservice features combined with lateral expansion.

These explored microservices layering options can also be seen as stages or tiers of transformation from pure monolithic architecture to pure microservices architecture. Pure microservices architecture should also retain some concept of dependent microservices, as basic database operations are likely better off as consolidated API interface regarding end-user development. Web application development usually leaves database handling activities to database handler instances and this factor effectively renders all microservices as dependers, as they require a connection to a database handler in order to interact with the data. Another option would be structuring the microservice so that it also acts as a database handler for itself, but as the database handlers can also be defined as microservices, some flexibility within the definition of microservices should be expected.

5.1.3 Microservice granularity

Defining the responsibilities of microservices is essential in order to develop them in accordance to the definition of microservices. Single-task orientation guides responsibility definitions towards ”one functional requirement implementation per microservice” approach. Given the possibility that the requirements are not directly connected to only a single data entity model, this approach may not fully cover all of the necessary microservice requirements. With this consideration, also the database structure may be used to determine additional microservices. Combining these two approaches covers most bases for microservices of the application, expressively ”microservice for each functional requirement or data entity”. However, extreme subdivisioning of responsibilities should be avoided in order to reduce the amount of microservice dependencies. Attachment file handling is one example, as that can be defined as one responsibility, yet it includes responsibilities for attachment file upload and download. Therefore some abstraction is needed in order to retain some granularity within the microservices structure.

Responsibility definitions also effectively generate guidelines for microservices' further development, as the responsible microservices can be generally identified for the new methods and functional requirements under implementation. Some components of the web application may also be connected to multiple underlying microservices in order to provide effective user experience, examples including the user interface. In this case, while it is also possible to define user interface within each microservice, it might be more beneficial to extract the user interface related features into another microservice under the task ”user interface handling”. These interconnected microservices expect aforementioned leniency within the definition, as they require connections to other microservices.

5.1.4 Internal communication definitions

Gradual transformation may require certain specifications to be made for host application microservices intercommunication in order to keep the software application operational in the transition phases to the use of the open API. There is also some security incentive in keeping another level of (closed) API available, so that the end-user developed microservices might not have direct access to databases or database tables containing sensitive information. This allows monitoring the API activity on multiple levels and potential tampering actions are easier to spot and react to.

Depending on the setup, the microservices can be developed to handle their respective open API requests themselves, circumventing the need for another microservice for the task of handling open API requests from the end-

user developed microservices. In either case, the open API handler, be it a microservice or the web server application, relay the API calls to the appropriate microservice handlers. If the microservices are structured to offer their open API functionality, they need to be exposed to the internet as separate service instances and they require some level of HTTP message handling, which in turn generates additional communications overhead. In this sense, it may be better if the open API is defined as another microservice that relays the HTTP requests within the local network interface to the appropriate microservices and the requirements for exposed services and HTTP communications handling are lessened.

5.1.5 External communication definitions

External communication definitions are used in defining the open API endpoints as well-structured definitions easen the end-user development activities as intuitivity increases. Each microservice should offer a basic set of activities that are available for open API usage where applicable. With this approach, the open API handler can expect a basic set of operations in addition to the more case-specific ones, which in turn offers support for the interface structure of the microservices.

In order to support the end-user development of microservices, the open API should also offer up-to-date viewmodels for communication with each underlying microservice, so that the viewmodels are also respected as a main way of communication between the feature extension microservices and the open API. This ties into the documentation of the open API features and exemplary viewmodels with attribute definitions should be also available programmatically using the API. This allows the end-user developers to examine the characteristics of the viewmodel-based communication with multiple available options to suit varying approaches to the feature extension development.

5.1.6 End-user development support

End-user development or ”extending the web application using feature development from the users of the application” is mainly enabled by API discovery and documentation features. These serve as ways for the user to realize the possibility of feature extensions and to help in the development by exposing API functionality to the users. As with any other types of development, well-written documentation, in this case of the API features, is especially beneficial, as the documentation writers single-handedly deliver most of the development support to the end-users. While reverse-engineering stays as a valid option to learn the characteristics and the use of the open API, documentation accelerates this necessary learning process to divert the focus of

the end-user developers from the API characteristics towards the original goals, i.e. the feature extensions.

API help documentation should at least consist of the user-visible open API features in a handy cheat-sheet like format as well as another feature-based format. Automatic generation of the help documentation is useful tool to extract commented open API functions' specifications, but even then, some consideration to the output and looks of the documentation should be put in place, as assumably bad, missing or outdated documentation is another factor that is likely to lessen the interest in feature extension development by the end-user. It should be also noted that while code examples for specific tasks are another commonly used method of teaching API features, these offer very limited views to the possibilities offered by the API when it comes to actual end-user development.

5.1.7 Microservice extension support

In order to benefit from the end-user developed microservices, some relay specifications should also be put in place. As with the monolithic development architecture, the feature set is what it is, the option to extend the feature set benefits from feature extension discovery, so that the users of the application may find and use these features with relative ease. The initial repositories or microservice discovery channels can be categorized as the host application developers' microservices, including the open API, general population microservices developed by end-users and the organization customer specific repositories that are intended to deliver ”in-house” microservices to other employees of a customer organization. The qualities of the offered microservices vary based on their respective repositories, as microservices delivered by the host application developers can certainly be considered to be of higher quality and stability than the wild varieties of end-user developed microservices and customer organization repository enables the end-user developed microservices to retain possible competitional advantage within the customer organization.

Monitoring the use of the feature extensions within the general population repository allows the host application developers to include high-quality or high-demand features to the host application set either via rewriting the feature in to the host application developers repository as a main microservice or other means of acquisition. This is a major benefit for enabling the end-user development, as the unexpected demand for specific features can be then used to steer the feature development of the web application.

5.1.8 Lasting compatibility

Continued development of features may introduce incompatibility between the host application microservices and connected end-user microservices, as changes to the delivered viewmodels or inner workings of a open API functionality affect the dependers. Microservice architecture enable the use of versioning the features so that any given version of a feature can be kept in the roster of the microservices as long as they are necessary. This necessity of a feature version availability can be determined by monitoring the usage of said feature version. Large amounts of usage is the main deciding factor between keeping and dropping a particular feature microservice from the set of available ones. Promoting the new, assumably better, versions of the microservice in the microservices discovery and documentation is one of the main ways to steer the end-user developments towards the newer versions, so that the intended effect is for the older versions of the microservices to eventually wither away.

This type of continued development process involves the duplication of the microservice under further development to another microservice and as these are complete applications by definition, this duplication can be done with a distinct ease as the whole structure is kept intact. Versioning the feature microservices also requires either version or identifier based access to any available microservice, so that the dependencies can be managed. Monitoring the use of the microservices in end-user development benefits from the segregation of the API layers and requires at least some level of tracking the use of the microservices in order to recognize dependencies further in the lifespan of the software application.

The open API handler can also be tuned for improved error handling by offering a fallback option for requests. This enables microservices with incremental updates to be used with more consistency in that should one microservice fail to handle the request, previous version of the microservice may be used to render the expected functionality until the intended version of the microservice is fixed. The open API handler can in this case also notify the related developer parties for attention to the issue.

5.1.9 Construct implementation model

Table 3: Construct implementation requirements

Explored characteristic	Requirement
Functional requirements of the host application	The API relays the functional requirements of the host application as well as the functional requirements of the microservices. The API

	functionalities can be extended with additional and external microservices.
Microservice architecture layering	The implementation should follow one of the explored layering options, in this case the extended monolithic architecture, i.e. microservices work with the host application back-end during the transition phase
Microservice granularity	Microservices responsibilities are assigned by feature or database element
Internal communication definitions	Microservices should communicate using the communication protocol definitions of the API, such in this case, the API end-points, HTTP and JSON.
External communication definitions	Viewmodels are to be used in communication with the user interface and user. The viewmodel definitions should also be available for end-user development support from any related microservice
End-user development support	The implementation should offer API help documentation as overview and per-microservice basis, in order to support API discovery and in determining the particularities of each microservice
Microservice extension support	The implementation and microservices exposure via the open API should offer suitable routing methods to enable the use of multiple microservice repositories. These repository options initially include developers' repository, general users' repository and customer organization repository
Lasting compatibility	The open API should offer alternate or previous versions for microservices to use in case of faulty implementations.

Table 3 compiles the previously explored characteristics of the microservices-enhanced web application for the implementation specifications. As the effects of these specifications are later on reviewed in comparison to the implementation, these specifications can also be defined as hypotheses for the study.

5.2 Development

This subchapter explores the construct implementation, technologies used in the development process, technology decisions and microservice implementations. Chapter 5.2.1. presents the host application within this implementation context, "creato.red" and it's functional requirements.

5.2.1 Host application: creato.red

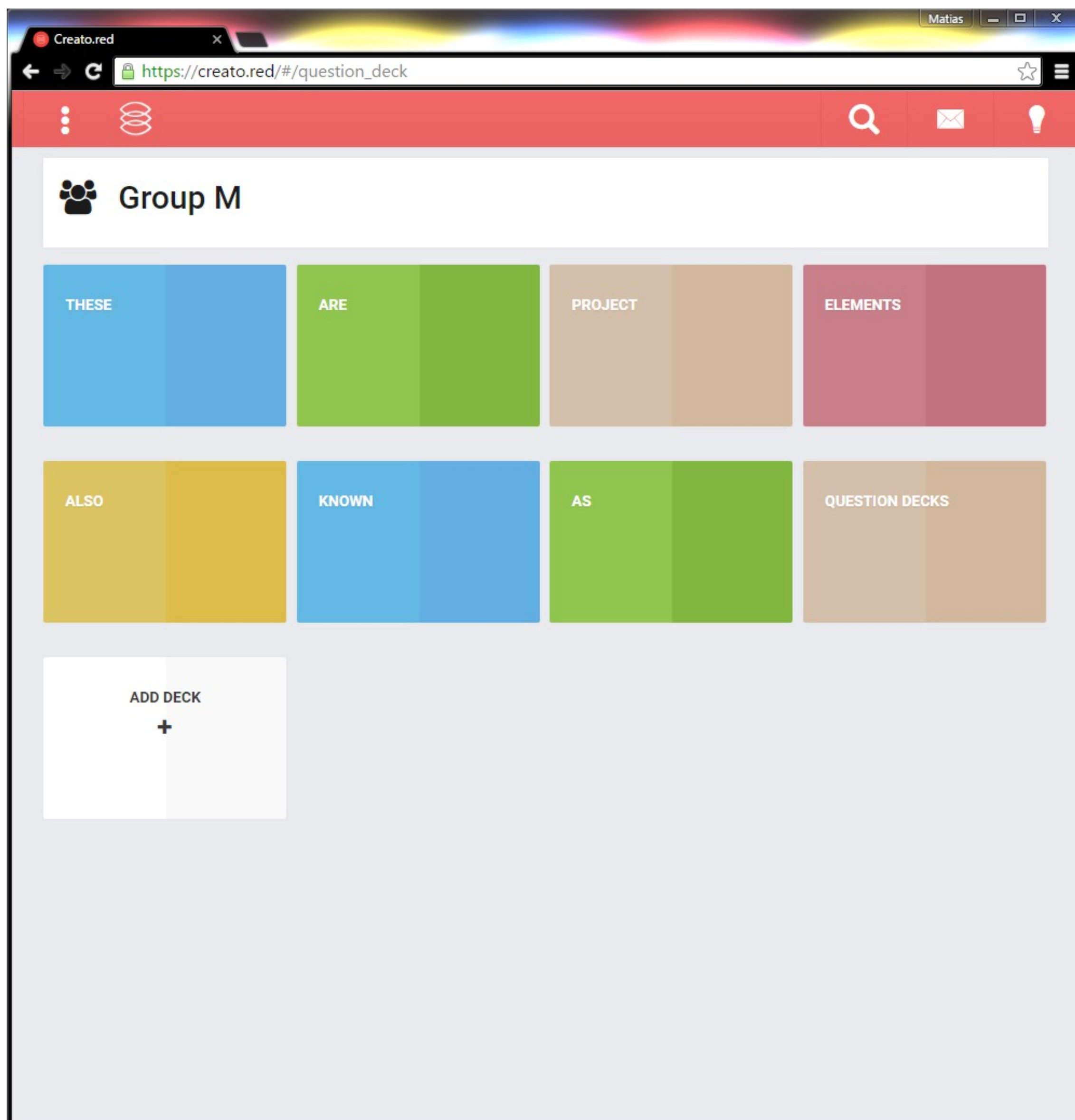

Illustration 4: Creato.red first level deck view

Creato.red is a question-based note-taking web application. Main functional requirement of the application is to enable end-users to ask and answer questions within a particular end-user group. These questions can be sorted in

to "decks" of questions, where each of these decks may represent a specific theme or event that the questions are related to. Exemplary use cases of the creato.red-application include product development meetings, where the application and it's reusable question-based format can be used to refocus the meeting towards solutions for identified issues from previous meetings. This question-based structure is also applicable to other unrelated themes, such as defining work processes or defining a thesis structure.

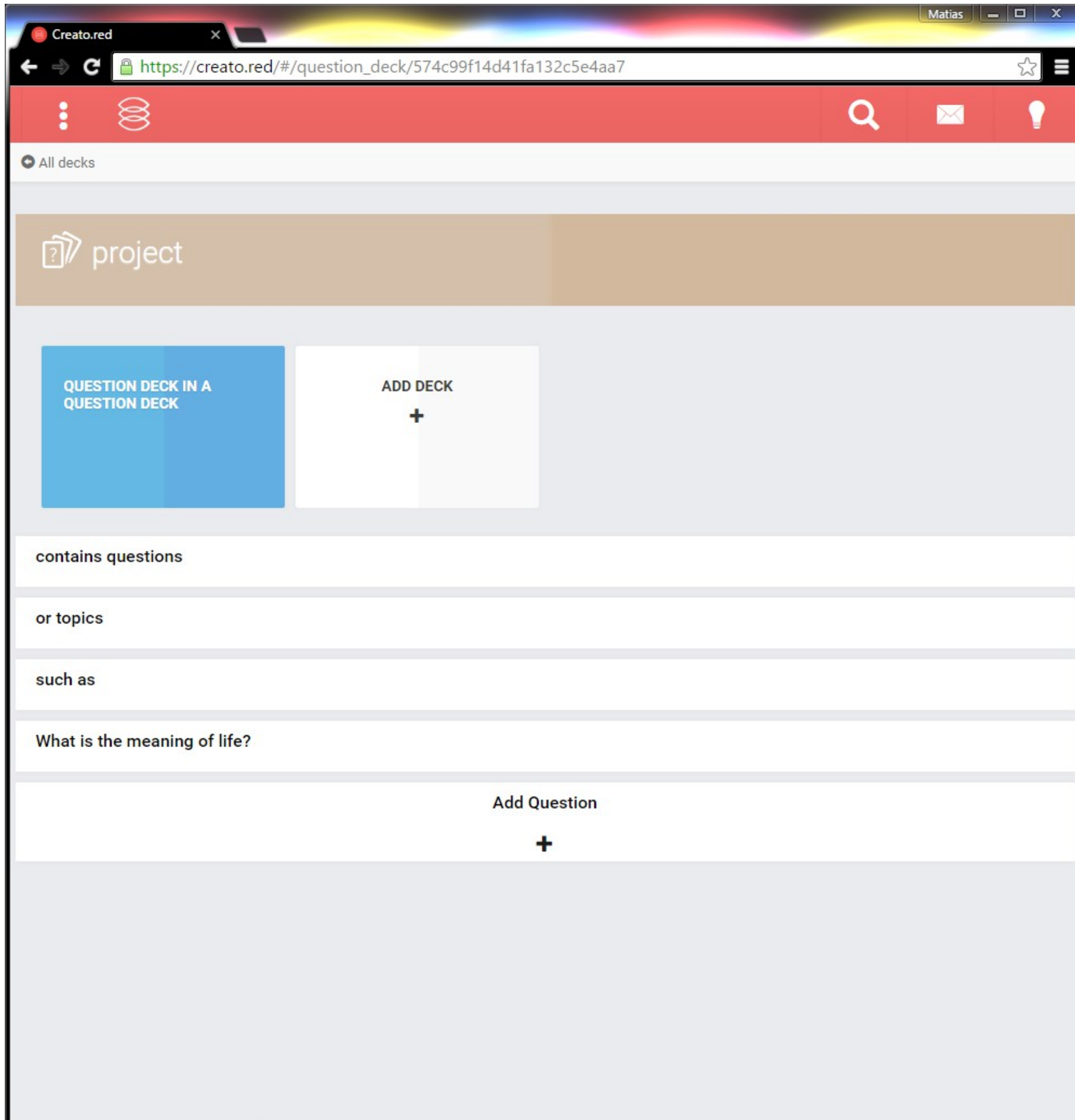

Illustration 5: Question deck view

Figure 5 presents the contents of a question deck element. Question decks can contain questions and other question decks inside them. These question decks within question decks are intended to offer additional structure to larger question decks, where questions can be sorted further.

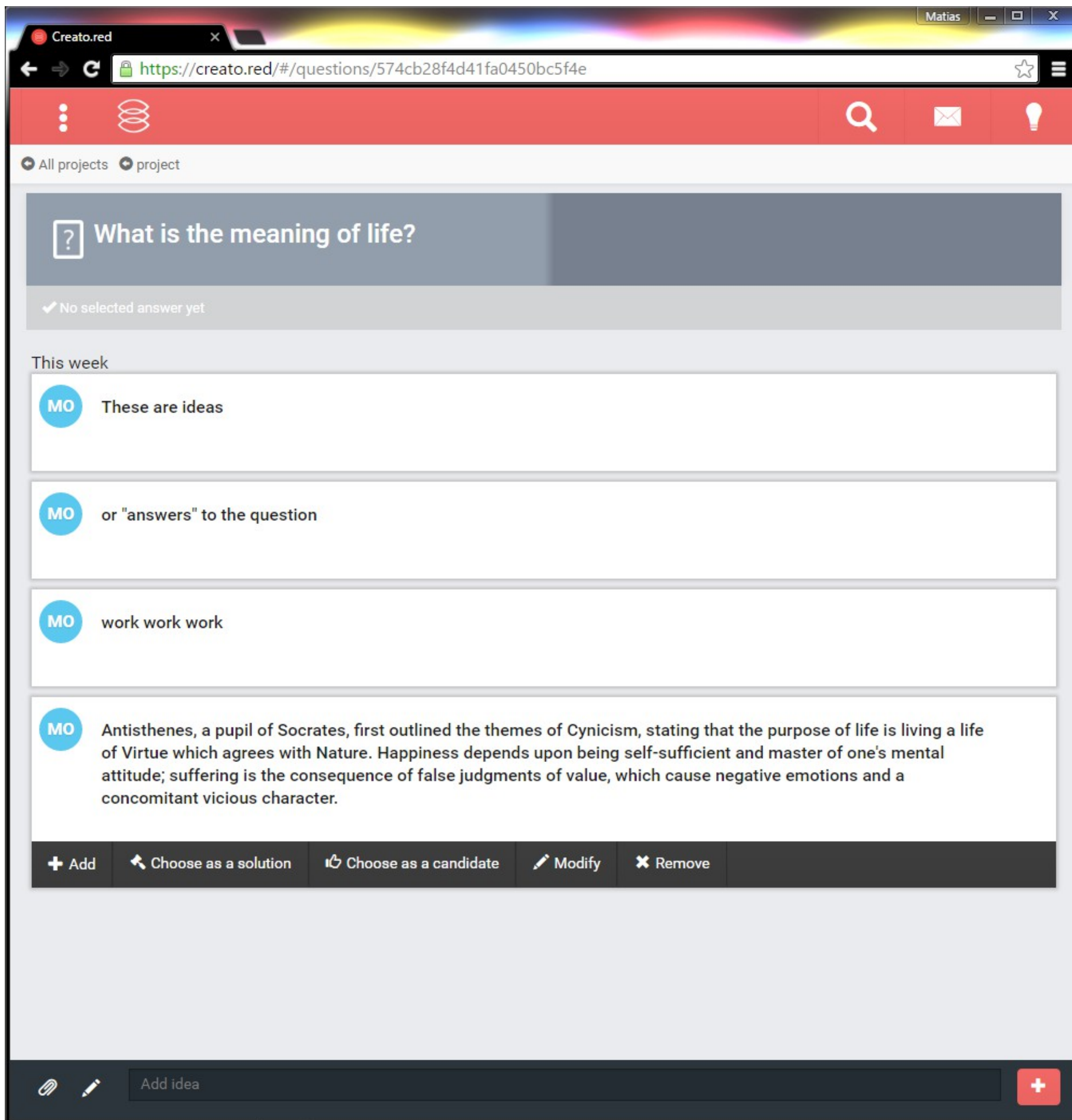

Illustration 6: Question element contents

Figure 6 presents the question element contents. These ideas can contain text content or various attachment files, including drawings, images, sound files and so forth. Ideas can then be nominated as candidates for the solution of the question as well as chosen as solutions for these questions. These sub-types of the idea elements may then be used later on to figure out past decisions. Question decks, questions and ideas are the three types of data models used by the application.

Functional requirements of this host application are:

1. User should be able to create, read, update and delete
    1. Question deck elements
    2. Question elements
    3. Idea elements

2. User should be able to attach images to question deck and idea elements.
3. User should be able to attach other files to idea elements, where recognized image and audio types can be viewed and played. Other types of files can be downloaded.
4. User should be able to search text in question deck, question and idea elements
5. User should be able to export question deck, question and idea elements in printable format
6. User should be able to choose ideas as solutions to questions and nominate ideas as candidates for solution choices.
7. User should belong in one or more groups that have separate, private collections of question decks.

5.2.2 Host application technologies

Creato.red uses a combination of ASP.NET MVC back-end server application that handles the data interaction calls from React -based user interface. ASP.NET MVC according to the website http://www.asp.net/mvc is introduced with ”ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and that gives you full control over markup for enjoyable, agile development. ASP.NET MVC includes many features that enable fast, TDD-friendly development for creating sophisticated applications that use the latest web standards.” React (the user interface technology, https://facebook.github.io/react/) is defined by the React developers with ”React is a JavaScript library for creating user interfaces by Facebook and Instagram. Many people choose to think of React as the V in MVC.”. Database technology used by the host application is MongoDB.

5.2.3 Implementation technology decisions

The implementation of the microservices, including the API, are implemented using Node.js. According to the website (https://nodejs.org/), ”Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.”. The user interface is Javascript-based as well, which improves the development process as a whole and the interaction between the microservices. Central Node packages used by the microservices include Express and Mongoose. Express is a ”minimalist web framework” for Node.js, which makes the route definition and HTTP request handling easy for web

applications such as the microservices here. Mongoose is one of the main MongoDB connection handlers for Node applications.

5.2.4 Construct model-based implementation exploration

As the host application is initially divided into the user interface and back-end server application, the implementation of the web API can be inserted between these two portions. The Web API filters the requests from the user interface to the microservices and the ”legacy” back-end system to gradually change the system from the monolithic architecture to the microservice architecture via the hybrid architecture.

Functional requirements of the host application

Functional requirements of the host application include handling of the previously introduced data elements, question decks, questions and ideas. Besides the content handling requirements, there are requirements for user and group information handling. The use of the application is limited to registered users and the ideas and other content elements are attributed to the users that created them. The content elements are visible to users based on the user groups the users belong to. Public creato.red applies this user group structure to limit the users' access to the content elements and offer virtual privacy in that form. Use cases also include private creato.red -based clusters where the users are grouped based on their ”security clearance”, so that company management related data, for an example, is only available to the management personnel. The implementation includes a deck microservice that handles question deck related tasks. Further development goals include microservices for all of the content elements, including questions and ideas microservices.

Microservice architecture layering options

Extended monolithic layering (layering option 2 as introduced before) is the layering option used for this implementation, as the legacy back-end server handles the API requests that are not handled by microservices at this point in the transition.

Microservice granularity

Microservice granularity within the implementation is single-task based. Microservices of the implementation consist of the API microservice, API help microservice that offers support for the end-user development activities regarding the API functionalities and ”decks” microservice which handles the requests related to the question deck data model.

Internal communication definitions

As the microservices are Javascript-based, Javascript Object Notation, ”JSON” is prominently available as a internal communication format. The database

connection to MongoDB consists of Mongoose Schemas that define the internal communication format between the microservices and the database. JSON is also previously used in the internal communication between the host application user interface and the legacy back-end server.

External communication definitions

The Web API uses view-models in JSON format to supply the user interface with the data element instances. Goals for further development include variety of text-based view-model formats, such as XML, as the JSON elements can be easily converted to these as well.

End-user development support

End-user development is supported in the implementation with the API help microservice, which collects the available documentation from the microservices in HTML format. The help documentation should also be directly downloaded from the microservices in JSON format and this is also implemented for the ”decks” microservice.

Microservice extension support

Microservice extension support is rather light at this point of the development, but the possibilities offered by the Express framework enable the API to extend using a database table that defines routes to the end-user developed microservices. These can then be offered to the user base as optional functionalities within the application.

Lasting compatibility

The Express framework enables the use of ”fallback” routing, where an alternate version of a microservice can be used in place of another to ensure at least some level of performance, should the primary microservice fail to perform the requested task. This offers some stability to the workings of the web application, as user input is likely to be saved at some point within the fallback routing chain.

5.2.5 Implementation-phase decisions

List of legacy back-end artefacts :

- ASP.NET Sessions
- ASP.NET Cookies
- Login tokens
- Secure HTTP (HTTPS)

As the legacy back-end server application is ASP.NET MVC-based and some artefacts of prior development include ”cookies” that the server

application uses to determine users' ”sessions” with the application, these session cookies are handled by the API in the implementation. Users using the user interface request content elements from the legacy server application and the microservices using a API key. The session cookies are saved by the API for each API key and then used in the communication between the legacy server and the web API to accommodate the legacy servers needs for session handling. These session elements in the legacy server contain user information and information about currently selected group. While the transition is underway to the microservices architecture, these artefacts are left in the legacy system as changes to the legacy system can be seen as unnecessary work.

Login tokens are another artefact from the legacy back-end and these tokens are encrypted strings that contain information about the user and the login details of the user. The login tokens are used in the legacy server to determine the validity of the user login information, as the token contains an expiration date. After the expiration date the user is prompted to log in again to the web application and as the token can be saved by the user's browser, it can be reused before the expiration to continuous use of the web application with lesser need for constant login actions. The API handles these tokens by generating API keys based on the tokens and while the implementation does not include automatically renewing the login tokens for the API keys, this can also be handled to an extent using the API.

Secure HTTP or HTTPS is, while necessary for secure user input handling over the web, somewhat problematic for the handling between the legacy back-end server and the microservices, as the HTTPS handling methods differ between these applications. While the public creato.red -application offers a valid HTTPS security certificate, the testing environment uses ”self-signed certificates” that are considered vulnerable from the security stand-point. In testing environment these self-signed certificates are accepted, but this should not be the case for the public implementation as the public implementation would be expected to offer valid security certificates for the HTTPS protocol.

6 RESULTS

This chapter consists of review of construct based on the planning specifications and comparison to similar, popular web application development frameworks, namely Django and WordPress. This follows the ”Design Evaluation” guideline of the design science paradigm.

6.1 Implementation construct review

6.1.1 Implementation construct review on the planning model

The implementation of the microservices architecture based construct model contains the main characteristics of the architecture and the model. While the whole back-end server is not covered by the implementation covered by this document, this implementation offers a software solution for the extended monolithic layering option, as described chapter 5.1.2, in that can be easily extended to transform the web application to pure microservices layering within a further transformation project.

The implementation for handling the end-user developed microservices is also missing from this implementation construct. While the microservice handler implementation may vary upon the target server system characteristics, the applicable model should include an user interface to register and upload microservices to a repository server. The repository server then runs the microservices and offers endpoints for the API in order to deliver the microservices to the general population of the users.

6.1.2 Planning model construct comparison

As the microservices can described as ”building blocks” of the web applications as well as web sites, it may be suitable to compare the planning construct to

similar, popular frameworks used in web site and web application development. Following chapters examine the planning model construct in comparison to these frameworks.

6.1.3 Django

Django is, as described by the developers, ”a high-level Python Web framework that encourages rapid development and clean, pragmatic design.” (https://www.djangoproject.com/). The framework consists of the Django core application and Django packages or ”apps” which can be combined to achieve development goals. This framework shares similarities to the microservices architecture, as the Django packages resemble microservices and could easily be deployed following the microservices architecture so that the Django-based web application would consist of several underlying Django-based applications to offer the functional requirements of the web application.

While the microservices architecture doesn't assume or expect anything from the underlying technologies of the microservices, the Django applications could be also used in combination with other frameworks and application development models to offer the desired non-functional requirements. However, it could be argued that the microservice architecture isn't an explicit core concept of the Django framework and package-based monolithic architecture is prominently featured within the framework and this slight difference sets the planning construct apart from the development architecture using the Django framework.

6.1.4 WordPress

WordPress, as described by the developers, is ”web software you can use to create a beautiful website, blog, or app […] built on PHP and MySQL”. The architecture consists of the WordPress core application and ”plugins” that extend the functionalities of the web sites and web applications based on the WordPress technology. While the architecture is or resembles monolithic architecture, the plugins offer customizable functional requirements not unlike the microservices in a microservices architecture -based application. Due to the monolithic architecture base of the WordPress applications, the suitability of this technology within a microservices architecture is at least questionable, assuming that in such a context, each of the necessary WordPress applications in a microservice-like role would include the core WordPress application as well, in turn generating one exquisite ”jungle” consisting of WordPress configurations and settings web sites.

6.2 Construct analysis

This chapter contains analysis of the implementation regarding issues of the reimplementation of the non-functional requirements or technical details as well as overall analysis of the success of the implementation.

6.2.1 Implementation issue analysis

Analysis of construct based on expected results and planning specifications is presented within this chapter. Analysis of the implementation based on expected results, namely a microservice architecture-based platform, is that the implementation construct follows the extended monolithic layering option. The implementation offers the initial implementation base upon which the explored characteristics are or can be implemented. While the implementation construct is not complete in that it offers the host application functional requirements as a whole using the microservices architecture, the implementation supports the transitioning project from monolithic to microservice architecture.

Analysis of the implementation based on the planning specifications has been largely done in chapter 5.2.4 with the implementation exploration focus. The main implementation goals were the microservice base structure, legacy back-end server support, microservices for the API, API help documentation and decks regarding functional requirements. The internal and external communication formats were kept as JSON as that format is widely used in the chosen technologies (mainly Javascript on nodejs and React). Microservice extension support and lasting compatibility support were included in the development goals as well.

Main issues regarding the development of the implementation construct were related to the technologies expected by the legacy server, as presented in chapter 5.2.5. These issues were mainly related to non-functional requirements, but the emergence of these issues stress the importance of examination of the communication technical details of the software parts, as extensive knowledge relating these factors is useful in determining the communication definitions of the host application.

6.2.2 Implementation success analysis

Implementation construct succesfully initializes the transition project for the architectural change from monolithic towards microservice architecture and offers implementations for the API, API help and one content-serving microservice. The implementation construct also successfully communicates with the legacy back-end server in order to properly use the extended monolithic layering architecture. The API handles the authentication using the legacy back-end, although this implementation should be one of the central things to be changed to an authentication microservice, further in the transition project.

The prerequisites for the implementations of the fallback system and the microservice repository systems are in place, as these systems can be implemented within the API routing system to redirect calls to specified microservices, be they just microservice versions or placed in a microservice repository. Further implementation analysis regarding these systems serves as a base for further research.

7 DISCUSSION

7.1 Discussion of findings

Business contributions of this study are the primer into microservices this document is and the solutions the microservices architecture may offer for web application development. Underlying business-related issue is the examination of the requirements for enabling the user base to act as a source for non-functional and functional requirements. Following the participatory design based assumption that the workers or actual users of a product know best what is needed to improve the application, enabling the user base can lead to reduced development time of important, new and possibly unforeseen functional requirements leading to more pleasing and efficient software and greater customer satisfaction. In business contribution context, human greed based elements such as copyright and licensing are intentionally avoided as being out of the scope of the study, although they hold a large amount of interest in certain business circles.

Scientific contributions of the study amount to the application of design science in web application development context as well as the planning and implementation constructs that are created during the study. The analysis method used on the constructs can also be seen as a construct for analysis, where the implementation construct is evaluated based on the planning construct and successful technologies or architectures that can be then used to assess the success of the implementation. This is not unlike a software project, where the planning construct resembles the initial project specifications and the resulting software application is then compared to the closest competitors to determine project success.

7.2 Implications and limitations of the study

Business implications of the study are that the microservice architecture is one of the rising web application development architecture technologies at the time of writing of this study. As such, the architecture can be evaluated based on the solutions it offers for various issues in web application development, some of which are examined more closely in this study. While it is acknowledged that the microservice architecture can be overly complex for simple web applications and for some projects the necessary intercommunication of the microservices may prove to be an unnecessary hindrance, the microservice architecture expects and promotes simplicity from the microservice applications that is certainly useful for continuous development and large web applications. Inherent scalability of the microservice architecture is another benefit for web applications that regularly handle heavy web traffic or large amount of users, as the microservices can be deployed on separate server hardware systems.

Limitations of the study include heavy traffic testing or "stress testing" of the microservice architecture -based applications and the comparison of a single application implemented using the monolithic architecture and the same or similar application using the microservice architecture to investigate the difference caused by factors such as the microservice intercommunication overhead that may affect, for an example, the responsiveness of the user interface between the applications. The implementational prerequisites for the fallback and repository systems are in place but the implementations are missing from the implementation construct and this is one of the limitations of the study as the systems are examined mainly on the theoretical level.

7.3 Future research options

Options for future research in the microservice architecture include previously mentioned implementation comparison of monolithic and microservice architecture in stress testing and overhead contexts. Assumably the microservice architecture also contains other characteristics that can be revealed in the development projects of microservice projects and may prove worthwhile to examine within the design science paradigm. Issues relating to the microservice architecture based development are also left mostly untouched within this study and these are probably more easily determined within microservice architecture based development projects.

Using the planning construct as examined within this study as a base for new web applications using the pure microservice layering option is another option for future research, as here the planning construct is used in conjunction

with an existing monolithic architecture based host application to initialize a transitioning project towards a microservice-based architecture.

7.4 Conclusion

As a general conclusive remark, microservice architecture offers benefits over monolithic architecture at least in certain situations. These situations include those with the issues described in chapter 1.1, namely software application not satisfying the needs of the actual users, utilizing beneficial innovations arising from the use of the application or stability issues caused by continuous development in web application development. For these issues, the microservice architecture offers solutions with the Web API and API help documentation that support the end-user development, repository system for the delivery of the end-user developed microservices and fallback system for microservice failure recovery and web application structure robustness.

The study also offers background theory for the questions that may arise such as why enabling the end-user development is beneficial for the software application and why the users would contribute to the software application. In short, the answers provided to these questions are that the end-users, the workers or the actual users of the application know best what improvements could affect the efficiency of the application especially in work tool context and the users would contribute to the application for various reasons, most importantly to improve their work performance.

The planning and implementation constructs created for this study offer a baseline for microservice architecture based web application development, offering more hands-on data on the implementational workings of a microservice based application. The planning construct is written in a more general level to cater to a wider array of (web) applications for transitioning project purposes as well as new developments. The implementation construct offers one point of view to the actual use of the architecture and the planning construct.

All in all, this study should offer a versatile examination of the microservice architecture, the development context benefits of the use of the architecture and business incentives to utilize the architecture in software development projects.

LIST OF REFERENCES

Alpers, S., Becker, C., Oberweis, A., & Schuster, T. (2015, September). Microservice based tool support for business process modelling. In *Enterprise Distributed Object Computing Workshop (EDOCW)*, 2015 IEEE 19th International (pp. 71-78). IEEE.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5), 61-72.

Cohen, D., Lindvall, M., Costa, P. (2003). Agile software development. *DACS SOAR Report,* 11.

Danado, J., Davies, M., Ricca, P., & Fensel, A. (2010). An authoring tool for user generated mobile services. In *Future Internet-FIS 2010* (pp. 118-127). Springer Berlin Heidelberg.

Hars, A., Ou, S. (2001). Working for free? Motivations of participating in open source projects. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on* (pp. 9-pp). IEEE.

Grance, T., Mell, P. (2011). The NIST Definition of Cloud Computing. *NIST Special Publication 800-145.*

Krylovskiy, A., Jahn, M., & Patti, E. (2015, August). Designing a Smart City Internet of Things Platform with Microservice Architecture. In *Future Internet of Things and Cloud (FiCloud)*, 2015 3rd International Conference on (pp. 25-30). IEEE.

Namiot, D., Sneps-Sneppe, M. (2014a). On IoT Programming. *International Journal of Open Information Technologies*, 2(10), 25-28.

Namiot, D., Sneps-Sneppe, M. (2014b). On M2M Software. International Journal of Open Information Technologies, 2(6), 29-36.

Namiot, D., Sneps-Sneppe, M. (2014c). On micro-services architecture. *International Journal of Open Information Technologie*s, *2(9)*, 24-27.

Namiot, D., Sneps-Sneppe, M. (2014d). Micro-service Architecture for Emerging Telecom Applications. *International Journal of Open Information Technologies*, 2(11), 34-38.

Namiot, D., Sneps-Sneppe, M., (2012). About M2M standards and their possible extensions. In *Future Internet Communications (BCFIC)*, 2012 2nd Baltic Congress on (pp. 187-193). IEEE.

Newman, S. (2015). Building microservices. (1st edition). California: O'Reilly Media

Maleshkova, M., Pedrinaci, C., & Domingue, J. (2010). Semantic annotation of Web APIs with SWEET.

Merriam-Webster. (2016). Definition of end user. http://www.merriam-webster.com/dictionary/end%20user Referenced 9/5/2016

Potvin, P., Nabaee, M., Labeau, F., Nguyen, K. K., & Cheriet, M. (2015). Micro Service Cloud Computing Pattern for Next Generation Networks. *arXiv preprint arXiv:1507.06858.*

Reenskaug, T. (2007). *The original MVC reports.* Oslo: Dept. of Informatics, University of Oslo.

Schuler, D., Namioka, A. (Eds.). (1993). *Participatory design: Principles and practices.* CRC Press.

Titlestad, O., Staring, K., Braa, J. (2009). Distributed Development to Enable User Participation: Multilevel design in the HISP network. *Scandinavian Journal of Information Systems.* 21(1), 27-50.

Uckelmann, D., Harrison, M., Michahelles, F. (2011). An architectural approach towards the future internet of things. *Springer Berlin Heidelberg*, 1-24.

von Alan, R. H., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly, 28(1)*, 75-105.

Wiegers, K., Beatty, J. (2013). *Software requirements*. Pearson Education.

W3Techs. (2016). Usage of server-side programming languages for websites. Accessed 9/19/2016. Available online: https://w3techs.com/technologies/overview/programming_language/all

APPENDIX

API microservice
src/app.js

```
var express = require('express');
var http = require('http');
var https = require('https');
var querystring = require('querystring');
var bodyParser = require('body-parser');
var util = require('util');
var fs = require('fs');

var lutilr = require('./utility.js');
var lutil = new lutilr.writer();
var urlConfig = require('./config');
var rootUrl = urlConfig.rootUrl;
var relay = require('./relay.js')(rootUrl);
var mongol = require('./databaseconnection/mongol.js');

var port = process.env.PORT || 16100;
var app = express();

var error = require('./error.js');

var deployPath = process.env.deployPath || "";
var STATIC_HEADERS = {
  'Allow': 'GET,PUT,POST,DELETE',
  'Access-Control-Allow-Headers': 'X-Auth-Token, Content-Type, Accept',
  'Access-Control-Allow-Origin': urlConfig.acceptedOrigin,
  'Access-Control-Allow-Methods': 'GET,PUT,POST,DELETE'
};

var decksrouter = require('./routers/decksrouter.js');
var apikeysrouter = require('./routers/apikeysrouter.js');

app.get(deployPath + '/favicon.ico', (req, res) => {
  res.sendStatus(404);
});

app.get(deployPath + '/testreq*', (req, res) => {
  var inspected = util.inspect(req);
  inspected = inspected.replace(',', ',\n');
  res.send(inspected);
});

app.use('/apikeys', apikeysrouter);

app.options(deployPath + '/*', function(req,res) {
```

```
  res.set(STATIC_HEADERS);
  res.sendStatus(200);
});

var no_token = function(req, _write, res) {
  no_token_callback(req, _write, function (reply) {
      back = {}
      reply.on('data', (chunk) => {
        back['response_body'] = chunk;
      });
      reply.on('end', () => {
        res.set(STATIC_HEADERS);
        res.send(back.response_body);
      });
    });
};

var no_token_callback = function(req, _write, callback) {
  var no_token_quest = relay.getRequest({
      path: req.url,
      _query: req.query,
      _body: req.body,
      _write: _write,
      method: req.method,
      headers: req.headers,
    }, (reply) =>{
      // mongol.handle_cookies(reply);
      callback(reply);
    } );
  no_token_quest.end();
};

var checkToken = function (req, res) {
  var has_token = req.headers['x-auth-token'];
  var userid;
  if (!has_token) {
    if (urlConfig.TESTING && req.query.token) {
      has_token = req.query.token;
    } else {
      res.set(STATIC_HEADERS);
          res.status(400).send(JSON.stringify({error: 'missing token', more: 'token-in-query:' +
req.query.token, query: req.query}));
      return false;
    }
  }
  return has_token;
};

app.use(bodyParser.json());

app.post(deployPath + '/logout*', function(req, res) {
  no_token(req, JSON.stringify(req.body), res);
```

```
});

app.post(deployPath + '/login*', function(req, res) {
  no_token_callback(req, JSON.stringify(req.body), function(reply)
  {
    back = {}
    reply.on('data', (chunk) => {
      try {
        back.jobj = JSON.parse(chunk);
      } catch (e) {
        back.jobj = false;
        logging(chunk);
      }
    });

    reply.on('end', () => {
      if (!back.jobj) {
        res.set(STATIC_HEADERS);
        res.status(400).send({error: true, msg: 'login failed'});
      } else {
        console.log('the stuff we got: ', back.jobj);
        mongol.generate(back.jobj.token, (robj) => {
            mongol.handle_cookies(robj.apikey, reply);
            back.jobj.token = robj.apikey;
            back.chunk = back.jobj;
            res.set(STATIC_HEADERS);
            console.log('then this happened: ', back);
            res.send(back.chunk);
        });
      }
    });
  });
});

app.get(deployPath + '/upload*', (req, res) => {
  out = {};
  changed_url = undefined;
  var queries = req.query;
  var tokened = function(req) {
    no_token_callback(req, querystring.stringify(req.query), (reply) => {
      reply.pipe(res);
    })
  }

  if(queries.token) {
    mongol.retrieve(queries.token, (result) => {
      token = result.token;
      changed_url = req.url.replace(queries.token, token);
      queries.token = token;
      req.url = changed_url || req.url;
      req.query = queries;
      tokened(req);
```

```
    })
  }
});

app.use('/projects', decksrouter);

app.get(deployPath + '/get_group/:id',  (req, res) => {
  var has_token = checkToken(req, res);
  if (!has_token) return;
  var target_group_id = req.params.id;
  var target_path = '/users/mygroups';

  var get_groups = function(token, cookies, callback) {
    var quest = relay.getRequest({
      path: target_path,
      method: 'GET',
      _write: '',
      headers: req.headers,
      cookies: cookies,
      token: token
    }, (reply) => {
      lutil.write_out('reply.txt', util.inspect(reply));
      var outs;
      reply.on('data', (chunk) => {
        outs = chunk;
      });
      reply.on('end', () => {
        lutil.write_out('end-data.txt', outs.toString() );
        callback(outs.toString());
      })
    });
    quest.end();
  };

  function handle_group_sending(users_groups) {
    // console.log(users_groups);
    // console.log('trying to find ', target_group_id, ' from users_groups..');
    var send_back;
    try {
      var json_groups = JSON.parse(users_groups);
      // console.log(json_groups);
      for (var i = 0; i < json_groups.length; i++) {
        var group_elem = json_groups[i];
        if (group_elem.id == target_group_id) {
          send_back = group_elem;
          break;
        }
      }
    } catch (e){
      console.error('json parse failed');
    }
```

```
    if (send_back) res.send(send_back);
    else {
      new error('group not found', [users_groups]).send(res);
    }
  }

  mongol.retrieve(has_token, (reply) => {
    console.log('got token: ', reply.token);
    mongol.get_cookies(has_token, (cookies) => {
      get_groups(reply.token, cookies, handle_group_sending);
    });
  });
});

app.all(deployPath + '/*', function (req, res) {
  // console.log('wtf');
  has_token = checkToken(req,res);
  if (!has_token) return;
  var request = function(token, cookies) {
    out = {};
    var userid;
    var _write = querystring.stringify(req.query);
    if (req.method == 'POST') {
      _write = JSON.stringify(req.body);
    }
    var quest = relay.getRequest({
        path: req.url,
        _query: req.query,
        _write: _write,
        method: req.method,
        headers: req.headers,
        token: token,
        cookies: cookies
      }, (backend_reply) => {
      backend_reply.setEncoding('utf8');
      backend_reply.on('data', (chunk) => {
        out['response_body'] = chunk;
      });
      out['backend_reply'] = backend_reply.statusCode == 200
      backend_reply.on('end', () => {
        mongol.handle_cookies(userid, backend_reply);
        // res.set(STATIC_HEADERS);
        res.set(backend_reply.headers);
        res.send(out.response_body);
      });
    });
    quest.end();
  }

  mongol.adapting_get(has_token, (reply) => {
    userid = reply.apikey;
    mongol.get_cookies(reply.apikey || has_token, (result) => {
```

```
    request(reply.token, result);
  });
 });
});

var certification = {
 key: fs.readFileSync(__dirname + '/cert/key.pem'),
 cert: fs.readFileSync(__dirname + '/cert/cert.pem'),
}

https.createServer(certification, app).listen(port, () => {
 console.log('https listening on ' , port);
});
http.createServer(app).listen(httpport= 16080, () => {
 console.log('http listening on ', httpport);
});
```

src/config.js

```
var Config = {
 rootUrlObj: {
  protocol: 'https',
  host: 'localhost',
  port: 44304
 },
 acceptedOrigin: 'http://localhost:3000',
 database_name: 'api',
 // database_name: 'apitest',
 TESTING: true,
 decks_url: 'https://localhost:16400'
};

Config.agentOptions = {
 host: Config.rootUrlObj.host,
 path: '/',
 port: Config.rootUrlObj.port,
 rejectUnauthorized: !Config.TESTING
};

Config.rootUrl = Config.rootUrlObj.protocol + '://' + Config.rootUrlObj.host + ':' + Config.rootUrlObj.port;

module.exports = Config;
```

src/relay.js

```
var urlConfig = require('./config.js');
var originUrl = urlConfig.acceptedOrigin;
var http = require('http');
var https = require('https');
var querystring = require('querystring');
var url = require('url');
var localutil = require('./utility.js');
var lutil = new localutil.writer();
```

```
var util = require('util');
require('ssl-root-cas').inject();
var error = require('./error.js');

var AddressFormatException = {
  name: 'Address Format Exception',
  message: 'Address format malformed',
  toString: function() {return this.name + ': ' + this.message;}
}

var Relay = function (address, agentOptions) {
  var parsed_address = url.parse(address);
  var host = parsed_address.hostname;
  var port = parsed_address.port;
  var protocol = parsed_address.protocol.split(':')[0];
  if (!agentOptions) agentOptions = urlConfig.agentOptions;
  console.log(protocol, host, port);
  return {
    requestBase: function(options, callback) {
      var base_opt = {
        host: host,
        port: port,
      };
      for (var key in options) {
        base_opt[key] = options[key];
      }
      if (protocol == 'https') {
        var Agent = new https.Agent(agentOptions);
        base_opt.agent = Agent;
        return https.request(base_opt, callback);
      }
      return http.request(base_opt, callback);
    },

    getRequest: function(options, callback) {
      var headers = {
        'Origin': originUrl
      }
      if (options.token) headers['X-Auth-Token'] = options.token;
      if (options.cookies) headers['Cookie'] = options.cookies;
      if (options.headers) {
        for (var key in headers) {
          options.headers[key] = headers[key]
        }
      }

      var request = this.requestBase(options, callback);
      request.on('error', (e) => {
        if (e.code == 'ECONNREFUSED') console.log('Make sure back-end is running at ', e.address,
':', e.port);
        callback(new error('error happened', [e, [host, port]]));
      });
```

```
    if (options._write) request.write(options._write);
    return request;
  },

  preflight: function (path, xac_headers) {
    var options = {
      path: path,
      method: 'OPTIONS',
      headers: {
        'Access-Control-Request-Headers': xac_headers
      }
    }
    var go = this.getRequest(options, (reply) => {
      lutil.write_out('preflighting.txt', util.inspect(reply));
      lutil.write_out('req-headers.txt', util.inspect(reply.headers));
    });
    go.end();
  },

  reply: function (response) {
    console.log('initializing relay.reply');
    return function(reply) {
      var data;
      reply.on('data', (d) => {
        data = d;
      });
      reply.on('end', () => {
        if (data.toString().indexOf('<') == 0) console.log('response probably html');
        // console.log(data.toString());
        response.send(data);
      })
    }
  }
}
};

module.exports = Relay;
```

Routers

src/routers/apikeys.js

```
var express = require('express');

var router = express.Router();
var deployPath = process.env.deployPath || "";
var mongol = require('../databaseconnection/mongol.js');
var urlConfig = require('../config.js');
var error = require('../error.js');

router.get(deployPath + '/*', (req, res, next) => {
  req.token = req.headers['X-Auth-Token'];
  if (!req.token && urlConfig.TESTING) req.token = req.query['token'];
  if (!!req.token) next();
```

```
  else {
    new error('token missing', [req.headers, req.query]).send(res);
  }
});

router.get(deployPath + '/list/', (req, res) => {
  mongol.list(req.token, (reply) => {
    res.send(reply);
  });
});

router.get(deployPath + '/token/', (req, res) => {
  mongol.retrieve(req.token, (reply) => {
    res.send(reply)
  })
});

router.get(deployPath + '/generate/', (req, res) => {
  mongol.generate(req.token, (reply) => {
    res.send(reply)
  });
});

router.put(deployPath + '/update/:newtoken', (req, res) => {
  mongol.update(req.token, req.params.newtoken, (reply) => {
    res.send(reply)
  });
});

router.delete(deployPath + '/delete/:token', (req, res) => {
  mongol.delete(req.token, req.params.token, (reply) => {
    res.send(reply)
  });
});

module.exports = router;
```

src/routers/decksrouter.js

```
var express = require('express');

var router = express.Router();
var querystring = require('querystring');

var config = {
  backend: 'https://localhost:16400',
  TESTING: true,

};

config.agentOptions = {
    host: 'localhost',
    path: '/',
```

```
    port: 16400,
    rejectUnauthorized: !config.TESTING
};

var relay = require('../relay.js')(config.backend, config.agentOptions);

router.all('/*', (req, res, next) => {
  console.log('middleware:: ', [req.method, req.path, req.query]);
  next();
});

router.all('/*', (req, res) => {
  var _write;
  if (['GET', 'DELETE'].indexOf(req.method) < 0) _write = req.body ? JSON.stringify(req.body) : (req.query ? querystring.stringify(req.query) : false);
        // console.log('dsr:: ', req.headers, req.path, req.host, req.port, config.agentOptions.rejectUnauthorized);
  var relayquest = relay.getRequest({
    path: req.url,
    method: req.method,
    _write: _write,
    headers: req.headers
  }, (reply) => {
    if (reply.error) {
      console.error(reply);
      reply.send(res);
      return;
    }
    var sendback;
    reply.on('data', (d) => {
      console.log(d.toString());
      sendback = d;
    });

    reply.on('end', () => {
      console.log('ending and sending projects/');
      res.set({
        'Content-Type': 'application/json',
        'Access-Control-Allow-Origin': 'http://localhost:3000',
      });
      res.send(sendback);
    });
  });

  relayquest.end();
});

module.exports = router;
```

Database connection
src/databaseconnection/mongol.js

```
var mongoose = require('mongoose');
var urlConfig = require('../config.js');

var ApiKeyModel = require('./models/apikey.js');
var CookieHandlerModel = require('./models/cookie_handler.js');

var CookieUtilityTools = require('./models/cookie_utility_tools.js');

mongoose.connect('mongodb://localhost/'+ urlConfig.database_name)

var Mongol = {
  generate: function(actual_token, callback) {
    var apikey = new ApiKeyModel();
    apikey.token = actual_token;
    apikey.save(function (err, saved) {
      if (err) {
        console.error('generating apikey: should not happen')
        callback({failure: true});
        return;
      }
      callback({apikey: saved.id, token: actual_token});
    })
  },
  retrieve: function (apikey, callback) {
    ApiKeyModel.findById(apikey, (err, found) => {
      if (err) {
        console.error('retrieving token: should not happen');
        callback({failure: true});
        return;
      }
      callback({token: found.token});
    })
  },
  adapting_get: function(key_or_token, callback) {
    this.retrieve(key_or_token, (reply) => {
      if (reply.failure) {
        console.log('adapting-reply-failure');
        this.generate(key_or_token, (generated) => {
          console.log('calling adapting_get callback with ', generated);
          callback(generated);
        });
      } else {
        callback  ({apikey: key_or_token, token: reply.token});
      }
    });
  },
  //unnecessary btw maby (flow => generate, use, generate, use)
  update: function(apikey, actual_token, callback) {
    ApiKeyModel.findByIdAndUpdate(apikey, {token: actual_token}, (err)=>{
      if (err) {
```

```
      console.error('updating token: should not happen');
      callback({failure: true});
    }
    callback({failure: false});
  })
},

list: function(actual_token, callback) {
  ApiKeyModel.find({token: actual_token}, (err, docs) => {
    if (err) {
      console.error('listing by token: should not happen');
      callback({failure: true});
    }
    var returnable = [];
    for (var i = 0; i < docs.length; i++) {
      var doc = docs[i];
      console.log(doc);
      returnable.push({apikey: doc.id, token: doc.token});
    }
    callback({'keys':returnable});
  });
},

'delete': function (apikey, token, callback) {
  ApiKeyModel.remove({ id: apikey, token: token }, (err) => {
    if (err) {
      console.error('removing token: should not happen');
      callback({failure: true});
    }
    callback({failure: false});
  })
},

get_cookies: function(userid, callback) {
  CookieHandlerModel.findOne({UserID: userid}, (err, doc) => {
    if (err) {
      console.error('error while getting cookie; ', err);
      callback(false);
      return;
    }
    if (!doc) {
      callback(false);
      return;
    }
    callback(doc.Cookie);
  });
},

save_cookies: function(userid, cookiestring, callback) {
  CookieHandlerModel.findOneAndUpdate({UserID: userid}, {
    Cookie: cookiestring
  }, {
```

```
    new: true,
    upsert: true
  }, (err, doc) => {
    if (err) {
      console.error('something went wrong in save_cookies findOneAndUpdate; ', err);
      return
    };
    callback(doc);
  })
},

handle_cookies: function (userid, reply) {
  CookieUtilityTools.parse_set_cookies(reply, (res) => {
    var cookiestringing = {};
    for (index in res) {
      var row = res[index];
      var splitrow = row.split(';');
      for (e_idx in splitrow) {
        var elem = splitrow[e_idx];
        var parsedelem = elem.split('=');
        if (parsedelem.length == 1) continue;
        if (['path'].indexOf(parsedelem[0].trim()) >= 0) continue;
        cookiestringing[parsedelem[0]] = parsedelem[1];
      }
    }
    var cookiestring = ""
    for (cookie in cookiestringing) {
      var s_cookiestring = cookie + '=' + cookiestringing[cookie] + ';';
      cookiestring = cookiestring + s_cookiestring;
    }
    console.log(cookiestring);
    this.save_cookies(userid, cookiestring, (reply) => {
      console.log(reply);
    });
  });
},

mongooseConnection: function() {
  return mongoose.connection
}

};

module.exports = Mongol;
```

API models

src/databaseconnection/models/apikey.js

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema,
  ObjectId = Schema.ObjectId;
```

```
var ApiKeySchema = new Schema({
  token: String
});

var ApiKeyModel = mongoose.model('ApiKey', ApiKeySchema);

module.exports = ApiKeyModel;
```

src/databaseconnection/models/cookie_handler.js

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;

var CookieHandler = new Schema({
  UserID: String,
  Cookie: String
});

var CookieHandlerModel = mongoose.model('CookieHandler', CookieHandler);

module.exports = CookieHandlerModel;
```

API help microservice

src/app.js

```
var express = require('express');
var app = express();
var internal = require('./internal.js');
var urlConfig = require('./config.js');
var port = process.env.PORT || 16201;
app.use(express.static(__dirname + '\\html'));
app.use(express.static(__dirname + '\\html\\css'));

var sendOpt = {
  root: __dirname + '/html/'
}

var links = [
  [
    {
      href:'/decks/latest',
      title: 'decks'
    },
    {
      href: '/decks/v1',
      title: 'v1'
    }
  ],
  [
    {
      href: '/ideas/latest',
```

```
      title: 'ideas/testing'
    },
    {
      href: '/ideas/v1',
      title: 'i/1'
    }
  ]
];

app.set('views', './src/pug');
app.set('view engine', 'pug');
app.get('/css', function(req, res) {
  res.sendFile('css/css.css', sendOpt);
});

var findLatestVersion = function(servicename) {
  var correct;
  for (var i = 0; i < links.length; i++) {
    var service = links[i];
    for (var j = 0; j < service.length; j++) {
      var servicelink = service[j];
      if (servicelink.href.indexOf(servicename) == 1) {
        correct = service;
        break;
      }
    }
    if (correct) break;
  }
  if (correct) {
    var latest = correct[correct.length-1].href;
    var version = latest.split('/')[2];
    return version;
  }
  // todo return static
  return correct;
}

app.get('/:service/:version', function(req, res, next) {
  if (req.params.version == 'latest') {
    req.params.version = findLatestVersion(req.params.service);
  }
  req.service = req.params.service;
  req.version = req.params.version;
  next();
});

var helproutes = {
  'decks': internal(urlConfig.decks_url)
};

app.get('/*', function(req, res) {
  var title = 'API Help';
```

```
  if (req.path !== '/') title += ': ' + req.path;
  if (helproutes[req.service]) {
    fetch(helproutes[req.service], (help_reply) => {
      res.render('main', {title: title, services: links, viewmodels: help_reply.viewmodels, functions:
help_reply.functions, divcontent:''});
    })
  } else {
    res.render('main', {title: title, services: links, helpObj: [], divcontent:'<p>this is the default help
page</p>'});
  }

  function fetch(helproute, callback) {
    helproute.apiget((reply) => {
      callback(reply);
    })
  }
});

app.listen(port, function () {
  console.log('API HELP app listening on '+port+'!');
});
```

src/internal.js

```
var https = require('https');

var Internal = function(urlObj) {
    var httpsAgent = new https.Agent({
      hostname: urlObj.host,
      port: urlObj.port,
      path: '/',
      method: 'GET',
      rejectUnauthorized: false
    });
    return {
      apiget: function(callback) {
        var request = new https.request({
          hostname: urlObj.host,
          port: urlObj.port,
          path: '/help',
          method: 'GET',
          agent: httpsAgent
        }, (res) => {
          var reply;
          res.on('data', (d) => {
            reply = d;
          });
          res.on('end', () => {
            var jsoned = reply.toString();
            try {
              jsoned = JSON.parse(reply);
            } catch (e) {
            } finally {
```

```
          callback(jsoned);
        }
      });
    });
    request.end();
  }
 }
};

module.exports = Internal;
```

HTML rendering
src/pug/main.pug

```
doctype html
html
  head
    title=title
    meta(charset='UTF-8')
    link(rel="stylesheet", type="text/css", href="/css")
  body
    div(id="main")
      div(id="header")
        h1=title
        div(id="links")
          each service in services
            div(class="help button")
              each link in service
                a(href=link.href)=link.title
      div(id="content-margin-wrapper")
        div(id="component-help-content")
          p=site
          div(id="help-content")
            div(id="fetched-content")
              h1="Microservice viewmodels"
              if (!viewmodels)
                - viewmodels = []
                p(class="empty")="Nothing to show"
              each model, count in viewmodels
                if model.viewmodelname
                  h2=model.viewmodelname
                else
                  h2='viewmodel ' + (count + 1)
                div(class="boxed model")
                  each value, key in model
                    div(class="padded")
                      p(class="inlined key")=key
                      p(class="inlined value")=value
              h1="Microservice functions"
              if (!functions)
                - functions = []
                p(class="empty")="Nothing to show"
              each functionelem in functions
```

```
        div(class="boxed function toplevel")
          each value, key in functionelem
            div(class="function padded")
              p(class="inlined key")=key
              if typeof(value) == 'object'
                div(class="indented object")
                  each sv, sk in value
                    p(class="inlined key")=sk
                    if typeof(sv) == 'object'
                      each ssv, ssk in sv
                        div(class="indented boxed pair")
                          p(class="inlined key")=ssk
                          p(class="inlined value")=ssv
                    else
                      p(class="inlined value")=sv
              else
                p(class="inlined value")=value

      div(id="help-default") !{divcontent}
```

Decks microservice

src/app.js

```
var express = require('express');
var app = express();
var config = require('./config.js');
var port = process.env.PORT || config.LOCALHOSTPORT;
var error = require('./error.js');
var http = require('http');
var https = require('https');
var fs = require('fs');
app.use(require('body-parser').json());

var util = require('util');
var ConnHandler = require('./databaseconnection/connectionhandler.js');

app.get('/help', function(req, res) {
  res.json({
    viewmodels: [
      {
        Name: 'Deck name',
        id: 'Database ID of deck',
        ParentProjectId: 'Object id of parent deck as string or null if first level deck',
      }
    ],
    functions: [
      {
        name: 'Create',
        route: '/',
        method: 'post',
        required_input: {
          headers: {
            'x-auth-token': 'user id'
```

```
    },
    request_body: {
      'name': 'name of deck'
    }
  },
  optional_input: {
    request_body: {
      'parentId': 'deck identifier as string for subdeck creation'
    }
  }
},
{
  name: 'Read',
  route: '/',
  method: 'get',
  required_input: {
    headers: {
      'x-auth-token': 'user id'
    },
  },
  optional_input: {
    query: {
      'id': 'id of deck for details',
      'group': 'id of group for first level deck fetching'
    }
  }
},
{
  name: 'Update',
  route: '/',
  method: 'put',
  required_input: {
    headers: {
      'x-auth-token': 'user id'
    },
    request_body: {
      id: 'id of deck to update',
      name: 'name to update deck with'
    }
  }
},
{
  name: 'Delete',
  route: '/',
  method: 'delete',
  required_input: {
    headers: {
      'x-auth-token': 'user id'
    },
    query: {
      id: 'id of deck to remove',
    }
```

```
      }
    },
    {
      name: 'Subdecks',
      route: '/subprojects',
      method: 'get',
      required_input: {
        headers: {
          'x-auth-token': 'user id'
        },
        query: {
          ppid: 'deck id of parent deck'
        }
      }
    }
  ]
 });
});

app.all('/*', (req, res, next) => {
 req.usertoken = req.headers['x-auth-token'];
 if (!req.usertoken) new error('x-auth-token missing..', [req]).send(res);
 next();
});

app.post('/*', function createDeck(req, res, next) {
 var groupid = req.query['group'];
 if (!groupid) new error('group id missing..', []).send(res);
 ConnHandler.create(req.usertoken, groupid, req.body, (err, dbres) => {
  if (!err) res.json(dbres);
  else {
   new error('something went wrong in deck creation', [err]).send(res);
  }
 });
 // next();
});

app.put('/*', function UpdateDeck(req, res, next) {
 ConnHandler.update(req.usertoken, req.body, (err, dbres) => {
  if (!err) res.json(dbres);
  else {
   new error('something went wrong in deck update', [err]).send(res);
  }
 });
});

app.delete('/*', function DeleteDeck(req, res, next) {
 console.log('hello from DELETE');
 var deckid = req.query['id'];
 if(!deckid) new error('missing deck-id query for DELETE').send(res);
 ConnHandler.delete(req.usertoken, {id: deckid}, (err, dbres) => {
  if (!err) res.json(dbres);
```

```
    else {
      new error('something went wrong in deck deletion', [err]).send(res);
    }
  })
});

app.get('/subprojects*', function subdecksById(req, res, next) {
  var parentId = req.query['ppid'];
  ConnHandler.get_subdecks_of(req.usertoken, parentId, (err, results) => {
    res.json({pid: parentId, subs: results});
  });
});

app.get('/totalQuestionAmounts', (req, res, next) => {
  res.json([]);
});

app.get('/breadcrumb', (req, res, next) => {
  res.json([]);
})

app.get('/*', function deckById(req, res, next) {
  if(!req.query['id']) next();
  else {
    var deckid = req.query['id'];
    ConnHandler.read(req.usertoken, deckid, (err, data) => {
      if (!err) res.json(data);
      else {
        new error('something went wrong', [err]).send(res);
      }
    })
  }
})

app.get('/', function decksFromGroup(req, res) {
  if (!req.query['group']){
     new error('group id missing from query', [req.query, req.path]).send(res);
     return;
  }
  var usertoken = req.usertoken;
  var groupid = req.query['group'];
  if (!usertoken || !groupid) {
    new error('usertoken or groupid invalid', [usertoken, groupid]).send(res);
    return;
  }
  ConnHandler.get_root_decks(usertoken, groupid, (err, results) => {
    if (err) {
      new error('get root decks failed', [err]).send(res);
      return;
    }
    try {
      res.json(results);
```

```
    } catch (e) {
      console.error(e);
    } finally {
    }
  });
});

app.all('/*', function notImplementedYet(req,res) {
  res.status(400).send(JSON.stringify(new error('not implemented yet', [req.path])));
});

var certification = {
  key: fs.readFileSync(__dirname + '/cert/key.pem'),
  cert: fs.readFileSync(__dirname + '/cert/cert.pem'),
}

https.createServer(certification, app).listen(port, () => {
  console.log('https server running at ', port);
});
if (port == config.LOCALHOSTPORT) http.createServer(app).listen(thisport = config.LOCALHOSTPORT - 20, () => {
  console.log('http server runnign at ', thisport);
});
```

Database connection
src/connectionhandler.js

```
var mongoose = require('mongoose'),
    Query = mongoose.Query;
var urlConfig = require('../config.js');

mongoose.connect('mongodb://localhost/'+ urlConfig.database_name)

var DeckModel = require('./models/deck.js');

var Authentication = require('./authentication.js');
var error = require('../error.js');
var fs = require('fs');
var DeckViewModel = require('./models/deck_viewmodel.js');

function user_has_access(err, group, callback) {
  if (err || !group) {
    var errorObj;
    if (err) errorObj = new error('error in authentication', [err, group]);
    else if (!group) errorObj = new error('no authorization', [err,group]);
    callback(errorObj);
    return;
  }
  console.log('authentication successful...');
  callback(null, group);
};

function findDeck(deck, callback) {
```

```
  if (deck.id) deck = deck.id;
  DeckModel.findById(deck, (err, found) => {
    if (err) callback(new error('no authorization', [err]));
    else {
      callback(null, found);
    }
  });
}

function sendDeck(deck, callback) {
  var dvm = DeckViewModel.fromDeck(deck).toJSON();
  dvm.Project = deck.toObject();
  callback(null, dvm);
}

function checkAccess(err, accessObject, callback) {
  if (err) {
    callback(new error('no authorization', [err]));
    return;
  }
  var deck = accessObject.deck;
  var user = accessObject.user;
  if (!deck || !user) {
    callback(new error('problem using accessObject', [accessObject]));
    return;
  }
  // assume deck has only one groupid
  // TODO || DEPRECATED: handle multiple deck => group ids.
  Authentication.user_has_access(user, deck.get('GroupIds')[0], (err, group) => {
    if (err) {
      callback(new error('no authorization', [err]));
      return;
    }
    callback(null, {
      user: user,
      deck: deck,
      group: group
    });
  })
}

var ConnectionHandler = {
  //todo create
  create: function (user, group, deck, callback) {
    if (deck.parentId) {
      findDeck(deck.parentId, (err, found) => {
        checkAccess(err, {
          deck: found,
          user: user
        }, success);
      });
    } else {
```

```
    Authentication.user_has_access(user, group, (err, group) => {
      success(err,{
        group: group
      });
    });
  }

  function success(err, replyObject){
    if (err) {
      callback(new error('error in creating new deck', [err]));
      return;
    }
    var group = replyObject.group;
    DeckModel.create({
      Name: deck.name,
      ParentProjectId: deck.parentId,
      GroupIds: [group.id],
      isRemoved: false
    }, (err, saved) => {
      if(!err) sendDeck(saved, callback);
      else {
        callback(new error('error in creating new deck', [err]));
      }
    });
  }
},
read: function (user, deck, callback) {
  findDeck(deck, (err, found) => {
    checkAccess(err, {
      deck: found,
      user: user
    }, success);
  });

  function success(err, replyObject) {
    if(err) {
      callback(err);
      return;
    }
    sendDeck(replyObject.deck, callback);
  };

},
update: function (user, deck, callback) {
  findDeck(deck, (err, found) => {
    checkAccess(err, {
      deck: found,
      user: user
    }, success);
  });
  function success(err, replyObject){
    if (err) {
```

```
      callback(new error('default error in deck update', [err]));
      return;
    }
    DeckModel.findByIdAndUpdate(deck.id, {
      Name: deck.name
    }, () => {
      callback(null, "ok")
    });
  }
},
delete: function (user, deck, callback) {
  findDeck(deck, (err, found) => {
    checkAccess(err, {
      deck: found,
      user: user
    }, success);
  });
  function success(err, replyObject) {
    if (err) {
      callback(new error('no authorization', [err]));
      return
    }
    DeckModel.findByIdAndUpdate(deck.id, {
      isRemoved: true
    }, () => {
      callback(null, "ok");
    });
  }
},

get_subdecks_of: function(user, deck, callback) {
  findDeck(deck, (err, found) => {
    checkAccess(err, {
      deck: found,
      user: user,
    }, success);
  });

  function success(err, replyObject) {
    if (err) {
      callback(new error('subdeck fetch failed', [err]));
      return;
    }
    var found = replyObject.deck;
    var query = DeckModel.find({ParentProjectId: found.get('_id'), isRemoved: false});
    query.find(returnDVM);
  }

  function returnDVM(err, results) {
    if (err) {
      callback(new error('error in database connection', [err]));
      return;
```

```
      }
      var returning = [],
          docs = results;
      for (var index = 0; index < docs.length; index++) {
        var document = docs[index];
        var dvm = DeckViewModel.fromDeck(docs[index]);
        dvm = dvm.toJSON();
        returning.push(dvm);
      };
      callback(null, returning);
    }
  },

  get_parents_of: function(user, deck, callback) {
    //placeholder
  },

  get_question_amounts_of: function(user, deck, callback) {
    //placeholder
  },

  get_root_decks: function (user, groupid, callback) {
    //check rights,
    Authentication.user_has_access(user, groupid, (err, group) => {
      user_has_access(err, group, run_fetch);
    });
    //fetch all matching not deleted
    function run_fetch(error, group) {
      if (error) callback(error);
              var query = DeckModel.find({ParentProjectId: null, isRemoved:
false}).where('GroupIds').in([group.id]);
      query.find(send_results_back);
    }

    //callback fetched
    function send_results_back(err, docs) {
      if (err) {
        callback(new error('error in database connection', [err]));
        return;
      }
      var returning = [];
      for (var index = 0; index < docs.length; index++) {
        var document = docs[index];
        var dvm = DeckViewModel.fromDeck(docs[index]);
        dvm = dvm.toJSON();
        returning.push(dvm);
      };
      callback(null, returning);
    }
  },
};
module.exports = ConnectionHandler;
```

src/databaseconnection/authentication.js

```
var https = require('https');
var config = require('../config.js');
var apiAddress = config.apiAddress;
var url = require('url');
var Authentication = {
  agent: new https.Agent(config.agentOptions),
  user_has_access: function (user, groupid, callback) {
    var headers = {
      'X-Auth-Token': user
    };
    var target_url = url.parse(apiAddress);
    var options = {
      host: target_url.hostname,
      port: target_url.port,
      method: 'GET',
      path: '/get_group/'+groupid,
      agent: this.agent,
      headers: headers
    };
    var req = https.request(options, (res) => {
      var data;
      res.on('data', (d) => {
        data = d;
      });
      res.on('end', () => {
        var parsed = JSON.parse(data);
        if (parsed.error) {
          callback(parsed);
        } else {
          callback(null, parsed);
        }
      })
    });
    req.on('error', (d) => {
      callback(d);
    })
    req.end();
  },
};

module.exports = Authentication;
```

Database models

src/databaseconnection/models/deck.js

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;
```

```
var Deck_Schema = new Schema({
  Name: String,
  ParentProjectId: String,
  GroupIds: [String],
  isRemoved: Boolean,
});

module.exports = mongoose.model('Deck', Deck_Schema);
```

src/databaseconnection/models/deck_viewmodel.js

```
var DeckViewModel = function () {
  this.attrlist = {
    'name': 'Name',
    'id': 'id',
    'parentProjectId': 'ParentProjectId'
  };

  for (var variable in this.attrlist) {
    if (this.attrlist.hasOwnProperty(variable)) {
      this[variable] = undefined;
    }
  }
};

DeckViewModel.fromDeck = function(deck) {
  var dvm = new DeckViewModel();
  for (var attribute in dvm.attrlist) {
    if (dvm.attrlist.hasOwnProperty(attribute)) {
      var modelattr = dvm.attrlist[attribute];
      dvm[attribute] = deck.get(modelattr);
    }
  }
  return dvm;
};

DeckViewModel.prototype.toJSON = function () {
  var jsony = {};
  for (var attr in this.attrlist) {
    if (this.attrlist.hasOwnProperty(attr)) {
      jsony[attr] = this[attr];
    }
  }
  return jsony;
};

DeckViewModel.prototype.toString = function() {
  return JSON.stringify(this.toJSON());
};

module.exports = DeckViewModel;
```