Service Oriented Architecture, SOA
Assignment

Joshua Harrison
Student number: 838252
27th February 2018

Part A. External Architecture

Question 1:

The main roles in the overall architecture as listed below:

1. *Content Provider*: The content provider is responsible for uploading and maintaining the content in the MightyMooC system. After the registration process, they can make POST requests to submit module content. A content provider could form many guises, and may not be strictly limited to an traditional university. For example, established business leaders might sign up to the platform to offer management and leadership courses.
2. *Student*: Students would be the primary consumers in the system. With the ability to enrol on modules or whole courses to gain qualifications, but with no ability to edit any of the content to which they have enrolled. They would be charged on a pay per module basis.
3. *Admin*: Internal MightyMooC staff users. They would use the system to handle vetting content providers and submitted content to ensure that it was up to standard also dealing with payment issues such as requests for refunds.
4. *Corporates*: Business could sign up to the platform as a means to facilitate training and e-learning.
5. *External examiners*: Examiners from universities with the responsibility of marking student assessments and providing quality control on subject matter.

Question 2:

The below table is provided as a reference for the core service that would be required to implement a minimum viable product for a fully operational system.

The following business process definitions were asserted as the core goals of *this* business process:

1) Many course providers can offer many courses to a collaborative collection of material.
2) Modules from disparate course providers can be 'bundled' to form a single qualification.
3) Students can sign up for a single standalone 'module' from a course provider or to a larger qualification offered from one or multiple institutions.
4) Course providers need to be able to provide content for modules, this may be in various media types (PDFs, Video Media Files etc.)
5) Students need to be able to access content at low latency at any time.
6) Students need to able to submit module assessments.
7) Course providers need to be able to access student submissions; providing grading and feedback to them.
8) Students need to be able to download any certificates awarded as a result of completing a given module/course.

Role	Service	Function	Description	HTTP verb	Media types
Corporate	Account_manage	Block book module credits	Block book a number of modules in approved subject areas.	POST	JSON
Corporate / Course	Account_manage	Add users	Add users affiliated with the institution	POST / DELETE	JSON
Corporate / Course	Account_manage	Add users	Add users affiliated with the institution on	POST / DELETE	JSON
Corporate / Course	Account_manage	Remove Users	Remove users affiliated to the institution	POST / DELETE	JSON
Corporate / Course	Account_manage	Remove subscription	Remove the institution for the platform	POST / DELETE	JSON
Course Provider	Account_manage	Add payment details	Add the account details of the institution to receive payments for content	POST	JSON
Student / Corporate	Account_manage	Download certificate	On completion of a module/course download a digital certificate	GET	JSON
Student / Corporate	Account_manage	Delete Account	Delete account from platform	POST/DELETE	JSON
Student / Corporate	Account_manage	Cancel subscription	Cancel a subscribed module / Course	POST / DELETE	JSON
Admin	Assignment	Mark assignments	Calculate marks to multiple choice student assignments	POST	JSON
Student	Assignment	Submit assignment	Upload a completed assignment activity	POST	PDF
Student	Assignment	Take online test	View online test and submit answers	POST/GET	JSON
Student	Assignment	Retake test	Ability to retake a test if unsatisfied with result	POST	JSON
Course Provider	Catalogue	Upload Module material	Upload material to form part of an online course	POST	Text, PDF, Media files
Course Provider	Catalogue	Preview Module material	View a preview of the uploaded material before it becomes 'finalized' and available to students	GET	Text, PDF, Media files
Course Provider	Catalogue	Submit module material	Submitting learning materials material	POST / DELETE	Text, PDF, Media files
Course Provider	Catalogue	Update material	Request to update previously registered material	POST	JSON
Course Provider	Catalogue	View rejected modules	Review modules that have been from the platform(for example if they didn't meet the assessment criteria).	GET	Text, PDF, Media files
Global	Catalogue	Access subscribed material	Access material that is offered as part of a subscribed/provided module.	GET	JSON
Admin	Content	Review content	Review content submitted by institutions	GET	Text, PDF, Media files
Admin	Content	Approve content	Approve review content, meaning no amendments required	POST	JSON
Admin	Content	Reject content	Reject reviewed content - providers need to make amendments to the content and resubmit .	POST	JSON
Student	Education	Enroll module	Subscribe to a module	POST	JSON
Student	Education	Enroll Course	Subscribe to a Course	POST	JSON
Student	Education	See progress	View progress on a module so far	GET	JSON
Student	Education	Update Progress	After a unit has been viewed, the client should send a POST to the user has view the material	POST	JSON
Student	Feedback	Rate Course	Give a rating out of 5 for the course	POST	JSON
Student	Feedback	Rate module	Give a rating out of 5 for the module	POST	JSON
Global	Login	Login (direct)	Login directly through the application	POST	JSON
Global	Login	Login (indirect)	Login through third party applications	POST	JSON
Global	Media	Stream media content	As above, but specifically for viewing video media content	GET	Media Files
Global	Media	Download service	Download digital certificates awarded to students	POST	PDF
All	Payment	Take payments	Take payments for modules / courses / credits	POST (transactic	JSON
All	Payment	Make payments	Make payments to course providers	POST (transactic	JSON
All	Payment	Make Refund	Make refunds to students/institutions providers	POST (transactic	JSON
Student / Corporate	Payment	Make Payment	Make a payment through a connected third party provider	POST	JSON
Admin	Refund	Review refund request	Review student refund request	GET	JSON
Admin	Refund	Approve refund	Approve refund to be sent. Upon approval the refund payment service be called to issue the refund	POST	JSON
Admin	Refund	Reject refund	Reject the refund request with a customer free text message to the student	POST	JSON
Student	Refund	Request refund	Request a refund for a cancelled module which was unattended	POST	JSON
Corporate	Signup	Register institution	Sign company for a corporate account	POST	JSON
Course Provider	Signup	Register institution	Sign up as an institution to provide learning materials	POST	JSON
Student	Signup	Register	Register to the platform	POST	JSON

Figure 1: Core service requirements

Question 3:

A skeleton schema has been created using the Python Library SQLalchemy this is listed in appendix J. Omitted from this schema is the structure for the content that course providers would upload to the system. This could take a number of forms, but the most suitable would be a JSON object. The structured nature of this data would enable dynamic delivery of content to the frontend. Further it would enable better interoperability between external service consumers through well-defined and consistently formatted output.

The core units of the schema are as follows:

Institution: A content provider capable of adding new learning materials.
User: A student user capable of enrolling on courses and modules.
Course: A course is a logical grouping of modules with a common theme.
Module: A single unit of educational material covering one specific area of study.
Tag: A descriptive tag for module content.

Both the modules and courses models have been implemented having many to many relationships with institutions, meaning that multiple course providers can contribute content towards a module or course (see Appendix J).

Question 4:

REST has been chosen as the technology interface for the SOA. This is due to its wide spread adoption as a framework, and the growing move away from SOAP architectures in general. Whilst the benefits of language definitions through WSDLs are an advantage of SOAP, and in many ways the content upload schema defined previously would be well suited to XML, such schemas can also be implemented and maintained through JSON through the use of the OpenAPI standard. The widespread adoption of the REST framework is advantageous programmatically as there are a plethora of external tools and libraries dedicated to implementing elements of the REST framework (Bloomberg, 2013). This project (which has been built using Python) uses Flask, Flask-SQLalchemy, Werkzeug, flask-RBAC, and Postman (for testing) all of which are examples of tools either specially designed for REST application development or which assume REST at their core.

Question 5:

The Catalogue service was created as a multi-purpose service to access and edit data in the MightyMooC catalogue for both students and content providers. Students accessing the catalogue service would be able to make GET requests to list all content provided by a given institution, POST requests to enrol on any courses or modules, and DELETE requests to cancel their enrolments. Content providers can make the same GET requests as students and additional POST requests to add content to the catalogue. DELETE requests are made sending 'soft_delete' signals to the MightyMooC backend as no module should be completely purged at will.

Postman was used as a testing suit for the API endpoints later in the process (see part B). Postman is a powerful tool as it can generate API documentation in JSON format concurrent to testing the endpoints.

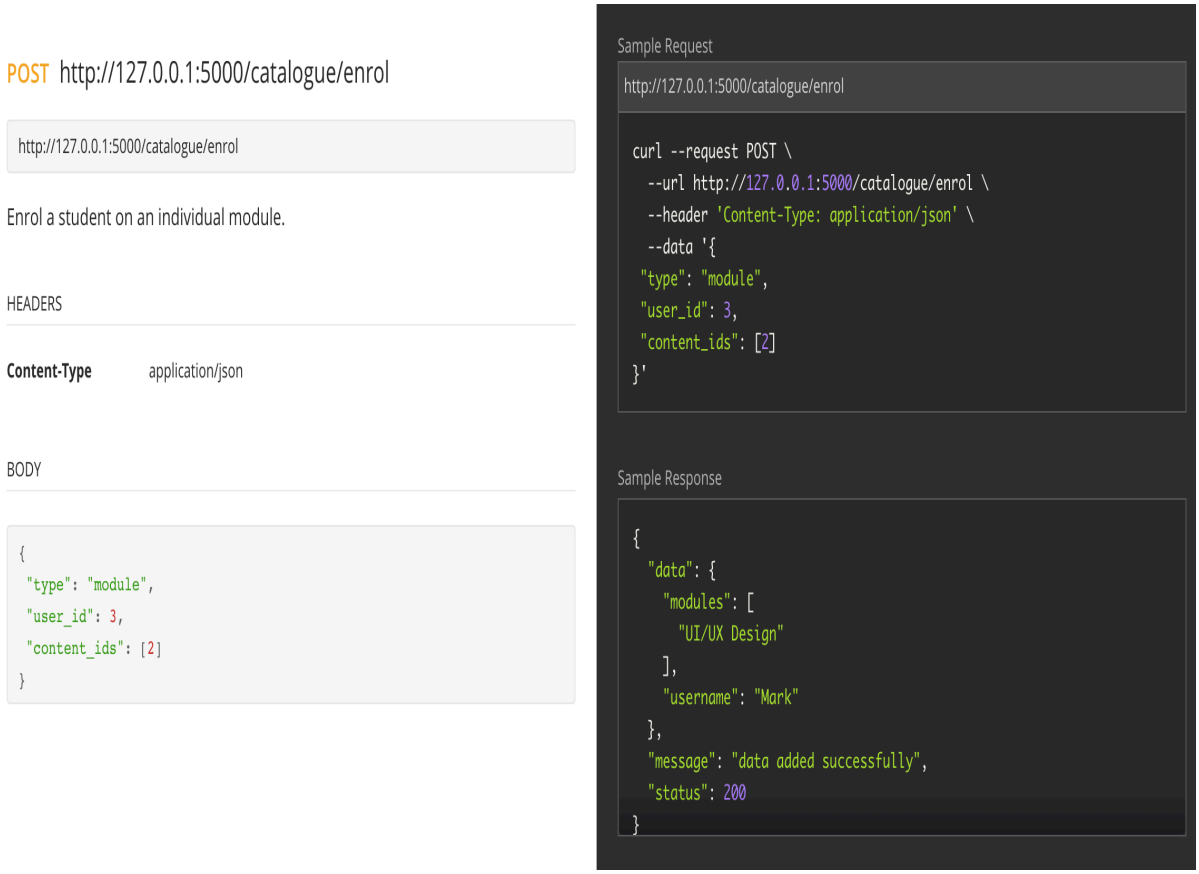

Figure 2: Postman documentation for the catalogue/enrol endpoint. The full API documentation is available in appendix A.

In Figure 2 we see the documentation of a POST request to the *catalogue/enrol* service endpoint. This request would be made by a client to enrol a student to a given module or course. Here we can see that the endpoint headers are application/JSON data and the JSON in the body includes the data detailing the enrolment request.

The right-hand output shows a sample cURL request. Also shown in the right-hand display is the sample response which is generated from the sample output. Here we can see a success message and echoed back are module and user details which have now been enrolled.

Question 6:

As well as the core services described previously, the interoperability of the model lends itself to the creation of several ancillary services.

Service	Duty
Analytics service	Connect to DB and perform custom reporting on user data
Content review service	Flag to admins if content is receiving low average ratings to review
Invoicing service	Automatically generate invoices payable to course providers based on the content submitted
Payment Processing	Handle inbound transactions from individual users and corporations
Central Registry	Relational database for all user and content data
Content Store	Non relational datastore for all module content and media files
Cache service	Connect to the content store and serve data to the cache server upon request.
Job queue service	Maintain a job que and manage on going jobs such as payment services
Notification service	Email/Text/push notifications to relevant parties

Figure 3: Ancillary services required of a minimal viable product of MightyMooc.

Question 7:

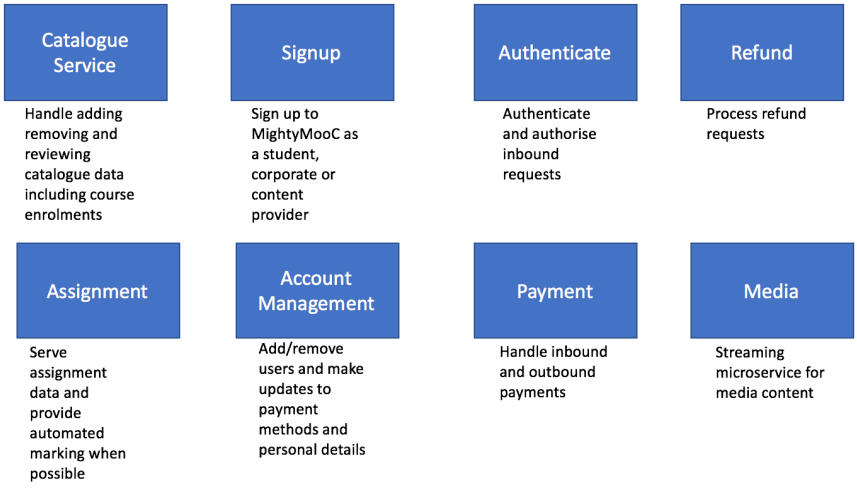

Figure 4 provides a brief description of all external facing MightyMooC services. Note: omitted from this diagram is the 'content' service as it is only available to MightyMooC admins and other internal services.

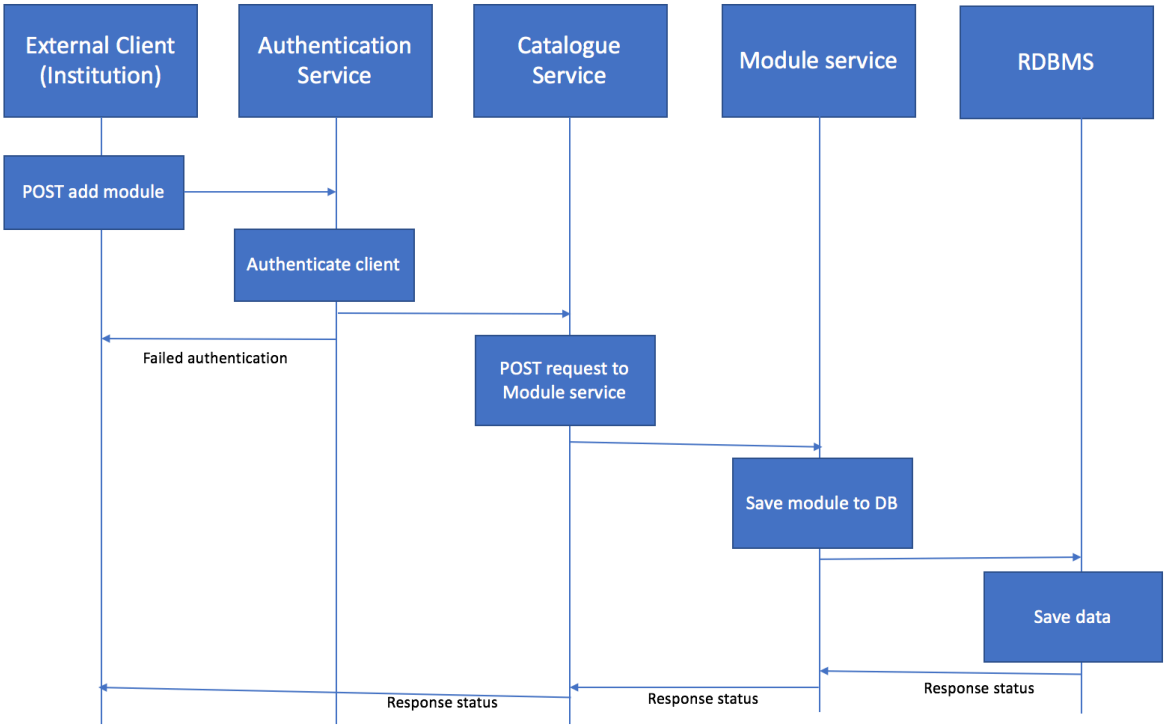

Figure 5: Sequence diagram of an external client making a post request to add a module to the MightyMooC catalogue.

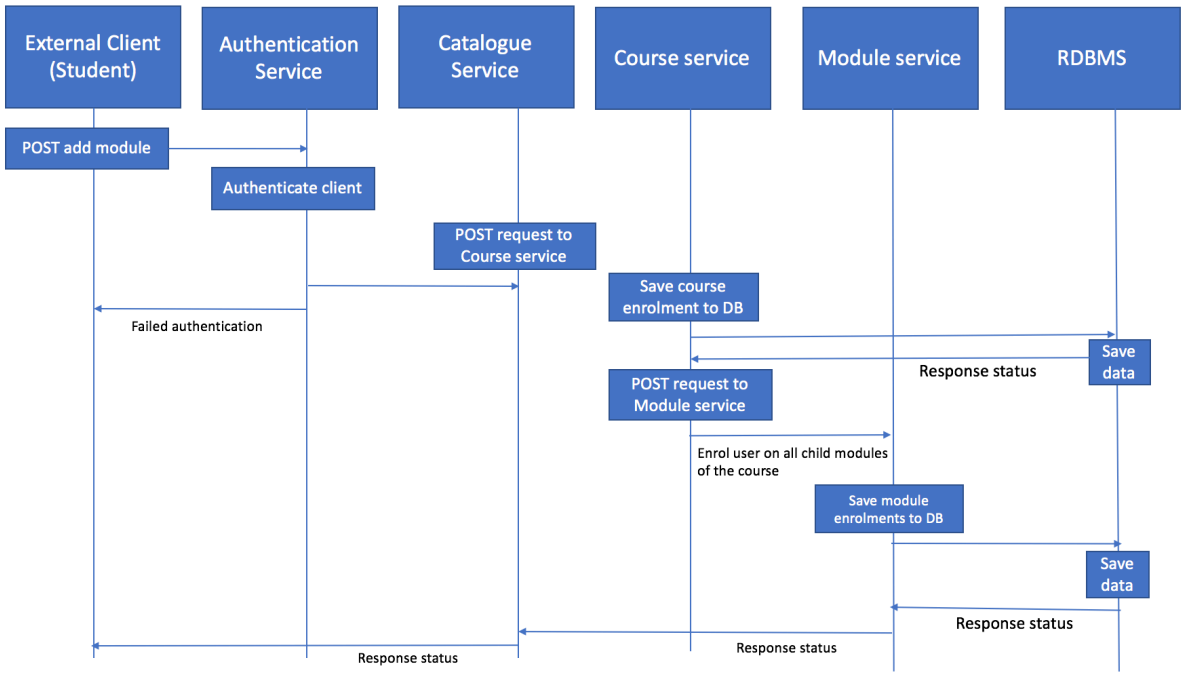

Figure 6: Sequence diagram depicting a user enrolling on a course

Part B: Internal Architecture

Question 1:

The catalogue service was designed and built using a top down design methodology. Erl (2016, pg. 96-97) conceptualises a cycle which aims to produce a *'service inventory blueprint'* from which services will be built. The application of this cycle in this project is detailed below.

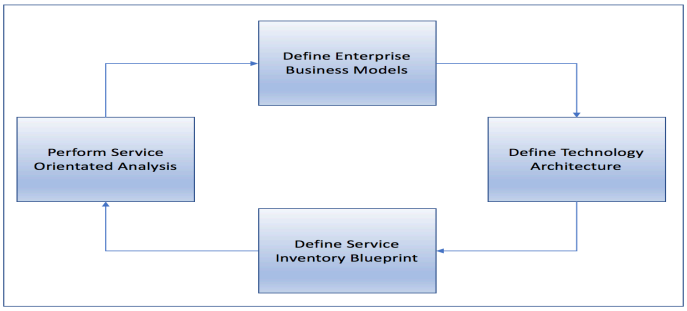

Figure 7: Service Inventory Analysis Cycle, Source: Erl. 2016, Pg. 96.

Define Enterprise Business Models

The catalogue service is an outward facing service which is a logical layer to many connected internal facing services.

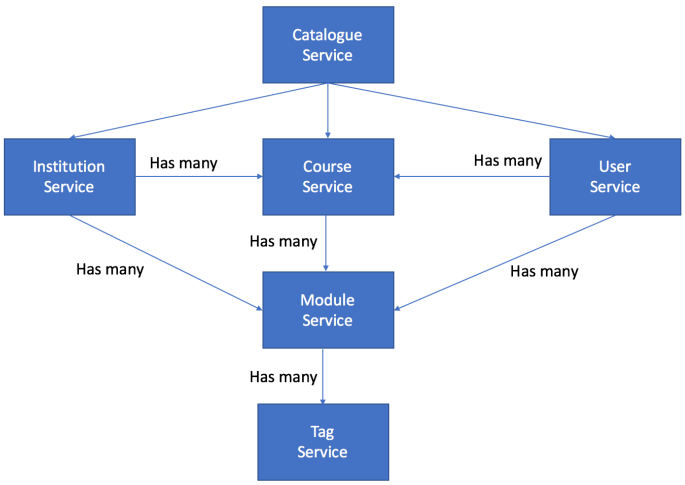

Figure 8: Hierarchy of connected services from the catalogue service

This interconnectivity allows the catalogue service to be reusable for both content provider and student clients. The services shown in *Figure 8* are listed in Appendices C-H.

Define Technology Architecture

It was decided that a centralised content store to which content providers would upload material was the most stable architecture (see *Part B, Question 2*).

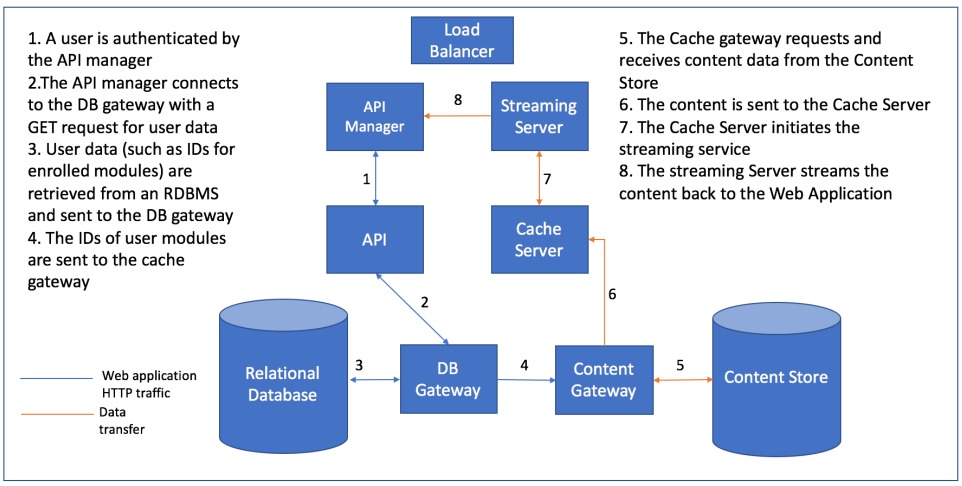

Figure 4: MightyMooC technology architecture and example data flow for a video stream.

Define Service Inventory Blueprint

A BaseService class was defined in python to act as a technological blueprint. The BaseService class should contain all logic that is universal in the project. This enables future services to use inheritance and polymorphism to extend and override methods in the base service.

```
def dyanmic_module(self):
    ''' Dynmically load the relevant class from mightMooc.models
    '''
    mod = __import__('mightyMooc.models', fromlist=[str(self.model)])
    db_module = getattr(mod, str(self.model))
    return db_module


def print_kwargs(self, method, kwargs):
    print ('{} {} object with data: {}'.format(method, self.db_module, ','
        .join(['{}: {}'.format(k, v) for k, v in kwargs.items()])))

def __len__(self):
    return len(self.get())

def to_results(self, remove_keys=None):
    '''
    Iterate over the list of raw results.
    Remove unwanted keys from the results.
    Append the cleaned results to a list of dicts.
    :param: remove_keys - custom list of additional keys remove
    '''
    REMOVE_KEYS = ['_sa_instance_state', 'deleted_at',
    'created_at', 'updated_up']
    if remove_keys:
        REMOVE_KEYS += remove_keys
    results = []
    for result_set in self.results:
        #  Make a copy as we are iterating the data we are mutating
        result_dict = copy.copy(vars(result_set))
        for remove_key in REMOVE_KEYS:
            del result_dict[remove_key]
        results.append(result_dict)
    return results

def add_many_to_many(self, parent, children, relationship):
    ''' Build m-t-m records based on the relationship key
    '''
    many_to_many_map = {
    'institutions': parent.institutions,
    'users': parent.users,
    'courses': parent.courses
    }
    build_data = many_to_many_map.get(relationship)
    for child in children:
        build_data.append(child)
    db.session.commit()
    return build_data
```

Figure 9: Helper methods in the BaseService class. (See Appendix B).

The method '*to_results' (Figure 9) is* used to pre-process any data that is returned from raw database lookups from an incoming GET request to a service. The '*add_many_to_many' (Figure 9)* method provides a means of creating child records in relational join tables. This can be triggered either directly (e.g. when a student makes a POST request to *catalogue/enrol* to sign up to a module/course) or as a side effect of a record being added to a parent table (e.g. when an institution adds a module through the *catalogue/add* endpoint). The '*dynamic_module'* method (*Figure 9)* is critical as it dynamically loads the service and database modules required to interact with the service that is extending the BaseService at runtime.

The BaseService also contains several CRUD methods to be invoked when HTTP requests are made. This is challenging as services which do not yet exist may eventually extend the BaseService. In order to achieve this agnostic extensibility of the BaseService the attribute *db_module* was added to all services extended from the BaseService. This attribute dictates with which database model a given service should interact when a POST or GET request is made.

```python
class CourseService(BaseService):
    def __init__(self):
        self.model = 'Course'
        self.db_module = self.dyanmic_module()
        super(CourseService)
```

```python
def create(self, **kwargs):
    kwargs['created_at'] = datetime.now()
    data = self.db_module(**kwargs)
    db.session.merge(data)
    try:
        db.session.commit()
        return self.get_raw(**kwargs)[0]
    except exc.IntegrityError as e:
        db.session.rollback()
        print('IntegrityError')
```

Figure 10: The instantiation of the CourseService from the BaseService(Left – see: appendix B). Here we see the attribute self.db_module being set using the self.dynamic_module() method discussed above. The create() method(right – see appendix D) comes from the BaseService class, and shows the referencing of self.db_module.

The *'soft_delete'* method is noteworthy for pragmatic reasons. When called, a deleted_at timestamp is set on the database record, this is advantagous as it is less drastic than a hard delete and preserves an audit trail in the data base (Heckman, R., 2016, pg. 60). The method checks that the source that made the DELETE request is the same source that created the record. If this check fails a *403 forbidden* status will be raised.

```python
# # # CRUD METHODS # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    def create(self, **kwargs):
        kwargs['created_at'] = datetime.now()
        data = self.db_module(**kwargs)
        db.session.merge(data)
        try:
            db.session.commit()
            return self.get_raw(**kwargs)[0]
        except exc.IntegrityError as e:
            db.session.rollback()
            print('IntegrityError')

    def get(self, **kwargs):
        ''' Query model and return jsonified repsonse
        '''
        kwargs['deleted_at'] = None
        self.results = self.db_module.query.filter_by(**kwargs).all()
        return {"status": "ok", "data": self.to_results()}

    def get_raw(self, **kwargs):
        ''' Returns raw, unjsonified db level data
        '''
        return self.db_module.query.filter_by(**kwargs).all()

    def get_by_id(self, id):
        self.results = [self.db_module.query.filter_by(id=id).first()]
        return {"status": "ok", "data": self.to_results()}

    def get_by_id_raw(self, id):
        return self.db_module.query.filter_by(id=id).first()

    def update(self, id, **kwargs):
        self.print_kwargs('Updating', kwargs)
        row = self.get_by_id(id)
        kwargs['updated_at'] = datetime.now()
        for key, value in kwargs.items():
            setattr(row, key, value)
        db.session.commit()

    def delete(self, id):
        row = self.get_by_id(id)
        print('Deleting {}'.format(row))
        User.query.filter(User.id == id).delete()
        db.session.delete(row)


    def soft_delete(self, type, id, requestor_id, reference):
        ''' Soft delete an item by setting its deleted at timestamp.
            If the requestor_id does not have permission for the deletion
            a 403 will be raised
            :param: type - type of item to delete
            :param: id - id of item to delete
            :param: requestor_id - id party requesting the deletion
        '''
        to_delete = self.get_by_id_raw(id)
        approved = [m.id for m in to_delete.reference.all()]
        if requestor_id in approved:
            print('Soft deleting {}'.format(to_delete.name))
            to_delete.deleted_at = datetime.now()
            db.session.merge(to_delete)
            return{' message': '{}: {} requested for deletion. Our \
            administrators will get back to you to confirm shortly.'.
            format(type, to_delete.name)}
        else:
            return{'message': 'Forbidden', 'status': 403}
```

Figure 11: BaseService CRUD operations. Full output in appendix B

Testing the Catalogue Service

In order to test the services a seed was used to create dummy data (see Appendix I). This seed file is a good means of testing the backend elements of the services, as create methods are called on the various models to insert data. The following traces show various inbound requests to the running catalogue service:

Figure 12: Initialisation of the MightMooC application

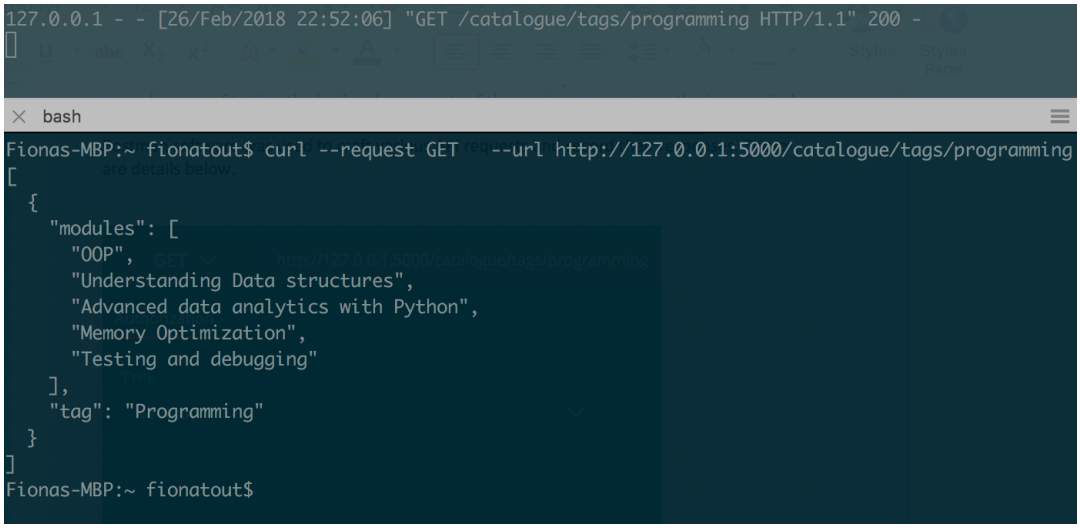

Figure 13: Get request for items in the catalogue with the 'programming' tag. (See Appendices C, K, and H).

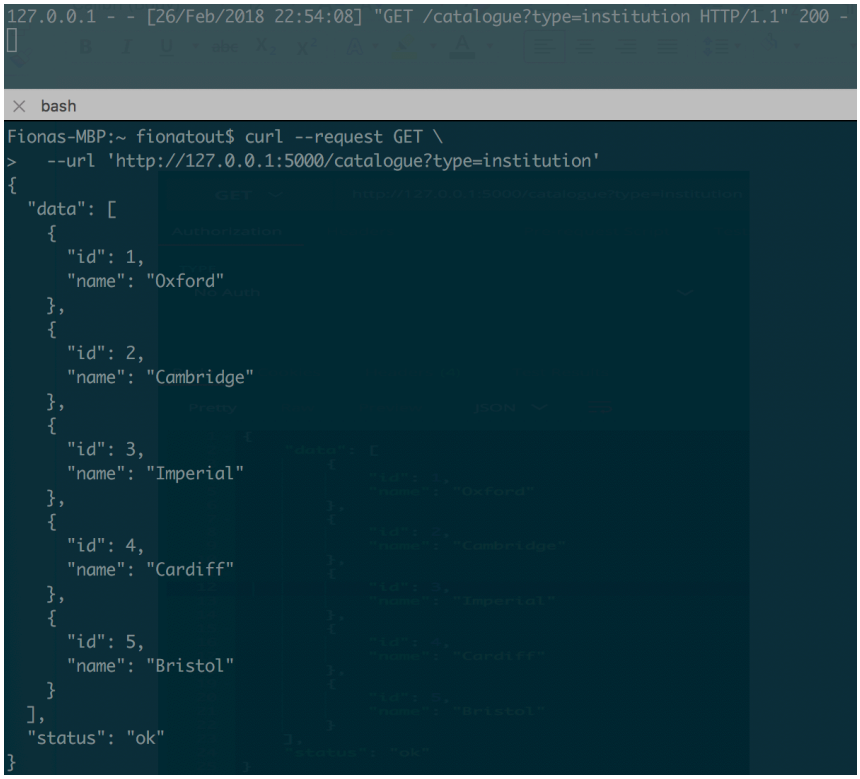

Figure 14: GET request to catalogue with the 'type 'parameter in the request arguments. Note, that send type=course or type=module are also suitable request arguments to this endpoint. (See Appendices C, F, and K).

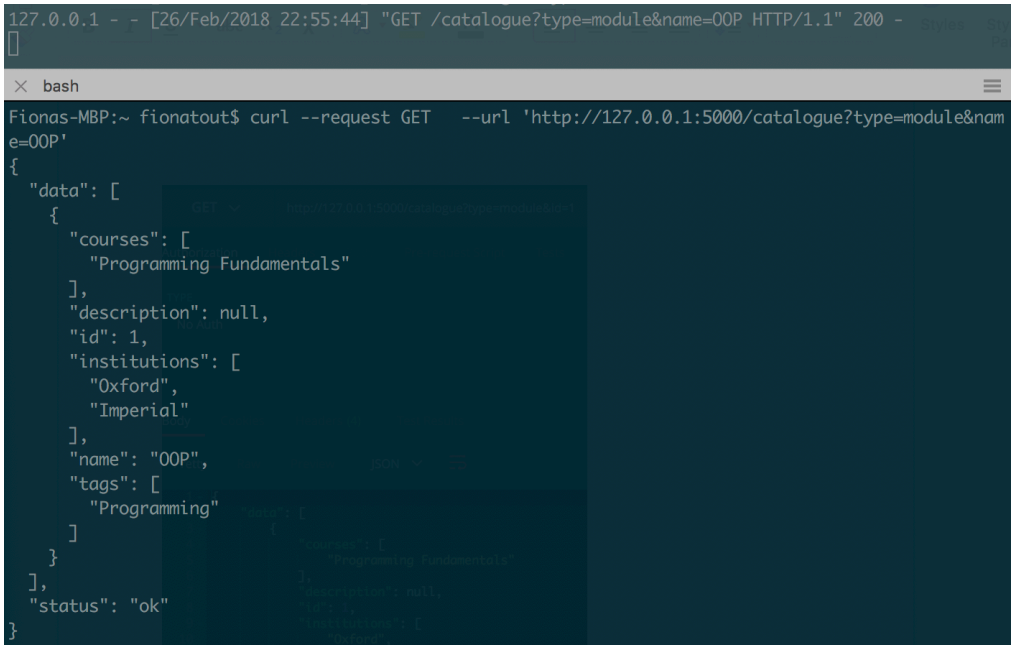

Figure 15: GET request to catalogue. This follows the same pattern as Figure 15 above, but here we see an additional parameter sent in the URL using the ampersand syntax. (See Appendices C, E and K).

```
def get(self, **kwargs):
    ''' Query model and return jsonified repsonse
    '''
    kwargs['deleted_at'] = None
    self.results = self.db_module.query.filter_by(**kwargs).all()
    return {"status": "ok", "data": self.to_results()}
```

Figure 16: Get method from the BaseService class. (See Appendix B).

Figure 16 shows the BaseService.get() method inherently used for all GET requests in the service. Here the python **kwargs syntax indicates that the endpoint accepts an unspecified dictionary as an input. With this syntax it is possible to query any data in the model without explicitly setting the request parameters. For example, the request *'http://127.0.0.1:5000/catalogue?type=module&id=1&name==OOP'* is also valid. Employing this method in a production environment should be done with a good deal of care to gate and validate the legitimacy of the incoming request. Allowing unspecified arguments could present a security vulnerability in the form of an injection attack if the arguments are not pre-validated (George, 2016., pg. 216).

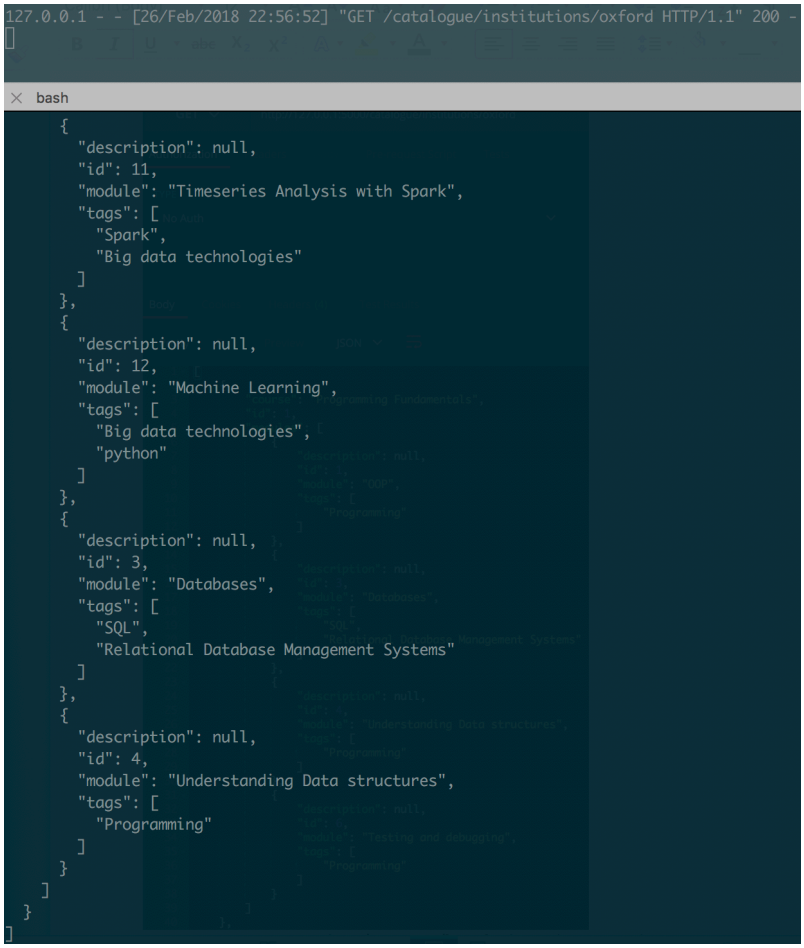

Figure 17: Get request to catalogue/institutions/[name] – this results in a nested JSON response of the courses and their modules that the institution contributes content towards (see Appendices C, F, and K).

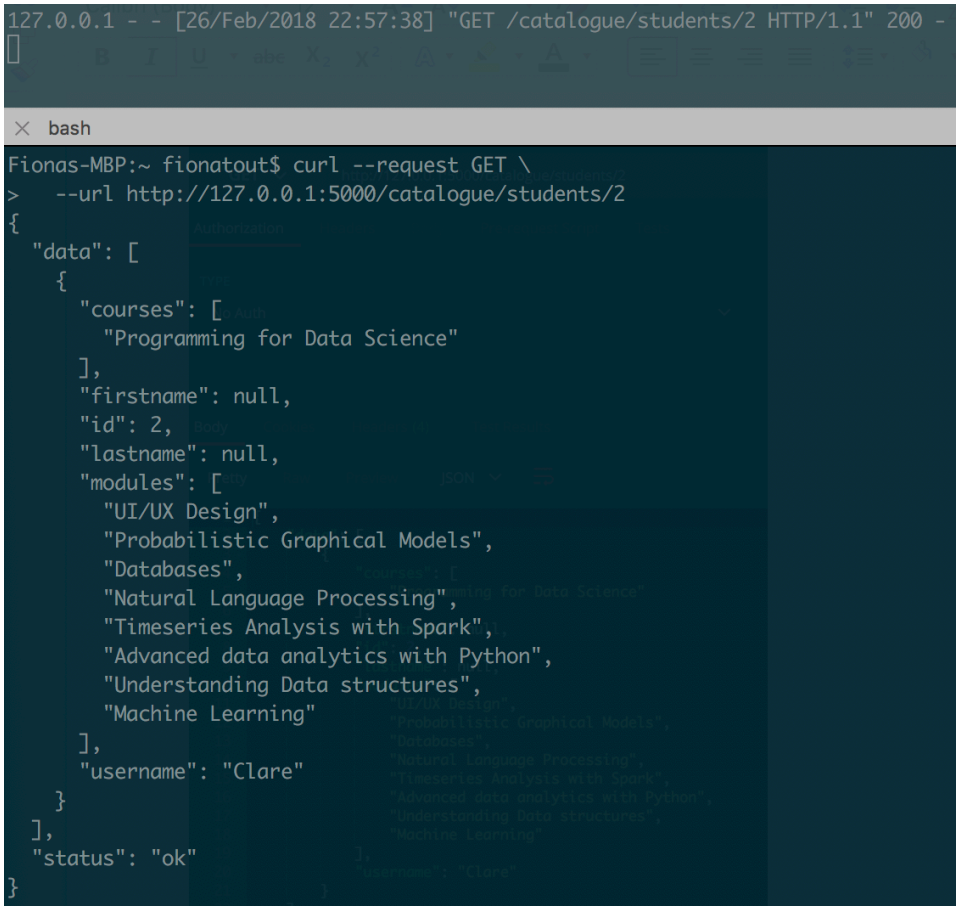

Figure 18: GET request to the catalogue/student/[id] endpoint – here we see all courses and modules that the student has enrolled on. (See Appendices C, G and K).

POST requests

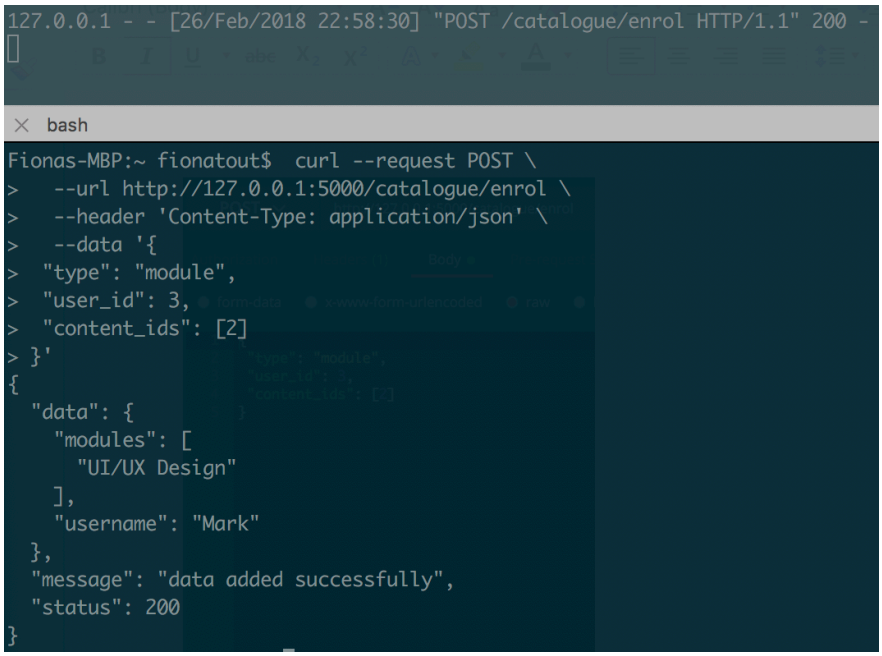

Figure 19: A POST request to the catalogue/enrol service. The response shows us that the user 'Mark' has enrolled on the 'UI/UX Design' Module. (See Appendices C, E and K).

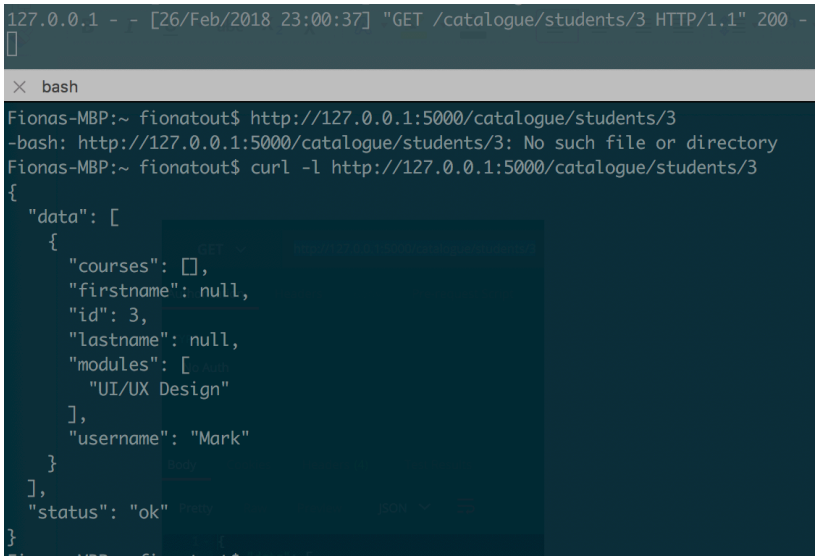

Figure 20: Verification of the above POST request by making a GET request to catalogue/students. (See Appendices C, D, E and K).

Figure 21: A POST request to catalogue/enrol this time giving the 'type' argument in the JSON body as 'course'. This has the effect of enrolling the student on the course and all modules that are children of the course (output abridged). (See Appendices C, D, E and K).

```python
@app.route('/catalogue/enrol', methods=['POST', 'DELETE'])
def enrolment():
    ''' Used to enrol a user on a module or course
        :param: content_id - maps to a module or course depending on type param
        :param: type - string - 'module' or 'course'
    '''
    request_data = request.get_json()
    service = SERVICE_ROUTER.get(request_data.get('type'))
    if request.method == 'POST':
        response = service.enrol(request_data['user_id'],
            request_data['content_ids'])
        return jsonify({'message': 'data added successfully', 'data': response,
         'status': 200})
```

Figure 22: Here we see the code that handles the above enrolment requests. The SERVICE_ROUTER.get dictionary is used to send the request to create the enrolments to either courses or modules based on the inbound JSON 'type' data. If a course is selected, the course.enrol service iteratively subscribes the user to every module in that course (see appendix D).

```
127.0.0.1 - - [26/Feb/2018 23:02:39] "POST /catalogue HTTP/1.1" 200 -
```

```
bash
Fionas-MBP:~ fionatout$ curl --request POST \
>   --url http://127.0.0.1:5000/catalogue \
>   --header 'Content-Type: application/json' \
>   --data '{
>  "type": "module",
>  "name": "SOA",
>  "institution_id": 2
> }'
{
  "data": {
    "name": "SOA"
  },
  "message": "data added successfully",
  "status": 200
}
```

Figure 23: POST request to catalogue. Here institution_id is used to add a record in the join table module_institutions. (See Appendices C, E, and K).

DELETE requests:

```
Soft deleting UI/UX Design
127.0.0.1 - - [26/Feb/2018 23:03:26] "DELETE /catalogue HTTP/1.1" 200 -
```

```
bash
Fionas-MBP:~ fionatout$ curl --request DELETE \
>   --url http://127.0.0.1:5000/catalogue \
>   --header 'Content-Type: application/json' \
>   --data '{
>  "type": "module",
>  "id": 2,
>  "institution_id": 2
> }'
{
  "message": "Our administrators will get back to you shortly",
  "status": 200,
  "to_delete": "UI/UX Design"
}
```

Figure 24: Soft delete of a module from an institution - note this results in a 200 success response as the institution_id 2 added the module originally (see seed file in Appendix I, Appendices C, E and K)

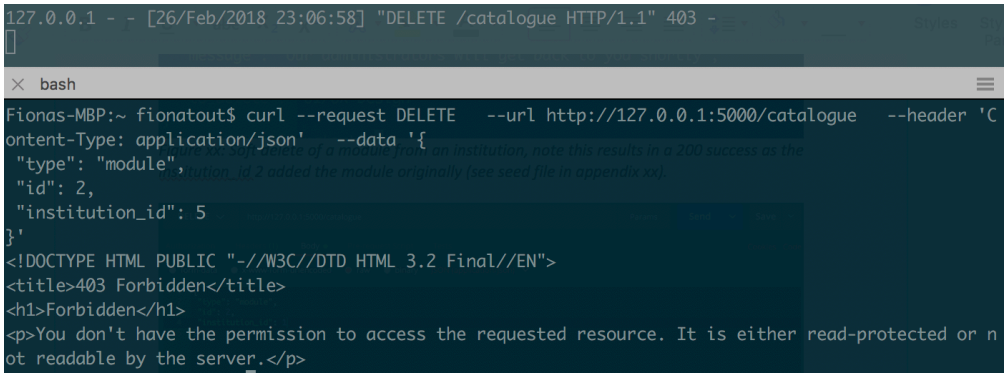

Figure 25: The equivalent deletion request is made by an institution that is not the content owner. This time a *403 forbidden* response is raised. This mechanism is handled in the BaseClass.soft_delete() method detailed in figure 10. *(See Appendices C, E and K).*

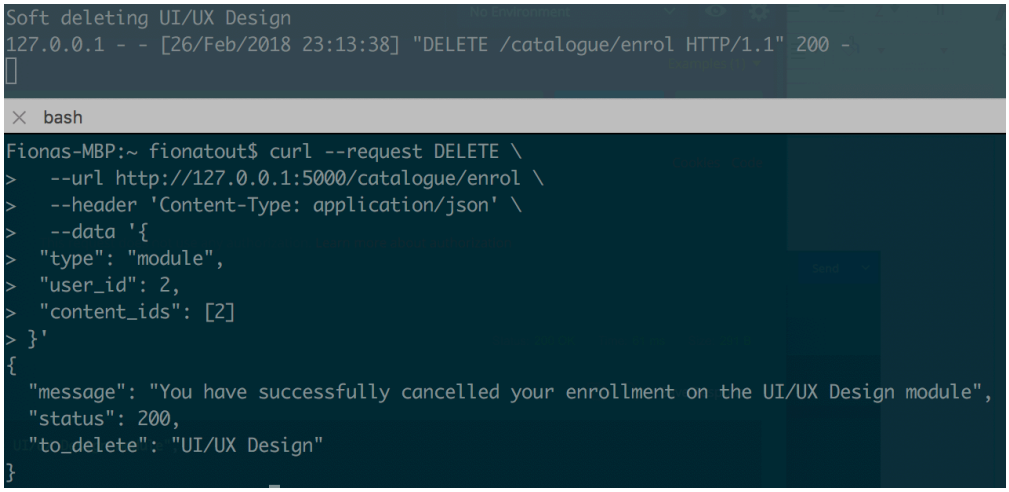

Figure 26: DELETE request to catalogue/enrol – here the user deletes an enrolment on a module. The BaseClass.soft_delete() function is again called to handle this deletion. (See Appendices C, E and K).

Question 2:

A key design decision is whether a centralised database of content should be used or should institutions store all content locally allowing the MightMooC service to connect to the content providers and *lazily* access the content upon request from a client?

Decentralised content storage has several pragmatic advantages as it allows institutions to maintain their own content locally. This means any issues of sharing intellectual property with a third-party service, and the security concerns that come with this, are avoided.

There is an interesting consideration in the decentralised content infrastructure regarding the relationship between course popularity and bandwidth constraints on the course providers. If content was POSTed to the MightyMooC service lazily upon request, *the bandwidth consumed by the course provider would be a function of the number of concurrent requests*. Given the assumption that the more subscribers on a course the greater probability of concurrent requests, course providers would become burdened with a high volume of requests at spiking times. Put another way, *the predicted quality of service for a given module of content is inversely proportional to the number of subscribers to that content.* This is a serious limitation and one that could affect the success of the platform.

Internal caching servers such as *Redis* could offer a solution to this problem by temporarily storing the content following a GET request from an external client. MightyMooC would then make a single GET request to the content provider and cache the response. This has the additional benefit of optimising performance through

data locality. Cache servers could be physically located in a cloud infrastructure in various global regions, enabling regional caching and serving of data to match the origin of inbound client GET requests.

This design is somewhat flawed in that it breaks the principle of *service loose coupling (Erl, 2016. Pg. 293)*. In the decentralised storage model there is tight coupling between the course provider and the content being served. If a university suffered an availability fault in its own infrastructure, subscribed students would not be able to access the enrolled content. Whilst arguably this is a greater issue in a centralised data storage platform as it introduces a single point of failure for all content, having data storage handled internally allows for controls, optimisations and provisions for redundancy to reduce risk of service disruption.

A further advantage of using a centralised storage platform is bandwidth efficiency. If all service providers were using in house storage mechanisms to serve data to MightyMooC, data would be sent whenever MightyMooC made a request for content. With a centralised data platform, MightyMooC would simply retrieve the requested data internally and serve it to the client. This brings security benefits as the data is less frequently in transit, meaning less opportunity for interception attacks.

A final more pragmatic advantage of storing content centrally is that it mitigates any chance of content providers drastically changing content out of the control of MightyMooC. Whilst content providers should have some control over their content, if given complete autonomy, the ephemeral nature of the content could lead to problems for end users. For example, users enrolling at the same time may see differing material for a given module. Centralised content also enables the specification and documentation of the expected format of content. This allows for better interoperability with external APIs and dynamic content generation on the frontend.

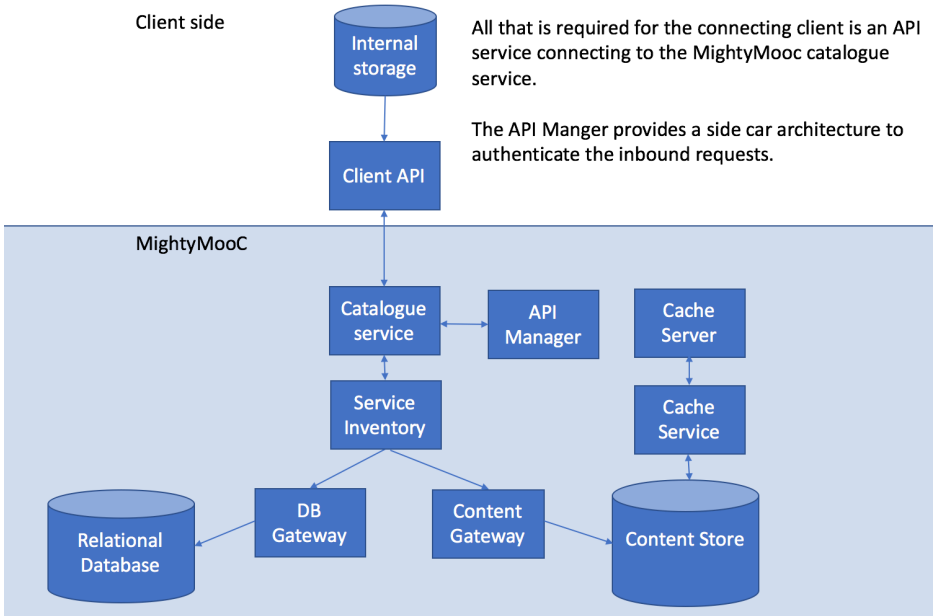

Figure 27: External connection to MightyMooC from a participating client

Part C: Non-functional Requirements

Question 1:

Integrity is ensured with the use of *soft delete* and *soft edits*. All attempts to delete/edit content in the catalogue are logged and verified by MightyMooC admins after conferral with the Course Providers. In cases where multiple content providers had contributed to module content, updates would be atomic and only permissible for content relative to the content originator (see Figure 28).

Any materials submitted to MightyMooC are confidential and viewable only to those who have uploaded the module or who have paid to enrol on it. Summary details of the entire catalogue are available to all users. The platform would use the TLS protocol to ensure that '*man-in-the-middle'* attacks could only eavesdrop on encrypted data. Sensitive data would also be encrypted at rest in the database to mitigate the risk of outside attacks.

The use of an internal central content store with redundancy and elastic scalability would ensure that a good level of availability was guaranteed in Service Level Agreements(SLAs), minimising the chance of service disruption for end users. The platform would use load balancers ensuring that physical servers were not overloaded with inbound requests; increasing the complexity of a denial of service attack.

Oauth2 would be implemented across the platform as the main means of authorising users logging into the system. A three-legged Oauth approach would be implemented for token based log in through third party applications such as Facebook and Linkedin, allowing for ease of sharing achievements gained through MightyMooC.

In the fully working SOA, a hybrid of role and policy based access controls would be implemented. Role based control policies would cover many scenarios including only allowing institution users to add/remove modules, and students to only sign themselves up for modules. Policy based control would be used for more nuanced access decisions such as requests to update content on a module which has multiple content providers. Here policy would enforce that if the updater was the same as the original poster of the content then that content could be updated, otherwise content could be requested to be updated but the original content provider would be notified and asked for approval. By preventing the user from changing content with complete flexibility, a system of non-repudiation is created as all content change requests are logged and specifically approved by the system and admin users.

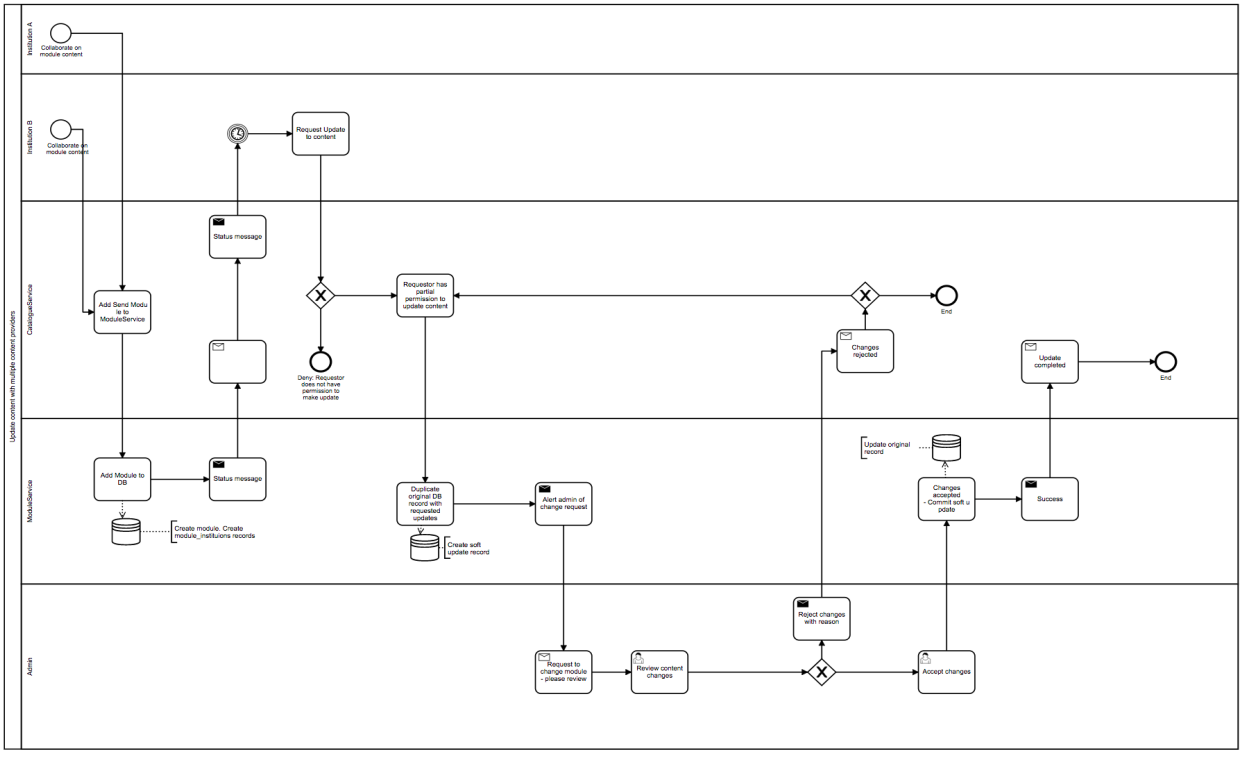

Figure 28: BPMN diagram of the content update process for multiple content owners.

Question 2:

This system has been built using Python and the REST library Flask. An object oriented interoperable approach was taken to the build process, in which all services are children of a base service class and inherit global methods such as those required by CRUD requests.

This enables for flexible extensibility of services and improved interoperability across services. It is also in line with the principle of *service agnosticism.* Individual service layers are sparse veneers containing little custom logic on top of the underlying base service class. Connecting service consumers do not need to concern themselves with the logic of the more complex base class, and can freely and simply interact via HTTP requests to the derived child services. This ties with the principle that service adoption and composability is inversely proportional to the amount of business process specific logic contained within a service (Erl, 2016, 2016, pgs. 35-37).

The platform is deployed using docker containers. Docker is a platform that enables software to be containerised and bundled into a runtime environment in which all of its dependencies pre-exist. This is in line with service autonomy (Erl, 2016, pg. 297), as it ensures that the services are run in self-contained, tailored environments, circumventing the risk of 'architectural drift'.

This project features several endpoint tests of the Catalogue service created using the testing suit Postman. In a full implementation of the architecture this would also extend to include backend unit testing to ensure that the underlying logic was sound. For frontend API tests an integration testing suit such as Behave could be implemented to ensure that the user interaction through the platform matches the expected desired outcomes of the system.

Question 3.

The ancillary analytics service would be used not only to gleam insights into user behaviour (trend in courses/subject matter) but also to highlight quality assurance issues with the content itself. For example, students would have the ability to rate modules which they complete, with consistently low rated modules flagged for review. These ratings would be normalised against the performance of the grade achieved by the student to protect 'spite marking' of module content.

Part D. Conclusions

Question 1.

An API manager would be used to handle identity management and validating the structure of content submitted from course providers. If a course provider were to submit a malformed JSON document (or valid JSON outside of the scope required schema) the API manager would reject the upload and inform the client of the error. This also has an added security benefit, as data gating at the API manager phase would prevent malicious or malformed JSON content from causing service outage due the back end not being able to process the data. This could potentially form an asymmetric denial of service attack in which heavy server resource consumption is triggered.

In its current form an ESB is not necessary for this architecture. The tightly defined JSON structure for content uploads limit the requirement for data transformations and Media uploads such as PDFs or video files fall outside of the scope of transformations available to an ESB. A potential use case for an ESB would be to farm portions of data in transit to sidecar services such as data processing. This may be a consideration for adoption as the platform grew but is too far removed from the core business proposition to be an initial architectural consideration.

Question 2

The service is relatively coarse grained. Comparing system generated JSON output to the original database models (Appendix J) reveals that granular metadata such as created_at and updated_at timestamps have been omitted in the response. This is illustrated through the use of '*get'* and 'get_*raw'*. Both have the same core functionality but get_raw does not pre-process the result. This means that *get_raw* is consumed by internal operations only, whereas *get* is exposed through an API. Although the same result could have been achieved through using a Boolean such as '*raw=True,'* this would lead the functionality of the method to change depending on its input, leading to unpredictable results if misused. A design decision was therefore made to

separate out internal usage functions that required data to be returned in raw format from consumable responses through an API endpoint.

```
def get(self, **kwargs):
    ''' Query model and return jsonified repsonse
    '''
    self.results = self.db_module.query.filter_by(**kwargs).all()
    return make_response(jsonify({"status": "ok", "data":
        self.to_results()}), 200)

def get_raw(self, **kwargs):
    ''' Returns raw, unjsonified db level data
    '''
    return self.db_module.query.filter_by(**kwargs).all()
```

Figure 29: Get and Get raw methods of the BaseService class. (See Appendix B)

The architecture diagram for this service highlights the possibility of three microservice candidates at the storage layer, namely the cache service, database service and content store. A key design point for whether to use a microservices architecture lies in whether the services have any specific requirements unique to that service (Erl, 2016. Pg. 223). These requirements may be technological choke points which specific infrastructure considerations, such as very high throughput network connectivity or specific storage or processing capabilities. For these reasons, the services identified are logical microservices candidates were the architecture to be explored further.

Question 3

Composability of services is achieved through the use of a central base service. Global utility functions which were ubiquitous to all services (such as CRUD operations) could then be created. This enables composability of future services as the base service acts as a service blueprint for the architecture, whilst being flexible enough for services to overload any inherited methods with their own functionality. An example of this practice can be seen in the catalogue service which overrides the *'get'* method. As the catalogue service has been built as an endpoint through which modules, courses and institutions can all be accessed (see Figure 8), the service must be able to dynamically query any one of the models depending on the request.

```
def get(self, **kwargs):
    '''
    Returns a module, course, or institution by the given id
    '''
    service = self.CATALOGUE_ROUTER.get(kwargs['type'])
    del(kwargs['type'])
    return service.get(**kwargs)
```

Fig 30 The catalogue service wraps many other services and directs the request according to the 'type' argument in the request headers. The CATALOGUER_ROUTER argument is a python dictionary which contains mappings to the services that the catalogue accesses. (See Appendix C)

```
def to_results(self):
    '''
    Iterate over the list of raw results.
    Remove unwanted keys from the results.
    Append the cleaned results to a list of dicts.
    '''
    REMOVE_KEYS = ['_sa_instance_state', 'deleted_at',
     'created_at', 'updated_at']
    results = []
    for result_set in self.results:
        #  Make a copy as we are iterating the data we are mutating
        result_dict = copy.copy(vars(result_set))
        for remove_key in REMOVE_KEYS:
            del result_dict[remove_key]

        results.append(result_dict)
    return results
```

Figure 31: BaseService.to_results() – This method is called by 'get' by not 'get_raw' and acts as a pre-processing mechanism to prune unwanted data from the JSON response. (See Appendix B).

The use of extendable URL patterns such as '*http://127.0.0.1:5000/catalogue?type=module&id=' (Figure 30)* creates complexity as the query parameters are customisable. Whilst this is excellent for querying the data, it paradoxically can mean that data is harder to access than through a URL with defined expected arguments. If the service consumers are unaware of what they can query in the model then the benefit of this flexibility is lost.

Further complexity arises with the need to implement role based access control across the data that is being queried. For example, both student and institution roles would have the ability to send in the same request data in the URL. It is expensive to perform access control checks on every inbound variable in the URL string relative to the user before permitting or rejecting the request. Whilst this pattern was employed with the goal of increased flexibility, a revised design would use more static routes and parameters to avoid this added complexity.

Question 4:

Building with interoperability and service composability concepts is a challenging design principle. In order to maximise reusability, services have to be sufficiently agnostic to any business process. Erl argues that the reuse potential of a service is inversely proportional to the quantity of business process specific logic contained in that service (pgs. 35-37). Ideally any given service can be adopted as part of any service composition should its functionality be suited to the purpose. This holy grail of interoperable services is challenging to build as it requires strict adherence to predefined service contracts and design approaches. Build and output inconsistencies will result in the failure of service composability (Erl, 2016, pg. 44). Visibility across the platform from both developers and product managers is essential for this to be a success. This gives credence to the core principle of discoverability across services through a programmatic service inventory, or a global documentation of services and their input/outputs. The successful implementation of SOA can be seen as a function of the discoverability and composability of the services.

To achieve the goal of *service reusability,* conditional logic must be avoided from the control flow wherever possible. This is challenging when considering database interaction, as in order to retrieve and create records in the database we need to explicitly call the target tables. Boolean logic fails to achieve service reusability as it cannot predict future service subscribers. Consider the logical flow:

If A:
Query table A
Else if B:
Query table B

This will work without issue if there are service consumers A and B however as soon a service C is added to the infrastructure this will error. The blueprint should be agnostic to handle any number of extended classes without knowledge of who those classes are or what their purpose is. This is in line with the principle of *service reusability* (Erl, 2016, pg. 295) and the '*agnostic capability design pattern'* (Erl, 2016, pg. 322).

The balance between top down and bottom up architecture is a key consideration when implementing SOA. Erl, is an advocator of a fully top down approach with iterative application of service orientated design principles throughout the design phases in order to evolve services which are optimally interoperable and composable. In reality this is very challenging, even in small projects with one team member such as *this project*. With a top down design process comes the overhead of increased pre-build requirements. Top down designs front-load the required man hours due to a greater emphasis placed on problem decomposition, analysing required service models and defining service contracts (Erl, 2016, pg. 94). However, the alternative of allowing developers to build first and define contracts and documentation after or during the build process (bottom up design) for the gain of quick iteration and agile development comes with a high probability that the code produced will be less interoperable to both internal and external service consumers.

As a project progresses it naturally evolves, unexpected challenges and edge cases can arise; subsequently new functionality and consumer roles can be added or removed. The planning and documentation generated from the top down design needs to be sufficiently flexible to facilitate the organic evolution of a project. A tightly scoped top down project will only succeed if its implementation is unobstructed by inflexibility and rigour.

References:

Erl, T., 2016. Service-Oriented Architecture. Prentice Hall.
Bloomberg, J., 2013. The Agile Architecture Revolution. John Wiley & Sons.
Heckman, R., 2016. Designing Platform Independent Mobile Apps and Services. John Wiley & Sons.
George, N., 2016. Mastering Django: Core. Packt Publishing Ltd.

Bibliography

https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world. (2018). [Blog].

Appendix A: Postman API documentation for catalogue service

GET http://127.0.0.1:5000/catalogue/tags/programming

http://127.0.0.1:5000/catalogue/tags/programming

Search for modules matching the given tag.

Sample Request

http://127.0.0.1:5000/catalogue/tags/programming

```
curl --request GET \
  --url http://127.0.0.1:5000/catalogue/tags/programming
```

Sample Response

```
[
  {
    "modules": [
      "OOP",
      "Understanding Data structures",
      "Advanced data analytics with Python",
      "Memory Optimization",
      "Testing and debugging"
    ],
    "tag": "Programming"
```

GET http://127.0.0.1:5000/catalogue/institutions/oxford

http://127.0.0.1:5000/catalogue/institutions/oxford

Search for institution by name and list all content offered by that institution.

Sample Request

http://127.0.0.1:5000/catalogue/institutions/oxford

```
curl --request GET \
  --url http://127.0.0.1:5000/catalogue/institutions/oxford
```

Sample Response

```
[
  {
    "course": "Programming Fundamentals",
    "id": 1,
    "modules": [
      {
        "description": null,
        "id": 1,
        "module": "OOP",
        "tags": [
```

GET http://127.0.0.1:5000/catalogue/?type=institution

http://127.0.0.1:5000/catalogue?type=institution

list institutions

PARAMS

type	institution

Sample Request

http://127.0.0.1:5000/catalogue/?type=institution

```
curl --request GET \
  --url 'http://127.0.0.1:5000/catalogue?type=institution'
```

Sample Response

```
{
  "data": [
    {
      "id": 1,
      "name": "Oxford"
    },
    {
      "id": 2,
      "name": "Cambridge"
    },
```

GET http://127.0.0.1:5000/catalogue/students/2

http://127.0.0.1:5000/catalogue/students/2

List enrolled subjects by student_id

Sample Request

http://127.0.0.1:5000/catalogue/students/2

```
curl --request GET \
  --url http://127.0.0.1:5000/catalogue/students/2
```

Sample Response


```
{
  "data": [
    {
      "courses": [
        "Programming for Data Science"
      ],
      "firstname": null,
      "id": 2,
      "lastname": null,
      "modules": [
```


POST http://127.0.0.1:5000/catalogue/enrol

http://127.0.0.1:5000/catalogue/enrol

Enrol a student on an individual module.

HEADERS

Content-Type	application/json

BODY


```
{
 "type": "module",
 "user_id": 3,
 "content_ids": [2]
}
```


Sample Request

http://127.0.0.1:5000/catalogue/enrol

```
curl --request POST \
  --url http://127.0.0.1:5000/catalogue/enrol \
  --header 'Content-Type: application/json' \
  --data '{
 "type": "module",
 "user_id": 3,
 "content_ids": [2]
}'
```

Sample Response


```
{
  "data": {
    "modules": [
      "UI/UX Design"
    ],
    "username": "Mark"
  },
  "message": "data added successfully",
  "status": 200
}
```


POST http://127.0.0.1:5000/catalogue/enrol

http://127.0.0.1:5000/catalogue/enrol

Enrol a student on a course and all of the modules of that course.

HEADERS

Content-Type	application/json

BODY


```
{
 "type": "course",
 "user_id": 4,
 "content_ids": [2]
}
```


Sample Request

http://127.0.0.1:5000/catalogue/enrol

```
curl --request POST \
  --url http://127.0.0.1:5000/catalogue/enrol \
  --header 'Content-Type: application/json' \
  --data '{
 "type": "course",
 "user_id": 4,
 "content_ids": [2]
}'
```

Sample Response


```
{
  "data": {
    "course": "Programming for Data Science",
    "id": 2,
    "modules": [
      {
        "description": null,
        "id": 8,
        "module": "Probabilistic Graphical Models",
        "tags": [
```

POST http://127.0.0.1:5000/catalogue

http://127.0.0.1:5000/catalogue

Add a new module to the Catelogue.

HEADERS

Content-Type	application/json

BODY

```
{
 "type": "module",
 "name": "SOA",
 "institution_id": 2
}
```

Sample Request

http://127.0.0.1:5000/catalogue

```
curl --request POST \
  --url http://127.0.0.1:5000/catalogue \
  --header 'Content-Type: application/json' \
  --data '{
 "type": "module",
 "name": "SOA",
 "institution_id": 2
}'
```

Sample Response

```
{
  "data": {
    "name": "SOA"
  },
  "message": "data added successfully",
  "status": 200
}
```

GET http://127.0.0.1:5000/catalogue?type=course&id=1

http://127.0.0.1:5000/catalogue?type=course&id=1

Get catalogue data by its type and ID. Accepted types: 'module', 'course', 'institution'

HEADERS

Content-Type	application/json

PARAMS

type	course
id	1

Sample Request

http://127.0.0.1:5000/catalogue?type=course&id=1

```
curl --request GET \
  --url 'http://127.0.0.1:5000/catalogue?type=course&id=1' \
  --header 'Content-Type: application/json'
```

Sample Response

```
{
  "data": [
    {
      "description": "An introduction to programming.",
      "id": 1,
      "level": null,
      "name": "Programming Fundamentals"
    }
  ],
  "status": "ok"
```

DELETE http://127.0.0.1:5000/catalogue

http://127.0.0.1:5000/catalogue

Institution request to delete a module.

HEADERS

Content-Type	application/json

BODY

```
{
 "type": "module",
 "id": 2,
 "institution_id": 2
}
```

Sample Request

http://127.0.0.1:5000/catalogue

```
curl --request DELETE \
  --url http://127.0.0.1:5000/catalogue \
  --header 'Content-Type: application/json' \
  --data '{
 "type": "module",
 "id": 2,
 "institution_id": 2
}'
```

Sample Response

```
{
  "message": "Our administrators will get back to you shortly",
  "status": 200,
  "to_delete": "UI/UX Design"
}
```

GET http://127.0.0.1:5000/catalogue?type=course&id=1

http://127.0.0.1:5000/catalogue?type=course&id=1

get by id

PARAMS

type	course
id	1

Sample Request

http://127.0.0.1:5000/catalogue?type=course&id=1

```
curl --request GET \
  --url 'http://127.0.0.1:5000/catalogue?type=course&id=1'
```

Sample Response

```
{
  "data": [
    {
      "description": "An introduction to programming.",
      "id": 1,
      "level": null,
      "name": "Programming Fundamentals"
    }
  ],
  "status": "ok"
```

DELETE http://127.0.0.1:5000/catalogue/enrole

http://127.0.0.1:5000/catalogue/enrol

Remove a student from an enroled on a module.

HEADERS

Content-Type	application/json

BODY

```
{
 "type": "module",
 "user_id": 2,
 "content_ids": [2]
}
```

Sample Request

http://127.0.0.1:5000/catalogue/enrole

```
curl --request DELETE \
  --url http://127.0.0.1:5000/catalogue/enrol \
  --header 'Content-Type: application/json' \
  --data '{
 "type": "module",
 "user_id": 2,
 "content_ids": [2]
}'
```

Sample Response

```
{
  "message": "You have successfully cancelled your enrolment on the UI/UX
  "status": 200,
  "to_delete": "UI/UX Design"
}
```

GET http://127.0.0.1:5000/catalogue?type=institution

http://127.0.0.1:5000/catalogue?type=institution

List all Catalogue items by type. Accepeted types: 'module', 'course', 'institution'

PARAMS

type	institution

Sample Request

http://127.0.0.1:5000/catalogue?type=institution

```
curl --request GET \
  --url 'http://127.0.0.1:5000/catalogue?type=institution'
```

Sample Response

```
{
  "data": [
    {
      "id": 1,
      "name": "Oxford"
    },
    {
      "id": 2,
      "name": "Cambridge"
    },
```


GET http://127.0.0.1:5000/catalogue?type=institution

http://127.0.0.1:5000/catalogue?type=institution

List all Catalogue items by type. Accepeted types: 'module', 'course', 'institution'

PARAMS

type	institution

Sample Request

http://127.0.0.1:5000/catalogue?type=institution

```
curl --request GET \
  --url 'http://127.0.0.1:5000/catalogue?type=institution'
```

Sample Response

```
{
  "data": [
    {
      "id": 1,
      "name": "Oxford"
    },
    {
      "id": 2,
      "name": "Cambridge"
    },
```

Appendix B: BaseService.py

```
from datetime import datetime
import importlib
import copy
import mightyMooc.models
from sqlite3 import IntegrityError
from sqlalchemy import exc
from mightyMooc import app, db
from flask import jsonify, request, make_response, abort


class BaseService():

    def __init__(self):
        self.db_module = self.dyanmic_module()

# # # # # # # # # HELPER METHODS # # # # # # # # # # # # # # # # # # # # #

    def dyanmic_module(self):
        ''' Dynmically load the relevant class from mightMooc.models
        '''
        mod = __import__('mightyMooc.models', fromlist=[str(self.model)])
        db_module = getattr(mod, str(self.model))
        return db_module


    def print_kwargs(self, method, kwargs):
        print ('{} {} object with data: {}'.format(method, self.db_module, ','
            .join(['{}: {}'.format(k, v) for k, v in kwargs.items()])))

    def __len__(self):
        return len(self.get())

    def to_results(self, remove_keys=None):
        '''
        Iterate over the list of raw results.
        Remove unwanted keys from the results.
        Append the cleaned results to a list of dicts.

        :param: remove_keys - custom list of additional keys remove
        '''
        REMOVE_KEYS = ['_sa_instance_state', 'deleted_at',
        'created_at', 'updated_up']
        if remove_keys:
            REMOVE_KEYS += remove_keys
        results = []
        for result_set in self.results:
            #  Make a copy as we are iterating the data we are mutating
            result_dict = copy.copy(vars(result_set))
            for remove_key in REMOVE_KEYS:
                del result_dict[remove_key]
            results.append(result_dict)
        return results

    def add_many_to_many(self, parent, children, relationship):
        ''' Build m-t-m records based on the relationship key
```

```
        '''
        many_to_many_map = {
        'institutions': parent.institutions,
        'users': parent.users,
        'courses': parent.courses
        }
        build_data = many_to_many_map.get(relationship)
        for child in children:
            build_data.append(child)
        db.session.commit()
        return build_data


# # # CRUD METHODS # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    def create(self, **kwargs):
        kwargs['created_at'] = datetime.now()
        data = self.db_module(**kwargs)
        db.session.merge(data)
        try:
            db.session.commit()
            return self.get_raw(**kwargs)[0]
        except exc.IntegrityError as e:
            db.session.rollback()
            print('IntegrityError')

    def get(self, **kwargs):
        ''' Query model and return jsonified repsonse
        '''
        kwargs['deleted_at'] = None
        self.results = self.db_module.query.filter_by(**kwargs).all()
        return {"status": "ok", "data": self.to_results()}

    def get_raw(self, **kwargs):
        ''' Returns raw, unjsonified db level data
        '''
        return self.db_module.query.filter_by(**kwargs).all()

    def get_by_id(self, id):
        self.results = [self.db_module.query.filter_by(id=id).first()]
        return {"status": "ok", "data": self.to_results()}

    def get_by_id_raw(self, id):
        return self.db_module.query.filter_by(id=id).first()

    def update(self, id, **kwargs):
        self.print_kwargs('Updating', kwargs)
        row = self.get_by_id(id)
        kwargs['updated_at'] = datetime.now()
        for key, value in kwargs.items():
            setattr(row, key, value)
        db.session.commit()

    def delete(self, id):
        row = self.get_by_id(id)
        print('Deleting {}'.format(row))
        User.query.filter(User.id == id).delete()
```

```
        db.session.delete(row)


    def soft_delete(self, type, id, requestor_id, deleted_by):
        ''' Soft delete an item by setting its deleted at timestamp.
            If the requestor_id does not have permission for the deletion
            a 403 will be raised
            :param: type - type of item to delete
            :param: id - id of item to delete
            :param: requestor_id - id party requesting the deletion
                e.g. 'modules'
            :param: deleted_by: str - string of the type of user attempting to
                    make the soft delete. e.g. 'institution / 'student'
        '''
        to_delete = self.get_by_id_raw(id)
        delete_dict = {'student': to_delete.users.all(),
                       'institution': to_delete.institutions.all(),
        }
        approved = [m.id for m in to_delete.users.all()]
        if requestor_id in approved:
            print('Soft deleting {}'.format(to_delete.name))
            to_delete.deleted_at = datetime.now()
            db.session.merge(to_delete)
            return{'status': 200, 'to_delete': to_delete.name,
                   'message': 'Our administrators will get back to you shortly'}
        else:
            abort(403)
```

Appendix C: Catalogue Service

```
from flask import abort

from mightyMooc import app, db
from mightyMooc.backend.base_service import BaseService
from mightyMooc.backend.module_service import ModuleService
from mightyMooc.backend.course_service import CourseService
from mightyMooc.backend.institution_service import InstitutionService
from mightyMooc.backend.tag_service import TagService

class CatalogueService(BaseService):
    '''
        A service for querying the whole mightyMooC catalogue
        adding module/course content, and
        for students to enrol on modules/courses
    '''
    def __init__(self):
        self.tag_service = TagService()
        self.module_service = ModuleService()
        self.institution_service = InstitutionService()
        self.course_service = CourseService()
        self.CATALOGUE_ROUTER = {
        'course': self.course_service,
        'module': self.module_service,
        'institution': self.institution_service,
        }

############ GET ##########################
```

```
  def get(self, **kwargs):
    '''
    Returns a module, course, or institution by the given id
    '''
    service = self.CATALOGUE_ROUTER.get(kwargs['type'])
    del(kwargs['type'])
    return service.get(**kwargs)


  def get_by_id(self, **kwargs):
    '''
    Returns a module, course, or institution by the given id
    '''
    try:
      service = self.CATALOGUE_ROUTER.get(kwargs['type'])
      return service.get_by_id(kwargs.get('id'))
    except:
      abort(404)

  def get_tags(self, tag):
    '''
    :param: tag: string
    :returns: JSON response of courses and modules matching the tag
    '''
    return self.tag_service.get_modules_by_tag(tag)

  def get_institutions(self, institution):
    '''
    :param: institution: string
    :returns: JSON response of courses and modules matching the institution
    '''
    return self.institution_service.build_institution_json(institution)
```

Appendix D: Course service

```
from mightyMooc import app, db
from mightyMooc.backend.base_service import BaseService
from mightyMooc.backend.module_service import ModuleService
from mightyMooc.backend.user_service import UserService

module_service = ModuleService()
user_service = UserService()

class CourseService(BaseService):
  def __init__(self):
    self.model = 'Course'
    self.db_module = self.dyanmic_module()
    super(CourseService)


  def enrol(self, user_id, course_id):
    ''' enrol a user to a course
      cascades to enrol them on every module in the course
```

```
        Currently built for one course id (even though a list)
        easily extendable to use multiple.
    '''
    user = user_service.get_by_id_raw(user_id)
    course = self.get_by_id_raw(course_id[0])
    module_ids = [module.id for module in course.modules]
    user.courses.append(course)
    module_service.enrol(user_id, module_ids)
    try:
        db.session.commit()
        return self.build_enrollment_json(course, user)
    except:
        db.session.rollback()


def build_enrollment_json(self, course, user):
    ''' Find all modules and courses subscribed to by a given user and
        build a JSON response

        :param: user - mightyMooC models object
        :param: course - mightyMooC models object
    '''
    module_data = []
    for module in course.modules:
        module_data.append(
            {'id': module.id,
            'module': module.name,
            'description': module.description,
            'tags': [tag.name for tag in module.tags]}
        )
    response =  {
            'user_name': user.username,
            'course': course.name,
            'id': course.id,
            'modules': module_data,
    }
    return response
```

Appendix E: Module service

```
import copy
from flask import current_app
from mightyMooc import app, db
from mightyMooc.backend.base_service import BaseService
from mightyMooc.backend.user_service import UserService

user_service = UserService()

class ModuleService(BaseService):
    def __init__(self):
        self.model = 'Module'
        self.db_module = self.dyanmic_module()
        super(ModuleService)

    def enrol(self, user_id, module_ids):
        ''' enrol a user to a module
```

```
        :param: module_ids - list of ids
    '''
    user = user_service.get_by_id_raw(user_id)
    modules = set()
    for m_id in set(module_ids):
        modules.add(self.get_by_id_raw(m_id))
    user.modules.extend(modules)
    module_names = [module.name for module in user.modules]
    try:
        db.session.commit()
        return {'username': user.username, 'modules': module_names}
    except:
        db.session.rollback()

def to_results(self, remove_keys=None):
    '''
    Iterate over the list of raw results.
    Remove unwanted keys from the results.
    Append the cleaned results to a list of dicts.
    :param: remove_keys - custom list of additional keys remove
    '''
    REMOVE_KEYS = ['_sa_instance_state', 'deleted_at',
    'created_at', 'updated_up']
    if remove_keys:
        REMOVE_KEYS += remove_keys
    results = []
    for result_set in self.results:
        #  Make a copy as we are iterating the data we are mutating
        result_dict = copy.copy(vars(result_set))
        for remove_key in REMOVE_KEYS:
            del result_dict[remove_key]
        result_dict['institutions'] = [i.name for i in
                                result_set.institutions.all()]
        result_dict['courses'] = [c.name for c in
                            result_set.courses.all()]
        result_dict['tags'] = [t.name for t in
                           result_set.tags.all()]
        results.append(result_dict)
    return results
```

Appendix F: Institution service

```
from mightyMooc import app, db
from mightyMooc.models import Institution
from mightyMooc.backend.base_service import BaseService
from mightyMooc.backend.course_service import CourseService

course_service = CourseService()

class InstitutionService(BaseService):
    def __init__(self):
        self.model = 'Institution'
        self.db_module = self.dyanmic_module()
        super(InstitutionService)

    def build_institution_json(self, name):
        ''' Build a dictionary for all courses that an institution provides
```

```
        content for
    '''
    institutions = Institution.query.filter(
        Institution.name.ilike(name)).all()
    for institution in institutions:
        response = []
        for course in institution.courses.all():
            response.append(self.json_institution_course_modules(
                course, institution))
    return response


  def json_institution_course_modules(self, course, institution):
    ''' Find all modules and courses offered to by a given institution and
        build a JSON response

        :param: institution - mightyMooC models object
        :param: course - mightyMooC models object
    '''
    module_data = []
    for module in course.modules:
        if self.check_publisher(module, institution.name):
            module_data.append(
                {'id': module.id,
                'module': module.name,
                'description': module.description,
                'tags': [tag.name for tag in module.tags]}
            )
    response =  {
            'course': course.name,
            'id': course.id,
            'modules': module_data,
    }
    return response

  def check_publisher(self, content, institution):
    '''
        Check the publisher of a given module or course
    '''
    institutions = set()
    for content_owner in content.institutions.all():
        institutions.add(content_owner.name)
    if institution in institutions:
        return True
    return False
```

Appendix G: User service

```
from mightyMooc import app, db
from mightyMooc.backend.base_service import BaseService

class UserService(BaseService):
  def __init__(self):
    self.model = 'User'
    self.db_module = self.dyanmic_module()
    super(UserService)
```

```
    def get_by_id(self, id):
        remove_keys = ['password_hash', 'email', 'last_sign_in', 'user_type']
        user = self.db_module.query.filter_by(id=id).first()
        self.results = [user]
        response = self.to_results(remove_keys=remove_keys)
        response[0]['courses'] = [course.name for course in user.courses]
        response[0]['modules'] = [module.name for module in user.modules]
        return {"status": "ok", "data": response}
```

Appendix H: Tag service

```
from flask_restful_swagger import swagger

from mightyMooc import app, db
from mightyMooc.backend.base_service import BaseService
from mightyMooc.models import Tag, Course, Module
from mightyMooc.backend.institution_service import InstitutionService

class TagService(BaseService):
    def __init__(self):
        self.model = 'Tag'
        self.db_module = self.dyanmic_module()
        super(TagService)


    def get_modules_by_tag(self, tag):
        tags = Tag.query.filter(Tag.name.ilike(tag)).all()
        response = []
        for tag in tags:
            response.append({'tag': tag.name,
                'modules': [m.name for m in tag.modules.all()]})
        return response
```

Appendix I: Seed file

```
from mightyMooc import app, db
from mightyMooc.backend.user_service import UserService
from mightyMooc.backend.institution_service import InstitutionService
from mightyMooc.backend.module_service import ModuleService
from mightyMooc.backend.course_service import CourseService
from mightyMooc.backend.tag_service import TagService


class RunSeed():
    def __init__(self):
        course_service = CourseService()
        module_service = ModuleService()
        tag_service = TagService()
        users =[{'username': 'John', 'email': 'john@example.com'},
        {'username': 'Clare', 'email': 'clare@example.com'},
        {'username': 'Mark', 'email': 'mark@example.com' },
        {'username': 'Nasser', 'email': 'nasser@example.com'},
        {'username': 'Fiona', 'email': 'fiona@example.com'}]
        self.generate(users, UserService, 'User')
```

```
        institutions =[{'name': 'Oxford'},
        {'name': 'Cambridge'},
        {'name': 'Imperial',},
        {'name': 'Cardiff'},
        {'name': 'Bristol'}]
        self.generate(institutions, InstitutionService, 'Institution')

        courses =[{'name': 'Programming Fundamentals', 'description': 'An introduction to programming.'},
        {'name': 'Programming for Data Science', 'description': 'Covers some of the mathmatical and programmatic
principles required for by modern data science.'}]
        self.generate(courses, CourseService, 'Course')


        p_course_id = CourseService().get_raw(**{'name':'Programming Fundamentals'})[0].id
        ds_course_id = CourseService().get_raw(**{'name':'Programming for Data Science'})[0].id

        prog_modules =[{'name': 'OOP'},
        {'name': 'UI/UX Design'},
        {'name': 'Databases'},
        {'name': 'Understanding Data structures'},
        {'name': 'Memory Optimization'},
        {'name': 'Testing and debugging'}]
        prog_ids = [i for i in range(1,7)]


        data_science_modules =[
        {'name': 'Statistical Modelling'},
        {'name': 'Probabilistic Graphical Models'},
        {'name': 'Advanced data analytics with Python'},
        {'name': 'Natural Language Processing'},
        {'name': 'Timeseries Analysis with Spark'},
        {'name': 'Machine Learning'}]
        ds_ids = [i for i in range(8,13)] + [3,4] # Databases and data structures

        modules = prog_modules + data_science_modules
        self.generate(modules, ModuleService, 'Module')


#################### M-2-M course_modules #######################################
        prog_course = course_service.get_raw(**{'name':'Programming Fundamentals'})[0]
        for module_id in prog_ids:
            module = module_service.get_by_id_raw(module_id)
            prog_course.modules.append(module)

        ds_course = course_service.get_raw(**{'name':'Programming for Data Science'})[0]
        for module_id in ds_ids:
            module = module_service.get_by_id_raw(module_id)
            ds_course.modules.append(module)

#################### M-2-M course_institutions ###################################

        oxford = InstitutionService().get_raw(**{'name':'Oxford'})[0]
        cambridge = InstitutionService().get_raw(**{'name':'Cambridge'})[0]
        imperial = InstitutionService().get_raw(**{'name':'Imperial'})[0]
        cardiff = InstitutionService().get_raw(**{'name':'Cardiff'})[0]
        bristol = InstitutionService().get_raw(**{'name':'Bristol'})[0]
```

```
    prog_institutions = [oxford, imperial, cardiff]
    ds_institutions = [oxford, cambridge, bristol]

    for institution in prog_institutions:
        institution.courses.append(prog_course)

    for institution in ds_institutions:
        institution.courses.append(ds_course)



################### M-2-M module_tags #######################################

    # Tags
    programming = tag_service.create(**{'name': 'Programming'})
    python = tag_service.create(**{'name': 'python'})
    sql = tag_service.create(**{'name': 'SQL'})
    rdbms =  tag_service.create(**{'name': 'Relational Database Management Systems'})
    frontend = tag_service.create(**{'name': 'frontend'})
    nodejs = tag_service.create(**{'name': 'node.js'})
    js = tag_service.create(**{'name': 'javaScript'})
    react = tag_service.create(**{'name': 'react'})
    stats = tag_service.create(**{'name': 'Statistics'})
    prob = tag_service.create(**{'name': 'Probabality'})
    analytics = tag_service.create(**{'name': 'Data analytics'})
    nlp = tag_service.create(**{'name': 'NLP'})
    spark = tag_service.create(**{'name': 'Spark'})
    big_data = tag_service.create(**{'name': 'Big data technologies'})

    # Modules
    oop = module_service.get_raw(**{'name':'OOP'})[0]
    ui = module_service.get_raw(**{'name': 'UI/UX Design'})[0]
    dbs = module_service.get_raw(**{'name': 'Databases'})[0]
    data_struct = module_service.get_raw(**{
        'name': 'Understanding Data structures'})[0]
    memory = module_service.get_raw(**{'name': 'Memory Optimization'})[0]
    testing = module_service.get_raw(**{'name': 'Testing and debugging'})[0]

    stats_mod = module_service.get_raw(**{'name': 'Statistical Modelling'})[0]
    prob_mod = module_service.get_raw(**{'name': 'Probabilistic Graphical Models'})[0]
    analytics_mod = module_service.get_raw(
        **{'name': 'Advanced data analytics with Python'})[0]
    nlp_mod = module_service.get_raw(**{'name': 'Natural Language Processing'})[0]
    timeseries = module_service.get_raw(
        **{'name': 'Timeseries Analysis with Spark'})[0]
    ml = module_service.get_raw(**{'name': 'Machine Learning'})[0]

    # Build M-2-M tags
    oop.tags.append(programming)
    ui.tags.append(js)
    ui.tags.append(nodejs)
    ui.tags.append(react)
    ui.tags.append(frontend)
    dbs.tags.append(sql)
    dbs.tags.append(rdbms)
    memory.tags.append(programming)
    testing.tags.append(programming)
```

```
        data_struct.tags.append(programming)
        stats_mod.tags.append(stats)
        prob_mod.tags.append(prob)
        analytics_mod.tags.append(python)
        analytics_mod.tags.append(programming)
        analytics_mod.tags.append(analytics)
        nlp_mod.tags.append(nlp)
        nlp_mod.tags.append(python)
        timeseries.tags.append(spark)
        timeseries.tags.append(big_data)
        ml.tags.append(big_data)
        ml.tags.append(python)


################### M-2-M module_institutions ################################

        oop.institutions.append(oxford)
        oop.institutions.append(imperial)
        dbs.institutions.append(oxford)
        ui.institutions.append(cardiff)
        memory.institutions.append(imperial)
        testing.institutions.append(oxford)
        data_struct.institutions.append(oxford)
        data_struct.institutions.append(cambridge)
        stats_mod.institutions.append(bristol)
        prob_mod.institutions.append(cambridge)
        analytics_mod.institutions.append(bristol)
        nlp_mod.institutions.append(cambridge)
        timeseries.institutions.append(oxford)
        ml.institutions.append(oxford)


        db.session.commit()


    @staticmethod
    def generate(data, import_object, model):
        create = 'create'
        for row in data:
            getattr(import_object(), create)(**row)


if __name__ == "__main__":
    print('Running Seed')
    RunSeed()
    print('Done')
```

Appendix J: Models.py

```
from mightyMooc import db, login
from datetime import datetime
from hashlib import md5


class Models():
```

```
    def __init__(self):
        self.taxonomy = set()

    def build_taxonomy(self, tablename):
        self.taxonomy.add(tablename)


################# MANY TO MANY RELATIONSHIP OBJECTS ######################
institution_users=db.Table('institution_users',
    db.Column('institution_id', db.Integer, db.ForeignKey('institution.id')),
    db.Column('user_id', db.Integer, db.ForeignKey('user.id'))
        )

module_institutions=db.Table('module_institutions',
    db.Column('module_id', db.Integer, db.ForeignKey('module.id')),
    db.Column('institution_id', db.Integer, db.ForeignKey('institution.id'))
        )

module_tags=db.Table('module_tags',
    db.Column('module_id', db.Integer, db.ForeignKey('module.id')),
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'))
        )

course_modules=db.Table('course_modules',
    db.Column('course_id', db.Integer, db.ForeignKey('course.id')),
    db.Column('module_id', db.Integer, db.ForeignKey('module.id'))
        )

course_institutions=db.Table('course_institutions',
    db.Column('course_id', db.Integer, db.ForeignKey('course.id')),
    db.Column('institution_id', db.Integer, db.ForeignKey('institution.id'))
        )

user_modules=db.Table('user_modules',
    db.Column('user_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('module_id', db.Integer, db.ForeignKey('module.id'))
        )

user_courses=db.Table('user_courses',
    db.Column('user_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('course_id', db.Integer, db.ForeignKey('course.id'))
        )

#########################################################################

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    firstname = db.Column(db.String(64))
    lastname = db.Column(db.String(64))
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    created_at = db.Column(db.DateTime)
    updated_up = db.Column(db.DateTime)
    deleted_at = db.Column(db.DateTime)
    last_sign_in = db.Column(db.DateTime)
    user_type = db.Column(db.String(32))
```

```
    modules = db.relationship('Module', secondary=user_modules,
        # primaryjoin=(user_modules.c.user_id == id),
        # secondaryjoin=(user_modules.c.module_id == id),
        backref=db.backref('users', lazy='dynamic'), lazy='dynamic')
    courses = db.relationship('Course', secondary=user_courses,
backref=db.backref('users', lazy='dynamic'), lazy='dynamic')

    # modules = many_to_many_relationship(user_modules, 'users', 'modules')

    def __repr__(self):
        return '<User {}>'.format(self.username)


class Institution(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name =  db.Column(db.String(64), index=True, unique=True)
    created_at = db.Column(db.DateTime)
    updated_up = db.Column(db.DateTime)
    deleted_at = db.Column(db.DateTime)
    users = db.relationship('User', secondary=institution_users,
        # primaryjoin=(institution_users.c.institution_id == id),
        # secondaryjoin=(institution_users.c.user_id == id),
        backref=db.backref('institutions', lazy='dynamic'), lazy='dynamic')


class Module(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name =  db.Column(db.String(64), index=True)
    created_at = db.Column(db.DateTime)
    updated_up = db.Column(db.DateTime)
    deleted_at = db.Column(db.DateTime)
    description = db.Column(db.Text)
    tags = db.relationship('Tag', secondary=module_tags,
        # primaryjoin=(module_categories.c.module_id == id),
        # secondaryjoin=(module_categories.c.category_id == id),
        backref=db.backref('modules', lazy='dynamic'), lazy='dynamic')

    institutions = db.relationship('Institution', secondary=module_institutions,
        # primaryjoin=(module_institutions.c.module_id == id),
        # secondaryjoin=(module_institutions.c.institution_id == id),
        backref=db.backref('modules', lazy='dynamic'), lazy='dynamic')


class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name =  db.Column(db.String(64), index=True, unique=True)
    created_at = db.Column(db.DateTime)
    updated_up = db.Column(db.DateTime)
    deleted_at = db.Column(db.DateTime)


class Course(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name =  db.Column(db.String(64), index=True, unique=True)
    description = db.Column(db.Text)
    level = db.Column(db.String(64))
    created_at = db.Column(db.DateTime)
```

```
    updated_up = db.Column(db.DateTime)
    deleted_at = db.Column(db.DateTime)
    institutions = db.relationship('Institution', secondary=course_institutions,
        # primaryjoin=(course_institutions.c.course_id == id),
        # secondaryjoin=(course_institutions.c.institution_id == id),
        backref=db.backref('courses', lazy='dynamic'), lazy='dynamic')

    modules = db.relationship('Module', secondary=course_modules,
        # Commented out as 'sqlalchemy.orm.exc.UnmappedColumnError' on insert
        # primaryjoin=(course_modules.c.course_id == id),
        # secondaryjoin=(course_modules.c.module_id == id),
        backref=db.backref('courses', lazy='dynamic'), lazy='dynamic')
```

Appendix K: Routes.py

```
from flask import request, current_app, jsonify

from webargs import fields
from webargs.flaskparser import use_args

from mightyMooc import app, db
from mightyMooc.backend.catalogue_service import CatalogueService
from mightyMooc.backend.module_service import ModuleService
from mightyMooc.backend.course_service import CourseService
from mightyMooc.backend.institution_service import InstitutionService
from mightyMooc.backend.user_service import UserService

catalogue_service = CatalogueService()
course_service = CourseService()
module_service= ModuleService()
institution_service= InstitutionService()
user_service= UserService()

SERVICE_ROUTER = {
    'module': module_service,
    'course': course_service
}

@app.route('/catalogue', methods=['GET'])
def catalogue():
    kwargs = request.args.to_dict()
    return jsonify(catalogue_service.get(**kwargs))


@app.route('/catalogue/tags/<string:tag>', methods=['GET'])
def get_by_tag(tag):
    return jsonify(catalogue_service.get_tags(tag))

@app.route('/catalogue/institutions/<string:institution>', methods=['GET'])
def get_by_institution(institution):
    return jsonify(catalogue_service.get_institutions(institution))


@app.route('/catalogue/<string:type>/<int:id>', methods=['GET'])
def get_by_id(type, id):
    return jsonify(catalogue_service.get_by_id(**{'type': type, 'id': id}))
```

```
@app.route('/catalogue', methods=['POST', 'GET', 'DELETE'])
def add_content():
    ''' Used to add a new module or course to the catalogue
    '''
    request_data = request.get_json()
    service = SERVICE_ROUTER.get(request_data.get('type'))
    if request.method == 'POST':
        institutions = [institution_service.get_by_id_raw(request_data[
            'institution_id'])]
        del(request_data['type'])
        del(request_data['institution_id'])
        result = service.create(**request_data)
        service.add_many_to_many(result, institutions, 'institutions')
        return jsonify({'message': 'data added successfully',
                        'data': request_data,
                        'status': 200})
    elif request.method == 'DELETE':
        return jsonify(service.soft_delete(request_data['type'],
                           request_data['id'],
                           request_data['institution_id'],
                           'institutions'))


@app.route('/catalogue/enrol', methods=['POST', 'DELETE'])
def enrollment():
    '''  Used to enrol a user on a module or course
        :param: content_id - maps to a module or course depending on type param
        :param: type - string - 'module' or 'course'
    '''
    request_data = request.get_json()
    service = SERVICE_ROUTER.get(request_data.get('type'))
    if request.method == 'POST':
        response = service.enrol(request_data['user_id'],
            request_data['content_ids'])
        return jsonify({'message': 'data added successfully', 'data': response,
         'status': 200})

    elif request.method == 'DELETE':
        for content_id in request_data['content_ids']:
            delete_response = service.soft_delete(request_data['type'],
                           content_id,
                           request_data['user_id'],
                           'student')
            delete_response['message'] = 'You have successfully cancelled your enrollment on the {} {}'.\
            format(delete_response['to_delete'],
                request_data['type'])
        return jsonify(delete_response)



@app.route('/catalogue/students/<int:id>', methods=['GET'])
def student(id):
    ''' Fetch a users enrollments
    '''
    return jsonify(user_service.get_by_id(id))
```

Appendix L: __init__.py

```
from flask import Flask, current_app
from config import Config
from flask_migrate import Migrate
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object(Config)

with app.app_context():
        print(current_app.name)
        db = SQLAlchemy(app)
        migrate = Migrate(app, db)
        login = LoginManager(app)
        login.login_view = 'login'

        from mightyMooc import routes, models
```