Modular Monoliths the way to Standardization

Michail Tsechelidis
tsechelidis.michail@gmail.com
Department of Applied Informatics University of Macedonia
Thessaloniki, Greece

Nikolaos Nikolaidis
nnikolaidis@uom.edu.gr
Department of Applied Informatics University of Macedonia
Thessaloniki, Greece

Theodore Maikantis
teomaik19@gmail.com
Department of Applied Informatics University of Macedonia
Thessaloniki, Greece

Apostolos Ampatzoglou
a.ampatzoglou@uom.edu.gr
Department of Applied Informatics University of Macedonia
Thessaloniki, Greece

Abstract

In the resent years the monolith architecture gains once again a lot of popularity in order to reduce costs and time compared to more complicated architectures. Taking into account the advantages of microservice, and trying to embed some of them to monolith architectures, we come to the creation of modular monoliths. This type of design can be consider quite new, and so there isn't yet a specific architecture design that someone could follow if they wish to use it. In this paper we present an architectural design and an implementation strategy for modular monoliths. To evaluate the usefulness of this architecture, we have conducted a study, validating the design and its implementation. In this study 12 architects from different companies took part, expressing some concerns regarding the feasibility in bigger project but also giving an overall positive feedback for the design.

CCS Concepts: • **Software and its engineering** → *Software design engineering*.

Keywords: modular monolith, services, architecture, evaluation

ACM Reference Format:
Michail Tsechelidis, Nikolaos Nikolaidis, Theodore Maikantis, and Apostolos Ampatzoglou. 2018. Modular Monoliths the way to Standardization. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

When designing software systems, architects and developers have plenty of options to choose from regarding the architecture. Microservice-based systems have become ubiquitous in the last couple of years, sifting from traditional monolithic architectures. Microservices have emerged as an architectural style, in which an application consists of a set of small services that are independently deployable and scalable [6, 7]. This gives a lot of advantages to the developers and its also very often in recent years a lot of monolithic system to migrate to microservice architecture [4, 8]

Taking into account the transition from monolithic to microservice architecture, lately a new type of architecture gained a lot of popularity. By applying a Domain-Driven Design (DDD) on a traditional monolithic architecture, we could have different domains which in turn could become modules. DDD [2] supports the division of a large domain model into several independent bounded contexts, in order to split a large development team into several smaller ones. So in simple terms the Modular Monolith is an architectural style where the source code is structured on the concept of modules, but under one project. The difference between the monolithic, microservice, and modular monolithic architectures is visible in figure 1.

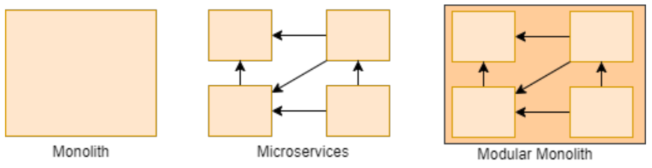

Figure 1. The differences of the architectures

Given the novelty of this architectural design and its limited existing literature, this paper represents an initial step towards exploring its potential applications. We proposed a way that this architecture could be implemented and used, with the help of Spring (but also applicable to other frameworks). We also evaluated the proposed architecture and usage methodology in a case study with 12 architects from several companies.

The rest of this paper is structured as follows: Section 2 gives an overview of the background information and related work. Section 3 presents the proposed approach in terms of architecture and usage in development. In Section 4, an evaluation of the proposed approach is presented. Finally, Section 5 presents the threats to the validity and Section 6 concludes the work and discusses potential limitations and future work.

2 Background Information

2.1 Hexagonal Architecture

Cockburn [1] introduces Hexagonal Architecture, which is also known as Ports and Adapters. This architecture was one of the first that broke up the traditional layering, which was used till that point, in favor of the onion layering. The main idea is that we have the core of the software, which is technology-agnostic, that contains the business logic of the application. Outside of the core we have the ports (protocols or interfaces) that define how the application can be used (by "driving adapters"; port implementations are in the core) and the data that it needs (provided by "driven adapters" implemented externally). This architecture offers a twofold advantage. Firstly, it provides adaptability through the use of adapters and secondly, it offers a high level of domain isolation, as it is not bound to any specific adapter.

2.2 Modular Compiles

In order to be able to have modular compiles, in the sense that a change in one module doesn't require a rebuilt of the entire monolithic software, we used runtime dependencies in combination with a set of patterns [3]. The used patterns are the following:

- *Separated Interfaces* pattern, "Defines an interface in a separate package from its implementation".
- *Layer Supertype* pattern, "A type that acts as the supertype for all types in its layer".
- *Abstract Factory*, "Lets you produce families of related objects without specifying their concrete classes".

3 Proposed Approach

3.1 Proposed Architecture

In our proposed architecture, we primarily use the hexagonal architecture for the "individual" services. Additionally, we have adopted modular compilation for all the proposed modules. These decisions were made for the following reasons:

- The hexagonal architecture was selected due to its design, since it isolates the business logic from the external system. Considering that in our case the domain is a different module, this architecture fits perfectly.
- The modular compiles are used due to its time saving characteristics, since a change in one of the modules does not lead to the rebuild of the whole application, resulting to an overall faster compile time.

The proposed modules and their connections can be seen in Figure 2. This figure provides an example application that includes user accounts and rentable movies. Below you can see what each module is responsible for:

- **main** - initializer scanning in runtime the dependencies/services. It is the executable.
- **domain** - global/core logic that any service can use (interface definitions only - like API).
- **domain_imp** - implementation of domain, and any other class to support the exposed domain to the services.
- **web_account** - inbound and outbound handling for account (service).
- **web_movies** - inbound and outbound handling for movies (service).
- **queries** - global/core queries that any service can use (interface definitions - like API).
- **databasePrimary** - schemas, and queries implemention (used by account).
- **databaseSecondary** - schemas, and queries implemention (used by movie).

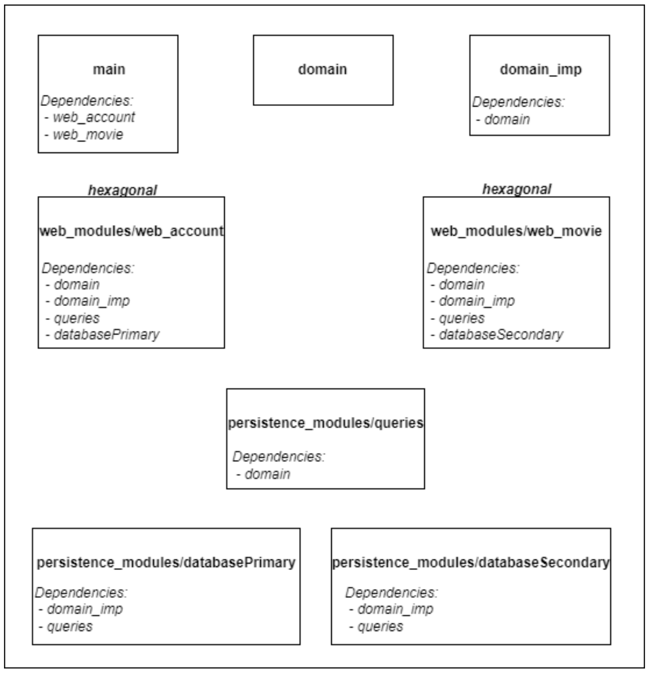

Figure 2. Proposed architecture components

Moreover, it can be noted that given the structure of the proposed architecture, it is possible to create a different module (instead of main) with only a subset of the modules. The dependencies of web_account and web_movies are runtime ones, so given the requirements we could easily create a main with only the web_account dependency. For a more in-depth view, the template code of the proposed architecture can be found online[1].

3.2 Proposed Development

Implementing the proposed architecture can be trivial for big projects, with the biggest reason being the number of developers working on a single repository. In a microservice architecture this is not a problem since each developer accesses only the repository of the service they are working on. So, we tried to create a similar abstraction where the developer accesses the "services", or modules in our case, that they are working on. We propose that each module can be developed in a separate repository branch with a set of GitHub actions, which are responsible for automatically creating the module artifacts and the final artifact/image of the application. In this way for example the developer that works on the database queries will be in the appropriate branch, and in turn have access only in the code of that module. Moreover, this could also be enforced, if needed, in the sense that specific developers could commit in specific branches. In Figure 3 you can see all the modules of the proposed architecture, with each one being developed in a separate branch.

GitHub actions can provide an easy and uniform way for developers to access the latest modules and automate the creation of the final image, but using this implementation should be done in parallel with the best practices in git versioning systems. This means that for each change in a module, a branch should be created and at the end be merged to the original branch of the module. This is vital, since it will keep the work of each developer isolated till the merge, and more related to this implementation, it will execute the GitHub actions only in the module related branches. Figure 3 depicts the development lifecycle of one of the modules, by using a new "development" branch. Finally, the template repository along with the branches of the proposed implementation can be found online[2]

4 Validation

To evaluate the proposed architecture and its usage we have performed an empirical study, designed and reported based on the guidelines of Runeson et al. [5].

4.1 Study Design

To study the proposed architecture and its acceptance, in terms of real word systems we have formulated the following three research questions:

[RQ1] - How is the proposed architecture perceived?

[RQ2] - How is the proposed implementation perceived?

[1]https://github.com/tsechelidisMichail/HexagonalSpring_Modular

[2]https://github.com/tsechelidisMichail/HSMB-v3

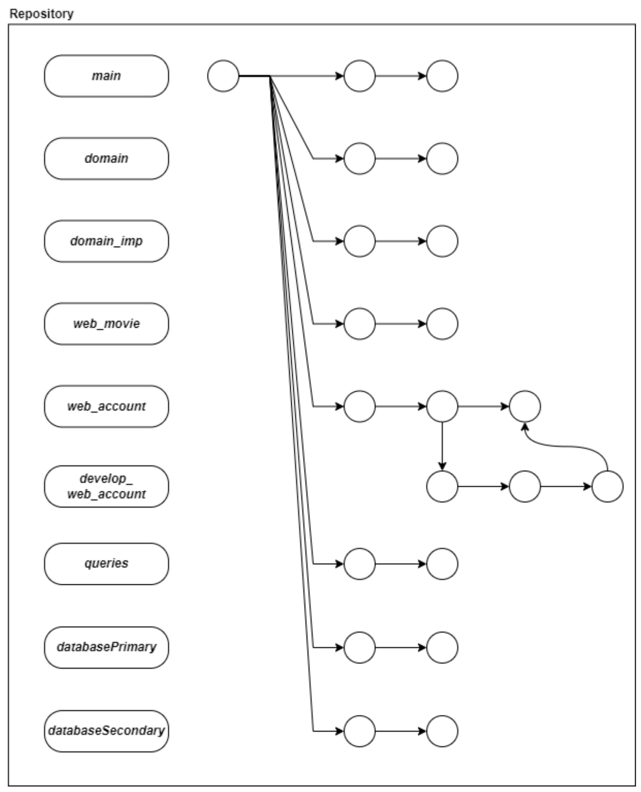

Figure 3. Development in different branches

[RQ3] - What is the architects acceptance?

In order to answer these RQs we circulated a questioner in 12 software architects from 8 different companies. First we presented the proposed architecture and implementation to the architects and asked them to evaluate them, in order to then answer RQ1 and RQ2. Finally, we asked the participants to fill out a set of questions that would provide insight to the acceptance of the overall architecture. For this we reused the system usability scale (SUS), tailored to the needs of software architecture acceptance.

4.2 Results and Discussion

Evaluation of the Proposed Architecture: In relation to the proposed architecture, all participants expressed a positive opinion; however, they also expressed some uncertainty regarding their ability to implement it fully. When queried about their intention to use this architecture 71% of them replied "Maybe" and 28% "Yes". Moreover, compared to other monolithic architectures 85% of the participants believed that this architecture is better. Regarding the comparison with microservice architecture, the feedback was not so positive. The majority of participants expressed a preference for the microservice approach, as evidenced also by their comments. Their primary concern was related to larger-scale projects, as the monolithic architecture may result in a larger codebase compared to the microservice one.

Evaluation of the Proposed Implementation: Regarding the proposed implementation, once again all of the participants liked the idea, but only 28% of them would use it, 42% were reluctant, and the rest 28% wouldn't use it. Having said that, all of them liked the separation of modules in their own branch, and they don't think that there will be a problem with it. The main reason that the majority of them wouldn't use this implementation is once again regarding the concern of big project, since the will start having a lot of branches, and as one stated "it would be very hard to keep up". Another participant stated that "Its very hard to be implemented in an actual big system, which is going to have too many branches".

Acceptance Evaluation: Finally, for the evaluation of acceptance, the results are visible in the following figure. We can see that for the majority of the questions we got very good responses, but there was a problem when it came to whether the participants would use the architecture frequently. Going back to the open questions regarding the architecture, the main reason seams to be with how it would perform in big projects. The majority of them pointed out concerns regarding the size and possible complexity when this solution would be applied in big real world software. But, in the rest of the acceptance questions we can see that the proposed architecture was evaluated with great scores.

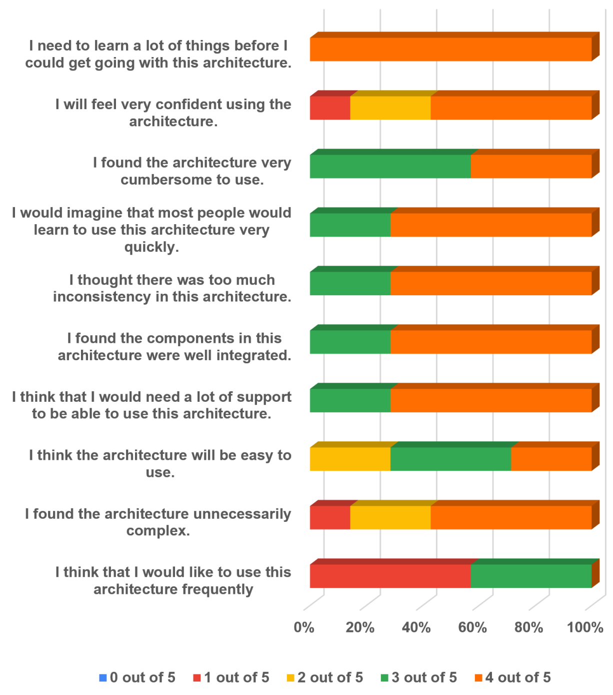

Figure 4. User Acceptance

5 Conclusions

In this paper we proposed an architecture and development methodology for the creation of modular monolithic software. The architecture is based on the hexagonal architecture along with modular compiles for all the associated modules. In this approach, we have isolated all of the modules and made possible the development of each one without the need of rebuilding the entire project. In order to develop this kind of software and provide flexibility for bigger projects, we proposed that each branch of the Git repository can be used as a placeholder to develop one module. Our approach was evaluated in a small study with 12 software architects through the use of a questioner. The participants gave very positive feedback for this design and implementation strategy, but weren't sure about its feasibility in bigger projects. Nevertheless, the key advantage lies in the acceptance of all other aspects of the architecture, which allows for the development of a practical and modular monolithic architecture.

Acknowledgments

This work has been partially funded by the Horizon Europe Framework Programme of the European Union under Grant agreement no 101058479.

References

[1] Alistair Cockburn. 2005. Hexagonal architecture. *alistair. cockburn. us* (2005).
[2] Eric Evans. 2004. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional.
[3] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., USA.
[4] Nuno Gonçalves, Diogo Faustino, António Rito Silva, and Manuel Portela. 2021. Monolith modularization towards microservices: Refactoring and performance trade-offs. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C).* IEEE, 1–8.
[5] Martin Host, Austen Rainer, Per Runeson, and Bjorn Regnell. 2012. *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons.
[6] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. *MartinFowler. com* 25, 14-26 (2014), 12.
[7] Genc Mazlami, Jürgen Cito, and Philipp Leitner. 2017. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS).* IEEE, 524–531.
[8] Sam Newman. 2019. *Monolith to microservices: evolutionary patterns to transform your monolith.* O'Reilly Media.