**Domain-Driven Design in Cloud Computing A .NET and Azure Case Analysis**

Jordan Jordanov1, Pavel Petrov1, Ivan Kuyumdzhiev1, Julian Vasilev1

*1University of Economics - Varna, Varna, Bulgaria*

*Abstract –* This paper explores the integration of Domain-Driven Design (DDD) into the cloud computing components of the Microsoft ecosystem. The study aims to show a proficient methodology for constructing a software architecture that is capable of growing, easy to maintain, and efficient. The research was driven by the necessity to close the gap between theoretical principles of DDD and their practical implementations in cloud-native services. The research utilized a case study methodology to offer an empirical overview. According to the results, DDD can be classified as a vital consideration for the application and data layers of the overall Platform-as-a-Service and Infrastructure-as-a-Service cloud models. The study findings indicate that the utilization of DDD in cloud computing improves scalability, maintainability, and cost-effectiveness. The case study explores the potential of employing Domain-Driven Design (DDD) as a conventional approach to enhance the efficiency of software architecture in cloud environments.

*Keywords –* Domain driven design, cloud computing, case study, software architecture, Azure .NET.

1. **Introduction**

Domain-Driven Design (DDD) has become an important framework in the constantly evolving field of software development, enabling the creation of advanced applications DDD creates a collaborative environment by closely linking software design with the main business domain. This approach encourages technical and domain experts to work together in

DOI: 10.18421/TEMxx-xx

<https://doi.org/10.18421/TEMxx-xx>

**Corresponding author: -** ,

*affiliation*

**Email:** -

*Received: -----.*

*Revised: -----.*

*Accepted: -----.*

*Published: -----.*

Description: Description: Description: Description: Description: Description: Description: by-nc-nd-.png© 2023 ----; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at <https://www.temjournal.com/>

developing software that is flexible and can easily adapt to evolving business requirements. Although this approach shows potential, there is still a notable lack of research in practical studies that examine the relationship between DDD concepts and cloud development frameworks for constructing web, mobile, desktop or IoT applications. This paper aims to investigate the implementation of DDD concepts using .NET and their deployment on the Azure platform. This study aims to offer a comprehensive perspective on the strategic decisions, architectural elements, and results related to these integrations. To accomplish this, the study utilizes a research methodology that involves multiple use cases.

DDD provides a philosophy and a set of guidelines, including bounded contexts, and ubiquitous language. In addition, there are programing models like “Aggregates” and “Value objects” as well as patterns as Command Query Responsibility Segregation (CQRS) and Event Sourcing (ES). These principles are especially applicable to microservices, functional programming, and event driven development. In addition, an integrated test suite is supposed to guarantee the integrity of all of them.

Table 1 provides a fundamental classification of cloud computing, which presents various categories based on different levels of abstraction and customized service provision to meet specific operational needs.

*Table 1. Classification across some of the fundamental cloud services*

|  |  |
| --- | --- |
| Infrastructure as a Service (IaaS) | Platform as a Service (PaaS) |
| Application | Application |
| Data | Data |
| Runtime | Runtime (Managed) |
| Middleware | Middleware (Managed) |
| OS | OS (Managed) |
| Virtualization (Managed) | Virtualization (Managed) |

From the models presented it is clear that Platform as a Service (PaaS) and, to a certain extent, Infrastructure as a Service (IaaS) have become the main focus areas for DDD. PaaS and IaaS offer customers frameworks that create, build, and manage applications. This eliminates the difficulties that come with developing and maintaining the underlying infrastructure. IaaS possesses inherent proficiency in managing elements such as networking, storage, servers, and virtualization. PaaS encompasses operating systems, middleware, and runtime environments, thereby assigning developers the task of managing applications and data. The significance of DDD concepts becomes apparent within these two fields.

The microservices architecture is defined by the process of breaking down applications into small, autonomous services, initiating the establishment of one of the cloud-native standards. Each microservice, which contains a specific business function, can be deployed, scaled, and maintained independently. This allows for the utilization of the natural flexibility and durability of cloud platforms. Microservices facilitate the implementation of continuous integration, continuous delivery, and dynamic resource allocation. As stated by the Cloud Native Computing Foundation (CNCF), microservices allow the creation of system components that are loosely connected, resilient, manageable, and observable. When used in conjunction with strong automation, they enable engineers to make significant and predictable changes frequently, with minimal effort. There are numerous scientific studies that have examined the leading corporations like Netflix and Uber. These companies are supporting online platforms, which are offering a wide range of services. The software responsible for these services frequently releases new versions, deploying thousands of instances on a weekly basis.

The primary objective of microservice architecture is to establish explicit and well-defined boundaries. This include identifying bounded contexts and associated aggregates, and determining the types of commands and queries that end users perform on the system. Bounded context (BC) is a fundamental concept in DDD that acts as a means of separating different components to enhance their ease of management and scalability. In addition, a BC emphasizes the importance of self-reliance by encompassing entities, repositories, factories, and application services. BCs are components of the solution architecture designed to address specific sub-domains that are logically separated. The degree of physical isolation introduces an additional level of intricacy, contingent upon factors such as precise specifications, codebase, and the size of the development team.

There is at least one aggregate present in BC. Aggregates are identified through thorough analysis sessions, typically leading to the recognition of different entities and value types that naturally form groups under the control of a main entity. When this kind of grouping happens, it signifies the demarcation of a collective, formed exclusively by business regulations. An aggregate function as a domain model by grouping multiple entities together under a single conceptual framework.

In order to design an approach for constructing aggregates and other DDD models, this study examines the practical aspects of using functional programming (FP). FP primarily focuses on two distinct features: maintaining the integrity of method signatures and ensuring referential transparency. The concept of method signature honesty ensures that a function's signature accurately and comprehensively represents all possible input and output values. Referential transparency guarantees that a function's output remains consistent for a given input, without any additional side effects. Furthermore, FP is supposed to the reduces code complexity, making it easier to understand and analyze logically. It also considered to simplifies unit testing and enhances the modularity and composability of software components.

The importance of immutability in FP is crucial, as mutable operations have the potential to introduce “dishonesty” into the code. The absence of clearness hampers our capacity to participate in rational reasoning, making the process of debugging more complex and creating barriers to multi-threading. Moreover, the utilization of FP is improved by the implementation of CQRS and the integration of fundamental domain logic. Railway-oriented programming, influenced by Scott Wlaschin, offers a more efficient method of structuring processes in contrast to conventional methodologies that involve lengthy and complex code blocks containing numerous "if/else" and "try/catch" statements. The functional approach employed in this context utilizes extension methods to enhance legibility by reducing redundant code and emphasizing the main logical sequence.

In this context, it is important to analyze the logic of the code in real time by putting the system under test (SUT). Unit testing for codebases of this nature primarily entails supplying input to functions and verifying the outcomes. To support these needs test doubles, particularly mocks, can be utilized to replace dependencies with unpredictable behavior, thus achieving the desired outcome. Unit testing offers a key benefit of ensuring the integrity of existing functionality while allowing for efficient modifications to code.

Based on a case study from the Computer Science department at North Carolina State University, unit testing is considered a crucial safeguarding measure. Within this framework, a key performance indicator (KPI) is code coverage, also known as test coverage. This metric quantifies the extent to which the source code of a program is tested by a particular test suite. Code coverage is expressed as the ratio of the number of lines of code covered by tests to the overall number of lines in the codebase, visually represented as:

***Code coverage = Lines of code covered / Overall number of lines***

This ratio provides a numerical value that reflects the thoroughness of testing and helps identify untested parts of the code. High code coverage is often associated with higher software quality, as it indicates that a significant portion of the code has been executed during testing, potentially uncovering defects and ensuring that the software behaves as expected under various conditions. However, achieving 100% code coverage does not guarantee the absence of bugs, as it does not account for the quality or comprehensiveness of the tests themselves. Nonetheless, striving for higher code coverage can contribute to more robust and maintainable code by encouraging comprehensive testing practices. In summary, code coverage serves as a useful benchmark for evaluating the effectiveness of test suites and guiding the development process toward better software reliability and performance.

1. **Methodology**

The aim of this study is to explore and give in-depth understanding of the software development with DDD, CQRS and ES patterns via .NET and Azure technologies. In order to reach this goal, the selection of an appropriate research approach is an important step. With regard to the uncertainty and a lag of research for the implementation of the DDD concepts, this goal of this study is to fill this gap and show strong and reliable development processes. To approach this goal, case study research was deemed as an appropriate research method. Case studies, representing qualitative research methods, are commonly used within the computer and social science. According to XXZ, the case study design may be chosen when the selected case represents a critical case in testing a well-formulated theory with clearly defined propositions, which is going to be shown in the 3rd sub-section of this chapter. The nature of the current case study is confirmative (explanative). The purpose is testing the DDD theories that have been deducted from preview’s research.

2.1. Tools & Technologies

The .NET is widely acknowledged as a prominent option for developing scalable and robust corporate applications. Based on statistics provided by Techempower [], it has been observed that ASP.NET exhibits superior efficiency and performance compared to several alternative web application platforms and full-stack frameworks, given in Table 2.

*Table 2. Comparison of server technologies*

|  |  |  |
| --- | --- | --- |
| Technology | Programming language | Processed requests per second |
| ASP .NET Core | C# / .NET | ~300 613 |
| NodeJS | Javascript / C++ | ~200 123 |
| Gin | Go | ~150 230 |
| Symphony | PHP | ~100 234 |
| Spring | Java | ~80 954 |

Microsoft has outlined a strategic plan [] for the future development and maintenance of .NET, guaranteeing regular upgrades and expanded library support until the year 2026. The framework of .NET is highly regarded due to its ability to seamlessly integrate with many programming languages, such as C#, F#, and VB, all of which have prominent positions on the Tiobe index []. According to data conducted by Statista [], C# has emerged as a prominent programming language used by developers for microservices. One of the factors contributing to this is the lightweight Minimal API [], which is a framework component specifically designed for microservices. Additional factors include the use supplemental libraries such as EntityFramework, MediatR, Optional, Marten, SignalR, AutoMapper, Serilog, Stylecop, Swagger, FluentValidation, xUnit, Autofixture, Moq and Shouldly. This interoperability further enhances the esteemed status of .NET.

how your research was conducted as well as to enhance credibility of your research. In case your research is quantitative, methodology should present the way numerical data was collected and how mathematical analyses are conducted to observe, analyse, access, and test experiments and hypotheses. Qualitative research involves collection and analysis of non-numerical data (e.g.: text, video, or audio) with the aim of explaining concepts, opinions, perspectives, or personal experiences.

Times New Roman 11 – point font should be used for normal text with “single” line spacing. The manuscript has to be prepared in two columns separated by 5 mm (0.2”). The text should be aligned to both the left and right (justified). The margins for A4 (210×297 mm2) paper are given in Table 1.

*Table 1. Page layout description*

|  |  |  |
| --- | --- | --- |
| # | System case | Description |
| Top margin | 20 mm | (0.79") |
| Bottom margin | 20 mm | (0.79") |
| Left margin | 20 mm | (0.79") |
| Right margin | 18 mm | (0.71") |
| Column Spacing | 5 mm | (0.2") |

1. **Results**

The results section describes the obtained findings gathered from your research. Provide appropriate figures and tables to effectively illustrate your results. Figures are used to present data trends or other visual information while tables are particularly useful when the exact values are important.

Addressing the research question will not only provide valuable insights to the academic discussion but also establish clear programming principles. In this context, the results of this study will provide valuable direction for software developers and architects in designing and implementing suitable data structures and algorithms.

1. **Discussion**

Discussion section should explain what the collected results mean and what is their importance and contribution to the field.

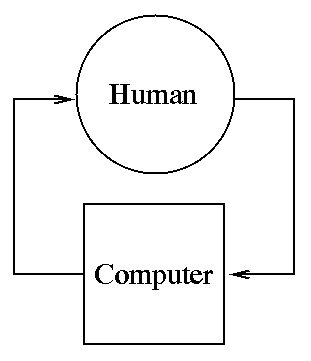
Times New Roman 11 – point font should be used for normal text with “single” line spacing. The manuscript has to be prepared in two columns separated by 5 mm (0.2”). The text should be aligned to both the left and right (justified). The margins for A4 (210×297 mm2) paper are given in Table 1.

*Table 1. Page layout description*

|  |  |  |
| --- | --- | --- |
| Paper size | A4 |  |
| Top margin | 20 mm | (0.79") |
| Bottom margin | 20 mm | (0.79") |
| Left margin | 20 mm | (0.79") |
| Right margin | 18 mm | (0.71") |
| Column Spacing | 5 mm | (0.2") |

A regular paper may consist of multiple sections. Title of each section (except references and acknowledgements sections) should be formatted with 11- point font in bold. Spacing between the title and the text should be 11 – point. The use of options “add space before/after paragraph” is not allowed.

Titles (subtitles) of sections can further be divided in subsections, which should use a 10 – point font in italicandbold, following title case capitalization. Figures should be one column wide, centered, and in line with the text (select the table, right click, chose options “wrap text” and then “in line with text”). If it is impossible to place the figure in one column, two column width figures are permissible. It is essential to ensure that all figures maintain a high quality with easily readable labels. Each figure must include a caption placed beneath it while table captions are placed above it. Tables/figures should be inserted in the text after they have been mentioned. When creating captions for figures and tables, use a 10-point italic font without punctuation at the end (e.g., Figure 1).



*Figure 1. Caption of the figure*

All sections within the paper should be properly enumerated. All enumerated titles and the regular text should be in line (on the left side of the paper) such as:

1. Introduction

Indent the first line of each paragraph of text with 0,5 – point from the left margin. For the numeration of subsection titles, use a multilevel list style as illustrated below:

1. Introduction

1.1. Writing Instructions

1.1.1. Subsection: Writing Introductions

Each section should contain a brief introduction. When introducing abbreviations for the first time, provide full form of it, for example: Artificial Intelligence (AI).

1. **Conclusion**

Be brief and state the most important conclusions from your paper as well as further implications for the field. Discuss benefits or shortcomings of your work and suggest future areas for research. Do not use equations, figures, or references here.

Acknowledgements (If any)

These and the Reference headings are in bold but have no numbers. Titles and text of sections references and acknowledgements should be formatted with 10 – point font, yet text in acknowledgements section should be in italic font. Include the names of the funding agencies or organizations, grant numbers - number of project, and any relevant details about the funding.

**References**

References section is not enumerated. The use of hyperlinks should be avoided as much as possible. When including in-text references, they should be on the same level as the rest of the text e.g.: “References give proper credit to all work included [1]”. For in-text references TEM Journal prefers IEEE style (reference numbers in square brackets “[2], [3]”) in combination with APA style (the use of authors’ names with reference numbers, yet it is better to use only IEEE format as much as is possible). When citing the author's name, use the following format: 'as shown by Brown [4].' In cases where only the reference itself is included, use this format: 'References are of great importance in scientific papers [2]’.When including multiple sources, format them as follows: [1], [2], [3].

When introducing references in the reference list use 10 point font, following the form provided below. Examples of the form:

1. Wong, B., & Kokko, H. (2005). Is science as global as we think?. *Trends in ecology & evolution*, *20*(9), 475-476.
2. Hennessy, J. L., & Patterson, D. A. (2012). *Computer architecture: a quantitative approach*. Elsevier.
3. Miller, T. (2019). *Enhancing readiness: An exploration of the New Zealand Qualified Firefighter Programme* [Master's thesis, Auckland University of Technology]. Tuwhera.
4. Herculano-Houzel, S., Collins, C. E., Wong, P., Kaas, J. H., & Lent, R. (2008). The basic nonuniformity of the cerebral cortex. *Proceedings of the National Academy of Sciences of the United States of America, 105*(34), 12593-12598.

<https://doi.org/10.1073/pnas.0805417105>

1. FESTO. (2019). Fluidic Muscle DMSP/MAS. Retrieved from:

<https://www.festo.com/rep/en_corp/assets/pdf/info_501_en.pdf> [accessed: 19 September 2022].