



UNSW

THE UNIVERSITY OF NEW SOUTH WALES

Group: 3

Student Name/Student ID/ E-mail:

[Fahd Alhamazani/ z5023495/f.alhamazani@unsw.edu.au]

[Pratulya Kasyap/z5020285/z5020285@studen.unsw.edu.au]

[Ali Haider Khan / z5107173/ z5107173@unsw.edu.au]

[Rohit Jose/z5131217/rohit.jose@unsw.edu.au]

Roles:

Project owner: Prof. Benatalla

Scrum Master: Mr. Shayan

Developer: Rohit, Ali, Fahd, PK

Submission date: 25/10/2017

Table of Contents

Introduction	3
Vision Statement.....	3
Problem Statement	3
Goals.....	4
Individual Team Member Goals	4
Overview (Architecture / design of the overall system & functionalities):	5
User documentation/manual:.....	7
Front-end:.....	7
Middleware:.....	7
API.AI:	7
Back-end:	10
Extraction:	10
Web-Service:	14
Deployment Instructions	19
MongoDB Datastore.....	19
REST Services Server	20
Resources:.....	21

Introduction

For a student commencing studies in UNSW, the primary form of reference for choosing a course is the UNSW handbook. The handbook offers the relevant content associated with a course such as the course prerequisites, outline and course page. However, most students find it difficult to track all this information during their enrollment. They might find it relatively overwhelming if they are new students and aren't used to the system. This application provides a simple solution that requires a simple query statement with as many details as required to search the UNSW courses database to find the course best suited to the needs of the user.

Vision Statement

Our Goal involves developing a software solution that allows an intuitive and enhanced interaction with the students for course enrollment related inquiries. We believe a simple solution can be found to relieve the stresses of enrollment and subject/course selection by building an application that can provide in helping new students acclimate to the UNSW course administration system.

Problem Statement

The goal of our project is to implement a software solution that assists students in relation to their enrollment related information. The current UNSW handbook pages contain information regarding the course requirements, course open/close/hold, day, time, lecturer name, enrollment capacity, semester, location, labs, tutorial, course description and course webpage. However, often students face the following issues with this interface:

- First-year students do not have the knowledge of what the university system is. They may still be used to their respective school system.
- UNSW timetable shows courses for undergraduate, postgraduate and research and some other courses which can be quite confusing for new students
- The UNSW Handbook pages are difficult to navigate and often outdated.
- Might contain incomplete/conflicting information with the faculty.
- Missing information in relation to a course difficulty.
- Missing the course outline information.
- The current UNSW Handbook platform is not interactive enough.
- Difficult to understand the details regarding specializations.
- The information provided is not visual enough and often quite confusing.

- The course prerequisites/eligibility information is difficult to interpret.
- The information presented is not personalized and often not completely relevant.
- The information in relation to a course spans across multiple pages.
- Number of courses available during the semester for enrollment is huge, difficult to know each course requirements and description.

Goals

1. Provide an interface to support student inquiries in relation to course enrollments.
2. Improve the enrollment experience of new students in UNSW.
3. Present the information in relation to the prerequisites of a course intuitively.
4. Build a simplistic and easy to navigate software solution; where students can resolve their inquiries without having to learn any tedious functionalities of the website.
5. Provide a centralized and personalized platform for students to get their course information.

Individual Team Member Goals

Our project consists of the following aspects of development:

- Identify and shortlist necessary data sources: PK
- Information extraction and pre-processing: Rohit and Fahad
- Data modelling and Loading: Fahad and Rohit
- Middleware and web services design and development: PK and Ali

Overview (Architecture / design of the overall system & functionalities):

Chappie is a chatbot that uses Facebook Messenger as its interface for communication. The reason for choosing Facebook Messenger is that it will be handy if clients can access the agent through mobile or any device. Chatbot consists of front-end user interface, middleware services layer and backend data store. For the front-end, we used a popular chat application interface as it will be easy and no need for manual installation.

We used API.AI as the chatbot framework that handles the Natural Language Processing section to identify user intents from messages relayed from the messaging interface. There are other options for NLP such as Amazon Lex (for more information about lex [link](#)), however, after comparing both products we found API.AI has better features and functionality. First API.AI is free for academic purposes, and second, API.AI provides a 'webhook' which simplifies communication between our user interface (messenger) and the agent, finally, API.AI also has a simplistic interface.

Back-end technology used is Node.js for handling JSON objects requests/responses (Fig. 1) Which is also used to extract information from the [UNSW timetable](#) & [UNSW HandBook](#). For extraction, there are two options, either extract data per queries or extract it once. Due to size of the data, (courses & timetables) we chose the latter option by comparing the process time between the two methods. We found one time extraction per semester is better approach, however, because the data's dynamic nature; changes during the first week of the semester and census date, we are forced to schedule a weekly update.

Some information requires a daily update which we can adjust to accordingly for the critical information such as course capacities. This information is stored in a MongoDB database for the client's queries. We chose MongoDB due to the disproportionate success of MongoDB is largely based on its innovation as a JSON document store that lets us more easily and

expressively model our data at the heart of our application. It dramatically simplifies the task of application development, and eliminates the layers of complex mapping code that are otherwise required. In summary, the motivation behind using MongoDB is due to its NOSQL nature and since our services, agent and interface all communicate with JSON, we believed modelling the data would be streamlined with MongoDB.

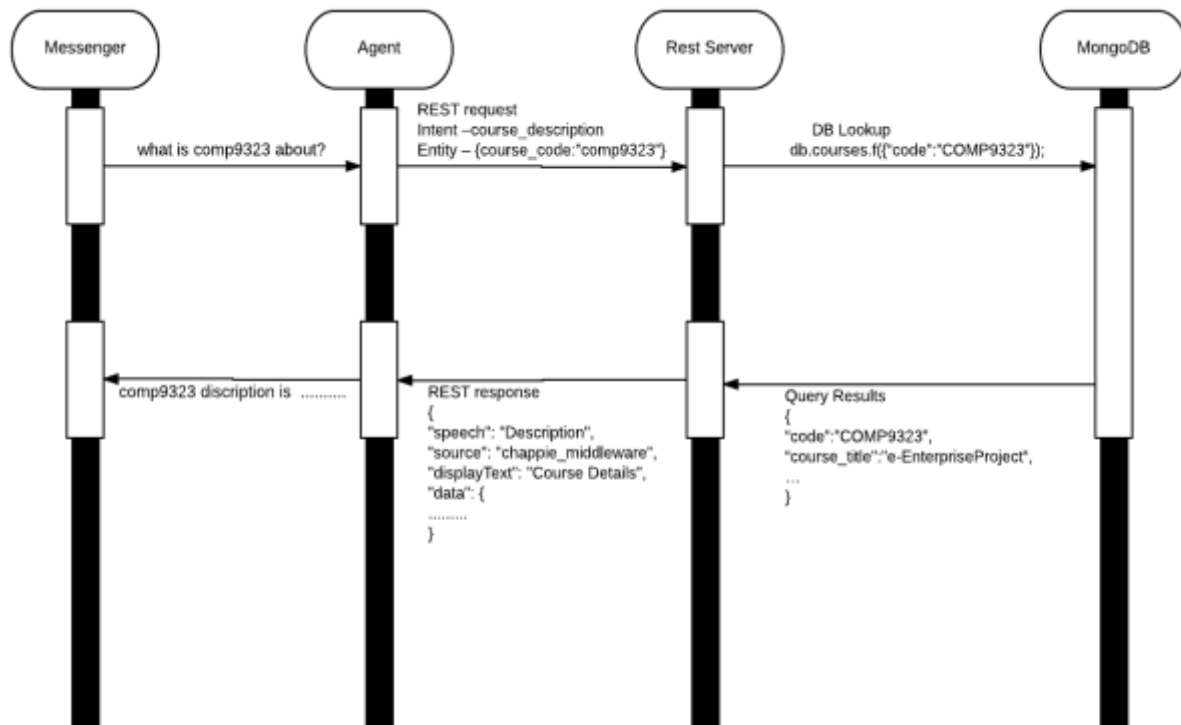


Figure 1: activity chart

User documentation/manual:

Front-end:

We used the Facebook Messenger as the front-end due to its known widespread popularity as the social media platform. The 2500 million active users make it handy and as is the expected case users' already have experience with the interface. API.AI provides an integration System "Webhook" which handles communication between the agents and messenger.

Middleware:

API.AI:

API.AI or as it's called now "Dialogflow" is a chatbot framework and a Natural Language Processing(NLP) service provided by Google. API.AI has the ability to process sequence data (text & audio). We use API.AI as the interpreter between the client and Rest service(Fig. 1).

API.AI's main purpose is to take the user's natural language question and find the intent or the "purpose of the question" and generate a JSON request object. This JSON request object is then passed to the REST service, which generates the response and API.AI then uses a suitable template to forward the required result to the user.

To start a project you need first to create an account ([Click here](#)) , then to create project follow this [link](#) . Following is the conceptual description of our AI.

Intents is the main objective you need to configure. Intents are used to catch the user's purpose of the question. The list of all the intents used in Chappie are shown(Table 1).In other words, intents are the domain of the program, each of the intents have expressions which are a set of questions or sentences that are linked to intent however this is not the final set. By using training methods, you have the option to deploy the agent (Chabot) and use every question

asked for training. Intents also need an attribute 'entity' which represent data type required eg. 'what is comp9323 about?' . Entity is the **course code** and intent is the **course description**) (see table 2) for a list of entities used.

The following is a description of the connection between API.AI and the Facebook messenger.

First, you need to create a [Facebook account](#). Second, create the Facebook developer app through the [Facebook Developer Console](#) which will generate a token that is used to interact with Facebook API. Finally, configure a 'Webhook' which uses the event handler, which then works as an interface between Facebook Messenger and the agent. For more details how to configure click on this [link](#).

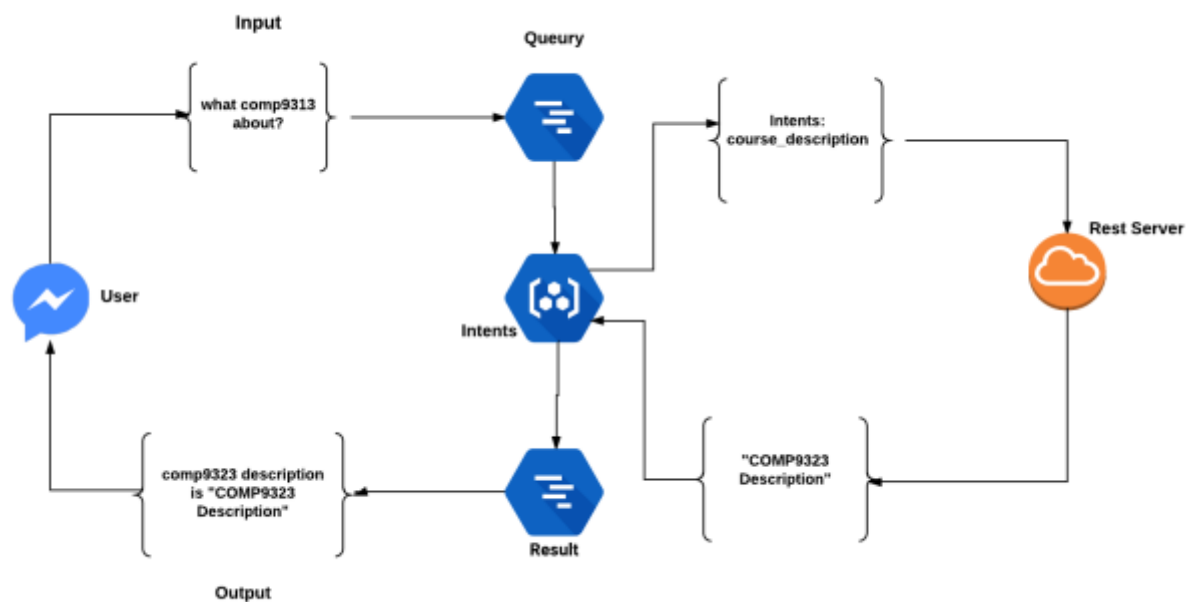


Figure 1: API.AI work as translator layer between User & Rest Server

Intents
<ul style="list-style-type: none"> • classdetail_clash • classdetail_day_info • classdetail_instructor • classdetail_lecture_duration • classdetail_lecture_location • classdetail_lecture_mode • classdetail_timetable • course_career • course_description • course_enrollment_capacity • course_enrollment_requirements • course_lookup • course_outline • course_page_link • course_units_of_credit • Default Fallback Intent • Default Welcome Intent

Table 1 : Intents list used in Chappie

Entity
<ul style="list-style-type: none"> • activity • course_code • day

Table 2: Entity list used in Chappie

Back-end:

Extraction:

As I explained early in this report we chose to extract it once. There are four JavaScript classes:

```
(class_timetable, course_description, course_details ,  
courses_overview)
```

To parse web page we used the [cheerio](#) library which provides advanced functions for retrieving information. For *class_timetable* we set a list of terms (tags) that related to the course timetable like *time*, *location* and *teaching period* (Fig. 3). We first parse the target page then for each line we check for “Class Nbr” tag as it appears to be first tag that appears in first row of course timetable (Fig. 4) The Source code shows the structure of the course information (Fig. 5). In the case of one of the tags matching *clean_attributes*, the function takes the line and cleans all html tags and unrelated information(Fig. 6), next an update function is called which records the course details and the link for the details(Fig. 7). Finally, a termination function is called when “Class Notes” tags are reached, this tag appears at the end of the course timetable record(Fig. 8).

```
9  
10 ▼ let label_tags = ['Class Nbr', 'Section', 'Teaching Period', 'Activity',  
11     'Lecture', 'Status', 'Enrols/Capacity', 'Offering Period', 'Meeting Dates',  
12     'Census Date', 'Instruction Mode', 'Consent', 'Day', 'Time', 'Location',  
13     'Weeks', 'Instructor'  
14 ];  
15
```

Figure 3 : full list of terms related to course time table

```

// Check if the value matches configured value for extraction
if (label && label === label_tags[0]) {
    queue_monitor = true;
    console.log("##### START #####");
}

```

Figure 4: trigger function

```

<tr>
  <td width="15%" class="label">Class Nbr</td>
  <td width="15%" class="data">5987</td>
  <td width="15%" class="label">Section</td>
  <td width="15%" class="data">H16A</td>
  <td width="15%" class="label"><a href="https://my.unsw.edu.au/s
  <td class="data">T1 - Teaching Period One</td>
</tr>
<tr>

```

Figure 5: each course timetable appear in source code with "Class Nbr" at first row

```

16 // Removed the ':' sign trims the values to
17 // form the attribute name value
18 ▼ var clean_attribute = function(attribute) {
19     attribute = attribute.toLowerCase();
20     attribute = attribute.replace(/:/g, "");
21     attribute = attribute.replace(/ /g, "_");
22     if (attribute === 'enrols/capacity')
23         attribute = "capacity";
24     return attribute;
25 };

```

Figure 6: clean_attributes function responsible for removing tags

```

27 ▼ var update_element = function(class_detail, object_id, details_page_link) {
28 ▼   try {
29 ▼     connection_handle.collection('courses').update({
30 ▼       "_id": object_id
31 ▼     }, {
32 ▼       "$push": {
33 ▼         "class_detail": class_detail
34 ▼       }
35 ▼     }, function(err, result) {
36 ▼       if (err) throw err;
37 ▼       else {
38 ▼         console.log("Update called for record: " + object_id);
39 ▼       }
40
41 ▼     });
42 ▼   } catch (err) {
43
44 ▼     console.log("Update error for: " + details_page_link);
45 ▼   }
46 ▼ };
47

```

Figure 7: update function which record course timetable to DataBase

```

if (label && label === 'Class Notes') {
  queue_monitor = false;
  // console.log(class_detail);
  update_element(class_detail, object_id, details_page_link);
  console.log("##### END #####");
  class_detail = {};
}

```

Figure 8 : terminate the extractor

For *course_description*, the information is located at the UNSW handbook website, and the course description always appears at the end of the page after “internalContentWrapper” tag (Fig. 9). Hence we use the same functions for extraction by just changing html tags that we use for triggering functions.

```

<div class="internalContentWrapper">
  <div></div><br />
  <h1> Securing Wireless Networks - COMP4337</h1>
  <div class="summary">
    <p><strong>Faculty:</strong>&nbsp;<a href="/faculties/2018/eng/eng.html">Faculty of Engineering</a></p>
:rong>School:</strong>&nbsp;<a href="http://www.cse.unsw.edu.au/">School of Computer Science and Engineering</a></p><p>
:rong>Career:</strong>&nbsp;<strong>Undergraduate</p>

```

Figure 9: tag used before description component at UNSW HandBook

The rest classes use same the methodology, however, with different html tags.

Web-Service:

The web service is divided into specific categories to collate and represent the stored information in the Database:

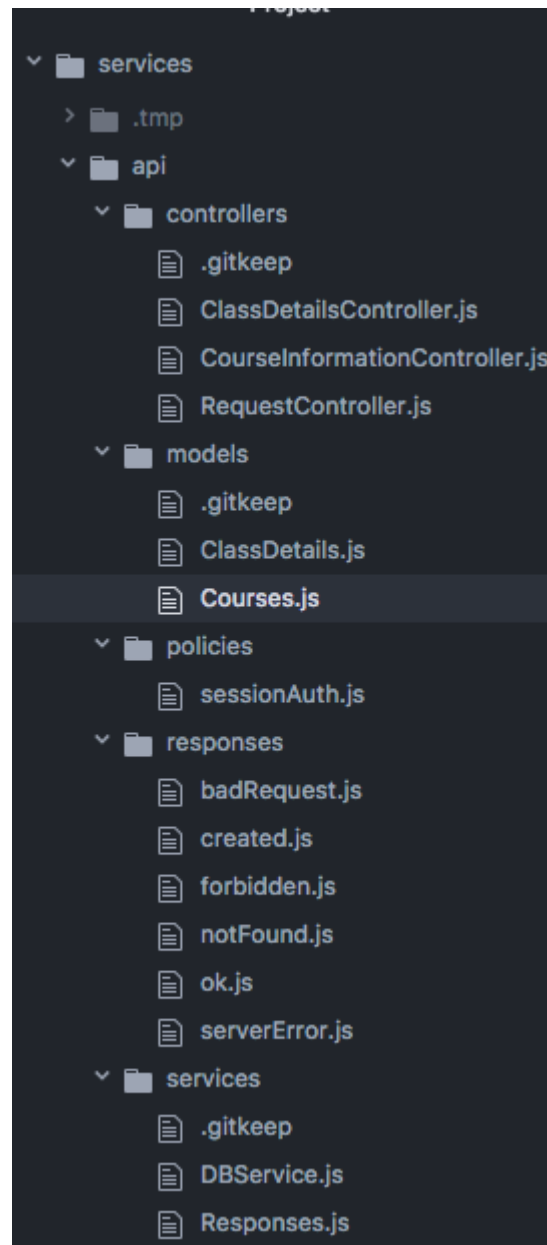


Figure 10: Service class design

The courses are represented as a “model” which allows us to represent the information in a JSON format that allows us to communicate between API.AI and the Messenger:

```
1  module.exports = {
2    attributes: {
3      _id: {
4        type: 'string'
5      },
6      code: {
7        type: 'string'
8      },
9      handbook_link: {
10       type: 'string'
11     },
12     course_title: {
13       type: 'string'
14     },
15     units_of_credit: {
16       type: 'string'
17     },
18     career: {
19       type: 'string'
20     },
21     faculty: {
22       type: 'string'
23     },
24     faculty_link: {
25       type: 'string'
26     },
```

Figure 11: Courses container for communication

When a request is received from API.AI, we need to process the parameters and convert them to a logical query to generate the results required. We process and parse them to a JSON file which queries the MONGO.DB instance and returns a result in JSON format. We make sure that multiple redundant queries are minimized so we develop and generate one JSON query depending on the parameters passed to the class.

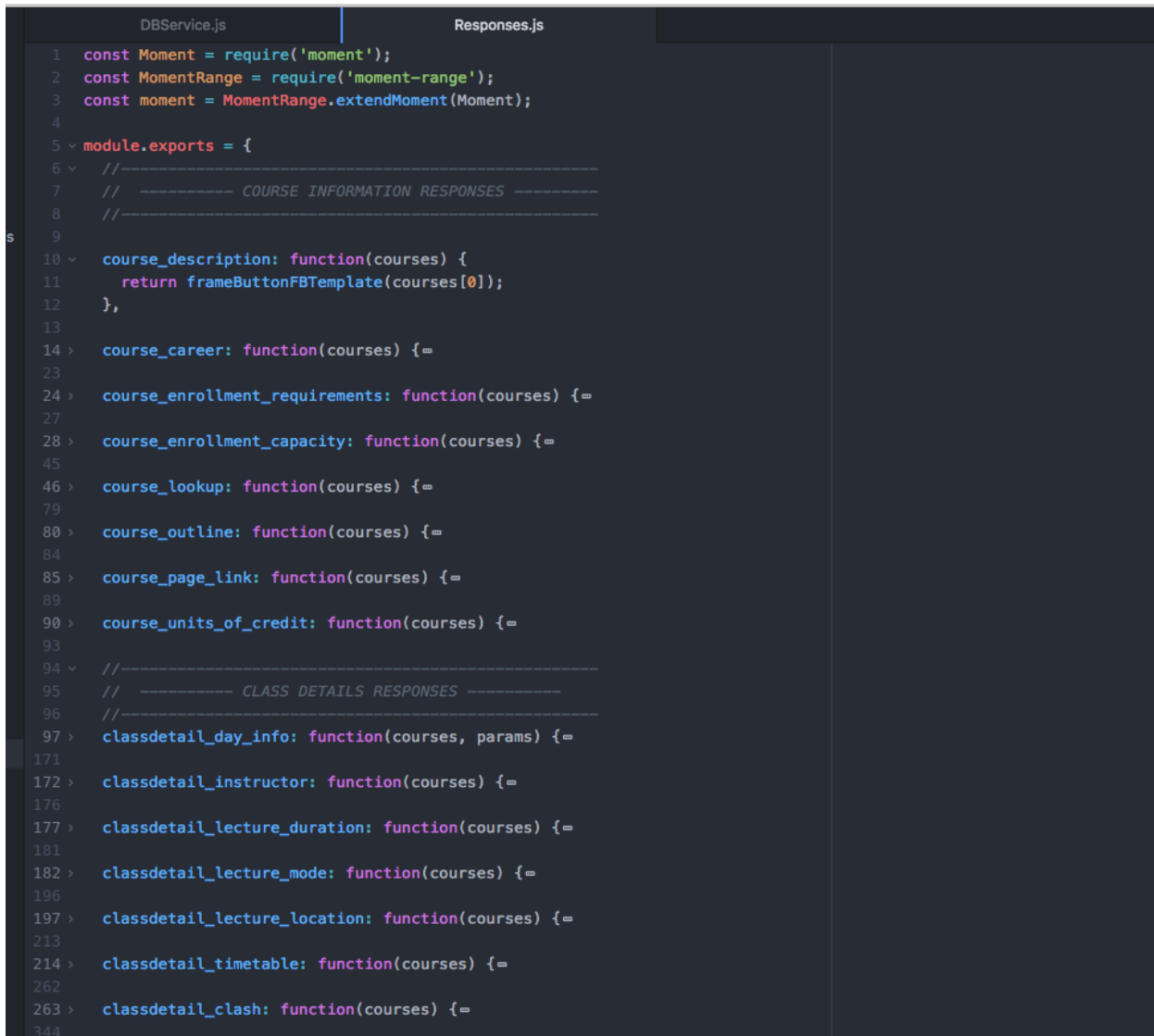
```

DBService.js  ClassDetailsController.js  CourseInformationController.js  Responses.js  Courses.js -- models  Courses.js --
116 function generateQuery(params, secondIter) {
117     let qry = '{}';
118     let key2;
119
120     Object.keys(params).forEach(function(key, index) {
121
122         if (key == 'course_code') {
123             if (secondIter) {
124                 key2 = 'course_title';
125             } else {
126                 key2 = 'code';
127             }
128         } else if (key == 'day' || key == 'time' || key == 'activity' || key == 'teaching_period') {
129             key2 = 'class_detail.' + key;
130         } else {
131             key2 = key;
132         }
133
134         if (params[key] instanceof Array) {
135             if (qry.length > 1) qry += ', ';
136
137             if (params[key].length > 1) {
138                 qry += '"or":["';
139                 params[key].forEach(function(param, idx) {
140                     qry += '{"' + key2 + "":{"contains":"' + param + '"}},';
141                 });
142                 qry = qry.substring(0, qry.length - 1);
143                 qry += '"]';
144             } else {
145                 params[key].forEach(function(param, idx) {
146                     qry += '"' + key2 + "":{"contains":"' + param + '"}';
147                 });
148             }
149         } else {
150             if (qry.length > 1) qry += ', '
151             qry += '"' + key2 + "":{"contains":"' + param + '"}';
152         }
153     });
154     qry += '}'
155     return JSON.parse(qry);
156 }

```

Figure 12: Generation of queries based on request parameters

This is then wrapped as a Node.js Rest response which is in responses.js. We split up the responses according to the request type. This means that we have responses for courses information, and class details. This allows us to differentiate and map the request to the intended purpose of that request.



```
1 const Moment = require('moment');
2 const MomentRange = require('moment-range');
3 const moment = MomentRange.extendMoment(Moment);
4
5 module.exports = {
6   //-----
7   // ----- COURSE INFORMATION RESPONSES -----
8   //-----
9
10  course_description: function(courses) {
11    return frameButtonFBTemplate(courses[0]);
12  },
13
14  course_career: function(courses) {=}
23
24  course_enrollment_requirements: function(courses) {=}
27
28  course_enrollment_capacity: function(courses) {=}
45
46  course_lookup: function(courses) {=}
79
80  course_outline: function(courses) {=}
84
85  course_page_link: function(courses) {=}
89
90  course_units_of_credit: function(courses) {=}
93
94  //-----
95  // ----- CLASS DETAILS RESPONSES -----
96  //-----
97  classdetail_day_info: function(courses, params) {=}
171
172  classdetail_instructor: function(courses) {=}
176
177  classdetail_lecture_duration: function(courses) {=}
181
182  classdetail_lecture_mode: function(courses) {=}
196
197  classdetail_lecture_location: function(courses) {=}
213
214  classdetail_timetable: function(courses) {=}
262
263  classdetail_clash: function(courses) {=}
344
```

Figure 13: Course Information and Class detail Responses

Finally, we generate a REST response with the code and the results through the response schemas defined according to each type, ie. OK, notFound, Created, forbidden and server error.

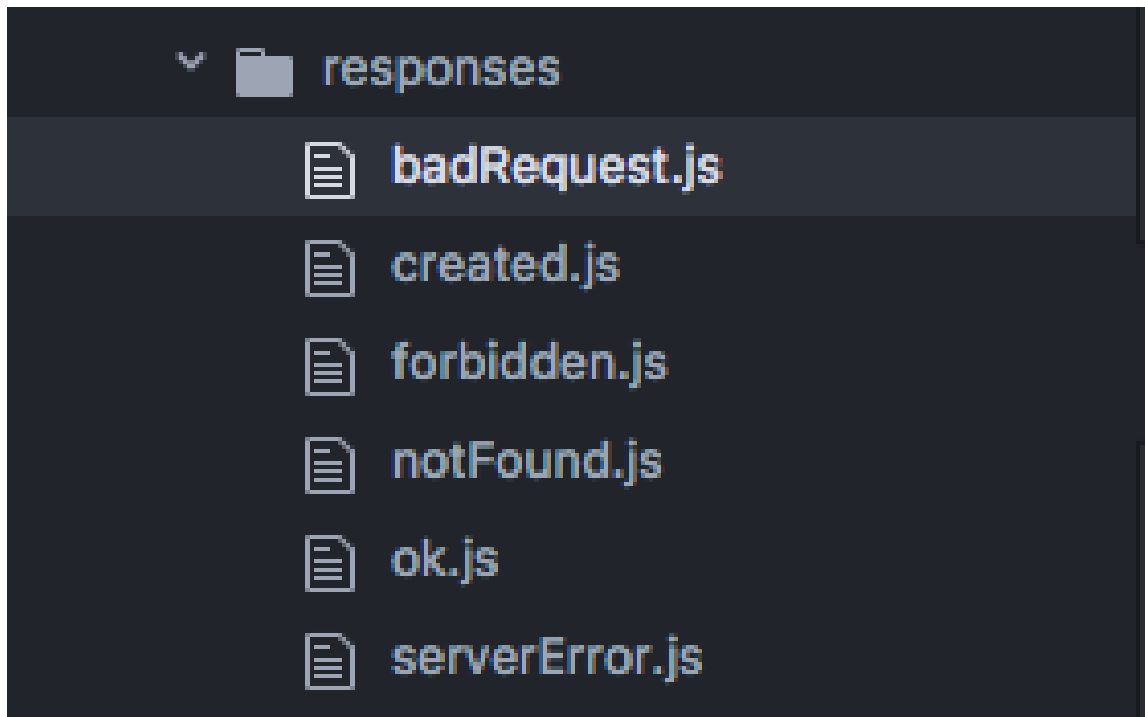
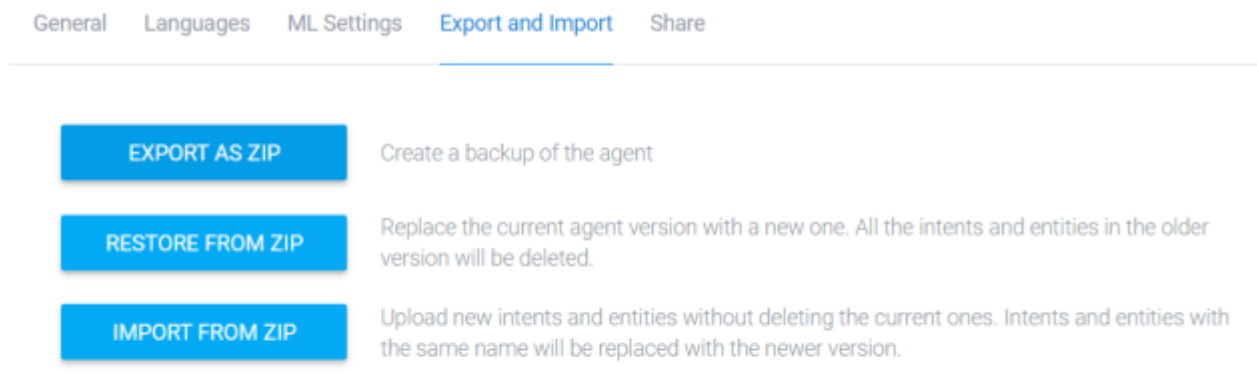


Figure 14: Response objects for REST service

Deployment Instructions:

The application source code is hosted in a remote git repository. The chatbot agent has been exported into a zip file under the '*ChatBot / chappie_export /*' folder. The agent can be restored from the settings menu under the 'Export and Import' tab using the 'Restore from ZIP' option.



The following steps describe how to deploy the application middleware and MongoDB data store on a linux server:

MongoDB Datastore

1. Install a local [MongoDB](#) server instance.
2. Start the local MongoDB instance
`sudo service mongod start`
3. Clone the git repository code to your source folder
4. Install [node.js](#).
5. Install dependencies: Navigate to the folder '*ChatBot / dataLoad_scripts /*' and run the command
`npm install`
6. Data Load: Run the following scripts in a sequence to load the data
`node course_details.js`
`node course_description.js`
`node class_timetable.js`

REST Services Server

1. Clone the git repository code to your source folder
2. Install [node.js](#).
3. Install the package [pm2](#) which is a process manager for node.js.
4. Install dependencies: Navigate to the folder '*ChatBot / services /*' and run the command [npm install](#)

The command installs the dependencies for the node.js services.

5. Run the server: Run the following command to start the node.js server in the services folder.
[pm2 start app.js](#)

Resources:

- [MongoDB](#)
- [API.AI](#)
- [Node.js](#)
- [FaceBook Messenger](#)
- [Cheerio](#)
- [Facebook Messenger templates](#)