B.Comp. Dissertation

Preparing the Backend of a Large-scale Cloud Application for a Million Users

By

Xiao Pu

Department of Computer Science

School of Computing

National University of Singapore

2018/2019

B.Comp. Dissertation

Preparing the Backend of a Large-scale Cloud Application for a Million Users

By

Xiao Pu

Department of Computer Science

School of Computing

National University of Singapore

2018/2019

Project No: H1221130

Advisor: Associate Professor Damith C. Rajapakse

Deliverables:

Report: 1 Volume

Abstract

TEAMMATES is a large-scale cloud application for instructors to manage peer feedback among students. It is being developed by a project based in School of Computing (SoC), National University of Singapore (NUS). Since its public release in 2011, it has been used by over 350,000 users from all over the world. This project explored ways to prepare TEAMMATES backend to be able to handle its increasing user base, specifically with regard to performance, scalability and maintainability. Based on the findings, three major components were enhanced in significant ways. The improvements in scalability and performance were proven both theoretically and empirically. While most of the tasks in this project were development-based, there were also some research components involved.

Subject Descriptors:

- **General and reference~Performance**
- **Information systems~Query optimization**
- **Information systems~RESTful web services**
- Information systems~Database design and models
- **Computer systems organization~Maintainability and maintenance**

Keywords:

Google App Engine, CRUD operations, batch operations, database cursor, representational state transfer

Acknowledgement

I would like to express my sincere thanks to A/Prof. Damith C. Rajapakse for his guidance, trust, and supervision throughout the project. I would also like to thank Wilson Kurniawan, the project mentor of TEAMMATES, for his timely and helpful code reviews despite having a full-time job. Last but not least, part of the project would not have been possible without the help and support from senior developer Darren Wee, my mentees Rahul Rajesh, Jeremy Choo, Jiang Chunhui and Wang Junming. Thank you for your greatest contributions. I hope I have been a good mentor. I would like to express my best wishes to all of my mentees and hope you find the mentorships beneficial in your software engineering journey.

Table of Contents

1. Introduction

TEAMMATES (https://teammatesv4.appspot.com/) is an online peer evaluation platform for instructors to manage peer feedback among students primarily.

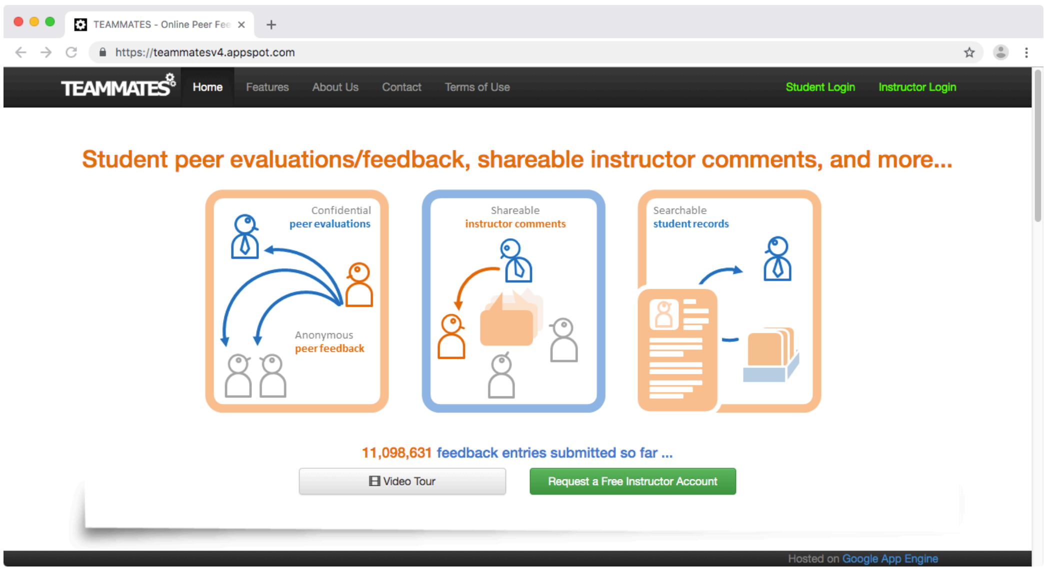

TEAMMATES - Online Peer Fee
https://teammatesv4.appspot.com
TEAMMATES
Home
Features
About Us
Contact
Terms of Use
Student Login
Instructor Login
Student peer evaluations/feedback, shareable instructor comments, and more...
Confidential peer evaluations
Anonymous peer feedback
Shareable instructor comments
Searchable student records
11,098,631 feedback entries submitted so far ...
Video Tour
Request a Free Instructor Account
Hosted on Google App Engine

Figure 1 – Main functionalities of TEAMMATES

The TEAMMATES user workflow is as follows:

1. An instructor creates a *course* in TEAMMATES and enrol students in the course.
2. The instructor creates a *feedback session* and adds different types of questions (e.g., essay questions, rubric questions, and multiple choice questions) for students to answer about their peers.
3. When the session opens for submission, students receive an email containing a unique link to submit responses.
4. Students submit their responses before the deadline of the feedback session.
5. The instructor views the feedback collected. Optionally, the instructor can publish the responses so that students can view feedback they have received from peers.

As of April 2019, TEAMMATES has been used by more than 350,000 users coming from over 800 universities, spanning 96 countries.

TEAMMATES is being developed as an open-source project (https://github.com/TEAMMATES/teammates) based in School of Computing (SoC), National University of Singapore (NUS). It is a project managed by SoC students and has received contributions from over 500 developers since 2010. The core team members are mainly NUS students while the contributors are from all over the world. Two main sources of contributors are from Google Summer of Code program[1] and the module CS3282 Thematic Systems Project.

Based on the current growth rate, TEAMMATES is expected to pass the 0.5 million mark and start moving toward 1 million users within the two years. A natural question to ask is whether the application is ready to serve a huge amount of users. More specifically, it is essential to know whether there are any aspects that can be improved to prepare TEAMMATES for a million users. This is the primary motivation of this Final Year Project (FYP).

There are multiple aspects of the application that can be improved for a million users. For example, a new feature could be added to improve the usability of the application. Alternatively, enhancements could be done to improve the reliability of an existing feature. However, there have been a number of past projects focusing on adding or enhancing features. Hence, this FYP project focuses on less discernible but more significant aspects — performance, scalability, and maintainability — to prepare the application for a million users. Although the effects of the aforementioned factors are less discernible in the short term, they will undoubtedly have significant impacts when more users use the system. Given below is a brief explanation of the three aspects, in the context of TEAMMATES:

- **Aspect 1: Performance:** It is important that the performance of the application should not degrade despite the application's growth. This is because the Google App Engine (GAE), the PaaS[2] platform on which TEAMMATES is running, requires every request to complete within 60 seconds (Google, How Requests are Handled, 2019). Requests that take more than 60 seconds will be killed by GAE, resulting in an unpleasant user experience and possibly causing the loss of some data. In fact, there have been reports from users complaining that the application becomes slower when the size of a course increase.

[1] Google Summer of Code (GSoC) is an international program initialized by Google to encourage more students to contribute to open-source projects. TEAMMATES participates in GSoC for 5 consecutive years from 2014 to 2018.
[2] PaaS: Platform as a Service

- **Aspect 2: Scalability:** The application should be able to handle a large amount of data. While the GAE's infrastructures claim to have the ability to auto-scale, such capabilities are not fool-proof. For example, it is not wise to load data that exceeds the maximum memory size in a single server. Such operation will result in out of memory errors.

- **Aspect 3: Maintainability:** The large-scale project should also be maintainable. Technical debts[3] and outdated designs should not become obstacles in the way of development. Any inappropriate designs and hacks[4] in the existing codebase should be eliminated as far as possible for better maintainability.

This FYP project has managed to address all three aspects. In doing so, it also has achieved the following three major achievements.

1. **Achievement 1: Improved Maintainability and Performance of the Storage Component.** The storage component of the system has been enhanced to improve its maintainability and performance. As a result, its APIs (Application Programming Interface) have become more standardized and the latency of database operations has reduced significantly. Chapter 3 gives more details of this achievement.
2. **Achievement 2: Improved Scalability and Performance of the Client Package.** The client package which is often used to batch-process large amounts of data (e.g., migrate all data into a new format) was identified as having low scalability. It was redesigned to improve its scalability and performance. By applying various database optimization techniques, the scripts were made more scalable so that they can perform reasonably even when TEAMMATES has a million users. Chapter 4 illustrates this achievement in detail.
3. **Achievement 3: Improved Maintainability and Performance of the API Endpoints.** The API endpoints of the backend have been migrated to follow the REST principles. Correspondingly, the uniformness, conciseness and self-descriptiveness of the

[3] Technical debt indicates the cost of additional rework caused by choosing a working solution instead of a better solution due to time constraints when a task is implemented.
[4] Hacks refer to the code that is designed to handle special cases. These hacks are usually hard to understand without knowing the whole context.

endpoints have been improved. In addition, the performance has also been improved noticeably. The details are given in Chapter 5.

In addition to the above three main achievements, I also contributed to the project in several other ways such as project management, bug fixing, and code review. I am also served as the project lead during the latter part of my project and helped to manage (and mentor) 10-15 new developers.

The remainder of this report is organized as follows. Chapter 2 explains the essential technical design and tool stacks used in the projects in lieu of a literature review[5]. Chapter 3, 4 and 5, as mentioned above, discuss the three main achievements of this project. In each chapter, a background that is specific to the achievement is given, followed by defining the problem addressed. After that, the proposed solutions are discussed and the improvements in term of the mentioned three aspects (i.e., performance, scalability, and maintainability) are shown. Lastly, some suggestions for future works are included if there are rooms for future improvements. Chapter 6 presents some miscellaneous tasks that were done. The report ends with a conclusion drawn in Chapter 7 where a brief summary of results and final thoughts are presented.

[5] This FYP project is a development project rather than a research project.

2. Background

TEAMMATES is a cloud-based application running on Google App Engine (GAE), which is a Platform as a Service (PaaS). GAE provides the basic infrastructures for the application such as database systems and Hypertext Transfer Protocol (HTTP) servers. In addition, it also scales applications and balances traffics automatically. As GAE handles the low-level details, developers could focus on high-level logic and interactions.

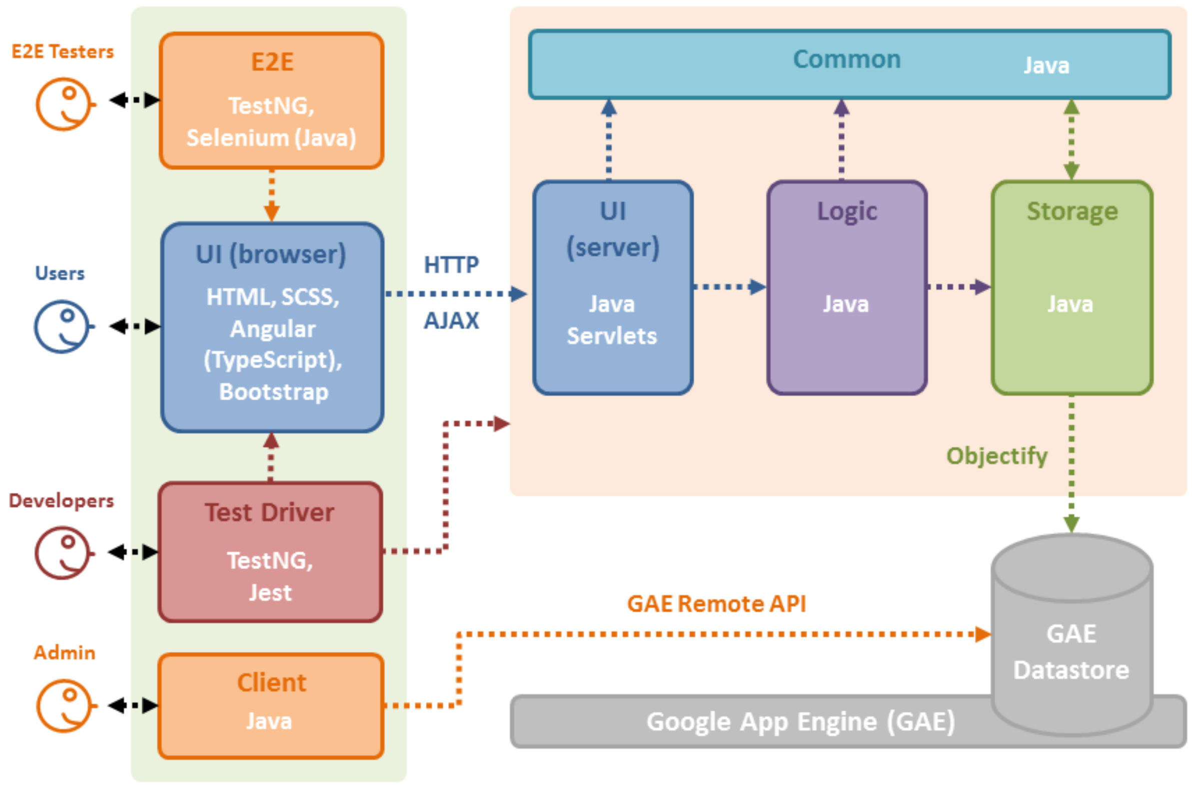


Figure 2 – Overview of the Architecture of TEAMMATES. Source: TEAMMATES developer guides

TEAMMATES is a large-scale project designed using a *layered* architecture, as shown in Figure 2.

The following shows the five distinct layers of the application.

- **UI (User Interface) (browser)** is responsible for rendering HTML pages based on the data passed by **UI (server).** The layer is usually called as the frontend. The rendering logic is built based on the Angular framework.
- **UI (server)** opens endpoints in the server for the frontend to retrieve and manipulate data. The layer is usually called as the backend. Responses from the backend are sent over HTTP using JavaScript Object Notation (JSON).

- **Logic** contains the core business logic.
- **Storage** is responsible for conducting create, read, update, and delete operations (CRUD).
- **Common** contains utility classes that will be shared among the application.

The following shows how different users of TEAMMATES interacts with it:

- **A typical user** would interact with the application using browsers. The frontend will send Asynchronous JavaScript And XML (AJAX) requests to fetch necessary data to generate HTML pages.
- **A developer** would use the test driver to test the application during the development process.
- **An End-to-End (E2E) tester** would test the system as a black-box to verify the correct behaviors of the application.
- **A project admin (manager)** would use the client package to directly connect to the database to perform administrative tasks. This is done by using the GAE Remote API.

3. Achievement 1: Improved Maintainability and Performance of the Storage Component

The background of this chapter will be given first, followed by several sub-sections in the fashion of problem, solution and results.

3.1. Background

TEAMMATES uses GAE Datastore as a NoSQL document database to store application data. The employment of such database eliminates any concerns of performance, scalability, and maintenance on the database side. According to Google (Google Cloud Datastore, 2019), the Datastore is a distributed database built for high performance, high availability and high storage capacity. Data is replicated and distributed geographically in Google's infrastructures. Objects with properties are stored as entities with a particular *kind* (type). A stored object is called an entity. An entity can be accessed by using a key associated with that entity or by querying the properties of that entity.

TEAMMATES's data model is designed to follow the characteristics of the Datastore. Figure 3 shows the logical data model. There are many cross-references between entity types. For example, there is a hierarchy structure from `Course` to `FeedbackSession`, from `FeedbackSession` to `FeedbackQuestion`, from `FeedbackQuestion` to `FeedbackResponse` and from `FeedbackReponse` to `FeedbackReponseComment`. In each entity type, the references of its *ancestors*[6] are stored so that an entity can easily locate its ancestors without fetching its parent. This is an optimization done in the early stage of TEAMMATES since Datastore does not support join operation.

[6] Ancestors are the parent together with all the ancestors of the parent.

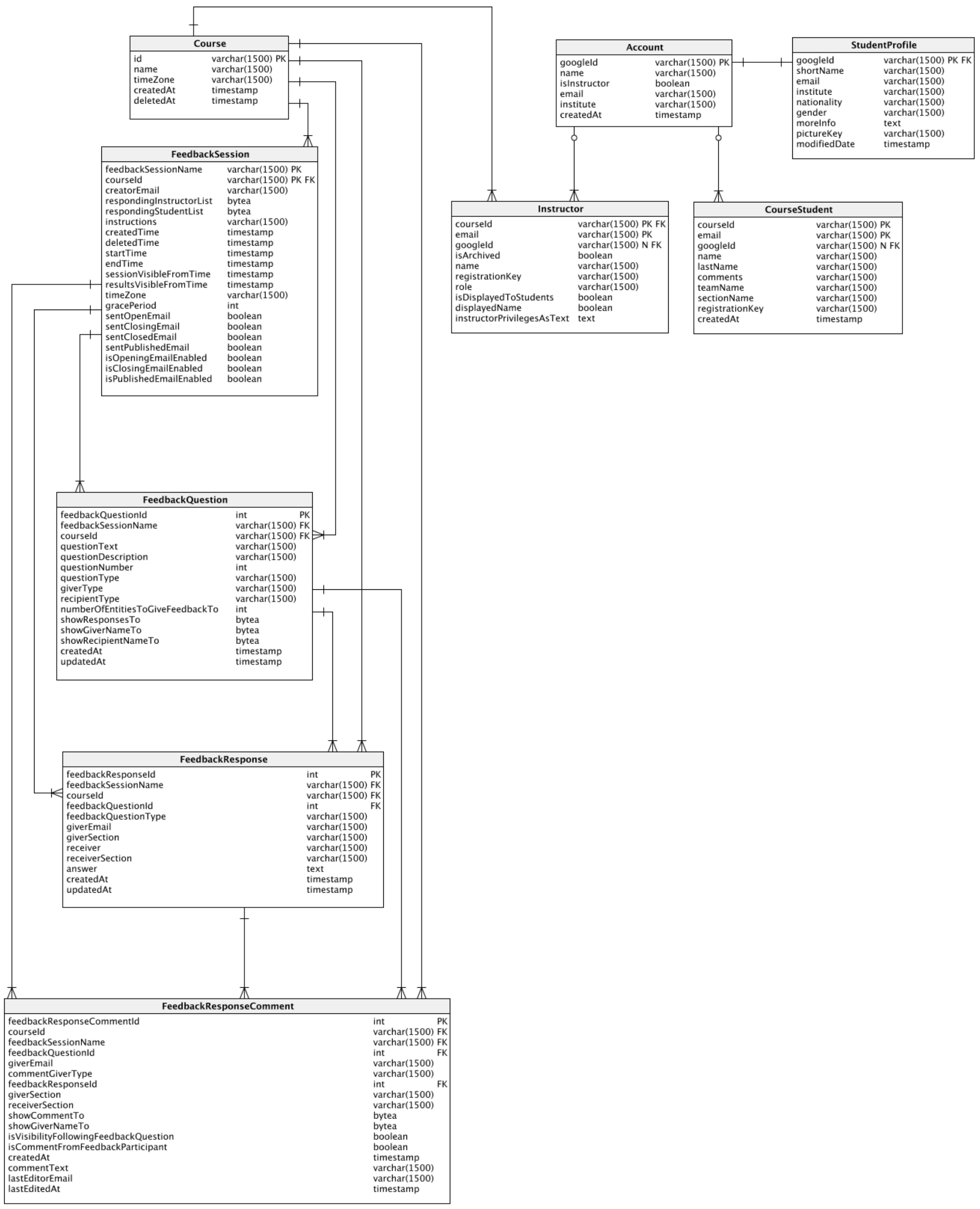

Course
id varchar(1500) PK
name varchar(1500)
timeZone varchar(1500)
createdAt timestamp
deletedAt timestamp
Account
googleId varchar(1500) PK
name varchar(1500)
isInstructor boolean
email varchar(1500)
institute varchar(1500)
createdAt timestamp
StudentProfile
googleId varchar(1500) PK FK
shortName varchar(1500)
email varchar(1500)
institute varchar(1500)
nationality varchar(1500)
gender varchar(1500)
moreInfo text
pictureKey varchar(1500)
modifiedDate timestamp
FeedbackSession
feedbackSessionName varchar(1500) PK
courseId varchar(1500) PK FK
creatorEmail varchar(1500)
respondingInstructorList bytea
respondingStudentList bytea
instructions varchar(1500)
createdTime timestamp
deletedTime timestamp
startTime timestamp
endTime timestamp
sessionVisibleFromTime timestamp
resultsVisibleFromTime timestamp
timeZone varchar(1500)
gracePeriod int
sentOpenEmail boolean
sentClosingEmail boolean
sentClosedEmail boolean
sentPublishedEmail boolean
isOpeningEmailEnabled boolean
isClosingEmailEnabled boolean
isPublishedEmailEnabled boolean
Instructor
courseId varchar(1500) PK FK
email varchar(1500) PK
googleId varchar(1500) N FK
isArchived boolean
name varchar(1500)
registrationKey varchar(1500)
role varchar(1500)
isDisplayedToStudents boolean
displayedName boolean
instructorPrivilegesAsText text
CourseStudent
courseId varchar(1500) PK
email varchar(1500) PK
googleId varchar(1500) N FK
name varchar(1500)
lastName varchar(1500)
comments varchar(1500)
teamName varchar(1500)
sectionName varchar(1500)
registrationKey varchar(1500)
createdAt timestamp
FeedbackQuestion
feedbackQuestionId int PK
feedbackSessionName varchar(1500) FK
courseId varchar(1500) FK
questionText varchar(1500)
questionDescription varchar(1500)
questionNumber int
questionType varchar(1500)
giverType varchar(1500)
recipientType varchar(1500)
numberOfEntitiesToGiveFeedbackTo int
showResponsesTo bytea
showGiverNameTo bytea
showRecipientNameTo bytea
createdAt timestamp
updatedAt timestamp
FeedbackResponse
feedbackResponseId int PK
feedbackSessionName varchar(1500) FK
courseId varchar(1500) FK
feedbackQuestionId int FK
feedbackQuestionType varchar(1500)
giverEmail varchar(1500)
giverSection varchar(1500)
receiver varchar(1500)
receiverSection varchar(1500)
answer text
createdAt timestamp
updatedAt timestamp
FeedbackResponseComment
feedbackResponseCommentId int PK
courseId varchar(1500) FK
feedbackSessionName varchar(1500) FK
feedbackQuestionId int FK
giverEmail varchar(1500)
commentGiverType varchar(1500)
feedbackResponseId int FK
giverSection varchar(1500)
receiverSection varchar(1500)
showCommentTo bytea
showGiverNameTo bytea
isVisibilityFollowingFeedbackQuestion boolean
isCommentFromFeedbackParticipant boolean
createdAt timestamp
commentText varchar(1500)
lastEditorEmail varchar(1500)
lastEditedAt timestamp
Vertabelo

Figure 3 – The logical data model

To hide the complexities in the data model and to abstract the storage processes, TEAMMATES puts the responsibility of doing CRUD into a dedicated component called the storage component as shown in Figure 4. The logic component will use the APIs exposed by the storage component.

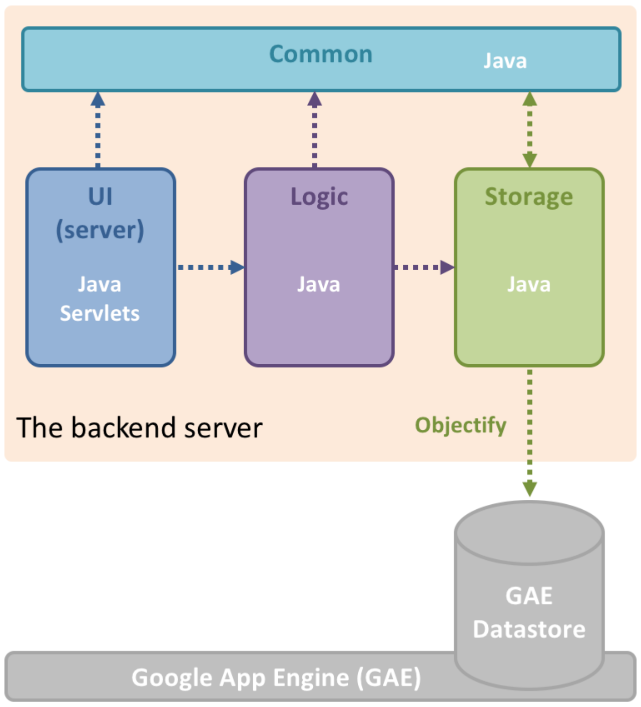


Figure 4 – Source: TEAMMATES developer guides

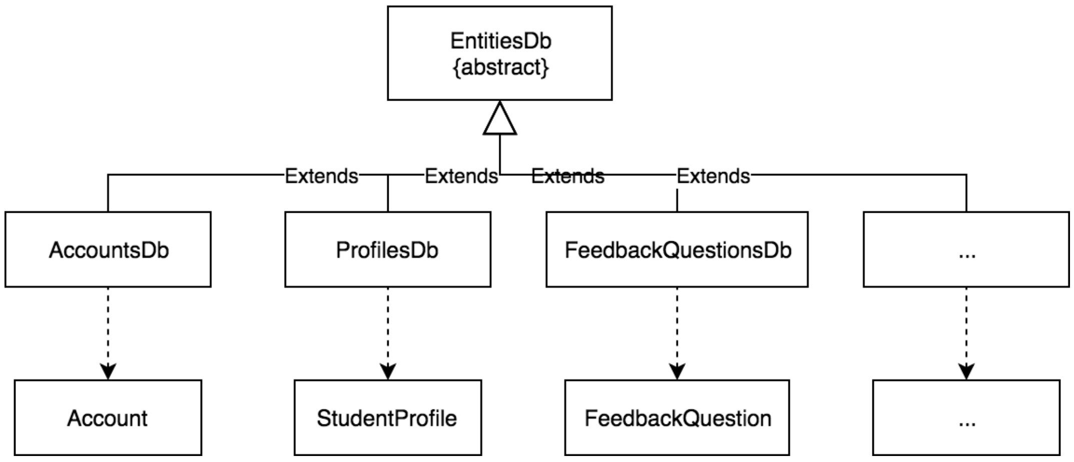


Figure 5

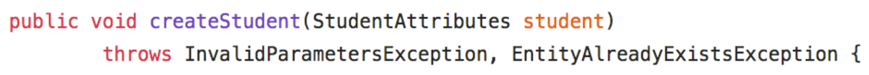
Inside the storage component, Objectify[7] is employed to interact with the Datastore. A dedicated DB class is created for each entity type (e.g., `AccountsDb` manages `Account` entities) as illustrated in Figure 5. This separates the concerns of entity management from the entities themselves.

[7] Objectify (https://github.com/objectify/objectify) is an open-source project designed to further abstract the Datastore's APIs.

Besides, a data transfer object (DTO) is created for each entity. These DTOs are used to carry entity data outside the storage component and hide any implementation-specific information. For example, `InstructorPrivilege` is a complex class that controls the privilege of instructors in different granularities. When it is stored, the storage component serializes the object into a JSON string. When the object is requested, the JSON string is deserialized into a concrete object. This hides the complexity of serialization and deserialization from the logic component, enabling it to focus on higher level logic.

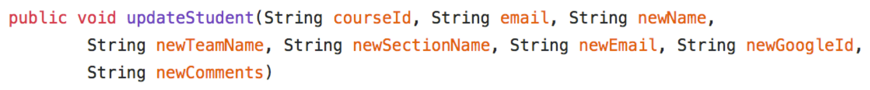
3.2. Problem Observed 1: Defective Write APIs

Upon investigation, it is found that the current create and update APIs do not return the created or updated entity. Figure 6 and Figure 7 show two typical method signatures for creating and updating a student.



```
public void createStudent(StudentAttributes student)
        throws InvalidParametersException, EntityAlreadyExistsException {
```

Figure 6

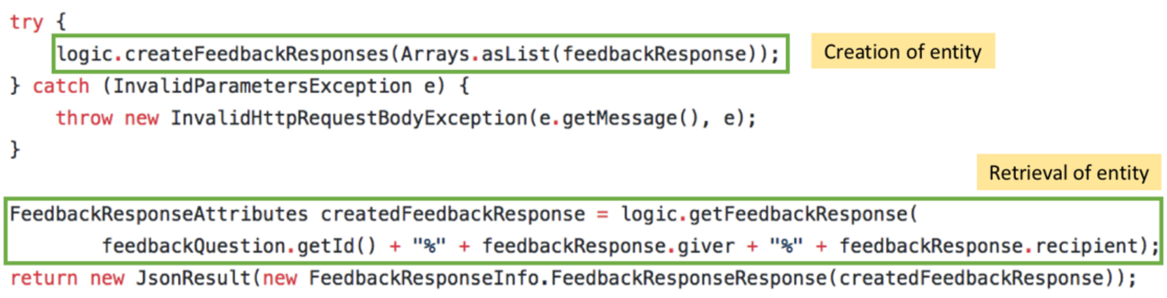


```
public void updateStudent(String courseId, String email, String newName,
        String newTeamName, String newSectionName, String newEmail, String newGoogleId,
        String newComments)
```

Figure 7

If the logic component wants to retrieve the created or updated entity, it needs to issue another unnecessary read request, which adds to the usage costs for database operations[8]. In addition, the extra read request increases the latency experienced by users. Figure 8 gives an example in the codebase where a feedback response is being retrieved immediately after the creation. As the creation of a feedback response entity is a frequent operation in TEAMMATES, one can expect this flaw to have a considerable impact on overall database read costs and request latency.

[8] Google charges applications based on theirs usages. The detailed charging plan of Google Cloud Datastore can be found in https://cloud.google.com/appengine/pricing#costs-for-datastore-calls.



```
try {
    logic.createFeedbackResponses(Arrays.asList(feedbackResponse));
} catch (InvalidParametersException e) {
    throw new InvalidHttpRequestBodyException(e.getMessage(), e);
}

FeedbackResponseAttributes createdFeedbackResponse = logic.getFeedbackResponse(
        feedbackQuestion.getId() + "%" + feedbackResponse.giver + "%" + feedbackResponse.recipient);
return new JsonResult(new FeedbackResponseInfo.FeedbackResponseResponse(createdFeedbackResponse));
```

Figure 8

In addition, the behavior of not returning the created or updated entity is also error-prone. The storage component sanitizes some fields before creating or saving entities, resulting in minor modifications of some fields. However, such modifications will not be reflected on the parameters passed in. In the case where the logic component fails to retrieve the modified entity, stale values will be used, which may cause bugs. In fact, there are already two critical bugs[9] because of this. The root cause of them is the same: some fields are modified by the storage component while the logic component still uses the old values.

3.3. Solution: Standardize Write APIs

The most logical way to fix the flaw in concern is to update the APIs so that create/update methods return the created/updated entity. There will be no extra cost in the storage layer in adding this behavior as the response from the Database will contain the necessary information to populate the fields in the entity involved. The newly added return statement is highlighted in Figure 9.

[9] https://github.com/TEAMMATES/teammates/issues/9157
https://github.com/TEAMMATES/teammates/issues/9090

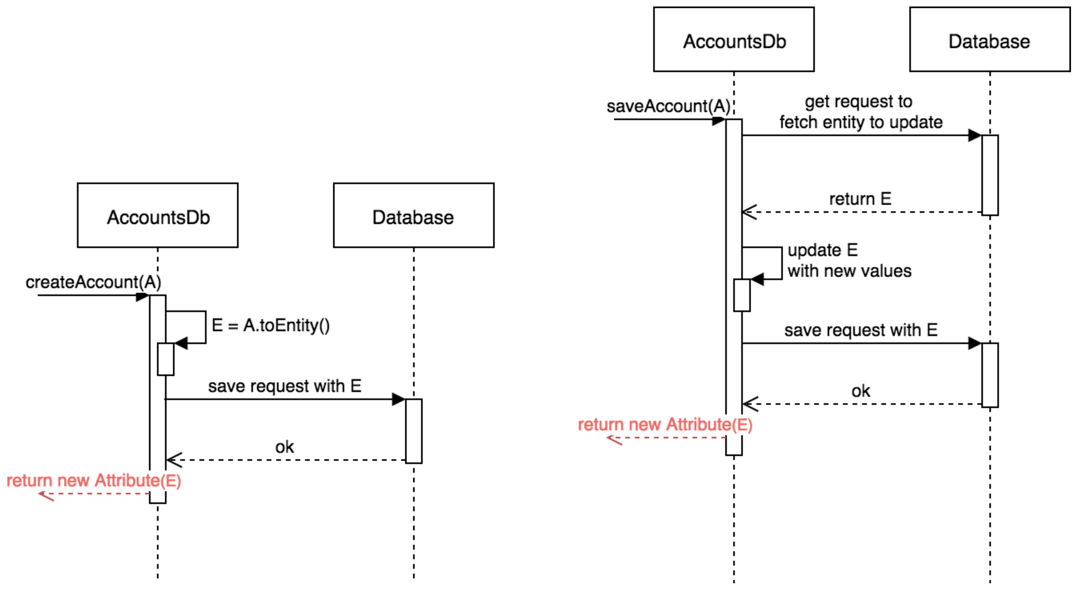
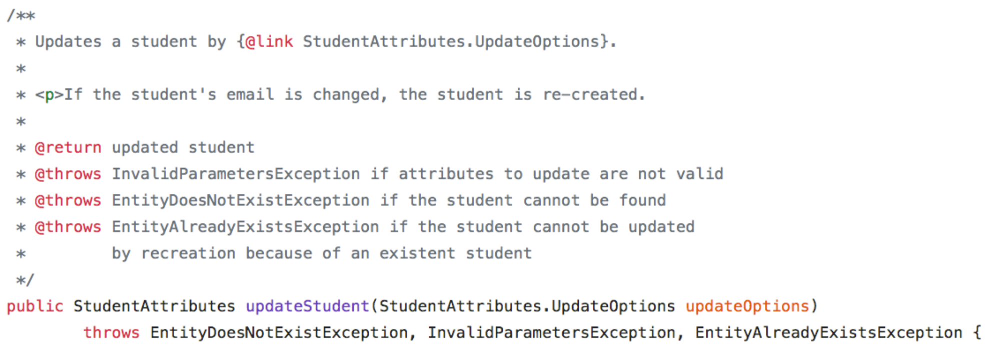


Figure 9

In doing so, the logic component can get the created and updated entity without any extra cost. It can also safely ignore the return value if the value is not needed. Javadoc was updated/created (shown in Figure 10) for each write API to reflect the new behavior. Besides, test cases were also updated to achieve 100% line and branch coverage to minimize regressions in future.



```
/**
 * Updates a student by {@link StudentAttributes.UpdateOptions}.
 *
 * <p>If the student's email is changed, the student is re-created.
 *
 * @return updated student
 * @throws InvalidParametersException if attributes to update are not valid
 * @throws EntityDoesNotExistException if the student cannot be found
 * @throws EntityAlreadyExistsException if the student cannot be updated
 *         by recreation because of an existent student
 */
public StudentAttributes updateStudent(StudentAttributes.UpdateOptions updateOptions)
        throws EntityDoesNotExistException, InvalidParametersException, EntityAlreadyExistsException {
```

Figure 10

3.4. Results

Most of the places where the created or updated entity is fetched immediately after write methods were changed to use the returned entity to save read costs and to prevent potential bugs. The code has been merged for more than 2 months and no regression has been found so far. It is expected that the Datastore read cost bill will be lower and similar bugs will not happen in the future because of the code refactoring, well-written Javadoc and fully-covered tests.

3.5. Problem Observed 2: The Outdated “KeepExistingPolicy”

Another problem lies in the “KeepExistingPolicy” for update methods (APIs). The policy requires update methods to accept the data transfer version of an entity to update. Non-null fields in the DTO will be updated with the corresponding new values while null fields are left untouched. This approach might work in the early stage of the project. However, the policy has become outdated and problematic in the current development process.

Firstly, some fields are needed to be updated to `null`. For example, a student might change his/her Google account and the association between `CourseStudent` and `Account` should be broken. In this case, the `googleId` field will be updated to `null`. However, this conflicts with the “KeepExistingPolicy” since `null` fields should be left untouched. The current solution in the codebase is to partially break the policy, which already indicates the outdatedness and incapability of the policy.

In addition, it is unclear to developers that which fields are used to identify the entity to update. The primary keys indicated in the logical data model are keys chosen to be stored in the database. Beside them, there are multiple unique constraints that are enforced by the code logic[10]. For instance, `Instructor` can be identified by `(courseId, email)` or `(courseId, googleId)`. By looking at the update method itself, which pair will be used to identify the entity to update is ambiguous. Even if there is a priority between each pair, updating `email` of an instructor to a new one would be impossible when only `courseId` and `email` of the original instructor are known. All the difficulties mentioned above have their complicated workarounds in the codebase which reduce the readability and eventually the maintainability of the code.

3.6. Solution: Upgrade the “KeepExistingPolicy”

To tackle the problematic and outdated updated policy, the “KeepExistingPolicy” is upgraded and extended. Instead of using `null` as a special value to indicate an unchanged property, a dedicated class called `UpdateOption` is used. In addition, a data transfer object

[10] Due to the limitation of the Datastore as a No-SQL database, the unique constraint is enforced by code rather than the database itself.

`UpdateOptions` is created for each entity to carry the values to update. Figure 11 uses `StudentAttributes` as an example to illustrate the relationships.

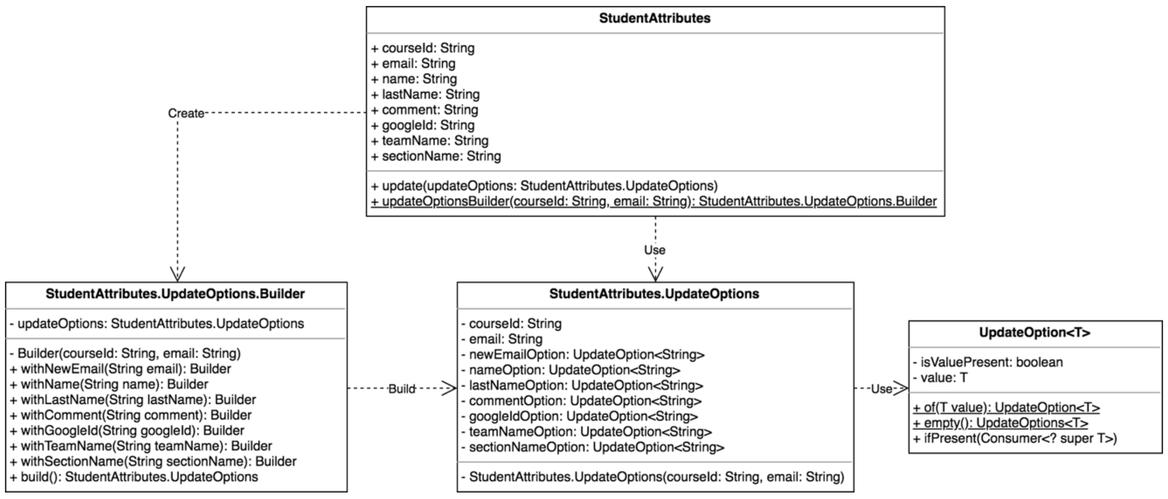

StudentAttributes
+ courseId: String
+ email: String
+ name: String
+ lastName: String
+ comment: String
+ googleId: String
+ teamName: String
+ sectionName: String
+ update(updateOptions: StudentAttributes.UpdateOptions)
+ updateOptionsBuilder(courseId: String, email: String): StudentAttributes.UpdateOptions.Builder
Create
Use
StudentAttributes.UpdateOptions.Builder
- updateOptions: StudentAttributes.UpdateOptions
- Builder(courseId: String, email: String)
+ withNewEmail(String email): Builder
+ withName(String name): Builder
+ withLastName(String lastName): Builder
+ withComment(String comment): Builder
+ withGoogleId(String googleId): Builder
+ withTeamName(String teamName): Builder
+ withSectionName(String sectionName): Builder
+ build(): StudentAttributes.UpdateOptions
Build
StudentAttributes.UpdateOptions
- courseId: String
- email: String
- newEmailOption: UpdateOption<String>
- nameOption: UpdateOption<String>
- lastNameOption: UpdateOption<String>
- commentOption: UpdateOption<String>
- googleIdOption: UpdateOption<String>
- teamNameOption: UpdateOption<String>
- sectionNameOption: UpdateOption<String>
- StudentAttributes.UpdateOptions(courseId: String, email: String)
Use
UpdateOption<T>
- isValuePresent: boolean
- value: T
+ of(T value): UpdateOption<T>
+ empty(): UpdateOptions<T>
+ ifPresent(Consumer<? super T>)

Figure 11

The static method `updateOptionsBuilder()` in `StudentAttributes` returns a builder for `UpdateOptions`. In the method signature, it clearly states the criteria used to identify the update entity. In this case, the pair `(courseId, email)` is used. A `UpdateOptions` can be built with a chain of method calls of the builder. Each method call of the builder will set the corresponding `UpdateOption` with the value to update. It should be noted that `null` value is allowed so that field such as `googleId` can be updated to `null`. It is also possible to update the email of a student with `(courseId, email)` as an identification pair by calling `withNewEmail()`.

When the logic component wants to update an entity, it calls the corresponding update method in the storage component with the populated `UpdateOptions` rather than its DTO.

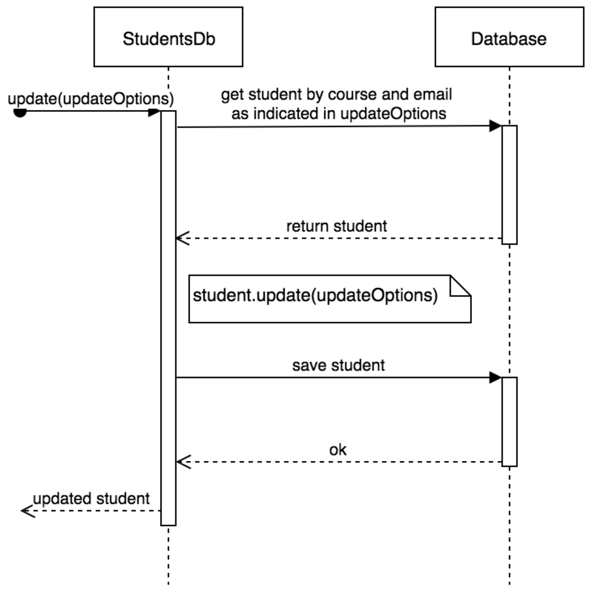

StudentsDb
Database
update(updateOptions)
get student by course and email as indicated in updateOptions
return student
student.update(updateOptions)
save student
ok
updated student

Figure 12

Figure 12 shows how a student is updated with `UpdateOptions`. Firstly, the student to update is fetched based on the `(courseId, email)` specified in `UpdateOptions`. Then, the `update()` method in `StudentAttributes` is invoked to populate fields with new values. Finally, a save request is issued and the updated student is returned, which is consistent with the solution introduced for the defective write APIs.

It is also possible to use different pairs to identify the entity to update as shown in Figure 13. An instructor can be updated by using `UpdateOptionsWithGoolgeId` and `UpdateOptionsWithEmail`, which corresponds to the pair `(courseId, email)` and pair `(courseId, googleId)` respectively. In each `Options`, fields that are updatable are shown explicitly to eliminate any confusion from developers. For example, in this case, email is not updatable by using `(courseId, email)`.

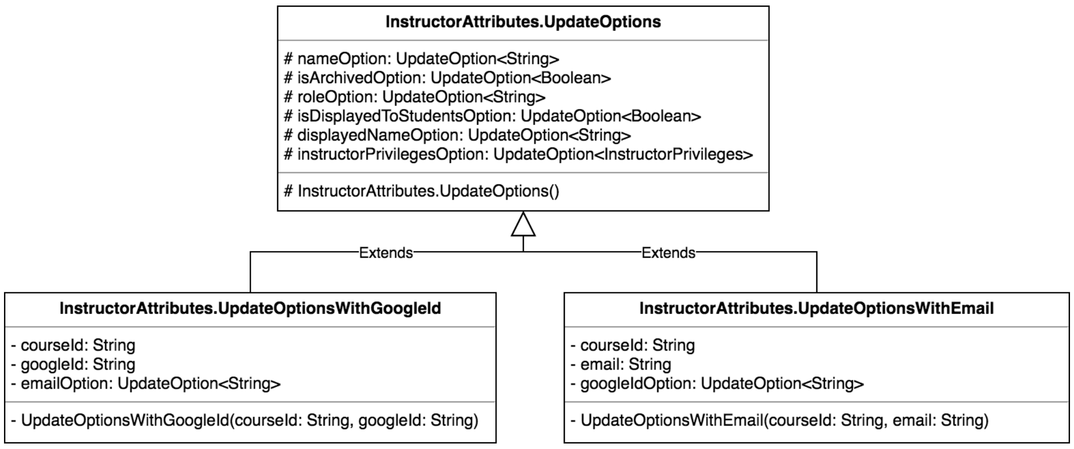

InstructorAttributes.UpdateOptions
# nameOption: UpdateOption<String>
# isArchivedOption: UpdateOption<Boolean>
# roleOption: UpdateOption<String>
# isDisplayedToStudentsOption: UpdateOption<Boolean>
# displayedNameOption: UpdateOption<String>
# instructorPrivilegesOption: UpdateOption<InstructorPrivileges>
# InstructorAttributes.UpdateOptions()
Extends
Extends
InstructorAttributes.UpdateOptionsWithGoogleId
- courseId: String
- googleId: String
- emailOption: UpdateOption<String>
- UpdateOptionsWithGoogleId(courseId: String, googleId: String)
InstructorAttributes.UpdateOptionsWithEmail
- courseId: String
- email: String
- googleIdOption: UpdateOption<String>
- UpdateOptionsWithEmail(courseId: String, email: String)

Figure 13

Furthermore, since we know which fields are going to be updated, it is possible to implement the optimized saving policy as mentioned in the interim report (Xiao, 2018). In the interim report, one of bottleneck regarding slow submission of feedback session was identified and the optimized saving policy was proposed to be one of the solutions. The optimized saving policy suggests entities be updated only if they are changed. The report also mentioned that the optimized saving policy could increase the throughput of feedback submission by 32% and decrease the latency by about 25% on average (shown as Figure 14 and Figure 15)[11]. Therefore, the saving process in Figure 12 is modified to adopt the policy.

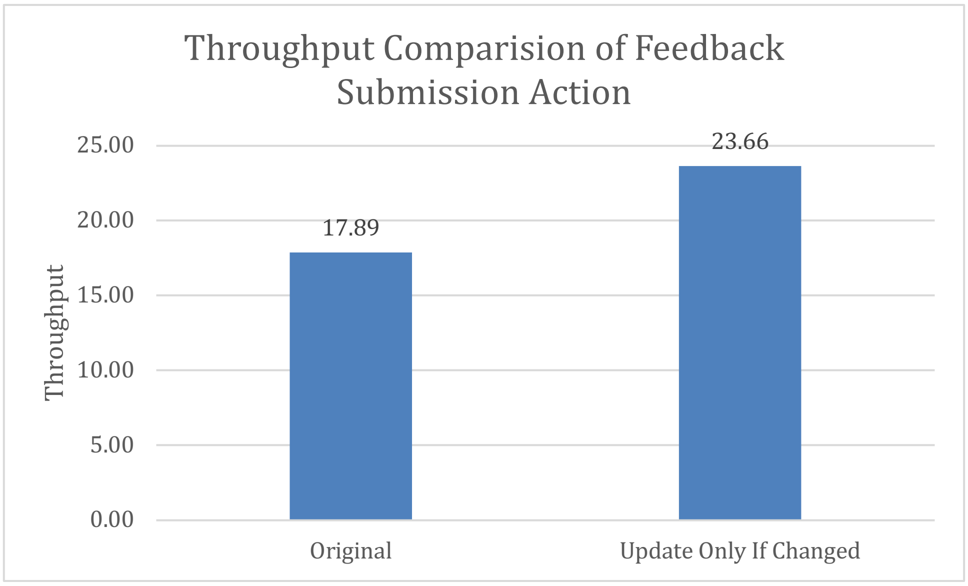


Figure 14

[11] The raw experiment data can be found in Appendix A – Optimized Saving Policy Benchmarking Data.

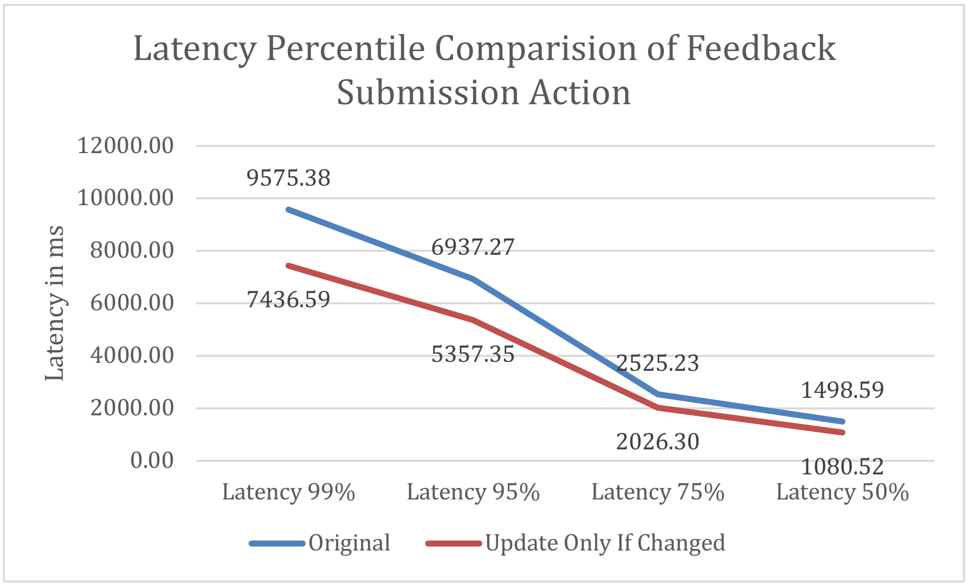


Figure 15

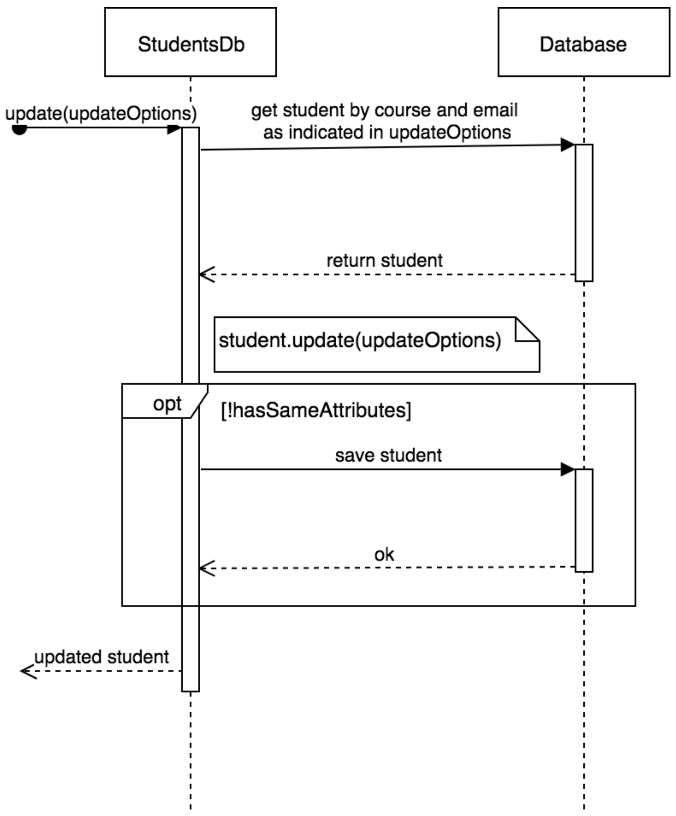


Figure 16

As indicated in Figure 16, save request is issued only if the updatable fields of a student are changed.

3.7. Results

The upgraded policy was implemented and the code was merged into the master branch. So far, the upgrade policy fits both the old and new requirements of the applications well, bringing the update methods to the next maintainable stage. Besides the improvements in maintainability, due to the refactoring, the optimized update policy gets implemented in a clean and clear way. This finish one of the leftover tasks in the interim report. The improvement regarding feedback submission action is proven in the benchmarking and it is expected that the overall performance of the application as a whole will also be improved as the optimized saving policy is implemented for all entities shown in the logic data model.

3.8. Problem Observed 3: Slow Cascade Deletion

The last problem is about the performance aspect and it is observed that the current cascade deletion strategy has a high latency.

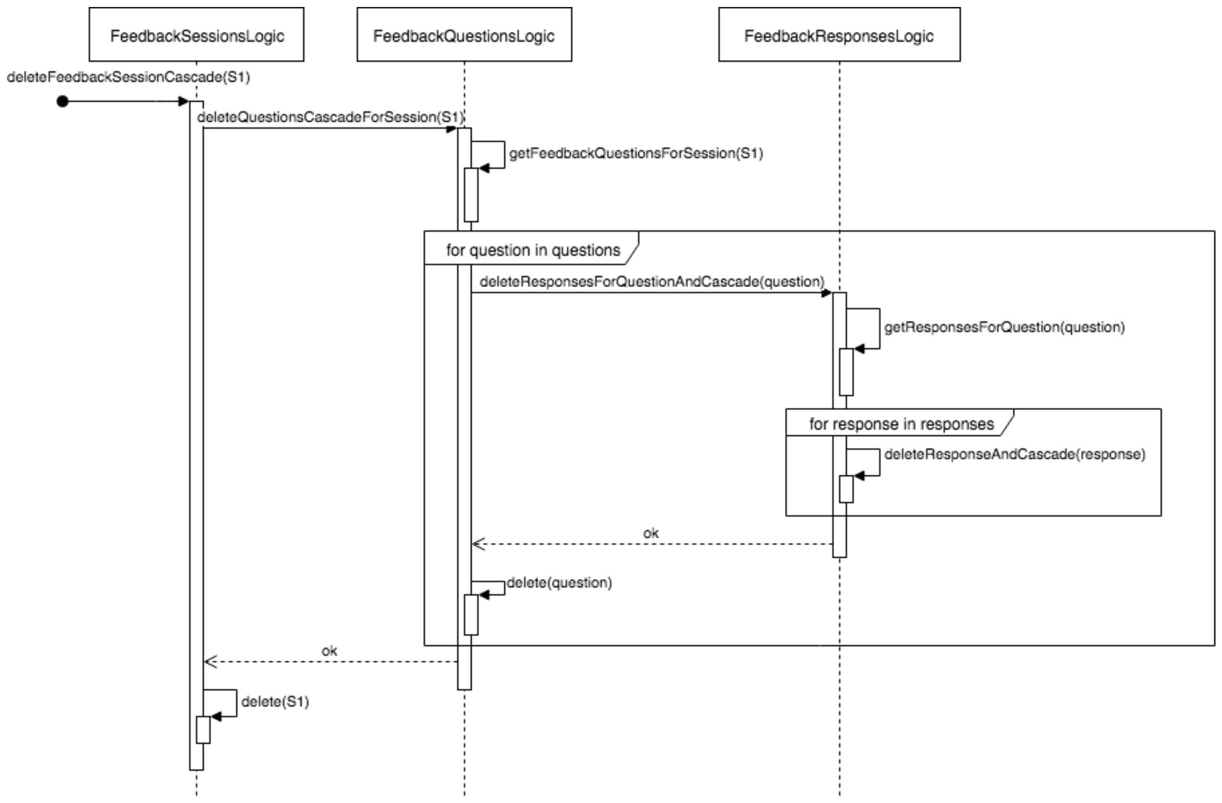

FeedbackSessionsLogic
FeedbackQuestionsLogic
FeedbackResponsesLogic
deleteFeedbackSessionCascade(S1)
deleteQuestionsCascadeForSession(S1)
getFeedbackQuestionsForSession(S1)
for question in questions
deleteResponsesForQuestionAndCascade(question)
getResponsesForQuestion(question)
for response in responses
deleteResponseAndCascade(response)
ok
delete(question)
ok
delete(S1)

Figure 17

Figure 17 shows a typical deletion process for the current cascade deletion strategy. Involved entities are fetched at once and are deleted one by one. In the deletion method in each logic, the deletion request is forwarded to the storage component.

Let us denoted the overhead such as communication cost between server and database when issuing database manipulation requests as `h`. If there are 10 questions for a typical feedback session, 100 responses for each question and 0 comment for each response, we would expect the overhead of the current deletion strategy to be

```
1h + 10h + 1000h + 1h + 10h = 1022h
```

The first `1011h` comes from the overhead of deletion requests and the second `11h` comes from the overhead of retrieval requests. This number can be subtle if `h` is small. However, in the benchmarking to a staging server where the same number of questions, responses, and comments is configured, the operation takes around 40 seconds to complete. Such latency is not acceptable as it is quite close to the 60 second time limit required by GAE. When there are more questions and responses, the request will simply time out because it exceeds 60 seconds, leaving an error message to users and an inconsistent database.

3.9. Solution: Apply Batch Deletion

To boost the performance of cascade deletion, the deletion strategy is redesigned so that batch deletion is supported. Thanks to the hierarchy as shown in the data model, when a parent is deleted, all of its *descendants*[12] can be obtained and deleted without fetching their associated parents. The process is illustrated in Figure 18 using the deletion of a feedback session as an example.

[12] The descendants of an element is the children of an element together with all descendants of the children.

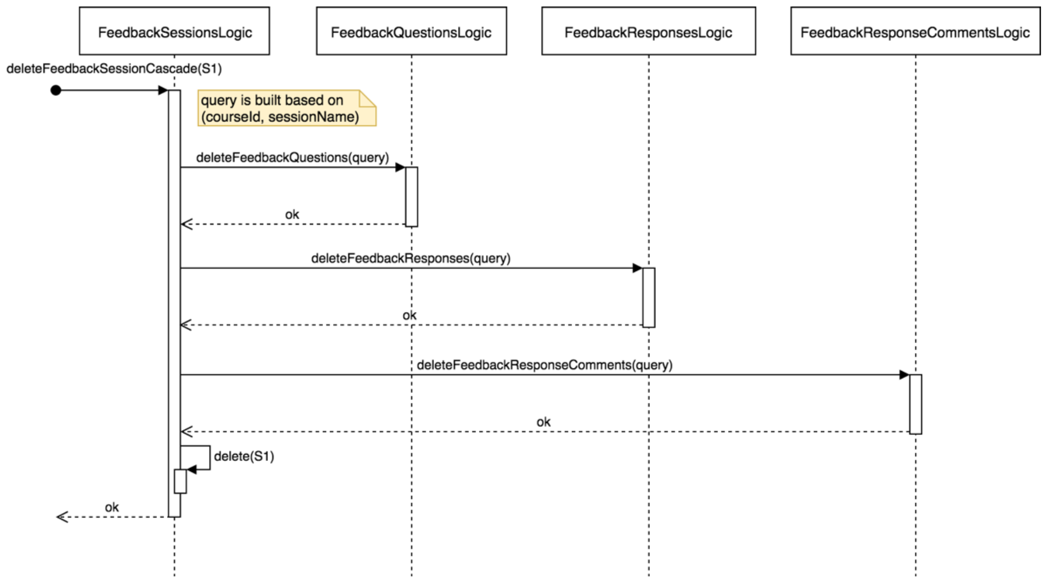
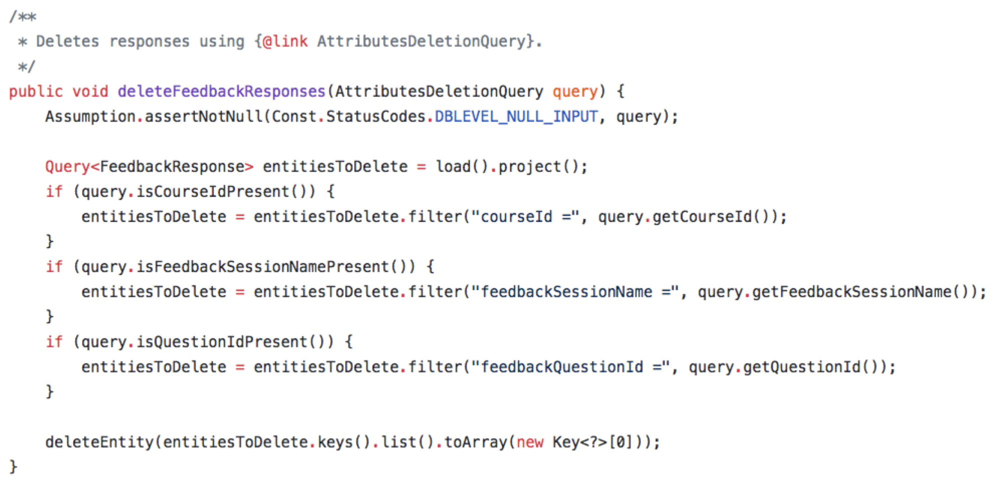

FeedbackSessionsLogic
FeedbackQuestionsLogic
FeedbackResponsesLogic
FeedbackResponseCommentsLogic
deleteFeedbackSessionCascade(S1)
query is built based on (courseId, sessionName)
deleteFeedbackQuestions(query)
ok
deleteFeedbackResponses(query)
ok
deleteFeedbackResponseComments(query)
ok
delete(S1)
ok

Figure 18

Again, the deletion requests in logic will be forwarded to the storage component. In the storage component, the keys of entities to delete are fetched based on the granularity specified by `AttributesDeletionQuery` (shown in Figure 19).



```
/**
 * Deletes responses using {@link AttributesDeletionQuery}.
 */
public void deleteFeedbackResponses(AttributesDeletionQuery query) {
    Assumption.assertNotNull(Const.StatusCodes.DBLEVEL_NULL_INPUT, query);

    Query<FeedbackResponse> entitiesToDelete = load().project();
    if (query.isCourseIdPresent()) {
        entitiesToDelete = entitiesToDelete.filter("courseId =", query.getCourseId());
    }
    if (query.isFeedbackSessionNamePresent()) {
        entitiesToDelete = entitiesToDelete.filter("feedbackSessionName =", query.getFeedbackSessionName());
    }
    if (query.isQuestionIdPresent()) {
        entitiesToDelete = entitiesToDelete.filter("feedbackQuestionId =", query.getQuestionId());
    }

    deleteEntity(entitiesToDelete.keys().list().toArray(new Key<?>[0]));
}
```

Figure 19

An `AttributesDeletionQuery` can be built with the associated builder (shown in Figure 20) and the granularity is defined based on the method calls. For example, a query to delete entities involved in a course can be constructed by calling `withCourseId()`.

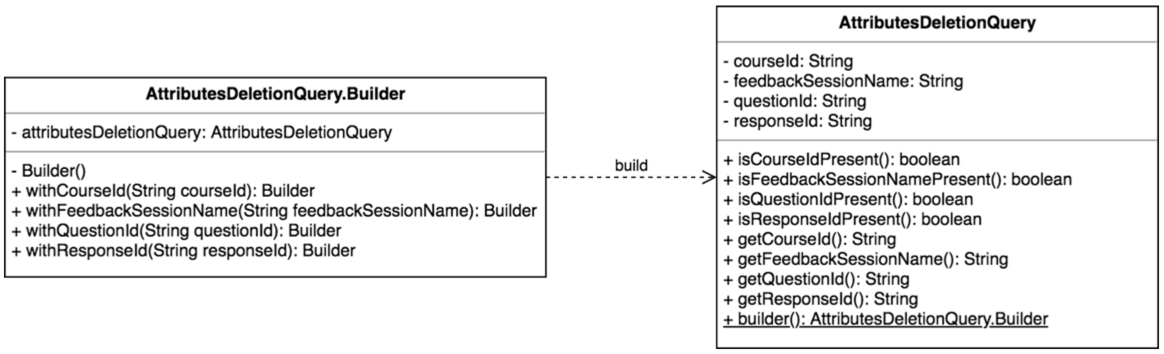

AttributesDeletionQuery.Builder
- attributesDeletionQuery: AttributesDeletionQuery
- Builder()
+ withCourseId(String courseId): Builder
+ withFeedbackSessionName(String feedbackSessionName): Builder
+ withQuestionId(String questionId): Builder
+ withResponseId(String responseId): Builder
build
AttributesDeletionQuery
- courseId: String
- feedbackSessionName: String
- questionId: String
- responseId: String
+ isCourseIdPresent(): boolean
+ isFeedbackSessionNamePresent(): boolean
+ isQuestionIdPresent(): boolean
+ isResponseIdPresent(): boolean
+ getCourseId(): String
+ getFeedbackSessionName(): String
+ getQuestionId(): String
+ getResponseId(): String
+ builder(): AttributesDeletionQuery.Builder

Figure 20

In the same scenario shown in section 3.8, the overhead caused by the redesigned deletion strategy is just

`1h + 1h + 1h + 1h + 1h + 1h = 6h`

The first `3h` comes from deletion of the feedback session, questions and responses and the second `3h` comes from keys retrievals of questions, response and comments[13]. Compared to the old overhead, the new deletion strategy significantly decreases the overhead by about 99.4%.

Moreover, since only keys of involved entities are retrieved rather than the actual entities, the deletion would be more cost-effective and efficient. In the charging plan of the Datastore, it indicates that there is no read cost for key-only queries[14]. Thus, the get operation to retrieve entities to delete is free-of-charge. Besides, according to Cooper (2009), when a query only requests to retrieve the keys of the matched entities, the operation is approximately two times faster than retrieving the full attributes. Therefore, it is expected that the batch deletion would be even faster. In fact, the benchmarking conducted in the next section confirms the hypothesis.

3.10. Results

The same benchmarking to the staging server where a feedback session is deleted has also been conducted for the new deletion strategy.

[13] There is no comment in the scenario so the retrieval of comments will return empty result. This is also why the first `3h` does not contain the cost for deletion of comments.

[14] Refer to the charging plan of Google App Engine: https://cloud.google.com/appengine/pricing#costs-for-datastore-calls.

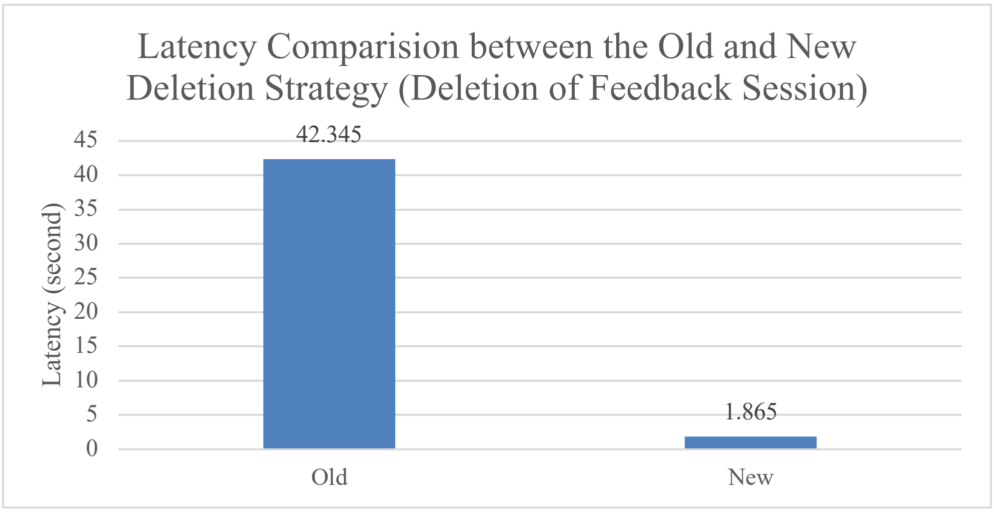


Figure 21

In Figure 21, while the old deletion strategy takes around 40 seconds to complete, the new deletion strategy finishes in less than 2 seconds, improving the performance by 95.5%.

An experiment has also been done for the cascade deletion of a typical feedback question. 100 responses are set for a question and each response is configured to have 20 comments.

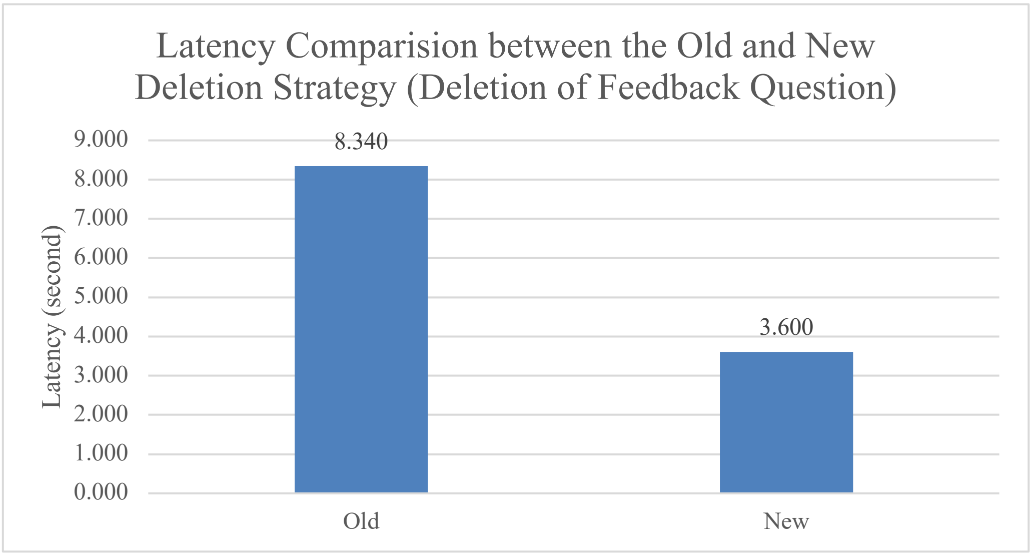


Figure 22

In this case (shown in Figure 22), the latency decreases by about 56.8%. The performance boost is not as much as that for the feedback session because the old deletion method partially adopts the concept of batch deletion.

For other entities, namely courses and comments, the concept of batch deletion is already adopted before the introduction of `AttributesDeletionQuery`. Those methods are refactored to have consistent method signatures. In fact, it would be messier if they are not refactored.

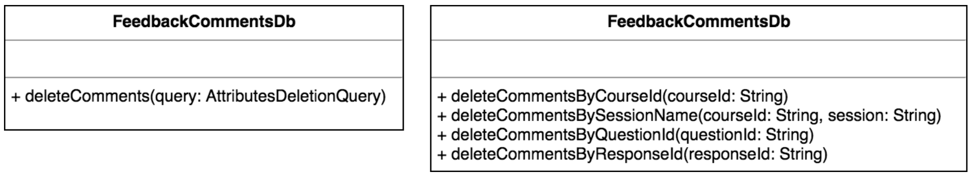

FeedbackCommentsDb
+ deleteComments(query: AttributesDeletionQuery)
FeedbackCommentsDb
+ deleteCommentsByCourseId(courseId: String)
+ deleteCommentsBySessionName(courseId: String, session: String)
+ deleteCommentsByQuestionId(questionId: String)
+ deleteCommentsByResponseId(responseId: String)

Figure 23

As indicated in Figure 23, without `AttributesDeletionQuery`, deletion methods in `FeedbackCommentsDb` would need to be duplicated four times, introducing more complexity and requiring more maintenance efforts.

In summary, all deletions methods are standardized and refactored to apply batch deletions. The performance of cascade deletion is improved significantly.

4. Achievement 2: Improved Scalability and Performance of the Client Package

4.1. Background

The client package in TEAMMATES is mainly used to do data migration or to collect the usage of the application.

For example, when a new field is added to an entity, in order to write consistent code without any special handler of legacy data format, a new client script is needed to add a default value to the new field for all old entities. Such script is known as the data migration script.

The project manager may also use a dedicated script called stats script to collect the overall usage of TEAMMATES such as the number of institutes which adopt TEAMMATES as a feedback system and the number of unique users in the system.

In order to monitor the progress of any client scripts, all scripts are run locally on a typical Windows computer owned by the project manager.

4.2. Problems Observed: Unscalable Process, Poor Performance and Data Inconsistency

This section presents three inter-connected problems found in the script package: unscalable process, poor performance and data inconsistency.

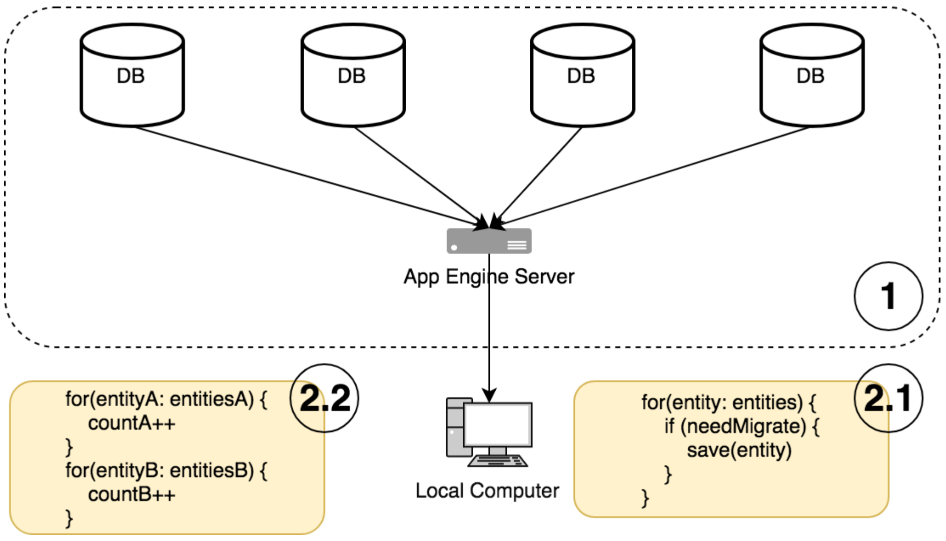


Figure 24

Figure 24 illustrates the process of doing data migration or statistics calculation in the client package. All concerned entities are loaded first (indicated as 1) and then processed one by one (indicated as 2.1 or 2.2).

It is found that the above-mentioned process is not scalable because it loads all involved entities at once. When the amount of data grows, it is impossible to load all of them because of the limited memory space. A benchmarking for a typical data migration script is conducted against a database with 100K entities and Java heap space is monitored using YourKit[15].

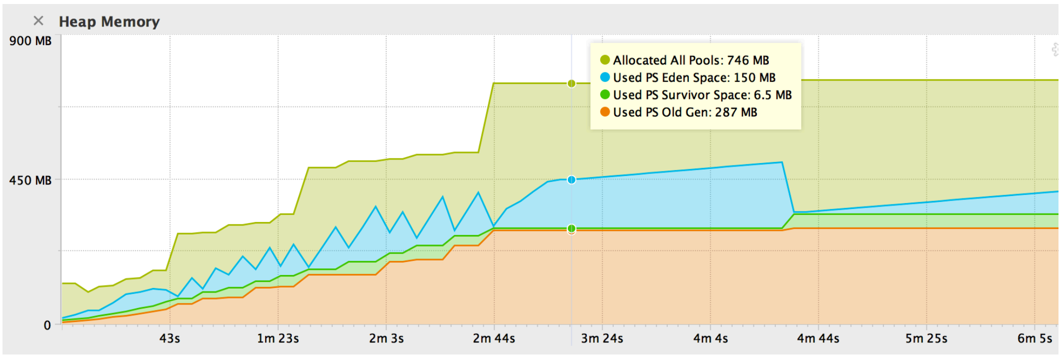


Figure 25

We can clearly distinguish the phase for step 1 and step 2 as indicated in Figure 25. When the script is migrating entity one by one, the stabilized occupied memory size is about 750MB. While a typical computer nowadays can easily handle such amount of memory, when the user

[15] YourKit (https://www.yourkit.com/) is an industry leading profiler tool for Java applications.

base grows to millions, the script will become brittle. In addition, the benchmarking only tests one entity type while the stats script requires more entities to be loaded in order to calculate the usage (show as 2.2 in Figure 24). Thus, the stats script requires even more memory space. In fact, on Nov 1, 2018, the stats script failed to run because of `OutOfMemoryError`[16] when the user base of the application exceeded 300K.

Beside the space complexity, it is also worth discussing the time complexity. According to the project manager, the completion time of the stats script is acceptable while the performance of data migration scripts is poor. It is reported that some data migration scripts take hours or even days to finish. This is reproduced by the above-mentioned benchmarking where the data migration script takes 4.3 hours to complete. Loading all 100K entities takes around 3 minutes while saving entity one by one takes about 4 hours. Thus, it is reasonable to estimate that migrating a million entities will take around 40 hours.

One might think that 40 hours are acceptable. However, the script faces the risk of interruption. The script might exit halfway unexpectedly due to various reasons such as unstable networks. If this happens, the script needs to be re-started. All procedures are executed again even some entities have been migrated. In the worst case, if 1M entities need to be migrated while the script always stops after 500K entities are migrated because of poor connections[17], the data migration will never complete. In fact, due to this concern, one of a data migration script is pending to be run for 6 months because it involves 30K entities.

Last but not least, data consistency will become a significant problem when the completion time for the data migration becomes longer and longer.

---

[16] Issue reported by project manager on https://github.com/TEAMMATES/teammates-ops/issues/7.
[17] It is reported by the project manager that GAE will kill the data migration script in several hours. Though the behavior is not stated in the GAE's documentation, it is a situation that needs to be taken into consideration.

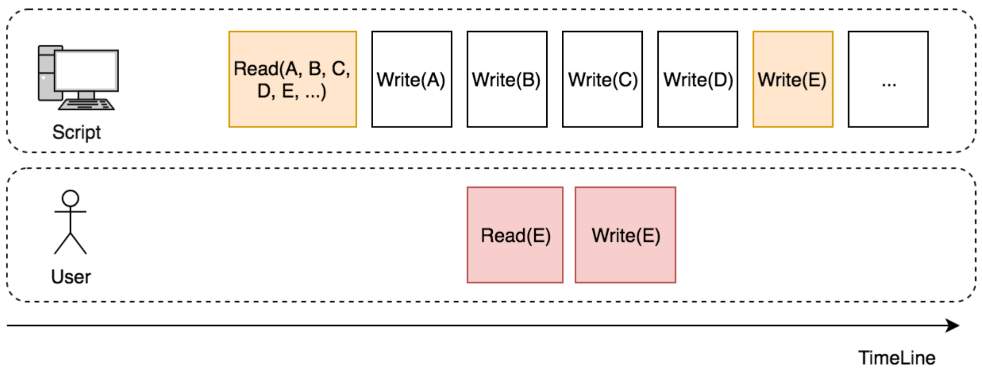


Figure 26

As shown in Figure 26, entity E might be changed by users during the data migration. Because all entities are loaded at the beginning, there is no way for the script to know the update of entity E. Thus, entity E might be saved with stale values, which might lead to data inconsistency. In the above-mentioned benchmarking, the time to save entity one by one is about 4 hours. That is to say; there is at most 4 hours interval between the read and the write operations. It is totally possible that users may update the involved entities during the period.

4.3. Solution: Fetch Entities in Batches

To occupy less memory space, one can only fetch a limited number of entities and run the script multiple times. For example, a script could be designed to get entities created within the past year first. After that, another round of execution could be conducted for entities created between last year and the year before. If the migration schemes are adequately scheduled, the script should be able to handle a large amount of data. Indeed, the stats script is partially redesigned with such technique applied. As shown in Figure 27, entities created within a month is fully loaded first[18]. After that, the calculation of statistics will begin and save the computed data locally. This is shown as 1.1, 1.2 and 1.3.

[18] Figure 27 and Figure 28 assume the involved data comes from a single Database server for illustration purposes. In real, the Datastore distributes entities based on their associated keys.

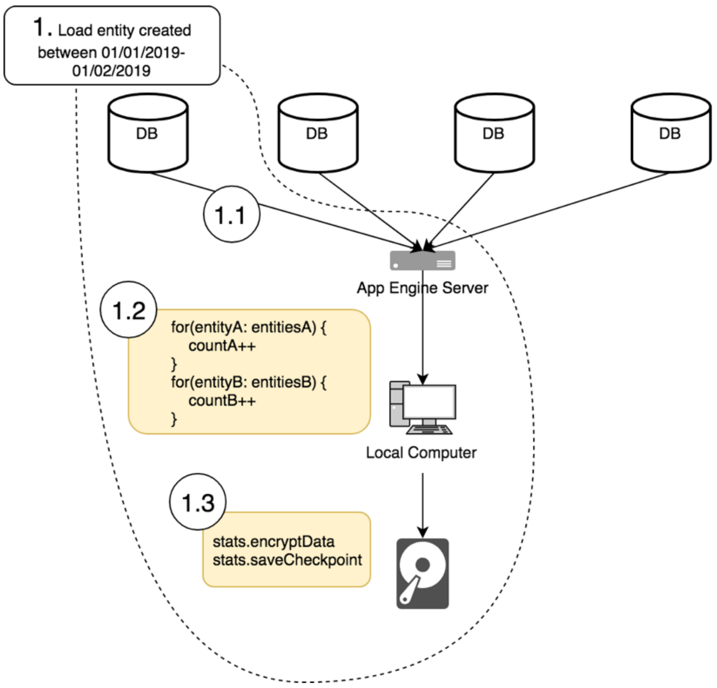


Figure 27

After one iteration is completed, another iteration begins and the stats are calculated based on the checkpoint saved in the previous iteration. This is shown as 2.1, 2.2 and 2.3 in Figure 28.

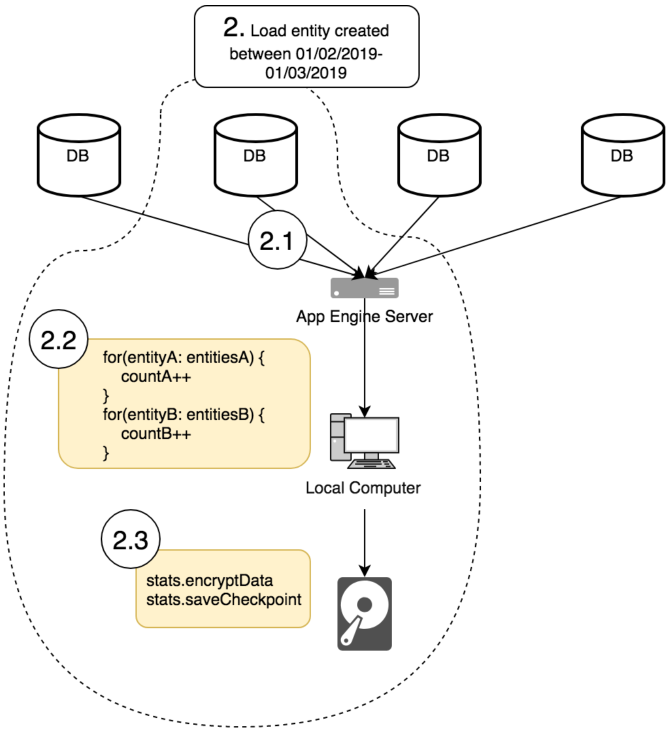


Figure 28

The iteration is made to be automatic and the script is able to remember the last query interval. Therefore, if the stats script was executed one month ago and the project manager wants to get the statistics of this month, only one iteration is needed as all previous statistics are stored locally.

However, the same idea cannot be applied to the data migration script as some entities do not track creation time. In addition, it is difficult to control the number of entities in each group as the number of entities created within a time interval is unknown. In fact, this is also a problem faced by the stats script. In the worst case, loading those entities will still result in `OutOfMemoryError`. The scripts are still not scalable.

After a detailed investigation on the documentation of Datastore, it is noted that the `Cursor` support could be used to solve the scalability issue. The `Cursor` represents the position of the currently matched entities. When supplied into a query, the `Cursor` tells the Datastore to start from the position indicated rather than querying from the beginning. For example, as shown in Figure 29, a script could query the first 100 entities and record down the `Cursor`. After the processing of the 100 entities, another query could be issued with the recorded `Cursor` attached. In doing this, only entities after the first 100 entities will be fetched. The technique could be applied again and again, enabling the script to do incremental data migration by fetching entities in batches.

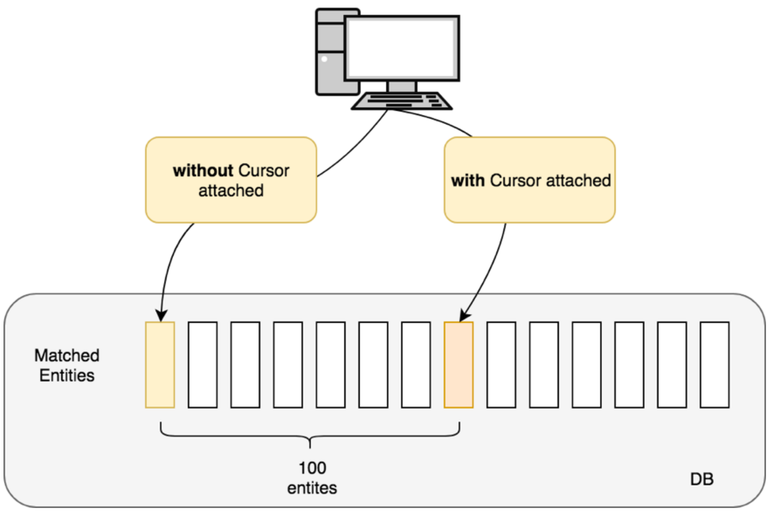


Figure 29

With this technique, the data migration script becomes scalable. Even if the script is stopped unexpectedly, because the script persists the latest `Cursor` position, it can be re-started from the last failing point.

The stats script also adopts the `Cursor` technique as the number of entities created in a time interval is unknown. Along with the technique mentioned in Figure 27 and Figure 28, the stats script also become capable of aggregating usage of the application for a million users.

4.4. Solution: Save Entities in Batches

Since 99% of the time is spent on saving entities one by one as presented in section 4.2, some optimizations should be done to the saving process. The best practices presented by Google (Limitations and Best Practices - Remote API for Java, 2019) are explored and the technique of saving entities in batches is applied. Figure 30 and Figure 31 illustrate the differences of migrating 100 entities.

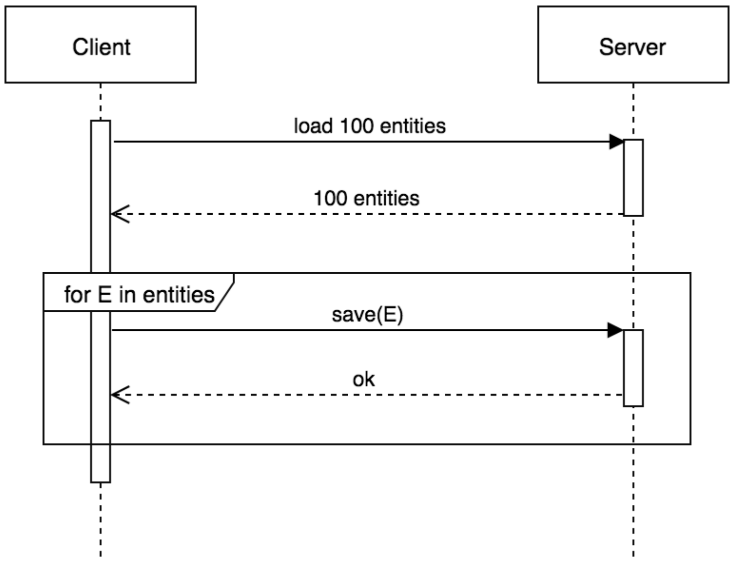


Figure 30

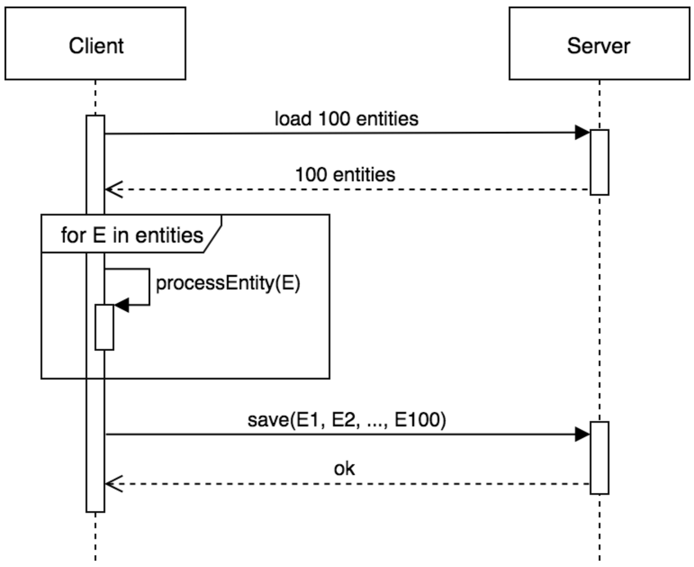


Figure 31

If the overhead such as network communication is denoted as `h`, the total estimated overhead for the old saving process (Figure 30) is:

`h + 100 * h = 101h`

For the new batch saving process (Figure 31), the total estimated overhead is

`h + h = 2h`

Thus, the total saved time is:

`101h - 2h = 99h`

It should be noted that `99h` is only the saved time for 100 entities. For thousands of entities, the saving process as shown in Figure 30 and Figure 31 will be applied again and again[19]. Thus, it is expected a significant decrease in completion time when a large number of entities is involved.

4.5. Solution: Introduce Transaction Support

If the time interval between the read and write operation is less than 1 second, the potential data inconsistency in TEAMMATES is tolerable as the scenario is unlikely to happen. However, for some entities, data consistency might be a critical business requirement. For example, there is a respondent set shared among all students in a feedback session. The respondent set will be updated based on whether a student has answered the feedback session

[19] Note that the `Cursor` technique introduced in the previous section is applied.

or not. Failure to achieve consistent respondent set may lead to wrong decisions made by instructors. Therefore, there is a need to have a more robust solution rather than manually minimizing the interval between the read and write operations.

Fortunately, the Datastore provides transaction support, which will ensure the isolation of the execution.

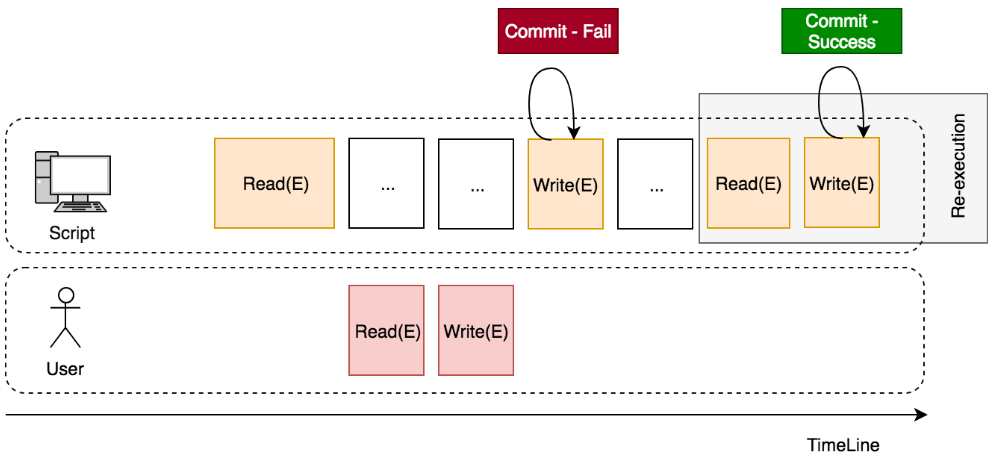


Figure 32

As shown in Figure 32, when the execution is wrapped in a transaction, instead of blindly executing write operation, the Datastore will execute an extra commit phase to check whether the involved entity has been updated or not. If it updated, it will fail the commit and retry it later[20]. Thus, there will not be any accidental overridden with stale values. However, a transaction comes with costs. According to Google (2019), transactions are less efficient when using with the GAE Remote API. Due to this, the data migration script is designed with the transaction option off. The decision on whether a transaction should be used for a specific data migration task is left for developers to decide.

4.6. Results

Both the new stats script and data migration scripts have been delivered to the project. According to the project manager[21], the new stats script continued to count statistics correctly when the user base reached 300K. With the help of the new data migration script, the leftover

[20] This is known as optimistic concurrency control.
[21] The discussion can be found in https://github.com/TEAMMATES/teammates/pull/9232.

data migration task with 30K entities to migrate was also completed without any error. Besides, many other scripts have been developed based on the new techniques introduced. For example, a scanning script has been introduced to check properties of entities, helping developers to verify the hypothesis made. A Google account migration script has been introduced to transfer users' data when they change their Google accounts. To sum up, all scripts are proven to be implemented correctly.

To verify the ability of those scripts for a million users, some experiments have also been conducted. Figure 33 shows the memory usage when 1 million simulated entities are involved for the stats script. The increase of the memory is still manageable as only 1.7GB memory is occupied under such a big scale.

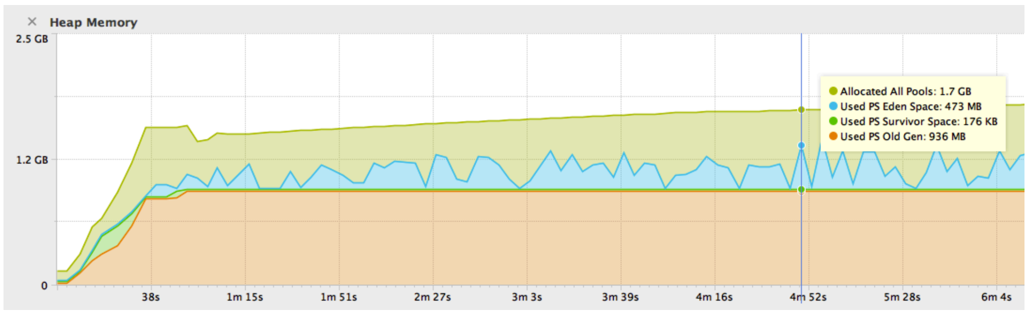


Figure 33

The benchmarking conducted for the old data migration script has also been run against the new data migration script. 100 entities are fetched each time with the `Cursor` support. As shown in Figure 34, only 170MB memory is occupied, which decreases the memory usage by 77.2% when comparing to the old one.

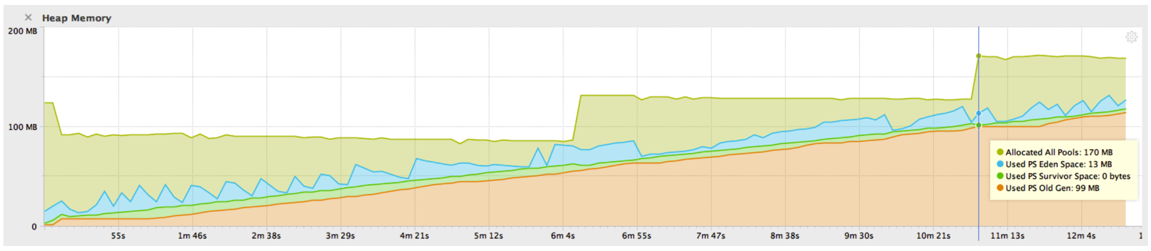


Figure 34

Moreover, both the stats script and the data migration script have much better completion time. The project manager reports that it only takes around 5 minutes to complete one iteration for the stats script. The incremental aggregation indeed helps to reduce the completion time significantly.

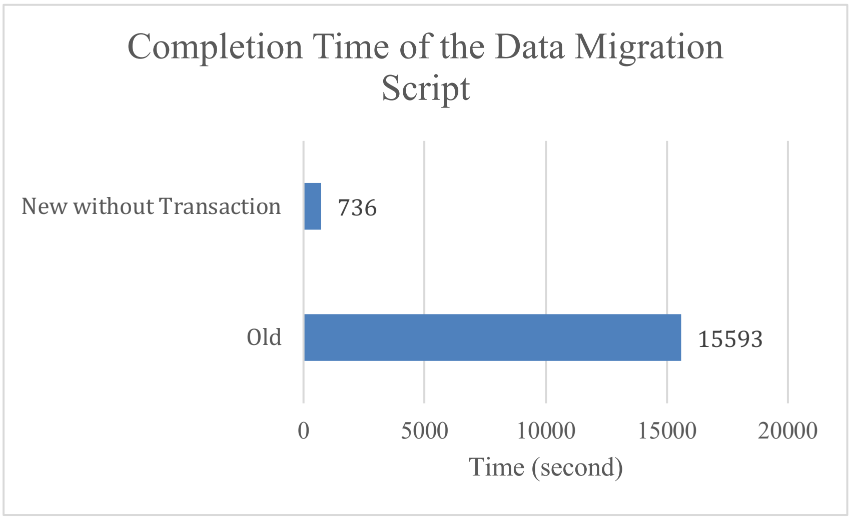


Figure 35

In the same benchmarking where 100K entities are scheduled to be migrated, the new data migration script finishes in 736 seconds when transaction is disabled as shown in Figure 35, decreasing the completion time by 95.2% when comparing to the old data migration script.

Last but not least, it is also worth to examine the time interval between the read and the write operations. In the new data migration script, the worst time interval is about 0.67 seconds, which is a big improvement compared to the old data migration script. Such smaller interval provides more guarantee for data consistency and the small risk is acceptable. If strong data consistency is needed, transaction can be used. However, it comes with significant costs as shown in Figure 36.

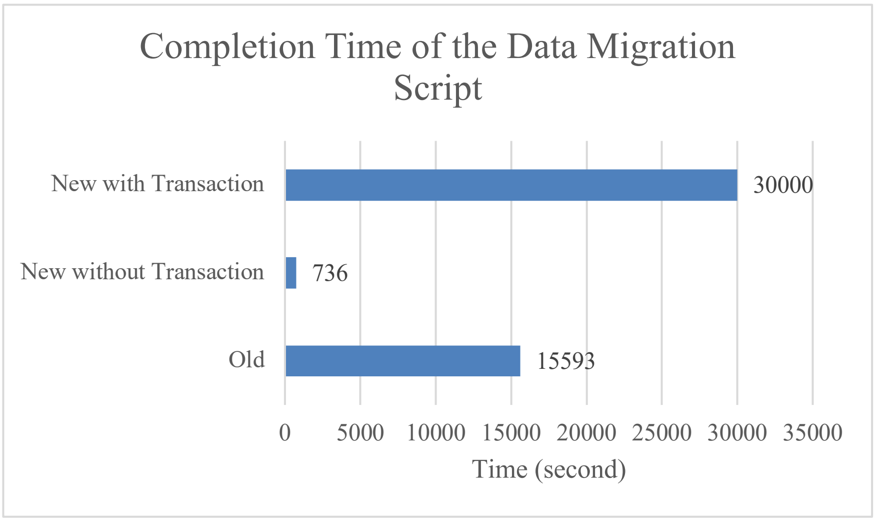


Figure 36

When the transaction context is introduced, due to the overhead of using transaction, the completion time is about 2 times worse than the old data migration script. Therefore, any developer should carefully decide whether a transaction should be used.

Before ending this section, it should be mentioned that the batch size for all new data migration scripts is set to be 100. It is possible to set it to a bigger number[22]. However, there is no much difference (as shown in Figure 37) when transaction is enabled because the biggest overhead is brought by the transaction itself.

[22] Due to a bug (https://stackoverflow.com/questions/41499505/objectify-queries-setting-limit-above-300-does-not-work/41509697#41509697) in GAE Remote API, the number cannot be set to more than 300.

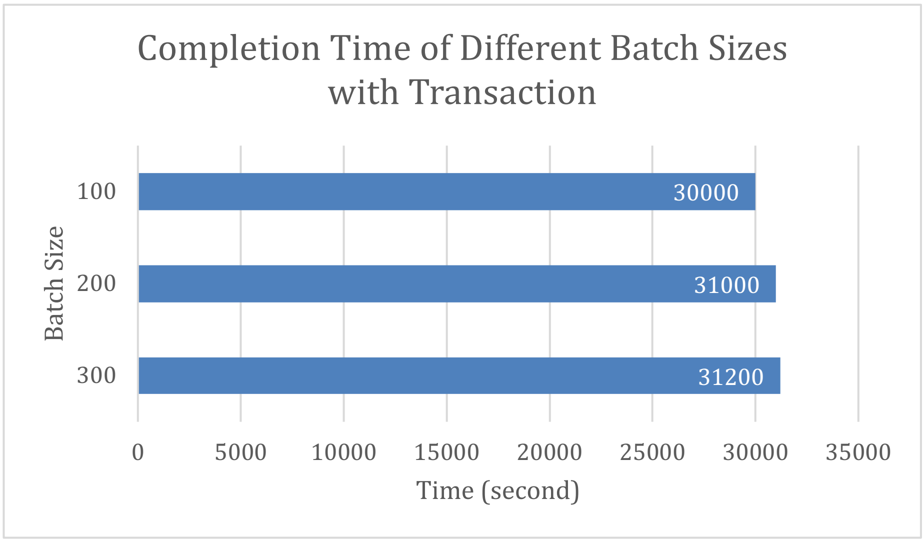


Figure 37

When transaction is disabled, there are improvements in completion time but with a tradeoff of data consistency.

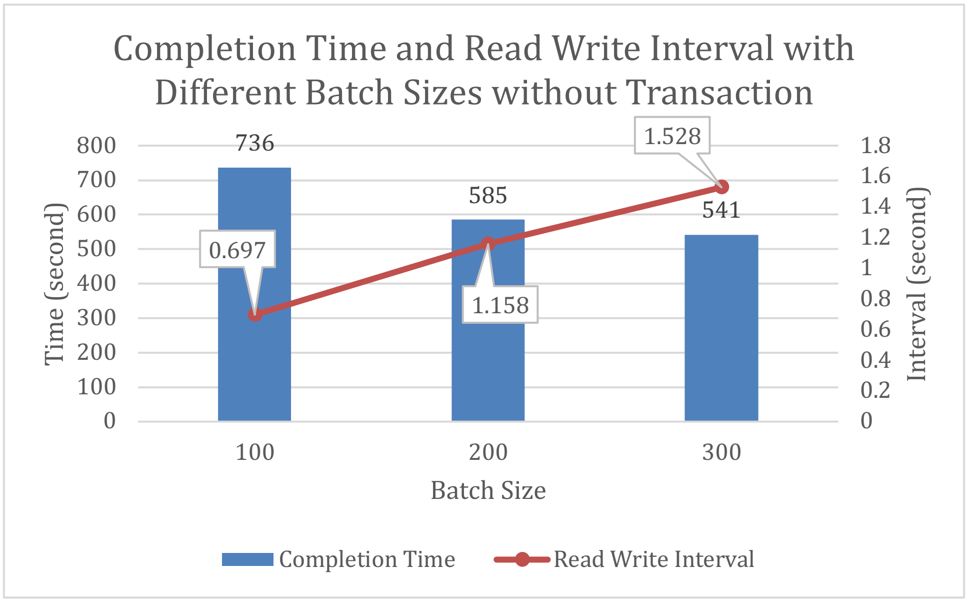


Figure 38

As shown in Figure 38, when the batch size is changed from 100 to 300, the risk of data inconsistency is around 2 times bigger while there is only 26% improvement on the completion time. Therefore, it is recommended to set the batch size to 100.

In conclusion, both the theoretical analysis and the practical benchmarking prove the correctness, scalability and performance of the new scripts. The redesign of the script package prepares TEAMMATES well for a million users.

5. Achievement 3: Improved Maintainability and Performance of the API Endpoints

5.1. Background

On January 29th, 2019, TEAMMATES V7.0.0-alpha.0[23] is released internally with 80% functionalities migrated from JSP to Angular. The frontend and the backend are separated. They communicate through HTTP requests and responses.

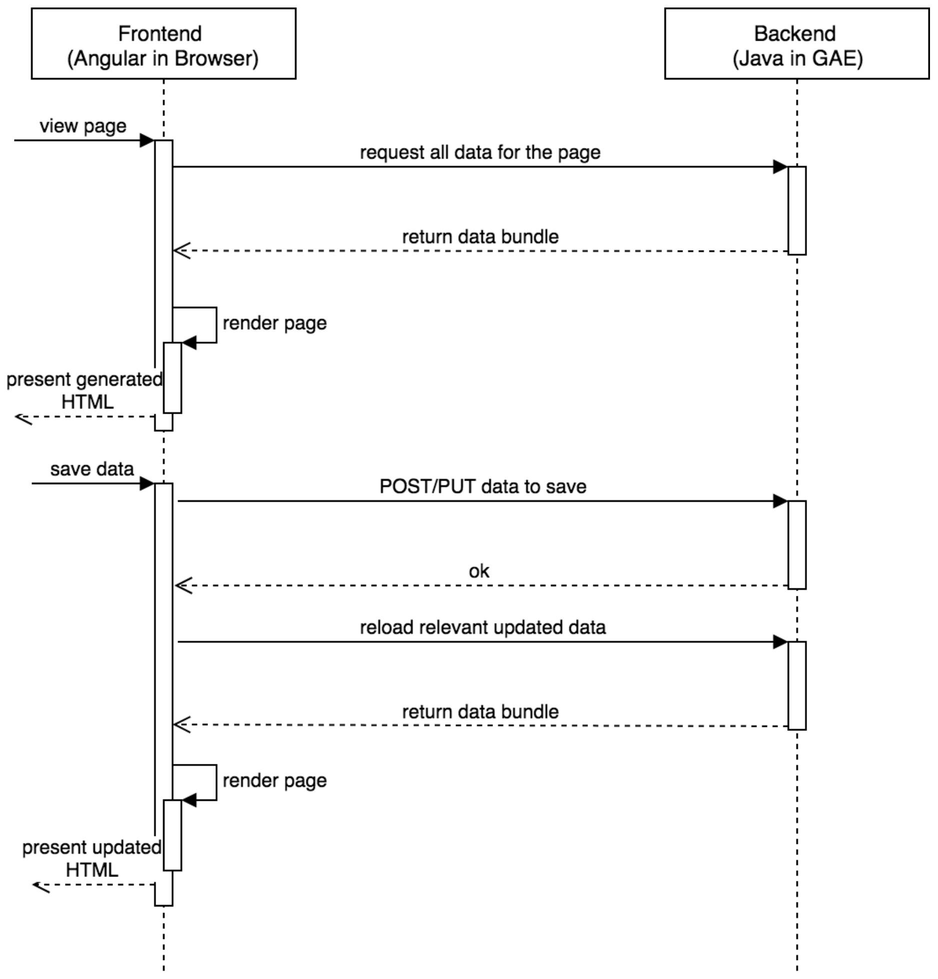


Figure 39

Figure 39 shows the interactions between the frontend and the backend[24]. The backend sends all data in JSON to the frontend and the frontend renders a page with the received data bundle.

[23] https://github.com/TEAMMATES/teammates/releases/tag/V7.0.0-alpha.0

[24] At that time, most interactions followed the pattern. There were several pages that followed different sequence diagrams.

When users trigger modifications of data, the frontend sends a `POST` or `PUT` request to persist the modifications and reloads all relevant data by issuing another request.

The process does not differ much from TEAMMATES V6 where server-side rendering is adopted[25].

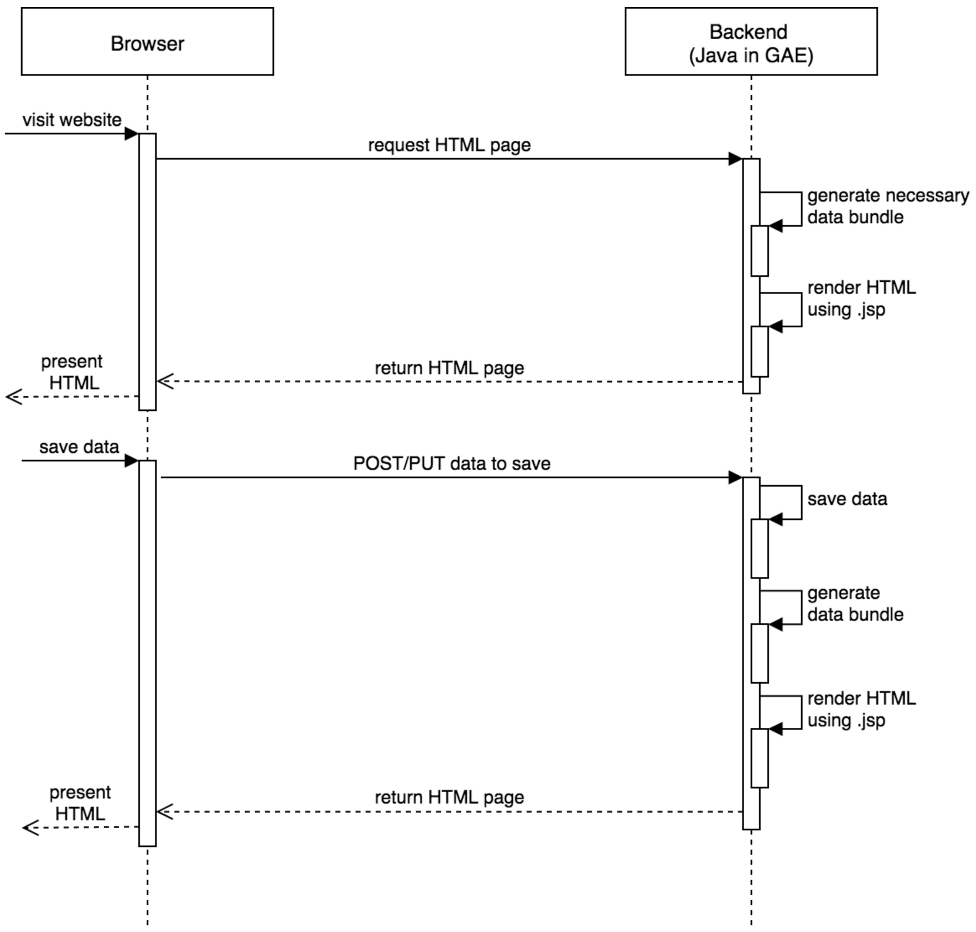


Figure 40

As shown in Figure 40, the rendering process is done on the server side. Instead of sending data in JSON format, the server sends fully-rendered HTML pages to the browser.

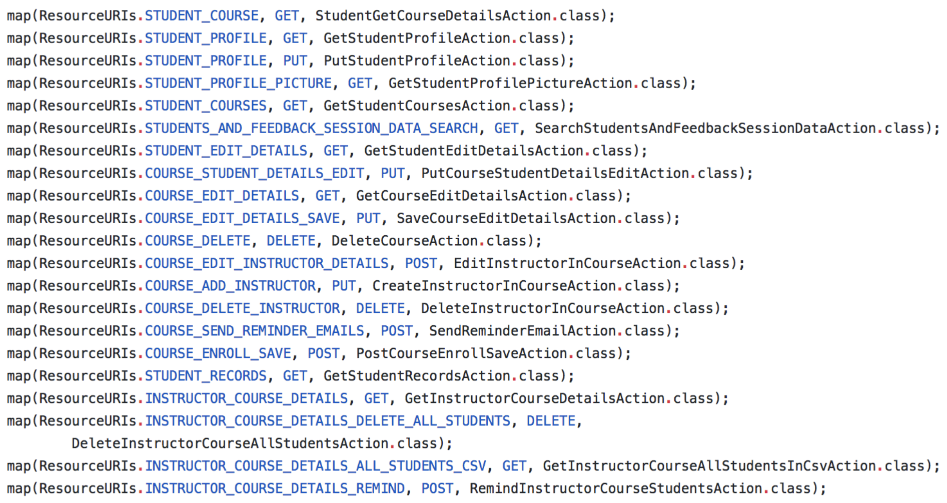
Thus, through the comparison, it is concluded that the frontend in V7 is just doing remote procedure call[26] (RPC) to the backend to get necessary data for rendering. HTTP is the medium

[25] The reason for changing from server-side rendering to client-side rendering is not within the scope of the report.
[26] It is also known as remote method invocation or remote procedure invocation.

used in the RPC. JSON and Uniform Resource Locator (URL) parameters are the representations of messages.

5.2. Problem Observed: Endpoints with Low Maintainability

It is found that the current endpoints are not maintainable because of the RPC way of handling client-server communication.



```
map(ResourceURIs.STUDENT_COURSE, GET, StudentGetCourseDetailsAction.class);
map(ResourceURIs.STUDENT_PROFILE, GET, GetStudentProfileAction.class);
map(ResourceURIs.STUDENT_PROFILE, PUT, PutStudentProfileAction.class);
map(ResourceURIs.STUDENT_PROFILE_PICTURE, GET, GetStudentProfilePictureAction.class);
map(ResourceURIs.STUDENT_COURSES, GET, GetStudentCoursesAction.class);
map(ResourceURIs.STUDENTS_AND_FEEDBACK_SESSION_DATA_SEARCH, GET, SearchStudentsAndFeedbackSessionDataAction.class);
map(ResourceURIs.STUDENT_EDIT_DETAILS, GET, GetStudentEditDetailsAction.class);
map(ResourceURIs.COURSE_STUDENT_DETAILS_EDIT, PUT, PutCourseStudentDetailsEditAction.class);
map(ResourceURIs.COURSE_EDIT_DETAILS, GET, GetCourseEditDetailsAction.class);
map(ResourceURIs.COURSE_EDIT_DETAILS_SAVE, PUT, SaveCourseEditDetailsAction.class);
map(ResourceURIs.COURSE_DELETE, DELETE, DeleteCourseAction.class);
map(ResourceURIs.COURSE_EDIT_INSTRUCTOR_DETAILS, POST, EditInstructorInCourseAction.class);
map(ResourceURIs.COURSE_ADD_INSTRUCTOR, PUT, CreateInstructorInCourseAction.class);
map(ResourceURIs.COURSE_DELETE_INSTRUCTOR, DELETE, DeleteInstructorInCourseAction.class);
map(ResourceURIs.COURSE_SEND_REMINDER_EMAILS, POST, SendReminderEmailAction.class);
map(ResourceURIs.COURSE_ENROLL_SAVE, POST, PostCourseEnrollSaveAction.class);
map(ResourceURIs.STUDENT_RECORDS, GET, GetStudentRecordsAction.class);
map(ResourceURIs.INSTRUCTOR_COURSE_DETAILS, GET, GetInstructorCourseDetailsAction.class);
map(ResourceURIs.INSTRUCTOR_COURSE_DETAILS_DELETE_ALL_STUDENTS, DELETE,
        DeleteInstructorCourseAllStudentsAction.class);
map(ResourceURIs.INSTRUCTOR_COURSE_DETAILS_ALL_STUDENTS_CSV, GET, GetInstructorCourseAllStudentsInCsvAction.class);
map(ResourceURIs.INSTRUCTOR_COURSE_DETAILS_REMIND, POST, RemindInstructorCourseStudentsAction.class);
```

Figure 41

Firstly, the number of endpoints is exploding. The data bundle returned by one RPC endpoint is very specific to the page shown in the frontend, which results in low reusability even though there is a high similarity between two data bundles. Due to this, the number of endpoints open in the backend is approximately proportional to the number of pages presented in the frontend. Figure 41 partially shows the mapping between the endpoints and the actions taken on the server side. There are around 30 such endpoints. As the project progresses, it is expected more endpoints will be added due to implementations of different features. Such a mass number of endpoints will become a significant maintenance burden.

Besides, the one-to-one mapping between the frontend pages and the backend endpoints indicates a strong dependency. It is ironic to have the coupling as the main objective of adopting client-side rendering is to separate the concern of presentation of data from business logic. A backend developer is forced to know how the data is used in the frontend, which is opposed to what TEAMMATES V7 wants to achieve — backend developers could have zero knowledge of the frontend.

Moreover, the interfaces of those RPCs are not uniform. Each endpoint has a specific format of data bundle. Some endpoints require data to be passed in terms of URL parameters while others read data from the request body. Besides, `POST` and `PUT` HTTP methods are used without any convention, which violates the RFC 7231[27] specification where `POST` and `PUT` methods carry meanings.

5.3. Solution: Migrate to RESTful Endpoints

Since setting the granularity of API endpoints to page level is problematic and result in low maintainability, a natural question to ask is whether we can lower the granularity to increase the reusability. If common data returned by the old endpoints or sent to the old endpoints can be extracted and grouped to new endpoints, the reusability of endpoints would increase and the number of them can be controlled. Therefore, the common resources provided by the backend are identified and it is expected that these resources will form the new endpoints.

Some inspirations can be gained from the logic data model shown in Figure 3 and it turns out that every entity type can be a resource provided by the system. A table is constructed to enumerate different resources required by the old endpoints. A partial table is shown as Table 1.

Resource Name	Related Old Endpoints
Student	COURSE_STUDENT_DETAILS_EDIT (PUT)
	STUDENT_RECORDS (GET)
	STUDENT_EDIT_DETAILS (GET)
	COURSE_STUDENT_DETAILS (GET)
Students	INSTRUCTOR_COURSE_DETAILS_DELETE_ALL_STUDENTS (DELETE)
	COURSE_ENROLL_SAVE (PUT)
	COURSE_ENROLL_STUDENTS (GET)
	INSTRUCTOR_COURSE_DETAILS (GET)

[27] https://tools.ietf.org/html/rfc7231#section-4.3

	COURSE_STATS (GET)
	STUDENT_COURSE (GET)
	INSTRUCTOR_STUDENTS (GET)
Course	INSTRUCTOR_COURSE_DETAILS (GET)
	COURSE_EDIT_DETAILS (GET)
	STUDENT_COURSES (GET)
	INSTRUCTOR_COURSES_PERMANENTLY_DELETE (DELETE)
	COURSE_EDIT_DETAILS_SAVE (PUT)
	INSTRUCTOR_COURSES (POST)

Table 1

After resources are identified, the next step is to define the manipulations of them. More specifically, the URIs, HTTP methods, API response formats and API request formats of the new endpoints need to be finalized. The architectural principles presented in Representational State Transfer (REST) is followed in this step as they are designed to help system that focuses on resources to address and transfer state of resources over HTTP (Rodriguez, 2008).

Richardson (2008) introduces the properties of REST by specifying different levels of *RESTFul*[28] maturity as shown in Figure 42.

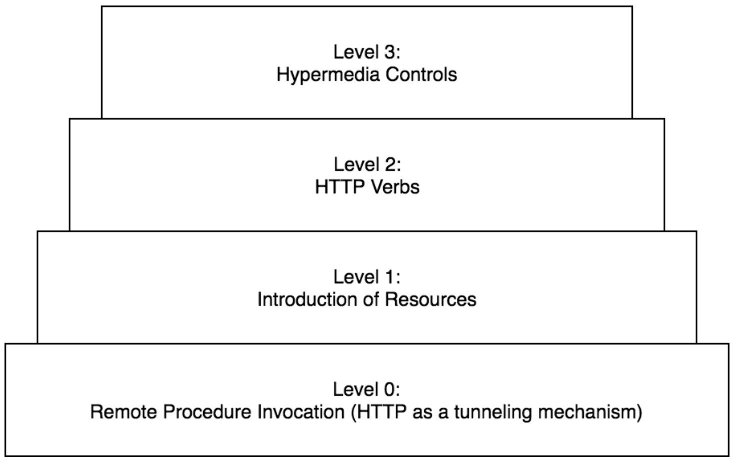

Level 3:
Hypermedia Controls
Level 2:
HTTP Verbs
Level 1:
Introduction of Resources
Level 0:
Remote Procedure Invocation (HTTP as a tunneling mechanism)

Figure 42 – Richardson Maturity Model

[28] When a service applies all REST principles, the service is a RESTFul service.

At level 0, HTTP is just used as a transport system for remote procedure invocation. This is exactly the current way of interactions between the frontend and the backend. We have shown that such design is not maintainable. At one step further, the resources used by the system are introduced. Endpoints are added according to the names of the resources. Thus, one more column is added to Table 1 and Table 2 is generated.

Resource Name	New Endpoint	Related Old Endpoints
Student	/student	COURSE_STUDENT_DETAILS_EDIT (PUT)
		STUDENT_RECORDS (GET)
		STUDENT_EDIT_DETAILS (GET)
		COURSE_STUDENT_DETAILS (GET)
Students	/students	INSTRUCTOR_COURSE_DETAILS_DELETE_ALL_STUDENTS (DELETE)
		COURSE_ENROLL_SAVE (PUT)
		COURSE_ENROLL_STUDENTS (GET)
		INSTRUCTOR_COURSE_DETAILS (GET)
		COURSE_STATS (GET)
		STUDENT_COURSE (GET)
		INSTRUCTOR_STUDENTS (GET)
Course	/course	INSTRUCTOR_COURSE_DETAILS (GET)
		COURSE_EDIT_DETAILS (GET)
		STUDENT_COURSES (GET)
		INSTRUCTOR_COURSES_PERMANENTLY_DELETE (DELETE)
		COURSE_EDIT_DETAILS_SAVE (PUT)
		INSTRUCTOR_COURSES (POST)

Table 2

After specifying the recourses' URIs, it is time to focus on the interactions within a resource, which leads to the level 2. Richardson's introduction to HTTP verbs is a bit outdated as HTML 5 was not the mainstream at that time. Masse (2012) gives a more detailed explanation of HTTP verbs, suggesting that HTTP methods should be used as verbs that define the actions within a resource. For example, `GET` can be used to retrieve a representation of a resource's state. `PUT` can be used to update a resource. `DELETE` can be used to remove a resource and `POST` can be used to create a new resource.

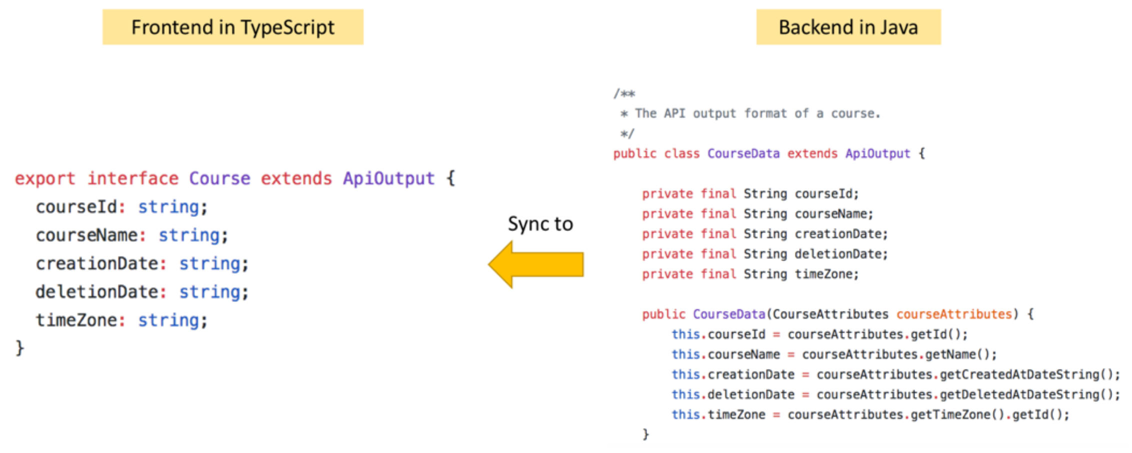
Resource Name	New Endpoint	Method	Related Old Endpoints
Student	/student	PUT	COURSE_STUDENT_DETAILS_EDIT (PUT)
		GET	STUDENT_RECORDS (GET)
			STUDENT_EDIT_DETAILS (GET)
			COURSE_STUDENT_DETAILS (GET)
Students	/students	DELETE	INSTRUCTOR_COURSE_DETAILS_DELETE_ALL_STUDENTS (DELETE)
		PUT	COURSE_ENROLL_SAVE (PUT)
		GET	COURSE_ENROLL_STUDENTS (GET)
			INSTRUCTOR_COURSE_DETAILS (GET)
			COURSE_STATS (GET)
			STUDENT_COURSE (GET)
			INSTRUCTOR_STUDENTS (GET)
Course	/course	GET	INSTRUCTOR_COURSE_DETAILS (GET)
			COURSE_EDIT_DETAILS (GET)
			STUDENT_COURSES (GET)
		DELETE	INSTRUCTOR_COURSES_PERMANENTLY_DELETE (DELETE)
		PUT	COURSE_EDIT_DETAILS_SAVE (PUT)
		POST	INSTRUCTOR_COURSES (POST)

Table 3

Table 3 shows the revised grouping of endpoints with HTTP verbs added. The table is finalized now. All old endpoints are to be removed in the backend and the frontend will use the new endpoints to retrieve resources. For example, instead of querying the old endpoint `INSTRUCTOR_COURSE_DETAILS` to retrieve the course details and all students, the frontend will make two separate `GET` request to `/students` and `/course` to retrieve necessary data.

There is one more level introduced in the Richardson Maturity Model. However, this migration from RPC to REST chooses not to follow it. According to Richardson (2008), the main benefit of including hypermedia controls is to tell the clients what can be done next and what is the corresponding URIs. Every response sent from the backend will include an extra link section for clients to browse. In doing so, the code written in client-side can be flexible or even generic if the backend adds, deletes or modifies any endpoint. In addition, frontend developers do not need to consult any documentation to understand the usage of endpoints as all instructions are included in the link section. Nonetheless, it is believed that hypermedia controls would add little values to TEAMMATES as a student project. The APIs provided in the backend are not made to the public and the only client is the frontend. Introducing hypermedia controls may not benefit anyone but overcomplicates the design of the backend. To improve the experience of frontend developers, an alternative approach is adopted as explained below.

JSON is used for resource representations in API requests and API responses as suggested by Masse (2012). On top of that, an automatic sync feature for the request and response formats between the frontend and the backend is introduced even one is written in TypeScript and the other is written in Java.

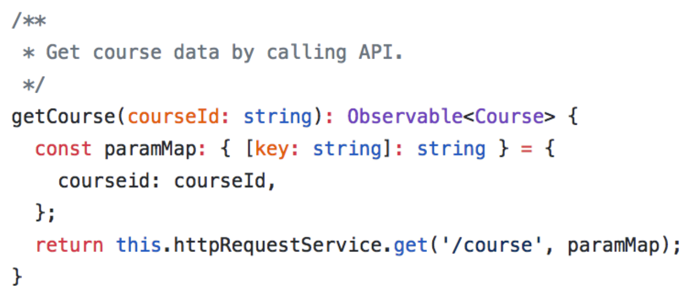


```
export interface Course extends ApiOutput {
  courseId: string;
  courseName: string;
  creationDate: string;
  deletionDate: string;
  timeZone: string;
}
```

```
/**
 * The API output format of a course.
 */
public class CourseData extends ApiOutput {

    private final String courseId;
    private final String courseName;
    private final String creationDate;
    private final String deletionDate;
    private final String timeZone;

    public CourseData(CourseAttributes courseAttributes) {
        this.courseId = courseAttributes.getId();
        this.courseName = courseAttributes.getName();
        this.creationDate = courseAttributes.getCreatedAtDateString();
        this.deletionDate = courseAttributes.getDeletedAtDateString();
        this.timeZone = courseAttributes.getTimeZone().getId();
    }
```

Figure 43

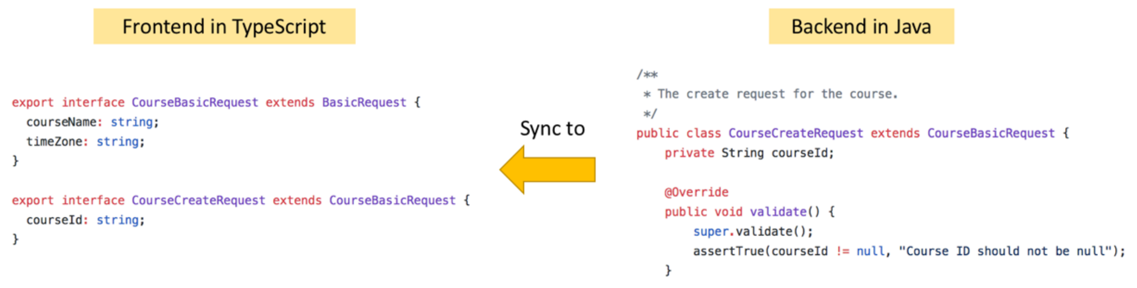
Figure 43 shows a typical example of how the representation of a course is synced between the frontend and the backend. Any changes in data format in the backend will be automatically reflected in the frontend. In addition, as shown in Figure 44, the API call to the backend is abstracted to a new method in the frontend which returns asynchronously, enabling the frontend developers to fetch data without knowing any details.



```
/**
 * Get course data by calling API.
 */
getCourse(courseId: string): Observable<Course> {
  const paramMap: { [key: string]: string } = {
    courseid: courseId,
  };
  return this.httpRequestService.get('/course', paramMap);
}
```

Figure 44

Like response format, the request format is also synced between the frontend and the backend as shown in the figure below.

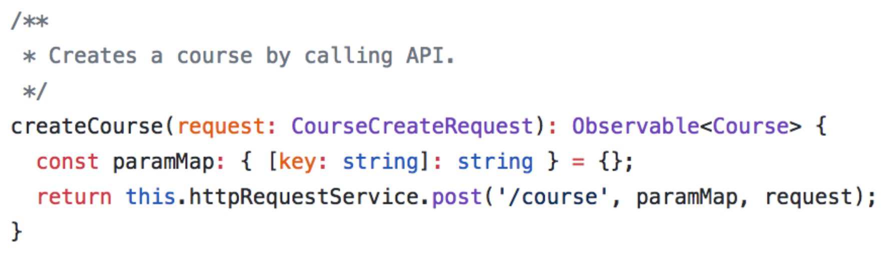


```
export interface CourseBasicRequest extends BasicRequest {
  courseName: string;
  timeZone: string;
}

export interface CourseCreateRequest extends CourseBasicRequest {
  courseId: string;
}
```

```
/**
 * The create request for the course.
 */
public class CourseCreateRequest extends CourseBasicRequest {
    private String courseId;

    @Override
    public void validate() {
        super.validate();
        assertTrue(courseId != null, "Course ID should not be null");
    }
```

Figure 45

A similar method is also constructed to provide abstractions for API requests. Without consulting any documentation, frontend developers can immediately know the format of an API request[29] as shown in Figure 46.



```
/**
 * Creates a course by calling API.
 */
createCourse(request: CourseCreateRequest): Observable<Course> {
  const paramMap: { [key: string]: string } = {};
  return this.httpRequestService.post('/course', paramMap, request);
}
```

Figure 46

It should also be noted that the API endpoints are designed to return the created or updated resource. The abstracted methods in the frontend follow the behavior by return the modified

[29] Unlike native JavaScript, TypeScript supports static checking of variable types.

resource asynchronously, saving any extra cost caused by fetching the created or updated resource. This behavior somehow is in alignment with the behavior introduced in chapter 3.

In summary, a set of concise, consistent and uniform resource-based new endpoints are built by following the REST principles.

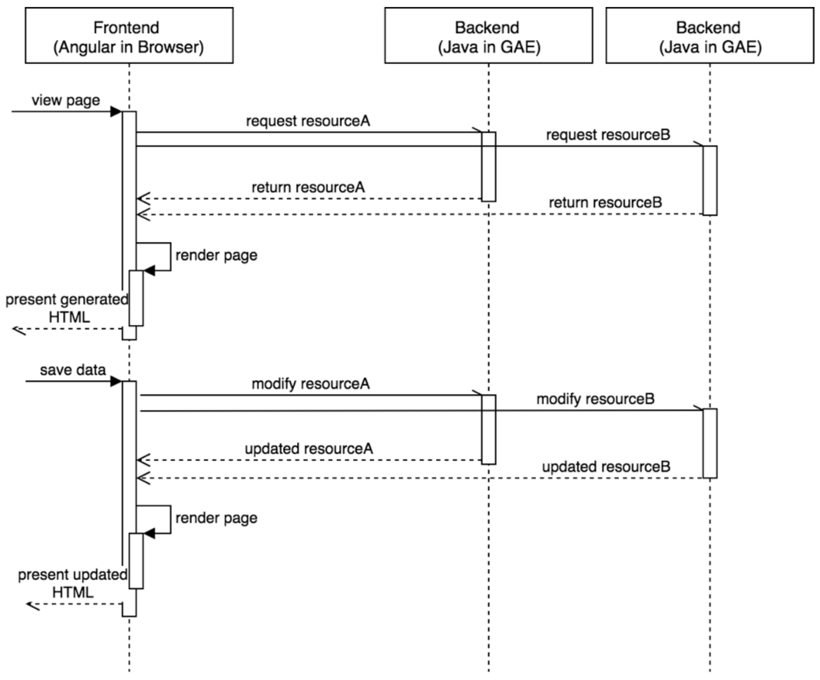


Figure 47

Figure 47 shows the new interaction between the frontend and the backend. Resources are fetched or modified asynchronously and combined in the frontend for displaying.

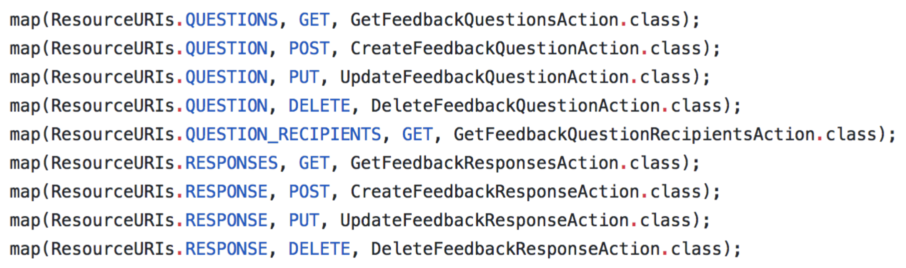
5.4. Results

The migrations are divided into several phases[30] and conducted together with CS3282 students. As of March 30, 2019, 80% of the migrations tasks are completed. Figure 48 partially shows the mapping for the new RESTful APIs.

It is obvious that the mapping becomes more readable and self-documenting compared to the previous mapping shown in Figure 41. The functionality of each endpoint can be easily

[30] https://github.com/TEAMMATES/teammates/issues/9420
https://github.com/TEAMMATES/teammates/issues/9494
https://github.com/TEAMMATES/teammates/issues/9564
https://github.com/TEAMMATES/teammates/issues/9595

understood by looking at the resource URIs and HTTP methods. Indeed, this is one of the benefits of RESTful APIs as discussed by Rodrigues (2008).



```
map(ResourceURIs.QUESTIONS, GET, GetFeedbackQuestionsAction.class);
map(ResourceURIs.QUESTION, POST, CreateFeedbackQuestionAction.class);
map(ResourceURIs.QUESTION, PUT, UpdateFeedbackQuestionAction.class);
map(ResourceURIs.QUESTION, DELETE, DeleteFeedbackQuestionAction.class);
map(ResourceURIs.QUESTION_RECIPIENTS, GET, GetFeedbackQuestionRecipientsAction.class);
map(ResourceURIs.RESPONSES, GET, GetFeedbackResponsesAction.class);
map(ResourceURIs.RESPONSE, POST, CreateFeedbackResponseAction.class);
map(ResourceURIs.RESPONSE, PUT, UpdateFeedbackResponseAction.class);
map(ResourceURIs.RESPONSE, DELETE, DeleteFeedbackResponseAction.class);
```

Figure 48

In addition, the performance of the application as a whole is improved. In the frontend, the waiting time for each action executed by users is shorter as multiple resources are fetched or modified in parallel. In the backend, the execution time for each endpoint is reduced because of the simplified logic for handling just one resource. Indeed, as documented by Google (2019), the shorter the execution time, the higher priority the application will gain in a physical server, which may result in lower response time.

The hypothesis for the improvement is also proven experimentally. In the interim report of this FYP project (Xiao, 2018), a benchmarking for the action of feedback submission was done and the throughput and the latency were recorded. The same scenario is set up and the benchmarking is also conducted for the migrated RESTful endpoints[31]. The comparisons are shown in Figure 49 and Figure 50.

[31] The raw experiment data can be found in Appendix B – RESTful Endpoints Benchmarking Data.

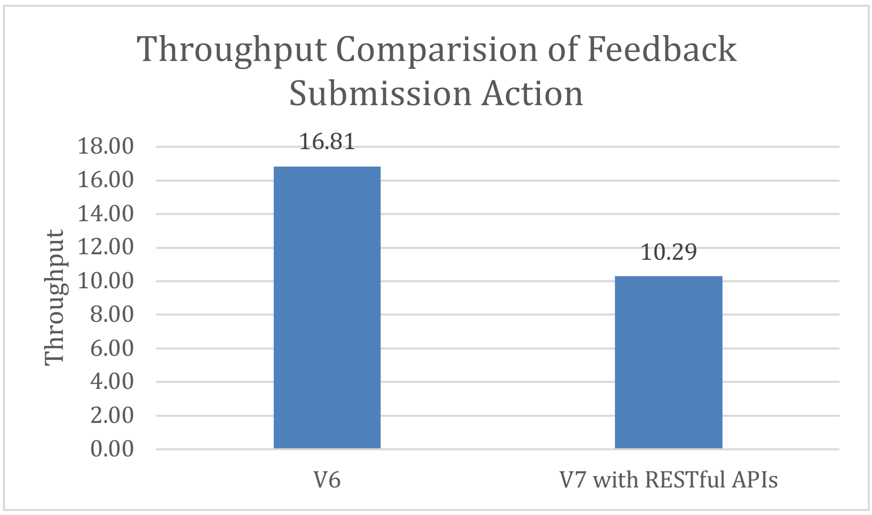


Figure 49

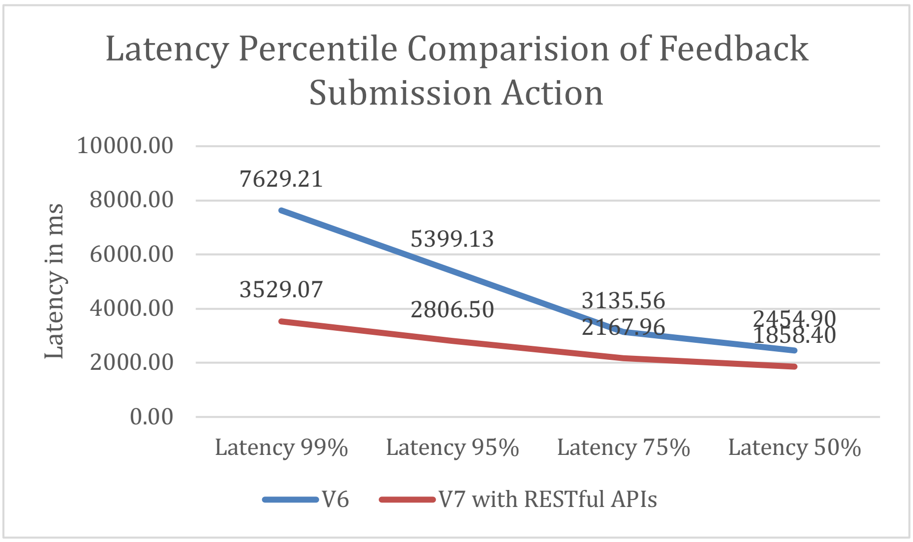


Figure 50

The latency gets significant improvements while the throughput decreases by about 38%. This can be explained by the fact that more HTTP requests are sent to the servers. In the scenario created in the benchmarking, the number of servers is fixed to 20. Thus, it is reasonable to guess that lesser submission actions will complete when the capability of handling HTTP requests for a single server is fixed.

The degraded throughput can be compensated by adding more servers. When the number of servers increases to 40, the throughput increases to 19.99, which is around 2 times bigger than the original result. The corresponding latency percentiles are almost not changed. It is true that more servers are needed to maintain the same throughput. However, compared to the dramatical improvements in latency, the cost is acceptable and is regarded as a trade-off to improve latency.

In summary, the quality of API endpoints, as well as the latency of actions, have been improved in this migration. Both the maintainability and the performance of backend endpoints have been well-prepared for entering the stage of a million users.

5.5. Future work

This migration only applies some essential REST principles as suggested by Masse (2012). More principles could be explored to improve the current design.

For example, the current way of identifying and filtering resources could be discussed. Taking the URI of getting a list of feedback sessions as an example, Masse recommends to use `/course/{courseId}/sessions?status=active` to identify and filter resources. However, due to the limitation of the framework used in TEAMMATES, the URI is set to be `/sessions?courseid={courseId}&status=action`. Thus, whether a new framework needs to be introduced to handle the URI conventions is worth discussing.

In addition, some improvements could also be made to the area of access controls. TEAMMATES requires some fields to be hidden in the API responses for some endpoints based on the authentication information. For example, both students and instructors can call `/session` to get the details of a session. However, a student is not permitted to see the grace period set by instructors. The application currently set the fields to hide to `null` in the API responses. However, this partially breaks the contract of response formats shown in Figure 43 where the frontend expects non-null fields. Therefore, there is a need to explore better design. Masse (2012) proposes the idea of partial responses in the chapter regarding response representation composition, which could be a possible solution.

6. Other Works

6.1. Migration from JSP to Angular for Three Pages

Migration from JSP to Angular is another big project in TEAMMATES besides this FYP project. As a core team member, I participated in the project by migrating three pages from JSP to Angular. The details can be found in the following links:

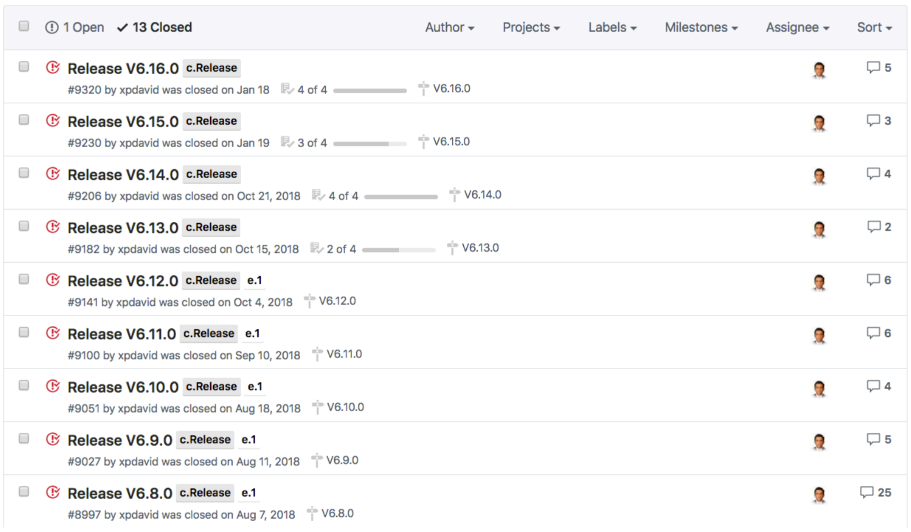
- https://github.com/TEAMMATES/teammates/pull/9345 (Migrate part of instructor feedback sessions page to Angular)
- https://github.com/TEAMMATES/teammates/pull/9339 (Migrate part of session submission page to Angular)
- https://github.com/TEAMMATES/teammates/pull/9281 (Migrate part of InstructorFeedbackEditPage page to Angular)

These experiences enrich my understanding of how the frontend interacts with the backend, contributing to some of the ideas presented in chapter 5.

6.2. Project Management

There are around 110 pull requests reviewed during this FYP project. They can be found on https://github.com/TEAMMATES/teammates/pulls?utf8=%E2%9C%93&q=is%3Apr+assignee%3Axpdavid+is%3Aclosed+closed%3A%3E2018-08-03. Part of them come from reviewing the frontend migration. Most of them come from mentoring the works done by CS3282 students and other developers.

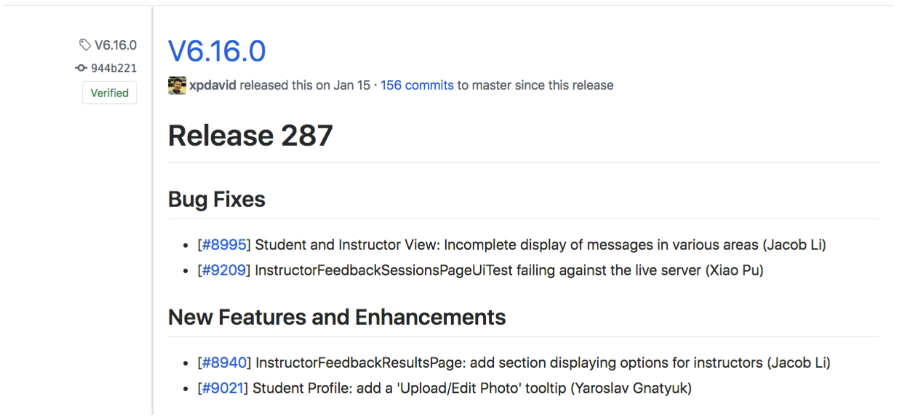
Releases are also created regularly for the project manager to deploy on the LIVE server.



1 Open ✓ 13 Closed — Author ▾ Projects ▾ Labels ▾ Milestones ▾ Assignee ▾ Sort ▾

- Release V6.16.0 c.Release — #9320 by xpdavid was closed on Jan 18 — 4 of 4 — V6.16.0 — 5
- Release V6.15.0 c.Release — #9230 by xpdavid was closed on Jan 19 — 3 of 4 — V6.15.0 — 3
- Release V6.14.0 c.Release — #9206 by xpdavid was closed on Oct 21, 2018 — 4 of 4 — V6.14.0 — 4
- Release V6.13.0 c.Release — #9182 by xpdavid was closed on Oct 15, 2018 — 2 of 4 — V6.13.0 — 2
- Release V6.12.0 c.Release e.1 — #9141 by xpdavid was closed on Oct 4, 2018 — V6.12.0 — 6
- Release V6.11.0 c.Release e.1 — #9100 by xpdavid was closed on Sep 10, 2018 — V6.11.0 — 6
- Release V6.10.0 c.Release e.1 — #9051 by xpdavid was closed on Aug 18, 2018 — V6.10.0 — 4
- Release V6.9.0 c.Release e.1 — #9027 by xpdavid was closed on Aug 11, 2018 — V6.9.0 — 5
- Release V6.8.0 c.Release e.1 — #8997 by xpdavid was closed on Aug 7, 2018 — V6.8.0 — 25

Figure 51 – Release tasks created



V6.16.0
944b221
Verified

V6.16.0

xpdavid released this on Jan 15 · 156 commits to master since this release

Release 287

Bug Fixes

- [#8995] Student and Instructor View: Incomplete display of messages in various areas (Jacob Li)
- [#9209] InstructorFeedbackSessionsPageUiTest failing against the live server (Xiao Pu)

New Features and Enhancements

- [#8940] InstructorFeedbackResultsPage: add section displaying options for instructors (Jacob Li)
- [#9021] Student Profile: add a 'Upload/Edit Photo' tooltip (Yaroslav Gnatyuk)

Figure 52 – A typical release created for the project manager to deploy

6.3. Bugs Fixed

The following are critical bugs fixed during the FYP project. The first number indicates the issue number and the second number shows the pull request number.

- [#9193] InstructorFeedbackEditPageUiTest failing on live server (#9194)
- [#9146] InstructorSearch: Error searching terms found in deleted sessions (#9147)
- [#9148] InstructorFeedbackSession: Fix data persistence for sessions with recycled names (#9153)
- [#9109] Deactivate deleted sessions (#9110)
- [#9090] Instructor edit email of student: send the correct link to student (#9097)
- [#9157] AssertionError: If the instructor hasn't been retrieved yet there is some problem in adding of instructor (#9159)

7. Conclusion

This FYP project successfully improves TEAMMATES in terms of performance, scalability, and maintainability, bringing TEAMMATES closer to the next stage of a million users. Several performance issues are identified and solved in the three enhancements. Any scalability concerns are eliminated in the script package. Lastly, the maintainability is enhanced with several iterations of refactoring and redesigning. The project also sees the learning progress of CS3282 students as new committers during code reviews and project management, preparing them well to be the next batch of core team members.

There are many lessons learned in this FYP project while the most important one is related to the quote by Kent Beck — "Make It Work, Make It Right, Make It Fast". It is this quote that explains why TEAMMATES does not follow the best design and consider performance tuning at the beginning. At the earlier stage of TEAMMATES, features and functionalities needed to be released as soon as possible to improve the user experience. However, after the project is stabilized, it is time to put efforts into making things right and fast. Indeed, that is development philosophy followed in this FYP project where the gap between making things work and making thing right and fast are addressed. In a word, Beck's words are worth following when developing large-scale projects such as TEAMMATES.

References

Cooper, J. (2009, December). *How Entities and Indexes are Stored*. Retrieved from Google App Engine - Documentation: https://cloud.google.com/appengine/articles/storage_breakdown

Google. (2019, January 31). *Google Cloud Datastore*. Retrieved from Google Cloud Datastore Overview: https://cloud.google.com/datastore/docs/concepts/overview

Google. (2019, January 7). *How Requests are Handled*. Retrieved from App Engine Documentaion: https://cloud.google.com/appengine/docs/standard/java/how-requests-are-handled

Google. (2019, March 23). *Limitations and Best Practices - Remote API for Java*. Retrieved from Google Cloud Documentation: https://cloud.google.com/appengine/docs/standard/java/tools/remoteapi#limitations_and_best_practices

Masse, M. (2012). *REST API Design Rulebook.* Sebastopol, CA: O'Reilly.

Richardson, L. (2008). *Act Three: The Maturity Heuristic.* Retrieved from Crummy: The Site: https://www.crummy.com/writing/speaking/2008-QCon/act3.html

Rodriguez, A. (2008). Restful web services: The basics. *IBM developerWorks, 33*, 18.

Xiao, P. (2018). *Preparing the Backend of a Large-scale Cloud Application for Million Users - Interim Report.* Singapore: B.Comp. Dissertation, National University of Singapore.

Appendix A – Optimized Saving Policy Benchmarking Data

Scenario:

- A course with ten sections, ten teams in each section, and eight students in each team is constructed.
- A feedback session with five questions is created for students to answer.
- HTTP POST requests are constructed randomly to simulate submission processes by students.
- It is found that 40 is the optimal concurrency. That is to say; there are 40 clients sending POST HTTP requests concurrently. Under such concurrency, the servers' capabilities to handle multiple requests are not underestimated. In addition, the servers are not overwhelmed.
- GAE is configured to use 20 instances with 256 MB memory and 1.2 GHz CPU.
- The benchmarking is designed to run 60 seconds.

It is noted that GAE needs time to warm up the instances. In addition, the more requests of a particular kind are, the more resources would be allocated by GAE to handle those requests. Due to this fact, all experiments are run until the results are stable enough. Three experiments with stabilized results are chosen. The average of them is taken to be the final result.

Data:

	Without Optimized Saving Policy	With Optimized Saving Policy
1st	Success (Throughput: 16.42): 985 Failure: 0 Latency 99%: 10021.65 ms Latency 95%: 7703.40 ms Latency 75%: 2694.33 ms Latency 50%: 1538.55 ms	Success (Throughput: 24.95): 1497 Failure: 0 Latency 99%: 6550.43 ms Latency 95%: 4819.73 ms Latency 75%: 2040.26 ms Latency 50%: 1002.03 ms
2nd	Success (Throughput: 17.68): 1061 Failure: 0 Latency 99%: 9626.40 ms Latency 95%: 7140.44 ms Latency 75%: 2536.56 ms Latency 50%: 1498.61 ms	Success (Throughput: 22.15): 1329 Failure: 0 Latency 99%: 8217.94 ms Latency 95%: 5755.00 ms Latency 75%: 2124.98 ms Latency 50%: 1199.87 ms
3rd	Success (Throughput: 19.58): 1175 Failure: 0 Latency 99%: 9078.08 ms Latency 95%: 5967.96 ms Latency 75%: 2344.81 ms Latency 50%: 1458.62 ms	Success (Throughput: 23.88): 1433 Failure: 0 Latency 99%: 7541.39 ms Latency 95%: 5497.32 ms Latency 75%: 1913.66 ms Latency 50%: 1039.66 ms

Final Result:

	Throughput
Without Optimized Saving Policy	17.89
With Optimized Saving Policy	23.66

	Without Optimized Saving Policy (ms)	With Optimized Saving Policy (ms)
Latency 99%	9575.38	7436.59
Latency 95%	6937.27	5357.35
Latency 75%	2525.23	2026.30
Latency 50%	1498.59	1080.52

Appendix B – RESTful Endpoints Benchmarking Data

Scenario:

- A course with ten sections, ten teams in each section, and eight students in each team is constructed.
- A feedback session with five questions is created for students to answer.
- HTTP POST requests are constructed randomly to simulate submission processes by students. Only requests that created new feedback responses will be sent and there are 800 in total.
- A optimal number of clients is configured. It is a point where the increasing number of clients does not improve throughputs but affect latency significantly.
- GAE is configured to use 20 instances with 256 MB memory and 1.2 GHz CPU.

It is noted that GAE needs time to warm up the instances. In addition, the more requests of a particular kind are, the more resources would be allocated by GAE to handle those requests. Due to this fact, all experiments are run until the results are stable enough. Three experiments with stabilized results are chosen. The average of them is taken to be the final result.

Data:

NC: Number of Client

	V6 (NC=40)	V7 with RESTful API (NC=20)
1st	Success (Throughput: 16.55): 800 Failure: 0 Latency 99%: 8220.97 ms Latency 95%: 5501.90 ms Latency 75%: 3172.44 ms Latency 50%: 2435.75 ms	Success (Throughput: 10.18): 800 Failure: 0 Latency 99%: 3550.82 ms Latency 95%: 2803.37 ms Latency 75%: 2183.27 ms Latency 50%: 1874.36 ms
2nd	Success (Throughput: 16.93): 800 Failure: 0 Latency 99%: 8025.76 ms Latency 95%: 5626.94 ms Latency 75%: 3034.51 ms Latency 50%: 2497.67 ms	Success (Throughput: 10.34): 800 Failure: 0 Latency 99%: 3604.00 ms Latency 95%: 2840.50 ms Latency 75%: 2129.53 ms Latency 50%: 1829.48 ms
3rd	Success (Throughput: 16.95): 800 Failure: 0 Latency 99%: 6640.91 ms Latency 95%: 5068.54 ms Latency 75%: 3199.73 ms Latency 50%: 2431.29 ms	Success (Throughput: 10.34): 800 Failure: 0 Latency 99%: 3432.39 ms Latency 95%: 2775.63 ms Latency 75%: 2191.09 ms Latency 50%: 1871.37 ms

Final Result:

	Throughput
V6 (NC=40)	16.81
V7 with RESTful API (NC=20)	10.29

	V6 (NC=40) (ms)	V7 with RESTful API (NC=20) (ms)
Latency 99%	7629.21	3529.07
Latency 95%	5399.13	2806.50
Latency 75%	3135.56	2167.96
Latency 50%	2454.90	1858.40

When increasing number of servers:

NS: Number of Server

	Throughput
V7 with RESTful API (NC=20, NS=20)	10.29
V7 with RESTful API (NC=30, NS=40)	19.99

	V7 with RESTful API (NC=20, NS=20)	V7 with RESTful API (NC=30, NS=40)
Latency 99%	3529.07	3427.80
Latency 95%	2806.50	2785.97
Latency 75%	2167.96	2098.37
Latency 50%	1858.40	1657.52