Siyuan Xue

Designing and implementing Web API for RAI software

Master's Thesis
in Information Technology
November 15, 2019

University of Jyväskylä

Faculty of Information Technology

Kokkola University Consortium Chydenius

Author: Siyuan Xue
Contact information: siyuan.xue@outlook.fi
Phonenumber: 045-356 0601
Title: Designing and implementing Web API for RAI software
Työn nimi: Design and implement Web API for RAI software
Project: Master's Thesis in Information Technology
Page count: 52+5
Abstract: The information silo becomes a significant problem with the software evolution from desktop applications to web-based applications. In this thesis, the research problem is derived from a customer's requirement to integrate two different health information systems. The design and creation research approach is employed in this study, which involves aware, suggestion, development, evaluation and conclusion. This paper focuses on building Web Service to break the information silo in healthcare systems, especially between the Electronic Health Record (EHR) and other health information systems. As the outcome of this study, a RESTful Web API is constructed to resolve the information silo issue. Meanwhile, the different alternative solutions to construct each component of the Web API and open issues are summarized in the evaluation phase.
Suomenkielinen tiivistelmä: NA
Keywords: Web Service, Web API, REST, resident assessment instrument, maturity model, service description
Avainsanat: NA

Glossary

AD	Active Directory
API	Application Programming Interface
CoHA	Classification of HTTP-based APIs
CORS	Cross-Origin Resource Sharing
CRUD	Create Read Update Delete
DL	Description Language
DTD	Document Type Definition
EHR	Electronic Health Record
FR	Functional Requirement
GUI	Grapic User Interface
HATEOAS	Hypermedia As The Engine Of Application State
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDE	Intelligent Development Editor
IDP	Identity Provider
JSON	JavaScript Object Notation
NFR	Non-Functional Requirement
OAI	Open API Initiative
OS	Operating System
OSI	Open Systems Interconnection
PHP	Hypertext Preprocessor
PID	Personal Identifier
POX	Plain Old XML
RAI	Resident Assessment Instrument
RAML	RESTful API Modeling Language
RDF	Resource Description Framework

REST	Representational State Transfer
RMM	Richardson Maturity Model
RP	Relying Party
RPC	Remote Procedure Call
RSS	Rich Site Summary
SDK	Software Development Kits
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UC	Use Case
UI	User Interface
URL	Uniform Resource Location
WADL	Web Application Description Language
WSDL	Web Service Description Language
WWW	World Wide Web
XSD	XML Schema Definition
YAML	Yet Ain't Markup Language

Contents

1 Introduction

With the progress of the Internet, software previously developed as desktop applications is now provided as web-based applications. This software is becoming more dependent on the Internet, and most application data is stored in the cloud rather than on a local computer. To access the stored data, the user must use a specific software. Therefore, software systems easily become information silos, which are systems that manage data and do not share it with other systems [29]. However, the data exchange between different systems brings many benefits, such as increasing data reusability and accuracy.

The information silo problem is a significant issue among healthcare systems, as the healthcare service field uses diverse information systems to gather and manage different health data. The most common is the Electronic Health Record (EHR) system, which is used in healthcare organizations to manage patients' health information. Besides the EHR, there are also other health information systems used within a healthcare organization (e.g., a nursing home or social care) to access, manage, and share health information.

The information managed by one health information system should be made accessible for the other systems, but in reality, this is rarely the case. The different health information systems can be from different manufacturers, which makes it challenging to share and utilize patient data among these distinct systems. Therefore, how to integrate disparate systems and bridge the gaps between them is an important question.

In this study, using a Web Service is considered a way to integrate different healthcare systems. A Web Service is a software system that enables machine-to-machine interaction by providing a machine-processable service description and a set of standards related to network transportation and data serialization [2].

The research problem addressed in this thesis is derived from a customer's need to integrate two different health information systems. Currently, the user must manually copy and paste to transfer data between the systems, which introduces a number of issues. First, the process creates extra workload (i.e., a lot of repetitive and meaningless work). Second, it increases the risk of introducing errors, especially as

the data can be life critical.

This thesis aims to solve the issue by adopting a design and creation research approach [38]. This approach involves five phases: awareness, suggestion, development, evaluation, and conclusion. In the awareness phase, the research motivation and problem are analyzed. The background research and literature review are carried out in the suggestion phase to offer supportive ideas for solving the research problem. These ideas are implemented in the development phase, a step that demonstrates the process of how an idea becomes an artifact that solves a problem. In the evaluation phase, the developed artifact is critically assessed. In the conclusion phase, the results from the design process and created artifact are summarized.

The expected result of this work is an artifact in the form of a Web Service that addresses the integration issue. The remainder of the thesis is organized as follows: Chapter 2 includes background information. Chapter 3 presents the development process, system requirements, and system design. Chapter 4 provides an evaluation of the developed artifact. Chapter 5 provides a summary and discusses open issues and future work.

2 Background

2.1 Application programming interface

Application Programming Interface (API) is an overloaded concept, which refers to different terms depending on the context. An API is a set of rules regarding the interaction between different software, such as a desktop application, an Operating System (OS), and a web application. For example, if a Linux desktop application needs to save a file, the desktop application must invoke the Linux Filesystems API to perform the saving operation.

When an API is implemented as a Web Service, it is called a Web API and refers to a set of rules for interactions between the software service provider and its consumer. Most popular web applications provide a Web API for programmers to retrieve data, utilize the web application's computing capability, and construct third-party applications. For instance, the Google Maps platform publishes Web APIs to third-party software vendors that wish to utilize the Google Map service. The user can take advantage of the Google Map service to meet massive calculation demands with the Google Cloud's computing capability.

There are different ways of implementing a Web API. Robert Daigneau [9] describes the most common styles, which include: The Remote Procedure Call (RPC) API, the Message API, and the Resource API. These are discussed in detail below.

2.1.1 Remote Procedure Call API

A Remote Procedure Call (RPC) is a mechanism by which an application invokes a service provided by another application [27]. The applications are isolated and can even be located in different machines. In the RPC process, the request is a message in RPC message format, which includes the RPC program number, program version number, and procedure number; an application sends one or more messages to invoke the remote service.

RPC is the core concept of the RPC API. In an RPC API, the invoking process starts with a client sending a request to a remote server. The client process is blocked until the response is received. When the server receives the request, it extracts the

procedure number and the corresponding parameters and dispatches the information to the correct process. The server returns the response to the client once the invoked process is complete.

An RPC API provides a narrow view of the data [34]. The business logic is implemented in the RPC API server side and published as a single endpoint to service all the RPC requests. When the API consumer must perform a specific operation, he/she only has access to specific data fields. Therefore, the bandwidth used by the protocol is reduced. Suitable for publishing a single endpoint, RPC API requires the API consumer to be tightly coupled with the API server, which limits the evolution of both the RPC API and its consumer.

2.1.2 Message API

A Message API is invoked by receiving a self-descriptive message via a Hypertext Transfer Protocol (HTTP) at the designated Uniform Resource Indicator (URI). The message includes two parts: a header and body. The header is optionally used to indicate the meta information of the request, such as the authentication credentials and the request state information, while the message body contains the primary data, such as the procedure to execute and the arguments for it.

In a Message API, the message format is diverse. The most typical are XML and the standardized XML, the Simple Object Access Protocol (SOAP). Besides those two options, there are some proprietary formats in use, usually called Plain Old XML (POX).

The Message API request sequence is similar to that of the RPC API. When a client consumes the Message API, it sends a request containing the message to the server. The server will process it to determine which procedure should handle the request. Once the process is complete, the server returns a result to the client. The message is the key concept of the Message API, which means the Web API's design is equivalent to the message's definition. The Web APIs perform as endpoints to receive the request, parse the message, and forward it to the procedure.

The request message includes its type of information and content. There are mainly three message types, Command Message, Event Message, and Document Message. The Command Message is for the API to complete a specific task; the Event Message is dedicated to describe the triggered event, and the Document Message is a document to an entity record. The response is also in the message. It contains the processing result; the request acknowledges or the failure if the process

fails.

A Message API usually provides a service description for the Message API consumer to generate the client-side code with tools. The most commonly used is the Web Service Description Language (WSDL). Message API also enables the Request/Acknowledge interaction pattern rather than the Request/Response pattern. In the Request/Response pattern, once the server receives a request from the client, it starts to process the request, and the client process will be blocked until a response is returned to the client. However, in the Request/Acknowledge interaction pattern, the server will forward the request to an asynchronous background process and return an acknowledgment to the client to notice the client that the request is accepted and in processing. The server only takes responsibility to dispatch the request and return the acknowledge. Once the acknowledge is returned, the server will be available to serve the client. This approach is considered to achieve asynchrony. With this approach, the Message API consumer can avoid being blocked by the server.

2.1.3 Resource API

A Resource API is a type of API that utilizes the URI to identify the request destination, the HTTP server method to represent the process action, and the media type to determine the content type. As the name implies, a Resource API enables the channel to manipulate the resources. Most, but not all, Resource APIs conform to the Representational State Transfer (REST) style. The resource and REST concept are discussed further in Section 2.2.1 and Section 2.2.

2.2 Representational State Transfer

The Representational State Transfer concept originally appeared in Roy Fielding's Ph.D. dissertation in 2000 [15]. Fielding aimed to present a network-based communication architecture concept for constructing high-quality application architectures. REST refers to a set of architectural constraints and principles. If an architecture conforms to the RESTful constraints and principles, it can be defined as a RESTful architecture. REST is not a new technology, component, or service. The idea behind it is to optimize the usage of the web's existing features and capabilities and the guidelines and constraints in the current web standards. Although web technologies profoundly impact REST, the REST architectural style is not bound to HTTP.

However, HTTP is currently the only REST-related instance.

2.2.1 Resource

A resource in REST is a key abstraction of information. Any information in a RESTful architecture, such as a file, an image, a service, or an object (e.g., a user account), can be abstracted as a resource. Therefore, when the author of a RESTful architecture designs a hypertext reference, the design must fit within the resource definition [15]. A resource maps a group of entities regardless of the entity in any particular moment; meanwhile, an entity in different moments can be recognized as distinct resources, even though the entity value is the same. The source code version control in software engineering is taken as an example to illustrate the resource mapping. The source code file is a resource, and the file has multiple versions (e.g., "latest version," "revision 1.0.0," or "production version"), such that each version is considered a resource.

2.2.2 Representation

Representation is the description of a resource's state, with a sequence of bytes and metadata for describing the bytes. A representation is transferred between the REST components, such as a web server and browser, to perform actions on a resource to acquire the current or intended state. A representation can be in a different data format, known as a media type (see Section 2.4). Since different media types are feasible for different use cases, some media types can be processed directly by the representation recipient; some require the recipient to render. For example, if a web server responds to a browser request in a Hypertext Markup Language (HTML) format, the browser must render the HTML response into a Graphic User Interface (GUI). If the web server responds in a JSON format, the browser can use the data directly with the JavaScript language.

REST is an architectural style that is derived from web architecture by applying the following list of constraints [26].

1. Client-Server: This constraint categorizes the user interface and data storage into the client and server modes to increase Web applications' cross-platform capabilities, simplify server components, and increase application extensibility.

2. Stateless: This constraint requires that every request sent by the client to the server must contain all the information necessary for the request, and the request's status information is stored in the client side to avoid using any information on the server. Thus, the client and server remain relatively independent.

3. Cache: This constraint specifies that the server responding with data must implicitly or explicitly state whether the response is cacheable or noncacheable to improve the efficiency of the response data usability in the client side.

4. Uniform interface: This constraint requires that each web component operation be performed through a unified interface to simplify the web architecture and clarify the request purpose. The uniform interface is an essential REST feature, which distinguishes it from other web architectures.

5. Layered system: This constraint divides the Web architecture into several levels of functional layers and requires that each component only interacts with the adjacent layer to make the Web application's structure clearer and more purposeful.

6. On-demand code: This constraint means that the client can extend its functionality in the form of downloading and running server code to enhance its flexibility.

2.2.3 REST API

As depicted in Figure 2.1, Web APIs perform as Web Service endpoints. A client will interact with a Web Service via the Web APIs. As a web architectural style, the REST is widely used in Web API design. If a Web API conforms to the REST architectural style, it can be defined as a REST API. There are many well-known public RESTful Web API services, such as Google APIs. A well-constructed RESTful Web API will attract third-party application programmers to utilize the Web Service.

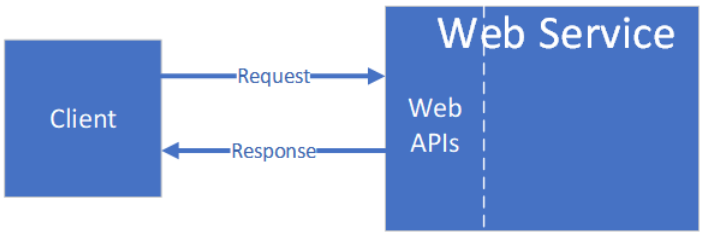

Figure 2.1: Web API and Web Service relationship

2.3 Communication protocols

2.3.1 HyperText Transfer Protocol (HTTP)

HTTP is an object-oriented communication protocol that belongs to the application layer in the Open Systems Interconnection (OSI) model, and it is suitable for distributed hypermedia information systems due to its speed and straightforwardness. For example, it defines how the browser requests the web document from the web server and how the web server responds to the browser request. In a hierarchical respect, HTTP is a transaction-oriented application-layer protocol, which is an essential basis for the reliable exchange of files (including text, sound, images, and other multimedia files) on the World Wide web (WWW). It establishes the rules for communication between the browser and the web server [14]. The basic HTTP features:

- Fast and straightforward indicates that when a client sends requests to a server, it only needs to transfer the request method and path. The most commonly used request methods are GET, HEAD, and POST, which indicate the request's purpose. Since the HTTP procedure is simple, the HTTP server's program size is small and fast.

- Connectionless tells the server to limit the processing to only one request per connection. The server disconnects after processing the client's request and receiving a response from the client. The client can save transmission time with this connectionless protocol.

- Stateless denotes that the protocol has no memory for transaction processing. The stateless feature indicates that if the subsequent processing requires prior information, it must be retransmitted, which may result in increasing

the amount of data transferred per connection. Meanwhile, it responds faster when the server does not need prior information.

2.3.2 HTTP methods

There are nine HTTP request methods, known as "request actions," defined in the HTTP/1.1 protocol. The different methods specify the resource mode specified by different operations. The server will also respond differently according to the disparate request methods [14].

- GET

  The GET request aims to retrieve the resource specified by the request. In general, the GET method should only be used for data reading instead of for nonidempotent operations , which may lead to side effects. The GET method requests the specified page information and returns the response body. This method is considered insecure because it is arbitrarily accessed by the web crawler.

- HEAD

  The HEAD request is similar to the GET method and is a request to issue a specified resource to the server. However, the server does not return the resource's content portion in response to the HEAD request. With this method, the client can obtain the server's response header information without transmitting the entire content. The HEAD method is often used by the client to view the server's performance.

- POST

  The POST request submits data, such as a form data submission or a file upload, to the specified resource and requests that the server process it. The request data comprises the request body. The POST method is a nonidempotent method because this request may create new resources and/or modify existing resources.

- PUT

  The PUT request uploads its latest content to the specified resource location. With this idempotent method, the client can transmit the specified resource's latest data to the server to replace the specified resource's content.

- DELETE

  The DELETE request is used to request the server to delete the resource identified by the requested URI. In this idempotent method, a DELETE request is used when the client intends to delete or archive the target resource.

- CONNECT

  CONNECT is a reserved keyword in the HTTP/1.1 protocol, and it is primarily assigned to create a proxy server in pipe mode. For example, CONNECT is used to establish access to the SSL encryption servers, which communicate with an unencrypted HTTP proxy server.

- OPTIONS

  The OPTIONS request is similar to the HEAD request and is for fetching client-side viewing of server performance. This method will request the server to return all HTTP request methods supported by the resource. It will replace the resource name with '*' and send an OPTIONS request to the server to test whether the server function is normal. When an "XMLHttpRequest" JavaScript object performs Cross-Origin Resource Sharing (CORS), it uses the OPTIONS method to send a sniff request to determine whether there is access to the specified resource.

- TRACE

  TRACE requests the server to echo the request information it receives. This method is mainly used for testing or diagnosing HTTP requests.

- PATCH

  The PATCH request is similar to the PUT request and performs the resource updates. Different from the PUT method, the PATCH method commits to updating part of the resource.

2.3.3 Simple Object Access Protocol (SOAP)

SOAP is a communication protocol that defines how applications share messages [37]. The SOAP specification is built upon the XML and involves the rules to represent the data in XML format. The main body of the SOAP specification wraps around the XML content, so it inherits the XML standards, such the XML Schema and XML Namespaces.

XML messaging is a method for applications to exchange information, but it lacks consensus when two applications communicate with each other. Consequently, SOAP performs as a collaboration specification to make the communication message format uniform. A SOAP message is a "letter" in an envelope, and "letter" contains one or two parts: an optional header and body. The header is for storing the metadata describing how the message should be delivered, such as authentication or authorization assertions and routing settings. The body contains the message content formatted into a valid XML. Besides the standard SOAP message, SOAP also provides a particular type of message, a fault message, which is used to describe the errors encountered during the communication process. It contains the fault code, string, actor, and details. The fault code should be found in its namespace. The fault string should occur in a human-readable expression. The fault actor states who processes the message to trigger the error, and the fault details show the application-specific error; furthermore, the error must relate to the message body.

SOAP is suitable for any transport protocol because it is a standardized packaging protocol. The most typical transport protocol is HTTP. When the HTTP is used to convey a SOAP message, the pattern naturally matches the SOAP RPC process. The SOAP request message is posted to the server with the POST method until the SOAP response returned.

2.4 Media types

Media types identify the data format used in the HTTP request and response; the HTTP request and response's Content-Type header references the media types. The media type follows the pattern

```
type "/" subtype *( ";" parameter )
```

The type in the pattern describes the primary media type, such as application, image, or text; the subtype is the subordinate of the primary media type, such as XML and JSON. The parameter is the alternative to supplying customized arguments. In the section below, two subtypes, XML (see Section 2.4.1) and JSON (see Section 2.4.2), are listed and elaborated.

2.4.1 XML

XML is a markup language that can be used to create a markup. It was created by W3C to overcome the limitations of HTML, which is the basis of all web pages. XML is a standard text format for representing structured information on the web, without complex syntax and all-encompassing data definitions. Derived from the Standard Generalized Markup Language (SGML) XML adapts to many purposes, for example, data exchange, constructing Web Services, and inventing new Internet languages. In the data transfer process, XML always retains data structures such as parent-child relationships. Different applications may share and parse an XML file to avoid the traditional string parsing or disassembly process. In Web Services, the transferred data is formatted into XML to enable the protocols to be standardized; the most representative is SOAP (see Section 2.3.3). Many new Internet languages are invented based on XML, such as WSDL, Rich Site Summary (RSS), and Resource Description Framework (RDF) [6].

When creating an XML document, a corresponding XML constraint document is preferred to regulate the XML format. The widely used constraint techniques are the Document Type Definition (DTD) and Schema.

The DTD acts as a template for one or more XML files. A valid XML file should conform to its DTD, regarding the elements and their attributes, arrangement/order, and available contents in the XML file [5]. The elements and attributes are created according to the application's requirements. Due to the different industries' characteristics, it is challenging to create a DTD with high integrity and adaptability. Therefore, a DTD is usually defined by a particular application area, such as medicine, construction, industry, commerce, and administration. The more extensive the range of elements defined by a DTD, the more complicated it is.

Similar to the DTD, the XML Schema Definition (XSD) is another XML definition language used to describe the structure of an XML document. The XML Schema is based on the XML language and is also an XML application. Although the DTD, which provides specifications for XML documents, solves the problem of XML document standardization, the file format type and XML file format type are still inconsistent. Meanwhile, the data types in DTD cannot always fulfill the industry's demands; therefore, the XML Schema was introduced. The Schema pattern describes the structure and data types of elements and attributes, the sequence of elements, the scope ranges, enumeration, and pattern matching [5].

2.4.2 JavaScript Object Notation (JSON)

JSON is a lightweight data exchange format. It is a subset of ECMAScript that stores and represents data in a text format that is completely independent of the programming language [30]. JSON become to be a competitive data exchange language because of its simplicity and clear. JSON is not only easy to read and write, it is also easy for machine parsing and generation, and it effectively prompts the network transmission efficiency. The structure of JSON is mainly composed of key-value pairs, in which structs or arrays can be used to organize key-value pairs. In addition, JSON supports nesting, allowing data nesting in a JSON format.

As a data format, JSON utilizes a documentation definition called the JSON Schema to define the format. Similar to the XML Schema, the JSON Schema was written under IETF draft in 2011 and was developed to Draft-07 version. Now, it is based on the JSON format for defining JSON data structures and validating JSON data content [30][17].

2.5 Versioning strategy

During a Web API's development, the Web API's stability is important to the client, as every slight change could impact the Web API consumer. However, as no one can predict the future, the system is inevitably necessary to add or modify existing resources, which will cause system upgrading. That means once the Web API service is publicly published to consumers, every revision of the Web API should consider the impacts to the Web API consumers. Thus, a version control strategy should be selected and applied [12]. A Web API version control strategy is like a long-term agreement between the Web API provider and consumer. The strategy will directly determine whether the consumer uses the Web API, or whether the consumer will abandon the Web API after any upgrading.

2.5.1 Version control models

The common Web API version control models are "The Knot," "Point-to-Point," and "Compatible Versioning [24]."

- The Knot: There is no versioning. Only one version is available online. All the users must use the latest API version. Any API modification will affect all the users and even the entire ecosystem.

- Point-to-Point: Every Web API revision is marked with a version number, and each version is standalone. The Web API consumer application must migrate to the corresponding Web API version, which contains the required new features.

- Compatible Versioning: It is similar to The Knot; there is a single version running on the service, but the later version of the Web API should be compatible with the earlier version.

A typical compatible versioning strategy, semantic versioning is demonstrated in the following section. Semantic versioning enables the versions' backward compatibility.

Semantic versioning is based on conventions that have been widely used by various closed and open-source software. For this theory to work, there must be a well-defined public API, which can be achieved through file definitions or code enforcement requirements. In any case, the clarity of this API is very important. Once the API is defined, the changes can be represented by modifying the corresponding version number. The version number pattern is MAJOR.MINOR.PATCH. The MAJOR number indicates the public API's stable version. When any change is released and not backward compatible, the MAJOR number is incremented. The MINOR number is used to mark when the public API releases new features, and the PATCH number increment represents bug fixes. All the version numbers should be nonnegative integers without leading zeros. A suffix such as -alpha or -belt may append the version number to denote the version is in some process. For instance, *1.0.1-alpha version* indicates that the first stable version of the public API with the first bug fixes is in the Web API development team's internal testing process [31].

2.5.2 Versioning method

The versioning method intends to where should the version number be placed. According to Troy Hunt [23], there are two mainstream versioning methods, version number in URI and Media Type.

- Version number in URI
  This method refers to the API version information denoted in the service Uniform Resource Location (URL), either in the path (/v1/user/1) or as a query parameter (/user/1?version=1). The benefit of this method is that the version

number is visualized in the request URL. The disadvantage is that it violates the principle of RESTful architecture. In theory, a URL should correspond to a specific resource in the server. Adding a version number will confuse the resource's concept, make the whole architecture confusing, and increase the cost of future maintenance.

- Version number in Media Type
  The media type in an HTTP request header is used to denote the API version information to announce to the API server which resource is requested. Similarly, if no media type is set, the latest version of the API will respond as a default.

2.6 Security

2.6.1 Hypertext Transfer Protocol Secure (HTTPS)

The Hypertext Transfer Protocol Secure (HTTPS) is a network security transmission protocol that is the combination of the HTTP and Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols. The primary purpose of HTTPS development is to provide identity authentication for network servers and protect the data packet privacy and integrity in network communications. In short, HTTPS uses the SSL or TLS technology to encrypt data during the HTTP protocol to ensure data security [8].

As Section 2.3.1 indicates, HTTP is only a network communication protocol not a secure protocol. Since HTTP does not support self-encryption, it is impossible to encrypt a data packet in network communications. That is, the data packets are sent in plain text, which means there is no way to guarantee data privacy. When data transfers use HTTP protocol, any user between the requester and responder can tamper with a request since there is no mechanism to verify the data packet integrity. Since the HTTP protocol cannot prove the data package integrity, there is no way to know whether the request or response content has been tampered with during the data packet transmission [22].

HTTPS was introduced to prevent security risks. An example of a client and server communication explains how HTTPS works: Before a client sends a request to a server using HTTPS, an SSL connection should be established between them. The SSL connection aids the client in verifying the server's identity and for both the

client and the server to reach a common understanding of which encryption algorithm is in use in their communication and what is the key to encrypt and decrypt the request data. Once the SSL connection is established, the client and server can encrypt and decrypt the request and response data with the key.

2.6.2 Authentication and authorization

Every request sent to the Web API must contain the information to validate the request. The validation involves two processes: one is authentication, which involves identifying the request sender to decide whether the Web API should accept and process the request; the second is authorization, which refers to whether the request sender has the privilege to use the target resource. There are several open specifications available for authentication and authorization.

OpenID

OpenID is an open authentication protocol supported by many websites, such as Google, Microsoft, and PayPal. To use OpenID, the user must first obtain an OpenID account, such as a Google account, on the OpenID Identity Provider (IDP). The user can use the OpenID account to log in to any Relying Party (RP) that accepts OpenID authentication. The OpenID protocol provides a framework for communication between the IDP and the RP. The latest version of OpenID is the third generation, named OpenID Connect, which is an interoperability authentication protocol based on the OAuth 2.0 specifications. It is implemented with a JSON data format, and it is easier for developers to integrate than the previous authentication protocol. OpenID Connect allows the developer to authenticate users across websites and applications without having to own and manage the users' credentials. OpenID Connect allows all types of clients, including browser-based JavaScript and native mobile applications, to initiate login traffic and receive verifiable assertions about logged-in users [33].

OAuth 2.0

OAuth 2.0 is an open-standard authorization protocol that is widely used in public APIs, such as Facebook and Google APIs. According to the OAuth 2.0 protocol specifications, an end-user can grant a third-party application access to protected resources, such as photos, videos, and contact lists that are stored on a server, without exposing the end-user's credentials to the third-party application. OAuth 2.0 allows

an end-user to provide a token to the third-party application instead of their user name and password. Each token authorizes the third-party application to access a certain group of resources within a limited time period. When the token expires, the third-party application must either obtain a new token or refresh using a refresh token. The OAuth 2.0 protocol defines four roles: client, resource owner, authorization server, and resource server (see Table 2.1) [20]. Figure 2.2 shows the OAuth 2.0

Table 2.1: OAuth 2.0 roles [20].

Role	Description
Resource owner	An entity who can grant access to a protected resource (e.g., a person as an end-user of a third-party application).
Resource server	The server stores the protected resources, validates access tokens, and responds with the requested resource.
Client	An application delegated by a resource owner to request a resource from a resource server (e.g., a third-party application).
Authorization server	The server takes the responsibility of issuing an access token after verifying the resource owner's identity and authorization.

abstract protocol flow, which describes the interactions among the OAuth 2.0 roles; the flow is listed below.

- (A) The client requests the resource owner to grant authorization. The resource owner can provide its credentials to the client directly or to the authorization server as an intermediary.
- (B) The client receives the credentials from the resource owner.
- (C) The client forwards the credentials to the authorization server.
- (D) The authorization server authenticates the client and the credentials from the previous steps and issues an access token, if valid.

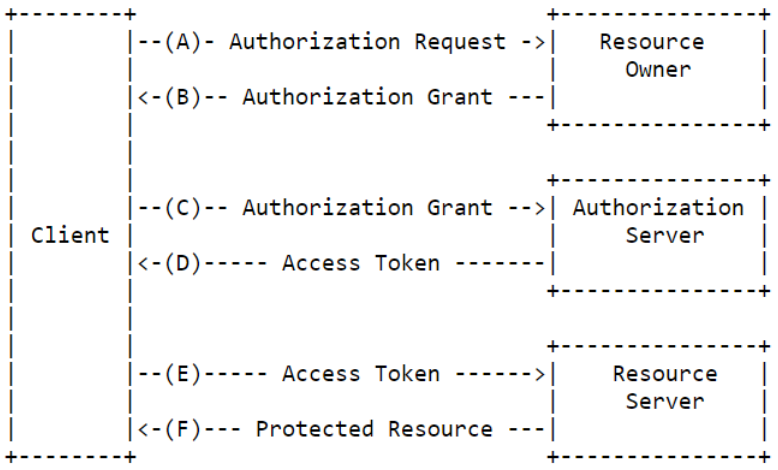

Figure 2.2: OAuth 2.0 abstract protocol flow [20]

- (E) The client requests the protected resources from the resource server using the access token.
- (F) The resource server responds with the protected resource if the access token is valid.

The credentials mentioned in the flow refer to how the client represents the resource owner's authorization to get an access token from the authorization server. This process is defined in OAuth 2.0 as an authorization grant. There are four grant types specified in [20]: authorization code, implicit, resource owner password, and client credentials.

In this thesis, the resource owner password grant type is focused on and explained. When a client uses a resource owner password as the grant type, the resource owner provides its user name and password to the client, who can pass directly, and they can pass directly to the authorization server to obtain an access token. The resource owner password should be assigned to a highly trusted client, since leaking the resource owner's credentials is quite a high risk if the client secretly records them. Even though the client is highly trusted by the resource owner, a long access token lifetime or a refresh token should be considered to reduce the risk.

Figure 2.3 demonstrates the resource owner password credentials flow in three steps:

- (A) The resource owner sends the username and password to the client.

- (B) The client requests an access token from the authorization server by presenting the resource owner's credentials.
- (C) The authorization server responds with an access token if the resource owner's credentials are valid.

When the client performs step B, a list of parameters is required in the request.

- username: the resource owner's credential
- password: the resource owner's credential
- grant_type: this should be "password" when using a resource owner password as an authorization grant
- client_id: the client identifier
- client_secret: the client password

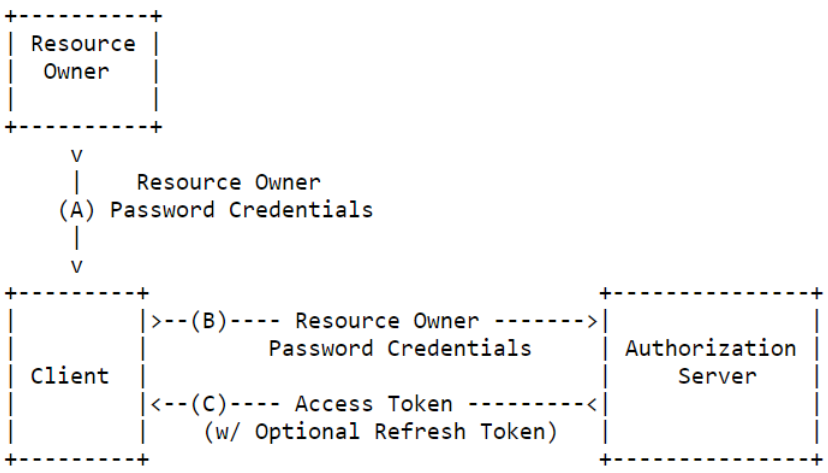

Figure 2.3: Resource owner password credentials flow [20]

2.7 Web API description

RESTful API description languages (DLs) are specific domain languages used for describing RESTful APIs, including resources, interfaces, and resource representation formats. The DLs describe RESTful API in a data serialization language, such

as JSON, Yet Ain't Markup Language (YAML), and XML, which are both human and machine-readable. Being human-readable means the DL is usable in API design and development and as API user documentation, since every stakeholder of the API has the same understanding. Meanwhile, being machine-readable means the DL can be processed by automated tools to generate code and render documentation User Interface (UI) , such as the Swagger Codegen [11] and Swagger UI [36].

In this section, several primary DLs are listed [13].

- The Web Application Description Language (WADL) [19] was introduced by Sun Microsystems in 2009 as a HTTP-based Web Service DL. It is extended from XML and suitable for describing Web Service interfaces.
- The RESTful API Modeling Language (RAML) [32] is an open-specification language created by the RAML Workgroup as a nonprofit project in 2014. The language is built on a YAML or JSON format and is lightweight and suitable for describing Web Services.
- API Blueprint [4] is an open-source project proposed Web API DL. Different from other DLs, it is built on a Markdown-flavor language , which can easily render to HTML using the Markdown tool.
- The OpenAPI Specification [18] is an open specification originally provided by the Swagger project in 2010. The project was later donated to the Open API Initiative (OAI) in 2016. The OpenAPI Specification is a standardized, language-agnostic, human-readable, and machine-acceptable language built on a YAML or JSON format, which enables the API stakeholders to discover and understand the API's capabilities without requiring any other assistance.

2.7.1 OpenAPI

In this thesis, the OpenAPI Specification is chosen as the RESTful API DL. Therefore, it is necessary to illustrate its schema. The OpenAPI Specification schema is composed of eight objects (see Table 2.2) under the root object, the OpenAPI object.

To utilize the OpenAPI Specification, Swagger provides several tools for different purposes. Swagger UI [36] is a tool to generate a web UI from the OpenAPI Specification; it allows the API development team and consumers to visualize and try out the APIs. The Swagger editor [11] is an OpenAPI Specification Intelligent Development Editor (IDE) to assist the API development team to modify the OpenAPI

Table 2.2: OpenAPI schema objects

Object name	Description
Openapi	The OpenAPI Specification version number
Info	It provides the API's metadata (e.g., the API version number).
Servers	It contains the API server configurations (e.g., base path).
Components	It is a common object that can be reused, such as a parameter, response, or schema.
Paths	It contains the endpoint's relative paths and operations.
Security	It specifies the security mechanisms required to authorize the requests.
Tags	A list of tags for sorting the endpoints.
ExternalDocs	API extension documents

Specification document. Swagger Codegen [11] is used for generating both server and client Software Development Kits (SDK) from the OpenAPI Specification.

2.8 Web API quality evaluation

Web APIs are widely used by various software [3]. Therefore, the quality (i.e., usability and stability) of a Web API is essential. There are different ways to evaluate a Web API's quality. For example, as a Web API is a type of software, the software quality criteria can be employed to evaluate them [28]. To this end, Meskens [28] proposes a set of software quality criteria to assist software developers in deciding whether the developed software should be redesigned or reimplemented. The criteria include reliability, flexibility, reusability, maintainability, and testability.

Daud and Kadir [10] and Chahal and Singh [7] also present three metrics for measuring the quality of software: cohesion, coupling, and complexity. Cohesion shows whether the features in a software module are related to one another [25]. Coupling evaluates the software modules' dependency. Complexity measures the difficulty to use the software.

The following sections will describe the Richardson Maturity Model (RMM), Classification of HTTP-based APIs (CoHA), and W3 maturity models.

2.8.1 Richardson Maturity Model (RMM)

Maturity models are created to evaluate the quality of software design. These models can also be applied to evaluate Web APIs. For this purpose, the most widely known is perhaps the RMM, which was invented by Leonard Richardson. The model classifies Web APIs into four maturity levels from the lowest to the highest: the Swamp of POX (level 0), Resources (level 1), HTTP Verbs (level 2), and Hypertext As The Engine Of Application State (HATEOAS) (level 3) [16].

```
GET https://api.example.com/profile

{
  "name": "Steve",
  "picture": {
    "large": "https://somecdn.com/pictures/1200x1200.png",
    "medium": "https://somecdn.com/pictures/100x100.png",
    "small": "https://somecdn.com/pictures/10x10.png"
  }
}
```

Figure 2.4: HATEOAS response example

The model's first maturity level indicates that HTTP only acts as a remote interaction transport system without utilizing any other HTTP features, such as different HTTP methods or the HTTP protocol's metadata. For example, SOAP belongs to this level, as it only sends "Plain Old XML" (POX) back and forth [16]. A server-side process is usually invoked instead of invoking the RPC interface explicitly. Each interface functions as an endpoint; the request body will be parsed to determine the processing to invoke. This method is equivalent to downgrading the HTTP application layer protocol to the transport layer protocol. The HTTP header and payload are entirely isolated, the HTTP header is only used to guarantee the transport, and no business logic is involved; the payload contains all the business logic so that the API can ignore any information in the HTTP header.

The second level, Resources, introduces the concept of web resource into Web APIs. The multiple resource groups gather all the available information from the APIs. Each resource is uniquely identified and addressed [16]. The client may individually access the resources through the corresponding operations, for example, "GET" to fetch resource information.

The third level is to introduce HTTP methods and handle HTTP response status codes. If there are multiple resources, multiple URIs should be created. Because the URI and resource utilize multiple-to-one mapping, multiple URIs may point to a single resource, but a single URI cannot map to multiple resources. The URIs may associate with multiple HTTP methods to operate these resources, for example, using POST/GET/PUT/DELETE to perform the Create, Read, Update, Delete (CRUD) operations, respectively [16]. In this case, both the HTTP header and payload contain business logic. Most so-called RESTful APIs existing in the public API market are at this level.

The highest level is HATEOAS. According to Fielding [15], hypermedia is a prerequisite for REST. Anything else should not be proclaimed as REST. Since the response includes the link address, the client is free to choose what information to download. The client receives the link, which states the available options. Consequently, the API does not have to simultaneously return three different versions of the user profile image, as illustrated in Figure 2.4. The response contains hints for the client that there are three available image sizes to choose from and the location. In this way, clients can make choices that suit their needs according to different scenarios. Moreover, if the client only needs images in one format, downloading all three versions of the image will waste resources. HATEOAS not only reduces the network load but also enhances the client's flexibility and the APIs' discoverability.

The core concept of hypermedia is the so-called link element, and these interlinked resources describe a protocol, a series of steps that lead us to a certain goal, such as ordering and payment. HATEOAS is the essence of hypermedia: through the links between resources, the state of the entire application is changed, that is, hypermedia converts the state of distributed applications. The exchange between the server and client is a representation of the resource's rather than the application's state, and the representation of the transfer includes a link that reflects the application's state [16].

2.8.2 Classification of HTTP-based APIs: the CoHA Maturity Model

CoHA [1] provides a five-level maturity model to evaluate Web APIs, which includes the levels WS-*, RPC URI-Tunneling, HTTP-based Type I, and HTTP-based Type II, shown in Figure 2.5(b).

At the lowest level, WS-*, the Web API consumer requests the Web API via HTTP, and the request payload is encapsulated in the SOAP protocol and sent to the URIs

assigned to the Web API. At this level, the Web API is the most difficult to use, maintain, and enhance because the request payload and assigned URIs are usually proprietary. Therefore, the Web API and its consumers are tightly coupled.

At the RPC URI-Tunneling level, the resource modeling is used without enforcing the operations (see Section 2.3.1), such as GET or POST, applied to the resource URIs. Again, the Web API and its consumers are tightly coupled.

At the HTTP-based Type I level, the HTTP protocol's resource modeling and semantic operations (see Section 2.3.1) are adopted. The resource is represented in multiple data formats as representations for the Web API consumer to choose from. Moreover, the Web API must be stateless for each request, which increases its scalability.

At the HTTP-based Type II level, the self-described message must be provided on the basis of HTTP-based Type I and should define the resource representation and operations. Therefore, the Web API should only notice the Web API consumer interacting with the uniform interfaces ; the rest of the request information is already in the self-described message. The highest level is REST. At this level, the Web API should conform to the REST architectural style (see Chapter 2.2).

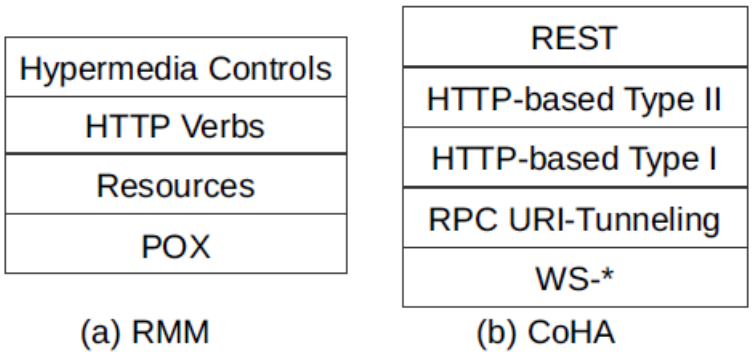

Figure 2.5: (a) Richardson and (b) CoHA maturity models [35]

2.8.3 WS3 maturity model

Ivan Salvadori and Frank Siqueira [35] present the WS3 maturity model for evaluating Web APIs. The WS3 maturity model is a three-dimension model, where each dimension focuses on one aspect of the Web API evaluation: the design dimension, the profile dimension, and the semantic dimension.

The design dimension, which is derived from the RMM, describes a Web API's structural characteristics. Since it is derived from the RMM, the design dimension is comprised of four levels. These maturity levels are listed from the lowest to the highest.

- RPC: The Web API that complies to this level is an RPC API (see Section 2.1.1).
- Resource: The data provided by the Web API is constructed into resource form, as described in the REST architecture style (see Section 2.2.1).
- Protocol compliance: The resources are accessible via URI and operated through the HTTP methods (see Section 2.3.2).
- Atomic resources: Both the resource and its properties are accessible and operated through the HTTP methods. For example, the resident as a resource can be operated via GET/POST/PUT/DELETE "/resident/id". However, the Web API also provides another endpoint to operate the resident's email via PUT "/resident/id/email".

A Web API profile dedicates to describing the resource structure and the operations on the resources provided by a Web API. The profile dimension evaluates how a Web API is described to guide its consumers. The profile dimension is divided into two levels: the lower level (interaction profile) and the upper level (domain profile).

- Interaction profile: The Web API documentation should include the resource description, resource operations descriptions, and resource representation description.
- Domain profile: The Web API documentation should also provide instructions for the Web API consumer on how to achieve a specific goal, such as the order of the resource operations to achieve a goal and the pre- and postcondition of a resource operation.

The semantic dimension focuses on how the resources are represented and consumed semantically. The semantic technology enables the Web API consumer to easily understand the resources and avoid ambiguity. The semantic dimension is divided into two levels: the lower level (semantic description) and the upper level (linked data).

- Semantic Description: The resources' properties and operations are semantically described.
- Linked data: The resources are not only semantically described, but their relationship is also described.

3 Design and implementation of the RAI Web API

3.1 Case description

The Resident Assessment Instrument (RAI) software is a web application that is used within different care facilities to assist nurses in developing individualized care plans for residents. Examples of care facilities where the system is used include nursing homes and hospitals. In this context, a resident is a client or patient who is receiving healthcare or therapy. The RAI software aims to help nurses collect information about a resident's health status, such as observation information, medication usages, and diagnosis history, which is used as indicators for drawing up and revising care plans as well as evaluating the care plans' goal achievement [21].

The RAI provides a standardized approach that applies a resident's health situation process evaluation, which includes the following steps:

1. Assessment: Collecting data from a resident, including the resident's demographic and clinical information, observations, and specific care type information.

2. Decision-making: Analyzing the assessment data and determining and understanding the resident's condition.

3. Care planning: Making a plan to achieve the resident's care goal(s).

4. Implementation: Implementing the care plan; the nurse will provide care services and therapies according to the care plan, such as monitoring the resident's activities of daily living.

5. Evaluation: Reviewing the care plan goal(s), determining the resident's needs, and implementing revisions to the care plan.

The RAI software supports nurses in completing the above process online. Each nurse that uses the system is granted a user account. The users are grouped into different roles associated with different user privileges. For example, the head nurse should be in an administration role to have the privilege of creating a new user account, which an ordinary nurse should not have.

The RAI software also provides an organization structure to map to the care facility's organization structure. For instance, Figure 3.1 illustrates a healthcare center's organization structure, where each team is mapped to the RAI software's organization structure. When a resident is admitted to the team, a resident case will be created in the ward in which the resident is admitted. The case indicates when a resident starts to be assessed by the nurse with a start date and end date in the team. During the case period, the nurse may create serial assessments to evaluate the resident's health status.

When a case has been created, the nurse can start the first resident assessment by creating an assessment form in the RAI software according to the resident's care type, for example, home or long-term care. The assessment form contains multiple questions regarding different aspects of a resident's health status. The nurse must collect the answers by interviewing and observing the resident. The completed assessment form is uploaded to the RAI software to analyze and calculate indicators to support decision-making. With the help of the analysis results, the nurse may obtain a better understanding of the resident's health issues. The nurse will make a care plan in the RAI software, focusing on the highlighted issues.

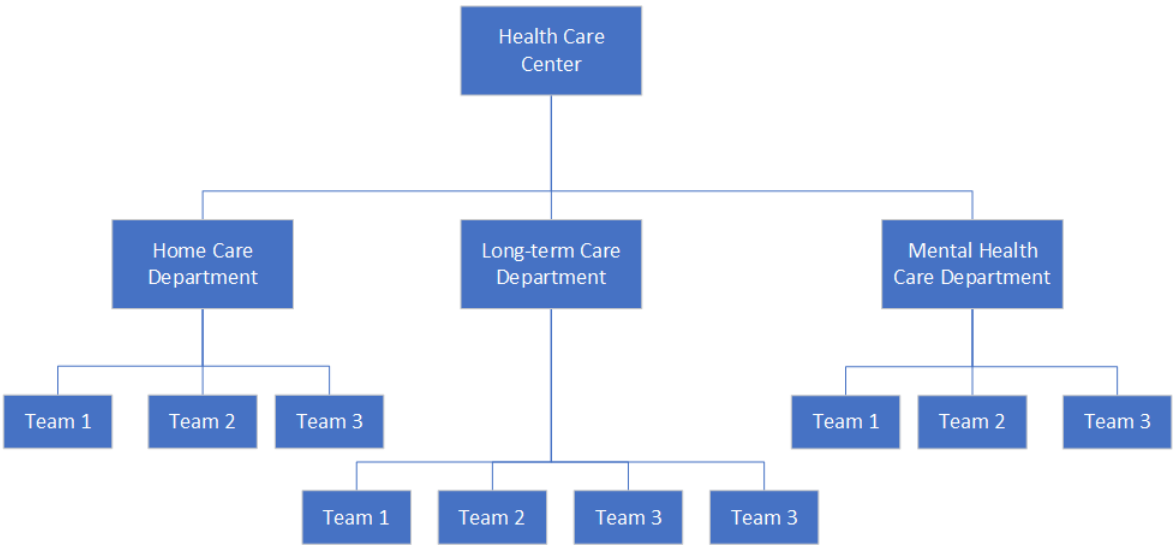

Figure 3.1: Healthcare center organization structure example

The RAI software is implemented as a web application that utilizes a 3-tier web architecture composed of the following tiers: a data tier, an application tier, and a presentation tier (Figure 3.2). The data tier is comprised of a Microsoft SQL database

server, and each customer owns an independent database instance. The resident data is collected and inserted by the nurses.

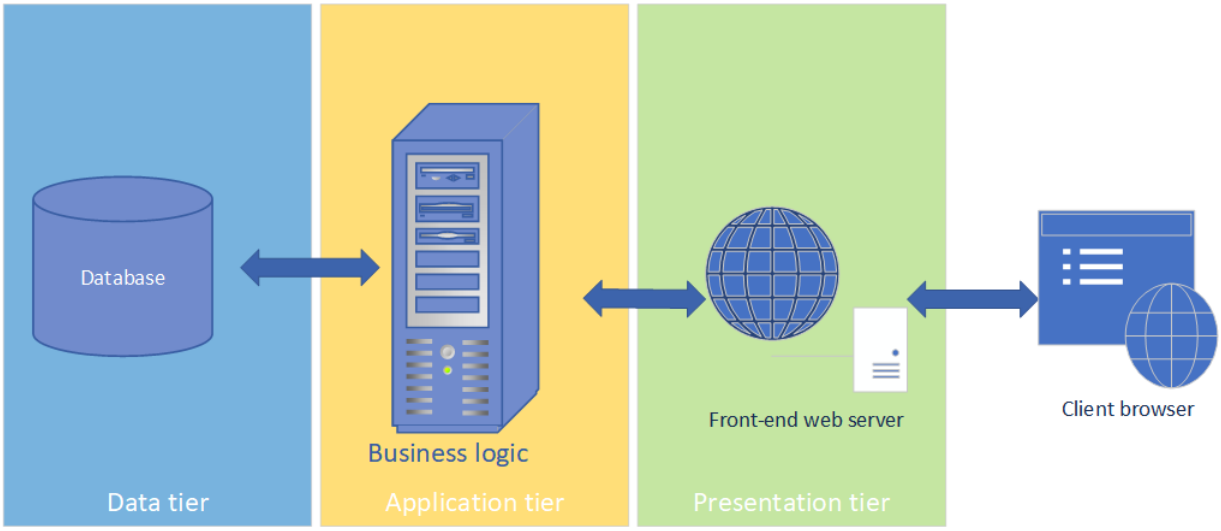

Figure 3.2: RAI software system architecture

The application tier functions as a middleware between the data and the presentation tiers and implements the business logic server, which is the only interface to the database. Hence, any data operation must be executed through the provided Message API (see Section 2.1.2). The business logic server also contains the authentication, authorization, and the business logic servers.

The presentation tier is placed on the top of the system's architecture and uses the data provided by the application tier to generate the User Interface (UI). Since the application is web-based, a separate web server is used within this tier to create the UI.

Figure 3.3 shows a sequence diagram of how the application processes an event in those three tiers. When the user queries for the information through the UI, the web server composes a request using a proprietary protocol in the HTTP request body and sends it to the business logic server. The business logic server checks the user's credential to determine its identity and privileges. After successful authentication, the business logic server processes the request and queries the requested data from the connected database. The business logic server composes the response and returns it to the web server so that the web server can render the UI for the user.

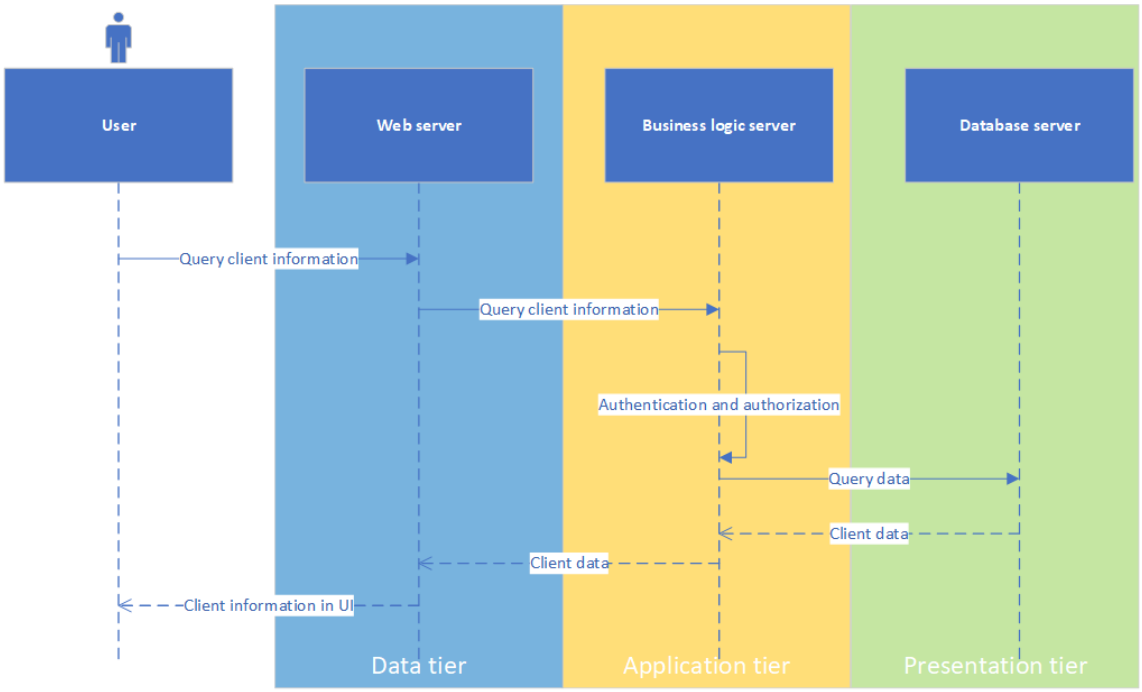

Figure 3.3: Client request sequence diagram

3.2 RAI API requirements

The RAI software only provides a Graphic User Interface (GUI) for user interaction, which means users can only send and receive data through the GUI. The GUI is only sufficient for a user to interact with the RAI software through a browser. It should be possible to reuse the health information data stored by other software systems. Therefore, a Web API should be provided on top of the RAI software (RAI API) to enable system interaction.

The requirements for developing a Web API are derived from two different user groups (Figure 3.4): the development team of the company that constructed the RAI software and collaborators that provide other health information systems, for example, an Electronic Health Record (EHR) system.

The development team would like to utilize the RAI API to access the business logic server to build an integration application to import and export data against the integration target system's API services. The integration application is an adapter between an external health information system and the RAI software to execute the operations on both the external system API and the RAI API for synchronizing user accounts, resident information, organization structure, and partial assessments.

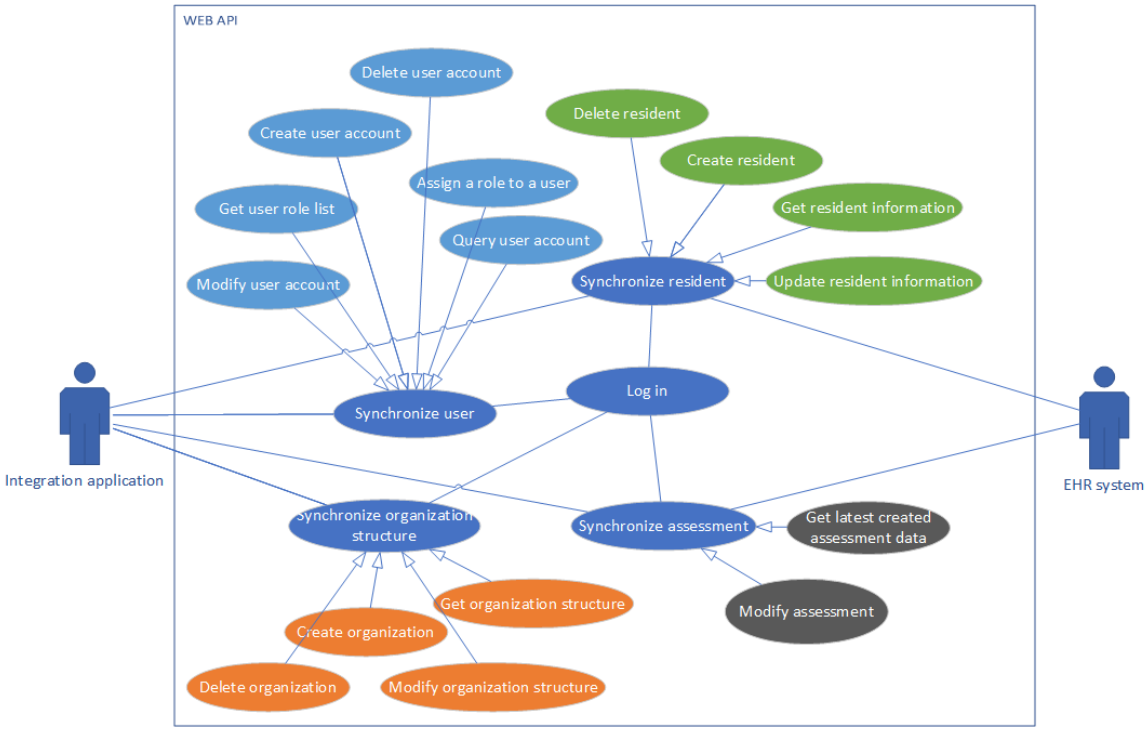

Figure 3.4: RAI API use case diagram

The following list contains the RAI API use cases (UC) and the functional requirements (FR) derived from these use cases.

- UC-1 Log in: The integration application logs into the RAI API to get authentication so the integration application is able to execute the other operations.
- UC-2 Synchronize user: The integration application fetches the user account list from the external system's API and maps it to the RAI software through the operations provided by the RAI API.
  - FR-1 Create user account
  - FR-2 Query user account
  - FR-3 Get user role list
  - FR-4 Modify user account
  - FR-5 Assign a role to a user
- UC-3 Synchronize organization structure: The integration application gets the organization structure from the external system API and maps it to the RAI software using the below operations in the RAI API.

  - FR-6 Get organization structure
  - FR-7 Create organization
  - FR-8 Modify organization structure

- UC-4 Synchronize resident: The integration application will get a notification from the external system when a change occurs on the resident records. The integration application will synchronize the resident records in the RAI software by utilizing the operations listed below.
  - FR-9 Create resident
  - FR-10 Get resident information
  - FR-11 Update resident information
  - FR-12 Delete resident

- UC-5 Synchronize assessment: The integration application will get a notification from the external system when an assessment question answer is available. The integration application will fetch the question answer and populate to the assessment using the operations listed below.
  - FR-13 Get the latest created assessment
  - FR-14 Modify assessment

A collaborator, such as an EHR manufacturer, uses the API to synchronize resident information and partial assessments.

- UC-6 Synchronize resident: The EHR system will push resident information to the RAI software through the RAI API using the below operations.
  - FR-15 Create resident
  - FR-16 Update resident information

- UC-7 Synchronize assessment: The EHR system will synchronize assessments data from the RAI software through the RAI API using the below operation.
  - FR-17 Get the latest created assessment

Besides the software requirements listed above, the following non-functional software requirements (NFR) should be implemented.

- NFR-1: The response time for each request should not be more than 2 seconds.
- NFR-2: The RAI API up-time shall not be less than 1 hour/month.
- NFR-3: The RAI API shall identify all the users before allowing them to access the data.
- NFR-4: The RAI API shall track all the user's operations in the system and provide a complete log.
- NFR-5: The RAI API shall provide an active documentation which allows the user to try out the system in real time.

The RAI API is used by the development team and collaborators (see Section 3.1). It is planned to be a public API to be used by various software systems to interact with RAI software in the future. In other words, systems will depend on the RAI API. Thus, it is under continuous development to enable new features and bug corrections. This means that the RAI API will inevitably perform necessary upgrades and generate more versions. Therefore, avoiding version lock and promiscuity between the software and the RAI API as much as possible is critical. The RAI API must be compatible with the current software, which depends on the RAI API and the upgrade itself.

- NFR-6 RAI API versioning strategy shall be backward compatible.

3.3 RAI API design

The RAI API architecture is composed of following list components.

- The version control component describes the versioning strategy implemented in the RAI API development and deployment. Semantic versioning is applied in the RAI API.
- Resource controllers manage the retrieval of data from the business logic server and construct it to the RAI API resources.
- The data format component determines how the resources are represented in the interfaces.

- The security component acts as a security protection layer in both data transmission and user access control.
- The interface component includes the RAI API's endpoints for the user to access and utilize the RAI API.
- The service description component aims to provide instructions corresponding to the different components.

3.3.1 Resource

A resource is a fundamental concept of a RESTful API, which is also implemented in the RAI API. A resource in this context is comparable to an object in object-oriented programming. Each resource has its own properties, such as an identifier, type, and common methods, including create, read, update, and delete.

There are six different resources implemented in the RAI API: user, role, organization, person, case, and assessment. A brief description of these resources follows.

- User: The RAI API user. Each RAI API user has a user account. Every operation against the RAI API is performed via a user account.
- Role: Represents a set of user privileges. A role can be granted to a user to enable specific application features.
- Organization: Represents a node within a customer's organization, for example, a hospital ward or a care facility team.
- Resident: Represents the resident, who is the primary object for which an assessment is made.
- Case: Represents an assessment period. The case starts from when the first resident assessment is created and ends when the last assessment is completed in an organization.
- Assessment: Represents an assessment form, which is an aggregation of questions. An assessment must be made as part of a resident's case.

Figure 3.5 shows the relationship between the resources. A single resident can be associated with multiple cases created in different organizations. A case can be

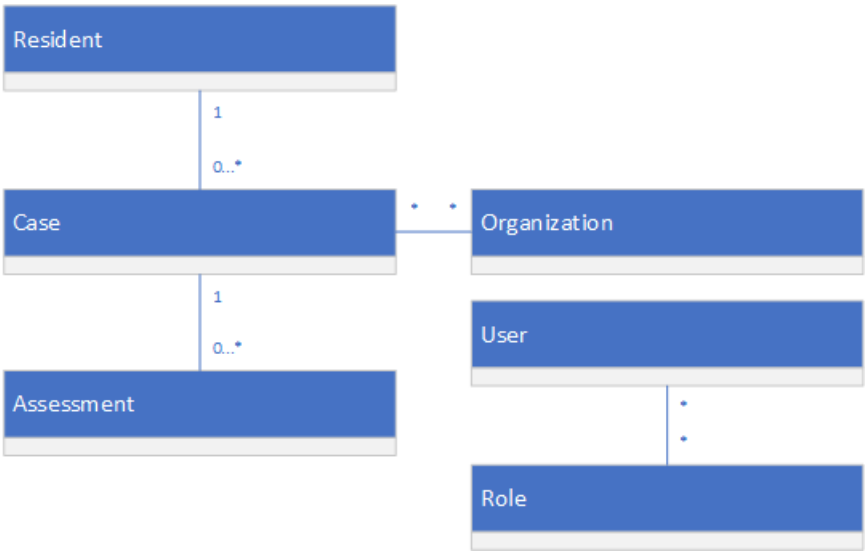

Figure 3.5: Resources relationship

associated with multiple assessments. As described in Section 3.1, a user may have multiple roles, while a role may be associated with multiple users.

The resource in RAI API can be one single type resource such as a resident or an assessment or collections of the single type of resources. For example, "GET /users" (see Table 3.1) represents a collection of user resources. A resource in the RAI API can be a single type resource, such as a resident or an assessment or a collections of single type resources. For example, "GET /users" (Table 3.1) represents a collection of user resources.

A resource can also be a different type of resource combination. For example, "GET /resident/id/cases" (Table 3.1) is an interface for fetching the combination of a resident resource by ID and a case resource collection associated with the resident.

3.3.2 Resource operations

As REST architecture 2.1.3 describes, the CRUD operations on the resources should use a specific HTTP method. In the RAI API, CRUD operations can be executed on each resource through the HTTP methods "POST," "GET," "PUT," and "DELETE," which correspond to the "Create," "Read," "Update," and "Delete" operations. Table 3.1 shows the resources' URIs and corresponding operations. Because each operation is an HTTP request, the server should return a response. The response indicates whether the operation was successful by parsing the HTTP status code. A Web Ser-

Table 3.1: Software requirements and corresponding resource operations

Software requirement	URI	Method	Description
FR-1	/user	POST	Create a user account with user properties, such as user name and credentials
FR-2	/user?username =USERNAME	GET	Search user account by username
FR-3	/users	GET	Fetch user list
FR-4	/user/{id}	PUT	Modify user account
FR-5	/user/{id}/role	POST	Assign a role to the user account
FR-6	/organization? organization_ name=ORG_NAME	GET	Get organization structure, if no query argument is supplied, the entire organization will be returned
FR-7	/organization	POST	Create organization node
FR-8	/organization/{id}	PUT	Modify organization node, (e.g., move organization, update organization name)
FR-9	/resident	POST	Create resident entity
FR-10, FR-15	/resident/{id}	GET	Get resident information by resident ID
FR-11, FR-16	/resident/{id}	PUT	Update resident information, e.g. change resident name
FR-12	/resident/{id}	DELETE	Remove resident entity by resident ID
FR-13, FR-17	/resident/{id}/cases	GET	Get resident and its cases collection
FR-13, FR-17	/resident/{id}/cases /assessment?latest=1	GET	Get latest assessment, which is the latest created under the resident

vice uses four sets of HTTP status codes: 2XX (Success), 4XX (Client errors), and 5XX (Server errors). Table 3.2

Table 3.2: RAI API response status code and description

Response status code	Description
200	OK. This code states the successful general request; the response will contain a related entity description.
201	Created. This is reserved to state the "POST" request to successfully create a new entity.
204	No content. This is reserved to state the "DELETE" request to successfully delete an entity.
400	Bad request. This states that the client sent an invalid request, so the server refuses the operation (e.g., the input parameter format should be a valid JSON ; an XML format parameter is supplied).
401	Unauthorized. This states that the client cannot pass through the authorization.
403	Forbidden. This states that the user does not have permission to access the target entity.
404	Not found. This states that the target entity is not found.
409	Conflict. This states that the operation on the target resource has encountered an error because of a conflict with another resource or itself.
500	Internal server error. This states that there is a general unknown error on the server side.

3.3.3 Media type

In the RAI API, JSON is chosen and utilized as the resource representation format on both the server and client side. In other words, the content data in both the service request and the response are in JSON format. As Appendix A shown, when an RAI

API client requests a resource from the server, the server responds in a default media type, such as "application/json;" meanwhile, an available media type list is attached to the response to indicate the alternative media types. This process is defined as the HTTP transparent content negotiation, whose mechanism is implemented in the initial phase of service development.

The service configuration allows the client to specify a request content type in the HTTP "Content-Type" header and the expected response content type in the HTTP "Accept" header. The specified content type is used to notify the server of which media type is chosen from the available media type list. In the RAI API, the client can specify the request content type as "Content-Type=application/json" if the request has a body and the body content type is JSON, and the response content type is presented by the HTTP "Content-Type" header if the response body content type is JSON.

Moreover, the client can also specify the expected format of the response content like "Accept=application/json" in the response header "Content-Type." If the expected content type is not supported, the HTTP status code 406 (Not acceptable) will be returned. The following syntax is a complete example of the request and response flow. "Content-Type" and "Accept" in the request state the requested body media type and the expected response body media type, and "Content-Type" in the response states the response body media type as "application/json."

3.3.4 RAI API versioning

Semantic versioning, a widely used backward-compatible versioning strategy, is used in the RAI API. According to the semantic versioning specification [31], because a legacy RAI software API version was released, the MAJOR version is noted as "2," which refers to the RAI API's second MAJOR version. The MINOR and PATCH versions are "0," which indicates that both the minor and patch are released as the first version. Before the first official RAI API is released, the suffix "alpha" is appended behind the version number to notate that the RAI API is in an internal testing stage, and the suffix "beta" indicates the RAI API is published to the public for a testing stage. For instance, "2.0.0-alpha" and "2.0.0-beta" are test versions, "2.0.0" is the first official release version, and "2.0.1" is the first MAJOR and MINOR version with the first bug correction version.

The version notation is placed in the URL before the resource, for example, "/v2/user." The "v2" represents the version "v" prefix of the MAJOR version number

"2". According to the backward-compatible strategy, only the latest version under the MAJOR version is deployed. Hence, from a API user's perspective, the minor versions are not visible. Furthermore, the full version number (e.g., 2.0.1) is noted in the RAI API documentation and upgrade changelogs.

3.3.5 RAI API security

In the RAI API, two major components are implemented to ensure information security. These are the Hypertext Transfer Protocol Secure (HTTPS) and OAuth 2.0.

There are two roles that consume the RAI API in this section: the RAI software user and the client system. The RAI software user refers to a person who has a user account in the RAI software. The RAI software is the authority of the user identity. The client system is a software that consumes the RAI API to access the resources stored in the RAI software. When a user wants to access the RAI software resource, he/she will delegate the client system to access the RAI software through the RAI API.

- HTTPS
  The HTTPS is used to prevent man-in-the-middle attacks in the RAI API, which could happen in the data transmission via the network. As described in Section 2.6.1, the TLS 1.2 protocol is the latest released version without any known security vulnerability. Therefore, TLS 1.2 is set as the minimum requirement for the client system to interact with the RAI API interfaces, which means that the client system must support the TLS 1.2 version during the SSL handshake.

- OAuth 2.0
  The user access security in the RAI API is responsible for verifying the user identity with the RAI software. The resource controller (see Section 3.3) will supply the requested resource based on the user identity. As a result, the business logic server has evolved from a legacy desktop application; it maintains all the users' identity and privileges information. Therefore, the business logic server acts as an authentication server.

  OAuth 2.0 defines a specification for authorization only, while OpenID Connect is placed on top of OAuth 2.0 to enable authentication (see Section 2.6.2). Since the business logic server is defined to manage the user identity authentication, the OpenID Connect does not need to be implemented in the RAI

API to perform the RAI software user identity authentication. Therefore, only OAuth 2.0 is implemented as the authentication and authorization flow for the RAI API client systems. There is a component in the RAI API that implements OAuth 2.0 as an authentication and authorization server.

The OAuth 2.0 implementation in the RAI API includes three procedures:

1. Enrolling the client system: When the client system wants to consume the RAI API, the manufacturer of the client system should submit an application to the RAI API administrator to enroll the client system so he/she may grant a client credential to the client system.
2. Verifying the user and client system identity and privileges to grant an access token: When an RAI software user wants to delegate the client system to get the RAI API interface access right, the client system should send a request to the RAI API to obtain an access token. The request should contain the user and client system credentials for the RAI API to verify the access token.
3. Verifying the access token to grant interface access: When an RAI software user wants to delegate the client system to access the RAI API resource, the client system should send a resource request with an access token. The RAI API will respond with the corresponding resource after verifying the access token.

When a new client system enrolls to use the RAI API, the RAI API administrator must create a client account in the RAI API that includes the parameters, client_id, client_secret, grant_type, and scope (see Section 2.6.2). The client_id and client_secret parameters are assigned in plain text by the RAI API administrator as the client system credential to identify the client system. The grant_type parameter should be set as a password to specify the client system requests, using an access token with the resource owner's password (see Section 2.6.2). The scope parameter states which interfaces the client system should be able to access, for example, read and create resident.

Figure 3.6 shows the sequence of verifying the user and client system identity and privileges.

I. Login: When a user logs in to the client system, he/she will provide user credential to the client system, which will forward the user credential,

client credential, grant type, and accessing interfaces scope to the RAI API.

II. GetAccessToken: The RAI API takes responsibility for verifying the client system using the client credential and interface scope.

III. VerifyUser: The user credential is forwarded to the business logic server to verify the user.

IV. ReturnUserInfo: The business logic server returns the user identity to RAI API to bind with the access token.

V. ReturnToken: If the client and user are valid, an access token, which is bound with the user information, is returned to the client system. The access token is stored in the client system for resource requests before the expiry time.

When the user logged into the client system, every request delegated by the user to the client system will be sent with the access token. RAI API validates and parses the access token to get the user's information. RAI API will request the resource for the user since the access token is valid. If the access token is expired, the user can use the refresh token to get a new access token without requiring the user to log in again.

3.3.6 Service description

The RAI API is implemented according to the OpenAPI Specification, so it can be defined with an OpenAPI definition. OpenAPI definitions are files in either JSON or YAML (YAML Ain't Markup Language) format. If a RESTful API provides an OpenAPI definition, the RESTful API user may utilize it as a service description, which is in machine-readable form. The OpenAPI definition is imported into many programming applications to automatically generate client code, for example, the Swagger editor [11] is an open-source software to generate a Software Development Kit (SDK) to help both Web API server and client development.

Since the RAI API is programmed in the Hypertext Preprocessor (PHP) language, the OpenAPI definition of the RAI API is generated from its source code with the help of an open-source software named Swagger-PHP. When an interface is implemented in the RAI API, a set of annotation codes that conforms to Swagger-PHP specifications (e.g., Listing B) is added to the source code to describe the interface,

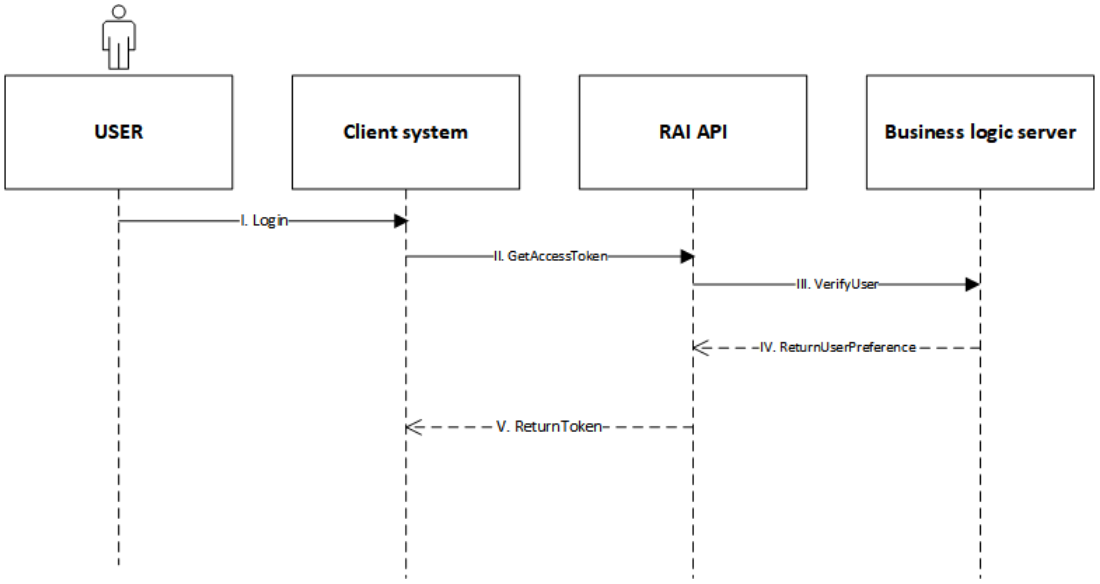

Figure 3.6: Verifying the RAI API user and client system identity and privileges to grant an access token

including the interface path, method, request and response schemas, and security schema. Swagger-PHP will scan through all the source code files and collect the annotations to generate the OpenAPI definition file.

For the RAI API, the JSON format file is chosen as the OpenAPI definition. Figure C is a code snippet of the RAI API OpenAPI definition. It shows the version of the OpenAPI Specification as 3.0.0, meta information, a server configuration, and an example interface for querying user account information.

The RAI API OpenAPI definition file is exported to the Swagger UI (Figure 3.7), a documentation generator made for generating visual documentation for the user to view and interact with the API. The documentation is published as a public web page, which will be supplied to the RAI API users as a visualized service description. Figure 3.7 is a screenshot that demonstrates the interface for querying a user account. There are mainly three fields in the screenshot. Field 1 shows the path and name of the endpoint. Field 2 lists the parameters of the request to the endpoint. Field 3 provides a response status code, content type and example value.

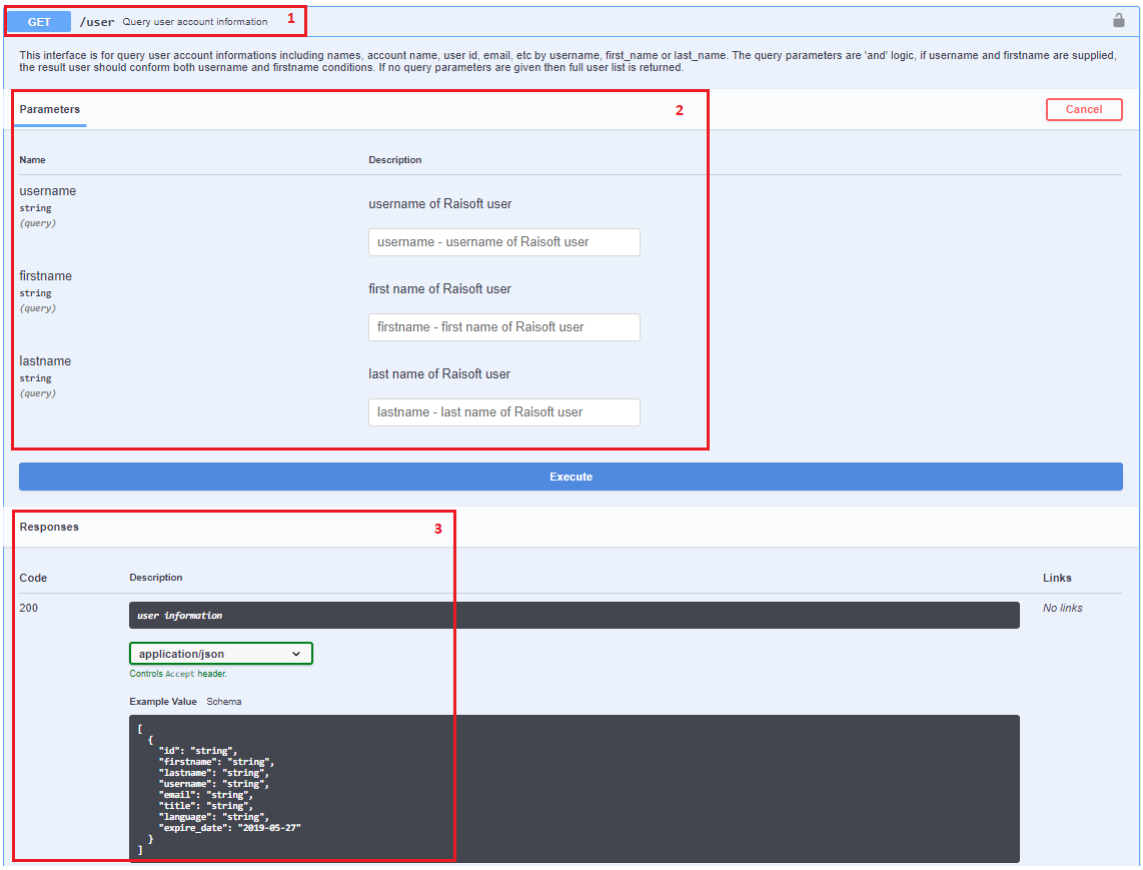

Figure 3.7: A screenshot of querying a user account interface in Swagger UI

4 Evaluation

This chapter focuses on evaluating the RAI API that has been implemented for the RAI software. The WS3 maturity model (see Section 2.5) is utilized to evaluate the RAI API. The evaluation is composed of three dimensions in the maturity model: the design dimension, the profile dimension, and the semantic dimension. Furthermore, the open issues in the RAI API design and implementation are discussed in this chapter.

4.1 Design dimension

The data provided by the RAI API is modeled in the resource forms. As described in Section 3.3.1, the resources include resident, case, assessment, role, user, and organization, and the operations are assigned to each resource to perform CRUD on it. The resources are manipulated through and self-described in their representation. For example, the resident resource contains the resident's first name, last name, birth date, gender, and personal identifier (PID). The data in the resident resource representation is named semantically, such as first_name, last_name, birth_date, gender, and PID, so that the developer of the RAI API consumer application can understand the data by the data name.

However, HATEOAS (see Section 2.8.1) is not adopted in the RAI API due to the project schedule limitation. That means there is no reference link in the resources to achieve HATEOAS. In another respect, even though the data in the RAI API is modeled in the resources, the resource properties (e.g., first name and last name) are not directly accessible. For instance, if the RAI API consumer wants to update a resident's last name, the RAI API consumer must perform a PUT operation on the endpoint of "/resident" instead of updating the last name directly by updating "/resident/last_name."

Therefore, the RAI API achieves the third level, HTTP Verbs, in the RMM (see Section 2.8.1) and the protocol compliance level in the design level of the WS3 maturity model.

4.2 Profile dimension

The RAI API is documented and described with an OpenAPI Specification (see Section 2.7.1) in which the security requirements, resources, operations, and representations are described. However, the description instructing the API to achieve a specific goal is not explicitly documented. For example, the approach of deleting a specific resident's assessment should be a combination of several operations. The RAI API consumer should first query the resident by the resident's property, such as their PID. Then, the assessment should be found with the resident's ID. In the end, the RAI API consumer can delete the assessment by its ID. The RAI API consumer must be familiar with the operations to achieve a goal, which is usually not the common case. So far, the RAI API documentation does not contain the descriptions to achieve any complicated goal. Therefore, the RAI API is assigned to the interaction profile level in the WS3 maturity model.

4.3 Semantic dimension

As discussed in Section 4.1, the resources are named semantically, such as resident and case, and are described by name and documentation. However, the RAI API consumer cannot understand the relationships by either the resource name or the documentation. For example, the relationship between the resident and the case is vague. The lack of resource relationship description results in extra workloads, such as more communication between the RAI API provider and consumer. Therefore, the RAI API conforms to the semantic description level of the WS3 maturity model.

4.4 Open issues

The open issues in this thesis involve two aspects: querying a resource with the GET method, which leads to security vulnerability, and lacking authentication methods.

4.4.1 GET method security vulnerability

According to the REST architectural style specification, the HTTP method should be adopted in the resources to perform the operations. The GET method is reserved to query resources, such as querying resident information using the resident's PID (see

Section 2.3.2). However, the GET method requires passing the query parameters via URL, and the parameters are recorded in the server logs and browser history. If the parameters include any personal information, such as PID, the GET method will lead to personal data leaking, which is considered a security vulnerability.

In the RAI API, the GET method is still employed as the method for querying resources, but the querying request should only contain insensitive information in the parameters. If the querying request involves any sensitive information, the POST method is adopted. Since the POST method is not specified for querying resources in the REST architectural style specification, a naming convention is applied on the request URL to distinguish it from the other regular endpoints. For example, the resident's national identifier is considered sensitive information; therefore, the URL of the endpoint for searching a resident by a national identifier is assigned as "/resident/_search," and the HTTP method is POST. The underscore is used to indicate that the endpoint is not for regularly querying with the GET method.

4.4.2 Lacking authorization method

The RAI API utilizes the OAuth 2.0 Password grant type as both an authentication and authorization solution (see Section 3.3.5). According to the OAuth 2.0 specification (see Section 2.6.2), the resource owner should send the user name and password to the authorization server via the client application. In the RAI API, the RAI software performs as the authorization server, which should store the resource owner's credentials. However, if the resource owner's credentials do not allow being stored in the RAI software, the third-party application cannot utilize the RAI API. For example, the Microsoft Active Directory (AD) server is usually employed as the authorization server in an organization. If the resource owner is required to be authorized by the Microsoft AD server instead of the RAI software, the RAI API will not be able to authorize the resource owner since there is no way for the RAI API to get the resource owner's identity. This reduces the RAI API's flexibility.

5 Conclusion

This Master's thesis focuses on solving the information silo problem that exists and hinders health data sharing among the different healthcare systems. The problem is exacerbated by the RAI software. When the RAI software needs to integrate with the EHR systems to exchange resident information (e.g., medication, patient demographic information), it lacks a method to interact with the EHR system since it provides the end-user with only a GUI to access the data. Therefore, this thesis aims to develop a solution for RAI software to interact with other EHR systems.

Using a Web API as a Web Service is considered a solution to the research problem. After reviewing the three types of API (see Section 2.1), the REST API is selected as the most effective Web API style. In the background chapter (see Chapter 2), different alternative techniques in RAI API components are demonstrated; for example, the data format component has two alternative options: XML and JSON. In the design and implementation phase, the RAI API requirement and specification are carried out; meanwhile, the alternative techniques are selected based on the customer requirements. The RAI API project started at the end of 2018. It is derived from a customer's business requirement of the RAI software. The RAI API is defined as a product that will be delivered continuously to different customers. The RAI API's first version was delivered to customers at the end of April, 2019. The author of this thesis is responsible for taking charge of the software development processes , including the requirement engineering, specification designing, and implementation.

The future work concentrates on enhancing RAI API security and maturity. As the open issues described, a safety search method is necessary to avoid sensitive data leaking. The search method is vital for the customer to query the resources using more keys (e.g., resident national number). Another work from the open issues is enabling more authorization grant types. That is also required by the customer when the he/she develops a mobile application. The mobile application demands the authorize code grant type. The last future work is leveling up the RAI API to HATEOAS. Nevertheless, this thesis demonstrates the study processes of constructing the RAI API; from the RAI API construction processes, the author of this thesis

learns how to develop a software as a solution to meet the customers' requirements as well as how to evaluate the solution, thereby improving it.

References

[1] ALGERMISSEN, J. Classification of http-based apis. URL `http://algermissen.io/classification_of_http_apis.html`, visited on 3.11.2019.

[2] ALONSO, G., CASATI, F., KUNO, H., AND MACHIRAJU, V. Web services. In *Web Services*. Springer, 2004, ch. 5, pp. 123–149.

[3] BERMBACH, D., AND WITTERN, E. Benchmarking web api quality. In *International Conference on Web Engineering* (Lugano, Switzerland, June 2016), Springer, pp. 188–206.

[4] BERNSTEIN, D., LUDVIGSON, E., SANKAR, K., DIAMOND, S., AND MORROW, M. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In *2009 fourth international conference on Internet and web applications and services* (Venice, Italy, May 2009), IEEE, pp. 328–336.

[5] BEX, G. J., NEVEN, F., AND VAN DEN BUSSCHE, J. DTDs Versus XML Schema: A Practical Study. In *Proceedings of the 7th international workshop on the web and databases: colocated with ACM SIGMOD/PODS 2004* (Paris, France, June 2004), ACM, pp. 79–84.

[6] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., MALER, E., AND YERGEAU, F. Extensible markup language (XML). *World Wide Web Journal 2* (1997), 27–66.

[7] CHAHAL, K. K., AND SINGH, H. A metrics based approach to evaluate design of software components. In *IEEE International Conference on Global Software Engineering* (Bangalore, India, October 2008), IEEE, pp. 269–272.

[8] CHOMSIRI, T. HTTPS Hacking Protection. In *21st International Conference on Advanced Information Networking and Applications Workshops, AINAW'07.* (Niagara Falls, Ont., Canada, May 2007), IEEE, pp. 590–594.

[9] DAIGNEAU, R. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Pearson Education, Upper Saddle River, New Jersey, 2011.

[10] DAUD, N. M. N., AND KADIR, W. M. W. Static and dynamic classifications for SOA structural attributes metrics. In *8th. Malaysian Software Engineering Conference (MySEC)* (Langkawi, Malaysia, September 2014), IEEE, pp. 130–135.

[11] DE, B. API documentation. In *API Management*. Apress, Berkeley, CA, Springer, 2017, ch. 4, pp. 59–80.

[12] DE SOUZA, C. R., REDMILES, D., CHENG, L.-T., MILLEN, D., AND PATTERSON, J. How a good software practice thwarts collaboration: the multiple roles of apis in software development. *SIGSOFT Softw. Eng. Notes 29*, 6 (2004), 221–230.

[13] DI MARTINO, B., POSILLIPO, A., NACCHIA, S., AND MAISTO, S. A. A Q&A tool to produce an Ad-Hoc OpenAPI Specification to identify equivalent REST API services. In *IEEE International Conference on Smart Computing (SMARTCOMP)* (Taormina, Italy, 2018), IEEE, pp. 375–380.

[14] FIELDING, R., AND RESCHKE, J. *Hypertext transfer protocol (HTTP/1.1): Semantics and content*, 2014.

[15] FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. PhD thesis, University Of California, Irvine, USA, 2000.

[16] FOWLER, M. Richardson Maturity Model: steps toward the glory of REST. URL `http://martinfowler.com/articles/richardsonMaturityModel.html`, visited on 3.11.2019.

[17] GALIEGUE, F., ZYP, K., ET AL. JSON schema: Core definitions and terminology. URL `http://json-schema.org/draft-04/json-schema-core.html`, visited on 3.11.2019.

[18] GITHUB, I. OpenAPI Specification. URL `https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0`, visited on 3.11.2019.

[19] HADLEY, M. J. *Web Application Description Language (WADL)*. Tech. Rep. SMLI TR-2006-153, Sun Microsystems, Inc., Mountain View, CA, USA, 2006.

[20] HARDT, D. *The OAuth 2.0 authorization framework*, October 2012.

[21] HAWES, C. H., MORRIS, J. N., PHILLIPS, C. D., FRIES, B. E., MURPHY, K., AND MOR, V. Development of the Nursing Home Resident Assessment Instrument in the USA. *Age and Ageing 26*, 0002-0729 (1997), 19–25.

[22] HEATON, R. How does https actually work? URL `https://robertheaton.com/2014/03/27/how-does-https-actually-work/`, visited on 15.01.2018.

[23] HUNT, T. Your API versioning is wrong, which is why I decided to do it 3 different wrong ways. URL `https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/`, visited on 15.01.2018.

[24] INFOQ. The Costs of Versioning an API. URL `https://www.infoq.com/news/2013/12/api-versioning`, visited on 15.01.2018.

[25] IZADKHAH, H., AND HOOSHYAR, M. Class Cohesion Metrics for Software Engineering: A Critical Review. *Computer Science Journal of Moldova 25*, 1 (2017), 44–74.

[26] MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., Sebastopol, CA, 2011.

[27] MERRICK, P., ALLEN, S., AND LAPP, J. *XML remote procedure call (XML-RPC)*. Google Patents, USA, 2006.

[28] MESKENS, N. Software quality analysis system: a new approach. In *Proceedings of the 1996 IEEE IECON. 22nd International Conference on Industrial Electronics, Control, and Instrumentation* (Taipei, Taiwan, August 1996), IEEE, pp. 1406–1411.

[29] MILLER, A. R., AND TUCKER, C. Health information exchange, system size and information silos. *Journal of Health Economics 33*, 0167-6296 (2014), 28 – 42.

[30] PEZOA, F., REUTTER, J. L., SUAREZ, F., UGARTE, M., AND VRGOČ, D. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web* (2016), International World Wide Web Conferences Steering Committee, pp. 263–273.

[31] PRESTON-WERNER, T. Semantic Versioning 2.0.0. URL `https://semver.org/`, visited on 3.11.2019.

[32] PROGRAMMABLEWEB. RAML-RESTful API modeling language. URL `http://raml.org/`, visited on 3.11.2019.

[33] RECORDON, D., AND REED, D. OpenID 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital Identity Management* (Alexandria, Virginia, USA, 2006), ACM, pp. 11–16.

[34] REYES C. RPC Style vs. REST Web APIs. URL `https://blog.jscrambler.com/rpc-style-vs-rest-web-apis/`, visited on 11.6.2019.

[35] SALVADORI, I., AND SIQUEIRA, F. A maturity model for semantic restful web apis. In *2015 IEEE International Conference on Web Services* (New York, NY, USA, August 2015), IEEE, pp. 703–710.

[36] SMARTBEAR. Swagger UI. URL `https://swagger.io/tools/swagger-ui/`, visited on 28.5.2019.

[37] SNELL, J., TIDWELL, D., AND KULCHENKO, P. *Programming web services with SOAP: building distributed applications*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[38] VON ALAN, R. H., MARCH, S. T., PARK, J., AND RAM, S. Design science in information systems research. *MIS quarterly 28*, 1 (2004), 75–105.

A RAI API request and response example

```
Request:
POST /index.php/person HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Accept: application/json
Authorization: Bearer ACCESS\_TOKEN
{
"firstname": "test\_firstname",
"lastname": "test\_lastname",
"nni": "020202-0202",
"gender": "1",
"birth\_date": "1949-05-01"
}
Response:
Content-Type ->application/json;charset=utf-8
Transfer-Encoding ->chunked
Connection ->keep-alive
X-Powered-By ->PHP/7.2.13
Expires ->Thu, 19 Nov 1981 08:52:00 GMT
{
"id": "09DC0050-EE5B-4B77-AF26-BFA96C59AB9F",
"firstname": "test\_firstname",
"lastname": "test\_lastname",
"nni": "020202-0202",
"gender": 1,
"birth\_date": "1949-05-01",
"deceased": false
}
```

B Swagger-PHP annotation code snippet for generating OpenAPI definition

```
/**
 *@OA\Get(
 * path="/user",
 * tags={"User"},
 * description="This interface is for query user account
 informations including names, account name, user id, email,
 etc by username, first\_name or last\_name.
 The query parameters are 'and' logic; if the username and
 first name are supplied, the resulting user should fulfill
 both username and first name conditions.
 If no query parameters are given then full user list is
 returned.",
 * summary="Query user account information",
 * operationId = "QueryUser",
 * @OA\Parameter(
 *   name="username",
 *   in="query",
 *   description="username of Raisoft user",
 *   @OA\Schema(
 *     type="string"
 *   )
 * ),
 * @OA\Parameter(
 *   name="firstname",
 *   in="query",
 *   description="first name of Raisoft user",
 *   @OA\Schema(
 *     type="string"
 *   )
```

```
* ),
* @OA\Parameter(
*   name="lastname",
*   in="query",
*   description="last name of Raisoft user",
*   @OA\Schema(
*     type="string"
*   )
* ),
* @OA\Response(
*   response=200,
*   description="user information",
*   @OA\JsonContent(
*     type="array",
*     @OA\Items(ref="#/components/schemas/rs\_user")
*   )
* ,
* security={
*   {
*     "AuthServerSchema":{},
*     "bearAuth":{}
*   }
* }
*)
*/
```

C RAI API OpenAPI definition code snippet


```
{
 "openapi": "3.0.0",
 "info": {
  "title": "RAI API service description",
  "description": "",
  "termsOfService": "http://swagger.io/terms/",
  "contact": {
   "email": "support@example.com"
  },
  "version": "0.0.1"
 },
 "servers": [
  {
   "url": "http://localhost:8080/index.php",
   "description": "server root path"
  }
 ],
 "paths": {
  "/user": {
   "get": {
    "tags": [
     "User"
    ],
    "summary": "Query user account information",
    "description": "",
    "operationId": "QueryUser",
    "parameters": [
     {
      "name": "username",
      "in": "query",
```

```
  "description": "username of Raisoft user",
  "schema": {
   "type": "string"
  }
 },
 {
  "name": "firstname",
  "in": "query",
  "description": "first name of Raisoft user",
  "schema": {
   "type": "string"
  }
 },
 {
  "name": "lastname",
  "in": "query",
  "description": "last name of Raisoft user",
  "schema": {
   "type": "string"
  }
 }
],
"responses": {
 "200": {
  "description": "user information",
  "content": {
   "application/json": {
    "schema": {
     "type": "array",
     "items": {
      "$ref": "#/components/schemas/rs_user"
     ...
```