***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**](#_Toc432114606)

[1.1 Document Master Location 3](#_Toc432114607)

[1.2 Document Revision History 3](#_Toc432114608)

[1.3 Group Members 3](#_Toc432114609)

[1.4 Approvers 3](#_Toc432114610)

[1.5 Reviewers 4](#_Toc432114611)

[**2. General Overview 5**](#_Toc432114612)

[2.1 Summary of Capabilities 5](#_Toc432114613)

[2.2 User Stories 5](#_Toc432114614)

[**3. Solution Integrations 6**](#_Toc432114615)

[3.1 Summary of Products 6](#_Toc432114616)

[3.2 Solution Use-Cases 6](#_Toc432114617)

[**4. Architecture Design 7**](#_Toc432114618)

[4.1 Overview 7](#_Toc432114619)

[4.2 System Design 7](#_Toc432114620)

[4.3 Dependencies 9](#_Toc432114621)

[4.4 Environment 9](#_Toc432114622)

[4.5 User Interfaces 9](#_Toc432114623)

[4.6 Integrations/Services Interfaces 10](#_Toc432114624)

[4.7 Data Models 11](#_Toc432114625)

[4.8 Security 11](#_Toc432114626)

[4.9 Extensibility 11](#_Toc432114627)

[4.10 Concurrency 11](#_Toc432114628)

[4.11 Performance 11](#_Toc432114629)

[4.12 Scale 12](#_Toc432114630)

[4.13 High Availability 12](#_Toc432114631)

[4.14 Installation/Deployment/Distribution 12](#_Toc432114632)

[4.15 Configuration and Administration 12](#_Toc432114633)

# **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 | Nov./16/2015 | Dec./17/2015 | 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 interactions between the gateway and k-12 microservices for managing queue information. This involves performing the traditional CRUD (Create, Read, Update, Delete) operations on k-12 Microservice by interacting with the corresponding queue, which is watched by the gateway. The system will need to be able to continuously check for updates and update the table of queues accordingly. The gateway is continuously watching for new requests and makes calls if the request is in the table of queues. A k-12 microservice contains student data, and waits for REST API calls from the Gateway Microservice.

## **2.2 User Stories**

**Create Student**

**Actors/Roles Involved:** User

**Pre-conditions:** This assume that a student has not been created in the database. Otherwise, the request will return an error message.

**Flow of Events:** The user sends a request that is stored in a queue of requests. The request passes through a gateway that checks for the request and makes a call to the K-12 microservice to an add a new student to the database. The fields in the row are filled according to the JSON payload sent with the request.

**Post-conditions:** Changes to the database and the configuration table with queues will be reflected immediately.

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

**Read Student**

**Actors/Roles Involved:** User

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

**Flow of Events:** The user sends a request that is stored in a queue of requests. The request passes through a gateway that checks for the request and makes a call to the K-12 microservice to a add a new student to the database. The fields in the row are retrieved and rendered to the user as a JSON.

**Post-conditions:** Changes to the database and the configuration table with queues will be reflected immediately.

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

**Update Student**

**Actors/Roles Involved:** User

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

**Flow of Events:** The user sends a request that is stored in a queue of requests. The request passes through a gateway that checks for the request and makes a call to the K-12 microservice to update the fields for a specific student in the database. The fields of the updated student are rendered to the user as a JSON.

**Post-conditions:** Changes to the database and the configuration table with queues will be reflected immediately.

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

**Delete Student**

**Actors/Roles Involved:** User

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

**Flow of Events:** The user sends a request that is stored in a queue of requests. The request passes through a gateway that checks for the request and makes a call to the K-12 microservice to a delete a student in the database. The deleted object is rendered to the user as a JSON.

**Post-conditions:** Changes to the database and the configuration table with queues will be reflected immediately.

**Limitations:** The user can only 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. Amazon Simple Queue Service is used for queuing data into a table. It is a robust queuing service that is scalable and used for queuing message requests. Because it is scalable, SQS, can handle any volume of data. According to Amazon, SQS runs on the highly- available Amazon servers, which means that the service is always available. This is advantageous for this project’s implementation because SQS will be continuously listening for new requests.
3. The k12 Microservice stores its data model in the DynamoDB database provided by Amazon. DynamoDB is a “fast and flexible” NoSQL database. DynamoDB is scalable and allows for continuous access to data. It is a NoSQL database because it does not rely on relationships between the data. It is a non relational database. This works perfectly well for something like a standalone web app or mobile app and for a microservice that may need to handle multiple requests at once.

## **3.2 Solution Use-Cases**

**Description:** The system allows users to send requests for information about a student’s data in k12. The intricacies of the system and the implementation details should not be exposed to the user or users. Instead, the gateway makes REST API calls that send requests to the k12 microservice. Internally, these simple REST API calls will actually translate to a series of private communications between a gateway microservice and a k12 microservice that fulfill the tasks of the request.

**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 an Amazon DynamoDB non 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 request sent to a queue. The gateway is continuously listening for a request from the queue. Once it receives a request, it issues a REST API call. The REST API is a series of uniform resource locators (URLs) that provide a simple and standardized method of communicating with the system. Internally, the microservices will maintain private URLs that they use to communicate and forward control and data information inside of a table in the gateway.

**Functions:** Ruby on Rails provides a web server that manages HTTP request resolution, database communication, and other overhead in microservice development. DynamoDB is a non- relational database management system 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 two microservices that make up the system. These microservices are labeled k12, and the Gateway. All requests in the request queue are destined for the Gateway which reads the configuration file mapping the URLs to the corresponding queue names. A table is built using this information and each item in the queue is handled. The Gateway generates a HTTP request to the k12 microservice which goes on to complete the desired actions.

## **4.2 System Design**

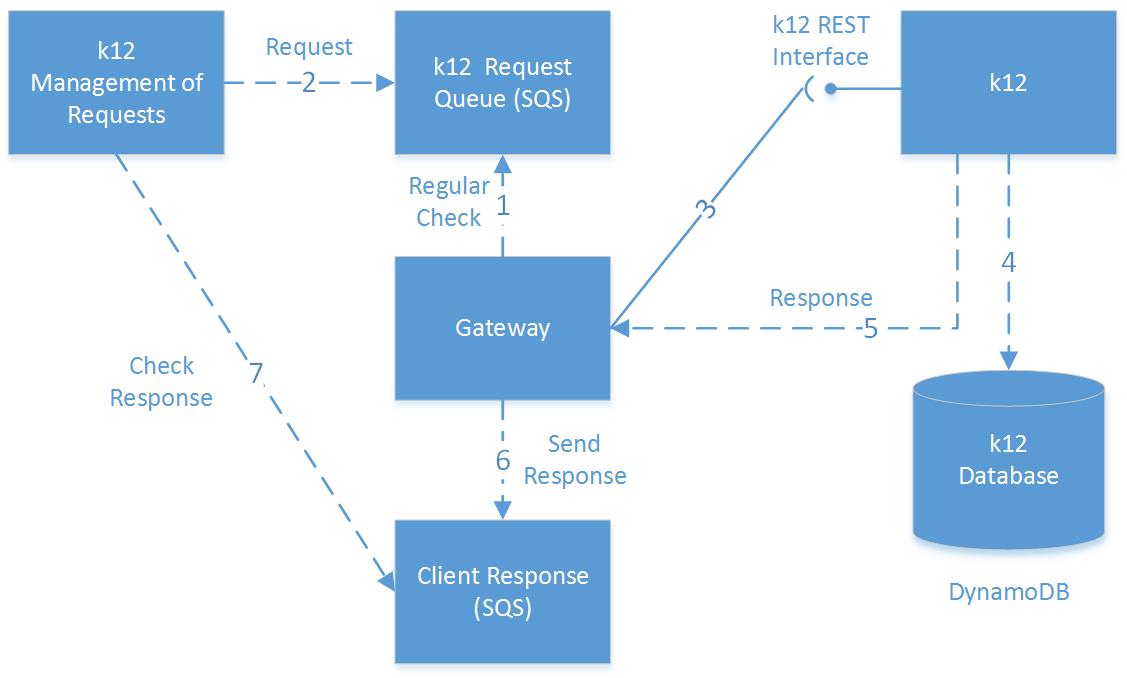


Fig. 2 Component Diagram

Figure 2 is a component diagram which displays the dependencies and interfaces between different system components. The Gateway is always checking request queues regularly, then when the client sends requests to k12 queue, the Gateway reads the request from that queue, and communicates with k12 microservice by sending REST calls. The Gateway reads the configuration file and maps the queue name “k12” to its corresponding microservice URL. A table is built using this information and each item in the queue is handled (In this case the table has just one line recording the k12 microservice). The Gateway generates a REST HTTP request to the k12 microservice which goes on to complete the desired actions. The k12 microservice stores student information in a DynamoDB database and information is removed by request. After finishing tasks from request, k12 microservice returns their response, together with the unique ID for this request, to Gateway, and when Gateway receiving the response, it sends the response back to client’s response queue. Since the client request message contains the response queue name, the Gateway can locate the back queue correctly. Finally, the client can get their response for each request when they checking their own response queue.

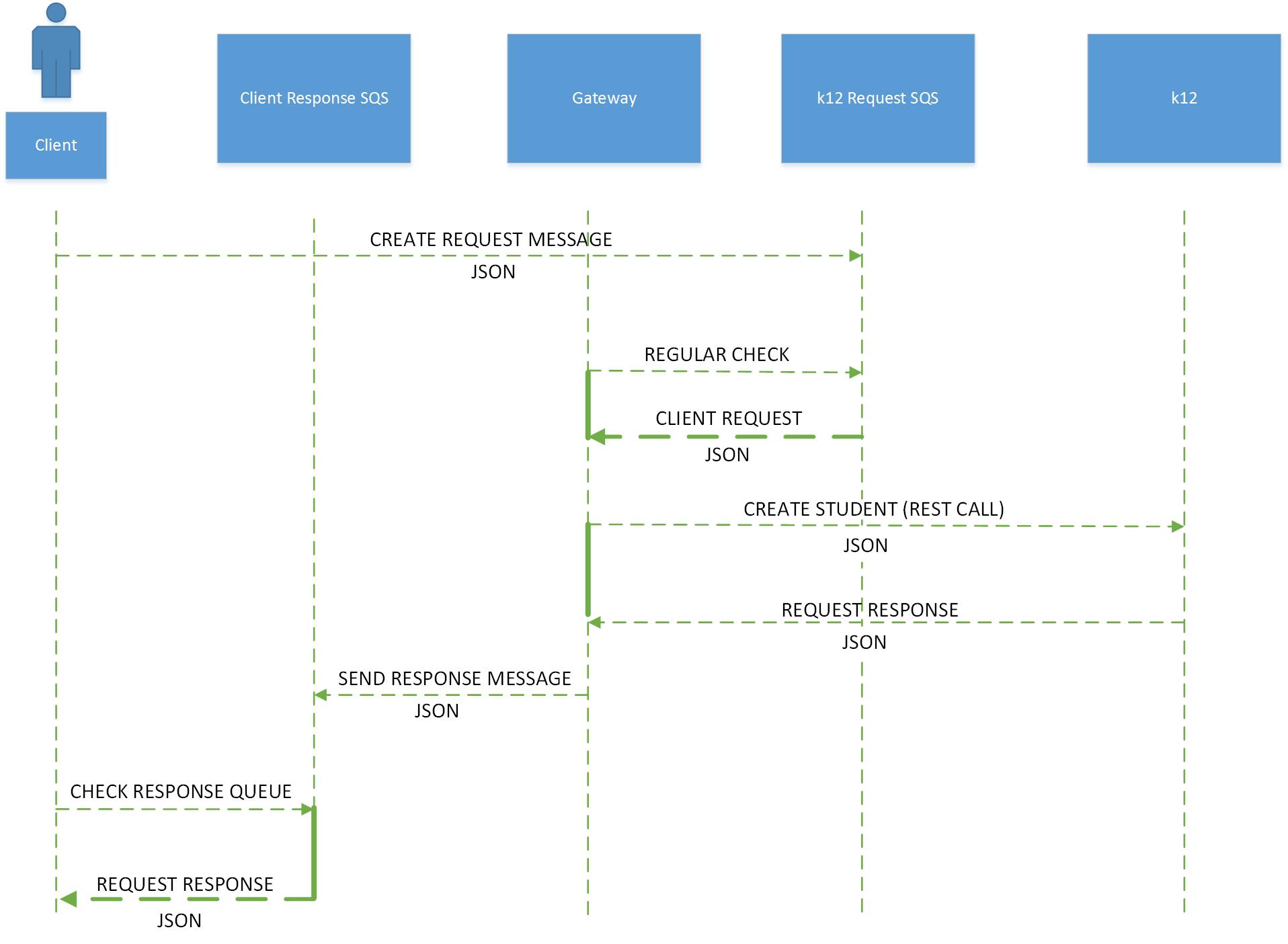


Figure 2: Create Sequence Diagram

Figure 2 represents a visualization of the control flow and data flow for the system. From the user’s perspective, the intricacies of the system are abstracted away exposing the public request queue, which is the only feature needed. The internal communications of the system are all specified by a private REST API that only the microservices need to be aware of.

In the **Create Sequence Diagram** a request to “Create Student” by the user must contain a JSON payload and is initially transmitted to the k12 request SQS. Afterwards a READ request is made by the Gateway microservice to abstract the message information. The Gateway microservice sends the appropriate REST URL to the k12 microservice so the CREATE request is handled. To return the JSON payload to the user a response message is created by the Gateway and sent to the the Client Response SQS. The user reads from the Client Response SQS and a JSON request result is returned.

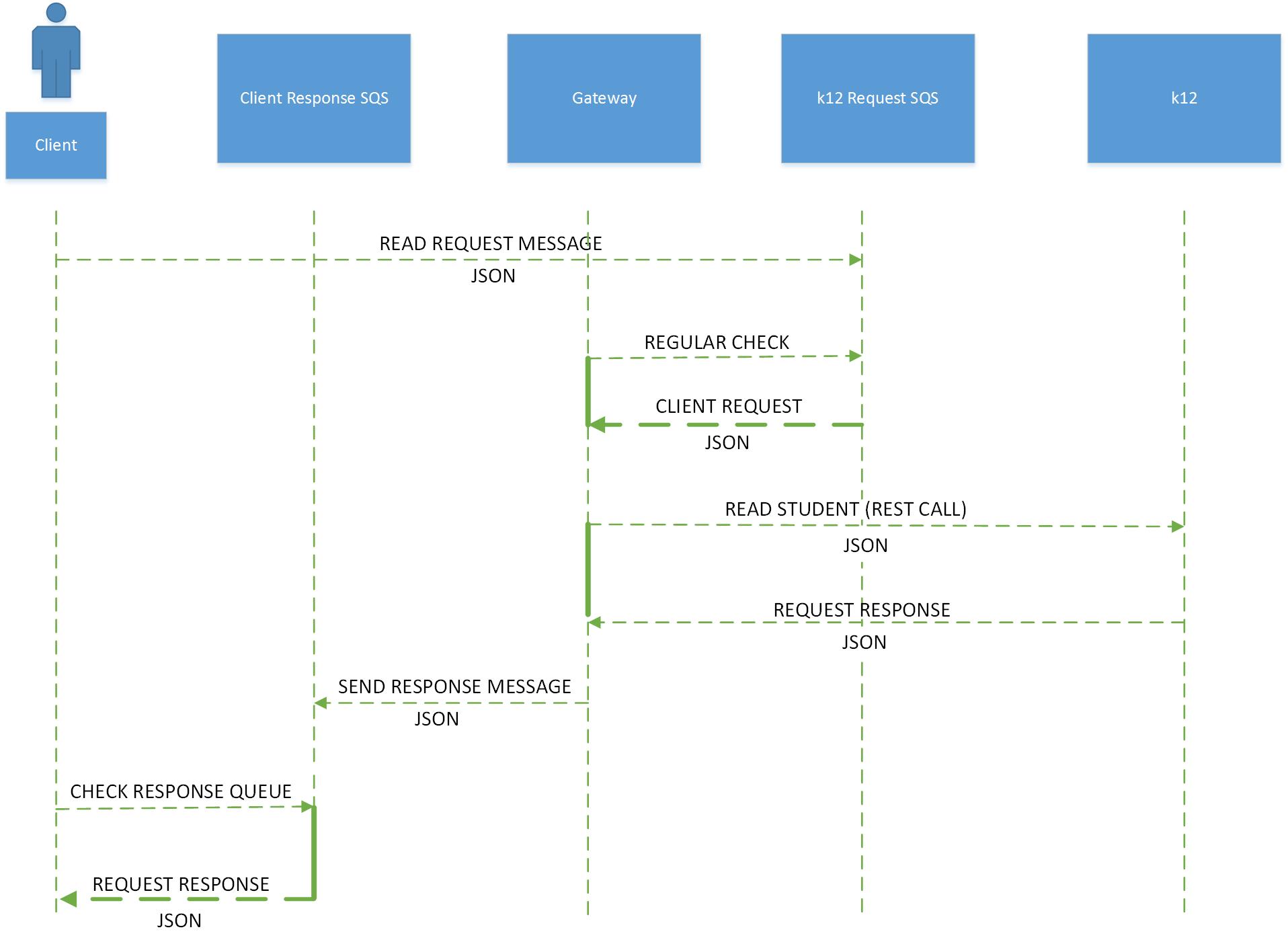


Figure 3: Read Sequence Diagram

Figure 3 shows the Read Sequence Diagram. In the **Read Sequence Diagram,** a request to “Read Student” by the user must contain a JSON payload and is initially transmitted to the k12 request SQS. Afterwards a READ request is made by the Gateway microservice to abstract the message information. The Gateway microservice sends the appropriate URL to the k12 microservice so the READ request is handled. To return the JSON payload to the user a response message is created by the Gateway and sent to the the Client Response SQS. The user reads from the Client Response SQS and a JSON request result is returned.

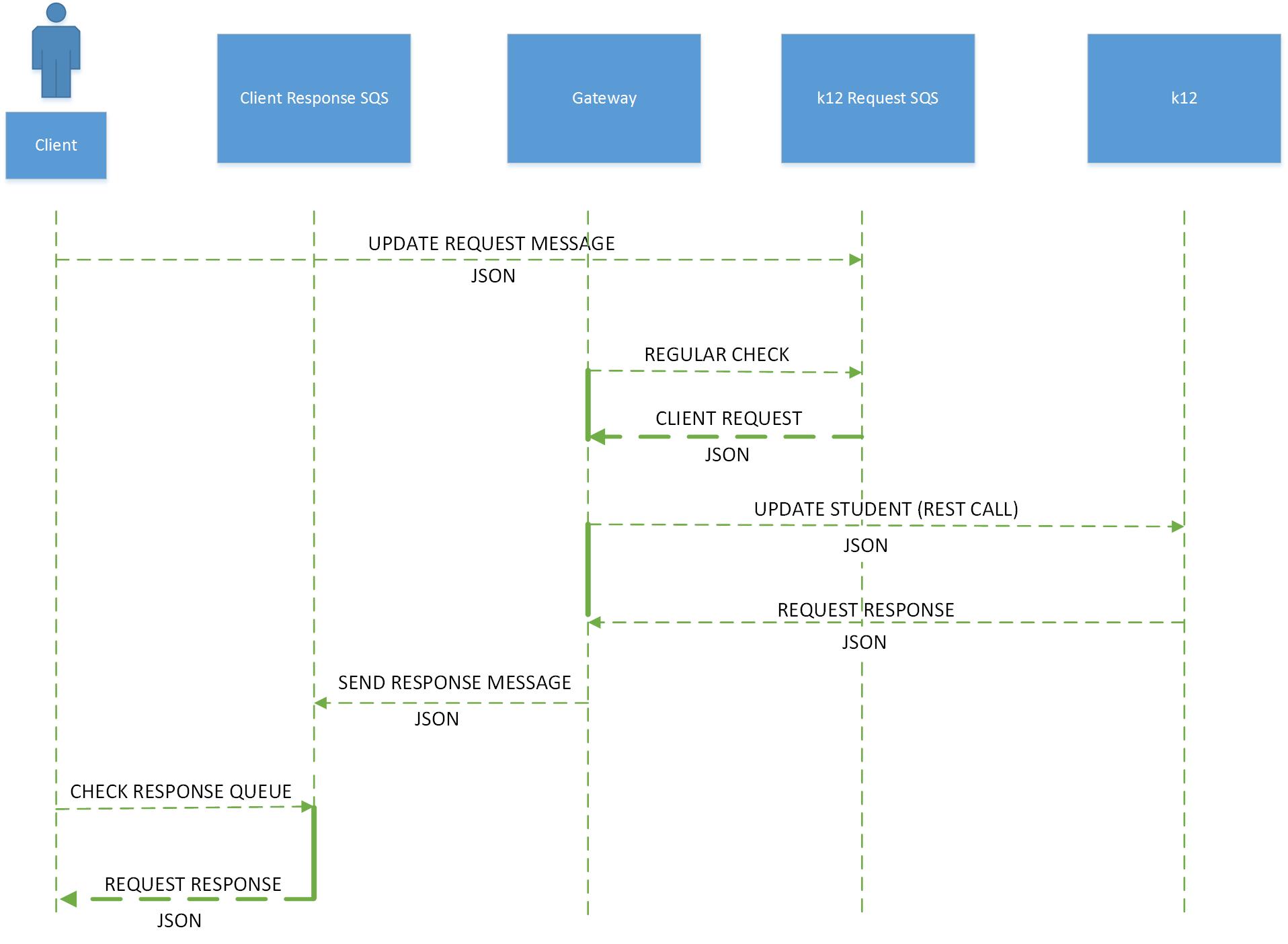


Figure 4: Update Sequence Diagram

Figure 4 shows the **Update Sequence Diagram,** a request to “Update Student” by the user must contain a JSON payload and is initially transmitted to the k12 request SQS. Afterwards a READ request is made by the Gateway microservice to abstract the message information. The Gateway microservice sends the appropriate URL to the k12 microservice so the UPDATE request is handled. To return the JSON payload to the user a response message is created by the Gateway and sent to the the Client Response SQS. The user reads from the Client Response SQS and a JSON request result is returned.

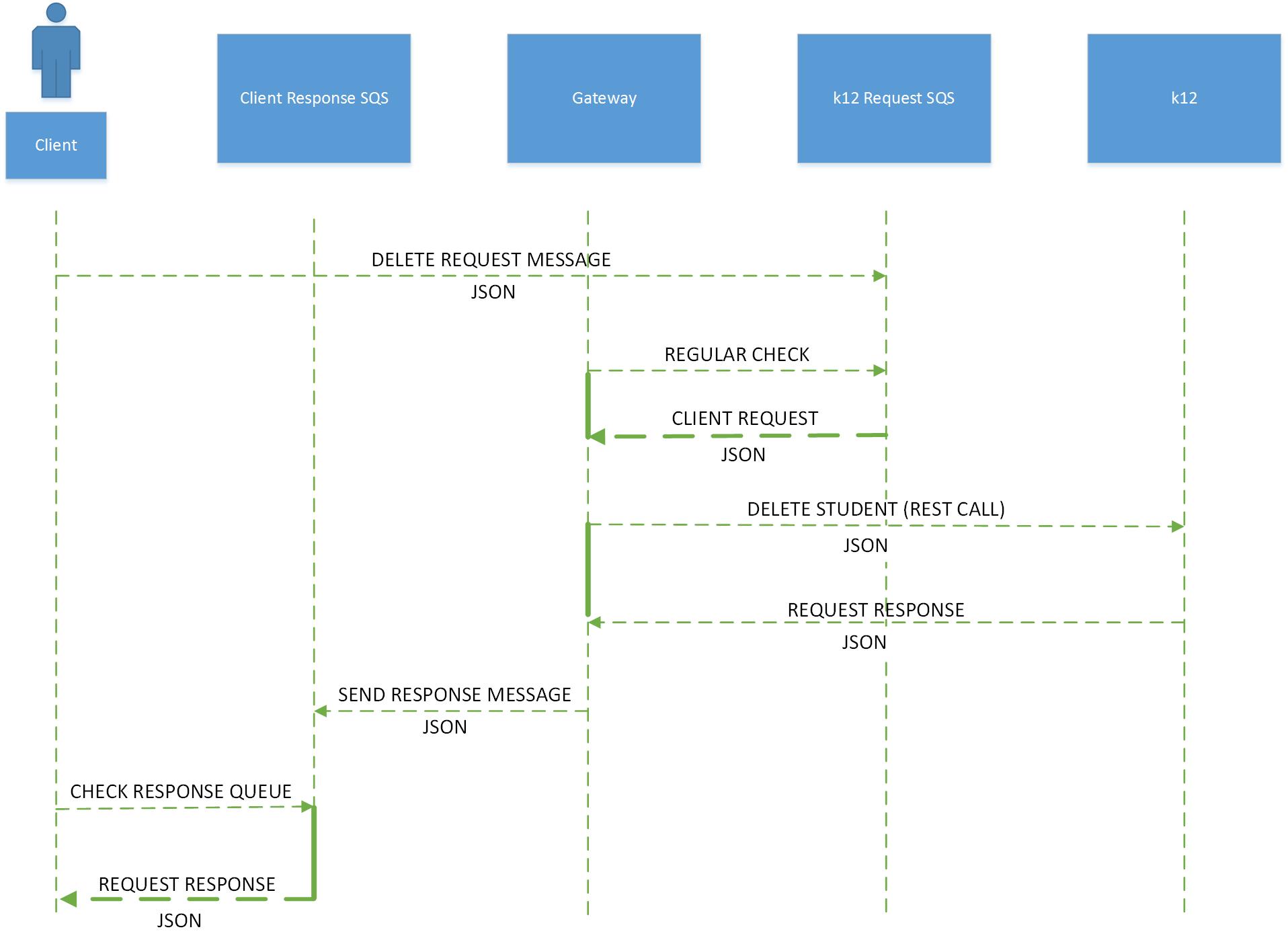


Figure 5: Delete Sequence Diagram

Figure 5 shows the **Delete Sequence Diagram,** a request to “Delete Student” by the user must contain a JSON payload and is initially transmitted to the k12 request SQS. Afterwards a READ request is made by the Gateway microservice to abstract the message information. The Gateway microservice sends the appropriate URL to the k12 microservice so the DELETE request is handled. To return the JSON payload to the user a response message is created by the Gateway and sent to the the Client Response SQS. The user reads from the Client Response SQS and a JSON request result is returned.

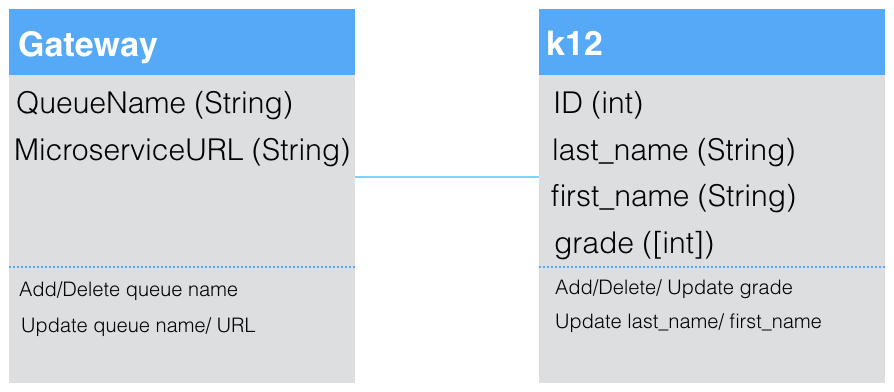


Figure 6: Class Diagram

Figure 6 contains two classes; Gateway and k12. The Gateway Microservice is the queueing mechanism. It takes in a log and puts the data in the log into a table. It contains a Queue name and a microservice URL. The Queue name serves as the queue identifier, each of which contain a unique Microservice URL. The Queue name and the Microservice URL are strings. A queue name can be added to the queue as deleted from the Gateway. It can also be added to the Gateway. The queue name and the Microservice URL can also be edited or updated.

k12 contains an ID, a last\_name, a first\_name, and a grade. ID is the uni of a student and is an integer. Last\_name is a string which represents the first name of a student. The grade array is an integer array that contains all of the grades of a student. In k12 a student can be added or dropped from the service. The student’s last name or first name can also be updated. Grades can be added or deleted from a student’s record. They can also be updated by the user. The ID serves as the identifier, and each record maintains a unique ID.

## **4.3 Dependencies**

1. Ruby on Rails
2. MySQL

## **4.4 Environment**

1. Ruby 2.0.0-p645
2. Amazon SQS
3. DynamoDB

## **4.5 User Interfaces**

The interface exposed to the user is a request queue that allows formatted request messages that can send standard CRUD operations on the k12 microservice which contains a database with student information. 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 way of user calling the microservice is quite easy by using Amazon SQS, user just need to send their request to the corresponding queue of the microservice (K12), following the format below:

Request format:

Queue\_name, Cid, Res\_queue, Option, Purpose, Body

Parameters:

queue\_name is the name of the queue which receives this request

Cid is a unique id related to this request

Res\_queue is the user’s queue for receiving the response

Option is the verb used for this call, namely, “GET, POST, PUT, DELETE”

Purpose is the jobs expected for this request

Body is the JSON payload for this request

Examples:

K12, dhskjhqwui112, response\_Toms, GET, /k12/, {}

K12, qcwi392892cee, response\_Roy, POST, /k12/grade, {“grade”:”3.75”}

The k12 microservice has a standard REST API:

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

Examples:

“C”: POST /k12 (Creates new student and returns k12 JSON)

“R”: GET /k12/0 (Returns JSON for k12 with id 0)

      GET /k12/3/lastname (Returns JSON for k12 with id 3's LastName)

“U”: PUT /k12/2/firstname (Updates Student 2's FirstName)

“D”: DELETE /k12/0 (Returns JSON for delete result)

JSON payload:

POST /k12 with data:

{

"k12": {

  "id": 0,

  "lastname": "Ferguson",

  "firstname": "Don",

  "gender": "male",

  "grade": 9

 }

}

This will return the newly added object as a JSON object:

{

  "lastname": "Ferguson",

  "firstname": "Don",

  "id": 0,

  "gender": "male",

  "grade": 9

}

ruby add\_queue\_message.rb K12 weewrf POST / {\"id\":4\,\"name\":\"roy\"}

ruby get\_queue\_message.rb

Example:

# ruby add\_queue\_message.rb K12 weewrf POST / {\"id\":3\,\"name\":\"tom\"}

{"id":3,"name":"tom"}

Sending message '{

    "header": {

        "op": "POST",

        "uid": "7",

        "response\_queue": "response\_tom",

        "url": "/",

        "cid": "weewrf"

    },

    "body": {"id":3,"name":"tom"}

}' to queue ...

A message is created on queue with id 'd2a6ea55-7c87-4931-8e4b-b4f3f28edf75'

# ruby get\_queue\_message.rb

Retrieving the first message from queue ...

## 

## **4.7 Data Models**

The non relational data model allows for the storage and retrieval of data outside of a relational context. This creates database flexibility, which means that any type of data can be added to an attribute. The primary parts of this data model are the queue, the gateway, and the k12 microservice. The k12 microservice includes last\_name, first\_name, grade, and an id as the primary key. The gateway consists of a table with 2 columns, one for the queue name, and the other for the microservice URL. Requests to change the data are sent to the queue, where the gateway is constantly listening for requests. Then upon a request, the gateway sends an HTTP request to the k12 Microservice, for which data is retrieved and sent to the gateway that reroutes to the user.

## **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 URLs should be available to all 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.

*Source:*

*Chapter 12 - Performance*

*Rails Crash Course: A No-Nonsense Guide to Rails Development*

*by* [*Anthony Lewis*](http://library.books24x7.com.ezproxy.cul.columbia.edu/SearchResults.aspx?qdom=author&scol=%7Ball%7D&qstr=Anthony%20Lewis)

## **4.12 Scale**

The microservice products are designed to fit different schemes and the partitioning of k12 data, which is handled by DynamoDB. 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.

[*http://microservices.io/articles/scalecube.html*](http://microservices.io/articles/scalecube.html)

## **4.13 High Availability**

If a database is removed the k12 microservice it will still be able to operate because the data is automatically partitioned by Amazon DynamoDB. 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 Amazon SQS, Amazon DynamoDB, and 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**

All users can make changes to the overall system. Once a user makes a request to SQS, the Gateway Microservice will take the request and do an HTTP call to the k12 Microservice. The sequence of how requests are transferred carries no dependencies on the type of request. The sequence from SQS to Gateway to k12, and then back to Gateway to the user, is always followed.

DynamoDB does this automatically. Every time we create a new instance for the database, you have to define in JSON object with the fields desired, and those fields will make up the object.

# **5. Challenges**

There were a few challenges faced while building this project. One of them is that each user has to set up their own SQS queue for receiving a response. The preferred method of completing this action is for it to be done autonomously. This is also the same for the Microservice. In order to add a user to the k12 microservice, it has to be done manually. Currently there is no way or easy way to autonomously add a user to the microservice. In the future, we would like to streamline this process making it automatic, so that a record could be added autonomously, which also means that the Microservice would get things autonomously.