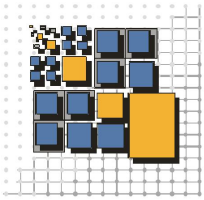

University of Bamberg

Distributed Systems Group

Project Report

With the topic:

Building REST APIs

Submitted by:

Istiaq Hasan, Kowshik Dipta Joy
Md Abdullah Al Mahmud, Md Aminul Islam
Md Imtiaz Abedin, Mohammed Mostakim Ornob
Tawhidul Islam

Supervisor:
Robin Lichtenthäler, M.Sc
Sebastian Böhm, B.Sc.

Examiner:
Prof. Dr. Guido Wirtz

Date of submission:
07.02.2020

Contents

List of Figures

List of Tables

Listings

Abbreviations

API	Application Programming Interface
RAML	RESTful API Modeling Language
YAML	YAML Ain't Markup Language
URI	Uniform Resource Identifier
JWT	JSON Web Token
SWT	Simple Web Token
SAML	Security Assertion Markup Language
XML	Extensible Markup Language
CORS	Cross Origin Resource Sharing
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
IoC	Inversion of Control
DI	Dependency Injection
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
RPC	Remote Procedure Call
GIF	Graphics Interchange Format
PNG	Portable Network Graphics

1 Introduction

objectives,objectives,methodology

2 REST

Md Abdullah Al Mahmud

2.1 What is REST?

Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations. Other kinds of Web services, such as SOAP Web services, expose their own arbitrary sets of operations.[54]

"Web resources" were first defined on the World Wide Web as documents or files identified by their URLs. The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service, a collection of other resources, a non-virtual object (e.g. a person), and so on.[54] In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

2.1.1 History

Prior to REST, developers used Simple Object Access Protocol (SOAP) to integrate APIs. To make a call, developers handwritten an XML document with a Remote Procedure Call (RPC) call in the body. They then specified the endpoint and POST their SOAP envelope to the endpoint. In 2000, Roy Fielding and a group of developers decided to create a standard so that any server could talk to any other server. Then the term representational state transfer was introduced and defined by Roy Fielding in his doctoral dissertation. Fielding's dissertation explained the REST principles that were known as the "HTTP object model" beginning in 1994, and were used in designing the HTTP 1.1 and Uniform Resource Identifier (URI) standards. The stated purpose was simply to create a standard that allows two servers to communicate and exchange data anywhere in the world. They therefore designed a set of principles, properties and constraints which they named REST.[16]

2.1.2 Guiding Principles of REST

Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful. These principles are listed below.[16]

- **Client–Server** – The client-server constraint works on the concept that the client

and the server should be separate from each other and allowed to evolve individually and independently. In other words, I should be able to make changes to my mobile application without impacting either the data structure or the database design on the server. At the same time, I should be able to modify the database or make changes to my server application without impacting the mobile client. This creates a separation of concerns. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

- **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. A REST API should not rely on data being stored on the server or sessions to determine what to do with a call, but rather solely rely on the data that is provided in that call itself. Session state is therefore kept entirely on the client in order to reduce memory requirements and keep your application as scalable as possible.

- **Cacheable** – To improve network efficiency, clients can also save responses sent by the server and use them again later for requests of the same type. Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. When data is cacheable, the response should indicate that the data can be stored up to a certain time (expires-at), or in cases where data needs to be real-time, that the response should not be cached by the client.

- **Uniform interface** – The components of REST-compliant services use a uniform, general interface that is decoupled from the implemented service. The uniform interface lets the client talk to the server in a single language, independent of the architectural backend of either. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: Resource identification in requests, Resource manipulation through representations, Self-descriptive messages, Hypermedia as the engine of application state (HATEOAS).

- **Layered system** – REST uses multilayer, hierarchical systems ("Layered System") - each component can only see directly adjacent layers. This allows, for example, encapsulation of legacy applications. Intermediaries acting as load balancers can also improve scalability. The disadvantages of this constraint are additional overhead and increased latencies.

- **Code on demand (optional)** – Perhaps the least known of the six constraints, and the only optional constraint, Code on Demand allows for code or applets to be transmitted via the API for use within the application. REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

3 Rest Over HTTP

Besides the constraints mentioned in Fielding's dissertation, Roy T. Fielding also clarified in a blog post that REST APIs must be hypertext-driven, just only invoking a service via HTTP does not make it RESTful. In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API[17]. A service should therefore also respect further rules which are summarised as follow:

- A REST API should not be dependent on any single communication protocol, though its successful mapping to a given protocol may be dependent on the availability of metadata, choice of methods, etc. In general, any protocol element that uses a URI for identification must allow any URI scheme to be used for the sake of that identification.
- A REST API should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols, such as HTTP's PATCH method or Link header field. Workarounds for broken implementations (such as those browsers stupid enough to believe that HTML defines HTTP's method set) should be defined separately, or at least in appendices, with an expectation that the workaround will eventually be obsolete.
- A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types).
- A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations.
- A REST API should never have "typed" resources that are significant to the client. Specification authors may use resource types for describing server implementation behind the interface, but those types must be irrelevant and invisible to the client. The only types that are significant to a client are the current representation's media type and standardized relation names.
- A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations. The transitions may be determined (or limited by) the client's knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand).

3.1 Richardson Maturity Model
Md Abdullah Al Mahmud

There are several ways to apply REST constraints over HTTP in order to obtain RESTful web services, Richardson Maturity Model is one of them. To identify the maturity level of REST services, Leonard Richardson analyzed different web service designs and divided them into four categories based on how much they are REST compliant. He mainly used three major factors to decide that. 1) URI, 2) HTTP Methods and 3) HATEOAS(Hyparmedia)[18]. Leonard Richardson divided applications into these 4 layers:

- **Level 0 (Use of HTTP for the transport):** Level 0 uses its implementing protocol (normally HTTP, but it doesn't have to be) like a transport protocol. These services have a single URI and use a single HTTP method (typically POST). Examples of these are SOAP and XML-RPC. Level 0 of maturity does not make use of any of URI, HTTP Methods, and HATEOAS capabilities.
- **Level 1 (Use of URL to identify resources):** Level one of maturity makes use of URIs out of URI, HTTP Methods, and HATEOAS. This level uses multiple URIs, where every URI is the entry point to a specific resource. That means every resource is separately identified by a unique URI.
- **Level 2 (Use of HTTP verbs and statuses for the interactions):** This level indicates that your API should use the protocol properties in order to deal with scalability and failures. Don't use a single POST method for all, but make use of GET when you are requesting resources, and use the DELETE method when you want to delete a resources. Also, use the response codes of your application protocol. Don't use 200 (OK) code when something went wrong for instance. By doing this for the HTTP application protocol, or any other application protocol you like to use, you have reached level 2.
- **Level 3 (Use of HATEOAS):** This is the most mature level of Richardson's model which encourages easy discoverability and makes it easy for the responses to be self-explanatory by using HATEOAS. The service leads consumers through a trail of resources, causing application state transitions as a result.

In our Student Residence Application, we followed Richardson Maturity Model as well. For example, we use HTTP as the transfer protocol, URL to identify resources, HTTP verbs and statuses for the interactions and HATEOAS.

3.2 Resource
Md Abdullah Al Mahmud

REST architecture treats every content as a resource. These resources can be Text Files, Html Pages, Images, Videos or Dynamic Business Data. REST Server simply provides

access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ Global IDs. REST uses various representations to represent a resource where Text, JSON, XML. The most popular representations of resources are XML and JSON. According to Roy T. Fielding, "The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g., a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time"[16].

Identification of Resources Each resource in a RESTful design must be uniquely identifiable via a URI, and the identifier must be stable even when the underlying resource is updated. This means that each resource you want to expose through a REST API must have its own URI.

Representation of Resources A resource in REST is a similar Object in Object Oriented Programming or is like an Entity in a Database. Once a resource is identified then its representation is to be decided using a standard format so that the server can send the resource in the above said format and client can understand the same format.

Good Resources Representation REST does not impose any restriction on the format of a resource representation. A client can ask for JSON representation whereas another client may ask for XML representation of the same resource to the server and so on. It is the responsibility of the REST server to pass the client the resource in the format that the client understands.

Following are some important points to be considered while designing a representation format of a resource in RESTful Web Services.

- **Understandability**: Both the Server and the Client should be able to understand and utilize the representation format of the resource.
- **Completeness**: Format should be able to represent a resource completely. For example, a resource can contain another resource. Format should be able to represent simple as well as complex structures of resources.
- **Linkablity**: A resource can have a linkage to another resource, a format should be able to handle such situations.

3.3 HTTP Requests and Responses
Md Abdullah Al Mahmud

A simple request message from a client computer consists of the following components:

- A verb (aka method), most of the time one of GET, POST, PUT, DELETE or PATCH.
- URI
- Headers (Example – Accept-Language: EN)
- A message body which is optional

A simple response from the server contains the following components:

- HTTP Status Code
- Headers (Example – Content-Type: html)
- A message body which is optional

3.4 HTTP Methods

Md Abdullah Al Mahmud

RESTful APIs enable you to develop any kind of web application having all possible CRUD (create, retrieve, update, delete) operations. REST guidelines suggest using a specific HTTP method on a specific type of call made to the server. A HTTP verb/method is a part of HTTP request. Request Methods indicate the desired action to be performed for a given resource. When a request is made by the client, it should send this information in the HTTP request:

- REST verb
- Header information
- Body (optional)

There are quite a few REST verbs available, but few of them are used frequently. The set of common methods for HTTP/1.1

- GET
- HEAD
- POST
- PUT
- DELETE
- CONNECT
- OPTIONS
- PATCH

GET GET requests are the most common and widely used methods in APIs and websites. Simply put, the GET method is used to retreive data from a server at the specified resource. If all is well, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

Since a GET request is only requesting data and not modifying any resources, it's considered a safe and idempotent method.

For example, in our Student Residence Application we used GET method in our different services to retrieve different resources. Some GET requests that we used in our application are given below

```
HTTP GET /api/appointments
HTTP GET /api/appointments/id
HTTP GET /api/contracts
HTTP GET /api/contracts/id
HTTP GET /api/contractors/id/contracts
HTTP GET /api/contractors/id/appointments
```

POST This method is often utilized to create new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. On successful creation, it returns HTTP status 201, returning a location header with a link to the newly-created resource with the 201 HTTP status. POST request is non-idempotent. It mutates data on the backend server(by creating or updating a resource).

Some POST requests that we used in our application are given below

```
HTTP POST /api/appointments
HTTP POST /api/contracts
```

PUT Simlar to POST, PUT requests are used to send data to the API to create or update a resource. The difference is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly make have side effects of creating the same resource multiple times. A successful update returns 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, it returns HTTP status 201 on successful creation.

PATCH A PATCH request is one of the lesser-known HTTP methods, it is similar to PUT. The difference with PATCH is that you only apply partial modifications to the resource. The difference between PATCH and PUT, is that a PATCH request is non-idempotent (like a POST request).

We used PATCH requests in our services. Like in appointment service, to accept an appointment or to deny an appointment we used PATCH just because we are doing

only a partial modification here. Also in contract service, we used PATCH to confirm , terminate or extend a particular contract.

```
HTTP PATCH /api/appointments/id
HTTP PATCH /api/contracts/id
```

DELETE The DELETE method is exactly as it sounds: delete a resource identified by a URL. Upon successful deletion, it returns HTTP status 200 (OK) along with a response body.

3.5 HTTP Headers

Md Abdullah Al Mahmud

HTTP headers are the core part of these HTTP requests and responses. It allows the client and the server pass additional information with an HTTP request or response. All the headers are case-insensitive key-value pairs. Whitespace before the value is ignored. The end of the header section denoted by an empty field header. There are a few header fields that can contain the comments.
Headers can be divided into four groups according to their contexts:

- **General Header:** A general header can be used in both request and response headers but not to the content itself. Depending on the context they are used in, general headers are either response or request headers. The most common general headers are Date, Cache-Control or Connection.

  ```
  Cache-control: no-cache
  Connection: keep-alive
  Sun, 02 Jan 2020 08:49:37 GMT  ;
  ```

- **Request headers:** A request header is an HTTP header that can be used in an HTTP request. This type of headers contains information about the fetched request by the client. The most common request headers are Accept, Accept-Charset, Accept-Encoding, Accept-Language, Authorization, Cookie etc.

  ```
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  Accept-Language: en-US,en;q=0.5
  Accept-Charset: iso-8859-5, unicode-1-1; q=0.8
  Accept-Encoding: compress;q=0.5, gzip;q=1.0
  Authorization: BASIC Z3Vlc3Q6Z3Vlc3QxMjM=
  ```

- **Response Header:** A response header is an HTTP header that can be used in an HTTP response. This type of headers contains the location of the source that has

been requested by the client. Response headers, like Age, Location or Server are used to give a more detailed context of the response.

```
Age: 1030
ETag: "xyzzy"
Location: http://www.localhost:3000/
Server: Apache
```

- **Entity Header:** An entity header can be used in both HTTP requests and responses. It describes the content of the body of the message. Entity headers are Headers like Content-Length, Content-Language, Content-Encoding are entity headers.

```
Content-Encoding: gzip
Content-Language: en
Content-Length: 3495
```

3.6 HTTP Status Codes
Md Abdullah Al Mahmud

The response or status codes of HTTP directly determine the meaning of the response to an HTTP request. It determines whether a specific HTTP request has been successfully completed or not. All HTTP response status codes are separated into five classes or categories. The first digit of the status code defines the class of response, while the last two digits do not have any classifying or categorization role.

- **1xx Informational Response** – the request was received, continuing process
- **2xx Success** – the request was successfully received, understood and accepted
- **3xx Redirection** – further action needs to be taken in order to complete the request
- **4xx Client Error** – the request contains bad syntax or cannot be fulfilled
- **5xx Server Error** – the server failed to fulfill an apparently valid request

Success

- **201 (CREATED):** resource has been created
- **202 (ACCEPTED):** request accepted, but process still in progress
- **204 (NO CONTENT):** request fulfilled, and no additional content
- Otherwise: **200 (OK)**

Redirection

- **304 (NOT MODIFIED):** client can use the cached version it has of the requested resource

Client errors

- **401 (UNAUTHORIZED):** an anonymous request accesses a protected API
- **403 (FORBIDDEN):** an authenticated request hasn't enough rights to access a protected API
- **404 (NOT FOUND):** resource not found
- **409 (CONFLICT):** resource state in conflict (e.g. a user trying to create an account with an already registered email)
- **410 (GONE):** same as 404, but the resource existed
- **412 (PRECONDITION FAILED):** request tries to modify a resource that is in an unexpected state
- **422 (UNPROCESSABLE ENTITY):** request payload is syntactically valid, but semantically erroneous (e.g. a required field that has not been valued)
- **423 (LOCKED):** resource is locked
- **424 (FAILED DEPENDENCY):** requested action depended on another action that failed
- **429 (TOO MANY REQUESTS):** user sent too many requests in a given amount of time
- Otherwise: **400 (BAD REQUEST)**

Server errors

- **501 (NOT IMPLEMENTED):** server does not support the functionality required to fulfill the request
- **503 (SERVICE UNAVAILABLE):** server is currently unable to handle the request due to a temporary overload or scheduled maintenance
- **507 (INSUFFICIENT STORAGE):** server is unable to store the representation needed to successfully complete the request
- Otherwise: **500 (INTERNAL SERVER ERROR)**

3.7 Filtering and Pagination
Istiaq Hasan

As the devlopment progresses of API goes on, the resources that are returning can expand in both size and count. With the time goes on, this can lead to microservices to be under extended load and extended response time. This is why it is necessary to support filtering and pagination.

3.7.1 Filtering

It is an easier way to add primary filtering in URL to REST APIs. The query component of a URI is a normal fit for providing search criteria to a collection or store[22]. The query component can offer clients with extra interaction capabilities which include ad hoc searching and filtering[22].As a result, the query part of a URI may be clear to a REST API's client which is unlikely to the other elements of a URI[22]. It is recommended to not change the behavior of cache based on the presence or absence of a query in a given URI[22].The components of query parameters are non-hierarchical that come along with path parameters[7]. The component of the query is indicated by the first question ("?") character and ended by a number sign ("#") character or by the end of the URI. RFC 3986[7] defines the generic URI syntax as represented below:
ŪRI = scheme "://" authority "/" path ["?" query] ["#" fragment].
There query component can support a partial response[22]. It can be happened that a resource states more information than the client wants to get. In that case, a client can use query component to not get all the data rather the data the clients wishes to receives with the field parameter. In this above example,the fields parameter is introduced in the request, which states the list of fields that should be included in the response's representation.
The fields query parameter is used to state an inclusion list. The media type must specify a parameter which accepts the response's field list in case-insensitive, alphabetical order. The partial response contains only the firstName and birthDate fields.
There are other rules that are can be done via query components are mentioned in the following[22].

- selecting media type for example,
  GET /bookmarks/mikemassedotcom?accept=application/xml

- embedding link resources for example,

In this project, we have a get /appoinments endpoint which lists all the appointments. This can filter via the desired Date such as /appoinments?desiredDate=2020-01-17 which will list all the appointments for this date. This only works for exact matches. To add this kind of filtering, there are three components required.

- The field of property name.

- The operator such as equal(=), greater than(>), less than (<) etc.
- The value that needs to be filtered.

3.7.2 Pagination

API endpoints can query to huge dataset which could have the capability to return a lot of results. If the pagination is not enabled, this could cause unnecessary network traffic. Pagination helps in this regard by setting up the number of results. The client should use the query parameter to limit the results of the resource with PageSize and pageStartIndex parameters[22]. The PageSize parameter defines the maximum number of results to get in the response. The pageStartIndex parameter defines zero-based index of the first result to receive in the response. For pagination, there can be various ways.

- Using PageNumber and PageSize.
- Using StartIndex and Limit.
- Using Cursors[45].

To paginate the results, which one is followed depends on the requirements. If the results are changing, then second apporach is more reliable. If the data changes more often using StartIndex and Limit depicts a good approach. The cursors approach is noted from database systems. In this approach, the server defines the pageSize. The client can not set the position of the page to get in the response but the server lets the client in fetching the results of the resources. As a outcome, the client have to start from the first page. The PageNumber and PageSize approach is more user friendly. The client can easily understand as it is more similar what the client is seeing and requesting.The PageNumber and PageSize approach helps to keep track of the results. The client does not require to calculate the StartIndex to display the results. This is why in this project we have followed the PageNumber and PageSize approach. This following is an example on how we have done pagination in the project. In the project, we have set maximum pageSize to 50.

```
http://localhost:8092/api/appointments?pageNum=2&pageSize=2
```

In the response, the paginated hyperlinks are also included. For example,

```
"metadata": {
        "previous": "http://localhost:8092/api/appointments?pageNum=1&pageSize=2",
        "next": "http://localhost:8092/api/appointments?pageNum=3&pageSize=2",
        "first": "http://localhost:8092/api/appointments?pageNum=1&pageSize=2",
        "last": "http://localhost:8092/api/appointments?pageNum=6&pageSize=2"
    },
```

The pagination feature of collection resources enables the client to filter the response and bound the received amount of results[45]. We have introduced pagination feature in case of filtering. For example,

```
http://localhost:8092/api/appointments?desiredDate=2020-01-17&pageSize=1&pageNum=1
```

The server produces an ordered list of results which follows at least one ordering rule into a set of pages[51]. In this project, the paginated results follows based on ”createdOn” field ascending order. For example,

```
# Request
http://localhost:8092/api/appointments?pageNum=1&pageSize=2
# The partial Response is shown below
"appointment": [
        {
            "appointmentId": "app01",
            "contractorsName": "cname",
            "contractId": "C01",
            "roomNumber": "101",
            "appointmentType": "MoveIn",
            "issue": "some",
            "desiredDate": "2020-01-09",
            "status": "Received",
            "priority": "Normal",
            "createdOn": "2020-01-05",

        },
        {
            "appointmentId": "9bc34191-b144-47b6-b782-c309bdb22bec",
            "contractorsName": "Resident User 1",
            "contractId": "7b48453e-599b-424a-ba14-a9a8ae812d47",
            "roomNumber": "Room_0",
            "appointmentType": "Miscellaneous",
            "issue": "some issue",
            "desiredDate": "2020-01-16",
            "status": "Received",
            "priority": "High",
            "createdOn": "2020-01-13",
        }
    ]
```

3.8 Content Negotiation

Md Aminul Islam

Content negotiation refers to the process as a part of HTTP that make it possible to serve different representation of a resource for a request or on the basis of a request. As a result, so that user can specify which version of resource fits their capabilities the best. One classical use of this mechanism is to serve an image in Graphics Interchange Format (GIF) or Portable Network Graphics (PNG) format, so that a browser that cannot display PNG images (e.g. MS Internet Explorer 4) will be served the GIF version[23].

A resource may be represented in several different way to the viewers; for example, it might be available in different languages or different media types. One way of selecting the most appropriate choice is to give the user a selection page and let them choose the most appropriate choice; however, it is possible to automate the choice based on some selection criteria which is helpful for the development. Resources can have several types presentations, mostly because there may be multiple different user demand for different representations. Asking for a preferred representation by a client request, is referred as content negotiation.

A HTTP responses include an entity which contains guideline for interpretation by a human user. Naturally, it is suggested to supply the user with the essential entity corresponding to the request. Unfortunately, for servers and caches, not all users have the same types preferences they required and not all user agents are equally capable of managing all entity types. For that reason, HTTP has provisions for several process for "content negotiation" which is defined as the process of selecting the best representation for a given response. There can be multiple representations available[50].Any response containing an entity-body may be subject to negotiation, including error responses.

There are two type of content negotiation which are possible in HTTP: server-driven and agent-driven negotiation. Both of these two kinds of negotiation are referred as orthogonal and thus may be used separately or in combination. One method of combination, referred to as transparent negotiation. The transparent negotiation occurs when a cache uses the agent-driven negotiation information provided by the origin server. Normally, these information are provided for server-driven negotiation for subsequent requests[50].

3.8.1 Server-driven Negotiation

When the selection of the representation for a response is made by an algorithm located at the server, it is called server-driven negotiation. Selection is based on the available representations of the response and the contents of particular header fields in the request message or on other information pertaining to the request. For example, the network address of the client[50].

Server-driven negotiation is better when the algorithm for selecting from among the available representations is challenging to describe to the user agent. In addition it can be required when the server desires to send its best suggestion to the client along with the first response[50].If the suggestion is accepted it can be avoid the round-trip delay of a subsequent request. In order to improve the server's suggestion, the user agent may include request header fields (Accept, Accept-Language, Accept-Encoding, etc.) which describe its preferences for such a response[50].

3.8.2 Agent-driven Negotiation

With agent-driven negotiation, selection of the representation for a response is performed by the user end after receiving the response from the server. Selection is based on a list of the available representations of the response included within the header fields or entity-body of the response. In addition, The representation can be identified by its own URI. Selection from among the representations may be performed automatically or manually

by the user selecting from a generated menu[50].

Agent-driven negotiation performs better than the server-driven negotiation when the response would vary over commonly used when the origin server is unable to realize a user agent's ability from analyzing the request, and generally when public caches are used for load balancing and reduce network usage[50].

So, most REST API implementations depends on agent driven content negotiations. In addition, Agent driven content negotiation depends on the implementation of HTTP request headers or resource URI patterns generally[50].

Agent-driven negotiation suffers from the disadvantage of needing a second request to obtain the best alternate representation. This second request is only efficient when caching is implemented[50]. In addition, this specification does not define any technique for supporting automatic selection process, though it also does not create any problem to be developed as an extension and used within HTTP/1.1.

HTTP/1.1 defines two types of status codes for enabling agent-driven negotiation when the server is unable to provide a varying response using server-driven negotiation. The status codes are the 300 (Multiple Choices) and 406 (Not Acceptable)[50]. In agent-driven negotiation, when receiving an ambiguous request, the server sends back a page containing links to the available alternative resources. The user receives the link and choose the one to use for resources[25].

Server side negotiations is not recommended to use in most of the cases because in this negotiation, lots of assumptions needs about client request. Some assumptions like client context or how client will use the resource representation is really challenging to determine. As a result, this approach makes the server side code really complex to implement.

negotiation implementation with HTTP headers At server side, a request from the user may have an attached entity .To determine the type, server implements the HTTP request header Content-Type. Some common examples of content types are "text/plain", "application/xml", "text/html", "application/json", "image/gif", and "image/jpeg" [50].

How actually the content type is mentioned at the header of a HTTP request.

```
POST /Appointments/create_appointment HTTP/1.1
Host: localhost:8000
Content-Type: application/json
Content-Length: 1000
```

At Student Reside Service the content negation was implemented for two types of content XML and JSON. So a request could be made as XML or JSON format .As a consequence, A response can be received as XML or JSON format. Here user can mention the type of content from both end which is provides user more capacity to handle the data.

Content negotiation with URL patterns Another way to pass content type information to the server side, client may use specific extension in resource URIs. For example, a client can request for details using:

```
http://localhost:8000/appointments/appointments.XML
http:// localhost:8000/appointments/appointments.JSON
```

In above case, first request URI will return a XML response whether second request URI will return a JSON response.

3.8.3 Transparent Negotiation

Transparent negotiation is a combination of both server-driven and agent-driven negotiation. In this case, the cache is provided with a form of the list of available representations of the response (as in agent-driven negotiation) and the dimensions of variance are completely understood by the cache. Then the cache can perform the server- driven negotiation on behalf of the origin server for subsequent requests on that resource. [50].

Transparent negotiation has the capability to distribute the negotiation work that would otherwise be required of the origin server .In addition it is also possible to remove the second request delay of agent-driven negotiation when the cache is able to correctly guess the appropriate response[50].

This specification does not define any process for transparent negotiation, though it also does not prevent any such process from being developed as an extension that could be implemented within HTTP/1.1. [50].

4 Specifications

4.1 Open API

Md Aminul Islam

An open API (also referred public API) is a publicly available programming interface for application developers to get the benefit of programmatic access to a software or web service[33].

Now a days, various software application uses APIs to connect with data sources at official level or third party data services with other applications. Open API provides the opportunity to open description format for API services that is vendor neutral, portable and developer independent that service organization can also register for access to the interface[33].

OpenAPI Specification is an API description format for REST APIs. An OpenAPI file allows to describe the entire API[48]. OpenAPI specification provide information regarding about the endpoints and the operations with specific endpoints. It also provide information about input and output parameter for each endpoints.in addition OpenAPI specification provide information about authentication process and contact related information with license and terms for an applicationswaggerio.

Swagger is a set of open-source tools that can be used to design, build, document and consume REST APIsswaggerio. Swagger is the most well-known, and widely used tools for implementing the OpenAPI specification. OpenAPI specification formerly was known as Swagger Specification[48].

Open API vs Private API A private API is an interface that opens the services of an application functionality for use by developers working within a specific organization only in an internal environment[12]. Therefore, the API publishers have total control over the access of the services and the implementations In contrast to a private API, an open API allow developers, outside of an organization's workforce, to access backend data that can be used to enhance their own applications. Open APIs can significantly helpful to increase revenue without the business having to invest in hiring new developers for the development of another software tool. However, with the access of public API it is also very important to remember about a range of security and management challenges[33].

Basic structure of OpenAPI Specification OpenAPI definitions can be written in YAML or JSON. For our student residence service application YAML example was used but JSON works equally well[48]. All keyword names are case-sensitive[48]. The basic parts of specification file are described below:

- **Metadata:** API definition must include the version of the OpenAPI Specification and the overall structure of an API definition is defined based on the OpenAPI version[48].

The currently available OpenAPI versions are 3.0.0, 3.0.1, and 3.0.2; they are functionally the same[48].The info section contains API information: title, description (optional), version[48].

For Student residence service the info section in the OpenAPI specification file is

```
info:
     title: Appointment Service
      version: 1.0.0
```

- **Servers:** The servers section specifies the API server and base URL. one or several servers can be defined in the server section of OpenAPI specification file[48].
- **Paths:** The paths section defines individual endpoints (paths) in the API as well as HTTP methods (operations) supported by all endpoints[48]. For example in the OpenAPI specification file for student residence application the path section is (for one post operation):

```
paths:
/appointments:
   post:
   tags:
       - appointments
   summary: Adds a new appointment
   description: Method to add a new appointment.
   requestBody:
        description: Appointment object that needs to be added
                    required: true
                    content:
                    application/json:
                          schema:
                               $ref: '#/components/schemas/BaseAppointment'
                  application/xml:
                         schema:
                               $ref: '#/components/schemas/BaseAppointment'
```

Here, Service name appointments has been mentioned. The operation description which is requesting a new appointments has been written. The content type of the request has been mentioned. As a result, all the necessary information is regarding endpoints is available here in the specification file.

- **Parameters:** Operations can have parameters passed via URL path (/contracts/contractID), query string (/appointments? desiredDate=2020-02-04), headers or cookies .In the parameter section data types, format and if it is required or optional can also be defiened[48].

```
parameters:
        - in: query
          name: desiredDate
          description: Desired Date to filter by
```

```
          required: false
          schema:
                  type: string
                   format: date
```

- **Request Body:** When an operation includes a request body, the requestBody keyword can be used to describe the body content and media type.in the request body request body format need to be specified[48].For example – Student residence service we defined XML and JSON format request.

```
requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/UpdateAppointmentRequest'
          application/xml:
            schema:
              $ref: '#/components/schemas/UpdateAppointmentRequest'
```

- **Responses** For each operation, possible status codes can be defiened, such as 200 OK or 401 authorization error, and the response body schema[48].

```
responses:
        200:
          description: Successful operation
        401:
          $ref: '#/components/responses/AuthorizationError'
        404:
          $ref: '#/components/responses/ResourceNotFoundError'
        412:
          $ref: '#/components/responses/InvalidOperationError'
        500:
          $ref: '#/components/responses/InternalServerError'
```

- **Authentication:** There are some authentication methods which are supported[48].
  - HTTP authentication: Basic, Bearer, and so on.
  - API key as a header or query parameter or in cookies
  - OAuth 2
  - OpenID Connect Discovery

  For student residence service we used Bearer authentication which is also known as token authentication. Bearer authentication is a HTTP authentication scheme that ensure security based on tokens which are called bearer tokens. The bearer token is a cryptic string, usually generated by the server in response to a login request[48]. For student residence service ,The client must send this token in the Authorization header when making requests to protected resources as a specific user role. This

user role based on the token ensure which is the client's role and which request he/she can make. The securitySchemes and security keywords are used to define the authentication methods used in API. In the specification it should be written in this way.

```
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

4.2 RAML
Istiaq Hasan

RESTful API Modeling Language (RAML) gives a organized, clear format to describe a RESTful API. It enables to mention API, the endpoints, the HTTP methods to be applied for each one along with parameters and format[43]. RAML is sole spec designed to include the full API lifecycle in a human frinedly format[47].RAML is concise and the code can be reused. RAML offers precisely to design, build, test, document, and share API all with one spec[47].

Design API
RAML gives the benefits to design the API visually throughout the full API Design Lifecycle and tests the API and gets feedback from the user.

Design Visually
RAML ensures to describe the API in human readable format. The designer can use common words to define the API using the tools like API Workbench or API Designer. This ensures to view the structure of the API as the development work goes on. It paves the way to learn RAML more easily.

Code Reuse
RAML gives the opportunity to resue the code. By providing libraries, overlays, traits, and resourceTypes, RAML ensures the API is consistent and saves of rewriting the code.

Prototype REST API
RAML lets to prototype the API and can share the potential API across the world. RAML enusres to test the API with built in integration in popular API tools such as Postman.

Document Structure
RAML follows YAML Ain't Markup Language (YAML) 1.2 as underlying format[41]. As RAML follows YAML, keys, values, tags such kinds of nodes are case-sensitive.

The nodes that are used in the RAML specification in the following.

- **#%RAML 1.0:** The document have to start with this which defines the version of RAML that is followed.
- **title:** A plain-text and short label for the API. The value is string. Example: title: Appointment Service.
- **description:** A significan, human-friendly description of the API. Example: description: Appointment API takes care of all operations related to appointment.
- **version:** The version of the API. The value is string. Example: **version: 1.0.0**.
- **baseUri:** The URI which serves as a base of URIs for all resources. This is frequently used as the base of the URL of the API that contains at the resource location. Example: **baseUri: http://appointment-service-base-uri/api**.
- **mediaType:** The default media types that are used for request and response bodies. Example: **mediaType:** [**application/json, application/xml**].
- **securitySchemes:** The security schemes that are used within the API. RAML provides different types of security schemes for example, OAuth 1.0, OAuth 2.0, Basic Authentication, Digest Authentication, Pass Through, x-other. As we have used JSON Web Token (JWT), we have deifined security schemes as x-other.Example:

```
securitySchemes:
   bearerAuth:
     type: x-custom
     description: JWT authentication
     describedBy:
        headers:
          Authorization:
            description: X-AuthToken
            type: string
            required: true
```

- **types:** There are different data types that are used in the RAML documentation. Example:

```
BaseAppointment:
    type: object
    properties:
      contractId:
        type: string
        required: true
        description: A valid contract user id is required.
      contractorsName:
```

```
      type: string
      required: true
    roomNumber:
      type: string
      required: true
    appointmentType:
      type: array
      required: true
      items:
        enum:
          - moveIn
          - moveOut
          - miscellaneous
    issue:
      type: string
      required: true
      maxLength: 200
    desiredDate:
      type: date-only
      required: true
    priority:
      type: array
      required: true
      items:
        enum:
          - low
          - normal
          - high
xml:
  name: BaseAppointment
```

From the example, it can be seen that, in the object type we can define parameters which supports default datatypes. The data types that are supported by RAML are in the following:

- **Scalar Type:** String, Number, Integer, Boolean, Date, File, Nil.
- **Array Type:** Array type are defined by array qualifier [] or array as the value of a type facet. Array type supports the following facets to limit the behavior of the array type.
  - uniqueItems: This supports boolean value that indicates if the items in the array have to unique.
  - items: This defines the items in the array. It can be reference to a custom type or an inline type declaration.
  - minItems: The defines the minimum items the array can contain. This value must be equal to or greater than 0 and default value is 0.
  - maxItems: The defines the maximum items the array can contain. This value must be equal to or greater than 0 and default value is 2147483647.
- **Object Type:** The object type data types use the following facets in their type declarations.

    - properties: This defines the properties of this type can have.
    - minProperties: The defines the minimum number of properties allowed for instances of this type.
    - maxProperties: The defines the maximum number of properties allowed for instances of this type.
    - additionalProperties: The defines a boolean value if the object allows additional properties. The default value is true.
    - discriminator: The examines the specific type of an individual object at runtime when, for example, due to unions or inheritance, payloads contain ambiguous types. The value must match the name of one of the declared properties of a type.
    - discriminatorValue: The defines the declaring type. This requires to include a discriminator facet in the type declaration.Here the inline declaration is not allowed. The default value is name of the type.
  - **The "Any" Type:** The any type is a data type which imposes no rules, by definition. All built-in and user-defined data types has the any type at the root of its inheritance tree.
  - **Union Type:** The union type enables instances of data to be defined by any of various types. This union type is defined via a type expression that combines 2 or more types delimited by pipe (|) symbols.
  - **Using XML and JSON Schema:** RAML authorizes to use of XML and JSON schemas to describe the body of an API request or response by integrating the schemas into its data type system. The RAML processor does not enable types that declare in an XML or JSON schema to be in type inheritance or specialization, or effectively in any type expression.

    ```
    types:
      Person: !include file.json
    ```

- **/<relativeUri>:** This defines the resource of the API. This is recognised as relative URIs that begin with a slash (/). Example: **/appointments**.
- **method:** A method name for example, get,post,put etc.
- **description:** A human-friendly description what does this method allows under the specified resource.
- **queryParameters:** The query parameters contains the detailed information about any query parameters needed by this method. The query parameters are mutually exclusive with queryString. Example:

  ```
  queryParameters:
        desiredDate:
          displayName: Desired Date
          description: Date to filter by
          required: false
          type: date-only
  ```

- **Requestbody:** If the method allows body, which data types will be passed through the body at the time of request. Example:

```
post:
    body:
      application/json:
        type: BaseAppointment
```

- **responses:** The responses which are supported by the endpoint, whose key is an HTTP status code.
- **Responsebody:** This node contains the body of the response.Example:

```
200:
        description: Successful operation
        body:
          application/json:
            type: AppointmentResponse
          application/xml:
            type: AppointmentResponse
```

- **securedBy:** This node allows to apply security schemes to every method of the API. The API methods can be authenticated by the stated security schemes. The methods those have their own securedBy node are not included here. Example: **bearerAuth**, is a custom security schemes that we have created. This security schemes is being used in every endpoint except the login, register, accessToken validation endpoint.

4.3 API Blueprint

4.4 Compliance to REST

4.5 Comparison

Istiaq Hasan

We have used Open API, RAML, API Blueprint specificatons to design our APIs. All of three specifications provide some features to design the APIs but some of the feaures are not available in all the specifications. For example,

- **Support for XML contents** All of the three specifications support various type of contents for example, JSON, XML. In Open API and RAML, we can define the XML data type using schema and type respectively but in the API Blueprint case we have to manually design the XML format in the request body or response body. The code reusability feature is more extended in Open API and RAML compared to API Blueprint.

- **Security schemas** All the three specifications support different types of security schemes. In Open API and API Blueprint, we can define the bearer format in JWT which is not available in RAML. To achieve the bearer format in JWT, RAML provides x-custom type security scheme.
- **Role based Authentication** In our project, we have followed role based authentication. This feature is not available in Open API and API Blueprint but can be implementable in RAML.
- **Maximum and minimum value in query parameters** The query parameters feature is supported by all the three specifications. In API Blueprint, it is not possible to define the maximum and minimum value for a query field which is supported by both Open API and API Blueprint.
- **provide summary for HTTP response**

4.6 Tooling Support

4.6.1 Open Api

Mohammed Mostakim Ornob

Swagger Codegen

Features and Short comings

- Used to generate code stub for Open API specification.
- Able to generate code stub for both server and client.
- Able to generate code stub from both YAML and JSON formats of specification.
- For client stub user can choose between a bunch of client libraries.
- Package name is configurable for both server and client stubs.
- Supports a bunch of programming languages (e.g. Java, C#, Python, JavaScript) and frameworks (Springboot, Jax-RS Jersey, Dot Net Core, DJango).
- Supports older and latest versions (2.x & 3.x) of Open API.
- Does not generate codes to secure endpoints.
- Does not support role-based authorization.

Swagger Inspector

Features and Short comings

- A web based API testing tool.
- Can generate requests from Open API specifications.
- Support both YAML AND JSON versions of Open API specification.
- Instead being a web based tool, supports testing of endpoints hosted in local host.
- Provides option to export endpoint testing suites.
- Does not provide variables.
- Does not support auto configuring request of one endpoint with the response from another endpoint.

4.6.2 RAML
Istiaq Hasan

With RAML introducing, the focus has been moved from "documenting API " to encompassing the full API lifecycle. There are various tools available for designing, building, testing, documenting, parsing.

Design

- **API Workbench:** API Workbench[5], a rich, full-featured integrated development environment(IDE) for designing, building, testing, documenting and sharing RESTful HTTP APIs.It supports both RAML 0.8 and the recently launched RAML 1.0. RAML makes it easyto manage the whole API lifecycle.
- **API Designer:** API Designer[30] is a web-based API development tool that allows API providersto design their API quickly, efficiently, and consistently, and socialize thedesign. It consists of a RAML editor side-by-side with an embedded RAML console(the API Console).
- **Restlet Studio:** Restlet Studio[9] is a lightweight web-based development environment thataccelerates the design of APIs.
- **RAML Store:** RAML Store[8] project provides an persistent store for RAML files created using thenembedded API Designer.
- **ATOM RAML package:** ATOM RAML Package[31] is a plugin (package) for ATOM (Github's text editor)that helps the user to write RAML specs by providing highlighting capabilitiesand snippets autocompletion.

Build

- **api-generator:** Api-Generator[46] is based on PHP-code generator for Laravel framework. This enables complete support of JSON-API data format.
- **RAML for JAX-RS:** RAML for JAX-RS[28] is used to provide a set of tools to work with these technologies in a way of being able to scaffold a JAVA and JAX-RS application based on an existing RAML API definition or its roundtrip, generate the RAML API definition.
- **RAML Tools for .NET:** RAML Tools for .NET[27] enables developers to easily integrate and consume APIs which expose a RAML definition, or create a new ASP.NET Web API implementation using a contract-first approach from a previously created RAML definition.
- **raml-server:** RAML Server[15] exposes a full fake REST API with zero coding by writing a RAML spec.
- **Spring MVC - RAML Spec Synchroniser Plugin:** Spring MVC - RAML Spec Synchroniser Plugin[37] is a Maven plugin. This is designed to generate Server and Client code in Spring from a RAML API descriptor and conversely, a RAML API document from the Spring MVC Server implementation.

Testing

- **Abao:** Abao[11] provides a command-line tool to test API documentation which is written in RAML format. It offers to test API against its backend implementation.
- **Vigia:** Vigia[32] is a adaptable API integration test suite. It provides support to test generation based on a RAML documentation file.
- **Sandbox:** In Sandbox[44], it is quick and easy mock RESTful API and SOAP webservices. Sandbox generates from API definitions,instant deploy, collaborative build, and debugging tools for integration.
- **Postman:** Postman[38] is a chrome app. It provides support to import RAML documentation to easily call and test APIs.
- **RAML Mock Server:** RAML Mock Server[34] consists of Java and JUnit library which lets to run a Mock Server then confirming interactions match the API specifications defined in the RAML specification.

Document

- **raml2html:** raml2html[4] is a documentation tool. This tool generates a single HTML page which is console based on a RAML definition. This tool developed in NodeJS. It can be carried out as a command line.

- **API Console:** API Console[29] provides a graphical user interface. This helps to visualise the API's structure and important patterns. This tool provides as interactive API documentation.
- **RAMLRenderer**: RAMLRenderer[6] is a HTML documentation builder for RAML 1.0 which is written in Crystal.
- **API Notebook**: API Notebook[26] is a web-based tool which is persistent and JavaScript scripting workspace that allows live testing and exploring of APIs. This also allows saving API use cases as markdowngists, so they are versioned, forkable and shareable. This tool can be regarded as an example of literate programming.
- **ramlo:** Ramlo[35] is a command line tool. The tool is based on Node.js to generate RESTful API documentations written in RAML.

Parser

- **RAML Java Parser:** RAML Java Parser[40] is a reference implementation for Java. The tool is based on SnakeYAML. RAML Java Parser is written in Java. The parser can be added as a Java library either into a project directly or through Apache Maven. It supports both RAML 0.8 and 1.0.
- **RAML Ruby:** RAML Ruby[10] is a reference implementation in Ruby. The parser is based on PsychYAML parser. RAML Ruby is written in Ruby.It supports both RAML 0.8.
- **RAML Parser for .NET:** RAML Parser for .NET[39] is a RAML Parser implementation in .NET for all CLR languages. The parser is implemented as a strongly-typed wrapper around the JavaScript parser, influencing Edge.js as a Node.js host.
- **ramlfications-php:** ramlfications-php[49] is PHP 7 implementation of the RAML 1.0 Specification. The pareser is highly influenced from the Spotify Ramlfications design.
- **Test Compatibility Kit for RAML Parsers:** Test Compatibility Kit for RAML Parsers[42] are a collection of RAML specifications for use to test a RAML Parsers capability. The parser can be used when developing own parser.

Utilities

- **RAML-to-Swagger2-converter:** The RAML-to-Swagger-converter[14] is a tool. The tool supports converting RAML 0.8 API definition to Swagger 2.0 API definition in this case a java project.
- **swagger2raml:** Swagger2RAML[1] is an utility tool. This tool is used to generate RAML from Swagger JSON.
- **Gradle RAML Plugin:** Gradle RAML Plugin[36] validates RAML documentation.It turns the documentation into HTML. This tool is based on raml2html.

- **gulp-RAML2HTML:** Gulp-RAML2HTML plugin[2] is used to automate the process of generating HTML documentation based on a RAML API documentation.
- **php-ramlMerge:** RAML Merge[24] provides the support to merge in any included RAML files (!include) into a single RAML file via the command line.

4.6.3 API Blueprint
Md Imtiaz Abedin

For tooling API Blueprint offers backing to accomplish goals like API expansion, organization or distribution. Some of the worth mentioning tools are in the following.

- **Apiary:** Although apiary supports both API Blueprint and OpenAPI, it harvests all the power of API Blueprint and suggests the users to use it too. It allows both developer and non-developer to craft the design of an API and later implement it. Eventually, it fashions a mock API framework and produces documentation. During the design, a user stipulates the operations, input parameters for the stipulated operations and output data model. The applications can consume the mock APIs and user can find out if the API design meets the application needs. If the user feels need for modifications, it can be easily comprehended and user can even go through a couple of iterations in a really short period of time without scripting any code in an online editor.
  Alongside an intuitive online editor, Apiary provides a command line interface named Apiary CLI Gem. It lets a user endorse his API Blueprint, screening documentation, circulate it to apiary or impeccably automate Apiary in workflow. Apiary also connects with Github and user can link Apiary to a github repository with minimum effort. The choice of keeping the API Blueprint public or private depends entirely upon the user. Apiary allows multiple teams to work simultaneously where one team can work on implementation of the API and the other team can work on API consumption.
- **Drakov:** Drakov is one of the mock servers which API Blueprint backs as it's tooling support. With the use of a mock server, even a non tech person can find out how an API would function without even actually constructing anything. Early on this becomes really handy as a user can struggle over which tools to choose, what names to pick up, design conclusions and all sorts of other confusions. Drakov is a mock server which enables implementation of API Blueprint specifications. It is functioning both as individual server as well as Node module. It possesses excellent documentation and has wide variety of possible configuration settings.
- **Protagonist:** Protagonist is a parser for API Blueprint. It's a wrapper of another parser named Drafter. Drafter is a native C C++ parser. Protagonist wraps Drafter with Node js. In addition, protagonist offers validating API Blueprint with its main focus on parsing. With multiple validating and parsing configuration options already existing, developer can use the full potential of API Blueprint.
- **Aglio:** Aglio is a rendering tool for API Blueprint. It is an amazing tool which takes an API Blueprint file as input and converts the input file to HTML. Sometimes user

want's easy understandable HTML file instead of apib file. That's when rendering tool like Aglio comes to spotlight. It's blazing fast. It's default theme is already pretty charming. Nevertheless, it provides support for custom themes. With custom theme user can style the HTML pages as he wishes on the go.

- **apiary2postman:** This is a conversion tool. It can generate collection in Postman from Blueprint API markup or the Apiary API itself. This conversion tool has support only for API Blueprint or API Blueprint hosted on Apiary. In order to function fully, this tool requires Drafter 2.0 or previous installed.
- **swagger2blueprint:** This one is also a conversion tool which can convert swagger API specifications in API Blueprint specifications. In comparison to other conversion tools this one stands aside because it's new version not only supports swagger as input but also OpenAPI 3 and legacy API Blueprint also. As output format it has support for yaml, json and text/vnd.apiblueprint.

5 API Design

Mohammed Mostakim Ornob

5.1 Background

We have designed the Application Programming Interface (API) to satisfy the following use cases.

Create Resident Contract An administrator should be able to create resident contracts with the necessary information like Resident Name, Email, Room Number, Contract Dates etc. Server should validate the inputs and response with an appropriate response code for successful, invalid and failed requests. For a successful request the server should response with the created contract.

Confirm Resident Contract A resident should be able to confirm a contract created only for him/her within the confirmation period (2 weeks from the creation). The server should validate the request and act accordingly.

Extend Contract A resident should be able to extend only his/her contract before three months of the current end date. The maximum extension period is 6 months. The server should validate the request and act accordingly.

Terminate Contract A resident should be able to terminate only his/her contract in advance. The new end date should be at least 3 months later than the date of termination. The server should validate the request and act accordingly.

Retrieve Contract An admin should be able retrieve all the contracts or a particular contract. On the other hand, a resident should be able to retrieve only his/her contracts or a particular contract. User should be able to apply filter (e.g. Contractor's Name) on the resources.

Request for Appointment A resident user should be able to create appointment of different types (Move In, Move Out, Miscellaneous) against only his/her a confirmed contract. User need to provide the necessary data to create the appointment lie Contract ID Desired Date, Name, Room Number, Type, Priority, Issue etc. Server should validate the inputs and response with an appropriate response code for successful, invalid and failed requests. For a successful request the server should response with the created appointment. Move In and Move out appointments can only be created prior to the 2 weeks of contract's start and end date.

Accept/Deny Appointment A caretaker should be able to accept or deny any request for the appointment. An appointment should only be accepted or denied before the desired date.

Retrieve Appointment An admin or caretaker should be able retrieve all the appointments or a particular appointment. On the other hand, a resident should be able to retrieve all the appointments or a particular appointment requested only by him/her. User should be able to apply filter (e.g. Desired Date) on the resources.

5.2 Design

5.2.1 Services

To comply with the microservice architecture we have designed 3 services based on the business capabilities.

- Authentication Service
- Contract Service
- Appointment Service

5.2.2 Security

The endpoints are secured against any unauthenticated request. A user must authenticate him/her to perform operations for the mentioned use cases. We have 3 user-roles to control access over the endpoints.

- Administrator
- Caretaker
- Resident

5.2.3 Endpoints

We have used 3 different specification tools (Open API, API Blueprint, RAML) to design our endpoints according to the use cases.

Authentication Service

Endpoint 1 - Logs In User This endpoint is for logging in a user with valid credentials and response with an access token for successful authentication.

- **URI** - /authentication/login
- **HTTP Method** - POST
- **Authentication** - Not required
- **Request** - Supports JSON and XML

```
{
    "userId": "string",
    "password": "string"
}
```

- **Response** - Supports JSON and XML
  - **200** - Successful operation

    ```
    {
        "accessToken": "string"
    }
    ```

  - **400** - Invalid credentials

Endpoint 2 - Validates Access-Token This endpoint is for validating an user access-token. Request with a successful access-token gets the user details in response.

- **URI** - /authentication/accessToken/{access-token}/validation
- **HTTP Method** - GET
- **Authentication** - Not required
- **Path Parameter**
  - **access-token** - The *access-token* needs to be validated.
- **Response** - Supports JSON and XML
  - **200** - Successful operation

    ```
    {
        "userId": "string",
        "userName": "string",
        "userEmail": "string",
        "userType": "string",
    }
    ```

  - **400** - Invalid access-token

Endpoint 3 - Retrieves a particular User This endpoint is for retrieving a particular User identified by the User ID.

- **URI** - /users/{user-id}
- **HTTP Method** - GET
- **Authentication** - Bearer
- **Allowed Roles**
  - **Administrator**
  - **Caretaker**
  - **Resident**
- **Path Parameter**
  - **user-id** - The *User ID* of the User needs to be retrieved.
- **Response** - Supports JSON and XML
  - **200** - Successful operation

```
{
    "userId": "string",
    "userName": "string",
    "userEmail": "string",
    "userType": "string",
}
```

  - **400** - Invalid access-token

Contract Service

Endpoint 1 - Creates Contract This endpoint is for creating Contract.

- **URI** - /contracts
- **HTTP Method** - POST
- **Authentication** - Bearer
- **Allowed Roles**
  - **Administrator**
- **Request** - Supports JSON and XML

```
{
    "contractorsUserId": "string",
    "contractorsName": "string",
    "contractorsEmail": "string",
    "contractorsPhone": "string",
    "roomNumber": "string",
    "startDate": "date",
    "endDate": "date",
    "status": "string"
}
```

Allowed values for ”status” property are ”Confirmed” and ”Unconfirmed”.

- **Response** - Supports JSON and XML
  - **201** - Successful operation

```
{
    "contractId": "string",
    "contractorsUserId": "string",
    "contractorsName": "string",
    "contractorsEmail": "string",
    "contractorsPhone": "string",
    "roomNumber": "string",
    "startDate": "date",
    "endDate": "date",
    "status": "string",
    "createdBy": "string",
    "createdOn": "date"
    "links": [
        {
            "href": "string",
            "rel": "string"
        }
    ]
}
```

  - **400** - Invalid input
  - **401** - Unauthorized
  - **412** - Invalid operation

Endpoint 2 - Retrieves Contracts This endpoint is for retrieving multiple Contracts.

- **URI** - /contracts
- **HTTP Method** - GET
- **Authentication** - Bearer
- **Allowed Roles**

  - **Administrator**

- **Query Parameters**
  - **contractorsName**
    - **Required** - No
    - **Type** - string
  - **pageNum**
    - **Required** - No
    - **Type** - integer
    - **Minimum Value** - 1
    - **Default Value** - 1 (only if pageSize is present)
  - **pageSize**
    - **Required** - No
    - **Type** - integer
    - **Minimum Value** - 1
    - **Maximum Value** - 50
    - **Default Value** - 1 (only if pageNum is present)

- **Response** - Supports JSON and XML
  - **200** - Successful operation

```
{
    "contracts": [
        {
            "contractId": "string",
            "contractorsUserId": "string",
            "contractorsName": "string",
            "contractorsEmail": "string",
            "contractorsPhone": "string",
            "roomNumber": "string",
            "startDate": "date",
            "endDate": "date",
            "status": "string",
            "createdBy": "string",
            "createdOn": "date"
            "links": [
                {
                    "href": "string",
                    "rel": "string"
                }
            ]
        }
    ],
    "metadata": {
        "previous": "string",
        "next": "string",
```

```
        "first": "string",
        "last": "string"
    }
}
```

- **401** - Unauthorized

Endpoint 3 - Retrieves a particular Contract This endpoint is for retrieving a particular Contract identified by the Contract ID.

- **URI** - /contracts/{contract-id}
- **HTTP Method** - GET
- **Authentication** - Bearer
- **Allowed Roles**
  - **Administrator**
  - **Resident** - A resident user can retrieve only his/her contract.
- **Path Parameter**
  - **contract-id** - The *Contract ID* of the Contract needs to be retrieved.
- **Response** - Supports JSON and XML
  - **200** - Successful operation

```
{
    "contractId": "string",
    "contractorsUserId": "string",
    "contractorsName": "string",
    "contractorsEmail": "string",
    "contractorsPhone": "string",
    "roomNumber": "string",
    "startDate": "date",
    "endDate": "date",
    "status": "string",
    "createdBy": "string",
    "createdOn": "date"
    "links": [
        {
            "href": "string",
            "rel": "string"
        }
    ]
}
```

  - **401** - Unauthorized
  - **404** - Contract not found

Endpoint 4 - Confirms/Extends/Terminates a particular Contract This endpoint is for confirming/extending/terminating a particular Contract identified by the Contract ID.

- **URI** - /contracts/{contract-id}
- **HTTP Method** - PATCH
- **Authentication** - Bearer
- **Allowed Roles**
  - **Resident** - A resident user can confirms/extends/terminates only his/her contracts.
- **Path Parameter**
  - **contract-id** - The *Contract ID* of the Contract needs to be confirmed/extended/terminated.
- **Request** - Supports JSON and XML

```
{
    "operation": "string",
    "newEndDate": "date"
}
```

  Allowed values for "operation" property are "Confirm", "Extend" and "Terminate". "newEndDate" property is not applicable for *Confirm* operation.

- **Response**
  - **200** - Successful operation
  - **400** - Invalid inputs
  - **401** - Unauthorized
  - **404** - Contract not found
  - **412** - Invalid Operation

Endpoint 5 - Retrieves Contracts of a particular Contractor This of endpoint is for retrieving Contracts of a particular Contractor.

- **URI** - /contractors/{contractors-user-id}/contracts
- **HTTP Method** - GET
- **Authentication** - Bearer
- **Allowed Roles**
  - **Administrator**

  - **Resident** - A resident user can retrieve only his/her contracts.

- **Path Parameter**
  - **contractors-user-id** - The *User ID* of the Contractor of whom the Contracts needs to be retrieved.
- **Response**
  - **200** - Successful operation

```
[
    {
        "contractId": "string",
        "contractorsUserId": "string",
        "contractorsName": "string",
        "contractorsEmail": "string",
        "contractorsPhone": "string",
        "roomNumber": "string",
        "startDate": "date",
        "endDate": "date",
        "status": "string",
        "createdBy": "string",
        "createdOn": "date"
        "links": [
            {
                "href": "string",
                "rel": "string"
            }
        ]
    }
]
```

  - **400** - Invalid inputs
  - **401** - Unauthorized
  - **404** - Contractor not found

Appointment Service

Endpoint 1 - Creates Appointment This endpoint is for creating Appointment.

- **URI** - /appointments
- **HTTP Method** - POST
- **Authentication** - Bearer
- **Allowed Roles**
  - **Resident** - A resident user can create appointment only for his/her contracts.

- **Request** - Supports JSON and XML

```
{
    "contractId": "string",
    "contractorsName": "string",
    "roomNumber": "string",
    "appointmentType": "string",
    "issue": "string",
    "desiredDate": "date",
    "priority": "string"
}
```

  Allowed values for "appointmentType" property are "Move In", "Move Out" and "Miscellaneous". Allowed values for "priority" property are "Low", "Normal" and "High".

- **Response** - Supports JSON and XML
  - **201** - Successful operation

```
{
    "appointmentId": "string",
    "contractId": "string",
    "contractorsName": "string",
    "roomNumber": "string",
    "appointmentType": "string",
    "issue": "string",
    "desiredDate": "date",
    "priority": "string"
    "status": "string",
    "createdBy": "string",
    "createdOn": "date"
    "links": [
        {
            "href": "string",
            "rel": "string"
        }
    ]
}
```

  - **400** - Invalid input
  - **401** - Unauthorized
  - **412** - Invalid operation

Endpoint 2 - Retrieves Appointments This endpoint is for retrieving multiple Appointments.

- **URI** - /appointments
- **HTTP Method** - GET

- **Authentication** - Bearer
- **Allowed Roles**
  - **Administrator**
  - **Caretaker**
- **Query Parameters**
  - **desiredDate**
    - **Required** - No
    - **Type** - date
  - **pageNum**
    - **Required** - No
    - **Type** - integer
    - **Minimum Value** - 1
    - **Default Value** - 1 (only if pageSize is present)
  - **pageSize**
    - **Required** - No
    - **Type** - integer
    - **Minimum Value** - 1
    - **Maximum Value** - 50
    - **Default Value** - 1 (only if pageNum is present)
- **Response** - Supports JSON and XML
  - **200** - Successful operation

```
{
    "contracts": [
        {
            "appointmentId": "string",
            "contractId": "string",
            "contractorsName": "string",
            "roomNumber": "string",
            "appointmentType": "string",
            "issue": "string",
            "desiredDate": "date",
            "priority": "string"
            "status": "string",
            "createdBy": "string",
            "createdOn": "date"
            "links": [
                {
                    "href": "string",
                    "rel": "string"
                }
            ]
```

```
            }
        ],
        "metadata": {
            "previous": "string",
            "next": "string",
            "first": "string",
            "last": "string"
        }
    }
```

- **401** - Unauthorized

Endpoint 3 - Retrieves a particular Appointment This endpoint is for retrieving a particular Appointment identified by the Appointment ID.

- **URI** - /appointments/{appointment-id}
- **HTTP Method** - GET
- **Authentication** - Bearer
- **Allowed Roles**
  - **Administrator**
  - **Caretaker**
  - **Resident** - A resident user can retrieve only his/her appointment.
- **Path Parameter**
  - **appointment-id** - The *Appointment ID* of the Appointment needs to be retrieved.
- **Response** - Supports JSON and XML
  - **200** - Successful operation

```
{
    "appointmentId": "string",
    "contractId": "string",
    "contractorsName": "string",
    "roomNumber": "string",
    "appointmentType": "string",
    "issue": "string",
    "desiredDate": "date",
    "priority": "string"
    "status": "string",
    "createdBy": "string",
    "createdOn": "date"
    "links": [
        {
            "href": "string",
```

```
                "rel": "string"
            }
        ]
    }
```

  - **401** - Unauthorized
  - **404** - Contract not found

Endpoint 4 - Accepts or Denies a particular Appointment This endpoint is for accepting or denying a particular Appointment identified by the Appointment ID.

- **URI** - /appointments/{appointment-id}
- **HTTP Method** - PATCH
- **Authentication** - Bearer
- **Allowed Roles**
  - **Caretaker**
- **Path Parameter**
  - **appointment-id** - The *Appointment ID* of the Appointment needs to be accepted or denied.
- **Request** - Supports JSON and XML

  ```
  {
      "operation": "string"
  }
  ```

  Allowed values for ”operation” property are ”Accept” and ”Deny”.
- **Response**
  - **200** - Successful operation
  - **400** - Invalid inputs
  - **401** - Unauthorized
  - **404** - Contract not found
  - **412** - Invalid Operation

Endpoint 5 - Retrieves Appointments of a particular Contractor This of endpoint is for retrieving Appointments of a particular Contractor.

- **URI** - /contractors/{contractors-user-id}/appointments
- **HTTP Method** - GET

- **Authentication** - Bearer
- **Allowed Roles**
  - **Administrator**
  - **Caretaker**
  - **Resident** - A resident user can retrieve only his/her appointments.
- **Path Parameter**
  - **contractors-user-id** - The *User ID* of the Contractor of whom the Appointments needs to be retrieved.
- **Response**
  - **200** - Successful operation

```
[
    {
        "appointmentId": "string",
        "contractId": "string",
        "contractorsName": "string",
        "roomNumber": "string",
        "appointmentType": "string",
        "issue": "string",
        "desiredDate": "date",
        "priority": "string"
        "status": "string",
        "createdBy": "string",
        "createdOn": "date"
        "links": [
            {
                "href": "string",
                "rel": "string"
            }
        ]
    }
]
```

  - **400** - Invalid inputs
  - **401** - Unauthorized
  - **404** - Contractor not found

5.3 Specification Support

URI All 3 specifications support URI with parameters.

- **Open API**

```
paths:
    /contracts/{contract-id}:
```

- **API Blueprint**

```
## Contracts By Contract Id [/contracts/{contract_id}]
```

- **RAML**

```
/contracts:
    ...
        ...
    /{contract-id}:
```

HTTP Method All 3 specifications support all the standards HTTP methods.

- **Open API**

```
paths:
    /contracts:
        get:
        ...
        post:
        ...
    /contracts/{contract-id}
        get:
        ...
        put:
        ...
        patch:
        ...
        delete:
```

- **API Blueprint**

```
## Contracts [/contracts]
### Retrieves contracts [GET]
...
## Contracts [/contracts]
### Retrieves contracts [POST]
...
## Contracts By Contract Id [/contracts/{contract_id}]

+ Parameters
    + contract_id (string, required)

        ID of the contract to be retrieved
```

```
### Retrieves a contract [GET]
...
## Contracts By Contract Id [/contracts/{contract_id}]

+ Parameters
    + contract_id (string, required)

        ID of the contract to be updated

### Updates (put) a contract [PUT]
...
## Contracts By Contract Id [/contracts/{contract_id}]

+ Parameters
    + contract_id (string, required)

        ID of the contract to be updated

### Updates (patch) a contract [PATCH]
...
## Contracts By Contract Id [/contracts/{contract_id}]

+ Parameters
    + contract_id (string, required)

        ID of the contract to be deleted

### Deletes a contract [DELETE]
```

- **RAML**

```
/contracts:
  get:
  ...
  post:
  ...
  /{contract-id}:
    get:
    ...
    put:
    ...
    patch:
    ...
    delete:
```

Authentication Open API and RAML support different types of security schemes (e.g. Bearer). API Blueprint does not support any. Security can only be mentioned in description.

- **Open API**

```
securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

- **API Blueprint**

```
## Authentication
This API uses JSON Web Tokens for its authentication.

The parameters that are needed to be sent for this type of authentication are
as follows:
+ ‘JWTHeaderName‘
+ ‘JWTAcquireURL‘
+ ‘JWTDestroyURL‘
```

- **RAML**

```
securitySchemes:
   bearerAuth:
     type: x-custom
     description: JWT authentication
     describedBy:
        headers:
          Authorization:
            description: X-AuthToken
            type: string
            required: true
```

Roles RAML explicitly supports roles allowed for the endpoints. On the other hand, description is the only option for Open API and API Blueprint to mention the allowed roles.

- **Open API**

```
paths:
  /contracts:
    post:
    ...
      description: Method to add a new contract.
      Users only with Admin role can invoke this method.
```

- **API Blueprint**

```
Methods for contract resources

## Contracts [/contracts]

### Adds a new contract [POST]
Method to add a new contract.
Users only with Admin role can invoke this method.
```

- **RAML**

```
...
types:
  Roles:
    type: array
    items:
      enum: [ ADMINISTRATOR, CARETAKER, RESIDENT ]
...
/contracts:
  POST:
    (access_permission):
      roles: [ ADMINISTRATOR ]
```

Path Parameter All 3 specifications support path parameters.

- **Open API**

```
paths:
    /contracts/{contract-id}:
        get:
            parameters:
            - in: path
              name: contract-id
              description: ID of the contract to be retrieved
              required: true
              schema:
                type: string
```

- **API Blueprint**

```
## Contracts By Contract Id [/contracts/{contract_id}]

+ Parameters
    + contract_id (string, required)

        ID of the contract to be retrieved


### Retrieves a contract [GET]
```

- **RAML**

```
/contract:
    /{contract-id}:
        uriParameters:
            contract-id: string
        get:
```

Query Parameter All 3 specifications support query parameters. But API Blueprint and RAML does not support *Format*, *Maximum* and *Minimum* values.

- **Open API**

```
paths:
    /contracts:
        /get:
            parameters:
            - in: query
              name: contractorsName
              description: Contractors name to filter by
              required: false
              schema:
                type: string
            - in: query
              name: pageNum
              description: Page number for the pagination.
              required: false
              schema:
                type: integer
                format: int32
                minimum: 1
                default: 1 (only if pageSize is present)
            - in: query
              name: pageSize
              description: Page size for the pagination.
              required: false
              schema:
                type: integer
                format: int32
                minimum: 1
                maximum: 50
                default: 20 (only if pageNum is present)
```

- **API Blueprint**

```
## Contracts [/contracts{?contractorsName,pageNum,pageSize}]

### Retrieves contracts [GET]
```

```
Method to retrieve multiple contracts.
Users only with Admin role can invoke this method.
+ Parameters
    + contractorsName (string, optional)

        Contractors name to filter by

    + pageNum (number, optional) -

        Page number for the pagination.

        + Default: 1 (only if pageSize is present)
    + pageSize (number, optional) -

        Page size for the pagination.

        + Default: 20 (only if pageNum is present)
```

- **RAML**

```
/contracts:
    get:
        queryParameters:
          contractorsName:
            description: Contractors name to filter by.
            required: false
            type: string
          pageNum:
            description: Page number for the pagination.
            required: false
            type: integer
            default: 1 #only if pageNum is present
          pageSize:
            description: Page size for the pagination.
            required: false
            type: integer
            default: 20 #only if pageSize is present
```

Request All 3 specifications support Request body with different types of contents (e.g. JSON, XML). These specifications also support defining schemas/data structures/types for requests and responses. Open API and RAML support extending schemas/types to reuse properties. Schema/Data structure/Type property supports different data-types including object, array, date , enum etc.

- **Open API**

```
...
requestBody:
```

```
      description: Contract object that needs to be added
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/BaseContract'
        application/xml:
          schema:
            $ref: '#/components/schemas/BaseContract'
...
schemas:
BaseContract:
  type: object
  properties:
    contractorsUserId:
      type: string
      description: A valid resident user id is required
    ...
    status:
      type: string
      enum:
        - Confirmed
        - Unconfirmed
  required:
    - contractorsUserId
    ...
    - status
  xml:
    name: contract
```

- **API Blueprint**

```
+ Request (application/json)

    + Attributes (BaseContract)
...
# Data Structures

## BaseContract (object)


### Properties
+ `contractorsUserId` (string, required) - A valid resident
user id is required
...
+ `status` (enum[string], required)
    + `Confirmed`
    + `Unconfirmed`
```

- **RAML**

```
body:
    application/json:
        type: BaseContract
    application/xml:
        type: BaseContract
...
types:
  BaseContract:
    type: object
    properties:
      contractorsUserId:
        type: string
        required: true
        description: A valid resident user id is required.
      ...
      status:
        type: array
        description: contract status
        items:
          enum:
            - confirmed
            - unconfirmed
    xml:
      name: BaseContract
```

Response All 3 specifications support response with different response code and content types.

- **Open API**

```
paths:
  /contracts:
    post:
        responses:
        200:
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ContractResponse'
            application/xml:
              schema:
                $ref: '#/components/schemas/ContractResponse'
        400:
          $ref: '#/components/responses/ValidationFailedError'
```

- **API Blueprint**

```
## Contracts [/contracts]

### Adds a new contract [POST]
Method to add a new contract. Users only with Admin role can
invoke this method.
...
+ Response 200 (application/json)

        Successful operation

    + Attributes (ContractResponse)

+ Response 400

        Validation failed.
```

- **RAML**

```
/contracts:
    post:
        responses:
            200:
                description: successful operation
                body:
                  application/json:
                    type: ContractResponse
                  application/xml:
                    type: ContractResponse
            400:
                description: Validation failed.
```

6 Archietecture

Mohammed Mostakim Ornob

6.1 System Architecture

A good and sustainable architecture is the backbone of any system for it's sustainability over the long term evaluations. Architecture defines the structure and behaviour of a system.

Monolithic and Microservices are two most dominating architectures for defining a system. Monolithic architecture is the part of the software system development from it's very beginning. On the other hand, Microservices architecture is much newer architectures that surfaced as a concept several years back and has already captured a very big chunk of market share. The concept of Microservices architectures was developed to support the large systems which became complex in regard of development and maintenance over the continuous development cycles. Choosing between the Microservices architecture and the Monolithic architecture requires the evaluation of the business model and future complexity. After evaluating the business model and future complexity of *Student Residence Application* we have chosen to with the Microservices architecture. For our solution we have divided the system into three (3) different microservices.

- **Authentication Service** - is used to authenticate a particular user with the provided credentials and validate the logged in users. This service also provides data of particular users.
- **Contract Service** - facilitates the user to create new contracts and apply different operations (e.g. Confirm, Extend, Terminate) on the created contracts.
- **Appointment Service** - facilitates user to create different types of appointments (e.g. Move In, Move Out, Miscellaneous) against a confirmed contract and apply different operations on the created appointments.

6.1.1 What is Microservices architecture?

Microservices architecture is a architectural pattern and a variant of *Software Oriented Architecture* that divides a single service into several loosely coupled independent services which can communicate among themselves.[53]

Unlike the huge single service of Monolithic architecture, in microservices architecture the core of a system is broken into multiple granular microservices based on the business capabilities. These services are decentralized and independently deployable. They should be capable of communicating among themselves using a common and lightweight protocol (e.g. HTTP, RPC). The services may be developed with different frameworks and also written in different programming languages. There is no limitation on the number of services. But the services must be organized around the business capabilities and service oriented.

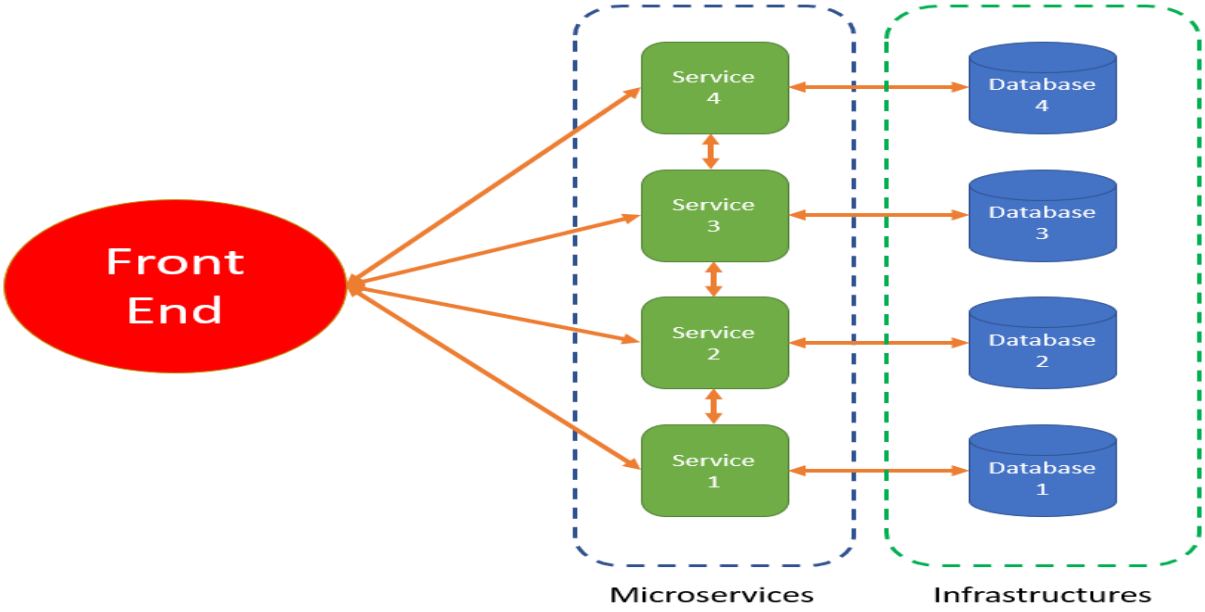

Figure 1: Microservices Architecture

6.1.2 History of Microservices architecture

The term *"Microservices"* were first discussed in 2011. Though the idea had been in practice with several industry experts for a while. It was same time the industry was leaning towards the REST services from SOAP services. REST was a perfect lightweight protocol to be implemented in the Microservices architecture. The industry leaders like Amazon, Netflix, Spotify, Uber have implemented into their system.[21]

6.1.3 Characteristics of Microservices architecture[21]

Microservices architecture possesses some unique characteristics.

- **Componentization via Services** - Every service is a independently replaceable and upgradeable component. One component can not have other component as library. They should be capable of communicating with each other through *Web Request* or *Remote Procedure Call*.
- **Organized around Business Capabilities** - While dividing a large system into microservices, the focus should be on the business capability not the technology stack. With the development of microservices the business capabilities gets modular thus being simple to maintain.
- **Products not Projects** - Instead of as a *Project*, a service should be developed as a *Product*. A service should be able to act as a product so that the services all together can act as a collection of products.
- **Smart endpoints and dumb pipes** - The client facing API should support well choreographed REST protocols with proper response and messages. On the other hand, the communication within the services can use light weight message bus services (e.g. RabbitMQ).

- **Decentralized Governance** - Decentralized governance means choosing appropriate and independent technology stack for each service. Each service can be developed on the different technology stacks. The deployment techniques and procedures can also be different.
- **Decentralized Data Management** - Every microservice should have their own database. Though a service can access multiple databases, multiple services should not share a single database.
- **Infrastructure Automation** - Microservices escalates the complexity of building, deploying and operation. To minimize this problem infrastructures should be automated. Services may be configured with automated tests and continuous delivery mechanism. Orchestrating the services using different tools (e.g. Docker Compose, Kubernetes) can be good option.
- **Design for failure** - A depended service should be designed to handle the unavailability and failure of the services it is depending on. The unavailability or failure of a service on the second layer should be handled properly with the proper message to the end user.
- **Evolutionary Design** - A microservices should be designed in a manner so that it can be evolved over time according to the business and technology changes. Migrating from a Monolithic to a Microservices architecture should evolve over time. The Monolithic structure can still be there with new microservices added as new features or value added services.

6.1.4 Why Microservices architecture?

Microservices is an ever growing architectural style since it's introduction. It has several benefits over the Monolithic architecture.

- **Vertically scale-able** - Microservices architecture is vertically scale-able in compare to Monolithic architecture which can be scaled horizontally. In Microservices architecture, a microservice which starts to get more complex and tough to maintain can be broken into several new microservices.
- **Less vulnerable to changes** - As the development and deployment of microservices are completely independent of each other and as long as the communication interface is unchanged, the chance for a microservice to be affected by the changes of another microservice is very low. This allows faster testing and deployment.
- **Allows different development approaches** - Microservices development approaches can vary from each other. Like, the service services can developed with different frameworks and written in different programming languages as well. This facilitates the development team to use the best matched programming language and frameworks for the development of a particular service.
- **Easily maintainable** - Loosely coupled and independent microservices make the maintenance easy. A portion of the system can be entirely brought down without hampering the other services.

- **Reduced business complexity** - Though the architecture itself seems to be a complex one, it reduces the complexity of the whole system. As the business logic is distributed, it reduces the business complexity for the services. It also reduces the development complexity..

6.2 Application Archietecture

A good application is always well architectured. Architecture of an application deals with the concerns beyond algorithm and data structures. Architecture defines some key factor of an application.[19] Such as -

- Control Structure
- Communication protocol
- Synchronization
- Data access

6.2.1 What is Application Architecture?

The architecture of an application defines it's fundamental structure. It also defines how an application should communicate with the end users and infrastructures (e.g. database). A clean and balanced architecture is key any application's sustainability. Application architecture solves the common problems related to software architecture. Application architecture is similar to software architecture but has a broader scope.[55] There are several widely used architectural patterns. Such as -

- Pipes and filters
- Client-server
- Component based
- Data-centric
- Event driven
- Layered

6.2.2 Hexagonal Architecture

We have implemented the Hexagonal Architecture in our services. Hexagonal architecture is an implementation of *Layered* architecture. It defines the conceptual layers and their responsibilities. Concept of this pattern was developed by *Alistair Cockburn.*[3]

Though the pattern is called *Hexagonal*, the number of the sides depends on the points of contact. Each side represents a communication adapter and port. The main concept of

this architecture is driven by the core of the application which holds the entire business logic of the application. The core is surrounded by the sides of the hexagon which are usually the layers responsible for communication and act as agents between the core and the end user/infrastructures. Each layer has its own responsibility. The core layer is responsible to hold the entire business logic of the application and applies the logic accordingly. The core layer does not communicate with the end user/infrastructures. On the other hand the outer layers act as the delegates of the core layer and responsible to communicate through the adapters and ports with end users, services and infrastructures (e.g. database). These layers feed the inputs from the end users and infrastructures to the core layer and convey the responses from the core layer to the end users. The layers responsible for the outer communication do not implement the business logic.

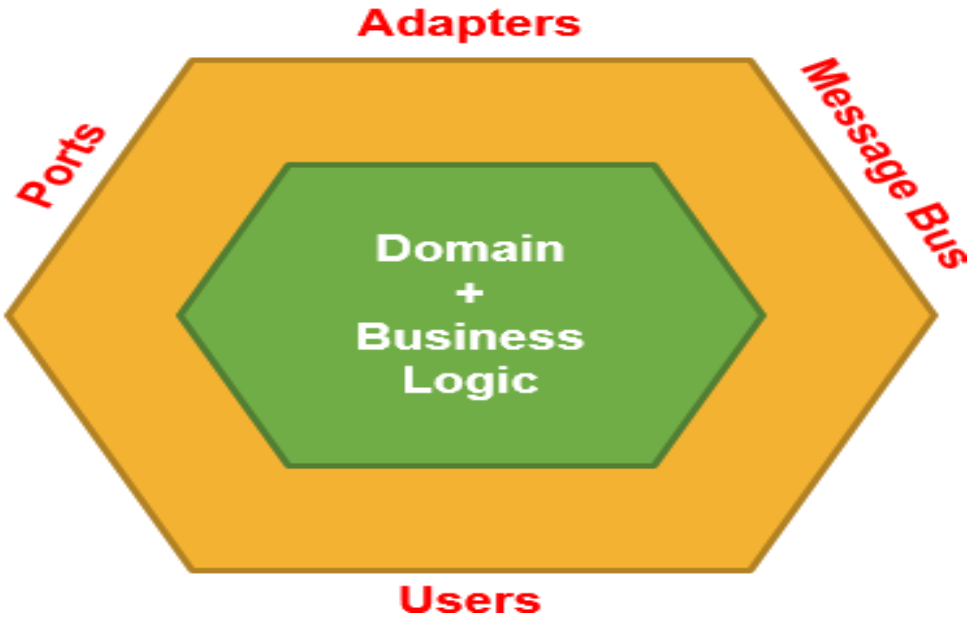

Figure 2: Hexagonal Architecture

6.2.3 Layers in the service

We have designed our services to have 3 different layers. One (1) is the core layer and other two (2) is for communication through adapters and ports.

- Service Layer - This the core layer of the application. It holds the entire business logic of the application. This layer is responsible for validating all the the input data and act accordingly. Service layer takes user input from the web layer and communicates with the database using the data-access layer.
- Web Layer - This layer is responsible for taking the input from the user and serving the response to the end user which it collects from the service layer. Web layer is also responsible for implementing the representational logic. Such as appending HAL links and metadata to the response.
- Data-access Layer - This layer act as an communication agent between service layer and database. This layer holds the necessary database queries and execute on the demand from the service layer.

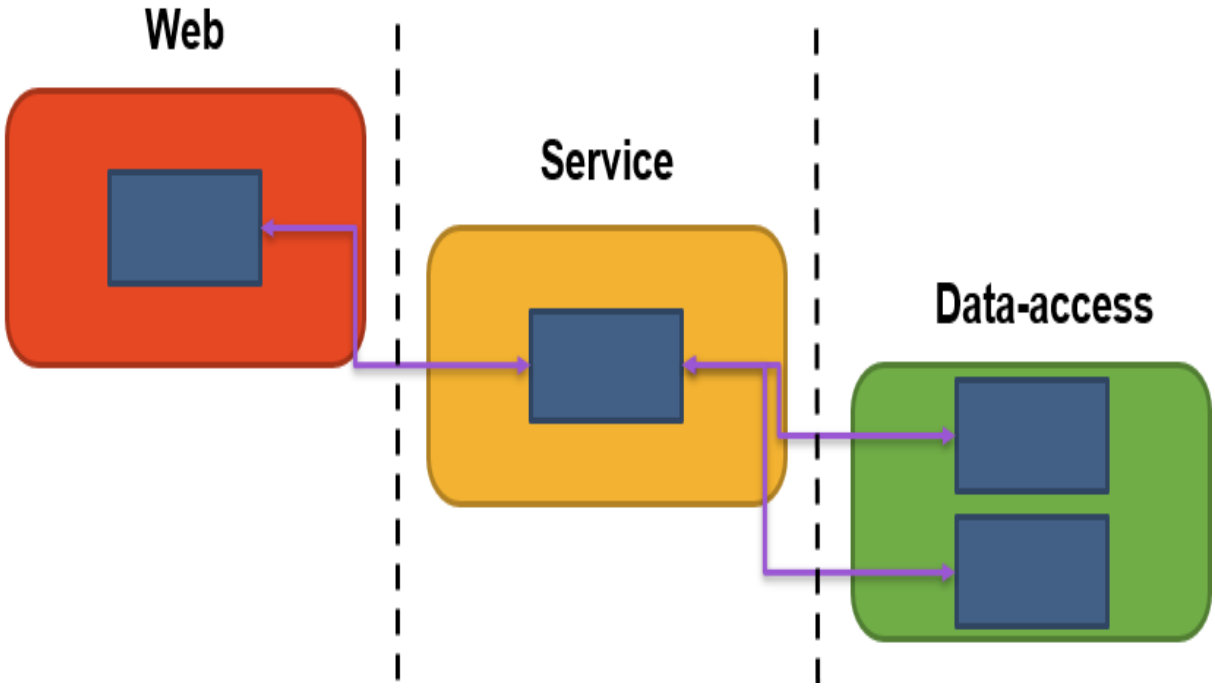

Figure 3: Application Layers

6.2.4 Framework

There are tons of frameworks from different programming languages for developing REST API. Such as-

- Springboot with JAVA
- JAX-RS Jersey with JAVA
- DJango with Python
- Dot Net Core with C#
- NodeJs with JavaScript
- Laravel with PHP

6.2.5 JAX-RS Jersey

We have chosen JAX-RS Jersey with version 2.0 for developing out REST API services. JAX-RS Jersey framework is developed collectively by Oracle Corporation and Eclipse Foundation[52]. It is a lightweight framework specially developed to facilitate REST API development. It supports all the basic needs of a REST API.[13] Such as -

- Routing
- HTTP methods
- Response codes

- Headers
- Request body
- Path and Query parameters
- Content negotiation
- Security

JAX-RS Jersey also supports JEE annotation for routing, specifying resources etc.[13] Some of the common annotation supported by JAX-RS Jersey are -

- @Path
- @GET, @PUT, @POST, @DELETE, @PATCH
- @Produces, @Consumes
- @PathParam
- @QueryParam

7 Implementation

7.1 Clean Code Convention

Md Imtiaz Abedin

Clean code is idiosyncratic concept and each developer has a individual interpretation of it. Although there is some established concept within the software industry which are contemplated as best exercise and what may be called clean code or not. But there exists no conclusive difference. Clean code has two fundamental concepts.

- Easy to understand
- Easy to change

Understandability refers to ease of reading of the code. The ownership of the code is not a considerable matter in this case. It depicts lessening the need for estimating and probability for confusions. Specifically, easy understandability should extend to

- The execution stream of complete application.
- How dissimilar parts network with each other.
- Function and accountabilities of each class.
- What accomplishment each method has.
- Resolution of expressions and variables

Easy to change terms towards better extension ability and refactoring opportunity. This way it becomes easy to sort out and fix bugs inside code. This is attainable if a developer who is trying to alter the code apprehends it and stays assertive that his change will not break the existing system. In order to achieve easy to change ability the codebase should have:

- Small and precisely responsible class and method
- Clear and concise APIs
- Anticipatable classes and methods which work as estimated
- Code with unit test and ease of testing procedure
- Tests are understandable with no complexity and available ease of changing of those tests.

Why Clean Code Developers should give attention to clean code since code is certainly not scripted just on one occasion and then overlooked. More often than not the same developer or somebody else need to give his time on that code. And to that end, if the developer needs efficient use of his time, he needs to apprehend that code. As other developers need to understand another one's code, it roughly translates to developers inscribing code not only for computer but also for human. Clean code helps the developer himself in future and also his colleagues. It massively lessens the cost of maintenance for the application. Because it makes it simpler to approximate the time desired for new functionalities. Clean code makes the whole coding experience far better for foreseeable future. A lot of developer's life will go smooth with use of another developer's clean code. The whole lifecycle of the application eventually gets affected by use of clean code.

Our approach to clean code We have adopted Inversion of Control (IoC) as a mean of clean coding practice in this project scope. Inversion of control is a design pattern which according to the name indicates reverses controls in OOP design to accomplish loose connection. In this context controls denote to supplementary tasks a class has like control over the stream of data of an application or maybe over the flow of object construction or object formation based on demand or binding of objects. IoC provides a high-level design structure but it does not pass any execution details. Developers can implement IoC principle as they seem fit.

Classes in object-oriented principle should follow loose coupling architecture. This depicts when the developer changes one of the classes, the changed class should not force any other class to change. The purpose of this principle is to maintain code and keep the codebase extendable. Towards this goal from time to time software design architect and OOP researchers have created patterns over time which inherits the IoC principle. Among the

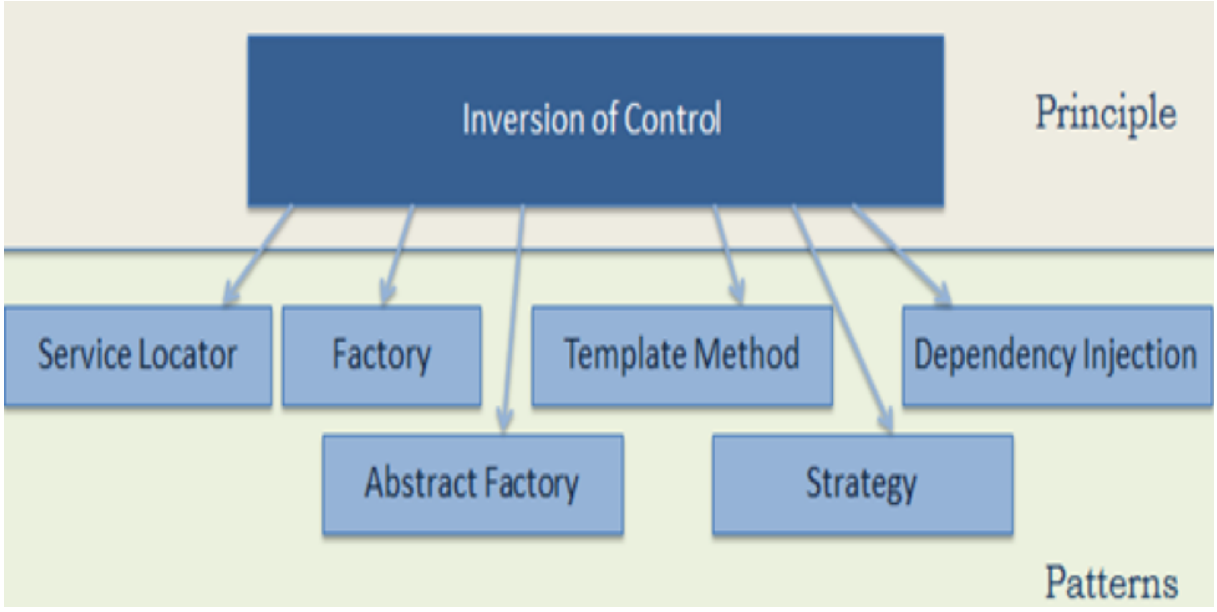

Figure 4: IoC and its patterns

available patterns we have gone with Dependency Injection (DI) as it seems to offer best possible improvement on our codebase.

Why Dependency Injection Dependency injection is a programming method that puts a class in a position where its independent of its dependencies. The independency comes by decoupling the practice of an object from its instantiation.

- Reusability is one of the most sought features inside a project's code. The most important impact of dependency injection is cultivating code reusability.
- Lessening frequency of changing a class.
- Enables developer to substitute dependencies without altering the class that manipulates them.
- Risk factor that developer must modify a class just because one of its dependencies has been changed is largely reduced.

Dependency injection technique has been implemented by a lot of modern application frameworks. All these frameworks offer the methodological parts of the procedure so that developer himself can emphasis on the execution of business logic. Some most used example of frameworks providing dependency injection.

- Spring
- Guice
- Weld
- Dagger
- Play

We have picked Guice as its wide adoptability and as google itself is behind this. Guice has excellent documentation and guide for developers.

The Technique We have used interfaces to discontinue the dependencies in between higher and lower level classes. Once that's done, both classes keep dependent on the interface and no more to each other. Implementing this principle expands the reusability of code and at the same time restricts the ripple effect in case developer needs to alter lower level classes. But there still remains a dependency on the lower level class even after it's executed flawlessly. What the interface does is detaching the usage of lower level class, but the instantiation is still not independent. At some point inside the code, developer needs to instantiate implementation of the interface because this is going to avert developer from substituting the implementation of the interface with another one. Dependency injection method eyes to remove this reliance by untying the usage from the instantiation of the object. This lessens quantity of boilerplate code and advances elasticity. Now this technique requires classes that can accomplish four elementary roles.

- Service cherished.
- Client which works on the desired service.

- Interface which is manipulated by the client and implemented by the service.
- Injector that's generating a service instance and inserting it into the client.

For our implementation we bounced the interface part and injected the service object straight into the client. In reality, this conflicts with the dependency inversion principle cause now the client has a plain dependency on the service class. To keep up with the time constraint and simplicity of the codebase we went on with this. But most often, it's better to write an interface to eliminate the reliance between the client and service implementations. Dependency inversion principle doesn't require the injector role to be essentially used. But that did not make it a problem for us cause the framework which we have used (Google Guice) comes equipped with implementation of it. For ease of developer google has implemented a code first tactic for dependency injection and management for Guice so that developers don't have to hurdle with huge number of XML. While we have used Guice provided @Inject annotation for injection purpose in our code, it's worth mentioning that Guice supports javax.inject.Inject annotations also.
For instantiating implementation of the interfaces Guice comes shipped with an exciting feature which is called Just-in-time binding. One catch for this feature is there can be no classes which comes with argument containing constructor. For just in time binding to work the implementation of the selected interfaces needs to possess default constructor (no-arg). Binding works the same way in Guice like wiring works in Spring. A binding is

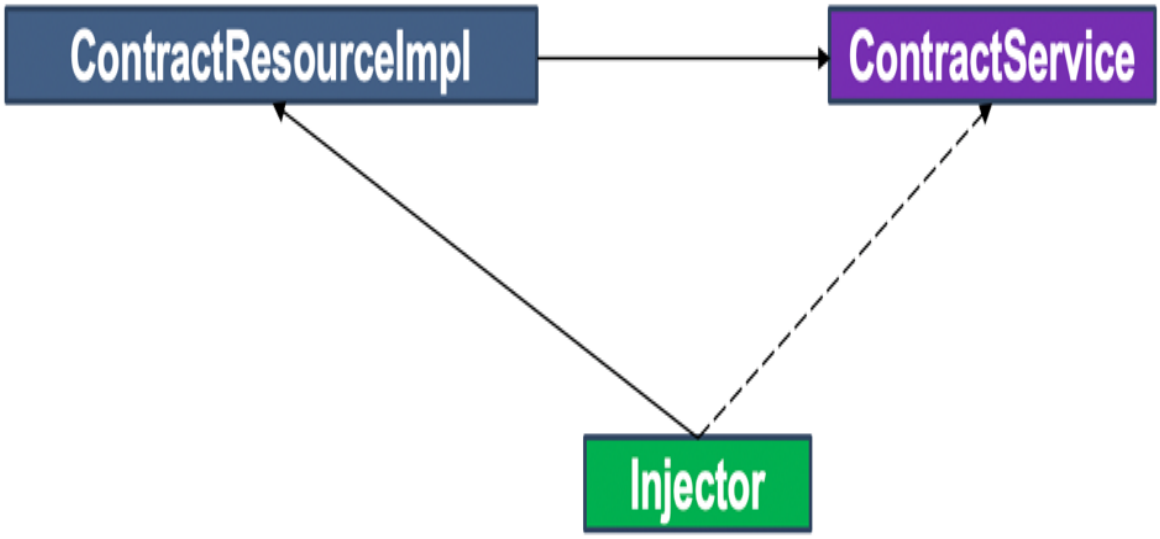

Figure 5: DI in our project scope

constructed in an implementation of com.google.inject.AbstractModule. For instance, in our implementation of binding one of the implementations was binding ContractServiceImplementation with ContractService which ultimately translates to whenever a variable of ContractService is found inside the code, an instance of ContractServiceImplementation has to be injected.

In our experience, aside from all the benefits of dependency injection we had one specific hurdle with it. Because dependency injection is a run time concept instead of compile

time, there's no way developer will see regular foreseeable error before actual application run. Even during the actual run it was difficult in regular way to debug because of the way dependency injection works.

7.2 Authentication and Authorization
Md Imtiaz Abedin

7.2.1 Authentication

JWT is an already established standard which outlines a solid and self-reliant mean for securely conveying data between receiver and sender in form of JSON object. Upon transmission the data can be confirmed and reliable as it's digitally signed. Using undisclosed (**HMAC algorithm**) or a public/private key pair it is possible to sign JWTs.

Why JWT
The most used purpose of JWT is authorization. When a user gets logged in for the first time, a token is issued for him. After that initial login all consecutive request sent from the user to server will contain the previously issued token which permits the user to call resource, service, route of the server. So, all this user privileges come bundled with the issued JWT. Another widely used implementation of JWTs is Single Sign On functionality but that is out of scope for our project. In addition to JWT there are two other kind of security token available to consider named Simple Web Token (SWT) and Security Assertion Markup Language (SAML) Tokens.

Extensible Markup Language (XML) consists way more characters than JSON. As a result, when JSON gets encoded, the total size of the encoding is definitely smaller in JSON. This instantly results in producing more compressed encoding in JWT than SAML. The compressed size of encoding gives JWT the edge while manipulating HTML and HTTP atmospheres.

With regards to security, it is possible to sign SWT with HMAC algorithm using shared secret key. Nonetheless, for signing purpose X.509 certificate in disguise of public/private key pair can be used for JWT and SAML tokens. Simplicity of signing JSON comes into consideration when XML signing is a necessity. XML signing requires XML Digital Signature and this procedure more often than not introduces security holes in the system. So, XML signing is way more complicated and error prone than JSON signing.

In most programming languages JSON parsers are usual since they plot straight to objects. Contrariwise, XML doesn't possess regular document-to-object mapping. For this reason, working with JWT becomes naturally comfortable than SAML. Considering us-

age, JWT has expanded at huge scale. As a result, for different platforms handling JWT on client side has already been smooth and fluent. Especially for mobile platforms where JWT is used more often.

Structure: JWT involves three different parts which are detached by dots.

- Header : Header is built on two part. Signing algorithm and type of the token.
- Payload: This part contains the claims which reflects an entity traditionally the user.
- Signature : Purpose of the signature is to verify that the message was not altered along the way. To proceed with a signature a header, a payload and the algorithm specified in the header is needed. Only then it can be signed.

The final output is three Base64 URL encoded strings which are detached by dot. These strings are compact and prepared for Hypertext Transfer Protocol (HTTP) and Hypertext Markup Language (HTML) settings.

Working procedure: During authentication, whenever a solo user effectively logs in the system via his credentials, a he will get a token in return. The returned token will be a JSON web token. As these tokens bears security essentials, they must be handled cautiously. For our application of JWT we have kept the lifespan of a token for five hours. During the designated five hours the token is active and working. After that the token is ineffective and worthless and new token needs to be issued. The JWT which has been generated contains the user id, the expiry time of the token and the generated time within it's hashed string.

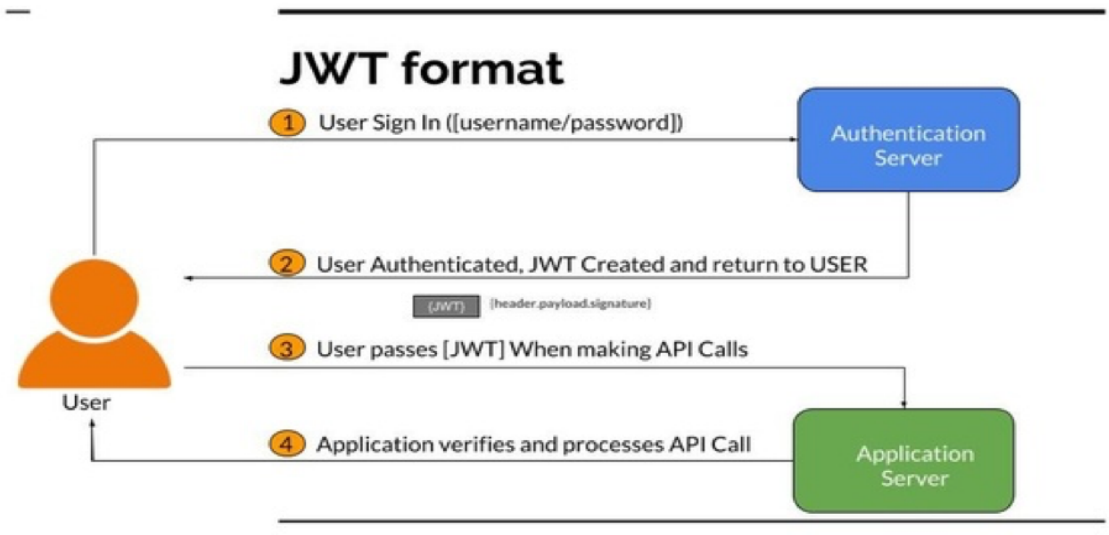

Figure 6: Data Flow for a JWT

When a user decides to access a specific route or definite resource which is protected, the user agent or the browser should send the JWT using the Bearer schema packing it inside the Authorization header. The header content should look something similar to this:
Authorization : Bearer <token>

After the server receives a request with this authorization header, the JWT is extracted from the header and then it gets cross checked for validation. If the incoming JWT is valid, the inbound request is good to go. User can then access the restricted resource or route. Due to the token being sent via authorization header mechanism not using cookies Cross Origin Resource Sharing (CORS) will not be a problem.

7.2.2 Authorization

While authentication refers to proving correct identity of the request, authorization refers to allow user or a user made request to perform certain action. More simply, authentication answers the question are you who you are and authorization answers the question are you allowed to do this. An API or an endpoint may authenticate a user or a request but may not authorize that request.

In our scope we have implemented custom role-based authorization. For the entirety of our application, we have three available roles for user.

- Administrator– Can manage contracts of the residents.
- Caretaker– Managing all the chores and appointment at the dormitories.
- Resident– Tenant living in dormitories.

Role based authorization means user with a role will not be permitted with actions which are out of bound for that role.

For our implementation, we have used javax security annotations pacify roles for our endpoints. The annotations take roles as parameter. We have used all these annotations at method level instead of class level. This way annotations are protecting all the methods from unauthorized access.

The following annotations are used to specify method permissions:

- **@DeclareRoles:** Specifies all the roles that the application will use, including roles not specifically named in a @RolesAllowed annotation. The set of security roles the application uses is the total of the security roles defined in the @DeclareRoles and @RolesAllowed annotations.
  The @DeclareRoles annotation is specified on a bean class, where it serves to declare roles that can be tested (for example, by calling isCallerInRole) from within the methods of the annotated class. When declaring the name of a role used as a

parameter to the isCallerInRole(String roleName) method, the declared name must be the same as the parameter value.

The syntax for declaring more than one role is

```
@DeclareRoles({"Administrator", "Caretaker", "Resident"})
```

- **@RolesAllowed("list-of-roles"):** Specifies the security roles permitted to access methods in an application. This annotation can be specified on a class or on one or more methods. When specified at the class level, the annotation applies to all methods in the class. When specified on a method, the annotation applies to that method only and overrides any values specified at the class level.
  To specify that no roles are authorized to access methods in an application, use the @DenyAll annotation. To specify that a user in any role is authorized to access the application, use the @PermitAll annotation.

  When used in conjunction with the @DeclareRoles annotation, the combined set of security roles is used by the application.

- **@PermitAll:** @PermitAll specifies that all security roles are permitted to execute the specified method or methods. The user is not checked against a database to ensure that he or she is authorized to access this application.

  This annotation can be specified on a class or on one or more methods. Specifying this annotation on the class means that it applies to all methods of the class. Specifying it at the method level means that it applies to only that method.

- **@DenyAll:** @DenyAll specifies that no security roles are permitted to execute the specified method or methods. This means that these methods are excluded from execution.

7.3 Database and Data Persistency

8 Deployment

8.1 Docker

8.2 Documentation

8.3 Manual Testing

Md Aminul Islam

Manual testing is testing process of applications to detect the bugs, to check the functionality of individual endpoints .It ensure the validity of software where testers manually execute test cases. For the development of new application must be manually tested before the testing process can be automated. Manual Testing requires more effort from the testers but is essential to check automation feasibility. Manual testing ensure the verification and validation of the endpoints for a application. In addition, manual test is also important to get an idea about the quality of the application[20].

8.3.1 Postman

Postman is common tool for checking RESTful API and its features also simplify each step of building an API and streamline collaboration.

Why postman for manual testing

- **Use of Collections:** In postman collections can be created for their API calls. Each collection can create sub folders and also possible to create for multiple requests. This helps in organizing the test for all endpoints[38].
- **Import/Export facility:** Collections and environments can be imported or exported making it easy to share files. Sharing collection is also possible[38].
- **Automation testing process:** With the implementation of Collection Runner or Newman tests can be run in multiple iterations saving time for repetitive tests[38].
- **Debugging the tests:** Postman console helps to check the requests and response in details which helps to debug tests[38].
- **Continuous Integration:** Development becomes more organized with the support of continuous integration[38].
- **Specify type of content:** For a request or response the data type can be select at postman like XML, JSON or TEXT.

Collections Users can group together requests into folders and collections and add a name and descriptions .It can also be used for the grouping created API definitions[38].

For the student residence application we created collection for each services. As there are 3 different services so we created

- Collection for Authentication service
- Collection for Contract service
- Collection for Appointments service

Postman also provide facility to Collection to be exported .The exported collections can be imported later and can be viewed in different postman remotely[38].Even for every request according to specific endpoints header and request body are visible to the user by postman.

Security validation For the testing each endpoints by postman at first login by any role need to be done in postman. Then for checking the endpoints postman need the access token which also ensure the testing endpoints with the validate user at postman also. As a result postman also ensures the security for testing every endpoints also.

References

[1] 8x8Cloud/swagger2raml. A utility to generate raml documentation from swagger json., 2020. https://github.com/8x8Cloud/swagger2raml, Last accessed on 2020-02-06.

[2] ajbrown/gradle-raml-plugin. A gradle plugin which validates your raml specification, and turns it into html documentation., 2020. https://github.com/PGSSoft/ramlo, Last accessed on 2020-02-06.

[3] Alistair Cockburn. Hexagonal architecture, 2005. [Online;Last accessed 2020-02-06].

[4] aml2html/raml2html. Raml to html documentation generator., 2020. https://github.com/raml2html/raml2html, Last accessed on 2020-02-06.

[5] API Workbench. Official documentation, 2020. http://apiworkbench.com/, Last accessed on 2020-02-06.

[6] beno/ramlrenderer. Htlm doc builder for raml 1.0., 2020. https://github.com/beno/ramlrenderer, Last accessed on 2020-02-06.

[7] Berners-Lee, Tim, Roy ,T. Fielding. Uniform resource identifier (uri): Generic syntax, rfc 3986, rfc editor. 1998. (http://www.rfc-editor.org/rfc/rfc3986.txt, Last accessed on 2020-02-06.

[8] brianmc/raml-store. This project provides an persistent store for raml files created using then embedded api designer, 2020. https://github.com/brianmc/raml-store, Last accessed on 2020-02-06.

[9] Cloud API Designer. Talend real-time open source data integration software, 2020. https://www.talend.com/products/application-integration/cloud-api-designer/, Last accessed on 2020-02-06.

[10] coub/raml_ruby. Raml ruby., 2020. https://github.com/coub/raml_ruby, Last accessed on 2020-02-06.

[11] cybertk/abao. Rest api automated testing tool based on raml, 2020. https://github.com/cybertk/abao, Last accessed on 2020-02-06.

[12] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56, 2015.

[13] Eclipse Jersey. Jersey, 2020. [Online;Last accessed 2020-02-06].

[14] esh-b/RAML-to-Swagger-Converter. A java project to convert raml 0.8 definition to swagger 2.0 definition., 2020. https://github.com/esh-b/RAML-to-Swagger-Convertero, Last accessed on 2020-02-06.

[15] farolfo/raml-server. run a mocked server just based on a raml api's definition .. zero coding, 2020. https://github.com/farolfo/raml-server, Last accessed on 2020-02-06.

[16] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD dissertation, University of California, Irvine, 2000.

[17] Roy Thomas Fielding. Rest apis must be hypertext-driven, 2008.

[18] Martin Fowler. Richardson maturity model. 2010. [Online;Last accessed 2020-02-06].

[19] David Garlan and Mary Shaw. An introduction to software architecture. *CMU Software Engineering Institute Technical Report*, 1994.

[20] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182. IEEE, 2016.

[21] Martin Fowler and James Lewis. Microservices, March 2014. [Online;Last accessed 2020-02-06].

[22] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[23] Microsoft. Microsoft - official home page, 2020. `http://www.mircosoft.com/`, Last accessed on 2020-02-06.

[24] mikestowe/php-ramlMerge. Raml merge lets you merge in any included raml files (!include) into a single raml file., 2020. `https://github.com/mikestowe/php-ramlMerge`, Last accessed on 2020-02-06.

[25] Mozilla. Resources for developers, by developers., 2020. `https://developer.mozilla.org/en-US/`, Last accessed on 2020-02-06.

[26] MuleSoft. Api notebook by mulesoft., 2020. `https://api-notebook.anypoint.mulesoft.com/`, Last accessed on 2020-02-06.

[27] mulesoft-labs/raml-dotnet-tools. Visual studio extension to work with raml and oas (openapi) specifications. you can consume rest apis, scaffold asp.net implementations and extract raml specifications from existing asp.net apps., 2020. `https://github.com/mulesoft-labs/raml-dotnet-tools`, Last accessed on 2020-02-06.

[28] mulesoft-labs/raml-for-jax-rs. This project is all about two way transformation of jax-rs-annotated java code to raml api description and back., 2020. `https://github.com/mulesoft-labs/raml-for-jax-rs`, Last accessed on 2020-02-06.

[29] mulesoft/api-console. An interactive rest console based on raml/oas files., 2020. `https://github.com/mulesoft/api-console`, Last accessed on 2020-02-06.

[30] mulesoft/api-designer. A web editor for creating and sharing raml api specifications, 2020. `https://github.com/mulesoft/api-designer`, Last accessed on 2020-02-06.

[31] nachoesmite/atom-raml. A simple package for atom (github's text editor) that helps the user to write raml specs., 2020. `https://github.com/nachoesmite/atom-raml`, Last accessed on 2020-02-06.

[32] nogates/vigia. Adaptable api integration test suite that supports api blueprint and raml files, 2020. `https://github.com/nogates/vigia`, Last accessed on 2020-02-06.

[33] OpenAPI. Openapi initiative, 2020. http://www.openapis.org/, Last accessed on 2020-02-06.

[34] ozwolf-software/raml-mock-server. Library for validating mockserver calls against a raml api specification, 2020. https://github.com/ozwolf-software/raml-mock-server, Last accessed on 2020-02-06.

[35] PGSSoft/ramlo. Documentation generator for raml 1.0., 2020. https://github.com/PGSSoft/ramlo, Last accessed on 2020-02-06.

[36] PGSSoft/ramlo. Documentation generator for raml 1.0., 2020. https://github.com/ajbrown/gradle-raml-plugin, Last accessed on 2020-02-06.

[37] phoenixnap/springmvc-raml-plugin. Spring mvc - raml spec synchroniser plugin. a maven plugin designed to generate server and client code in spring from a raml api descriptor and conversely, a raml api document from the springmvc server implementation., 2020. https://github.com/phoenixnap/springmvc-raml-plugin, Last accessed on 2020-02-06.

[38] Postman. The collaboration platform for api development, 2020. https://www.getpostman.com/, Last accessed on 2020-02-06.

[39] raml-org/raml-dotnet-parser. A raml parser implementation in .net for all clr languages., 2020. https://github.com/raml-org/raml-dotnet-parser, Last accessed on 2020-02-06.

[40] raml-org/raml-java-parser. A raml parser based on snakeyaml written in java., 2020. https://github.com/raml-org/raml-java-parser, Last accessed on 2020-02-06.

[41] raml-org/raml-spec. Raml specification http://raml.org, 2020. https://github.com/raml-org/raml-spec, Last accessed on 2020-02-06.

[42] raml-org/raml-tck. Test compatibility kit for raml 1.0., 2020. https://github.com/raml-org/raml-tck, Last accessed on 2020-02-06.

[43] REST. Rest - semantic web standards., 2020. https://www.w3.org/2001/sw/wiki/REST, Last accessed on 2020-02-06.

[44] Sandbox. Quickly create rest api and soap mock web services, 2020. https://getsandbox.com/, Last accessed on 2020-02-06.

[45] Vitaliy Schreibmann and Peter Braun. Model-driven development of restful apis. In *WEBIST*, pages 5–14, 2015.

[46] SoliDry/api-generator. Php-code generator for laravel framework, with complete support of json-api data format, 2020. https://github.com/SoliDry/api-generator, Last accessed on 2020-02-06.

[47] Vijay Surwase. Rest api modeling languages-a developer's perspective. *Int. J. Sci. Technol. Eng*, 2(10):634–637, 2016.

[48] Swagger. Api development for everyone, 2020. https://swagger.io/, Last accessed on 2020-02-06.

[49] therealgambo/ramlfications-php. Php 7 compliant implementation of the raml 1.0 specification., 2020. https://github.com/therealgambo/ramlfications-php, Last accessed on 2020-02-06.

[50] W3C. W3c, 2020. https://www.w3.org/, Last accessed on 2020-02-06.

[51] Chao Wang, Link Yu, James Pendergraft, and Wei Wang. Managing paginated data, April 3 2018. US Patent 9,934,202.

[52] Wikipedia. Project jersey, 2019. [Online;Last accessed on 2020-02-06].

[53] Wikipedia. Microservices, 2020. [Online;Last accessed on 2020-02-06].

[54] Wikipedia. Representational state transfer, 2020. [Online;Last accessed on 2020-02-06].

[55] Wikipedia. Software architecture, 2020. [Online;Last accessed on 2020-02-06].