Alexander
TLDS

Alexander Version: 1.0

Document Version: 1.0

NOTE: The hard copy version of this document is FOR REFERENCE ONLY. The online version is the master.
It is the responsibility of the user to ensure that they have the current version.
Any outdated hard copy is invalid and must be removed from possible use.
It is also the responsibility of the user to ensure the completeness of this document prior to use.

OWNER: The owner of a Top Level Design Specification is expected to be in a role as Chief Architect and/or Lead Programmer.

	M0TL3Y **Microservice Applications and API Development** **COMS E6998 Section 7** **Donald Ferguson** **Columbia University**
Issue Date:	10/9/2015
Authors:	Nina Baculinao (nb2406), Whitney Bailey (wvb2103), Agustin Chanfreau (ac3680), Melanie Hsu (mlh2197)

"A little learning is a dangerous thing.
Drink deep, or taste not the Pierian Spring;
There shallow draughts intoxicate the brain,
and drinking largely sobers us again."
– Alexander Pope, *An Essay on Criticism*

Table of Contents

1. Document Control

1.1 Document Master Location

Filename:	Project1_TLDS_M0TL3Y_V1
Document Location:	See Courseworks, or contact nb2406@columbia.edu

1.2 Document Revision History

The table below contains the summary of changes:

Version	Date Changed	Completed By	Description of changes
1.0	10/9/2015	M0TL3Y	First TLDS draft

1.3 Approvers

The table below contains the record of approver, or delegate, signoff:

Approver Name	Approver Title	Version	Date Approved

1.4 Reviewers

The table below contains the record of reviewers:

Reviewer Name	Reviewer Title	Version	Date Reviewed

2. General Overview

The following top-level design specification provides an overview of the functions of Alexander, a student and course information management system for universities.

As stated above, Alexander is a student and course information management system. The system allows students to add and delete classes from their schedule. It also keeps track of current and past course enrollment. Alexander encompasses four microservices: Router, Students, Courses, and an Integrator.

Alexander allows users to send RESTful URL requests to the system. Through the Router, the request is sent to either Students or Courses, which will perform an action matching the request (ex. create, read, update, or delete records from the database). After performing a CRUD action, these microservices will send ChangeEvents to the Integrator microservice, which keeps a log of changes and maintains referential integrity in the application by ensuring that any data fields with associations (ex. enrolled_courses in Students or students_enrolled in Courses) are consistent with each other and the system as a whole.

2.1 Summary of Capabilities

Alexander's four microservices are:

A) *Students*: a student info management microservice, including data model changes. It includes information about the student, including the student id.
B) *Courses*: a course info management microservice, including data model changes. It includes info on courses, both past and present, using the course name attribute.
C) *Router*: a request router microservice fronting the Students info microservice.
D) *Integrator*: a basic referential integrity manager microservice that maintains referential integrity between the other microservices.

The Students and Courses microservices use a RESTful API to manipulate the contents of their databases, which maps the CRUD operations to HTTP methods:

1) Create (POST)
2) Read (GET)
3) Update (PUT)
4) Delete (DELETE)

<u>Supporter User Actions</u>

User requests may correspond to the following actions in Students:

1) Add student to university
2) Get student info (including enrolled courses)

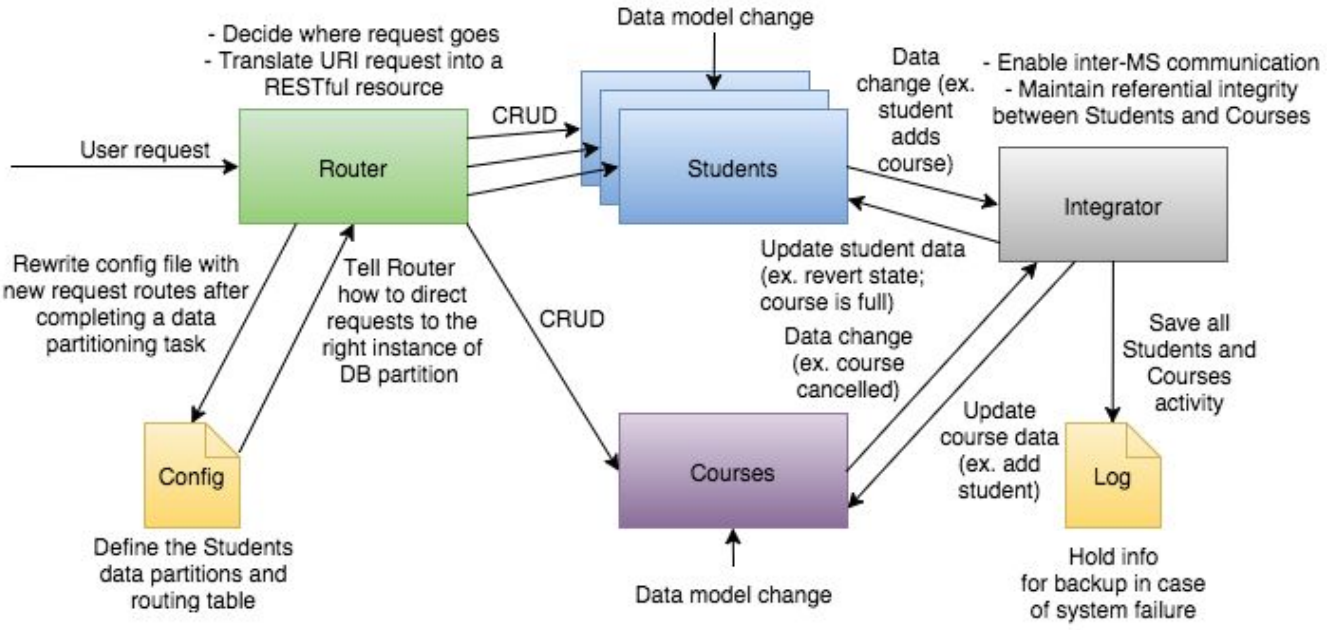
3) Update student info, including adding a class to a list of the student's enrolled classes
4) Delete student from university

the following actions in Courses:

1) Add course
2) Get course info (including students enrolled)
3) Update course info
4) Delete course

Additionally, the following two operations are supported:

1) Deleting a student from a class
2) Adding a student to a class



***Students and Courses read messages from the Integrator to update their own status. More details on the component diagram and system architecture can be found in Section 4.2 System Design.**

Referential Integrity

The Integrator helps maintain referential integrity in the Students and Courses databases, so that when major data changes occurs, such as when the student leaves the university and needs to drops all their courses, these two databases are updated to reflect these changes.

It ensures data consistency and resolves dependency issues between Students and Courses by logging

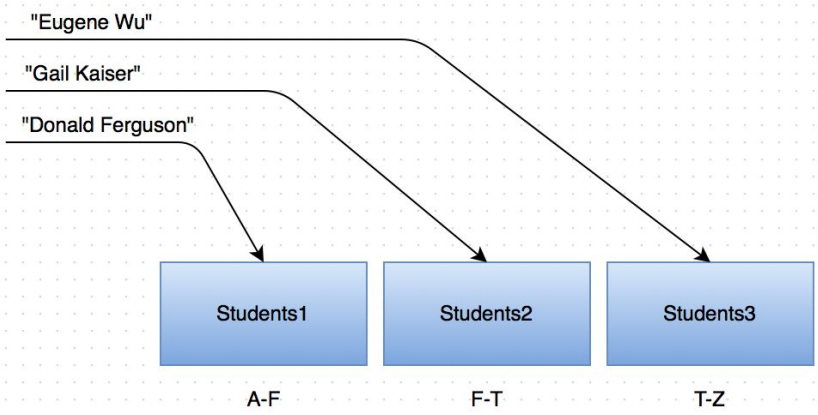
ChangeEvents from both microservices and serving as a middleman to enable inter-microservice communication. In case of a system failure, the Integrator will also maintain a master file containing all logs of ChangeEvents, so that the databases can be restored to their last working state.

Data Partitioning

Finally, the Students microservice also has the capability to scale along the z-axis of the proverbial scale cube of microservices architecture via data partitioning. The Router contains SPI functionality available only to a system administrator that handles the data partitioning of Students and rewrites the Router's config file. Thus, Students can have its data partitioned (ex. take one existing instance of Students and migrate to three instances of the Students microservice and database, which will respectively contain students' data whose last names begin with A-F, F-T, and T-Z respectively). After the partitioning, the config file must be altered to reflect these changes, and the Router will use this information to update its routing table so that the user's request is sent to the correct instance of Students.

The config file for the router ensures that the router is able to direct incoming requests to the appropriate database and/or database partitions.

Example of a partition:



2.3 User Stories

Title	Add (POST) a specific student
Description	As a user, I can type in a URL and the application adds a student to the student database.
Products Involved	Router microservice, Course microservice, Integrator microservice, Course database.

Pre-conditions	All products involved must be live.
Flow of Events	url -> router (POST) -> student microservice -> student database |-> integrator
Post-conditions	1) If the new student does not have the required fields (ex. uni), return 412 Precondition Failed. 2) If successful, student is added to the database and returns 201: Resource created.
Assumptions	User is able to add classes until the database runs out of space. In that case, a system administrator can partition the Course microservice to scale.
Limitations	All products involved must be online.

Title	Request (GET) specific student data
Description	As a user, I can type in a URL and the application displays information about a certain student. This information comes from the database.
Products Involved	Router microservice, Course microservice, Integrator microservice, student database.
Pre-conditions	All products involved must be live.
Flow of Events	url -> router (GET Request) -> student microservice -> student database |-> integrator
Post-conditions	1) If student exists, return 200: OK and student data to the user. 2) If student does not exist, return failure due to ERROR 404: Resource not found.
Assumptions	One entry for student data per unique primary key.
Limitations	All products involved must be online.

Title	Update (PUT) specific student data
Description	As a user, I can type in a URL and the application updates information about a certain student in the student's database.
Products Involved	Router microservice, Course microservice, Integrator microservice, Course database.
Pre-conditions	All products involved must be live.
Flow of Events	url -> router (PUT Request) -> student microservice -> student database |-> integrator (PUT Request) -> course microservice
Post-conditions	1) If student exists, return 200: OK and student data is updated/changed in the database. 2) If student exists and the request is to add a course to the student's list of enrolled classes, Integrator is informed of the event and makes the PUT request to the Courses microservice to add the student's ID to the course's list of students enrolled. a) If successful, Integrator returns 200: OK to Students, and finally Students returns 200: OK to the user. b) If unsuccessful (ex. class is full), Integrator receives a 404: Not Found and makes a PUT request to Students to revert to the previous version of the student in question. Students returns 404: Not Found to the user. 3) If student does not exist, return failure due to ERROR 404: Resource not found.
Assumptions	One entry for student data per unique primary key.

Limitations	All products involved must be online.

Title	Delete (DELETE) specific student data
Description	As a user, I can type in a URL and the application deletes a student from the student's database.
Products Involved	Router microservice, Course microservice, Integrator microservice, Course database
Pre-conditions	All products involved must be live.
Flow of Events	url -> router (DELETE Request) -> student microservice -> student database |-> integrator
Post-conditions	1) If student exists, softdelete student data from database. Integrator is informed of the event and makes the DELETE request to the Courses microservice for all the courses the student was enrolled in. a) If successful, Courses returns 200: OK to Integrator, Integrator returns 200: OK to Students and Students returns 200: OK to user. b) If unsuccessful (ex. system failure), Integrator receives a 404: Not Found and makes a PUT request to Students to revert to the previous version of the student in question (undo delete). Students returns 404: Not Found to the user. 2) If student does not exist, return failure due to ERROR 404: Not Found.
Assumptions	One entry for student data per unique primary key.
Limitations	All products involved must be online.

Title	Add (POST) a specific course
Description	As a user, I can type in a URL and the application adds a course to the course database.
Products Involved	Router microservice, Course microservice, Integrator microservice, Course database.
Pre-conditions	All products involved must be live.
Flow of Events	url -> router (POST) -> course microservice -> course database |-> integrator
Post-conditions	1) If the new course does not have the required fields (ex. title), return 404: Not Found. 2) If successful, course is added to the database and returns 201: Resource created.
Assumptions	User is able to add classes until the database runs out of space. In that case, a system administrator can partition the Course microservice to scale.
Limitations	All products involved must be online.

Title	Request (GET) specific course data
Description	As a user, I can type in a URL and the application displays information about a certain course. This information comes from the database.
Products Involved	Router microservice, Course microservice, Integrator microservice, course database.
Pre-conditions	All products involved must be live.

Flow of Events	url -> router (GET Request) -> course microservice -> course database |-> integrator
Post-conditions	1) If course exists, return 200: OK and course data to the user. 2) If course does not exist, return fail due to ERROR 404: Not Found.
Assumptions	One entry for course data per unique primary key.
Limitations	All products involved must be online.

Title	Update (PUT) specific course data
Description	As a user, I can type in a URL and the application updates information about a certain course in the course's database.
Products Involved	Router microservice, Course microservice, Integrator microservice, Course database.
Pre-conditions	All products involved must be live.
Flow of Events	url -> router (PUT Request) -> course microservice -> course database |-> integrator
Post-conditions	1) If course exists, return 200: OK and course data is updated/changed in the database. 2) If course does not exist, return failure due to ERROR 404: Resource Not Found.
Assumptions	One entry for course data per unique primary key.
Limitations	All products involved must be online.

Title	Delete (DELETE) specific course data
Description	As a user, I can type in a URL and the application deletes a course from the course's database.
Products Involved	Router microservice, Course microservice, Integrator microservice, Course database
Pre-conditions	All products involved must be live.
Flow of Events	url -> router (DELETE Request) -> course microservice -> course database |-> integrator
Post-conditions	1) If course exists, return 200: OK and course data is deleted from database. 2) If course does not exist, return fail due to ERROR 404: Resource not found.
Assumptions	One entry for course data per unique primary key.
Limitations	All products involved must be online.

Note: User cannot directly make requests to the Student, Courses, or Integration Microservice. Integrator Microservice can only handle POST requests to take a ChangeEvent object and create new logs, but this is not externally observable from the user perspective.

3. Solution Integrations

3.1 Summary of Products

- Amazon Web Services - VM to run the application
- Python - Language being used to program all of our microservices.
- Flask - Framework to be used with Python for writing web applications.
- Eve - Framework to be used with Python and MongoDB in order to build RESTful APIs.
- MongoDB - Framework used to build and query the student databases and course databases.
- Redis - In-memory data structure used as database, cache and message broker.

3.2 Solution Use-Cases

Title	Get Information
Description	Users retrieve information using the GET request
Products Involved	AWS, MongoDB, Redis
Types of Integration	APIs
Functions	Use MongoDB for storage, Redis for caching data and serving recent requests

Title	Insert New Data
Description	Users insert new data using the POST request
Products Involved	AWS, MongoDB, Redis
Types of Integration	APIs
Functions	Use MongoDB for storage, Redis for caching data

Title	Update Data
Description	Users update existing data using the PUT request
Products Involved	AWS, MongoDB, Redis
Types of Integration	APIs
Functions	Use MongoDB for storage, Redis for caching data and serving recent requests

Title	Delete Data
Description	Users delete data using the DELETE request

Products Involved	AWS, MongoDB, Redis
Types of Integration	APIs
Functions	Use MongoDB for storage, Redis for caching data

4. Architecture Design

4.1 Overview

The overall architecture consists of four microservices. The Router microservice receives the request from the user. It then routes the request to either instances of the Students microservice, or to the Courses microservice depending on the paths indicated on its Config file. The Integrator microservice enforces referential integrity. We developed Alexander using the Python Eve framework and Mongo DB on an Amazon Web Services virtual computer (Linux).

4.2 System Design

First, the user sends a request to the Router microservice. This request may be, for example, adding a student to the university (please see the full set of supported requests under “2.1 - Summary of Capabilities”). The router parses the URL to determine which microservice to route the request to. For example, if the URI looks like “<IP address of Router>/students...” it will go to the students microservice, and if the URI looks like “<IP address of Router>/courses...” the request will go to the courses microservice. The Students microservice may have numerous instances for scalability purposes, so the target IP addresses of the Students and Courses microservices are not fixed; they are defined in a configuration file inside the Router microservice. This configuration file contains the addresses of all Students and Courses microservices. Each Students microservice manages a subset of students depending on the first letter of the students’ last names. So we define in our configuration file where to route the request based on the first letter of the student’s last name. There may only be one instance of the Student microservice or up to 26 instances (one for each letter of the alphabet). This is easily extensible to 26*26 instances, 26*26*26 instances, and so on, based on how many letters we consider, but for now, it is just the first letter.

The Students microservice updates its Mongo database based on the following supported CRUD URI patterns:

POST

URI: .../courses/students

Description: This allows a user to post a new student.
Success: 201 (Created) with header containing containing new unique ID.
Failure: 404 (Not Found), 409 (Conflict) if resource already exists.

GET

URI: .../students/<unique student ID (UID)>

Description: Returns information from the student with the specified UID
Success: 200 (OK), with all data from specified student (in JSON format).
Failure: 404 (Not Found), if UID is not found or invalid.

URI: .../students/<UID>/courses
Description: Returns the courses of the specified student based on his or her UID
Success: 200 (OK), JSON list of courses for specified student.
Failure: 404 (Not Found), if UID is not found or invalid.

URI: .../students
Description: Returns all student data (a JSON list of all students).
Success: 200 (OK), list of all students and their data.

PUT

URI: .../students/<unique student ID (UID)>
Description: Updates the student's fields with the specified header information
Success: 200 (OK) or 204 (No Content)
Failure: 404 (Not Found), if UID is not found or invalid.

DELETE

URI: .../students/<unique student ID (UID)>
Description: deletes a student
Success: 200 (OK)
Failure: 404 (Not Found), if UID not found or invalid.

The Courses microservice updates its Mongo database based on the following supported CRUD URI patterns:

POST

URI: .../courses/courses
Description: This allows a user to post a new course.
Success: 201 (Created) with header containing containing new unique ID.
Failure: 404 (Not Found), 409 (Conflict) if resource already exists.

GET

URI: .../courses/<unique course ID (CID)>
Description: Returns information from the course with the specified CID
Success: 200 (OK), with all data from specified course (in JSON format).
Failure: 404 (Not Found), if CID is not found or invalid.

URI: .../courses
Description: Returns all course data (a JSON list of all students).
Success: 200 (OK), list of all courses and their data.

PUT

URI: .../courses/<unique student ID (CID)>
Description: Updates the course's fields with the specified header information (adding a student to a course)

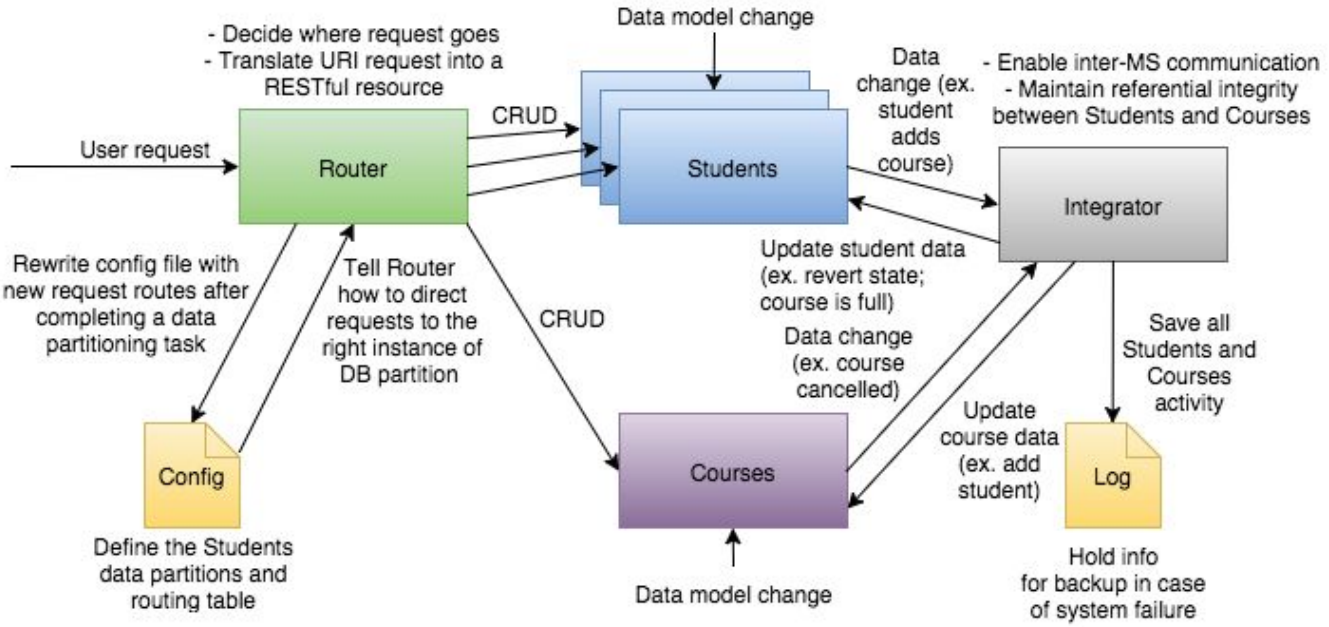
Success: 200 (OK) or 204 (No Content)
Failure: 404 (Not Found), if CID is not found or invalid.

DELETE
URI: .../courses/<unique student ID (CID)>
Description: deletes a course
Success: 200 (OK)
Failure: 404 (Not Found), if CID not found or invalid.

We allow the schemas of the Students and Courses databases to be changed. This requires a manual modification. We restrict the changing of a data type of a column (for example, the Students microservice will throw an error if you have changed the name last_name column to an integer type). However, one is freely able to add columns or move them around without impacting the functionality of the Students and Courses microservices.

For scalability purposes, we have designed the concept of data partitioning between instances of the same microservice. This is done through the Router, by making a GET call to ".../students", returning all student data. Then, the Students microservice is shut down (manually), and three instances of the Students microservice are executed (manually). Each instance is then populated with their third of the data through POST requests.

Up to this point, we have covered the yellow (Config), green (Router), blue (Students), and purple (Courses) microservices and files visible below.



In addition, we have implemented the Integrator microservice (in gray), which keeps track of the operations logged by the Students and the Courses microservices and enforces referential integrity. Once the Integrator detects that a change in Students or Courses is made that influences the other party, it makes a request to the other microservice to make the necessary changes. For example, if a student adds a new course, the Integrator service will receive a POST request from the Students microservice, which will populate the Integrator's log file with one more entry. This log entry looks as follows:

StudentChange event {
event: event_string
new_student: {new_student_data}
original_student: {old_student_data}
time: time_string
}

As one can see above, the log entry contains not only the new student data, but also the old student data. When the Integrator performs a PUT on the Courses microservice to update the class, adding the new student, if that request fails, the Integrator now has the old student data to be able to revert the student via a PUT call to the Students microservice.

4.3 Dependencies

Our 3rd party products have the following dependencies:

- Amazon Web Services - Independent.
- Python - Independent.
- Flask - Dependent on Python.
- Eve - Dependent on Python and MongoDB.
- MongoDB - Dependent on JSON.
- Redis - Independent.

Our application has the following dependencies:

- Router - Dependent on the Configuration File.
- Configuration File - Independent.
- Student Microservice - Dependent on student database.
- Student Database - Independent of other services within this application.
- Course Microservice - Dependent on course database.
- Integrator - Dependent on logging file.
- Log File - Independent.

4.4 Environment

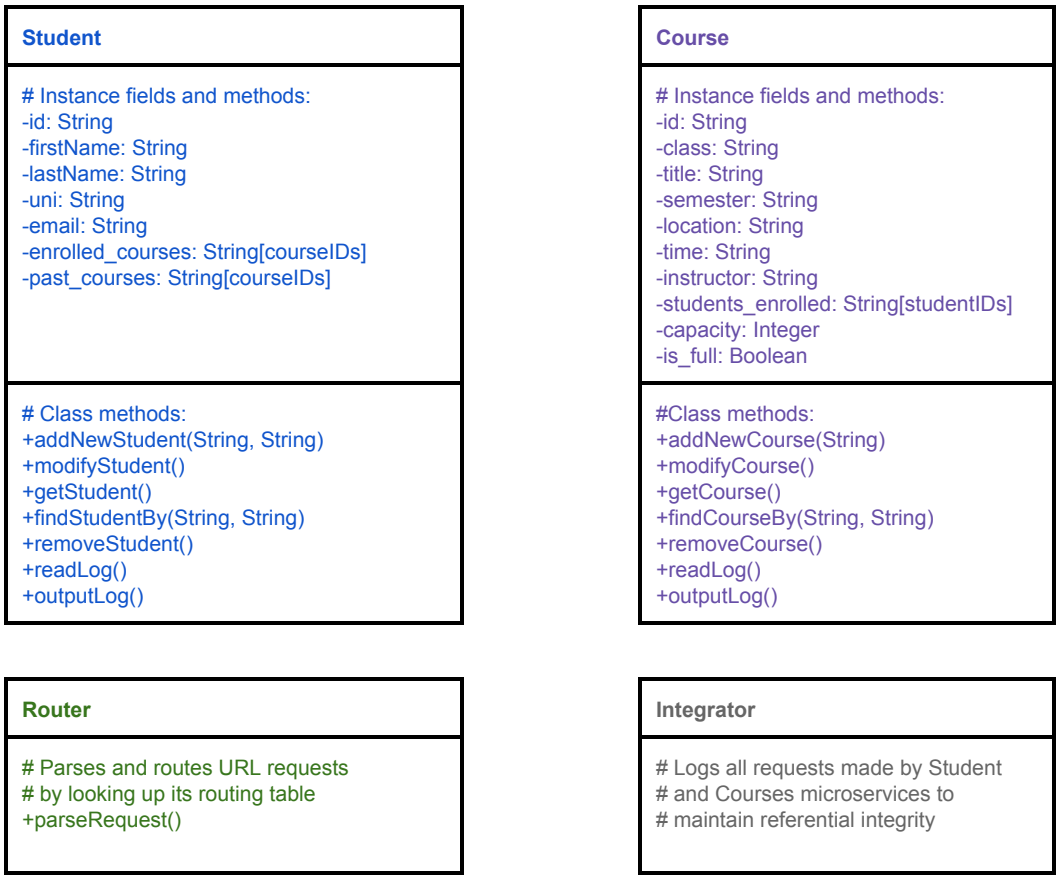
As stated above we will be using a specific virtual machine created on Amazon Web Services to run our virtual machine on an online cloud environment. We plan to upload the Linux operating system onto our virtual machine. Within this environment we will then start to install the packages, libraries and tools required for completing this project.

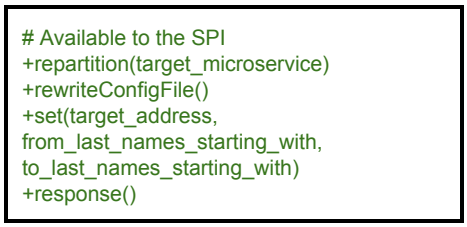
4.6 Integration/Services Interfaces

As explained in Section 2, user requests will be served through the following four operations: GET, POST, PUT, and DELETE. See the next section for the data models.

4.7 Data Models

The data models we will implement are Student, Course, Router and Integrator. Their class diagrams describe the fields and class methods in the schema.





The choice of persistent storage in MongoDB means that if we want to make any changes to the schema, we don't necessarily need to write a migration script to convert the data from the old to the new schema, but could just update each document with a version number. Using a versioning system for the document instances of our data model saves us from the toll on the server of having to read all the documents and rewriting them with the new schema (consider the example of an application with 100 million users but only 100,000 active users). Our application should support reading documents in all versions and update/write only the latest version.

Consistency and Integrity Considerations:
The Courses and Students microservices have the following associations:

1) Students contains the list enrolled_courses, which depends on Courses data
2) Courses contains the list students_enrolled, which depends on Students data

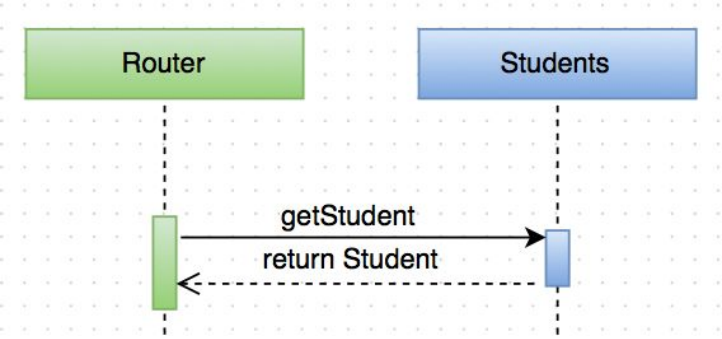
The Integrator microservice maintains data integrity between the two microservices. However, when a an existing student adds an existing course to its list of courses, for a brief gap there is a loss of data integrity where the course in Courses does not yet know the student wishes to add the course. Other similar scenarios where there is a temporary break in integrity include:

- Courses: when a course has been canceled, all the students have the course deleted from the schedules (but the student should not be deleted from the school)
- Students: when a student has been deleted, they must be removed from the enrollment listings of all the courses that they are in

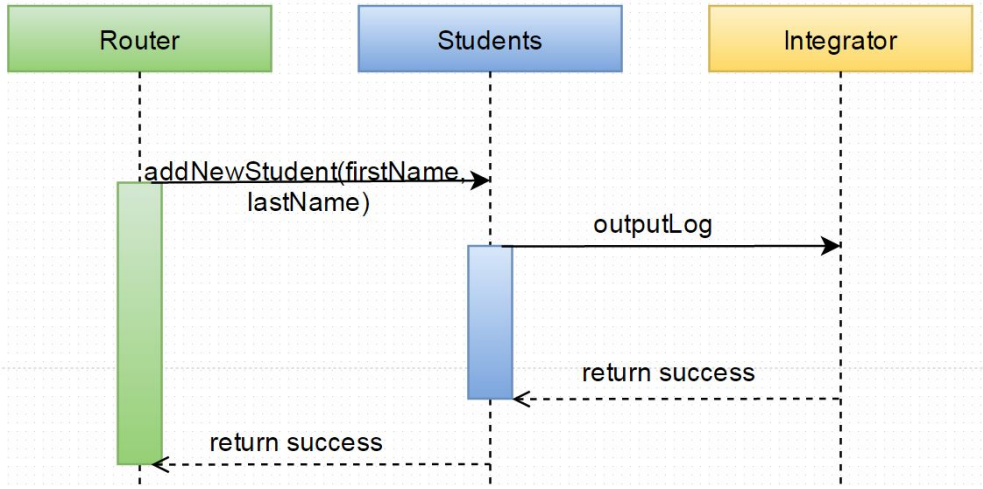
The Integrator exists to ensure the system application has a consistent view of its data as a whole and any such breaks in integrity are quickly resolved in order to keep referential integrity and consistency across the two databases.

Sequence Diagrams:

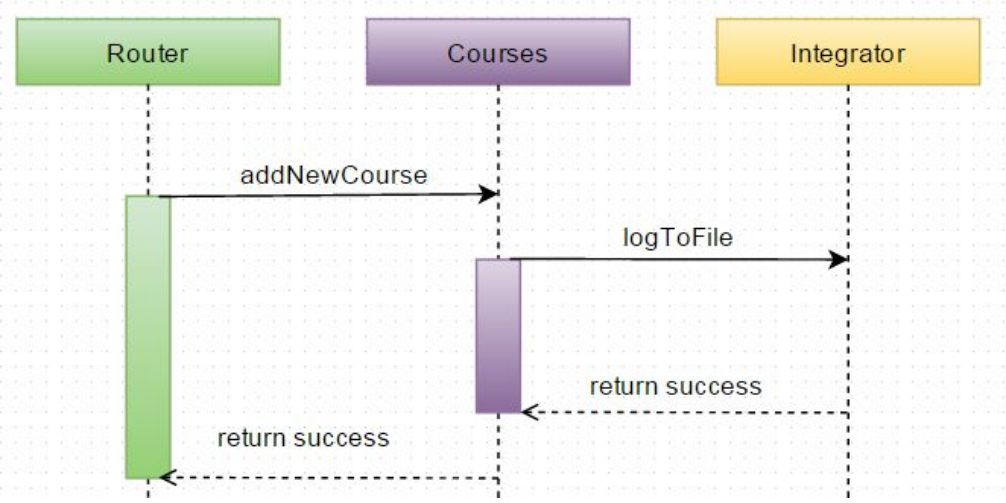
Get Information: The example below shows the result of a get student operation. The diagram would be similar if for a get course operation.



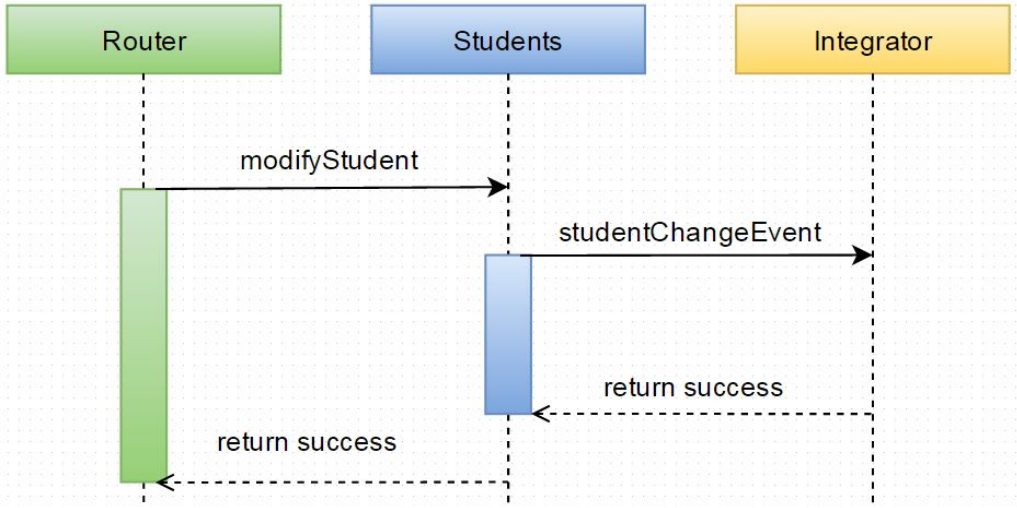
Insert New Data: The diagram below shows the result of an insert new student operation. The sequence diagram for an insert new course operation is similar.



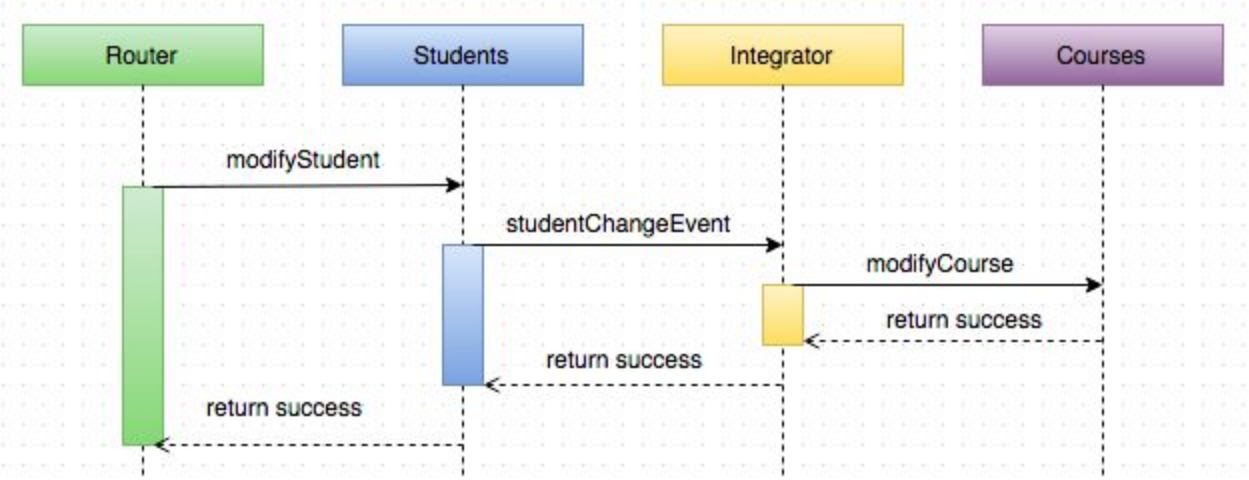
One may also want to POST a new course. In that case, the sequence diagram would look as follows:



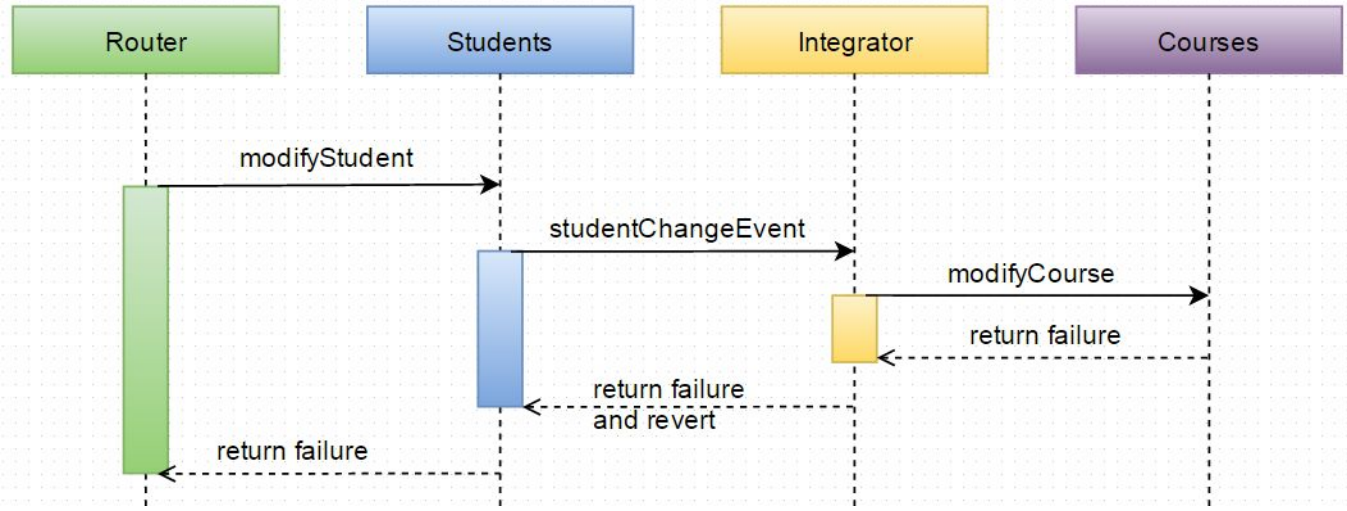
Update Data: The diagram below shows the result of an update student operation. The sequence diagram for an update course operation is similar, with the Students microservice column representing the Courses microservice column. An operation of this type may be updating a student's first name (which does not concern the Courses microservice)



When a PUT operation is done to the Students microservice that affects a Course microservice field, the Integrator makes a PUT request to the Courses microservice to reflect the change:

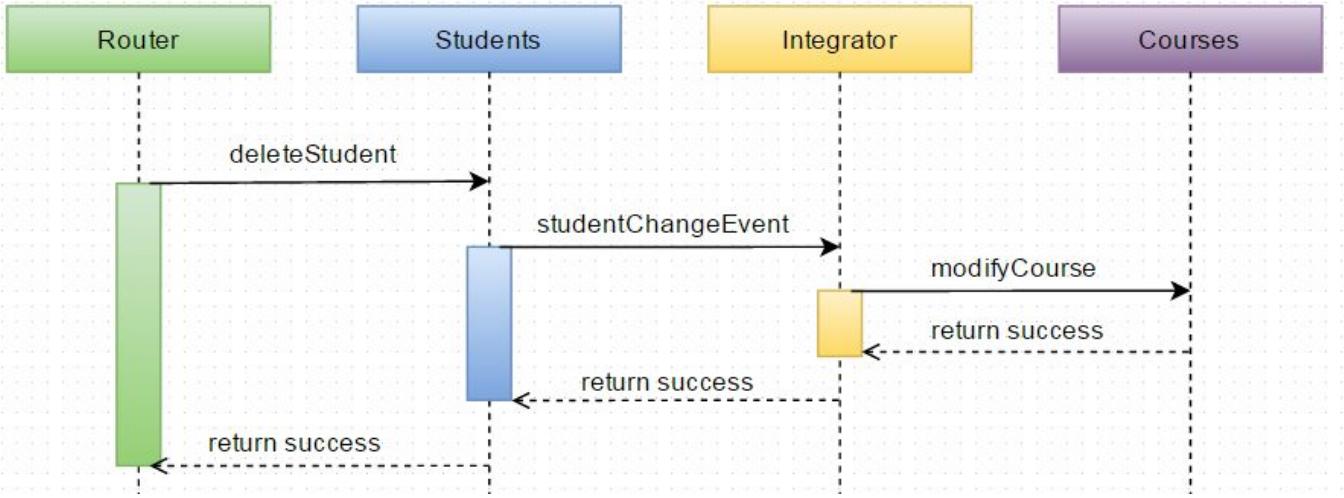


However, if that request fails when updating a course in Courses, the sequence looks as follows:



Note that when Integrator receives a failed response from Courses, it updates the studentChangeEvent status to failed. In turn, when Students receives a failed response from Integrator, internal logic will cause the student to revert to its previous version in the database.

Delete Data: The diagram below shows the result of a delete student operation. The sequence diagram for a delete course operation is similar.



4.9 Extensibility

The system provides the option to change the schemas of the Students and Courses data models, so that different universities can adapt it for their systems.

Future extensibility possibilities:

- Make the Courses microservice scalable, so that several instances of the Courses microservice can fulfill requests that fall within their specified course ID range.
- Add authentication so that a student cannot view or change another student's course list.
- Add an extra attribute for course schedules, so that students are not allowed to take courses whose timeframes overlap.

4.11 Performance

We would like for our databases to be persistent. Meaning that the databases should be able to be run for long periods of time while maintaining consistently preserved data. The system is allowed to be down only when the data model or the configuration file is being changed. Additionally, our Course Microservice and Student Microservice will be returning messages back to the Router based on the status of each operation completed. We would like for these messages to be communicated to the Router in a reasonable amount of time. For a computing system, this shouldn't take more than a few milliseconds. Moreover, the Router will be communicating with the user based on the messages it receives from the Student and Course Microservices. We would like for the Router to relay this message in a reasonable amount of time.

4.12 Scale

There are three forms of scaling:

- X-axis scaling: This form of scaling involves running multiple instances (copies) of an application behind a load-balancer. This is not implemented in Alexander.
- Y-axis scaling: This type of scaling involves the splitting of the functionality of a web service into different services. We implement this type of scaling in Alexander. We have divided our design into Courses, Students, Integrator, and Router microservices. Because of the form of our architecture, a new microservice can easily be added. For example, one can split the Integrator into a logging microservice and an integrity constraint microservice without much additional work.
- Z-axis scaling: This form of partitioning involves running identical copies of an instance of a service, partitioning that service's data among the new services. We have implemented this in Alexander for the Students microservice. This is done through the Router, by making a GET call to ".../students", returning all student data. Then, the Students microservice is shut down (manually), and three instances of the Students microservice are executed (manually). Each instance is then populated with their third respective of the data through automated POST requests. As described previously, the router's configuration file is then updated (manually) to route certain letters of the alphabet to different addresses (to different instances of the Students microservice).

The major scaling issue of our application is the scaling of our databases. We have already mentioned that in order to handle this issue, we will employ a data partitioning mechanism along the z-axis of the scale cube within our Student Microservice and Course Microservice. Within these microservices there are functions that we as maintainers will call in order to repartition the data. As mentioned previously, when making this change to the data model, the system and application will be taken offline.

4.13 High Availability

The system is not expected to stay online when the data model and/or router Config file is being changed. Otherwise, the biggest threat to the system's availability is the ability of the Students and Courses database to stay current to each other's changes. Since this system does not make use of asynchronous requests, only one user's request can be fulfilled at a time. Therefore, if the system fails, then it is not because it was flooded with requests.

If the request to Students or Courses is unsuccessful, the user will receive a message from the system containing the error code and message that best describes the failure.

4.14 Installation/Deployment/Distribution

It has come to our attention that it is difficult to ship an application while working under different operating systems. Thus, although the application itself can be run on a variety of operating systems

(Macintosh, Windows, and Linux), we have decided to develop the application on an Amazon Web Service (AWS) virtual machine (VM). Thus, if you want to run it on an environment that is guaranteed to work, please follow the instructions here to setup an AWS VM and import a VM image: https://aws.amazon.com/ec2/vm-import/

We will provide the VM image. Otherwise, you may opt to work with the project on the environment of your choice, in which case you will need to follow the directions on the following link in order to setup Python Eve and Mongo DB:
http://python-eve.org/install.html

Aside from this, there is no authentication; the simple download of our project will suffice. You will run the app by entering “python run.py” in your terminal in the root folder of Alexander’s directory system.

4.15 Configuration and Administration

For the sake of Alexander, we have not separated APIs (application programming interfaces), which accept requests from the average user from SPIs (system programming interfaces), which are used by admins for configuration purposes. In other words, no authentication is performed to control a regular user from taking an administrative role from the surfaced functions.

- Changing a server for scalability purposes or data partitioning requires manual input and the server to be shut down.
- Alexander was built on a Linux platform, although it is compatible with Windows or Mac. It is also compatible with any operating system that supports Python Eve and Mongo DB.

5. Challenges

One major challenge has been on how to make the microservices modular and configurable, as well as on how to arrange and design the modular pieces as to minimize the number of dependencies. There are various patterns to handle referential integrity such as using an integrator or the pub/sub pattern; after some discussion, our group reached the consensus to use an integrator. Another challenge was deciding between creating a new LogReader file or just keeping a master log in the Integrator (we decided on the latter). In addition, determining which microservice should handle data partitioning also posed somewhat of an issue. We decided to put such logic in the Router, in order to simplify the step of rewriting the config file. Finally, another challenge was integrating these tools in order to make them work together while maintaining data integrity.

6. References

- Building REST APIs Using EVE: https://code.tutsplus.com/tutorials/building-rest-apis-using-eve--cms-22961
- AWS SDK for Node Js: https://aws.amazon.com/sdkfornodejs/
- The MongoDB 3.0 Manual: https://docs.mongodb.org/manual/