***Top Level Design Specification (TLDS)***

***PURPOSE***

*The purpose of a Top Level Design Specification (TLDS) is to ensure that a proposed high level architecture approach will adequately satisfy the requirements of a new product, new major component(s), or major changes to an existing solution. The completed document is a single repository for the collection of information needed to record the design of a product, for R&D Tax, Legal and IP Purposes, with as little duplication as necessary.*

***APPLICABILITY***

*This document is ONLY applicable to new products, and major releases of existing products.*

*It is NOT applicable to minor product releases, or to continuous development (aka Agile, SaaS) products provided they already have a completed TLDS.*

***INSTRUCTIONS***

*The TLDS can be saved, and/or exported to a different file format, as needed. Either PDF or DOC format are acceptable as the final version.*

* *For New products, you can provide links to the appropriate data sources, as well as brief descriptions in each section.*
* *For a major release of an existing product, are encouraged to do the same, but at a minimum should provide brief descriptions for the new functions. You can embed a copy of a prior TLDS, or other high level design document as a PDF or DOC file.*
* *The TLDS template should be used as a guideline, and a reminder of the type of information that should be covered in order to establish a quality, approved architecture direction.*
* *There are some important objectives of the TLDS to keep in mind as a guide when producing the content. The content is relevant regardless of the development process that will be followed. Whether it is a waterfall or agile development process to be used, there has to be an adequate amount of information available to produce the high level use-cases that will drive the rationale for the proposed high level architecture approach of a TLDS in order to move forward with the project and gain approvals.*
* *Links to other documents and repositories should be accessible*

***Note:*** *the term “document” may refer to either a physical document, or an electronic system for collecting and reporting the information, with version history. Development teams are encouraged to use whatever tools work for them assuming that all of the data is present.*

***Template Revision History***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date Changed** | **Completed By** | **Description of changes** | **Approved By** |
| 1.0 | Sep/30/2015 | Oct/08/2015 | Initial Version |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*This section (Purpose, Applicability, Instructions, and Revision History) should be deleted from the TLDS for your product.*

**Table of Contents**

[**1.** **Document Control** 3](#_Toc437687719)

[1.1 Document Master Location 3](#_Toc437687720)

[1.2 Document Revision History 3](#_Toc437687721)

[1.3 Group Members 3](#_Toc437687722)

[1.4 Approvers 3](#_Toc437687723)

[1.5 Reviewers 4](#_Toc437687724)

[**2. General Overview** 5](#_Toc437687725)

[2.1 Summary of Capabilities 5](#_Toc437687726)

[2.2 User Stories 5](#_Toc437687727)

[**3. Solution Integrations** 7](#_Toc437687728)

[3.1 Summary of Products 7](#_Toc437687729)

[3.2 Solution Use-Cases 7](#_Toc437687730)

[4.1 Overview 8](#_Toc437687731)

[4.2 System Design 9](#_Toc437687732)

[4.3 Dependencies 14](#_Toc437687733)

[4.4 Environment 14](#_Toc437687734)

[4.5 User Interfaces 14](#_Toc437687735)

[4.6 Integrations/Services Interfaces 14](#_Toc437687736)

[4.7 Data Models 16](#_Toc437687737)

[4.8 Security 16](#_Toc437687738)

[4.9 Extensibility 16](#_Toc437687739)

[4.10 Concurrency 17](#_Toc437687740)

[4.11 Performance 17](#_Toc437687741)

[4.12 Scale 17](#_Toc437687742)

[4.13 High Availability 17](#_Toc437687743)

[4.14 Installation/Deployment/Distribution 18](#_Toc437687744)

[4.15 Configuration and Administration 18](#_Toc437687745)

[**5.** **Challenges** 18](#_Toc437687746)

[**6.** **References** 18](#_Toc437687747)

[**7.**  **TLDS Template Revision History** 19](#_Toc437687748)

# **Document Control**

## **1.1 Document Master Location**

|  |  |
| --- | --- |
| **Filename:** | **Description of changes** |
| Document Location | <Hyperlink and/or document owner contact name> |

## **1.2 Document Revision History**

*The table below contains the summary of changes:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date Changed** | **Completed By** | **Description of changes** |
| 1.0 | Sep/30/2015 | Oct/08/2015 | Initial Version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## **1.3 Group Members**

|  |  |  |
| --- | --- | --- |
| **Name** | **UNI** | **Email** |
| Bonnie Tran | bt2389 | bt2389@columbia.edu |
| Kay Chioma Igwe | kci2104 | kci2104@columbia.edu |
| Qianyuan Chen | qc2200 | qc2200@columbia.edu |
| Thomas Przemysław Huzij | tph2109 | tph2109@columbia.edu |

## **1.4 Approvers**

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Approver Name** | **Approver Title** | **Version** | **Date Approved** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## **1.5 Reviewers**

*The table below contains the record of reviewers:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Reviewer Name** | **Reviewer Title** | **Version** | **Date Reviewed** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# **2. General Overview**

## **2.1 Summary of Capabilities**

The system will be a series of interactive microservices with the ultimate purpose of managing student and course records. This involves performing the traditional CRUD (Create, Read, Update, Delete) operations on students along with the courses they have taken and are currently enrolled in. The system will need to track the courses by room number, course Id, Title, and the enrolled students. The router accepts REST API calls from users and configuration requests from the system administrators. An intermediate microservice will be used to maintain referential integrity between the student and course microservices.

## **2.2 User Stories**

**Create Course**

**Actors/Roles Involved: User**

**Pre-conditions: Not applicable**

**Flow of Events:** User sends request through browser and router routes request to course so that a new line in the database is created. The fields in the row are filled according to the JSON payload sent with the request.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

**Read Course**

**Actors/Roles Involved: User**

**Pre-conditions:** This assumes that a course has already been created in the database. Otherwise the request will return an error message.

**Flow of Events:** The user sends a request through the browser and the router routes the request to course microservice so that the courses are retrieved and rendered to the user as a JSON object.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

**Update Course**

**Actors/Roles Involved: User**

**Pre-conditions:** This assumes that a course has already been created in the database. Otherwise the request will return an error message.

**Flow of Events:** Theuser makes a series of REST API requests to the router which directs the request to the course microservice. The user provides a JSON object with the changes he/she wants to make in the payload of their request and they will always receive a JSON object in the payload of the response.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

**Delete Course**

**Actors/Roles Involved: User**

**Pre-conditions:** This assumes that a course has already been created in the database. Otherwise the request will return an error message.

**Flow of Events:** Theuser makes a series of REST API requests to the router which directs the request to the course microservice. The user specifies in its request the ID of the course they would like to remove and the course from the database will be removed.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

**Create Student**

**Actors/Roles Involved: User**

**Pre-conditions:** Not applicable

**Flow of Events:** Theuser makes a series of REST API requests to the router which directs the request to the student microservice.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

**Read Student**

**Actors/Roles Involved: User**

**Pre-conditions:** This assumes that a student has already been created in the database. Otherwise the request will return an error message.

**Flow of Events:** Theuser makes a series of REST API requests to the router which directs the request to the student microservice. The course(s) are retrieved and rendered as a JSON to the user.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

**Update Student**

**Actors/Roles Involved: User**

**Pre-conditions:** This assumes that a student has already been created in the database. Otherwise the request will return an error message.

**Flow of Events:** Theuser makes a series of REST API requests to the router which directs the request to the student microservice. The user provides a JSON object with the changes he/she wants to make in the payload of their request and they will always receive a JSON object in the payload of the response.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

**Delete Student**

**Actors/Roles Involved: User**

**Pre-conditions:** This assumes that a student has already been created in the database. Otherwise the request will return an error message.

**Flow of Events:** Theuser makes a series of REST API requests to the router which directs the request to the course microservice. The user specifies in its request the ID of the student they would like to remove and the student from the database will be removed.

**Post-conditions:** Any changes made in one microservice that possesses a referential integrity constraint with any other microservice must be reflected immediately and data must not become stale. The referential integrity constraint must prevent updates from occurring to data that has not yet been synchronized with anything it depends on or vice versa.

**Limitations**: The user can only possibly perform commands that are exposed by the REST API.

# **3. Solution Integrations**

## **3.1 Summary of Products**

1. The backend of the system is completely implemented using Ruby on Rails. This framework designed for web applications suits the requirements of the system perfectly because it handles URL parsing and request handling. All the programmer needs to do is define the routes in Rails’ configuration files and describe the logic in Rails’ standardized controllers.
2. Each microservice stores its data model in a MySQL database. Although Ruby on Rails comes preconfigured with SQLite, MySQL better fulfills the capabilities required by the microservices. One shortcoming of SQLite is that it can only perform one write operation to the database at a time. This works perfectly well for something like a standalone web app or mobile app but a microservice that may need to handle multiple requests at once would quickly become a bottleneck with such a constraint. MySQL is industry tested and proven to maintain data integrity and provide high performance.

## **3.2 Solution Use-Cases**

**Description:** The system allows an administrator to manage student and course data. The intricacies of the system and the implementation details should not be exposed to the user. Instead, the system exposes a public REST API that the administrator can use to easily interact with the system. Internally, these simple REST API calls will actually translate to a series of private communications between a router microservice, a student microservice, a course microservice, and a microservice that guarantees referential integrity.

**Products Involved:** The system uses the Ruby on Rails framework for backend communication, exposing interfaces, and communicating with a persistence mechanism. The system’s persistence mechanism uses MySQL relational database management software to easily store the data and achieve high performance.

**Types of Integration:** All public interactions with the system will be facilitated by a REST API exposed to the user. The REST API is a series of uniform resource identifiers (URIs) that provide a simple and standardized method of communicating with the system. Internally, the microservices will maintain private URIs that they use to communicate and forward control and data information.

**Functions:** Ruby on Rails provides a web server that manages HTTP request resolution, database communication, and other overhead in microservice development. MySQL is a relational database management system (RDMS) used to store data that is perfectly suited for the system’s purposes and performance requirements.

**4. Architecture Design**

## **4.1 Overview**

There are a total of four microservices that comprise the system. These microservices can be labeled Router, Students, Courses, and the Referential Integrity Coordinator (RIC). All requests are originally destined for the Router which parses the request and forwards it to the appropriate microservice. The system uses a fourth microservice to manage an integrity constraint so all requests will most likely be forwarded there at first. Then the RIC detects the referential problem and determines what changes should be made, then sends new requests to the Router to redirect to both Students and Courses, in a coordinated manner to maintain the integrity constraint and assure that all the data is consistent and cohesive.

## **4.2 System Design**

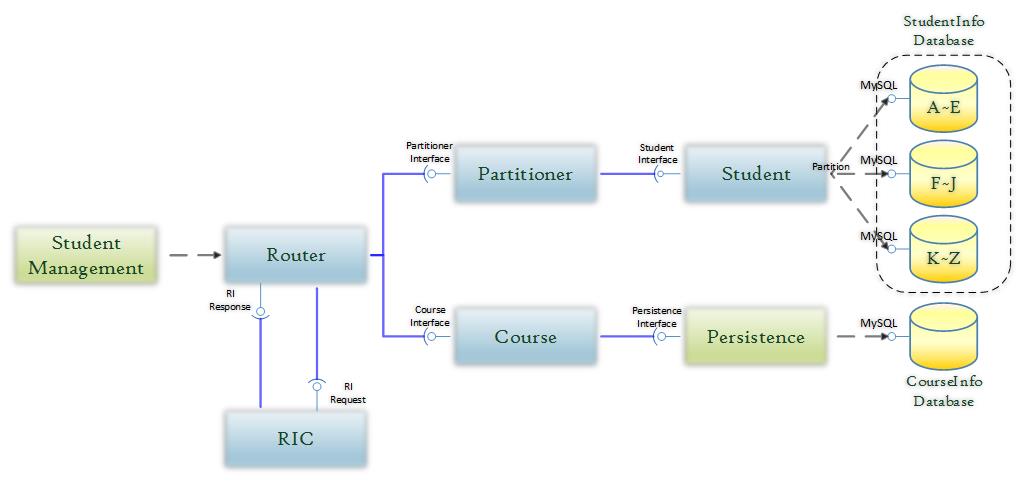


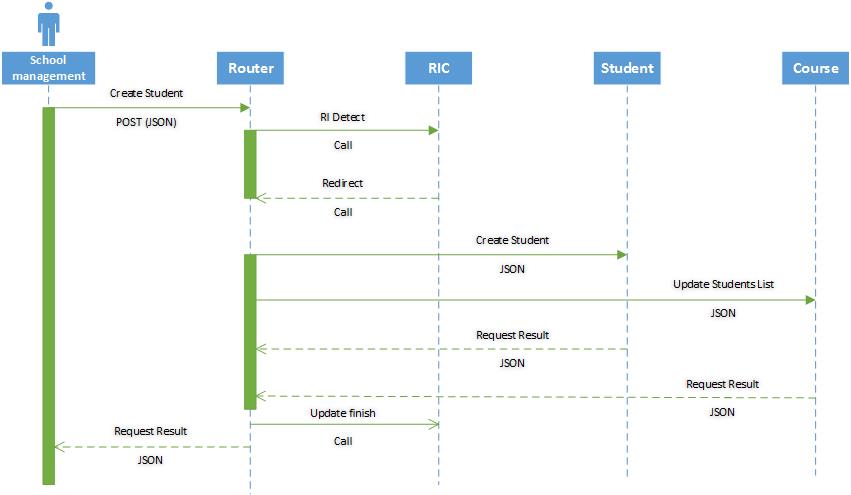
Fig. 1 Component Diagram

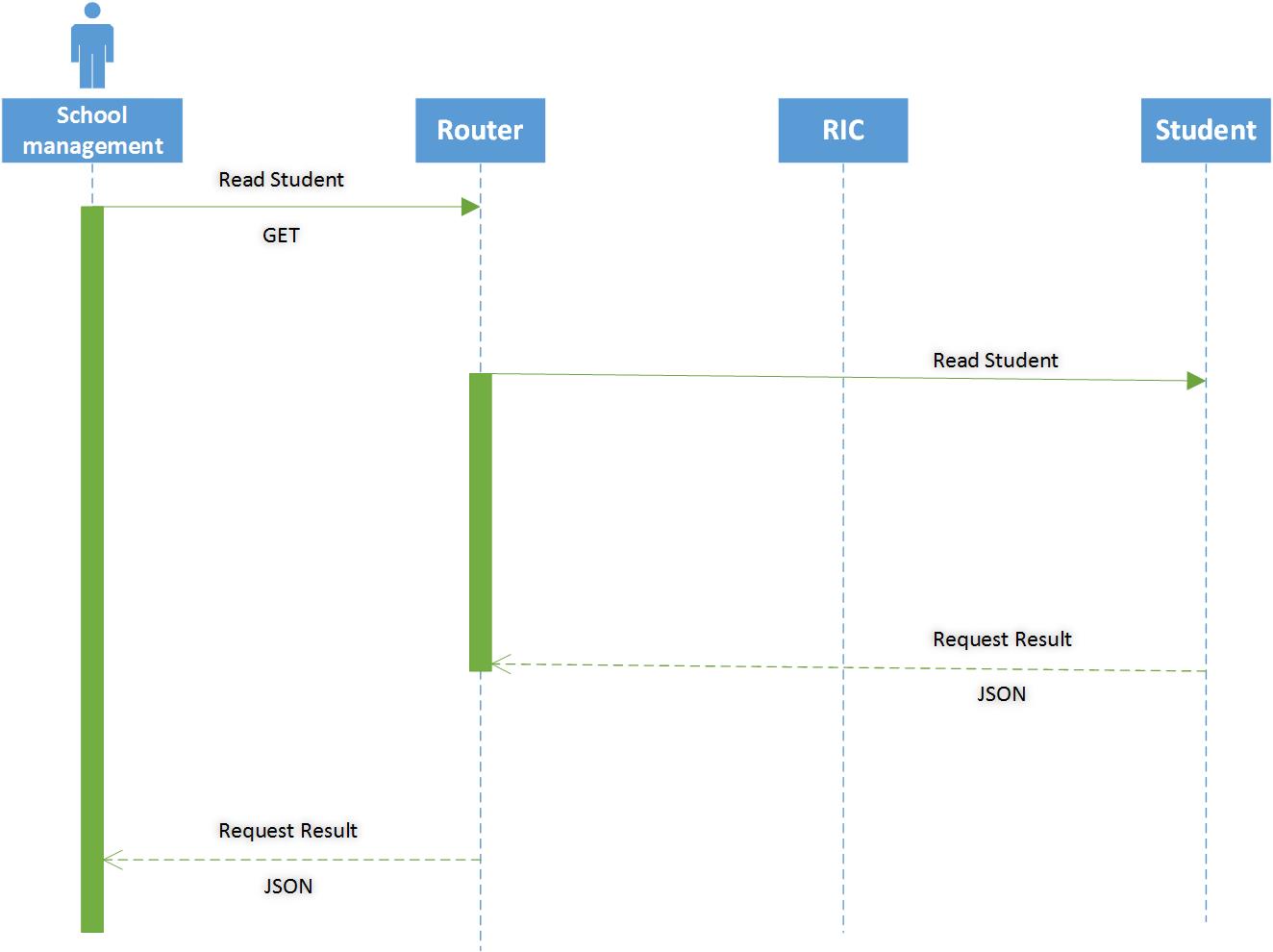
Figure 1 is a component diagram which displays the dependencies and interfaces between different system components. Student Management sources requests to the router. The **router** interfaces with the student and course microservices by sending appropriate commands. The **Referential Integrity Component (RIC)** microservice communicates dependencies to the router and dictates the commands sent by it. Microservices have a way to persist data and this dependency to the MySQL databases is represented by the Persistence component.

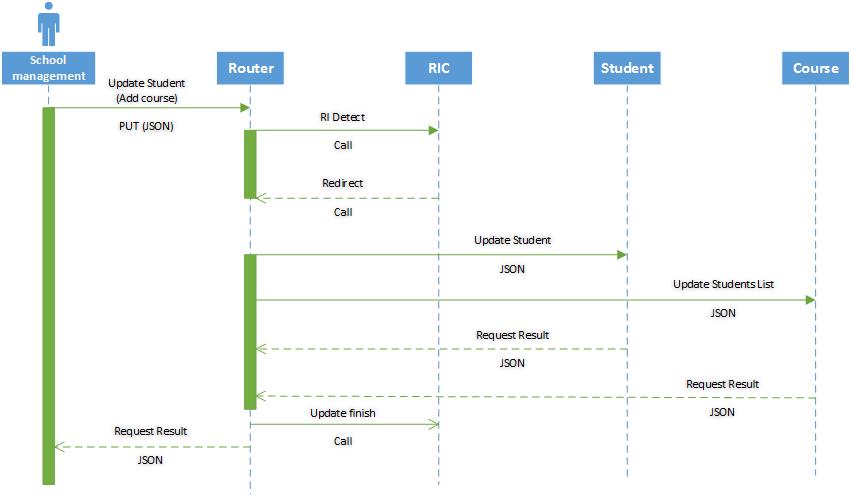
Data is partitioned by last name for the student Microservice. There are three separate groups, students with last name starting with A- E, F- J, and finally K- Z. When a request is called to pull data for a specific student, the Partitioner requests the data from Student, and student GETS from the specific MySQL database, Student does not make any decisions, it executes the Partitioner’s requests. The Partitioner has knowledge of the Student Microservice and decides which database to READ from.

When a DELETE request is issued, the RIC performs two tasks. First it issues a READ, and then, upon retrieval of the correct trigger, it will issue the DELETE. If a DELETE request is issued the RIC sends a READ request to see if the data exists. If it is a DELETE from the Student Microservice, the RIC will issue a request to the Partitioner to READ from a specific Student’s information. The Partitioner will then send a request to READ from a specific database. If the data exists, the Partitioner is able to set a trigger that tells the RIC that the information exists. Similarly, if a course is getting the DELETE request, The Course receives a READ request, and if the information exists, a trigger is set for the RIC to continue. If the RIC receives a trigger that the information does not exist, then nothing happens. However, if the information does exist, a DELETE request is sent to both the Student and the Course Microservice. If there is a DELETE issued by the Course Microservice, a request for to DELETE the Course will be sent to the Course Microservice, and a request to DELETE all fields in the Students Microservice that match the name of the Course deleted. However, if a student is being deleted, then a request to DELETE the student is issued to the Partitioner, while another request is issued to the Course Microservice that will DELETE the field from all Courses containing an exact match to the Students field.

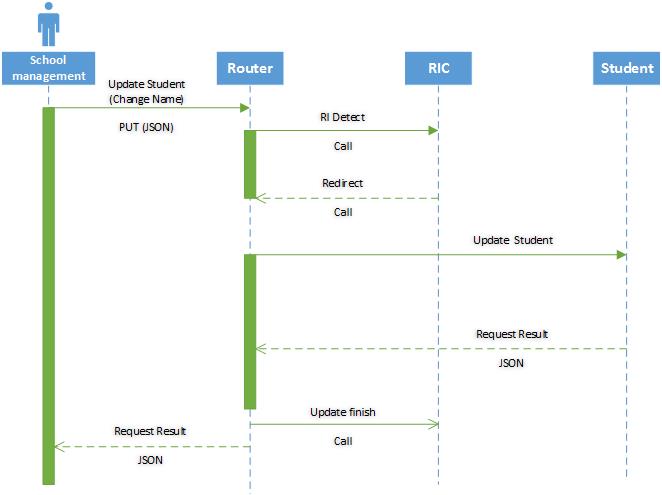
The student data is partitioned to three databases for scalability and increased reliability. In the database, student data is organized based on the first letter of each student’s last name. In order to increase availability of data in the case of a failure, the students data is partitioned into three databases. These databases include, A-E, F-J, and K-Z. The Courses microservice only contains one database called CourseInfo Database. The purpose of this database is to store course data, such as course id, students taking the class, room number, and the course title.











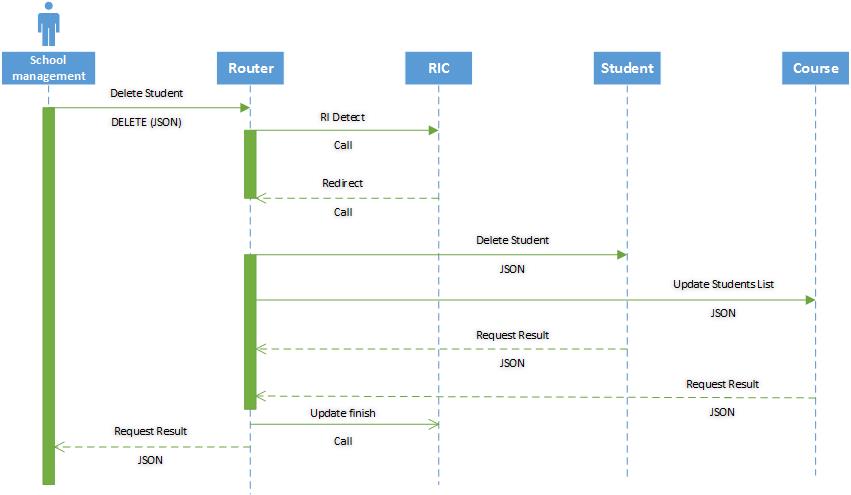


Fig.2 Sequence Diagrams: a) Request from user to create student. b) Request from user to read. student. c) Request from user to update student. d) Request from user to read student. e) Request from user to read student.

Figure 2 visualizes the control flow and data flow for the system. From the user’s perspective, the intricacies of the system and abstracted away and all they need to know is the public REST API that the system exposes. The internal communications of the system are all specified by a private API that only the microservices need to be aware of. Any request by the user must contain a JSON payload and is initially transmitted to the Router microservice. Because of the referential integrity constraint, any request made by the user that updates one of the models must first be passed to another microservice.

The system refers to this microservice as a Referential Integrity Controller (RIC). The RIC dictates which models the Router needs to forward the original user’s request to. This allows the router to only worry about forwarding the requests to the correct microservices (i.e., Students, Courses, or both). Once the models have been updated using their persistence mechanisms, they send the result back to the router with a JSON payload. Before finally sending the response back to the user, the Router sends a message to the RIC to inform it that the models had successfully finished updating their information. The RIC enforces referential integrity by queueing up any new requests and disallowing them from being forwarded to their respective models until the previous update commands have successfully completed. This is akin to holding a lock on a critical section of code and only allowing one party to hold said lock at any given time. This message to the RIC can occur asynchronously and the router does not need to wait to send the JSON payload back the originator of the request.

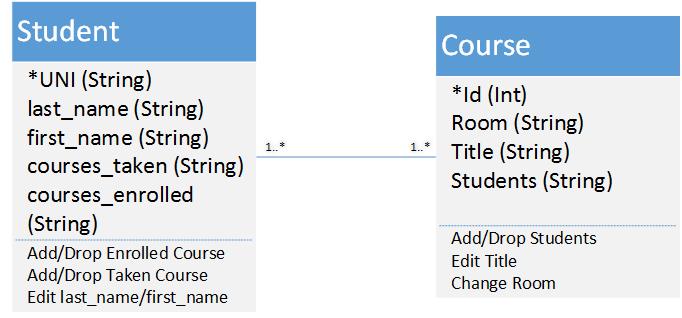


Fig. 3 Class Diagram

Figure 3 contains two classes; Student and Course. The Student class has four parameters and they include, LastName, FirstName, CoursesTaken, and CoursesEnrolled. LastName and FirstName are both strings representing the name of the students, while CoursesTaken and CoursesEnrolled are arrays of integers. CoursesTaken and CoursesEnrolled are both arrays of course IDs. The available methods for the Student class are to drop a taken course, add a taken course, drop an enrolled course, or add an enrolled course.

The course class has four parameters, which include, which include Id, Room, Title, and Students. Id is an integer, which represents the class identification number. Students is an array of integers, representing the student’s identification number. Room is a string that contains both the building name and the room number. Title is also a string that represents the title of the class. The course microservice also contains four methods, which include add students, drop students, edit title, and change room.

## **4.3 Dependencies**

1. Ruby on Rails
2. MySQL

## **4.4 Environment**

1. Ruby 2.0.0-p645
2. Rails 4.2.4

## **4.5 User Interfaces**

The interface exposed to the user is a REST API that allows standard CRUD operations on the two data models (Student and Course). Because of the flexibility of REST APIs, the user can either perform requests to the URIs programmatically in their language of choice, or from a standard command line shell using a standard tool like cURL. The user must be familiar with JSON objects because they are the payload structure that the system uses.

## **4.6 Integrations/Services Interfaces**

The system exposes a REST API to interact with its services. The user can send GET, POST, PUT, and DELETE HTTP requests to manipulate the data models encapsulated by the services. Each one of these requests must be accompanied with a properly formatted JSON object in the payload of the request. The API guarantees that the response will contain a JSON object in the payload as well.

**REST API format:** /model/model\_id/field/field\_id

**Examples:**

“C”: POST /students (Creates new student and returns student JSON)

“R”: GET /students/0 (Returns JSON for Student 0)

GET /students/0/lastname (Returns JSON for Student 0's LastName)

“U”: PUT /students/0/lastname (Updates Student 0's LastName)

PUT /students/0/courses (Updates Student 0’s EnrolledCourses by adding a course # by sending a JSON)

“D”: DELETE /students (Returns JSON for deleted results)

DELETE /students/0 (Returns JSON for deleted student 0)

DELETE/students/0/courses (Returns JSON for Student 0’s deleted course)

---------------------------------------------------------------------------------------------------------

“C”: POST /courses (Creates new course and returns course JSON)

“R”: GET /courses/0 (Returns JSON for Course 0)

GET /courses/0/room (Returns JSON for Course 0's room)

“U”: PUT /courses/0 (Updates Course 0's fields and returns JSON )

PUT /courses/0/students (Updates Course 0’s students by adding student)

“D”: DELETE /courses (Returns JSON for delete results)

DELETE /courses/0 (Returns JSON for deleted course 0)

DELETE/courses/0/students (Returns JSON for Course 0 with a student deleted)

**Configuration API format:** /model/field/type

Include functions to change schema.

“C”: POST /students/schema (In the JSON we create a new field ex. gender/ string)

“R”: GET /students/schema (Returns JSON revealing the current student schema)

“U”: PUT /students/schema/gender (In JSON we create a new field ex. gender: boolean)

/students/schema (In JSON we create all the new fields needed, creating a new database schema for student)

“D”: DELETE /students/schema/gender (Deletes gender column from model and returns JSON of new schema)

The system also implements two methods of organizing data for request and response:

**Method 1)** "Query" arguments:

POST /students?LastName=Ferguson&FirstName=Don&CoursesTaken=3,7,21&CoursesEnrolled=2,5,8

POST /students?LastName=Ferguson&FirstName=Don&CoursesTaken=3,7,21&CoursesEnrolled=2,5,8

**Method 2)** JSON payload:

POST /students with data:

{

"Student": {

"LastName": "Ferguson",

"FirstName": "Don",

"CoursesTaken": [3, 7, 21],

"CoursesEnrolled": [2, 5, 8]

}

}

Both methods will return the newly added object as a JSON object (note that it now possesses an Id):

{

"Student": {

"LastName": "Ferguson",

"FirstName": "Don",

"Id": 0

"CoursesTaken": [3, 7, 21],

"CoursesEnrolled": [2, 5, 8]

}

}

## **4.7 Data Models**

The relational data model allows for the addition of rows or columns and overall modification to be manageable. The primary parts of this data model are the router, student, course and RIC microservice. The student microservice includes student profile information, a unique ID and tracks the number of courses enrolled and taken. Similarly, the course microservice includes course information, a unique ID, and tracks the number of students enrolled in the course. Requests of change the data is sent to the router which commands modification of data in student and course. In cases where the course would need to change as a result of a student change the RIC would ensure that the router sends the appropriate commands.

## **4.8 Security**

There are no security assumptions for the designed products. The user capabilities are not controlled by authentication. Although the API is “public” in the sense that the system exposes an interface with which a user can communicate with the service, the URIs should not be leaked to any unauthorized parties who may wish to access or damage the data.

## **4.9 Extensibility**

No graphical user interfaces will be required but ability to change the data model is.

The system will expose a configuration API that the user can access to modify the data model (e.g. adding gender to the student profile).

## **4.10 Concurrency**

Multi-tenancy is not required for the system.

## **4.11 Performance**

Later versions of Ruby such as 2.0 and its Rails framework includes several features designed primarily to improve performance. The two built-in optimization features, asset pipeline and turbolinks, are inherent in the products created. The asset pipeline feature minimizes the number of requests a browser needs to invoke a web page because it combines individual JavaScript/ CSS files to one JavaScript/CSS file. Web browsers can only make a limited number of requests so the asset pipeline feature allows for faster loading pages. These files also run more quickly because they are compressed by removing whitespace. Additionally, Rails uses manifest files improve performance. These files include directives which includes instructions that indicates what other files to include.

The Turbolinks feature increases process speed by replacing content in the body and title of the current page rather than reloading the entire page.

## **4.12 Scale**

The microservice products are designed to fit different schemes and the partitioning of student data to three databases increases the system’s storage capacity. The microservice products can be scaled in three additional ways. In reference to the scale cube model, it can be scaled along the x, y and z axis. To scale along the x axis multiple copies of the microservice can be made so that each copy handles a proportion of the load. However, to be effective, this will require more caching of memory, more complicated management schemes and with complexity comes increased cost.

This management system can be scaled in the Y direction as the client demand for more varied services increases and the inter-dependencies of these services will be defined accordingly. To support z-axis scaling the data in the database can be partitioned across different servers. This improves cache utilization and lower memory allocation but also increases complexities. The cost of these additions and their inherent complexities will be negotiated with future client needs.

## **4.13 High Availability**

If a database is removed the student microservice will still be able to operate because the data is partitioned to three databases. Under situations of failure an error message will alarm the user and service will be suspended until the bug is handled with.

## **4.14 Installation/Deployment/Distribution**

Given the Ruby on Rails framework’s ease of use, the only special installation requirements are that the dependencies described earlier are strictly maintained to guarantee complete compatibility.

## **4.15 Configuration and Administration**

Administrators are the only users that can make changes to the overall system. Once a user makes a request, the request goes to the router. The router then sends the request to the referential integrity controller (RIC). Based on the request, the router communicates with the router and tells it where to route the request. The router can then handle the request.

**Configuration API format:** /model/field/type

Examples:

“C”: POST /students/gender/string (Creates new gender column and returns JSON revealing new schema)

“R”: GET /students/schema (Returns JSON revealing the current student schema)

“U”: PUT /students/gender/boolean (Changes gender column to boolean and returns JSON of new schema)

“D”: DELETE /students/gender (Deletes gender column from model and returns JSON of new schema)

# **5. Challenges**

Several factors introduced technical challenges for delivering a robust Microservice Project. The challenges faced were, creating a robust referential integrity controller, RIC, deciding how to do partitioning, and with deciding how communication between the RIC and the Router would be done.

Initially the Router was going to do the partitioning depending on the request. If a request was called to READ data, the Router would directly send a request to the appropriate Microservice. This would have meant that the Router would have decision making skills. We knew that the Router could not have these skills and is strictly in place to issues request. In order to account for this, we decided to have the request travel through the Router, but have another Microservice make the decision of where the request should go. In order to do this, we decided to make all request pass through the RIC no matter what the type was. The RIC now makes all of the decisions. Once the RIC makes a decision, it sends a request back up to the Router, where the Router now knows exactly where to send it.

In creating a robust Referential Integrity Controller, RIC, a decision had to be made of which type of errors do we find most important. This is because we could try to watch for errors, such as request format errors, however, it would take too much time to implement. There are too many errors that need to be coded for. Instead the RIC pays close attention to “data does not exist” errors, and does nothing for other types of errors.

Partitioning was another soar spot. In order to do partitioning, it had to be decided which microservice would have decision making capabilities, and whether or not to add an additional Microservice to complete the task. At first the Student Microservice was doing all of the partitioning, however, this added unnecessary complexities to the Student Microservice. The Student Microservice was only suppose to perform the functions of a specific task. However, if an additional Microservice is added, it would create another layer a request had to go through. It was eventually decided that a Partitioner Microservice would be added. The Partitioner Microservice would make the decisions because it had knowledge of the Student Microservice and could issue a request to CREATE, READ, UPDATE, or DELETE, information from a specific database, or all databases, within the Student Microservice, of which has all data partitioned into separate databases.

# **6. References**

|  |
| --- |
| *Lewis, A. (2015). Performance. In Rails crash course: A no-nonsense guide to Rails development. San Francisco, California: No starch press.* |
|  |
|  |
|  |
|  |
|  |
|  |

# **7. TLDS Template Revision History**