Implementation of Command Query Responsibility Segregation (CQRS) in the Competency Test Information System

I Gusti Ngurah Bagus Caturbawa, Sri Andriati Asri, I Wayan Suasnawa,
Anak Agung Ngurah Gde Sapteka, Ni Gusti Ayu Putu Harry Saptarini and Kadek Amerta Yasa
Department of Electrical Engineering, Politeknik Negeri Bali, Kampus Bukit Jimbaran, Badung, Indonesia

Keywords: Certification, CQRS, Pattern System, System Architecture.

Abstract: Lembaga Sertifikasi Profesi (LSP) is an institution that carries out professional certification activities in Indonesia. The process of implementing the competency test at LSP includes the registration process, pre-assessment, assessment, up to the issuance of certificates. The purpose of this research is to build a competency test information system for LSP. This competency test information system was built by applying a pattern architecture that divides the processes of writing and reading data in different processes. This architecture is known as the Command Query Responsibility Segregation (CQRS). CQRS separates writes and reads into different models, using commands to update data, and queries to read data. The write model may treat a set of associated objects as a single unit for data changes and ensure that these objects are always in a consistent state. The read model just returns a DTO for use in a view. Implementing CQRS in an application prevents updating commands from causing merge conflicts at the domain level. A CQRS pattern system development can avoid complex joins when creating queries.

1 INTRODUCTION

Professional certification in various fields is one of the main programs of the Indonesian government in improving workforce competence. This is necessary because of the need to standardize services between Asean countries. The target for professional certification participants and the funds provided by the government has increased every year. The aim of the government is to trigger professional certification institutions that are starting to grow to cultivate professional certification in the industrial world.

Lembaga Sertifikasi Profesi (LSP) is an agency for implementing professional certification activities that has obtained a license from the Badan Nasional Sertifikasi Profesi (BNSP). Licenses are granted through an accreditation process by BNSP which states that the relevant LSP has met the requirements to carry out professional certification activities. The process of granting competency certificates is carried out systematically and objectively through a competency test that refers to the Indonesian national work competency standards, international standards and/or special standards. Competency testing is a procedure that is part of the assessment to measure the competence of certification participants using one or several methods such as written, oral, practical, and observational, as stipulated in the certification scheme (BNSP, 2017). The activities of a professional certification body determine that a person meets the requirements for certification, which includes registration, assessment, certification decisions, maintenance of certification, recertification, and use of certificates. According to the procedure, it will go through a fairly long process and will require the management of a lot of documents for the entire competency test process. If the implementation of this competency test is carried out with a large number of competency test participants (assessments), the LSP needs to create a computerized system to streamline the competency test process. This is due to the large number of documents that must be prepared by both the LSP manager and the assessor.

Therefore, this study will raise the topic of designing a Competency Test Information System in a professional certification institution. More specifically in this article, we will discuss the use of the Command Query Responsibility Segregation (CQRS) pattern architecture. Command Query Responsibility Segregation (CQRS) is a development

Caturbawa, I., Asri, S., Suasnawa, I., Sapteka, A., Saptarini, N. and Yasa, K.
Implementation of Command Query Responsibility Segregation (CQRS) in the Competency Test Information System.
DOI: 10.5220/0010964500003260
In *Proceedings of the 4th International Conference on Applied Science and Technology on Engineering Science (iCAST-ES 2021)*, pages 1322-1326
ISBN: 978-989-758-615-6; ISSN: 2975-8246

of the Command Query Separation (CQS) technique introduced by Bertrand Meyer which can be applied to the development of distributed software systems (Meyer, 1997).

2 THEORY

System development always changes with the needs of application developers related to the Relational Database Management System (RDBMS). Dominguez & Melnik suggested the need for a method to organize the system in a different way when interacting with the database (Domínguez, et al., 2012). In 2013 CQRS has been widely used in various systems developed for financial institutions and health institutions (Betts, et al., 2012).

Information systems generally use a client-server architecture (Niltoft, et al., 2013). Figure 1. illustrates a commonly used client-server architecture, the MVC Architecture/Pattern. Model-View-Controller (MVC) is a method for separating data (Model) from the view (View) and how to process it (Controller). In its implementation, most frameworks are based on the MVC architecture. MVC separates application development based on its components, namely the main components that build an application such as the part that is the control, the data manipulation section, and the user interface. In general, a system is usually composed of a user interface, business logic and a database-linked model. In the model, the services provided are creating, reading, updating and deleting records. This is not the only way to design and design systems, but is a very commonly used system.

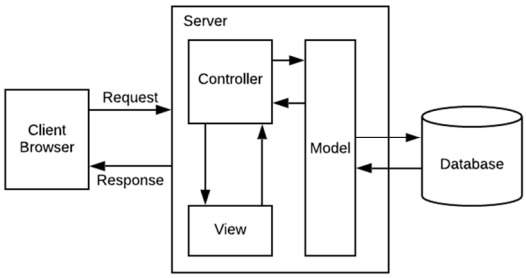

Figure 1: Client server architecture.

The concept of using CQRS will note the difference between reading a record and writing a record. This requires one class for inserting, updating or deleting notes and one class for reading notes. This class is implemented in the form of two models, namely the query model and the command model. This reduces the complexity because it differentiates command logic and query logic (Erb, et al., 2014) as shown in Figure 2. This means that there is a separation of responsibilities between the two classes (Fitzgerald, 2012).

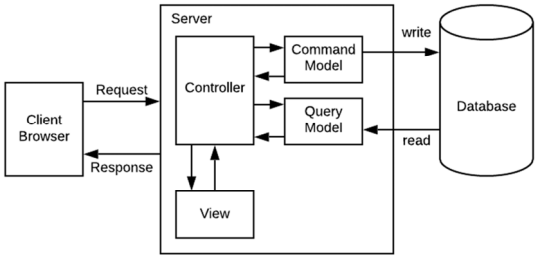

Figure 2: A model of CQRS with one database.

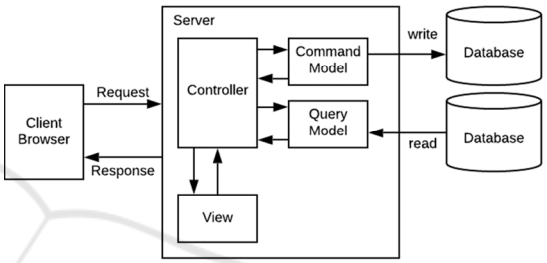

Figure 3: A model of CQRS with separated databases.

In its development, this separation of responsibilities can also be done in the database (Erb, et al., 2014). This can be seen in Figure 3. This requires 2 databases with the same records. This will reduce the burden compared to if the process only uses one database. However, this separation will require a very reliable synchronization. In designing this competency test system, the implementation of the CQRS used is one database, which is separated only from the query model and command model.

3 RESEARCH METHODOLOGY

This research uses a case study of competency test software. The software built is used by the Professional Certification Institute (LSP). In implementing this system, the first thing to do is collect data about the concept and application of CQRS. After that we carried out an analysis to design how the CQRS concept was applied.

At the next stage, the development team involved were invited to discuss how to implement the CQRS concept into the system being built. System analyst creates designs which are then translated into program code.

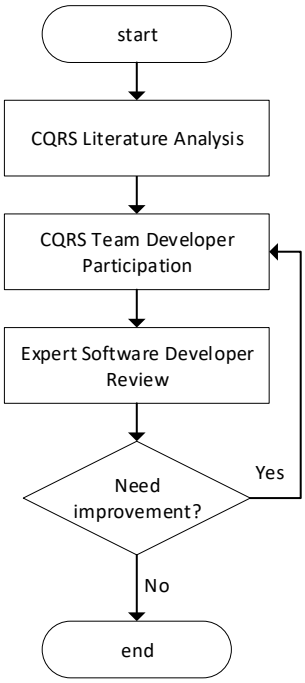

Figure 4: Research Methodology.

In the final stage, the implementation results are reviewed by experienced software developers. If there is any program code that needs improvement, it will be discussed again by the development team and improvements will be made according to the suggestions given. If it is appropriate, the system development cycle using the CQRS concept is complete.

4 RESULTS AND DISCUSSION

In implementing CQRS in the domain, the structures can be differentiated directly into separate folders, namely the command folder and the query folder.

Figure 5: The structure of the model in the domain.

Figure 5 illustrates the structure of the system being built. Furthermore, in Figure 6 shows model in the form of classes that have write functions. This write function takes the form of the process of adding, changing, and deleting notes. Models related to the read function are specifically placed in the query folder as shown in Figure 7.

Figure 6: Write models.

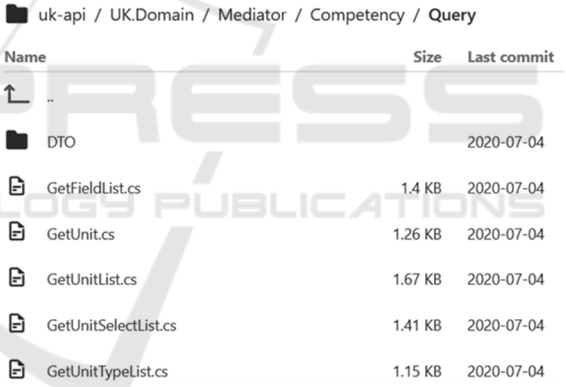

Figure 7: Read models.

In the creation of a class for the read function, it will have two parts. The main part is a query class that will be processed further in the Handler class. The output of this query class is in the form of base results and result values. When a query is executed successfully, the resulting status is successful. If this fails, a message is required to provide information via the user interface. The class for this read function is shown in Figure 8.

Next, on the class for the write function. As stated in the previous section, this writing function includes creating, updating, and deleting records. Basically the writing process can be implemented in one model class. One model class is possible if the level of code complexity is not too large. In conditions where the

system being built has a large level of complexity, each of these functions can be made separately. This will make it easier for programmers to develop and maintain the system being built. The three processes that are included in this write function are made separately as shown in Figure 9, Figure 10, and Figure 11.

```
namespace UK.Domain.Mediator.CompetencyAssessment.Query
{
    public class GetCompetencyAssessmentList
    {
        public class Query : IOperationRequest<List<CompetencyAssessmentListResponse>>
        {
            public int Status { get; set; }
            public Query(int status)
            {
                Status = status;
            }
        }

        public class Handler : IOperationHandler<Query, List<
        CompetencyAssessmentListResponse>>
        {
            private readonly DatabaseContext _dataContext;
            private readonly UserManager<UserIdentity> _userManager;
            private readonly ItemsCache _itemsCache;

            public Handler(DatabaseContext dataContext, UserManager<UserIdentity>
            userManager, ItemsCache itemsCache)
            {
                _dataContext = dataContext;
                _userManager = userManager;
                _itemsCache = itemsCache;
            }

            public async Task<OperationResult<List<CompetencyAssessmentListResponse>>>
            Handle(Query request, CancellationToken cancellationToken)
            {
                var user = await _userManager.FindByIdAsync(_itemsCache.UserId);
                var status = (AssessmentStatusEnum)request.Status;
                var result = new OperationResult<List<
                CompetencyAssessmentListResponse>>();

                .
                .
                .

            }
        }
    }
}
```

Figure 8: Implementation of class for the reading model.

```
namespace UK.Domain.Mediator.CompetencyAssessment.Command
{
    public class AddAssessment
    {
        public class Command : IOperationRequest
        {
            public AddAssessmentRequest Dto { get; set; }
            public Command(AddAssessmentRequest dto)
            {
                Dto = dto;
            }
        }

        public class Handler : IOperationHandler<Command>
        {
            private readonly DatabaseContext _dataContext;
            private readonly ItemsCache _itemsCache;

            public Handler(DatabaseContext dataContext, ItemsCache itemsCache)
            {
                _dataContext = dataContext;
                _itemsCache = itemsCache;
            }

            public async Task<OperationResult> Handle(Command request, CancellationToken
            cancellationToken)
            {
                var result = new OperationResult();

                var assesseeCompetency = new Assessment
                {
                    Status = AssessmentStatusEnum.PendaftaranTerkirim,
                    AssesseeId = _itemsCache.UserId,
                    ScheduleId = request.Dto.ScheduleId,
                    SchemaId = request.Dto.SchemaId,
                    IsDeleted = false
                };

                request.Dto.Documents.ForEach(file =>
                {
                    assesseeCompetency.Files.Add(new AssessmentFile { AssessmentDocumentId
                    = file.DocumentId, FileUrl = file.FileUrl, FileName = file.FileName });
                });

                await _dataContext.Assessment.AddAsync(assesseeCompetency);

                return result.Success();
            }
        }
    }
}
```

Figure 9: Implementation of class for the write model.

Unlike the model class in the read function, the model class in this write function has one output, namely the base result, which indicates whether the write process has been successfully carried out or not. If not successful it will give an error message on the system user interface. The output of this write function does not require an output in the form of a result value.

```
namespace UK.Domain.Mediator.CompetencyAssessment.Command
{
    public class UpdateSchedule
    {
        public class Command : IOperationRequest
        {
            public int Id { get; set; }
            public EditScheduleRequest Dto { get; set; }
            public Command(int id, EditScheduleRequest dto)
            {
                Id = id;
                Dto = dto;
            }
        }

        public class Handler : IOperationHandler<Command>
        {
            private readonly DatabaseContext _dataContext;
            private readonly IMapper _mapper;
            public Handler(DatabaseContext dataContext, IMapper mapper)
            {
                _dataContext = dataContext;
                _mapper = mapper;
            }

            public async Task<OperationResult> Handle(Command request, CancellationToken
            cancellationToken)
            {
                var result = new OperationResult();

                var schedule = await _dataContext.Schedule.FindAsync(request.Id);
                _mapper.Map(request.Dto, schedule);
                if (!string.IsNullOrEmpty(request.Dto.TestVenueName))
                {
                    var testVenue = new TestVenue {
                        Name = request.Dto.TestVenueName,
                        IsDeleted = false
                    };

                    testVenue.Schedules.Add(schedule);
                    await _dataContext.TestVenue.AddAsync(testVenue);
                }

                .
                .
                .

                return result.Success();
            }
        }
    }
}
```

Figure 10: Implementation of class for the write model.

```
namespace UK.Domain.Mediator.CompetencyAssessment.Command
{
    public class DeleteSchedule
    {
        public class Command : IOperationRequest
        {
            public int Id { get; }
            public Command(int id)
            {
                Id = id;
            }
        }

        class Handler : IOperationHandler<Command>
        {
            private readonly DatabaseContext _dataContext;
            public Handler(DatabaseContext dataContext)
            {
                _dataContext = dataContext;
            }

            public async Task<OperationResult> Handle(Command request, CancellationToken
            cancellationToken)
            {
                var result = new OperationResult();

                var unit = await _dataContext.Schedule.FirstAsync(u => u.Id == request.Id);
                unit.IsDeleted = true;

                return result.Success();
            }
        }
    }
}
```

Figure 11: Implementation of class for the write model.

Some of the advantages when we apply the CQRS method are better performance of the built system due to better concurrent access handling, as well as a domain model and a simple query model.

5 CONCLUSIONS

A common approach used by most people to interact with information systems is to treat it as a data store (Create Read Update Delete). We have a model of multiple record structures for creating new records, reading records, updating existing records, and deleting records when we are done with them. In the simplest of systems, our interaction is about keeping and retrieving these records. As technology develops in information systems, our needs become more sophisticated, we try to develop models that are commonly used. We view information in different ways, perhaps breaking it down into several records.

CQRS using a significant change structure, can be a significant solution to create information systems. There are several benefits to using CQRS. The first is that some complex domains are easier to handle using CQRS. There is usually a fair amount of overlap between the command and query sides to make sharing the model easier. Another benefit is that in handling high-performance applications, CQRS separates the load from reads and writes so we can scale it independently. When it comes to distinguishing between reading and writing, it is very useful.

Based on the description that has been described in the previous section, it can be concluded as follows. The CQRS implementation basically simplifies the design so that it has separate read (query) model and write (update) models. Separation of the read and write sides will make system maintenance easier. Most of the complex business logic will be in the write model. Compared to the written model, the reading model can be relatively simpler. By using a pattern like this, system development will be more flexible when changes need to be made. On the other hand, the application can avoid complex joins when creating queries.

REFERENCES

BNSP, (2017). *Peraturan Badan Nasional Sertifikasi Profesi No. 2/BNSP/VIII/2017*, Badan Nasional Sertifikasi Profesi, Jakarta.

Meyer, B. (1997). Object-oriented software construction, 2nd edn. Prentice Hall, New Jersey.

Domínguez, J. and Melnik, G. (2012). A Journey into CQRS, Patterns & Practices Symposium Online 2012, Retrieved from http://channel9.msdn.com/Events/Patterns-Practices-Symposium-Online/Patterns-Practices-Symposium-Online-2012/A-Journey-into-CQRS.

Betts, D., Domínguez, J., Melnik, G., Simonazzi, F. and Subramanian, M. (2012). Exploring CQRS and Event Sourcing - A journey into high scalability, availability, and maintainability with Windows Azure, 2012, Retrieved from http://msdn.microsoft.com/en-us/library/jj554200.aspx.

Niltoft, P. and Pochill, P. (2013). Evaluating Command Query Responsibility Segregation. Lund: Department of Computer Science, Faculty of Engineering, LTH, Lund University.

Erb, B. and Kargl, F. (2014). Combining discrete event simulations and event sourcing, *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pp. 51-55.

Fitzgerald, S. (2012). State Machine Design, Persistence and Code Generation using a Visual Workbench, Event Sourcing and CQRS, School of Computer Science and Informatics, University College Dublin, Dublin.