

# 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Any outdated hard copy is invalid and must be removed from possible use.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 |
| 2.0 | 11/8/2015 | M0TL3Y | Final 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 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 a message 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. This microservice can be programmatically
partitioned into three student microservices.

*B) Courses*: a course info management microservice, including data model changes. It includes info on present courses using the course id attribute.

*C) Router*: a request router microservice fronting the Students and Courses microservices.

*D) Integrator*: a basic referential integrity manager microservice that maintains referential integrity between the students and courses 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>Supported 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)

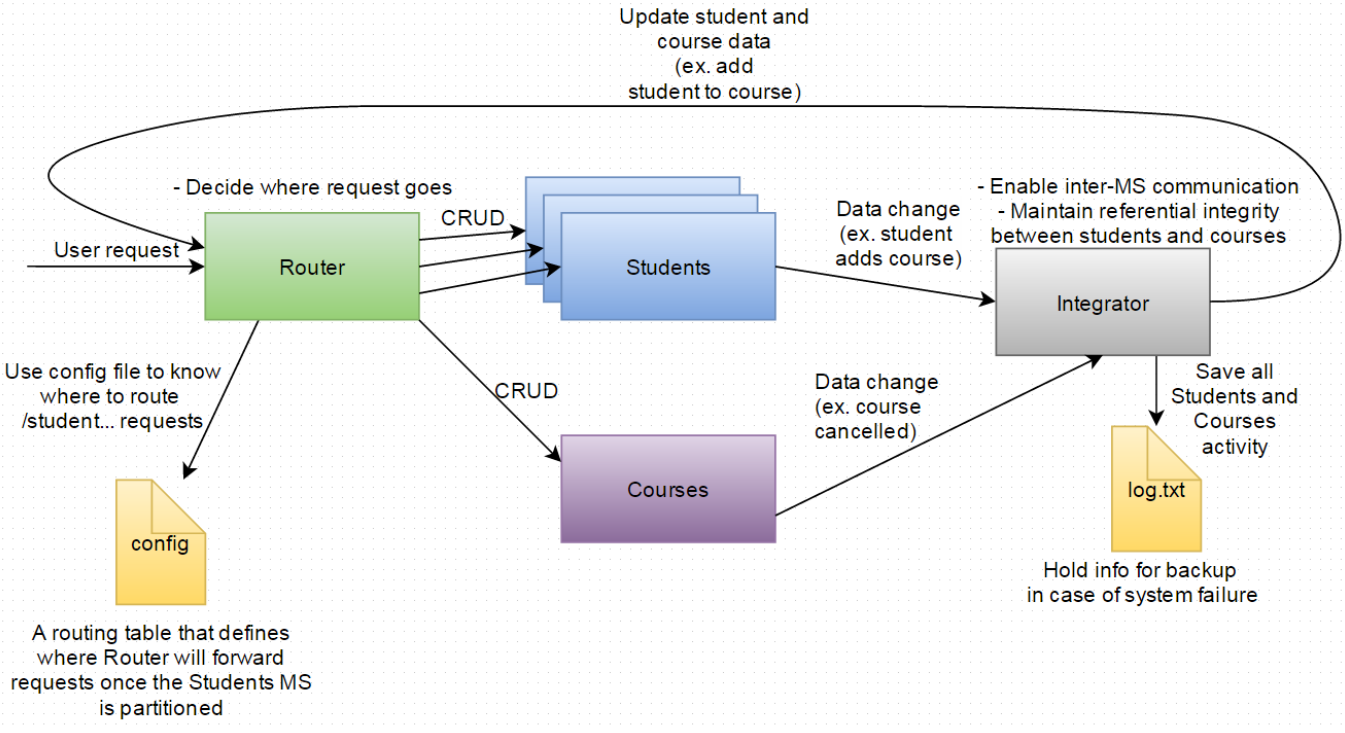
3) Update student info, including adding a class to a list of the student's enrolled classes
4) Delete student from university

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




***Note that this diagram does not show responses. 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 occur, such as when the student leaves the university and needs to drops all their

courses, these two databases are updated to reflect these changes.

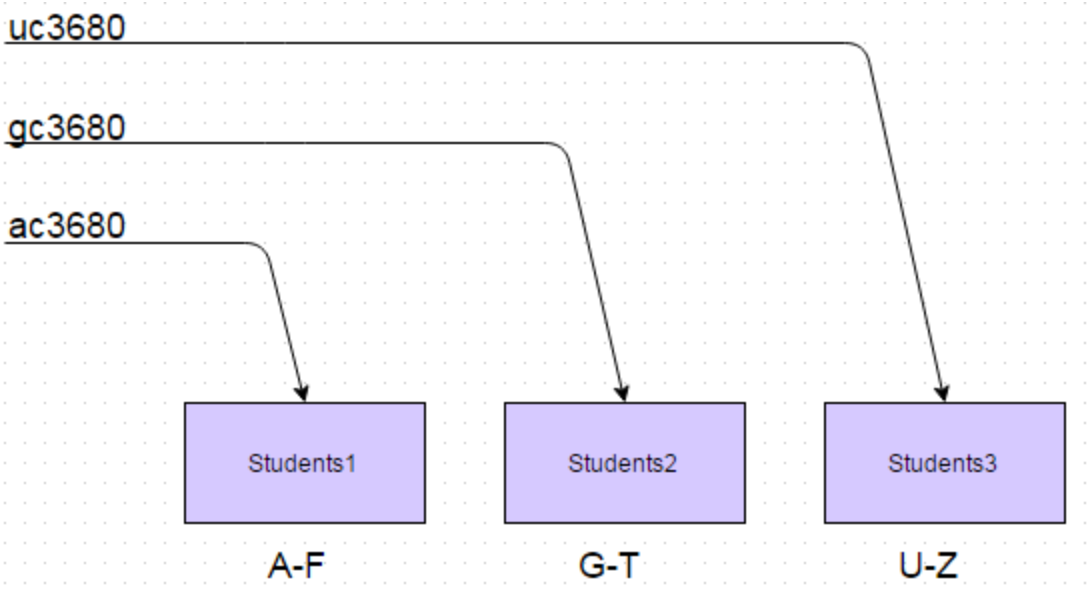
It ensures data consistency and resolves dependency issues between Students and Courses by receiving messages from both the Students and Courses microservices whenever either database is changed, then forwarding messages about important changes to the other microservice through the Router.

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. We have created a script called “partition.py” that handles the data partitioning of Students based on 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 microservices and databases, which will respectively contain students’ data whose last names begin with A-F, G-T, and U-Z respectively).

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, Student microservice, Integrator microservice, Student database. |
| Pre-conditions | All products involved must be live. |
| Flow of Events | url -> router (POST) -> student microservice -> student database<br>\|-> integrator |
| Post-conditions | 1) If the new student does not have the required fields (ex. uni), return 412 Precondition Failed.<br>2) If successful, student is added to the database and returns 201: Resource created. |
| Assumptions | User is able to add students until the database runs out of space.<br>In that case, a system administrator can partition the Student 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, Student microservice, Integrator microservice, Student database. |
| Pre-conditions | All products involved must be live. |
| Flow of Events | url -> router (GET Request) -> student microservice -> student database<br>\|-> integrator |
| Post-conditions | 1) If student exists, return 200: OK and student data to the user.<br>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, Student microservice, Integrator microservice, Student database. |
| Pre-conditions | All products involved must be live. |
| Flow of Events | url -> router (PUT Request) -> student microservice -> student database<br>\|-> integrator (PUT Request) -> course microservice |

| | |
|---|---|
| Post-conditions | 1) If student exists, return 200: OK and student data is updated/changed in the database.<br>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 POST request to the Courses microservice to add the student's ID to the course's list of students enrolled. User then receives 200, OK.<br>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, Student microservice, Integrator microservice, Student database |
| Pre-conditions | All products involved must be live. |
| Flow of Events | url -> router (DELETE Request) -> student microservice -> student database<br>\|-> 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. User then receives 200, OK.<br>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<br>\|-> integrator |
| Post-conditions | 1) If the new course does not have the required fields (ex. title), return 404: Not Found.<br>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.<br>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<br>\|-> integrator |
| Post-conditions | 1) If course exists, return 200: OK and course data to the user.<br>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<br>\|-> integrator |
| Post-conditions | 1) If course exists, return 200: OK and course data is updated/changed in the database.<br>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<br>\|-> integrator |
| Post-conditions | 1) If course exists, return 200: OK and course data is deleted from database.<br>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, receiving a message about the changes in a microservice. This is not externally observable from the user's perspective.

# 3. Solution Integrations

## 3.1 Summary of Products

- Python - Language being used to program all of our microservices.
- Flask - Framework to be used with Python for writing web applications.
- MongoDB - Framework used to build and query the student databases and course databases.

## 3.2 Solution Use-Cases

| Title | Get Information |
|---|---|
| Description | Users retrieve information using the GET request |
| Products Involved | MongoDB |
| Types of Integration | APIs |
| Functions | Use MongoDB for storage. |

| Title | Insert New Data |
|---|---|
| Description | Users insert new data using the POST request |
| Products Involved | MongoDB |
| Types of Integration | APIs |
| Functions | Use MongoDB for storage. |

| Title | Update Data |
|---|---|
| Description | Users update existing data using the PUT request |
| Products Involved | MongoDB |
| Types of Integration | APIs |
| Functions | Use MongoDB for storage. |

| Title | Delete Data |
|---|---|
| Description | Users delete data using the DELETE request |
| Products Involved | MongoDB |
| Types of Integration | APIs |
| Functions | Use MongoDB for storage. |

# 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 Flask framework and Mongo DB.

## 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 three instances, each covering a subset of the alphabet.

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

POST
- URI: .../students
  - Description: This allows a user to post a new student.
  - Success: 201 (Created) with header containing new unique ID.
  - Failure: 409 (Conflict) if resource already exists.
- URI: .../students/<UID>/courses
  - Description: This allows a user to enroll a student in a course.
  - Success: 201 (Created) with header containing new unique ID.
  - Failure: 404 (Not FOund) if student does not exist.

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.

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

Description: Removes a student from a course

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

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.

URI: .../courses/<unique course ID (CID)>/students

Description: Returns students enrolled in specified course.

Success: 200 (OK), list of all students in the course.
Failure: 404 (Not Found), if CID is not found or invalid.

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)
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.

URI: .../courses/<CID>/students/<UID>

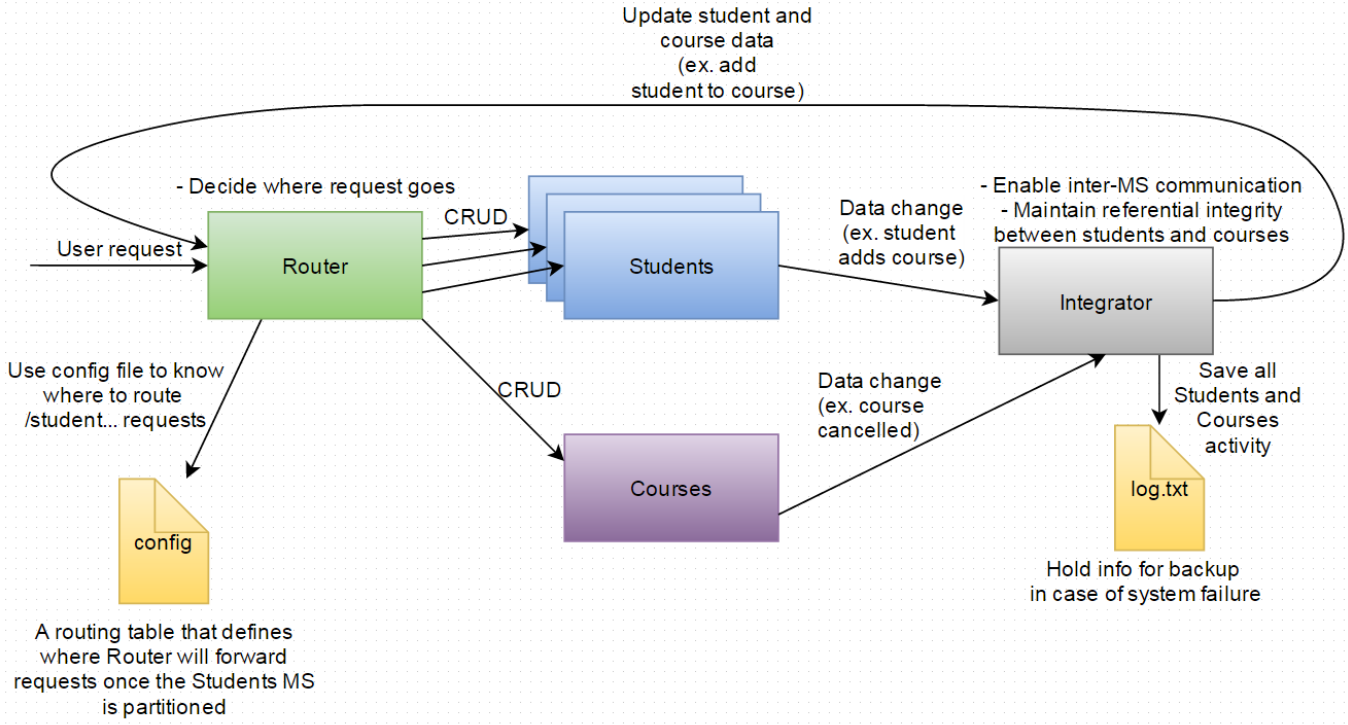
Description: deletes a student from a course
Success: 200 (OK)
Failure: 409 (Conflict), if UID not found within course.

We allow records with flexible schemas into the Students and Courses databases. This is inherited from Mongo DB's flexibility in terms of data type and record structure (schema). The only restriction are the primary keys, CID and UID, which must be present in every record. All of Alexander's functions take into account the fact that some records may be posted with more or less fields than others.

For scalability purposes, we have designed the concept of data partitioning for the Students microservice. There is a "config" file in the main directory which indicates the router's routes (target port numbers) for each letter of the alphabet. Initially, since there is only one Students microservice, there is only one target port. However, after partitioning, there are three different ports, one for each letter set: A-F, G-T, and U-Z. To partition, (1) terminate all executed processes, starting from the router (disables incoming requests). (2) Delete the "config" file. (3) Use the script: SetupTools/createConfig2.py to create "config2" or simply move "config2" to the api folder and rename it to "config". You may also choose to rename the routes in "config" yourself rather than delete the file; the method proposed of swapping "config2" in the place of "config" is for your convenience. (4) Run partition.py (this partitions the Students database and inputs the contents of the database into the three new databases created for the three new Students microservices). (5) You can run the program at this point, executing "python router.py", "mongod", "python integrator.py courses 9001 3 students 9003 9004 9005 ", and "python courses.py". The three students.py microservices are run as follows: "python students.py 9003" "python students.py 9004" "python students.py 9005". Note that they take the port number as arguments. You should now be running the application with three Students microservices, each with its own partition of the data that was in the original Students microservice.

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 enforces referential integrity between Students and Courses. Once the Integrator receives a message from Students and/or Courses that necessitates a database change in the other microservice, it will send a request to the other microservice through the Router, making the necessary changes.

To invoke the integrator: python integrator.py <courseMS_URL> <courseMS_port> <number of student partitions> for each student partition, list the following: <studentMS_URL> <studentMS_port>

Example with two Students partitions: python integrator.py http://127.0.0.1:9001/ 9001 2 http://127.0.0.1:9002/ 9002 http://127.0.0.1:9003/ 9003

where Courses runs on port 9001, and the two Students partitions run on ports 9002 and 9003.

There are essentially ten cases where Students and/or Courses interact with the Integrator. Their diagrams and descriptions are in Section 4.7. In all cases, Students and/or Courses will POST to the Integrator using the route: /integrator/<primary_key>, supplying their primary key (cid or uid).

Additionally, they will send a data payload in the following format:
{"port": <port number>, "v1" : <old record>, "v2": <new record>, "cid": <cid>, "uid": <uid>, "verb": <CRUD

operation>}

The Integrator needs the port number to figure out which micro-service sent the request, and thus whom to forward information to. The verb represents what operation the micro-service performed, and Integrator needs to know this in order to figure out the route that it should contact the other micro-service through. The old, new records represent the state of the micro-service's record before and after they performed the CRUD operation on that record.

When needed, the Integrator sends a message to Students and/or Courses, adding the following data: {"uid":uid, "forward":"False"} or {"cid":cid, "forward":"False"}

Where "forward":"False" tells Students and/or Courses that they should not contact Integrator after it receives the message from it. This prevents an infinite loop where there Students and Courses continuously send messages to each other through the Integrator, and make the same changes over and over again.

## 4.3 Dependencies

Our 3rd party products have the following dependencies:

- Python - Independent.
- Flask - Dependent on Python.
- MongoDB - Dependent on JSON.

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 - Independent.
- Log File - Independent.

## 4.4 Environment

Alexander is compatible with Windows, Linux, or Mac. It is also compatible with any operating system that supports Python and Mongo DB.
To setup your environment please perform the following steps:

(1) Install MongoDB. Please follow the steps on the Installation link on this page: https://docs.mongodb.org/manual/
(2) Install Python 2.7.
(3) Do "pip install pymongo" on your terminal

(4) Do “pip install flask” on your terminal
(5) Do “pip install Werkzeug” on your terminal
(6) Do “pip install requests” on your terminal

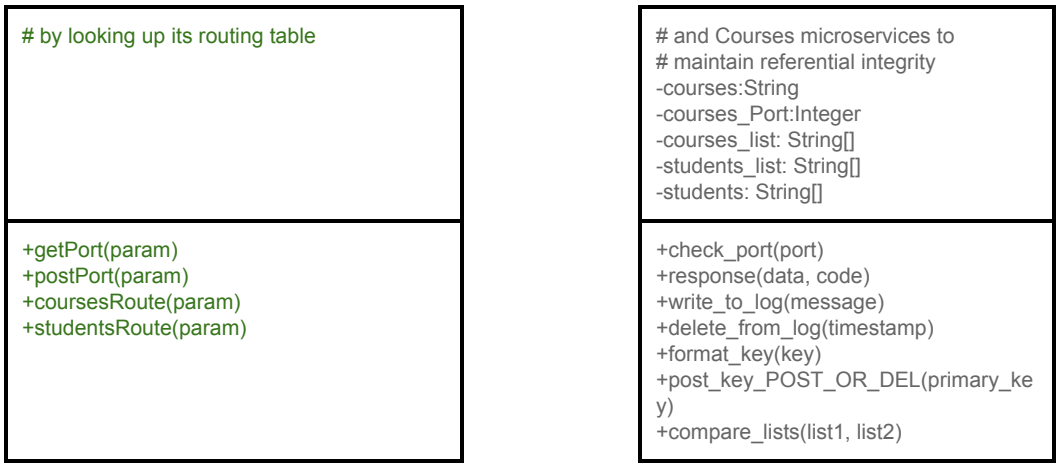
At this point your environment is setup. Please see the README.md, included in the submitted code, for example calls and procedures. The majority of that information is also found in this documentation, but not in a concise way specific to practical use.

## 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

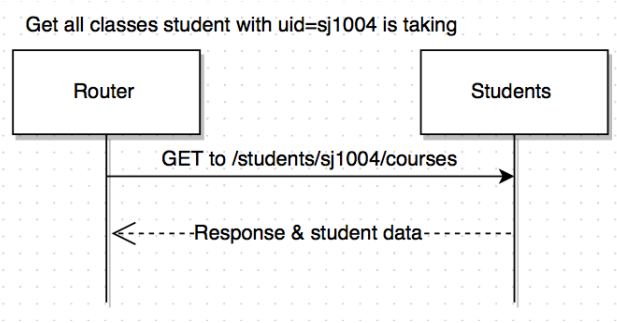
The Integrator exists to ensure the system application has a consistent view of its data as a whole and any such breaks in integrity are quickly resolved in order to keep referential integrity and consistency across the two databases.

**Sequence Diagrams:**

In all of these examples, the user sends the request through the Router and receives the response and any associated data through the Router. In some of these calls, the Router will send additional data via a request form or request data, that the recipient micro-service must parse.

These sequence diagrams are not drawn to illustrate the exact order in which responses are returned, ie. the Router is not waiting around for the entire Router <-> Students <-> Integrator <-> Courses sequence to complete before returning a response to the user.

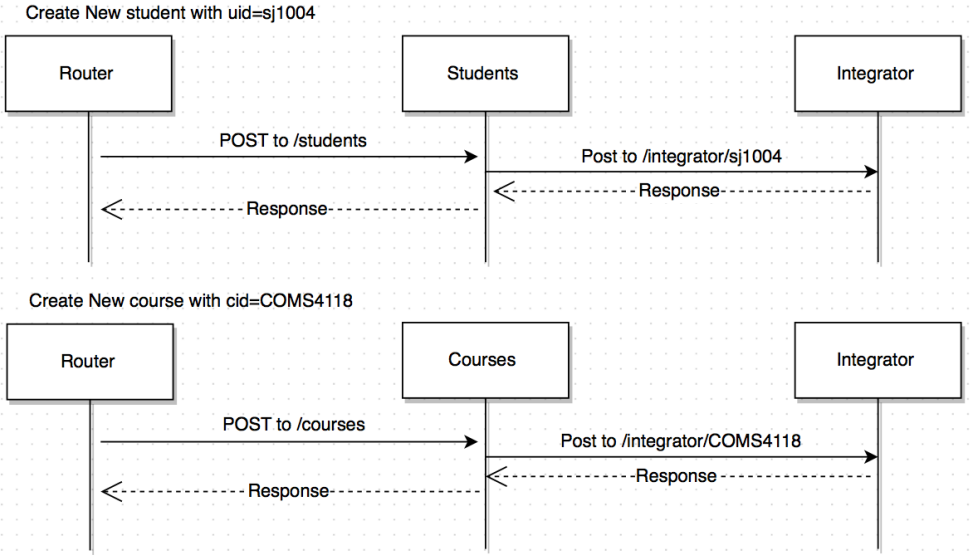
**Get Data:** There are several types of get operations (ex. get all student info, get all course info, get the specific classes a student is taking), but their sequence diagrams are very similar. Below is the sequence diagram for retrieving all of the classes a student is taking from Students.




**Insert New Data:** There are two types of inserts, creating a new student and/or course, and enrolling a student into a course. The latter case requires communication between Students and Courses, which is accomplished through contacting the Integrator.

Creating and new student, and creating a new course does not require sending a notification to the other micro-service. An example of the data sent while posting to students is below:
"firstName=Steve&lastName=Jobs&uid=sj1004"

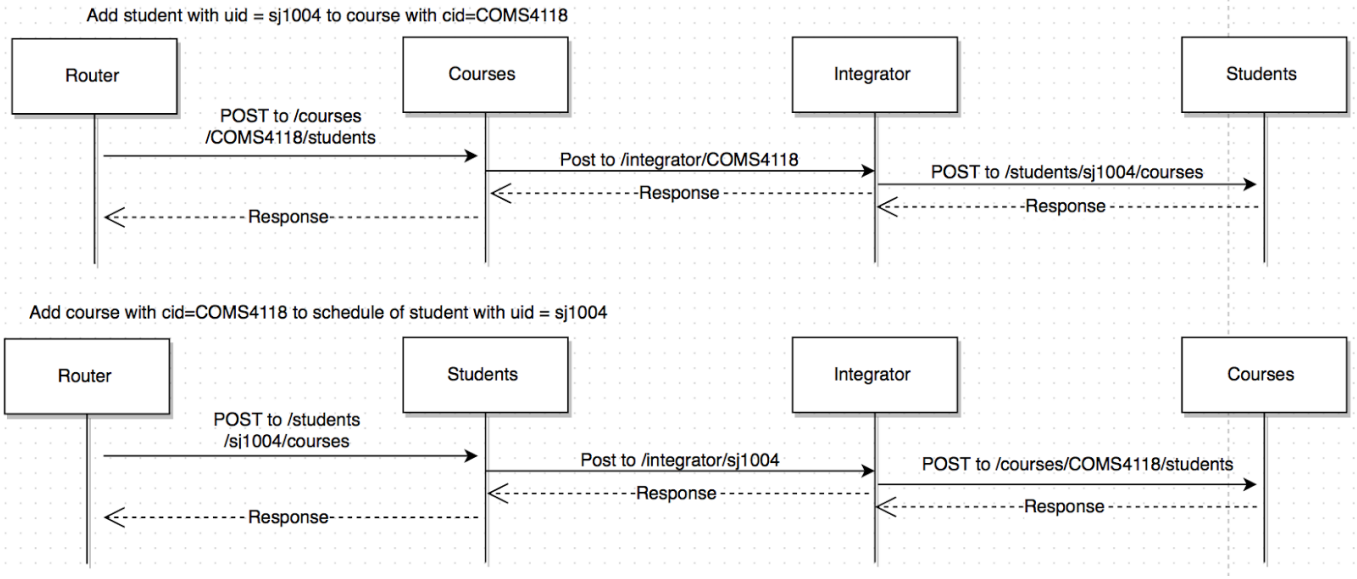
An example of the data sent while posting to courses is below:
"cid=COMS4118"



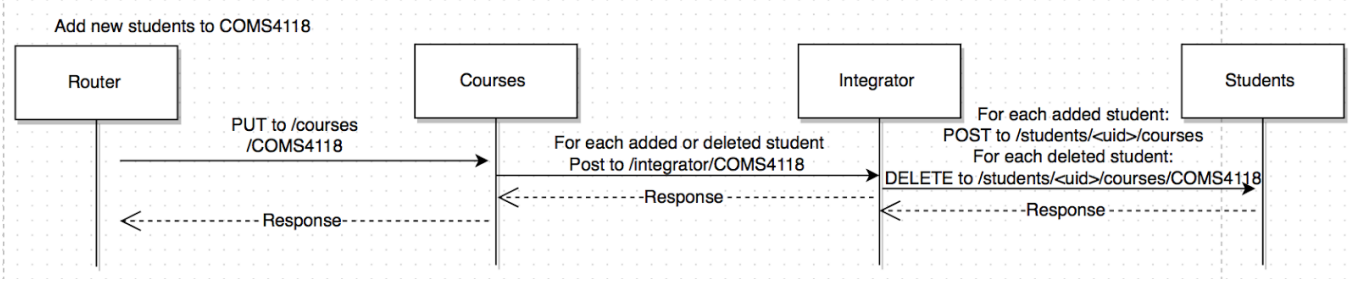

Adding students to classes can be done from either Students or Courses. From the Students side, the operation is described as adding a class to a student's schedule. From the Courses side, it's described as adding a student to a class. Either change will have the same result, that of the student being enrolled in the class.

Example of data sent if you try to enroll a student in a course through the Router contacting Courses: "uid=sj1004"

Example data sent if you try to enroll a student in a course through the Router contacting Students: "cid=COMS4118"

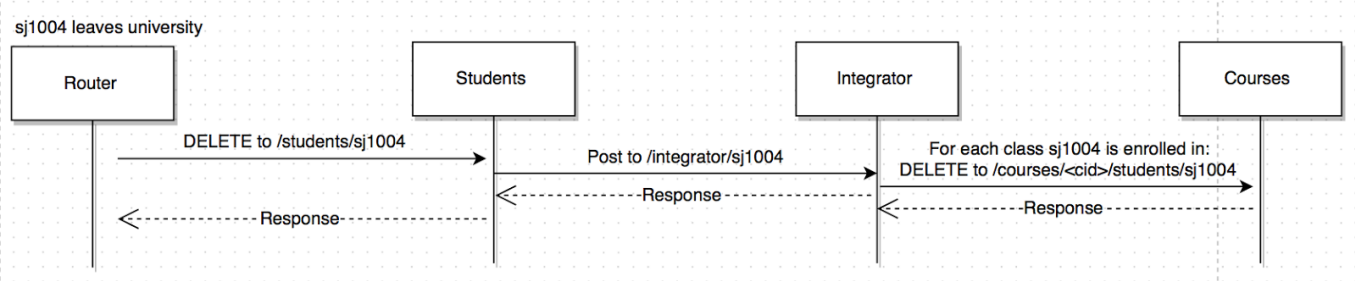


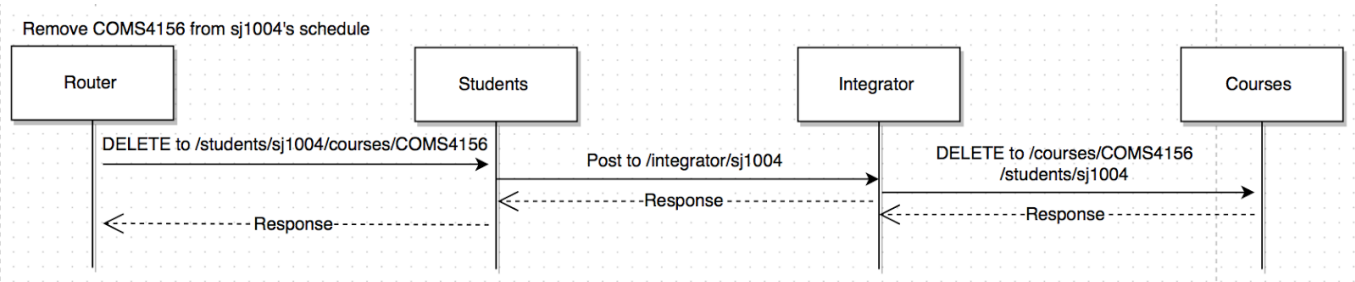

**Update Data:** There are two types of updates, updating data in a student through Students, and updating info about a class via Courses. Below is an example of adding more students to a course through Courses, additionally supplying the following data:
"uid_list=Nina,Melanie,Agustin,Bailey"




In this case, the Integrator will tell Students to add Nina, Melanie, Agustin, and Bailey to COMS4118.

**Delete Data:**
There are two types of deletion, deleting a student or course, and removing a class from a student. Below is an example of deleting a student (top diagram), and deleting a class from a student's schedule through Students (bottom diagram).




*Note: The primary keys resource in the integrator route was primarily used for clarity and ensuring that the Students and Courses calls were correct.

## 4.9 Extensibility

The system provides the option to input records with different schemas into 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, for example, 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 down when partitioning is being performed through our provided automated script and some brief administrative actions. Additionally, our Course Microservice and Student Microservice will be returning messages back to the Router based on the status of each operation completed. We believe that these messages to be communicated to the Router are done in a reasonable amount of time (milliseconds).

## 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 partitioning is occurring. 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, the application itself can be run on a variety of operating systems (Macintosh,

Windows, and Linux).

Aside from this, there is no authentication; the simple download of our project will suffice. You will run the app by entering "python router.py", "mongod", "python integrator.py courses 9001 1 students 9002", and "python courses.py" in your terminal in the api 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 is compatible with Windows, Linux, or Mac. It is also compatible with any operating system that supports Python 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. Another challenge was deciding between creating a new logging microservice or just have the Integrator deal with the logs directly (we decided on the latter). Finally, another challenge was integrating multiple microservices in order to make them work together while maintaining data integrity.

## 6. References

- The MongoDB 3.0 Manual: https://docs.mongodb.org/manual/