# CM4108 Diary Application Report

Oliver Aarnikoivu

This report details the steps taken into building a functional personal diary application. I begin by describing the design of the Appointment class and my reasoning behind the different annotations that have been made use of. Next, the report outlines how the database and web services were implemented by placing on emphasis on the different URL paths, HTTP methods, status codes and parameters that were applied. All the requirements specified in the coursework documentation have been met.

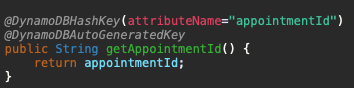
## Appointment class

The appointment class has been setup such that it consists of a *String* value for identifying an appointment, a long value containing the date and time of the appointment as the number of milliseconds since 01/01/1970, the duration of the appointment in minutes as a *Double*, the owner of the appointment as a *String*, and the description of the appointment as a *String*.

### Annotations

#### Appointment ID

The appointment ID has been provided with both the *DynamoDBHashKey* and *DynamoDBAutogeneratedKey* annotations. The hash key annotation ensures that the class property is mapped to the partition key of the table. The auto generated key annotation ensures that the system automatically generates an ID for an appointment. Specifically, the *DynamoDBMapper* generates a random UUID which represents a 128-bit value.



#### 

#### Date and time

A close up of a logo

Description automatically generatedThe *dateAndTime* field has been provided the *DynamoDBIndexRangeKey* annotation with a global secondary index (GSI) name of *“OwnerIndex”*. The index range key is used here as it allows querying a global secondary index and the refining of results using the index sort key discussed below. In particular, using a GSI here enables me to use a *DynamoDBQueryExpression* as opposed to a *DynamoDBScanExpression* for retrieving appointments that belong to a specific owner between two specified dates and times.

#### Duration

The duration field has been mapped with the simple *DynamoDBAttribute* annotation which simply maps a property to a table attribute. As shown by the image below, the *DynamoDBAttribute* maps the *duration* property to the duration attribute name in the table.

#### 

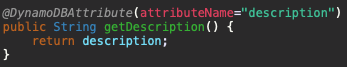
#### Owner

As was briefly mentioned when discussing the *dateAndTime* attribute, the GSI was made use of to refine results using the index sort key. Thus, using the index hash key for the owner maps…

#### A close up of a logo Description automatically generated

#### Description

Similar to the duration field, the description has also been mapped with a *DynamoDBAttribute* annotation simply mapping the property to a table attribute.



## Web Services

### AppointmentDatabase

The *AppointmentDatabase* is an interface which consists of all the required operations. In particular, I’ve provided methods for finding an appointment given by its ID, for finding appointments for a specific owner between two specified dates, for adding a new appointment, deleting an appointment given its ID and for updating an appointment given its ID and new attribute values.

A picture containing monitor, screenshot, television

Description automatically generated

### As can be seen, I’ve created both a *queryAppointmentBetweenDates* and *scanAppointmentBetweenDates* functions. Both these methods serve the same purpose, however, they are provided to compare the use of both a *DynamoDBQueryExpression* and a *DynamoDBScanExpression*.

### PersistentDB

The *PersistentDB* class implements the *AppointmentDatabase* interface in order to handle all the necessary logic for each database operation. In addition, the *PersistentDB* class consists of all the required configurations for generating a new instance of a DynamoDB table. I’ve configured the DynamoDB table name as *cm4108-coursework*, the region to local, and the local endpoint to port 8000.

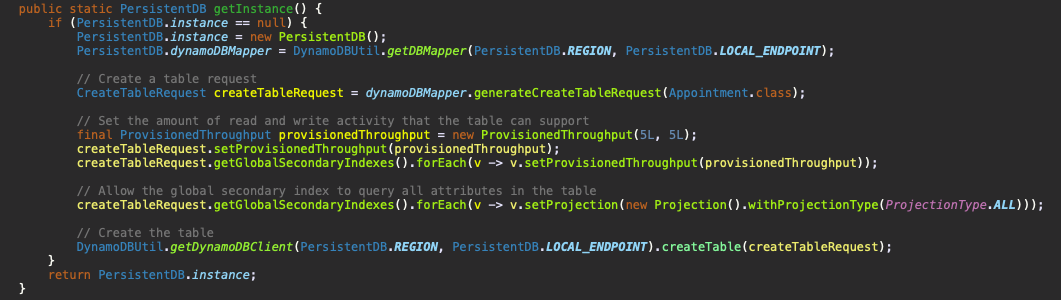
#### Creating a new instance

As shown by the image below, I’ve decided to create a new DynamoDB table programmatically. First, we determine if the current database instance is null. If this is the case, a new instance is created using the *generateCreateTableRequest* function provided by the *DynamoDBMapper* class. Next, we set the amount of read and write activity that the table can support. It’s important to note that a global secondary index has no size limitations and has its own provisioned throughput settings for read and write activity which are separate from the table. Therefore, the values for the provisioned throughput need to be set separately. This is being done using the snippet of code shown below.

*createTableRequest.getGlobalSecondaryIndexes.forEach(v -> v.setProvisionedThroughput(provisionedThroughput);*

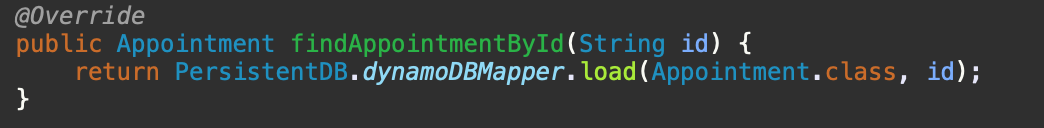
Furthermore, we need to allow the global secondary index to query all the attributes in the table. By default, the projection type is set to *KEYS\_ONLY* which ensures that only the index and primary keys are projected into the index. Whereas, when using the projection type *ALL*, we are ensuring that all of the table attributes are being projected. This is being done using the following snippet of code.

*createTableRequest.getGlobalSecondaryIndexes().forEach(v -> v.setProjection(new Projection().withProjectionType(ProjectionType.ALL)));*



#### Finding an appointment given its ID

In order to retrieve an appointment by its ID, I simply make use of the *load* function provided by the *DynamoDBMapper* where I pass in the appointment class as well as the partition/hash key of the appointment.



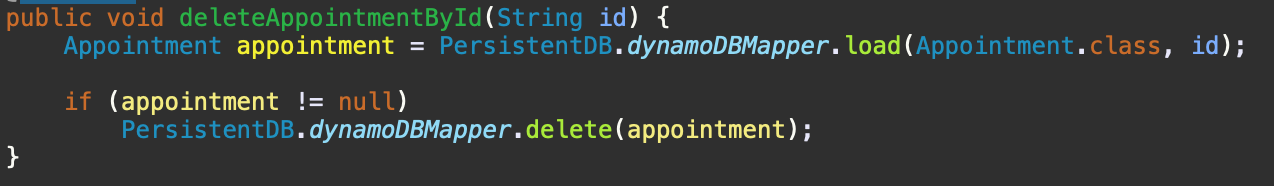
#### Adding an appointment

In order to add an appointment, I define a new appointment object and pass in the form-filled parameters. I then use the mapper to persist the appointment object. I round the value of the appointment duration to the nearest integer in order to simplify the formatting of the duration on the frontend.



#### Deleting an appointment

To delete an appointment, I first load the load the appointment object using the partition/hash key. I then check if the object exists, and if this is the case the appointment is simply deleted using the mapper *delete* function.



#### Updating an appointment

To update an appointment, I first load the appointment object using the partition/hash key. Next, I check if the appointment to update exists. If it does, then the loaded appointment is updated with the client-side form-filled parameters. Here I also round the duration to the nearest integer as a means to simplify the duration formatting on the frontend.

A screenshot of a cell phone

Description automatically generated

#### Retrieving appointments between two dates

In order to retrieve appointments between two dates for a specific user, one can make use of either DynamoDB’s query expression using a global secondary index or using a scan expression. If we take into account the table displayed below and consider that the table is identified by a partition key (AID), and we would like to retrieve the appointments for a specific owner between two different dates. Then, we would need to make use of a *Scan* operation. Evidently, as we add more appointments into the table, the scanning of all the data would become extremely inefficient. On the other hand, by providing a global secondary index named “OwnerIndex” with a partition key (Owner) and sort key (dateAndTime), DynamoDB is able to retrieve the projected attributes much more efficiently [1].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AID** | **dateAndTime** | **Duration** | **Owner** | **Description** |
| 123235 | 1574933188336 | 60 | Bob | Fishing |
| 135264 | 1570834800000 | 25 | Dale | Lecture |
| 136235 | 1576412100000 | 45 | Bob | Reading |
| 1522354 | 1575980100000 | 120 | Daren | Coursework |
| 1563245 | 1573388100000 | 120 | Bob | Walking |

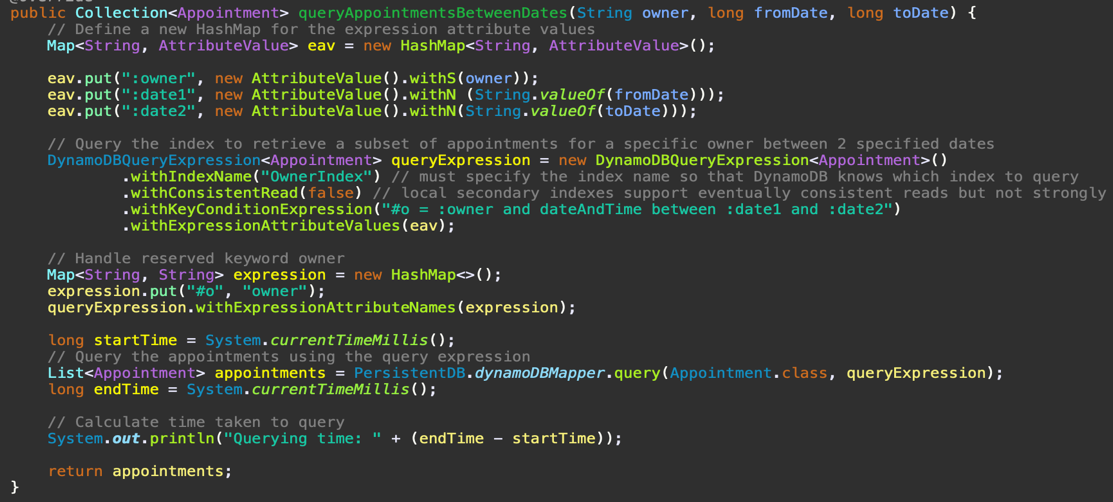
Table 1: Base appointment table without global secondary index

Without the projection type set to “ALL”, our query expression would return the table displayed below since the query is not able to fetch any non-key attributes from the source table. However, when the projection type is defined as “ALL” then the secondary index takes into account all of the attributes. Of course, when using “ALL”, we need to take into consideration additional throughput and storage costs [1].

|  |  |  |
| --- | --- | --- |
| **Owner** | **dateAndTime** | **AID** |
| Bob | 1574933188336 | 123235 |
| Dale | 1570834800000 | 135264 |
| Bob | 1576412100000 | 136235 |
| Daren | 1575980100000 | 1522354 |
| Bob | 1573388100000 | 1563245 |

Table 2: Appointment table using global secondary index

As shown by the images below, I’ve implemented both a *DynamoDBQueryExpression* and *DynamoDBScanExpression,* thus, either option can be made use of. For the query method, I first define a new HashMap for storing the expression attribute values. I then query the index to retrieve a subset of appointments for the specified owner between 2 long values for dates. It’s essential to specify the index name so that DynamoDB knows which index to query, therefore, I set the index name as *OwnerIndex* as this is the global secondary index I name I decided to make use of for both the index hash key and index range key as discussed above.



For the scan expression I simply scan the table with the relevant filter expression in order to retrieve the subset of appointments for a specific owner between the two specified dates.

### AppointmentResource

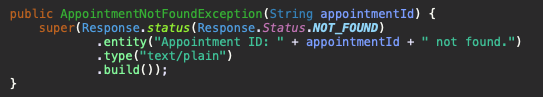
#### Adding a new appointment

In order to add a new appointment, I specify the method to handle post requests using the *@POST* annotation. Furthermore, I set the response content-type to plain text such that the server can respond with a status code and a plain-text response. If the adding of an appointment is successful, the server replies with a 201 status code implying that the appointment was successfully added to the database. If the adding of an appointment is unsuccessful, the server responds with a 400 status code to indicate that something went wrong.



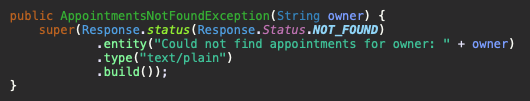
#### Retrieving an appointment by its ID

Here, I specify the method to handle get requests using the *@GET* annotation. Additionally, I specify the response content-type as JSON and append the appointment ID to the path. The appointment is then retrieved from the database using the *findAppointmentById(id)* function mentioned above. If the appointment exists, I simply return the appointment, whereas, if the appointment cannot be found, I return a new *AppointmentNotFoundException* which is a custom exception case that extends the *WebApplicationException* class. The exception returns a 404 status code to imply that the appointment was not found with the given ID.



#### Retrieving appointments between two specified dates

To retrieve appointments for a specific owner between 2 specified dates, I also make use of the *@GET* annotation in order to handle the request. I append the owner, the start date, and the date to the path in which I separate each path parameter with a slash. The appointments for the owner can then be retrieved by either using the *queryAppointmentsBetweenDates* or *scanAppointmentsBetweenDates* function. If the appointments between the two given dates for the specified owner are found, then the appointments are simply returned as JSON, whereas, if the appointments cannot be found, an *AppointmentsNotFoundException* is returned which simply indicates to the user that the appointments for the specific owner could not be found.

#### Deleting an appointment given by its ID

To delete an appointment, I use the *@DELETE* annotation and append the appointment ID to the URL path. The appointment is then removed by using the *deleteAppointmentById* function where I pass in the appointment ID. If the removal is successful, the server responds with a 200-status code to indicate that the appointment was successfully removed, whereas if unsuccessful, the server responds with a 500 status to imply that removing the appointment has failed.

#### Updating an appointment given by its ID

In order to update an appointment, I use the *@PUT* annotation and append the appointment ID to the URL path. The appointment is then updated using the *updateAppointmentById* function which takes in the appointment ID as well as the appointment object. If the updating of the appointment is successful, the server responds with a 200-status code to imply that the appointment was successfully updated. If unsuccessful, the server responds with a 400-status code indicating that something has gone wrong.



## Client side

#### Adding a new appointment

To add a new appointment into the database, I make use of the jQuery Ajax function where I specify the type as ‘PUT’ as well as other necessary parameters. Most importantly, I pass in the data using the *formToJSON()* function shown below. This function converts the form entries into a JSON object using the *JSON.stringify()* method. If the adding of a new appointment is successful, I alert the client with the success message received from the backend, whereas if the adding is unsuccessful, I alert the client with the server error message.



#### 

#### form2json

The *saveAppointment* function is called when a user clicks on the save button of the “Add Appointment” dialog box. As can be seen by the images below, if the appointment is successfully saved then the user is alerted with the appropriate success message.

A screenshot of a cell phone

Description automatically generated

#### Updating an appointment by its ID

Similar to adding a new appointment, for updating an appointment I make use of the jQuery Ajax function where I specify the type as ‘PUT’ as well as pass in the data using the *formToJSON()* method mentioned above to convert the form entries into a JSON object. If the updating on an appointment is successful, I alert the client with the successful response message retrieved from the backend, and with the error message if the updating failed.



#### 

An appointment is updated when the “update” button is clicked within the “Update Appointment” dialog box. As can be seen below, if the updating is successful then the user is alerted with the successful server response message.

A screenshot of a social media post

Description automatically generated

#### Deleting an appointment by its ID

Likewise to updating an adding an appointment, when deleting an appointment given by its ID, I use the Ajax function in which I specify the type as ‘DELETE’, pass in the url, and alert the client with either the success message, or error message received from the back-end.



#### 

The appointment can be deleted from within the “Update Appointment” dialog box. If successfully removed, the server responds with the success response message as shown by the image below.

A screenshot of a social media post

Description automatically generated

#### Retrieving appointments for a specific owner between two dates

In order to retrieve the appointments between two different dates, I make use of the jQuery shorthand Ajax function to retrieve the JSON data. Using a *forEach* loop, I retrieve all fields for each appointment. Since the date is being sent from the back-end to the front-end as a long value (timestamp), it’s essential to convert the value to a JavaScript date object.

*var date = new Date(dateTime);*

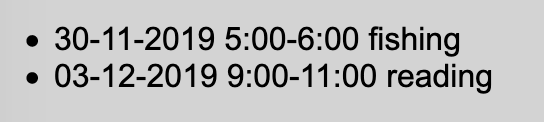
Once converted to a date object, I can simply receive the hours from the date object using:

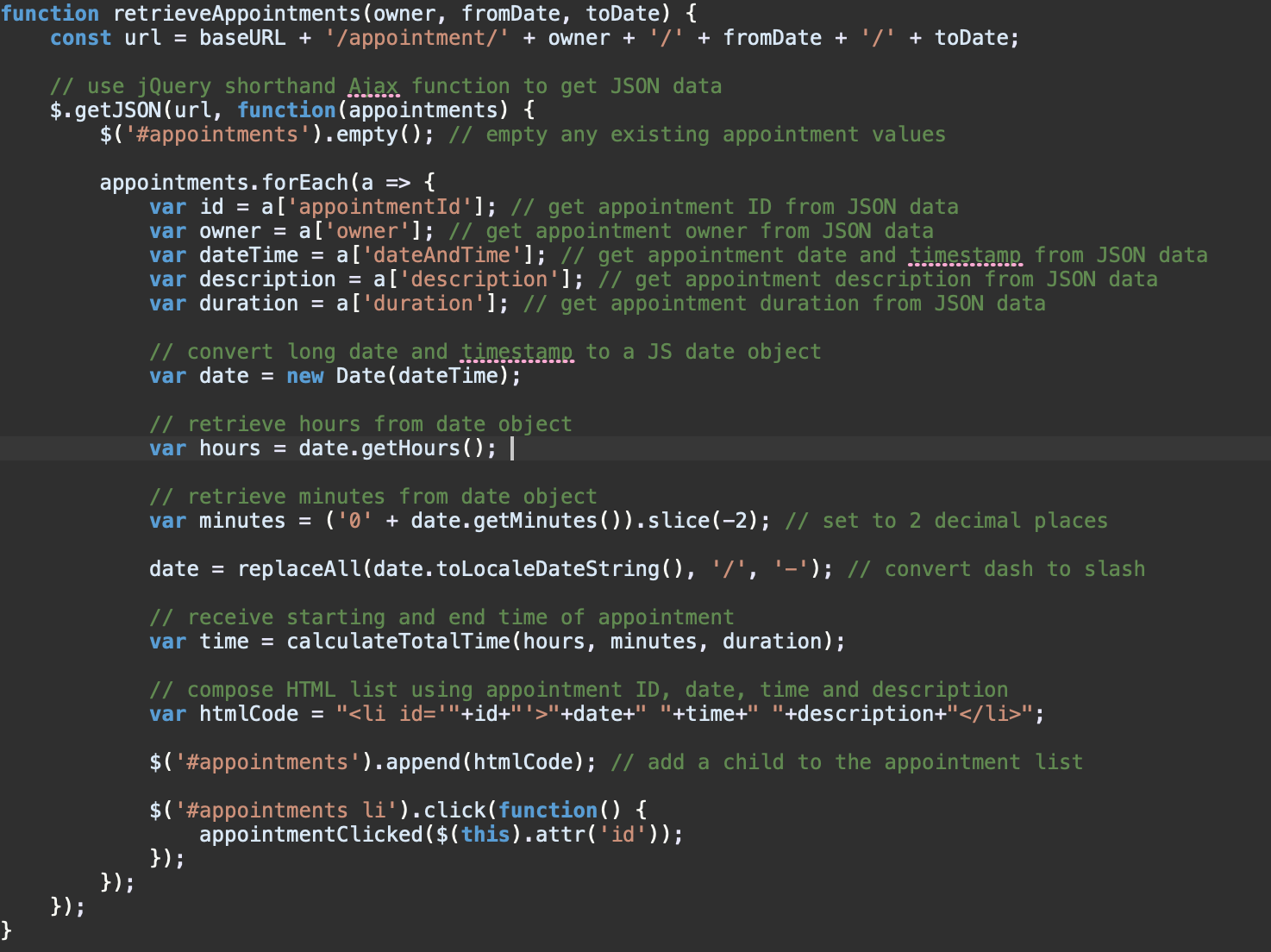
*var hours = date.getHours();*

as well as the minutes using:

*var minutes = date.getMinutes();*

Using the hours, minutes and duration I’m able to calculate the time when an appointment ends and display the appointment details in a clear and concise format as can be seen by the image below.





The *retrieveAppointments* function is triggered when the user fills in the input fields (User, From and To) and clicks on the “Show appointment(s)” button.

A screenshot of a cell phone

Description automatically generated

## Testing

The *AppointmentResourceTest* class consists of 6 different automatic tests using JUnit to ensure that the functions for adding, querying, scanning, deleting and updating appointments works as intended.

The *testAddAppointment* test makes sure that appointments can be added into the database.

A screenshot of a computer

Description automatically generated

The *testRetrieveAppointment* tests that an appointment can be retrieved from the database given by an ID.



The *testQueryAppointmentFromDates* test ensures that appointments for a specific owner can be retrieved between two different dates using a *DynamoDBQueryExpression*.

A close up of a logo

Description automatically generated

The *testScanAppointmentsFromDates* test ensures that appointments for a specific owner can be retrieved between two different dates using a *DynamoDBScanExpression*.

A close up of a logo

Description automatically generated

The *testUpdateAppointment* test retrieves an appointment from the database given by an ID. The retrieved appointments description is then changed and stored in a separate variable. Once the appointment is updated with the new description, I retrieve the same appointment from the database again and check that the retrieved description is not equal to the description stored in the separate variable.

A screenshot of a cell phone

Description automatically generated

The *testDeleteAppointment* test deletes an existing appointment, then attempts to retrieve the deleted appointment and checks that the fetched appointment is equal to null.

A screenshot of a cell phone

Description automatically generated

As seen below, all tests ran successfully suggesting that the methods have been correctly implemented.

A screenshot of a cell phone

Description automatically generated

In order to run the tests, first make sure to remove any existing instances of the PersistentDB. Then within eclipse, you can simply right click anywhere in the test class, click *Run As* and select *Junit Test*.

A screenshot of a video game

Description automatically generated