Research Report

SUE Data Science platform – SA-RB-01

# Version history

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Author(s) | Amendments | Status |
| 0.1 | 16-09-2024 | Mohammed Bouali, Calvin Kwan, Andrei Pieleanu, Björn Wildeman | Initial Version | In process |
| 1 | 13/12/2024 | Mohammed Bouali, Calvin Kwan, Andrei Pieleanu, Björn Wildeman | Final version | FINISHED |

Table of Contents

[Version history 2](#_Toc177564687)

[Main Research Question 4](#_Toc177564688)

[Sub-Questions 4](#_Toc177564689)

[How will we implement the CLI application’s functionalities into the API? 4](#_Toc177564690)

[Literature study: 4](#_Toc177564691)

[Interview: 4](#_Toc177564692)

[IT architecture sketching: 4](#_Toc177564693)

[How will the API be deployed? 4](#_Toc177564694)

[Design pattern research: 4](#_Toc177564695)

[Conclusion: 5](#_Toc177564696)

[IT architecture sketching: 5](#_Toc177564697)

[How can we ensure that our software solution is GDPR compliant? 6](#_Toc177564698)

[Security check: 6](#_Toc177564699)

[Ethical check: 6](#_Toc177564700)

[How can we make our API able to communicate with SUE Infrastructure? 6](#_Toc177564701)

[Literature study: 6](#_Toc177564702)

[Software Architecture 6](#_Toc177564703)

[3-Tier Architecture 6](#_Toc177564704)

[Event-Driven Architecture 7](#_Toc177564705)

[Conclusion 7](#_Toc177564706)

[References 7](#_Toc177564707)

# Main Research Question

How do we implement the functionalities that are in the current CLI into an API that can be integrated into SUE’s infrastructure so that they can offer it as a service to their clients?

# Sub-Questions

* How will we implement the CLI application’s functionalities into the API?
* How will the API be deployed? (Design pattern research + IT architecture sketching)
* How can we ensure that our software solution is GDPR compliant? (Security test + Ethical check)
* How can we make our API able to communicate with SUE Infrastructure? (Explore user requirements + literature study + system tests)

Using ICT DOT Framework (Bonestroo, Meesters, Niels, Schagen, & van Turnhout, n.d.) (van Turnhout, et al., 2015) Methodology to find answers for these sub-questions:

## How will we implement the CLI application’s functionalities into the API?

Literature study: To search the most-suited architecture that can be used for the API

For this project, there are some architectures that are considered suitable. In this section, those architectures will be described and chosen according to user requirements.

## 3-Tier Architecture

The 3-tier architecture utilizes as the name suggests, 3 layers, each with its own responsibility:

* Data layer – responsible for database command and retrieval of raw data.
* Logic layer (business layer) – responsible for getting the raw data from Data layer and perform operations on it, such as conditions, exceptions handling, etc.
* Interface layer – responsible for displaying data that has been calculated by the logic layer.

**Pros**:

* Clear separation of layers (Presentation, Business, Data).
* Scalability and maintainability due to modular structure.
* Easier to adapt to changes in specific layers.

**Cons**:

* May require more effort to set up initially.

## Event-Driven Architecture

According to (Amazon AWS, 2023), Event-Driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website.

Event-Driven architecture have 3 important components:

* Event producers – they publish an event to the router.
* Event routers – responsible for managing events and send them to the corresponding consumers.
* Event consumers – they listen to the producer events and produce an output.

**Pros**:

* Scalability by having the possibility of expanding the system endlessly with as many microservices as possible.
* Decoupling when one service fails the rest will keep running. The event router acts like an elastic buffer that will accommodate surges in workloads.

**Cons**:

* Steep learning curve for beginner, so it will take more time to learn and use it.
* Difficult to provide root-cause analysis of any given failure. As event producers and consumers increase in numbers, it becomes harder to track the activities occurring withing and between them.

After careful consideration, we realize that both architectures have their strengths and weaknesses. So, we conclude that we will utilize a hybrid approach with those two architectures.

Prototyping: To create a prototype of the API to prove the chosen architecture able to work properly together

To prove that it is possible to implement the hybrid architecture, a proof of concept will be created to prove the possibility of creating the gRPC API with golang utilizing the chosen architecture.

The application only has one functionality, which is accepting a login request with gRPC protocol. If the user is not registered to the application, it will automatically register the user with the provided credentials then sign the user in. On the other hand, if user is registered but send the wrong password, it will return a bad request status response back. If user is successfully logged in, the application will create a log in the database the user is logged in.

The proof of concept’s architecture will be shown in image below

A white rectangular object with black text

Description automatically generated

proof of concept’s architecture diagram

Here is the explanation for the diagram above:

1. A login request is sent to the Kong gateway
2. Kong will redirect the request to user service
3. User service will perform the login logic
4. After login logic successfully executed, return the response to Kong gateway
5. The gateway will return the response to the request’s sender
6. If the login is successful, a RabbitMQ message will be sent to the log service
7. The log service will perform the logging functionality

For each service, 2-tier architecture is implemented. Below is the sample code for each layer at user service:

* Controller layer

syntax = "proto3";

package user;

option go\_package = "../controllers";

message LoginRequest {

    string email = 1;

    string password = 2;

}

message LoginResponse {

    string accessToken = 1;

    string refreshToken = 2;

}

message LogMessage {

    uint64 userID = 1;

    string action = 2;

}

service UserService {

    rpc Login(LoginRequest) returns (LoginResponse);

}

For controller layer, a proto file is created to generate a controller layer’s part of the code

* Business layer

func (service \*UserServiceImpl) Login(ctx context.Context,

request \*controllers.LoginRequest) (\*controllers.LoginResponse, error) {

    /\*

        Function that authenticate user's login attempt

        return:

        LoginResponse   : response filled with token and refresh token

        error           : error that happen during authentication

    \*\*/

    // Get the user from repository layer

    user, err := service.userRepo.FindByEmail(request.Email)

    // Return the error if there is an error during user data retrieval

    // that is not related to user not found

    if err != nil && err != gorm.ErrRecordNotFound {

        return nil, err

    }

    if user == nil {

        // Register the user if user data is not found

        log.Printf("User not found.\nRegistering new user: %s", request.Email)

        // Encode the password

        hashedPassword, err := security.EncodePassword(request.Password)

        // Checks if there is any error during hashing

        if err != nil {

            return nil, err

        }

        // Create a new user object

        newUser := &models.User{

            Email:    request.Email,

            Password: hashedPassword,

        }

        // Store the new user object in the database

        user, err = service.userRepo.CreateUser(newUser)

        // Check if there is any error during the insertion process

        if err != nil {

            return nil, err

        }

    } else {

        // If user is found, check if the password match with the one in database

        if match, err := security.MatchPassword(request.Password, user.Password); !match {

            return nil, err

        }

    }

    // Generate the jwt tokens

    accessToken, refreshToken, err := security.GenerateTokens(user.ID)

    // Checks if there is any error during token generation

    if err != nil {

        return nil, err

    }

    // Create a message

    message := &messages.LogMessage{

        UserID: user.ID,

        Action: "LOGIN",

    }

    // Encode the message

    messageBody, err := json.Marshal(message)

    if err != nil {

        return nil, err

    }

    // Send the message to log service

    go func() {

        service.rabbitMQ.SendMessage("log", "log", "direct", string(messageBody))

    }()

    // Return the tokens in the LoginResponse format

    return &responses.LoginResponse{

        AccessToken:  accessToken,

        RefreshToken: refreshToken,

    }, nil

}

Note that there is a message sending functionality with RabbitMQ inside of a goroutine function, which indicates an asynchronous message sending using RabbitMQ

* Repository layer

func (repo \*UserRepositoryImpl) CreateUser(user \*models.User) (\*models.User, error) {

    /\*

        Create a new user object and store it to database

        parameter:

        user    : user object that want to be stored

        return:

        user    : stored user object

        error   : error during the insertion process

    \*\*/

    // Create the user

    result := repo.db.Create(user)

    // Check if there is any error during user creation

    if result.Error != nil {

        return nil, result.Error

    }

    // Return the user object

    return user, nil

}

func (repo \*UserRepositoryImpl) FindByEmail(email string) (\*models.User, error) {

    /\*

        Find user data in the database by email

        parameter:

        email   : user's email

        return:

        User    : found user

        error   : error during the search

    \*\*/

    var user models.User

    // Get the user from database

    result := repo.db.Where("email = ?", email).First(&user)

    // Check if user retrieval failed

    if result.Error != nil {

        return nil, result.Error

    }

    // Return the user

    return &user, nil

}

For Kong Api Gateway, a docker-compose is used to run the Kong Gateway. Since the gateway will run in DB-less mode, a kong.yml is used to register services and routes (and possibly plugins for FamTime development). This is the setup for kong.yml:

\_format\_version: "3.0"

\_transform: true

services:

  - name: login-service

    host: host.docker.internal

    protocol: grpc

    port: 50051

    routes:

      - name: login-route

        paths:

          - /user.UserService/Login

If you notice the host of the service, it is not localhost, but “host.docker.internal”, this is caused by the fact that Kong is running inside the docker network, which is a different network from the user service that is running in developer’s local machine environment, that is why the host is not localhost.

IT architecture sketching: Provides developers insights on the software architecture which can be seen and commented by other developers and stakeholders

Below, images depicting the planned end product’s architecture will be shown in the form of a C4 diagram.

A diagram of a data scientist

Description automatically generated

C1 Diagram

A diagram of a software company

Description automatically generated with medium confidence

C2 Diagram

A diagram of a software company

Description automatically generated

C3 Diagram

## How will the API be deployed?

Design pattern research: To find information about the existing technologies that can be used for our case in articles and/or books.

For our API to be deployed we have a lot of options, but it all comes down to our needs and the requirements of the client, below are the methods of deployment that we will use with an explanation of why.

**Containerization with Docker:** In order to make it easier to deploy, we will containerize our application using docker, here are some important advantages of containerization with docker:

* **Portability:** Applications run consistently across different environments, whether on a developer’s machine or in production.
* **Isolation:** Containers encapsulate applications and their dependencies, reducing conflicts and improving security.
* **Scalability:** Easily scale applications up or down by adding or removing containers based on demand.
* **Resource Efficiency:** Containers share the host OS kernel, making them lightweight and faster to start compared to traditional VMs.
* **Simplified DevOps:** Streamlines CI/CD processes, enabling faster deployment and easier rollback of applications.
* **Microservices Architecture:** Facilitates the development of applications as a set of small, independent services, improving maintainability and flexibility.
* **Version Control:** Allows for easy versioning of container images, making it simple to manage and deploy different versions of applications.

**Deployment in the AWS Kubernetes cluster:** Since the Kubernetes cluster will be deployed in AWS (Please refer to the Software Architecture Document for more details), we originally considered deploying the API within the same AWS cluster. The initial reasoning was that having the API and Kubernetes in the same network and cluster would facilitate easier communication, as the API would essentially be a pod within the Kubernetes cluster.

However, after further evaluation, we have decided that the best course of action is to keep the API external to the Kubernetes cluster. The primary reason for this change is that SUE prefers to deploy the API according to their own infrastructure setup. Since they will not have the API within the same cluster as Kubeflow, it makes more sense to provide an external API that can be deployed independently on their infrastructure.

**Conclusion:**

To conclude, the best course of action is to keep the Go API external to the Kubernetes cluster. This approach allows SUE to deploy the API according to their own infrastructure preferences, which aligns with their operational requirements. The API will be containerized using Docker, but it will not be deployed as a pod within the Kubernetes cluster. Instead, it will be set up as an external service that can be reached via gRPC, with appropriate AWS networking configurations to ensure security and accessibility.

IT architecture sketching: We make some sketches of how our IT architecture could potentially look like so that we can choose the one that mostly works for us and the client

There are multiple ways to implement our software solution, we have come up with 4 different versions of an architecture, here they are:

In this first version, the data scientist connects to SUE, then their Infrastructure will connect with our GO API, which in return will connect with the Kubernetes cluster which is hosted in AWS, and based on the config file that our GO API will send the Kubernetes cluster will spin up a Kubeflow instance.

**Advantages of this approach:**

* The API is a monolith application, which makes it less complex to develop
* The API is deployed externally, therefore it will be easier to have it communicate with SUE’s Infrastructure using gRPC

**Disadvantages of this approach:**

* The API is a monolith application, which makes it difficult to scale
* The API is deployed externally, which makes it more complex to communicate with Kubernetes

A diagram of a software process

Description automatically generated with medium confidence

In this second version, the difference is that the API is deployed as a pod inside the Kubernetes cluster.

**Advantages of this approach:**

* The API is a monolith application, which makes it less complex to develop
* The API is deployed inside the Kubernetes cluster, which makes it easier to communicate between them

**Disadvantages of this approach:**

* The API is a monolith application, which makes it difficult to scale
* The API is deployed in the cluster which makes it more difficult to communicate with SUE’s Infrastructure

A diagram of a diagram

Description automatically generated

In this approach the API is in a microservices architecture, in addition to that it is part of the Kubernetes cluster

**Advantages of this approach:**

* The API is a microservices architecture, which makes it easier to scale
* The API is deployed inside the Kubernetes cluster, which makes it easier to communicate between them

**Disadvantages of this approach:**

* The API is deployed in the cluster which makes it more difficult to communicate with SUE’s Infrastructure

A diagram of a company

Description automatically generated

In this approach the API is in a microservices architecture, but it is external, so not inside the Kubernetes cluster

**Advantages of this approach:**

* The API is a microservices architecture, which makes it easier to scale
* The API is not in the Kubernetes cluster, therefore it is easier to communicate with SUE’s Infrastructure

**Disadvantages of this approach:**

* The API is external, therefore it is more complex to communicate with Kubernetes

A diagram of a software company

Description automatically generated

## How can we ensure that our software solution is GDPR compliant?

Security check: Find vulnerabilities in the IT system by conducting security tests and ensure our solution is error-free.

According to Durgaprasad Budhwani, we can use Gosec provided by the Go team. gosec — Golang Security Checker is an open-source SAST tool specifically designed for Golang. It is designed to detect common security issues in Golang code such as SQL injection, cross-site scripting (XSS), and buffer overflows. Gosec works by analyzing the source code of a Golang program and generating a report that highlights any security issues found. (Budhwani, 2023)

Ethical check: Ensure that our solution is in accordance with the GDPR standards.

For ethical check, we have to make sure that only necessary data is being collected. Golang offers ‘struct’ validation to ensure the app doesn’t process unnecessary/excessive personal data.

It is designed to detect common security issues in Golang code such as SQL injection, cross-site scripting (XSS), and buffer overflows. Gosec works by analyzing the source code of a Golang program and generating a report that highlights any security issues found. (Budhwani, 2023)

### How do we use Gosec in our application?

We first began by downloading Gosec. To achieve this, we ran the following command in the terminal: ‘go install [github.com/securego/gosec/v2/cmd/gosec@latest](mailto:github.com/securego/gosec/v2/cmd/gosec@latest)’

Then we went on each folder (microservice) and began running the following command: ‘gosec ./…’. This command looks in every file recursively from that folder and looks for security and potential issues.

Below is an example of running the command on notebook-service

***Results:***

***Summary:***

***Gosec : dev***

***Files : 23***

***Lines : 2267***

***Nosec : 0***

***Issues : 0***

This means out of a total of 23 files and 2267 lines of code, there are no security issues or issues in general. Our code is clean and safe to be used, as well as safe from XSS and injection attacks.

The same procedure has been applied to all microservices. No issues have been found.

## How can we make our API able to communicate with SUE Infrastructure?

Explore user requirements:

To determine how we can communicate with the SUE infrastructure, we first needed to gain a deeper understanding of the system. To achieve this deeper understanding, we conducted field research method; Explore user requirements. This approach allowed us to gather detailed insights into how the stakeholder plans to use our API. We met with the stakeholder and asked specific questions regarding the SUE infrastructure and the best methods for integrating with it. During these discussions, the stakeholder informed us that most of their infrastructure relies on the gRPC protocol for communication and that it would be preferable that our solution uses this protocol as well. Important is that the API is well documented.

Literature study: After identifying that the SUE infrastructure uses gRPC for communication, we realized that none of us were familiar with this protocol. To address this, we conducted a literature review to better understand gRPC.

Through our research, we learned that gRPC is an open-source remote procedure call (RPC) framework developed by Google. It uses HTTP/2 for transport, which enables features like multiplexed streams and low-latency communication. Additionally, gRPC supports multiple programming languages, making it well-suited for connecting services in a distributed system. It also uses Protocol Buffers (protobufs) for efficient data serialization, ensuring fast and compact data transmission. (Google, n.d.)

System Tests: After understanding gRPC, we began integrating it into our API implementation. Once we had implemented key parts of the system, we used Postman to test the API calls. Since full integration of our API was out of scope, demonstrating that Postman successfully fulfilled the functional requirements through gRPC API calls was sufficient to validate the system's functionality. In addition to this, we conducted other tests, including unit testing, to ensure the robustness of the system.

# References

Amazon AWS. (2023). What is an Event-Driven Architecture? *AWS*.

Bonestroo, W., Meesters, M., Niels, R., Schagen, J. D., & van Turnhout, K. (n.d.). *ict researchmethods*. Retrieved from ictresearchmethods: https://ictresearchmethods.nl

Budhwani, D. (2023). Bulletproofing Your Golang Applications with Automated Security Checks Using Gosec and GitHub Actions. *medium.com*.

Google. (n.d.). *A high performance, open source universal RPC frameworkf*. Retrieved from gRPC: https://grpc.io/

van Turnhout, K., Jacobs, M., Kamp, I., Mulholland, C., Neuman, A., Rouwhorst, S., & van Vlies, L. (2015). *CMD Methods Pack: Find a combination of research methods that suit your needs.* (HAN University of Applied Sciences - Amsterdam University of Applied Sciences, the Netherlands. ) Retrieved from cmdmethods: https://cmdmethods.nl

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. Linux Journal, 2014(239), 2. https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71-79. https://doi.org/10.1145/2723872.2723882

Pahl, C., & Lee, B. (2015). Containers and microservices: A new approach to application deployment. IEEE Cloud Computing, 2(2), 68-71. https://doi.org/10.1109/MCC.2015.47

Kelsey Hightower, Brendan Burns, & Joe Beda. (2017). Kubernetes Up & Running: Dive into the Future of Infrastructure. O'Reilly Media.