## Group member details: names and SIDs

Our group will consist of three members. Bradley Thurlow (44779496), James Ridley (44805632) and Bryce Altman (44914792).

## System Architecture

The core concept of microservices is to break down every tier of our application into smaller more manageable sub components. These subcomponents will encapsulate its own service containing business logic and other adaptors which will expose API's that can be consumed by other microservices or by application clients. At one level of abstraction this design will appear to be a single one system, but in fact, it encapsulates many systems and services integrated together, to appear as a single cohesive entity. It is important to note in a microservice type architecture that each service is responsible for its own component and that is it, to build highly successful microservices this design must be factored into the design so each app doesn't have access to backend servers, instead communication is mediated by an intermediary API. The API gateway will handle all the requests and tasks such as load balancing, caching, access control and API monitoring. This form of modularity is one of the primary strengths of the system, as each service will have its primary function. As long as this service has its functional API contract it can be called and take form in the overall system.

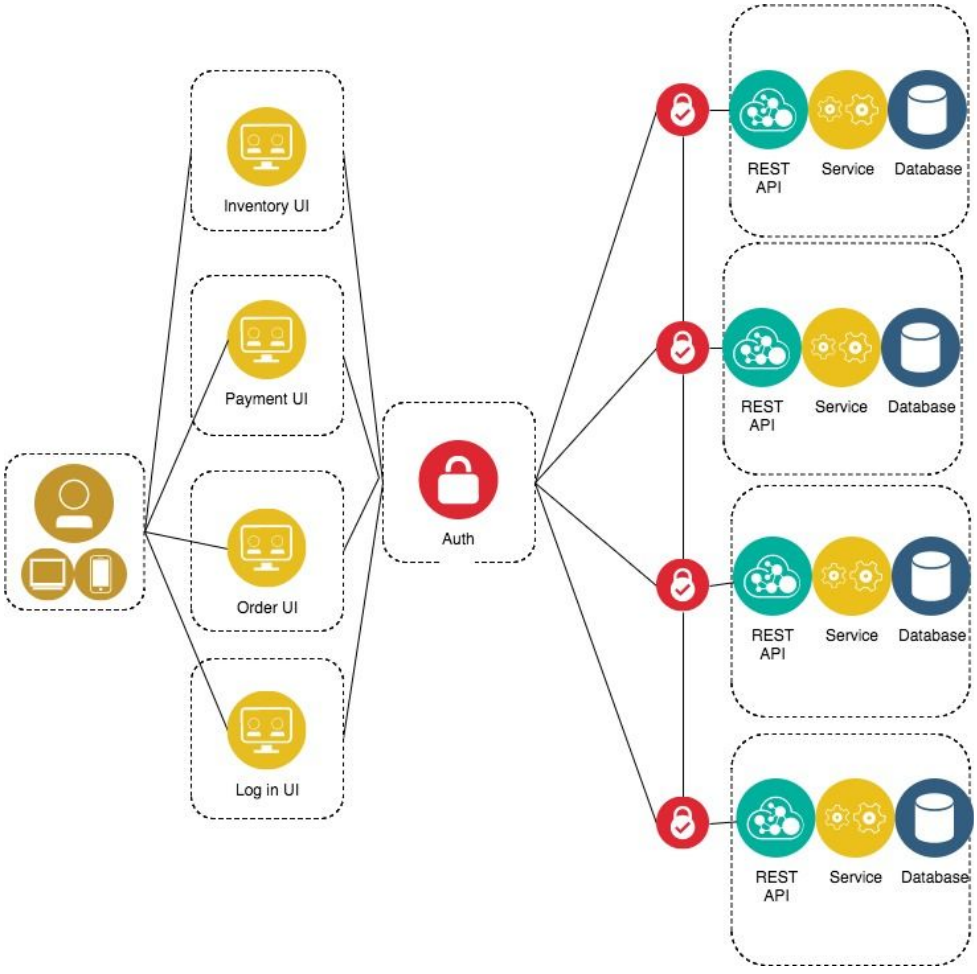
Each backend service must expose a ReST API and the other services will consume API's provided by these other services. Other services will invoke others in order for the total system to operate seamlessly. These services will communicate with each other via various communication models especially inter-service communication messaging services, furthermore allowing for the decoupling the client from the service. The services will all communicate via HTTP requests over ReST. Since these services will all utilise their own component such as Angular and Node.js front end, Python and flask framework backend and a database in MariaDB it will be imperative that these services all communicate via there respective API thus meaning that each component will need to have zero knowledge of how each other component works in order to consume it.

Since the system design encompases a wide variety of different services we gain a very significant bonus over traditional monolithic architectures and that is scalability. Traditionally if there was demand for a certain application we would need to scale the whole service thus creating a tremendous amount of work and carrying a significant cost. Since our system embraces the microservice architecture we will be able to run each service independently thus allowing us to scale each service which needs to be scaled and only this service. This is extremely cost efficient as it also allows is to scale our services down when they are not experience these times of load. We can do this by incorporating technologies such as docker which can deploy many instances of a certain service on an Amazon server, coupled with its REST API and correlation database.
The microservice architecture pattern contains another strength which ensures loose coupling and that is databases. Microservices about the traditional mechanism of other services which offer a single database schema. Each service in our system will have its own database schema which corresponds to it. This level of functionality is also paramount to the success of the microservice architecture because it allows us to adopt a polyglot persistence architecture, that

is, to allow us to choose a database service best suited for that service. This feature will also be important in reducing latency and increasing system health as it will minimise cross-domain calls.

Continuous delivery and deployment will need to be factored into the overall system architecture due to the nature of fast and agile functionality of microservices. A pipeline will need to be established in order for new content and services to be deployed quickly. These new features will be rolled out in a sandboxed manner utilising containerization. This will embrace another strength of the architecture allowing for not only quick deployments but the ability to test and monitor independent services. Traffic to these new services should be “mimicked” before they take on actual production traffic. If they pass the automated tests then deploy them to production. This pipeline needs to be created with the very purpose of speed and deployable in a state which will not destroy other services. In order to repeat these benefits, this process should be automated.

Our system must integrate many safeguards in order for it to be available at all times. We must allow our service to prioritise impact features in order for the core components of the system to stay operational at all times. We will need to integrate aspects circuit breakers in order to stop cascading fails and latency gridlocks. Hystrix style reliance engineering incorporated by many industry leads such as Netflix, will need to be integrated in order to isolate points of access between services thus stopping cascading failures and providing fallback options, in order to improve the overall system's resilience. The service will try to avoid ACID and aim to achieve BASE thus providing options to roll back and appropriate conflict resolution method. The services key components will also need to integrate replication of the database in order to safeguard against failures and to maintain a backup in case something catastrophic occurs.



## Microservices and Components

### Login

The login will be a key component of the service. The login okays a vital part to any service regardless of industry. It allows for the client to have all their information which relates to them, thus creating a more personalised feel and user experience on the service. This information will not only store the private information about the user but also curate all their books, saved notes, payment information, saved wish lists etc. This ultimately shapes the user experience to be one of positivity but more importantly allows us as a business to fulfill the requests and services which relate to that specific user.

### Checkout and Payment

This service will be the area in which the client can make new purchases on books be it rental, subscription or outright. This component will be important in not only retrieving the information related to payments but also the item in which the customer wants to purchase. The payment process will be a secure shopping cart type layout in which the customer can pay for all their products. Components of this may be integrated with other services such as Paypal or Stripe, thus creating a more user-friendly, secure and automated process for the client and the business.

### Inventory

This service will will allow the client to browse the various offerings in which BookBarter has to offer whilst also allowing for a place to tenants to store, market and distribute their offerings directly the consumer. The inventory system will need to integrate user friendly components such as search and offer suggestions based on their previous purchase history. The service will have a large collection of all the books in which we offer, categorized in an easy and user intuitive way to locate.

### Order

The ordering system will be the service which allows for the user to fulfill their purchase. This will be where the user will have their book distributed to them and credited to there account. Since this book will be in their profile they will be able to view it from anywhere in which they have a logged into their account. The book may also be downloaded via protocols such as FTP.

## Implementation and Technologies

### Front-End:

The front end of our application will be mainly implemented using the Angular framework, specifically Angular 2. We will also be following the Model, View, Controller (MVC) design pattern, as it has many advantages which we believe will be beneficial to us as a group. For example, the MVC design pattern enables a faster development process, especially within groups, using this method we can have one member working on the Model, one working on the View, and one working on the controller. This will enable our team to work in parallel which we believe is essential for our productivity as we are dividing the work evenly between all member. Another advantage of the MVC design pattern is that code duplication is very limited, as the business logic is completely separate from the display, this will enable our application to be as

lightweight as possible, whilst still being built on a stable foundation. Finally, MVC supports asynchronous techniques, this will allow our application to load quite quickly, which we believe is very important when dealing with a distributed system and microservices in general. The Angular Framework will help deliver dynamic, rich and fast content which will mimic that of a desktop application thus allowing the client to be routed to various services via REST requests.

Node.js will make our apps extremely fast and efficient. Angular will be served with Node.js and this process will happen quickly due to the features such as auto reloading and hot swapping. Node.js is extremely efficient for serving REST APIs and allows for asynchronous communication incorporating a style in which our architecture aims for, whilst incorporating an event-based architecture appropriate for BookBarter.

## *Back-End*

Authentication - JWT Tokens will offer the authentication on our service. The tokens will allow users to authenticate themselves in order to gain access to restricted areas, which require these login credentials. Each of the microservices will also contain a JWT key which will be used as authentication for communication between services.

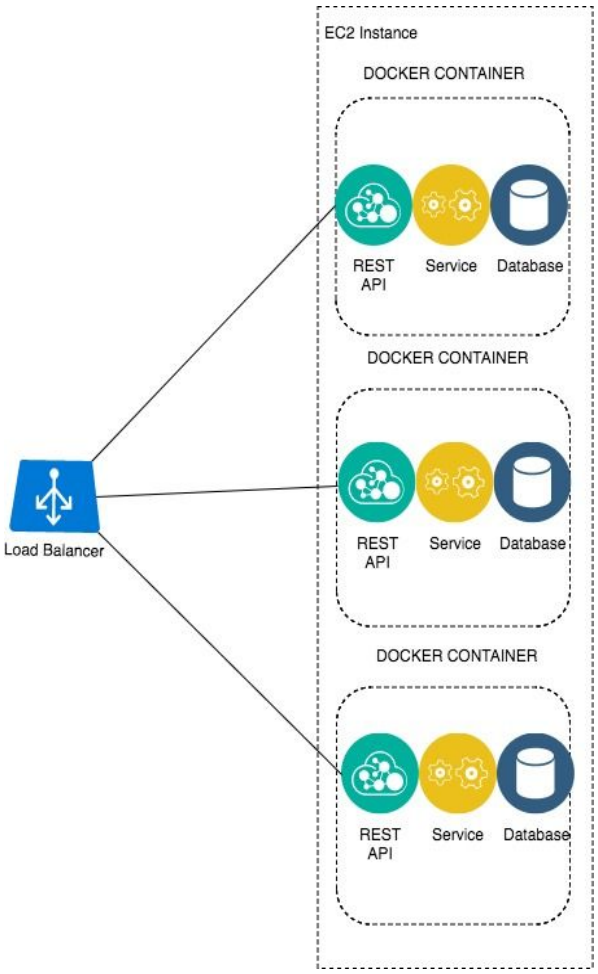
Flask framework and python will help us develop the appropriate services and make them consumable by the client and other services via their RESTful APIs. The services will all be programmed using flask as it offers many powerful tools without the need for a fully fledged implementation like Django.

Database - MariaDB. The database component of any microservice is one of its most paramount features. Bookbarter will have an independent database for each service. Since our services are so tightly connected the database will need to be updated on an event basis in order for all the records across all the services to be updated. This is because our services will be very closely related, thus meaning that a resilient messaging service will need to be incorporated. This messaging will need to occur in an event driven asynchronous via a broker in order to keep the services decoupled. This will not only ensure data correctness but decrease error states and increase load reduction. The database needs to be optimised for efficiency whilst also integrating numerous sites for replication. We must design these databases with failures in mind because ultimately the database will not only contain all the private and secure information of customers but will also contain all the book resources and files of our libraries. Since these databases are so interconnected we will need to use an event driven architecture, in which all of the other services are subscribed to the publisher when a notable event occurs. For example when updating a business entity, they all update their independent records, the services exchange events via a message broker.

Load balancer, Routing to services, DNS - Nginx. Nginx a popular web server will allow us to use appropriate load balancing, reverse proxy and HTTP cache. This service will allow us to reduce load on our services by allowing them to scale quickly if they experience a tremendous load. Caching will also be useful in reducing load and improving performance. Finally, service discovery via DNS integration will be paramount in order for each server instance to be found by the API gateway or another service.

## *Other*

Containers - Docker. Containerisation of the services will be vital to the overall service but also extremely beneficial in the development of the applications. Firstly containerisation the services will allow all dependencies, files, applications and files all work in their own instance. This will be paramount for testing, development and deployment. Since containerisation will allow us to keep all the apps independent of each other we will also be able to delete and install new features as we need them. Containerisation will not only allow instances of each service easily, rapidly and cost efficiently but will also allow each developer to work on a specific component as they will have all the tools required to without having to make sure their dependencies and other components are installed or up to date.




Edge servers and API gateway will be important for both clients and for the system health of BookBarter. This service will act as the middleman between front-end apps and the suit of microservices in our SaaS. Authentication will also play a paramount role in this stage as it will be the first communication between client and the system, thus signifying where the client can make route requests to based on the authentication state of the client. Edge services will also greatly improve the reliability of the service not only for the business but for the consumer as integration will allow for a consistent interface even if there are underlying service changes.

## *Testing*

### *Unit Testing:*

The first type of testing we will make use of in our project will be unit testing so that we can test individual code and methods before they are integrated into the project as a whole. Our tests will be written using the Jasmine framework and we will also take advantage of the helpful tools with Karma to run them. Jasmine and Karma are the most widely used and accepted methods for testing within Angular, all members of our group have also had experience using these tools in the workforce, so it is an extra advantage for us that we will already be comfortable writing unit tests in Angular. Unit testing goes hand in hand with the Agile methodology that we will be adopting for our project, this is because changing old code can be a risky process, but it is something that takes place often in Agile programming. However, with unit tests in place, we will be able to proceed with refactoring confidently, so it makes sense to adopt them into our work. Another added benefit of unit testing is that it forces you to constantly review the design of your code. This will be hugely important for us, as following the microservices architecture can be quite challenging at times, so it is easy to slip back into the monolithic structure that comes

naturally. Therefore, our unit tests will make sure that we are constantly and consistently reviewing and critiquing our design, which will ensure that we are sticking the microservices pattern. There are many further advantages of unit testing that can apply to many situations, however, the final one I will mention is documentation. Unit testing can provide documentation to a system, as if there are questions about the purpose of a single method within a system, a programmer can then refer to the unit tests to gain further understanding of the unit, by following things such as input, output, logic, what is accepted by the unit test, etc. Again, this will prove to be very useful within our group as we will have multiple people coding and building of off each others work, so as long as we strive to continue to unit test and maintain 100% test coverage, it will be a very useful tool.

### *Integration Testing:*

After successfully testing our modules individually with unit testing, we will need a way to ensure that these modules work correctly when integrated together. This is where integration testing comes in, we will make use of top-down integration testing so that the lower levels within each microservice and within our application as a whole can be tested before they are completely integrated. One large reason that we believe integration testing will be so crucial to our project is because we will have three different people all programming simultaneously if one member of the group has a different understanding of a certain business requirement, our separate modules will most likely not work together well the first time. Our integration tests will ensure that each of us stays accountable and focused towards the common goals of the project, as soon as one member becomes misaligned to any business requirements, this will be reflected in our integration tests.

### *End to End Testing:*

Our project will also make use of End to End (E2E) testing, to ensure that our entire project continues to behave the way we expect it to. As the functionally of our application begins to grow, having to manually test everything every time a new feature is added will quickly become infeasible. This is why we need to automate this testing process as much as possible so that we can see how our application reacts when it is being used exactly how a normal user would. The technologies involved in the E2E tests are Selenium, Protractor, and once again, Jasmine. Selenium is a tool that enables us to automate any browser, so we can force it to simulate a real user, however we will not be directly using Selenium, instead we will take advantage of Protractor, which is the testing framework provided by the Angular team that wraps around Selenium so that we still make use of all of its benefits. Finally, we will again use Jasmine for writing our tests, in order to keep everything consistent. We will also be following the page object model with our E2E tests, this will ensure that our tests are as readable as possible and will also enable many methods to be reusable. The page object model also allows for minimal changes when refactoring code in the future, which again is very useful with Agile programming. The end goal with our E2E tests is to have them integrated with our CI/CD pipeline so that before any code is shipped to production, we can make sure that no features have been compromised.