ISSN 2347 - 3983
Volume 8. No. 7, July 2020
International Journal of Emerging Trends in Engineering Research
Available Online at http://www.warse.org/IJETER/static/pdf/file/ijeter124872020.pdf
https://doi.org/10.30534/ijeter/2020/124872020

# Designing and Evaluating Representational State Transfer Architecture for School Management Information System

**David Alfa Sunarna[1], Gede Putra Kusuma[2]**
[1] Computer Science Department, BINUS Graduate Program - Master of Computer Science, Bina Nusantara University, Jakarta, Indonesia, 11480, david.sunarna@binus.ac.id
[2] Computer Science Department, BINUS Graduate Program - Master of Computer Science, Bina Nusantara University, Jakarta, Indonesia, 11480, inegara@binus.edu

## ABSTRACT

Technology is rapidly evolving to became main component in education process. Nowadays, schools are relying on Management Information System (MIS) to handle operations. In this paper we propose to design SMIS architecture capable of serve multiple schools by using Representational State Transfer (REST) as architectural style. Case study, REST constraint and Service Oriented Architecture (SOA) design pattern is being used to design the architecture. From the design process, multitenant architecture is chosen. Kubernetes is chosen as deployment orchestration to make architecture scalable. Then we examine and evaluate engineering trade-off and risks in architectures design decision by using Architecture Trade-off Analysis Method (ATAM). The evaluation is based on scenarios and each scenario related to several software quality attributes. At the end, the architecture implementation is improved overtime based on the evaluation result.

**Key words:** Educational Technology, School Information System, REST Architecture, Web Service, Trade-off Analysis.

## 1. INTRODUCTION

As the technology developed rapidly, school also improve in terms of technology implementation. School that used SMIS to manage operation are ready in term of fundamental technology, network system, and internet connection [1]. School nowadays integrate school fees into virtual account, make push notification to stakeholder by email or smartphone, and share educational data to government. School using ID card with Radio-frequency identification (RFID) and IoT Technology [2][3]. It being used for entering the building, park vehicle, borrow books in library, record attendance, and real-time monitoring student activities. All of this comes down to SMIS that should be capable to adapt with school's needs.

Because SMIS nowadays requires many integrations, Service-Oriented Architecture (SOA) is chosen. SOA is an architectural concept in software design that emphasizes the use of combined loosely coupled services to support business requirements directly [4]. As the web technology is developing rapidly, web service used as service-based technology. Web service technology make the SMIS can be used across different education company and school. The functionality can be reused in different platform according to school needs and configuration [5]. Also with SOA, different application can communicate with each other by using same security protocol [6].

This web service will be implemented by using Representational State Transfer (REST) principle. REST is a SOA design for hypermedia or distributed system [7]. REST is an architectural style for creating web SOA and often called RESTful web server. It has become the industry standard on large-scale SOA [8].

This research creates new centralized architecture for SMIS based on case study. To design the architecture, this research use REST constraint, SOA design pattern, and REST design pattern [7][9]. Architecture Trade-off Analysis Method (ATAM) evaluate engineering trade-offs and risks in architectures design decision [10]. The evaluation is scenario based and focuses on software quality attributes [11][12].

## 2. LITERATURE REVIEW

REST is architectural concept founded by Roy Fielding in 2000 and became popular due to its simplicity and lightweight development model [13]. Khan and Abassi describe REST have better performance in latency and smaller packet size [14]. The application of REST is gaining popularity in 2011 and shows REST is the correct architecture for the web [15]. This architecture follows six REST constraint described by the founder, Roy Fielding [7].

Beside six REST constraint, this paper also uses design pattern to create the architecture. Design patterns provide proven solution to a common problem in software design [9]. The problem is documented in a standard format and part of larger collection. Design pattern is being used to designing architecture based on case problem because provides field-tested solution. Because the solution is already available and tested, design patterns can speed up the development process. Erl has already defined eighty-five design pattern profiles for SOA [9]. Also, there are seven REST-inspired new design patterns to solve problem by using REST capabilities [16].

Quality of an architecture/software is defined by software quality attributes as parameters. Quality attributes capture functional requirement that achieved by the application and set a minimum standard for application [17].

To be useful, quality attributes must be specified clearly along with its general scenario. A statement saying, "The application must be scalable", is not enough to clearly define what kind of scalability an application is facing off. Is application should be capable to handle more request? how much request? How many concurrency users the system should be capable of? Or all them is needed in the system. That is why a quality attributes must be described or realized as a scenario such as:

*When average CPU usage is above 70%, it must be possible for the system to clone/raise up server specification until average CPU usage is below 70% with zero downtime.*

Statement above is a good example of scenarios, precise and meaningful. Then, to evaluate architecture decision from scenarios a framework called ATAM is being used. ATAM is a scenario-based testing framework to evaluate quality attributes and understand trade-off between architecture decision [18].

A single testing scenario should be able to reflect what software quality attributes to be achieved. Architecture Trade-off Analysis Method (ATAM) help determine every architectural trade-off points location by using scenario testing, and made us understand design limitation [19]. This information is useful for making action plans for evaluation, started new iteration of the method, and modifying architecture based on the evaluation. ATAM trying to improve architecture qualities in each iteration of the method. ATAM output also raise awareness to stakeholder [10].

Costa et al. present guidelines to evaluate architecture in REST-based system based on ATAM [12]. Based on the interview with architecture evaluators, REST must meet six foundation that described in its original research [7]. The research also generated basic template for REST quality attributes scenarios and how REST design can affect software quality attributes (design question). These guidelines then being used in for evaluating REST architecture and define the trade-offs.

## 3. METHODOLOGY

Figure 1 described the research activity we followed to design and evaluate REST architecture for SMIS. Literature review have already discussed in previous chapter. This chapter discuss design process to create initial architecture in SMIS based on case study, REST constraint, and SOA design pattern.

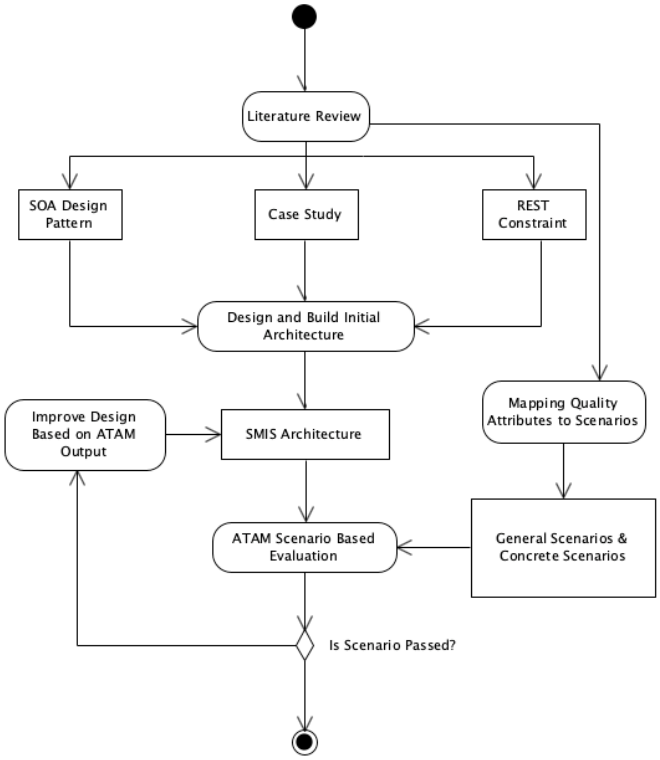


**Figure 1:** Research Activity Diagram

This research focused on SMIS RESTful architecture design for K-12. It designs, evaluate, and modify architectural strategies to handle challenges in SMIS. Then the problem is formulated and mapped to REST-inspired SOA design pattern [16]. This research also designs the architecture by following REST constraint [7]. The design result is initial SMIS architecture and deployment strategy. The ATAM evaluation being used in this research is specialized for REST architecture [12]. This evaluation use scenarios related to quality attributes obtained from literature review. It examines REST standard, REST design question, ATAM scenario-based testing, and the tradeoff in architecture decision.

### 3.1 REST Constraint

As mentioned above this architecture use six REST constraint described by Roy Fielding. The first and basic constraint in REST is client-server. The client makes request and the server responds to request. This architecture separate

client-side-logic and server-side-logic. The server-side-logic expose Application Programming Interface (API) and uses HTTP specification as communication protocol. It can be accessed through URI and return JavaScript Object Notation (JSON). The main benefits of the Client-Server style are separation of responsibilities, independent evolution and maintainability [12].

Second REST constraint being implemented is stateless. The fundamental explanation of stateless is no client session state is stored on the server. It means that the server-side-logic does not store any state about the client session on the server side. This constraint is very important because the system should be capable to scale up and scale down. To keep server-side-logic stateless, this architecture use JSON Web Token (JWT). JWT are an open, industry standard RFC 7519 method for representing claims securely between two parties as a JSON object [20].

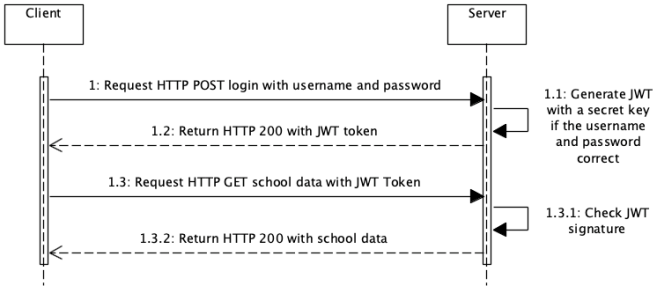


**Figure 2:** JSON Web Token

From Figure 2, a login request is coming from the client-side-logic to server-side-logic. If the username and password is correct, server-side-logic generate token based on secret key. Then, token is being saved to a client-side-logic HTTP header with the format '*Authorization: Bearer [JWT token]*'. If there is another request from client-side-logic again, server-side-logic match JWT signature by using a secret key (HMAC algorithm) or public/private key using RSA.

The next REST constraint is being used is uniform interface. It contains three elements: methods, media types, and resource identifier syntax. SMIS using URI standard to express where the data is being transferred to or from. It is also URL because we can use it as a resource identifier and apply methods upon it [16]. Below is the general syntax of the URI being used in initial architecture:

*{scheme}://{authority}{path}?{query}* (1)

An example of URI that is using all the components is "*http://client.smis.example/customer/school-levels/10?page=2*". Table 1 shows breakdown of each component in URI and its function to support the architecture.

**Table 1:** URI's component breakdown

| URI's Part | Usage |
|---|---|
| http | scheme/methods |
| client.smis.example | authority |
| /customer/school-levels | path |
| 10 | resource identifier refers to school level primary key in the table |
| ?page=2 | query |

Same URI can be used multiple times to serve different request by using HTTP method. The URI below shows HTTP method available for school level in RESTful API. The URI is "*http://client.smis.example/customer/school-levels*". Table 2 shows meaning of each HTTP method in school level API. The resource identifier being used is school level primary key with the value of 10.

**Table 2:** HTTP Method Pattern

| HTTP Method | URI | Result |
|---|---|---|
| GET | *http://client.smis.local/ customer/school-levels* | Get all school level list |
| PUT | http://client.smis.local/ *customer/school-levels* | Create new school level |
| PATCH | http://client.smis.local/ *customer/school-levels/10* | Edit school level with primary key 10 |
| DELETE | http://client.smis.local/ *customer/school-levels/10* | Delete school level with primary key 10 |

This architecture using layered system architecture as one of REST constraint. For example, attendance contains of three layers: SMIS API server, SMIS database server, RFID tapping machine API server. When client-side-logic requests attendance data, it interacts only with API server without knowing there is also another layer supporting it.

Redis are being used for cache management in this architecture design. Meanwhile, Code-on-Demand (COD) is the only optional constraint in REST because it reduces visibility. For instance, with Code-On-Demand, a client can download a JavaScript, java applet or even a flash application in order to encrypt communication so servers are not aware of any encryption routines / keys used in this process. Interoperability also decrease because code must be compatible with target consumers. Security also became a concern because it can be injected with malicious code [14].

### 3.2 Design Pattern

REST-inspired SOA design pattern are used to solve the case study problem. This design pattern is already optimized for

REST architectural style [16]. The first design pattern used is content negotiation and its related to media types in uniform interface. Media type are expressed both in HTTP request header and HTTP response header. Client-side-logic determine media type by using *Accept* in *Request Header*. When the server-side logic returned the value, it also confirms media type that being returned to the client with *Content-Type* in Response header.

The next design pattern is reusable contract. The goal of this architecture is to serve many education companies and centralized the system. Reusable contract is related with uniform contract constraint. One contract should be available to all education company. Also, each of education company is having many schools spread in different location. There must be a contract that also shared internally between education company.

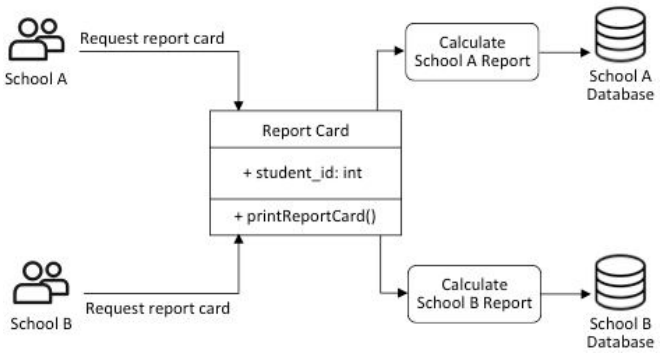


**Figure 3:** Reusable Contract Implementation

Figure 3 explain how SMIS handle multiple education company by using reusable contract. Report card is different on each client and need different service. Both of client accessing the same contract which is *GET – printReportCard()*. The request is forwarded into different service with capabilities "print report card for client 1" and "print report card for client 2". The only thing differentiates the request is base URL. Client 1 using base URL *http://client1.smis.example* and Client 2 using *http://client2.smis. example*.

## 4. RESULT AND EVALUATION

From the design process, the best approach for architecture pattern in SMIS is multitenancy. Multitenancy allow multiple customer called tenant, sharing system resources but keep configuration and data for each tenant exclusive [21]. The server-side-logic and client-side-logic can serve multiple tenants which means multiple education company. In every tenant there is many schools with certain year level and curriculum. This also need specialized configuration. Database configuration for each tenant is separate database and separate schema [22]. Database configuration for each school in tenant is shared database and shared schema. Separation in tenant data is critical properties in multitenant applications [23].

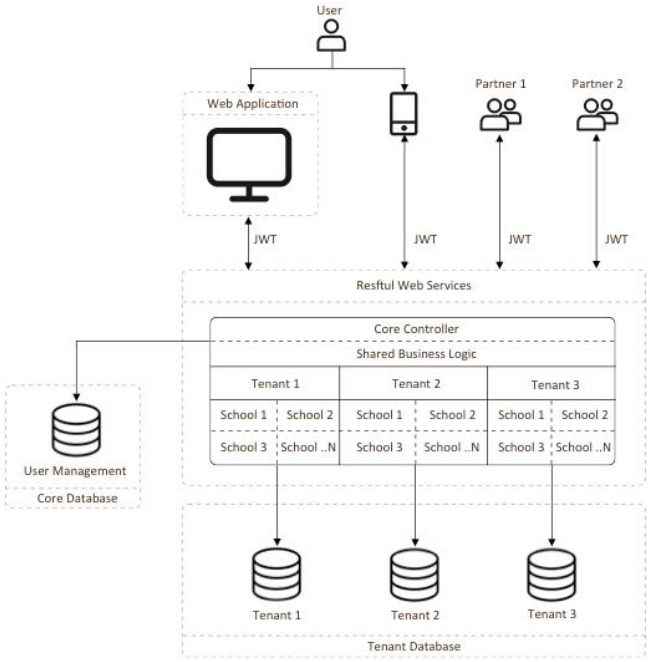


**Figure 4:** Initial Architecture Design

Figure 4 described how initial architecture handle multitenancy. Core controller and core database handle shared business logic (model) being used together such as authentication, tenant setting and user management. Core controller and core database define user request belong to which tenant. It responsible to manage request and forward it into specific tenant controller, model, and database. When new tenant created, a new database is created in tenant database section. Tenant data is isolated and not mixed with others. User access SMIS through single web application and mobile phone but with different theme and styling. There is also third-party application accesses the system by using authorization token.

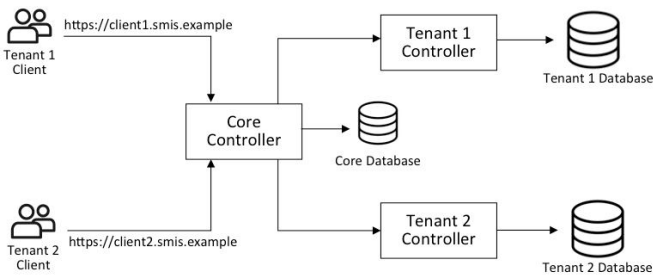


**Figure 5:** Core Controller

Figure 5 explain how request is separated into different tenant. In this architecture, each tenant has different API endpoint. Every request goes through core controller. First, core controller matches the API endpoint. All the API endpoint with belonging tenant stored in core database. If the API endpoint is registered, core controller checks the request access token. Valid access token and its belonging user also stored in core database. Finally, request is forwarded to correct tenant.

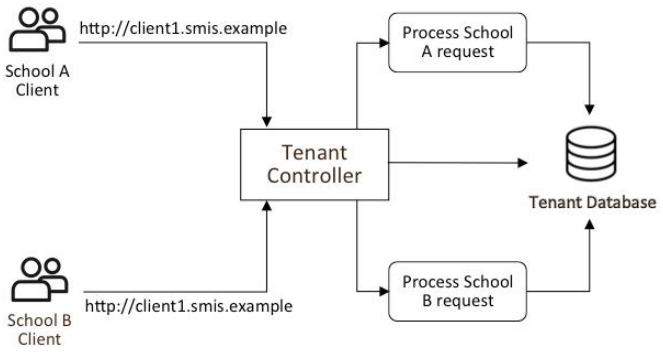


**Figure 6:** Tenant Controller

This architecture provides two levels of multitenancy. Although lies in the same tenant, it is possible for each school have special configuration. In Figure 6, tenant controller is responsible to handle school specific request and forward it to school controller and model. This architecture not only separate each education company but also separate each school inside education company. This approach is done by creating database relation that coupled school with year level and school level.

### 4.1 Deployment Strategy

All the benefit of multitenant architecture needs a right deployment strategy. Software deployment takes a role key in relevant aspect of multitenant system such as performance, availability, reliability, and fault tolerance [23].

This architecture use cloud server, docker container engine and Kubernetes for architecture orchestration. Container have equal and better performance than Virtual Machine (VM) [24]. Kubernetes cluster is created by multiple machines both virtual and physical. Each machine called node and can be increased to scale out application. A Pod is a group of one or more container with shared storage and network to make application running [25]. Pod can be replicate to scale up and destroyed to scale down application.

This architecture deployed in cloud that provide Kubernetes cluster. The application images are saved on the cloud registry. Application database lies outside Kubernetes cluster and using cloud database. To create stateless architecture, shared files and all the multimedia data are stored in cloud storage.

### 4.2 Evaluation Scenarios

The evaluation use quality attributes as a benchmark. It mainly focusses on system performance, scalability, and security quality attributes. Performance needed to make sure tenant creation is fast and increasing number of tenant endpoint does not decrease user experience. Scalability needed to make sure system is capable of scaling in and scaling out when the number of concurrent users increased dramatically. Template for creating general scenarios is obtained based on the research created by Costa [12].

**Table 3:** General Scenarios

| Quality Attribute | General QA Scenario |
|---|---|
| Security | SE1 - Client 'A' makes a request to service 'B' with correct authentication, service 'B' will give the token. |
| | SE2 - Client 'A' makes a request to service 'B' with false authentication, service 'B' will reject the request. |
| | SE3 - Client 'A' makes a request to service 'B' with correct authentication but false tenant, service 'B' will reject the request. |
| Scalability | SC1 - Server-side logic is proven to be stateless and scalable. |
| | SC2 - Server-side logic save and get media data (picture, videos, etc.) in another storage to support horizontal scaling. |
| | SC3 - Tenant website and service API are created seamlessly with zero configuration on the server. |
| | SC4 – REST Application can auto scale up and scale out when the CPU threshold is reaching certain level. |
| Performance | P1 – Tenant creation is fast, easy, and does not slow down entire system. |
| | P2 – System is being hit by certain amount of tenant at measured time and does not decrease user experiences. |
| | P3 – System is being hit by certain amount of concurrent request at measured time and does not decrease user experiences. |
| Interoperability | I1 - Client 'A' is created with different platform, make a request to service 'B' with specified media type. Service `B` return correct media format. |
| | I2 - Client 'A' is created with different platform, make a request to service 'B' and filling the authorization http header with the token, service 'B' will validate the request and responses to 'A'. |
| Testability | T1 - Service can be configured to give information needed to identify the fault. |

General scenarios, as shown in Table 3, tell us what need to be achieved in by the architecture. It mostly gathered from stakeholders. Interoperability and testability only support the capabilities of three main quality attributes. Security needed to make sure data in each tenant is secure and cannot be accessed by other tenant. General scenarios detailed in a way as described on Table 4. It contains recommendation on how to implement it on real scenarios.

**Table 4:** General Scenarios in details for Performance

| General QA Scenario | Recommendation for Real Scenarios | Design Question |
|---|---|---|
| P1 – Tenant creation is fast, easy, and does not slow down entire system. | Should be defined how fast in second. | How are service packaged and deployed? |
| P2 – System is being hit by certain amount of tenant at measured time and does not decrease user experiences. | Should be defined how many tenant requests coming to server. | How to protect the Web server from request overload? |
| P3 – System is being hit by certain amount of concurrent request at measured time and does not decrease user experiences. | Should be defined how many concurrent requests coming to server. | Is there replication of the REST service at runtime? |

To make general scenarios applicable on evaluation, real scenarios is needed. It derived from general scenarios and contains expected result in a form of number or benchmark. Table 5 tells a story about the evaluation step and expected result.

**Table 5:** Real Scenarios

| Quality Attribute | Real Scenarios |
|---|---|
| Security (SE1) | A POST request is sent to login API in tenant 'A' with body contain of username and password for tenant 'A'. System identify correct username and password and return access token with expiration date. |
| Security (SE2) | A POST request is sent to login API in tenant 'A' with body contain of random username and password. System identify false username and password and return error message. |
| Security (SE3) | A POST request is sent to login API in tenant 'A' with body contain of username and password for tenant 'B'. System identify correct username and password but false tenant and return error message. |
| Security (SE4) | A GET request is sent to school list API in tenant 'A'. The request contains access token from tenant 'B'. System detect invalid token. |
| Scalability (SC1) | A GET request is sent to academic calendar API in tenant 'A'. Academic calendar API is protected resource and need authentication to access. The request must contain access token in JWT form to be properly authenticated and no session state stored in the server or database. |
| Scalability (SC2) | A GET request is sent to student profile API in tenant 'A'. Student profile image data come from another storage. System will not save user's data on the same storage with server. |
| Scalability (SC3) | Tenant created and configured through admin panel with no technical or server configuration needed. Once tenant created, the system automatically deploys tenant database, website (client-side logic) and tenant API endpoint (server-side logic). |
| Scalability (SC4) | School level API in tenant 'A' is being hit with certain amount of concurrent user in one minute. Amount of concurrent user increase in each iteration and stop until reach 250 concurrent users. Kubernetes Pod duplicate if the total Pod CPU usage reach 50%. If the total Cluster size is not enough for Pod to duplicate, Kubernetes Cluster scales out to increase computing power. |
| Performance (P1) | Tenant created and configured through admin panel with no technical or server configuration needed. New tenant will have school name, logo, website color scheme, and credentials for admin to login and insert school data. New tenant is created and deployed under 5 minutes. |
| Performance (P2) | 25 concurrent users accessed certain number of tenants randomly in one minute. The number of tenants increase in each repetition and stop until reach 30 tenants. System availability remain 98% and the response time average is under 2 seconds. |
| Performance (P3) | School level API is being hit with certain amount of concurrent user in one minute. Amount of concurrent user increase in each iteration and stop until reach 250 concurrent users. System availability remain 98% and the response time average is under 2 seconds. |
| Interoperability (I1) | Request with Request Header application/json accessing tenant API. System return value with Response Header application/json and list of school level in JSON. |
| Interoperability (I2) | Request using authentication bearer contain JWT token used for accessing protected source. System can identify false or expired token. |
| Testability (T1) | System give proper error response to let user know if something wrong. |

### 4.3 Architectural Analysis

Architectural analysis is the result of real scenario evaluation. Based on the real scenarios, ATAM analyze architectural analysis, risk, and tradeoff from multiple quality attributes. Table 6 show architectural analysis in Security (SE1). It evaluates architecture login procedure to get access token.

**Table 6:** Architectural Analysis on SE1

| | |
|---|---|
| **Scenario Summary** | A POST request is sent to login API in tenant 'Client' with body contain of username and password for tenant 'Client'. System identify correct username and password and return access token with expiration date. |
| **Business Goal(s)** | Make authentication system secure and stateless. |
| **Quality Attributes** | Security (**SE1**) |
| **Architectural Analysis** | System successfully return HTTP status 200 when username and password is correct. With, user token, and token expiration. Token expires in three days after login. Request also return user data, roles, and permission. |
| **Risk** | If the token expires, user should login again to the system. Token expiration should be defined correctly, not too long but also not too short depends on user's habit. |
| **Tradeoff** | Token expiration increase **security** because the token has a lifetime and cannot be used forever but it makes user should repeat login process if the token expires. |

ATAM will require us to explain future risk and tradeoff between quality attributes after architectural analysis result. This is important to determining which quality attributes is strengthen and which are weaken when architectural decision is made.

Sometimes one real scenarios in specific attributes can intersect another real scenario. SE2 scenario should return error response when using wrong username. It matches with T1 that require architecture to have proper response when something wrong. Table 7 explain further about real scenario in Security (SE2) and Testability (T1).

**Table 7:** Architectural Analysis on SE2 and T1

| | |
|---|---|
| **Scenario Summary** | A POST request is sent to login API in tenant 'Client' with body contain of random username and password. System identify false username and password and return error message. |
| **Business Goal(s)** | Make authentication system secure and stateless. |
| **Quality Attributes** | Security (**SE2**), Testability (**T1**) |
| **Architectural Analysis** | System return HTTP status 400 Bad Request when username and password is false. With error code 401 and message "invalid_credentials". |
| **Risk** | JWT just using one key, if the key is leaked and attacker can generate the token then the rest of the system is vulnerable. |
| **Tradeoff** | Implementing token authentication can make system **stateless** and increase **scalability**. But since there is no state between client and server, token became **the only security** system on the server. |

JWT is commonly used is REST based architecture. We can improve the security by adding two-factor authentication, so the system does not rely to JWT only. Architecture can detect

false username and password. But since this is a multitenant architecture, the authentication system must be isolated per tenant and have no correlation on each other. Table 8 shows architectural analysis in Security (SE3) and Testability (T1) to prove authentication is isolated.

**Table 8:** Architectural Analysis on SE3 and T1

| | |
|---|---|
| **Scenario Summary** | A POST request is sent to login API in tenant 'A' with body contain of username and password for tenant 'B'. System identify correct username and password but false tenant and return error message. |
| **Business Goal(s)** | Make authentication system isolated in each tenant. |
| **Quality Attributes** | Security (**SE3**), Testability (**T1**) |
| **Architectural Analysis** | System return HTTP status 400 Bad Request when username and password is false. With error code 4000 and message "errUnregisteredUser_HostMismatch". |
| **Risk** | User authentication are isolated between tenant. User belong to multiple tenants have multiple credentials. |
| **Tradeoff** | User belong to multiple tenants have multiple credentials because we cannot use same account to access another tenant. This make system **not flexible** but increase **security** because protected resources are isolated on each tenant. |

Authentication is proved isolated in every tenant. The authentication system is proven support multitenancy with certain tradeoff in system flexibility. Using perimeter security is one of the solution on addressing security challenges in cloud based architecture [26]. Table 9 evaluate Scalability (SC1) and Interoperability (I1) in authentication process.

**Table 9:** Architectural Analysis on SC1 and I1

| | |
|---|---|
| **Scenario Summary** | A GET request is sent to academic calendar API in tenant 'A'. Academic calendar API is protected resource and need authentication to access. Request contains JWT token from tenant 'A' in authentication bearer. Request also contain value of Request Header application/json. System authenticate the request and return value with Response Header application/json and list of school level in JSON. |
| **Business Goal(s)** | Make authentication system stateless. |
| **Quality Attributes** | Scalability (**SC1**), Interoperability (**I1**) |
| **Architectural Analysis** | System successfully return HTTP status 200 and return academic calendar data in JSON. |
| **Risk** | To access protected resource, every request should contain JWT token and increase request size. |
| **Tradeoff** | • Implementing token authentication can make system stateless and increase **scalability**. **Request size also increased** due to token addition in every request.<br>• Service only response to authorized token and increase **security**. But making **implementation/integration to other client more difficult**, especially client that using non-stateless system.<br>• The server request is standardized and improves **interoperability**. But become **less flexible** due to JSON return value. |

To make architecture support horizontal scaling in Kubernetes, token-based authentication is not enough. Database and media data should be push out from the server to another storage. Table 10 evaluate storage in architecture to support horizontal scaling based on real scenario in Scalability (SC2)..

**Table 10:** Architectural Analysis on SC2

| | |
|---|---|
| **Scenario Summary** | A GET request is sent to student profile API in tenant 'A'. Student profile image data come from another storage. System will not save user's data on the same storage with server. |
| **Business Goal(s)** | Make architecture system scalable. |
| **Quality Attributes** | Scalability (**SC2**) |
| **Architectural Analysis** | Student profile comes from amazon s3 storage. Server did not contain media data. Database and media are saved on the cloud storage. |
| **Risk** | Increase cost to rent cloud storage. The configuration is important to make sensitive data cannot be accessed by public. |
| **Tradeoff** | By separating media storage, it improves **scalability** and **performance** because server did not clone media data when do horizontal scaling (scale out). But it increases cost to rent cloud storage and it should be **configured** properly to avoid data breach. |

The architecture is proven scalable by implementing token-based authentication and external storage. There is no state on the server and theoretically can perform scale up and scale out. Since the architecture is multitenant, we must consider about tenant creation. To improve user experience, tenant creation, configuration, and management must be done through admin panel with no technical configuration on the server. Table 11 evaluate Scalability (SC3) and Performance (P1) on tenant creation.

**Table 11:** Architectural Analysis on SC3 and P1

| | |
|---|---|
| **Scenario Summary** | Once new tenant created, the system automatically deploys tenant database, website (client-side logic) and tenant API endpoint (server-side logic). New tenant will have school name, logo, website colour scheme, and credentials for admin to login and insert school data. New tenant is created and deployed under 5 minutes. |
| **Business Goal(s)** | Tenant deployment is easy and doesn't decrease user experience |
| **Quality Attributes** | Scalability (**SC2**), Performance (**P1**) |
| **Architectural Analysis** | - Tenant management, user management are managed through admin panel.<br>- Tenant logo, colour scheme, can be customized through admin panel.<br>- Tenant database, website and API endpoint deployed instantly in 15 seconds.<br>- Tenant deletion time is depending on tenant data. |
| **Risk** | Anyone has the valid access for system admin can deploy and delete tenant. |
| **Tradeoff** | Tenant can be created easily and speed up deployment speed. It increases **performance** and **scalability** rather than single tenant architecture. But since it does not need another configuration or two step verification anyone with access can create and delete tenant and reduce **security**. |

After the requirement to create horizontal scaling is fulfilled, it is time to evaluate the architecture scalability in Kubernetes. This scenario is mentioned in Scalability (SC4) and Performance (P3). To create this scenario, Kubernetes specification must be defined. In this scenario, Kubernetes running in Google Kubernetes Engine (GKE) located in region asia-southeast-1b.

Cluster contain of two nodes specification or called node pool. First node pool specification is 1 vCPU and 3.75 GB of RAM, second 1 vCPU and 0.6 GB of RAM. Both of node pool does autoscaling with minimum size of 3 VM and maximum 5 VM. Pod autoscaling is set to 50% CPU utilization with minimum Pod is 2 and maximum 50.

In this scenario, a number of concurrent users are trying to access School API every second during with one-minute period. Concurrency is a number of simultaneous connections. The number started from 25 concurrent users and increases in every iteration until reach 250 concurrent users. The reporting format is modeled after Lincoln Stein's torture-testing web servers [27].

Siege is being used to create stress testing with defined concurrent users to server. It accesses School Resource API with content type JSON. School API size is 52KB when returned. Number of delays in every request is between 0 and 1 second. This delay allows for the transactions to stagger rather than to allow them to pound the server in waves. Table 12 show the result of testing scenarios.

**Table 12:** Evaluation Result for Scalability Concern

| Concurrency (user/seconds) | Pod Replicas | Availability | Average Response Time (seconds) |
|---|---|---|---|
| 25 | 4 | 100% | 1.24 |
| 50 | 8 | 100% | 4.39 |
| 75 | 10 | 99.43% | 3.47 |
| 100 | 12 | 98.09% | 6.32 |
| 125 | 12 | 98.53% | 7.61 |
| 150 | 18 | 99.75% | 9.19 |
| 175 | 20 | 99.43% | 8.2 |
| 200 | 20 | 100% | 9.7 |
| 225 | 20 | 100% | 9.95 |
| 250 | 35 | 100% | 10.81 |

From the table above, all the number have tendency to increase linear with concurrent user. The application never crash or unable to access during one minutes of testing. The availability is 99.52% in average, but the average response time failed to reach the scenario expectation (7.08s). Success transaction return HTTP 200 OK implies that the response contains a payload that represents the status of the requested resource. Table 13 summarize evaluation in Scalability (SC4) and Performance (P3) in the architectural analysis.

**Table 13:** Architectural Analysis on SC4 and P3

| | |
|---|---|
| **Scenario Summary** | School level API in tenant 'A' is being hit with certain amount of concurrent user in one minute. Amount of concurrent user increase in each iteration and stop until reach 250 concurrent users. Kubernetes Pod will duplicate if the total Pod CPU usage reach 50%. If the total Cluster size is not enough for Pod to duplicate, Kubernetes Cluster will scale out to increase computing power. System availability remain 98% and the response time average is under 2 seconds. |
| **Business Goal(s)** | Prove the system reliable, scalable and always available to client. |
| **Quality Attributes** | Scalability (**SC3**), Performance (**P3**) |
| **Architectural Analysis** | The pod replicate as the number of concurrent users increases. When Pod cannot replicate because of node pool reach maximum capacity, node pool will increase capacity by adding more virtual machine to the cluster. System availability always above 99.52% but the average response time is 7.08 and failed to reach scenario expectation. |
| **Risk** | Increased latency when number of concurrent user increases. As the number of Pods increases, load balancer works harder to spread incoming connection to each Pod. |
| **Tradeoff** | Kubernetes makes application always available and increase **scalability**, and **availability**. Latency (response time) increase as the number of concurrent users increases and reduce **performance**. |

From this evaluation, we can see that the number of concurrent user responsible in increasing latency. The system is abused to serve many requests in every second and cause Pod to duplicate. Pod duplication process take time to finish. During that time, load balancer works very hard to distribute the request evenly. If the incoming request is too much for Pod capacity, the request should wait until new Pod is created and make latency/response time increases.

The previous evaluation is to test architecture capability in handling concurrent user. In the next scenario we evaluate architecture capability in handling multiple tenants at once. The real scenario is described in Performance (P2). 25 concurrent users are trying to hit different number of tenant URI every second during one-minute period.

The specification of Kubernetes Cluster is the same with scalability testing. Autoscaling set to minimum 10 Pods replica and maximum 50 Pods replica with CPU threshold set to 50%. Tenant URI is different in each iteration, but the resource remains the same which is School API. The number started from 3 tenants and increases in every iteration until reach 30 tenants. Table 14 show the result of testing scenarios.

**Table 14:** Evaluation Result for Multitenant Performance

| Number of Tenant | Total Transaction | Availability | Average Response Time (second) |
|---|---|---|---|
| 3 | 1238 | 100% | 0.94 |
| 6 | 1062 | 100% | 1.13 |
| 9 | 826 | 100% | 1.44 |
| 12 | 878 | 100% | 1.21 |
| 15 | 713 | 100% | 1.3 |
| 18 | 658 | 100% | 1.44 |
| 21 | 964 | 100% | 1.28 |
| 24 | 619 | 100% | 1.65 |
| 27 | 976 | 100% | 1.19 |
| 30 | 877 | 100% | 1.19 |
| **Average** | **881** | **100%** | **1.27** |

The availability is always 100% in meaning there is no failed request and always return HTTP 200 OK code. Longest transaction is increases linear with number of tenant increase means there still some request facing latency issue. Shortest transaction remains the same under 0.45 second which is very good.

Unlike the previous evaluation, response time is 1,27 seconds in average and really shows good performance. This happened because only the number of tenants increase not the number of concurrent users. Total transaction decrease as the tenant number increases. Kubernetes capabilities to handle transaction decrease as the number of tenant increases. Table 15 resume the evaluation in architectural analysis.

**Table 15:** Architectural Analysis on P2

| | |
|---|---|
| Scenario Summary | 25 concurrent users accessed certain number of tenants randomly in one minute. The number of tenants increase in each repetition and stop until reach 30 tenants. System availability remain 98% and the response time average is under 2 seconds. |
| Business Goal(s) | System capable of handling concurrent requests from different tenant. |
| Quality Attributes | Performance (**P2**) |
| Architectural Analysis | Architecture proven capable in handling multiple tenants at once. The system remains 100% available until reach 30 tenants. Response time is 1.27 second in average which is above expectation in 2 second. |
| Risk | Transaction speed decrease as the number of tenant increase. This is caused total transaction decreased as the number of tenant increase. |
| Tradeoff | System remain **responsive** and **available** as the number tenant increase but transaction speed **decrease**. |

Since it only 25 concurrent users, there is no problem in load balancer and Pod capacity to handle the load. That is why the system remain very responsive and 100% available. Request come into the system from different tenant URI. The core controller inside Pod then spread request into each tenant and make request to different database simultaneously. This is caused the transaction rate is decreasing since the application should spread the request into different database.

## 5. CONCLUSION

The main contribution of this research is to solve current educational technology problem based on case study by proposed new architecture design in SMIS. The architecture is created based on design pattern and REST constraints to meet REST standard. By providing multitenant architecture, SMIS is able to serve many educational companies from private sector to government area. With this new multitenant architecture, the application in educational technology are centralized and easy to integrate. This architecture also provides two levels of multitenancy to identify each school inside education company.

By providing scenario test, ATAM is able to identify system capabilities in multitenancy. Security aspect is proven to be stateless and isolated in every tenant (SE1, SE2, SC1, SC2). Tenant data is isolated and cannot be accessed by another tenant (SE3, SE4). The architecture is capable to scale up/down and scale out/in when traffic increases/decreases (SC4, P3). This architecture accessed by 250 concurrent user per seconds and the availability still above 99.52%. Tenant management proven to be easy with zero additional configuration on the server or database (SC3, P1). Increasing number tenant with 25 concurrent users per second still makes application responsive with average 1.27 second of response time (P2). This architecture gives clear error state and message when something wrong in the system (T1).

ATAM output in tradeoff and risk used to enhance architecture in the next iteration. This make the architecture improved overtime. From the evaluation result, we also realize that this architecture need improvement in the next iteration. Token is the only security system to access protected resource (SE2). Two factor authentications can be used in the next iteration, so the system did not only rely in username and password to get token access. The average response time in evaluation SC4 and P3 reach 7.08 second. To fix this latency issue, architecture must provide faster Pod duplication. This can be achieved by making Pod size as small as possible. Increase bandwidth size in the cloud configuration also helps to deal with increasing request. To keep the transaction rate stable when the number of tenant increases (P2), slave and master database replication can be used to enhance performance. All of this will be implemented and evaluated in the next ATAM iteration.

## REFERENCES


1. S. Srima, P. Wannapiroon, and P. Nilsook, "**Design of Total Quality Management Information System (TQMIS) for Model School on Best Practice**," *Procedia - Soc. Behav. Sci.*, vol. 174, pp. 2160–2165, 2015, doi: 10.1016/j.sbspro.2015.02.016.
2. P. Tan, H. Wu, P. Li, and H. Xu, "**Teaching Management System with Applications of RFID and IoT Technology**," *Educ. Sci.*, vol. 8, no. 1, p. 26, 2018, doi: 10.3390/educsci8010026.
3. C. Z. Zulkifli, H. N. Hassan, A. A. Zalay, S. M. Kamis, and N. H. A. Hassan, "**Special Issue INTEGRATED RFID TECHNOLOGY AND WIRELESS MESH NETWORK**," 2018.
4. I. Graham, ***Requirements modelling and specification for service oriented architecture***. John Wiley & Sons, 2008.
5. M. Halim, N. Adadi, D. Chenouni, and M. Berrada, "**Web services composition in E-Learning platform**," *Int. J. Emerg. Trends Eng. Res.*, vol. 8, no. 2, pp. 525–532, 2020, doi: 10.30534/ijeter/2020/41822020.
6. M. Rizky, A. Nurul Fajar, and A. Retnowardhani, "**Microservices Architecture Design: Proposed for online HealthCare**," *Int. J. Emerg. Trends Eng. Res.*, vol. 8, no. 1, pp. 8–11, 2020, doi: https://doi.org/10.30534/ijeter/2020/14842020.
7. R. T. Fielding, "**Architectural Styles and the Design of**

**Network-based Software Architectures**," *Building*, vol. 54, p. 162, 2000, doi: 10.1.1.91.2433.

8. R. T. Fielding *et al.*, "**Reflections on the REST architectural style and 'principled design of the modern web architecture' (impact paper award)**," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. pp. 4–14, 2017, doi: 10.1145/3106237.3121282.
9. T. Erl, ***SOA Design Patterns***, 1st ed. Prentice Hall PTR, 2009.
10. M. Barbacci, P. Clements, A. J. Lattanze, L. M. Northrop, and W. Wood, "**Using the Architecture Tradeoff Analysis Method ( ATAM ) to Evaluate the Software Architecture for a Product Line of Avionics Systems : A Case Study**," *Tech. Note - C.*, no. July, p. 31, 2003.
11. B. Costa, P. F. Pires, F. C. Delicato, and P. Merson, "**Evaluating a Representational State Transfer (REST) architecture: What is the impact of REST in my architecture?**," *Proc. - Work. IEEE/IFIP Conf. Softw. Archit. 2014, WICSA 2014*, pp. 105–114, 2014, doi: 10.1109/WICSA.2014.29.
12. B. Costa, P. F. Pires, F. C. Delicato, and P. Merson, "**Evaluating REST architectures - Approach, tooling and guidelines**," *J. Syst. Softw.*, vol. 112, pp. 156–180, 2016, doi: 10.1016/j.jss.2015.09.039.
13. A. Ejsmont, "**Programming & Web Dev-Web scalability for startup engineers : tips & techniques for scaling your Web application**." 2015.
14. M. W. Khan and E. Abbasi, "**Differentiating Parameters for Selecting Simple Object Access Protocol ( SOAP ) vs . Representational State Transfer ( REST ) Based Architecture**," *J. Adv. Comput. Networks*, vol. 3, no. 1, 2015, doi: 10.7763/JACN.2015.V3.143.
15. M. Garriga, C. Mateos, A. Flores, A. Cechich, and A. Zunino, "**RESTful service composition at a glance: A survey**," *J. Netw. Comput. Appl.*, vol. 60, pp. 32–53, 2016, doi: 10.1016/j.jnca.2015.11.020.
16. T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian, ***SOA with REST: Principles, Patterns &Constraints for Building Enterprise Solutions with REST***, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2012.
17. I. Gorton, "**Essential software architecture**," *Essent. Softw. Archit.*, pp. 1–283, 2006, doi: 10.1007/3-540-28714-0.
18. L. Bass, M. Klein, G. Moreno, and S. E. Institute, "**Applicability of General Scenarios to the Architecture Tradeoff Analysis Method**," *Carnegie Mellon University Technical Report*, no. CMU/SEI-2001-TR-014, ESC-TR-2001-014. 2001.
19. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "**The architecture tradeoff analysis method**," *Proceedings. Fourth IEEE Int. Conf. Eng. Complex Comput. Syst. (Cat. No.98EX193)*, pp. 68–78, 1998, doi: 10.1109/ICECCS.1998.706657.
20. M. Jones, J. Bradley, and N. Sakimura, "**Json web token (jwt)**," 2015.
21. J. Kabbedijk, C. P. Bezemer, S. Jansen, and A. Zaidman, "**Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective**," *J. Syst. Softw.*, vol. 100, pp. 139–148, 2015, doi: 10.1016/j.jss.2014.10.034.
22. E. J. Domingo, J. T. Niño, A. L. Lemos, M. L. Lemos, R. C. Palacios, and J. M. G. Berbís, "**CLOUDIO: A Cloud Computing-oriented Multi-Tenant Architecture for Business Information Systems**," *Proc. - 2010 IEEE 3rd Int. Conf. Cloud Comput. CLOUD 2010*, pp. 532–533, 2010, doi: 10.1109/CLOUD.2010.88.
23. A. Furda, C. Fidge, A. Barros, and O. Zimmermann, ***Chapter 13 - Reengineering Data-Centric Information Systems for the Cloud – A Method and Architectural Patterns Promoting Multitenancy***, 1st ed. Elsevier Inc., 2017.
24. W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "**An Updated Performance Comparison of Virtual Machines and Linux Containers**," pp. 171–172, 2015.
25. V. Medel, R. Tolosana-Calasanz, J. Á. Bañares, U. Arronategui, and O. F. Rana, "**Characterising resource management performance in Kubernetes**," *Comput. Electr. Eng.*, vol. 68, no. May 2017, pp. 286–297, 2018, doi: 10.1016/j.compeleceng.2018.03.041.
26. S. Sharaf, "**Security Issues in Serverless Computing Architecture**," *Int. J. Emerg. Trends Eng. Res.*, vol. 8, no. 1, pp. 8–11, 2020, doi: https://doi.org/10.30534/ijeter/2020/43822020.
27. Lincoln D. Stein, "**Web Techniques: Torture-Testing Web Servers**," 1999. [Online]. Available: https://people.apache.org/~jim/NewArchitect/webtech/1999/07/stein/. [Accessed: 11-Nov-2018].