**Service Oriented Architecture**

Joshua Harrison

838252

February 2018

**Discussion Points**

**Access control**

When a logged in user requests a resource, should a database call be made to validate the user has access to that resource (e.g. they’ve enrolled?)

OR on login should a

**Introduction**

For this project a MooC service in which universities can provide course material as both standalone units and as part of a wider qualification has been designed. A service orientated architecture(SOA) has been used to design the *‘MightyMooC’* platform. Erl(2010) defined eight core Service Oriented Design Principles as:

* ***Standard Service Contract****: Services must adhere to a standardized shared contract in order to promote standardization of design and ease of adoption.*
* ***Service Loose Coupling –*** *The edges between services should be minimally weighted, with no one service too heavily reliant or any other in order to perform its function.*
* ***Service Abstraction –*** *Service should be coarse grained with details of the internal logic of the abstracted from external consumers.*
* ***Service Reusability –*** *Services should be significantly agnostic so that they are candidates from multiple disparate business processes.*
* ***Service Autonomy –*** *Services should have significant control over their runtime environments and resources required for perform their function. This is in line with containerization and if taken to its logic limit, a microservices architecture.*
* ***Service Statelessness –*** *State information should not be handled by the services runtime in order to enhance resource scalability.*
* ***Service Discoverability –*** *Services should be discoverable through defined discovery mechanisms in order to maximize service reuse and thus avoid rewriting existing service logic*
* ***Service Composability –*** *Individual services can be grouped into ‘compositions’ with other services in a service inventory to solve high level business problems.*

Erl(2010) reasons that interoperability is the fundamental concept of Service Orientated Architecture as achieving interoperability leads to demonstrable return on investment (RoI) through the continual application and reuse of services in multiple business processes, both internal and external to the service owners. The eight principles highlighted above serve to drive the overall architecture towards interoperability. The MightMooC project has been design with these principles in mind, and their application shall be discussed in greater detail in the follow sections.

**Section A – Internal architecture**

This section shall focus on the design and implementation of the internal architecture and supporting business logic defined for MightMooC.

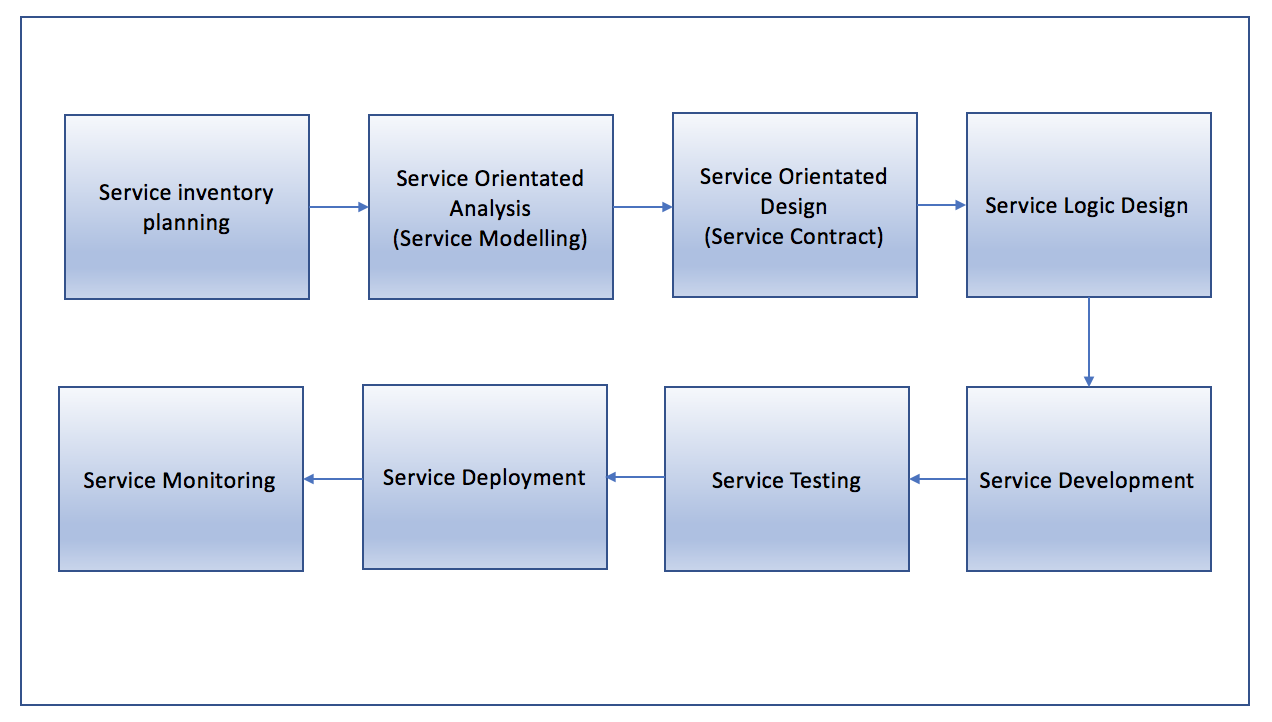
**A.1. Design Approach**

The project has been designed using a *'top down’* modelling implementation, meaning that prior to code creation and schema definition, a prerequisite step is to analyse and deconstruct the central business requirements to the problem. This approach is considered best practice for SOA projects as it enables a global definition of services and their capabilities allowing logical boundaries to be defined, leading to increased ubiquity, reduced code overlap between services, and crucially, increased interoperability and composability of services.

There are caveats that can limit the success of implementing a top down design that should be considered. Namely, as a project progresses it naturally evolves, unexpected challenges and edge cases can arise; subsequently new functionality and consumers can be added. The planning and documentation generated from the top down design need 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.

A secondary concern is speed of development iterations which can suffer as a result of top down design. 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, 2010, 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, therefore against the core principles of service orientation.

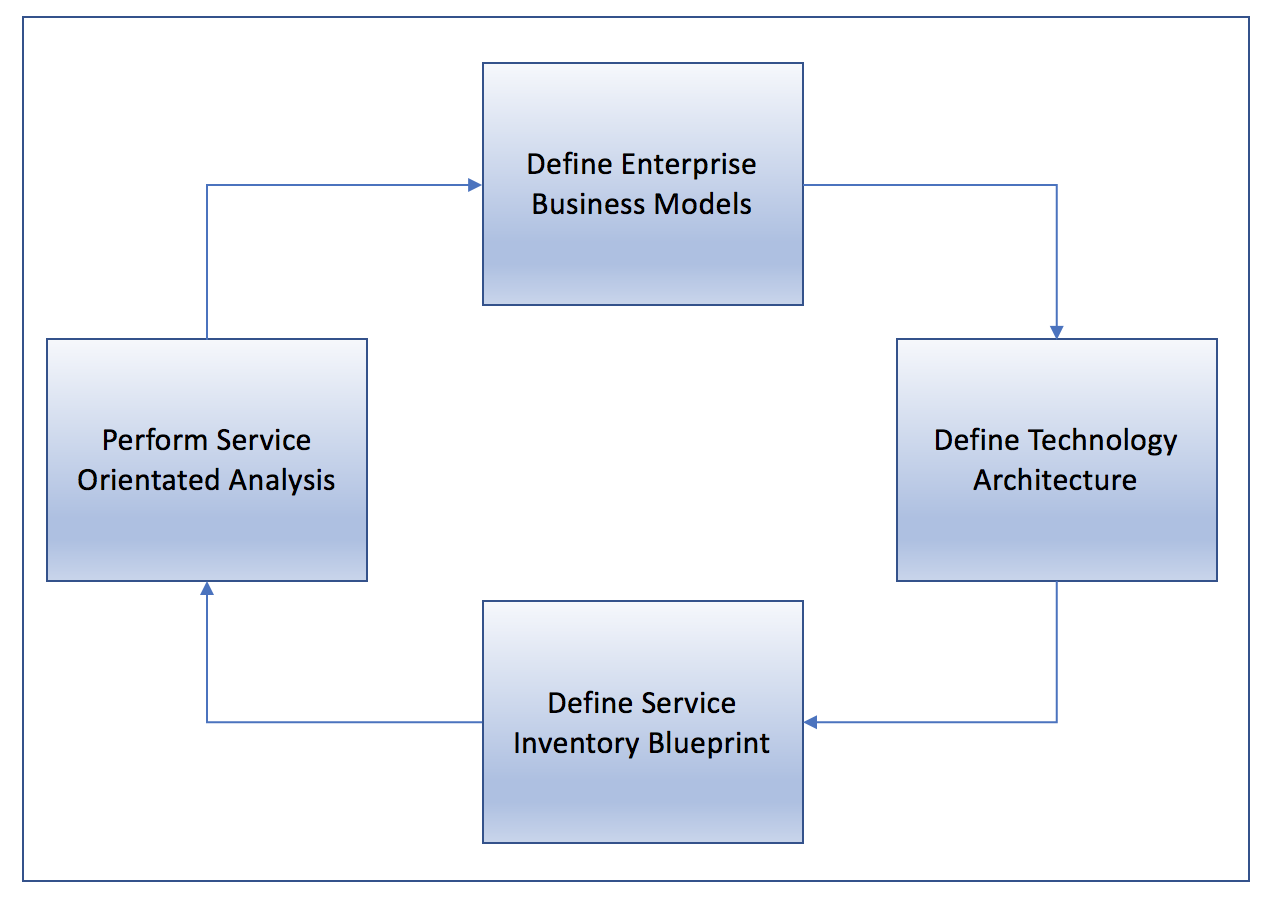
Figure 1 depicts the top down design approach that that is utilised by this project. This design has been adapted from Erl’s original specification ‘SOA project and lifecycle phases’ (2010, pgs 92-103). Phases related to the wider business context of SOA, such as organizational buy-in/adoption, service discoverability and versioning have been omitted.



*Figure 1: Project design phases*

**A.1.0 Service Inventory Planning (SIP)**

Erl (2005, pg. 96-97) defines SIP as a four stage cycle which aims to produce a *‘service inventory blueprint’* from which services will be built. The blueprint is derived from following the four step cycle depicted in Figure 2. The application of this cycle in this project is detailed below.



*Figure 2: Service Inventory Analysis Cycle, Source: Erl. 2005, P.g. 96.*

**A.1.1 Define Enterprise Business Models**

This phase typically involves analysing existing and required business process and entity models to conceptualised fundamental requirements of the service inventory. The following core business process definitions were asserted as the goals of *this* business process:

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

In addition to these corerequirements, there are also several ancillary requirements for a fully functional system, such creating an account, logging in, signing up to a service and making payments. A deeper analysis of these ancillary tasks has been omitted from this phase of the report as they are fundamental to any given web application and not unique to *this* project.

Upon decomposing the core units; a key design choice arises. Should a centralised database of content be used for content providers to upload their material, or alternatively, should institutions store all content locally allowing the MightMooC service to connect to the content providers and *lazily* access the content as and when requested from an end-user client?

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

It could also mean a lower barrier to entry for the content providers, as rather than having to upload content in a standardized manner suitable for the schema defined by MightyMooC, course providers could simply build a client service which connected to their own content stores to access material and serve it back up to MightMooC on request. This is in line with SOA principles of interoperability, as given the client was well built it could potentially act as an interface for any content the institution wanted to offer. It is worth noting that depending on the IT capabilities of the institution in question, this could potentially lower or raise the barrier to entry, as the alternative would be a manual task of uploading material in an approved data format. IS THIS RELEVANT???

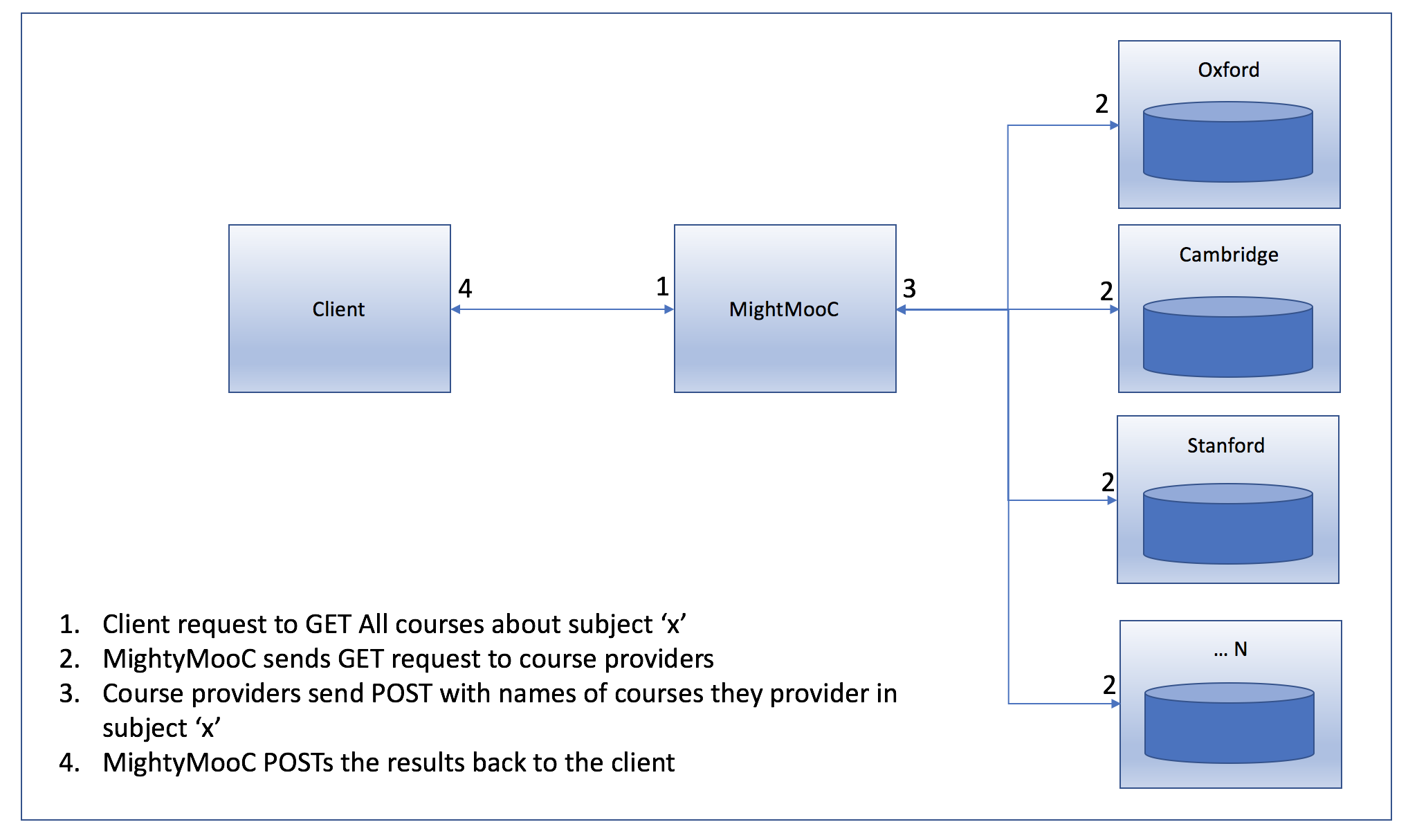
There is an interesting consideration in the decentralised content infrastructure regarding the relationship between course popularity and bandwidth constraints on the course providers. If content was POSTed to the MightyMooC service lazily upon request, *the bandwidth consumed by the course provider 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.

One potential solution would be to the use of internal caching servers such as *Redis* to temporally store the content following a GET request from an external client. This way, MightyMooC could make a single GET request to the content provider and cache the response. Any clients subsequently requesting the same content would have it served from the cache server. 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, allow for regional caching and serving of data to match the location of inbound client GET requests.

There are however several issues with this design. A large problem is that is breaks the principle of *service loose coupling*. In the decentralised storage model there is tight coupling between the course provider and the content being served. If for any given reason a university suffered an availability fault in its own infrastructure, any 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, the fact that the data storage would be handled at least mean that this single point of failure is controlled by MightyMooC, meaning that optimisations and provisions for redundancy can be made 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 have to be sent across the wire whenever MightyMooC made a request for content. With a centralised data platform, MightyMooC would simply have to retrieve the requested data internally and then serve it to the client. This also brings security benefits as there is data is less frequently in transit, meaning less opportunity for interception attacks.

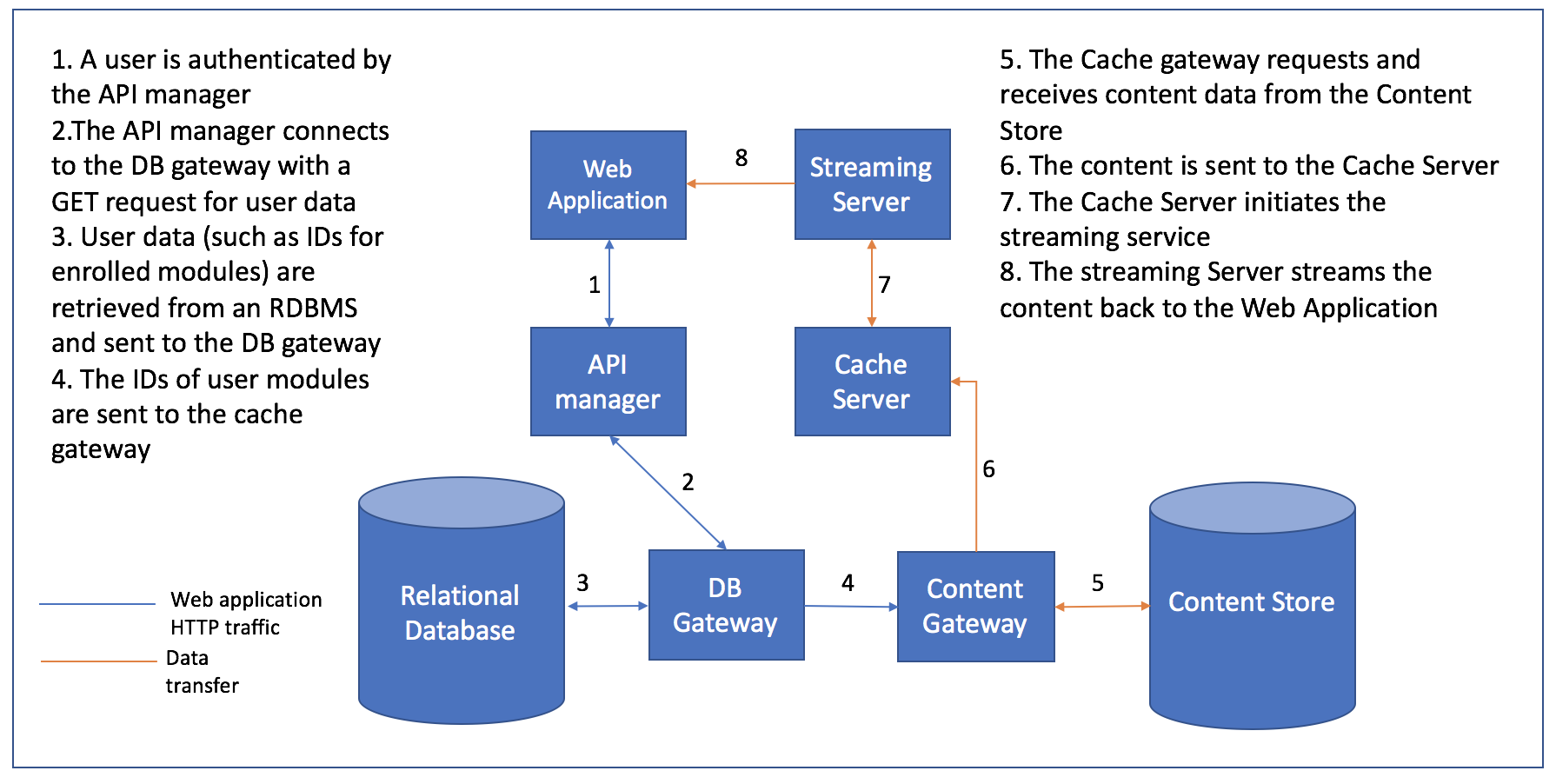
A final more pragmatic advantage of storing content centrally is that it mitigates any chance of content providers drastically changing content out of the control of MightyMooC. Whilst content providers should be able to control and change their content if required, if given complete autonomy over this process then the ephemeral nature of the content could lead to problems for end users such as different users enrolling at the same time access different content, or even different assessments for a given module. This also means that MightyMooC can better specify and document the expected format of content. Which will allow for external APIs to also pull the content.



*Figure 3: An example of a system whereby content storage is handled locally by the course providers and lazily served upon request. Here we see the process by which an end user would search for courses in a given subject.*

**A.1.2 Define Technology Architecture**

Given the above discussion it was considered that a centralised content store to which content providers would upload material was the best architecture. The discussion around caching and data locally are still considered relevant in this architecture and so these have been incorporated into the technology architecture design.

**

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

*NOTE TO SELF: check the API manager is in the correct position (as opposed to position 1)*

**Key Technology**

**Web Application**

A frontend connection to a headless API using the REST framework. REST has been chosen as it is better and more popular than SOAP.

**API Manager**

Handle authentication process and ferrying HTTP requests through the API.

**Database Gateway**

A gateway for any in/outbound connections to the Relational Database.

**Relational Database**

A traditional RDBMS tasked with storing non content specific data. For example tables such as users, modules, and users\_modules would be stored here as well as metadata from generated from application activity. Given the nature of the data, there are no immediate scalability considered, and so a standard RBMS such as MySQL or Postgres would suffice.

**Content Gateway**

Analogous to the Database Gateway – provides a gateway into the Content Store.

**Content Store**

The Content Store is a data repository where all content provided by external providers would reside. This is liable to have scalability considerations as the number of providers participating in the system grows. Given that providers will upload a range of materials such as video media files, PDF, and images one solution would be that a simple scalable store system like Amazon S3 buckets would be a suitable solution. An alternative approach would be to use a NOSQL database as a document such as MongoDB which uses the BSON data format to store large binary objects such as videos. This could have speed and analytics benefits in the long run.

**Cache Server**

A server to Cache media content from the Content Store upon request to enhance performance of streaming and reduce bandwidth usage.

**Streaming Server**

A server which connects to the Cache server and streams content back to the web application for the logged in user to access.