Architecture optimization for Google Course Builder on Google AppEngine

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Adam Lapworth

Tutor/es:
Pedro Agustín Pernías Peco
Luis Moreno Izquierdo
Architecture optimization for Google Course Builder on Google AppEngine

Autor
Adam Lapworth

Directores
Pedro Agustín Pernías Peco
Departamento de Lenguajes y Sistemas Informáticos
Luis Moreno Izquierdo
Instituto Interuniversitario de Economía Internacional

Grado en Ingeniería Informática

Escuela Politécnica Superior
Universitat d’Alacant
Universidad de Alicante

ALICANTE, 12 de septiembre de 2016
Preamble

In December of 2015 I started working as an intern at UniMOOC. I was initially employed as a content manager, with the task of creating and uploading the course content onto the server. Shortly after starting at the company, most of the employees where taken to Lucentia Labs, leaving UniMOOC without any programmers and with just me to manage all of the software development and server management. I quickly started developing time-saving features, such as a content management system (CMS) to reduce the time needed to upload the content onto the server. This allowed me to focus my attention on the main problems that riddled the platform: Server costs, errors, server timeouts, etc.

Shortly before my arrival, UniMOOC underwent a massive user surge after being publicized by ACTIVATE BY GOOGLE (see figure 1). This surge increased the number of daily users and page views, requiring increased server performance and additional AppEngine instances to respond to the requests. The server costs started escalating rapidly.

The server costs the month I started at UniMOOC where around 1.100€, a huge amount of money for what was being done on the server. I decided to explore options to make the platform more cost efficient otherwise it would not have lasted much longer. I started by solving all of the server errors (there where about 500 a minute) and then went on to optimizing the architecture of the whole platform to obtain a reliable, scalable and sustainable software system. This project is about the steps taken to carry out those
tasks: How I optimized the architecture of UniMOOC so that the company that had employed me would not die under its own success.

My experience at UniMOOC over this period and up to this day is what has motivated me to carry on improving and optimizing the platform. Without the support of the UniMOOC team, backing-up my every decision, I could not have worked under such stress.

Acknowledgements

Before dealing with the substance of the project, I would like to begin by thanking the current UniMOOC team, as well as all those who contributed to UniMOOC prior to my involvement.

Thanks to my great friend David, who was instrumental in recommending me to the UniMOOC team in the first place, and whose experience with Google AppEngine and Python has inspired me to improve my knowledge of both.

I would also like to thank my tutors Pedro and Luis for their help, advice and guidance, which has been invaluable to me in the conception, development and writing of this project.

Finally, I thank all those who have contributed to my training, from University teachers to anyone who has shared their knowledge and experience on the internet and who has been useful to me in one way or another.
Dedicated to my parents,
for their help and support at all times,
and for inspiring me to always be the best I can.
L’enfer, c’est les autres.

Jean-Paul Sartre
# Table of Contents

1. **Introduction**  
   1.1. Presenting the problem ........................................ 1  
   1.2. What is currently used? ...................................... 2  

2. **Analysis of UniMOOC**  

3. **Objectives**  

4. **Architecture**  
   4.1. Modularity .................................................. 9  
      4.1.1. What is modularity? .................................. 9  
      4.1.2. The solution for modular software ................. 10  
      4.1.3. The steps taken to achieve a modular design ...... 13  
   4.2. Data Store Management .................................... 16  
   4.3. AppEngine Configuration .................................. 20  
      4.3.1. What is AppEngine? ................................ 20  
      4.3.2. What can we configure? .............................. 21  
   4.4. Resulting Architecture .................................. 23  

5. **Results** ......................................................... 27  
   5.1. What metrics will be used? ................................ 27  
   5.2. Reduced Datastore reads .................................. 28  
   5.3. Fewer front-end instances ................................. 29  
   5.4. Overall cost .............................................. 30  

6. **Conclusions** .................................................. 33  

7. **Bibliography** .................................................. 35
## List of Figures

1. Growth of the number of registered students on UniMOOC.  
1.1. Examples of MOOC platforms.  
4.1. Software modularity and cost curve.  
4.2. Cocoa Model-View-Controller architecture.  
4.3. Model-View-Controller-Service architecture.  
4.4. Example of Course Builder’s bad modularity.  
4.5. Set of new defined modules.  
4.6. Example of models grouped in a module.  
4.7. Course Builder’s original Student entity definition.  
4.8. DB and NDB lookup speed difference for 1000 entities.  
4.10. Optimized Student entity definition.  
4.12. Initial Course Builder architecture.  
4.13. Final architecture zoomed out.  
5.1. UniMOOC’s registered students growth with marked results time period.  
5.2. Datastore reads throughout the test period.  
5.3. Total monthly instance hours.  
5.4. Gigabytes of bandwidth per month.  
5.5. Overall monthly server costs.  
5.6. Server cost per Gigabyte of bandwidth.
1 Introduction

1.1. Presenting the problem

The rise of new technologies in the last decade has contributed significantly to the improvement and growth of virtual and digital education. Within this field we find Massive Online Open Courses (MOOCs), open online courses with large numbers of students wishing to improve their knowledge on different topics. The fundamental difference between this model and others is the letter M in the acronym, which implies that there are many students enrolled on each MOOC. As defined by (Pernías and Luján-Mora, 2013), «the massive nature allows for certain dynamics to arise, and for certain activities to be offered than would not be possible with a more reduced number of students».

This large number of users challenges the efficiency and performance of the most commonly used MOOC platforms, such as UNIMOOC, UDACITY, COURSERA or ACTIVATE BY GOOGLE. These platforms boast a high degree of analysis, running large statistical algorithms on their user data for personalized interaction and gamification.

There are currently a huge number of MOOCs available on the multiple online platforms. It is estimated that there are more than 500 universities involved in the production of MOOCs, with more than 4200 courses and 35 million students that have been registered and have enrolled on at least one of them. (Shah, 2015).

Two of the MOOC platforms mentioned above, UNIMOOC and ACTIVATE BY GOOGLE, both use Google’s Learning Management System (LMS), Course Builder, which is an Open Source (Apache 2.0), online education platform written in Python running on Google AppEngine. The AppEngine environment of Google’s Cloud Platform running Python should allow for huge scalability and efficiency for thousands of users. But this is not the case with Course Builder. The design and architecture of Course Builder does not take advantage of AppEngine’s performance benefits like cached datastore requests and automatically scaling instances. Instead it uses static files to serve data and a low-modularized code which is hard to change and add to without significant reprogramming of the Course Builder core.
1.2. What is currently used?

To understand the performance and efficiency differences between the different platforms, we should look into what technologies and architecture the other platforms that do not use Course Builder use. By seeing how these MOOC giants design their platforms, we can better understand the need for an architectural upgrade for Google’s Course Builder.

The largest of the mentioned platforms, COURSERA, uses a services-oriented architecture (SOA) hosted on Amazon Web Services EC2. The service back-end is written in Scala and the Play Framework and they use Cassandra, S3 and MySQL for their data stores. This is one of the most optimum solutions to get the performance, reliability and scalability needed in order to manage thousands of students and generate a huge amount of statistical data. This services-oriented approach allows for different types of technologies to be used together for their intended purpose. In the case of COURSERA, using different data store technologies (relational and non-relational) gives them the best of both worlds, permitting them to have large amounts of data for statistical analysis as well as a well defined relational model for the more user-oriented interactions, not to mention improved reliability.

UDACITY also uses python and AppEngine for its back-end, boasting high scalability thanks to the AppEngine environment (although with its small hiccups every now and again) without needing to spend lots of time maintaining servers and services. They run Django architecture with a MongoDB (non-relational) data store for their statistics.

The open source platform EDX also primarily uses Python and Django for the back-end on their servers and MongoDB and MySQL for their data stores. This is similar to COURSERA’s setup, where they use the non-relational MongoDB for the course data and analytics data, and the relational MySQL database for the student data. This, as explained before, gives the advantages of both speed and queryable data.

As we can see then, the scalability of these MOOC platforms and their ability to analyse large amounts of statistics play a huge role in their success. The statistical analysis of these platforms is in fact one of the fundamental principles of their ability to
"tailor" the courses for each individual student, also offering personalized gamification and advertising. This factor is only made available by the «massive» quality of these online courses. With MOOCs, it is the first time big-data and statistics have been used in education to adapt the learning experience for each individual student.

The goal of this project is to explore a more optimized architecture which can be applied to the CourseBuilder platforms to take advantage of the performance benefits and scalability options that Google AppEngine offers. As well as performance optimization, we will also address maintainability and design issues with Google’s LMS.
2 Analysis of UniMOOC

Since the focus of this project is on changes made to the UniMOOC platform to improve cost effectiveness and performance, in this section I will examine how and why UniMOOC uses Google Course Builder.

UniMOOC started in 2012 as a MOOC course «oriented toward the fostering of entrepreneurship» hosted on Google Course Builder. The course was an open schedule course with continuously added activities and assessments.

When deciding what technology to use to develop UniMOOC, three options were contemplated: edX, OpenMOOC and Google Course Builder. The founders of UniMOOC decided on the Course Builder option because, compared with other LMS options available at that time, it was able to manage large numbers of students (in the thousands). Platforms such as Moodle and Sakai had been used to provide MOOCs but these options were discarded because of the platforms’ limits and scalability problems.

Another reason for choosing Google Course Builder over other technology options was the fact that it is a Google open source project. This means that the code can be used, changed and added to without having to buy a license or sign any contracts, as long as you abide by the Apache 2.0 regulations.

UniMOOC started off with 22,000 students, with an initial goal to reach 40,000 by the end of the course. But it did not stop there. Over the first two years, developments were made to improve some of Course Builder’s functionality and features, such as allowing for various different courses to be uploaded and made available to users rather than just having one course, which is what Course Builder was initially intended for. The number of registered students grew to almost 100,000 students, not enough yet to overly increase server costs, but enough to grab the attention of large companies and institutes with which to work.

As we saw in the previous chapter, the number of registered students grew considerably after the initial period. After being publicized by Google and striking agreements with new companies, UniMOOC could no longer carry on with the Course Builder technology without changes being made.
That said, the achievements of the UniMOOC team in this initial period were considerable, and I pay tribute to all of the people who have worked before me at UniMOOC and have contributed to turning it into what it is today. The developments made to Google Course Builder won UniMOOC various Google prizes and have given the platform the recognition it deserves.

As the Technology Director of UniMOOC, I am delighted to have the opportunity to contribute further to the improvement of this platform and to the Google Course Builder architecture as a whole.

In the following chapters I will explain the changes made to the Google Course Builder architecture and to UniMOOC that have allowed the platform to continue to grow, and I will examine the results obtained from the use of UniMOOC in the 6 months following the implementation of these changes.
3 Objectives

The general objective of this project is:

- GO1: Develop and validate an optimized architecture for Google Course Builder to improve scalability and efficiency within Google AppEngine.

The specific objectives are:

- SpO1: Improve the modularity of Google Course Builder to reduce duplicated code and improve code comprehension.
- SpO2: Reduce AppEngine costs.
- SpO3: Reduce Datastore reads and server costs.
- SpO4: Use new Datastore models to improve data access performance and efficiency.
- SpO5: Optimize Datastore entity models.
- SpO6: Apply the architecture in a real case (UniMOOC).

In addition, secondary objectives implicit to the development of the project are:

- SeO1: Apply an iterative software development methodology to the production of the project, not only the implementation.
- SeO2: Broaden Google AppEngine knowledge.
- SeO3: Use a non-relational database (GAE Datastore).
4 Architecture

I have divided the development of the architecture into three different phases. In the following sections I will explain the solutions I decided upon in order to achieve the specific objectives of the architecture optimization as well as the difficulties I encountered.

- Modularity.
- Data store management.
- AppEngine Configuration.

4.1. Modularity

4.1.1. What is modularity?

One of the key factors in software architecture and design is the modularity of the code and interfaces. Modularity is where software is divided into separately named components which are integrated to satisfy the functional requirements of the software system.

It has been stated that «modularity is the single attribute of software that allows a program to be intellectually manageable» (Myers 1978). Modularity makes our software easier to develop, especially when working in teams, because the development can be divided. Independent modules are easier to maintain and test because code modification within each module does not affect the rest of the modules. The propagation of errors and duplication of code is also reduced throughout the software system. Modularity is a key to good design, and good design is the key to software quality.

The concept of modularity brings us to one fundamental question: How do we divide software to get the best set of modules? The principle of information hiding (Parnas 1972) suggests that the best set of modules should be «characterized by design decisions
that (each) hides from all others». Basically, this means that each module should contain code and functions that return information that is information not needed in other modules. Therefore, we can obtain a set of modules that communicate with each other only the information needed to obtain the correct functionality of the software system. The idea around modularity is to design a set of modules where each module addresses a specific set of requirements and has independent interfaces.

Therefore, one might think that subdividing software indefinitely would make the effort and cost required to develop it increasingly lower. This however is not the case. As we can see in figure 4.1, the cost to develop software decreases as the number of modules increases, up to a point. As the number of modules grows, the cost required to integrate and maintain all of the modules also increases. This leads to a total cost curve, where M is the optimum amount of modules in which to divide our software to achieve the best cost, design and quality.

![Figura 4.1: Software modularity and cost curve.](image)

### 4.1.2. The solution for modular software

**Course Builder’s modularity**

In our case study, Google Course Builder with its added functionality for the requirements of the UniMOOC platform, attempts to be modular in certain aspects, with a clear set of modules and separated model files. The modules folder contains sub-folders for each module, which contain the code for the handlers and routing for each handler. This, however, is not always the case, and it is sometimes hard to find any type of correlation between some of the files and code within one given module.

As explained, effective modularity should make code more maintainable and testable. In the case of Course Builder, there does not seem to be any obvious reason why the software has been divided into its particular set of modules, which differ enormously in
terms of size and content. The lack of database models within the modules themselves means each module is no longer an independent functional unit and they can therefore not be developed in parallel. In short, we have a division of the system into modules without the benefits of software modularity.

**How to achieve modularity**

My solution to this was to greatly change the module set, re-designing the structure of the whole software system. The goal of this was to create easy-to-understand code, which is well divided into independent and coherent modules which communicate with one another to achieve the functionality of the whole software system. I started by analysing the models used by the system, looking at how they communicated and how to group them to obtain independent groups of models. The groups were then the basis on which to build the system’s structure. The grouping of the models is based on their function and use. One of the key factors used to group the different modules is by their datastore models. The datastore models used in each module play a key role in defining the data held within each module, therefore it makes sense to group models which have a common use or function so as to limit the need for communication between the models. For example, the BadgeEntity model and the IssuedBadges model should be grouped into the Badges module; the StudentProfileCache model, the StudentProfileInfo model and the ProfileBaseInfo model should be grouped into the Profile module. The result of this is an effective grouping of the models into independent modules.

With this definition of independently functioning modules, with independent datastore models and data use, we should be able to turn off and turn on modules without changing large amounts of code within the other modules. We will get self-contained functionality that does not depend on other code or data.

With the module set that had been defined based on the database models, as described above, I copied all of the handlers and routing code that managed those models into their corresponding modules, creating almost independently functioning modules which could be independently maintained and tested. The different application objects are defined using the Model-View-Controller pattern as introduced by Cocoa (see figure [4.2]), where the model-view communication is done through a controller.

The following step was to add service classes for the communication between the handlers (controller classes which handle the HTTP requests) and the models. These services are then imported by the services in the other modules to achieve inter-module communication without the need to import the models themselves (seen in figure [4.3]). These services are where all of the data access and data logic are managed. This way, different modules can share the same functions in the services, and we add a sort of pro-
tective layer over the database models themselves: all maintenance and possible changes or additions will be mainly on the services rather than on the database models.

Obviously, after a restructuring of the code of this magnitude, a thorough refactoring was required. Refactoring is a reorganization technique that simplifies the code of a software component without changing its behavior. Fowler [Fowler (1999)] defines refactoring as «the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure». To refactor software, we examine the code for redundant lines or functions, unused components and inefficient or poorly constructed algorithms and data models. The emphasis of this refactoring was to find commonly used or shared functions which could be optimized in the services to be used by all of the modules. The database models were also optimized and refactored, but these changes will be explained in more detail later in this project.
4.1.3. The steps taken to achieve a modular design

My solution to obtain a modular design, as explained above, consists of two steps: Defining the modules, and re-writing the code and refactoring to obtain functional modules. In this section I will provide examples of how these two steps were carried out.

Defining the modules

The indecipherable non-modular design of Course Builder can be seen in figure 4.4. As we can see, the modules do not have clear descriptive names and the «course explorer» module has a student file that contains the profile page handler. The structure of the system has no apparent logic and we can see no functional relationships at a glance, making maintenance of the code a somewhat time-costly task.

Following the steps explained in the previous section, the modules have been re-defined based on their function and datastore models. We can see this in figure 4.5 which shows the whole list of modules that have been re-defined.

In figure 4.6 we can see an example of the datastore models that have been grouped together into one module. These models are chosen because they all share a common use or only require one another to work correctly.

Refactoring the modules

As mentioned above, the refactoring process was focused on eliminating unnecessary code and finding commonly used functions to build the module services.
The following code shows the before and after of the refactoring process, and therefore the creation of the service, for the module used as an example in the last section.

```python
## --- Code Before ---

class StudentProfileInfo(db.Model):
    email = db.StringProperty()
    name = db.StringProperty()
    interests = db.StringProperty()
    scores = db.StringProperty()
    country = db.IntegerProperty()
...

class StudentBaseInfo(db.Model):
    sectors = db.TextProperty()
    studies = db.TextProperty()
...

class ProfileHandler(BaseStudentHandler):
    def get(self):
        email = self.request.get('email')
        profile_info = StudentProfileInfo.query(
```
```python
StudentProfileInfo.email == email).get()

## --- Code After ---

class UM_StudentProfileInfo(ndb.Model):
    email = ndb.StringProperty(indexed=True)
    name = ndb.StringProperty(indexed=False)
    interests = ndb.StringProperty(indexed=True, repeated=True)
    scores = ndb.StringProperty(indexed=False)
    country = ndb.IntegerProperty(indexed=True)

class StudentBaseInfo(ndb.Model):
    sectors = ndb.StringProperty(repeated=True)
    studies = ndb.StringProperty(repeated=True)

class StudentProfileInfoDAO(object):
    @classmethod
    def get(cls, email=None, country=None, interest=None):
        query = UM_StudentProfileInfo.query()
        if email is not None:
            query = query.filter(UM_StudentProfileInfo.email == email)
        if country is not None:
            query = query.filter(UM_StudentProfileInfo.country == country)
        if interest is not None:
            query = query.filter(UM_StudentProfileInfo.interests == interest)
        return query.fetch()

class StudentService(object):
    @classmethod
    def get_student_by_id(cls, student_id):
        return StudentProfileInfoDAO.get_by_id(student_id)

    @classmethod
    def get_student_by_email(cls, email):
        return StudentProfileInfoDAO.get(email=email)

    @classmethod
    def get_student_by_country_and_interest(cls, country,
4.2. Data Store Management

Data Access plays an important role in any software system and optimizing data access therefore has a significant impact on software performance and quality. As with any application using Google’s App Engine on the Google Cloud Platform, the native database that is used is the Google Cloud Datastore. In this section I am going to examine in detail the functionality of the Datastore and the optimizations that can be achieved.

Google Cloud Datastore is a NoSQL document database built for automatic scaling, high performance, and ease of application development. Cloud Datastore features include:

- Atomic transactions.
- High availability of reads and writes. Cloud Datastore runs in Google data centers, which use redundancy to minimize impact from points of failure.
- Massive scalability with high performance. Cloud Datastore uses a distributed architecture to automatically manage scaling. Cloud Datastore uses a mix of indexes...
and query constraints so your queries scale with the size of your result set, not the size of your data set.

- Flexible storage and querying of data.

- Balance of strong and eventual consistency. Cloud Datastore ensures that entity lookups by key and ancestor queries always receive strongly consistent data. All other queries are eventually consistent. The consistency models allow your application to deliver a great user experience while handling large amounts of data and users.

- Encryption at rest.

- Fully managed with no planned downtime.

This has been taken from the Google Cloud Datastore documentation for Python.

The most important features on this list for the optimization that I performed in order to achieve my goals are points 3 and 5: The scalability and high performance of the datastore, and the strong consistency of the lookups.

The main performance improvement that has been made to Course Builder, as well as drastically reducing the number of datastore reads, is to change the way the data in the Cloud Datastore is used and accessed. Course Builder uses the old `DB.Model` version of the datastore, whereas, in order to achieve the goals of this project, this has now been changed to the newer `NDB.Model` version.

The NDB client library builds on the older DB Datastore library, adding amongst other things, a new integrated automatic caching feature, providing fast and inexpensive reads with two levels of caching: in-context cache and Memcache. Despite the different APIs, NDB and the old ext.db package write exactly the same data to the Datastore. This means that no model conversion was required, and I was able to switch between NDB and ext.db code, as long as the schema was the same.

The in-context cache is kept only for the duration of each HTTP request. It is very very fast (in the order of microseconds) and is used to speed up reads and writes made within the same request. The Memcache is AppEngine’s standard caching service and although slower than in-context cache (milliseconds) it is still much faster than reading from the Datastore. All writes are made to the cache and any read that exists in cache is returned directly without making any Datastore interactions.
To reap the benefits of the automatic caching of the NDB.Model we have to reference the entities that we want to fetch using the correct key. This means that the entity has to be saved with the key that we are going to use for the majority of lookups for that particular entity. For example, as we can see in figure 4.7, the Student entity in the original model does not use a specified key, instead it uses an ID that is automatically generated by the datastore when the entity is created. However, the majority of the lookups for the Student entities are done using the student’s email property, with the result that it is not able to use caching and has to carry out a query over the whole datastore, which is far more inefficient and time-costly.

![Figure 4.7: Course Builder’s original Student entity definition.](image)

We can see in figure 4.8 that the NDB lookups are substantially faster than the non-cached Datastore queries using the DB model. This speed comparison has been obtained by doing multiple lookups for one thousand identical entities. We can see how the first NDB lookup takes longer than the rest, this is because at least on lookup has to be made to populate the cache.

![Figure 4.8: DB and NDB lookup speed difference for 1000 entities.](image)

The caching of the datastore, as well as allowing for faster and more efficient lookups, also reduces the cost of Datastore reads because the Cloud Platform Quotas are based on raw requests to the Datastore and do not count the cache hits (seen in figure 4.9). The implication of this is that a high percentage of cache hits dramatically reduces Datastore costs and speeds up client requests as a result of fewer time-costly Datastore reads, whilst also reducing the time taken to respond to clients and the number of server instance hours required to handle requests.

With this in mind, I started by changing all of the models in each module to inherit from the NDB.Model class instead of the older class. This in itself would not achieve the
desired improvements without making further changes to the models.

As we can see in figure 4.10, the Student model referred to above has now been changed to specify the email as the key in the `create` function of the Data Access Object (here seen as `<id=email>` when passed to the model constructor).

There have also been some design improvements that the NDB Client Library allows for, such as using a repeated property for the courses list instead of using an `ndb.StringProperty` as a JSON list. Using repeated properties within the Datastore saves the property as a Python List of the specified property type which is queryable by just one value of the type. This makes it faster and more efficient to find students with a particular course by querying the Student entity by course, rather than having to fetch all Student entities, loop over the list and parse the courses property to find a list of students that have that course.

Other changes that can be made to improve the data access performance is the use of batch requests. The NDB Client Library allows for the use of the `get multi` and `put multi` functions, which optimize the reading and writing of multiple Datastore entities.
at once without having to iterate through each entity. These functions reduce the time cost of reading and writing large amounts of data, so if they are used in the right places, this can greatly improve performance.

As mentioned above, the benefits of adopting these changes to take advantage of AppEngine’s features are reflected in both performance and server cost. Google is, at the moment, not charging for the use of the NDB’s cache and the Memcache services, allowing developers to take advantage of these features to reduce the number of Datastore reads (charged at 0.06$ for every 100 thousand entity reads). As well as this cost elimination, the server cost is further reduced by ensuring the application uses fewer cloud instances.

The main cost of an AppEngine hosted application is due to the cloud instances that have to be run to handle the requests from the clients. These Front-end instances are spawned when the available instances are full. By reducing the time it takes the server to access the data from the Datastore, we can make a large overall difference in the number of requests that can be handled by one given instance. By allowing instances to handle more requests, we reduce the number of instances that have to be spawned, therefore reducing the total cost of instances.

### 4.3. AppEngine Configuration

In this section we will see how to use AppEngine’s configuration files to reduce the total number of spawned Front-end instances as mentioned above.

#### 4.3.1. What is AppEngine?

AppEngine is a Platform as a Service (PaaS) and as such allows us to skip certain server configuration and installation steps and just focus on the development of the application itself. One might say that it comes ready to use out of the box. That said, there are some parameters that can be changed to greatly improve the way AppEngine works and performs.

In figure 4.11 we can see the Google AppEngine architecture and the different services the platform provides. Here we can see elements that have already been spoken about, such as the Cloud Datastore and the Memcache.
4.3.2. What can we configure?

The AppEngine Front-end app is the service that uses the «app.yaml» file to configure the auto-scaling options of the Front-end instances. This section will focus on the following options:

- Scaling type.
- Instance class.
- Maximum concurrent requests.
- Maximum and minimum idle instances.
- Maximum and minimum pending latency.

These options can be changed to affect how the application manages requests and how AppEngine spawns instances, with the aim of either reducing the server costs or increasing performance.
4.3. AppEngine Configuration

Scaling type

The scaling type of the application defines the way the instances are managed. We have 3 possible options: Automatic scaling, basic scaling and manual scaling.

Automatic scaling is the most customizable scaling type, allowing us to change all other options that are going to be explained below. It is the default scaling type and uses the Front-end instances that have been mentioned above. Basic scaling is similar to the automatic option but does not allow for as much customization. We can only change the instance class and the «idle timeout» which is the time AppEngine waits before shutting down an idle instance. The manual scaling option is the simplest of the three, only allowing us to specify the number of instances that should be run. There are no additional options and the application does not really scale, it simply runs the number of instances specified.

To achieve my goals, I have used the automatic scaling option as it gives us a wider range of options and more customization.

Instance class

The instance class is the type of instance that we can set our app to spawn, and it essentially determines the amount of memory and CPU allocated for each instance. This option directly effects the monthly cost of AppEngine. The instance class defines the quota we pay and the performance of each instance. The higher the level, the better the performance but the more we pay.

I changed this option to instance class F2 to increase instance performance and have more memory on each instance to handle more requests.

Maximum concurrent requests

This option defines the number of concurrent requests an automatic scaling instance can accept before the AppEngine scheduler spawns a new instance.

This option was changed from 8 to 32 concurrent requests, allowing us to take more advantage of the larger F2 instance class.
Maximum and minimum idle instances

The maximum and minimum idle instances defines exactly that: the maximum and minimum number of instances that can be kept idling. Both values affect the performance of the application in different ways. The maximum idle instances option can be used to limit the number of instances that are left to idle (they shut down after 15 minutes) and can therefore reduce costs if your application suffers user peeks that leave many instances idling. The minimum idle instances option, on the other hand, can be used to keep one or more instances constantly idling, waiting for new user requests. This will make your application seem faster to the end-user because you cut out the need to spawn new instances to handle new incoming traffic (the instances can take a few seconds to spawn properly). A compromise between both options is best, and this is why I set these values to 1 and 2 respectively, to always have one instance waiting but limit the idle instances to two.

Maximum and minimum pending latency

Finally, this option allows us to change the time the scheduler waits and holds a new request in the pending queue before spawning a new instance to handle the request.

Increasing the maximum pending latency will reduce server costs by spawning fewer instances, but having it set too high will dramatically effect the user-experience, making users wait before their request is processed.

Likewise, increasing the minimum pending latency will also reduce server costs, making requests wait longer if all instances are active but reducing performance.

These values should be set depending on our performance and cost requirements. In our case I wanted to reduce server costs but without effecting user experience, so I set the minimum pending latency to 50ms (from 30ms) and the maximum pending latency to 80ms (from the default automatic setting).

4.4. Resulting Architecture

The primary goal of this project is to develop and validate an optimized architecture for Google Course Builder. This section will illustrate the architectural changes referred to previously in this chapter using before and after diagrams of the architectural structure.
4.4. Resulting Architecture

The first diagram, shown in figure 4.12, shows a simplified version of the initial Course Builder architecture, demonstrating that there is no layer between the controllers and the models, which make direct interactions with the Datastore. A more descriptive diagram would show the many relationships between each controller and the many Datastore models available, but would be too complex and messy to represent properly.

In diagram 4.13 we can see how the business logic has been divided into corresponding modules as well as now using memcache interactions to speed up data access. Diagram 4.14 represents the interactions within each module (between the controller, service and data access object within the models) as well as the interactions between modules via the service layer. The app.yaml file is present in both the before and after diagrams but is only configured and optimized in the latter.
Figura 4.14: Final architecture zoomed in.
5 Results

In this chapter I am going to demonstrate how the changes that have been made to the Course Builder architecture achieve the project objectives and improve efficiency and performance on AppEngine.

5.1. What metrics will be used?

The metrics used to measure the improvements made to the architecture are obtained from the monthly usage summary of the UniMOOC application from AppEngine as well as some usage statistics that have been calculated on the UniMOOC platform. The time period used to obtain the results is from January 2016 to October 2016, which is the period of time in which the changes were made to the UniMOOC platform. I have chosen these months because January was the month after I arrived at UniMOOC and started developing for the company, and after October many more changes were made to UniMOOC’s functionality and architecture, changes which would significantly affect the metrics obtained if I had included subsequent months.

As UniMOOC was directly changed based on the architecture improvements described in this project, it is the perfect case study to obtain statistics which can prove the benefits of implementing these architecture improvements. The usage statistics that have been calculated on UniMOOC have been generated using map-reduce functions to obtain the number of daily registered students. This data can then be used to see how traffic has risen on the platform and how the number of students has grown significantly. This can be seen in figure 5.1 which shows how the number of registered students grows in the time period specified for these results.

AppEngine issues a usage report along with the billing export for each month. In this report, as well as including the price for each service used, they also include: the number of instance hours used by the application; the bandwidth used during that month; the total number of Datastore reads and writes. I am going to use this data to show how the improvements made have reduced server costs and Datastore reads, two of the specific objectives of this project.
5.2. Reduced Datastore reads

The number of Datastore reads the application makes is an important measure, influencing not only the cost of the Datastore reads themselves (0.06$ per 100 thousand reads) but, as explained in chapter 4, also reducing the number of spawned instances due to the performance improvements of using Memcache rather than normal Datastore queries. The change to the NDB datastore models and use of caching was implemented around the end of March 2016, as seen in figure 5.2. These relatively easily implemented changes reduced the Datastore reads by 94% (from approximately 313 million Datastore reads a month to approximately 19 million). The cost charged for this service therefore dropped by the same magnitude and the changes could also be seen in the number of total monthly instance hours, which dropped from an average of 10733 hours to an average of 9224 hours per month (a reduction of 14%), but this will be explained in more detail in the next section.

This graph clearly shows how using NDB models and efficiently accessing data, either by caching commonly used data or by using batch requests and key-only requests for
large amounts of data can drastically reduce the number of Datastore transactions and improve both efficiency and performance, therefore demonstrating how applying the proposed architecture achieves objectives SpO3, SpO4 and SpO5.

5.3. Fewer front-end instances

Since the cost of running any AppEngine application depends primarily on the number of spawned instances, reducing this number should be our key objective if our aim is to reduce costs. As can be seen in figure 5.3, by the end of our study period, the number of billed instance hours has reduced significantly, in spite of dramatically increased bandwidth 5.4 in the last two months (after the summer period), an indicator that although traffic increased significantly, server instance numbers remained low following the configuration changes made to the app.yaml file (as described in chapter 4) at the end of April of 2016.

As can be seen in these graphs, the improvements become obvious towards the end of the time period, where even with a higher amount of bandwidth than ever experienced on the UniMOOC platform, the number of instances still remains much lower than before.
As mentioned in the previous section, the total number of instance hours fell from an average of 10733 hours to 9224 hours (only a 14% reduction), but when compared to the bandwidth, which increased from an average of 10.2GB to an average of 28.9GB (not including the summer months) we can really see the scale of the improvement.

5.4. Overall cost

The overall monthly cost of an AppEngine application, as mentioned on numerous occasions in this project, depends mostly on the number of monthly instance hours. Therefore, the server cost graph should be very similar to that of the total instance hours. This can be seen in figure 5.5 which represents the monthly server costs, and which, when compared to figure 5.3 shows the almost identical monthly proportions. This cost graph shows a reduction of total cost from an average of 698$ per month to an average of 158$ per month. This represents a 77% total server cost reduction, proving that the objective SpO2 has been met.

![Figure 5.5: Overall monthly server costs.](image)

The bandwidth figures are a very good indicator of the amount of traffic received. Although not directly proportional, they can be used along with the number of instance hours to calculate a new metric: the server cost per Gigabyte of bandwidth, represented in figure 5.6.

This graph reinforces the results obtained, showing the correlation between the amount of traffic received by the server, and the server cost reduction experienced after the changes where implemented in May 2016.
Figura 5.6: Server cost per Gigabyte of bandwidth.
6 Conclusions

In this project I have examined the architectural problems inherent to Course Builder, motivated by my day to day work to find a more cost-efficient and scalable architecture to which to adapt the UniMOOC platform. In researching how the Google AppEngine platform works, I have found new ways to improve both efficiency and performance by making some relatively small changes to the way Course Builder is structured and how data is accessed from the Datastore.

The changes that have been made have been shown to be effective, having met the project objectives: server costs have been reduced by 77% and Datastore reads by 94%, at the same time as speeding up the end user experience. These reductions will allow the UniMOOC platform, and any other platform using Google Course Builder, to continue to grow without being held back by rapidly increasing server costs, costs which could eventually result in projects becoming financially unviable.

The modularity improvements allow the architecture to be expanded and upgraded more efficiently. Allowing Datastore models to be changed without the need to make changes throughout the whole software system, as well as allowing for new modules to be easily added-in for additional functionality to the platform.

Although these improvements fixed the cost and scalability issues with UniMOOC, the Course Builder architecture can still be greater improved by making many changes to the model definitions and by adding new features.

As the Technology Director at UniMOOC, for the last 7 months I have re-written UniMOOC from the ground up, re-defining models and functions to allow for greater performance improvements taking into account the AppEngine architecture. These changes have reduced server costs by another 70% and sped up user experience by about 80%. The new model definitions also allow for more statistical analysis of the data, by making statistics available through simple Datastore queries instead of needing to run map-reduce functions on all Datastore entities. This newly written platform is an big improvement on Course Builder. The platforms’ new styling, along with added payment and admin features, have greatly improved UniMOOC’s technology and have given us more options and features to offer our users and collaborators. The only down-side of
such a big change to the software system is the amount of data migration that is needed to deploy the new architecture, although this can be easily automated with some well tested migration functions.

Future developments to improve the architecture even more could be to use an additional relational database, as we have seen other large MOOC platform do. A relational database can be used to make certain course data and student data queryable which can have its advantages when it comes to dynamic data filters and searches. Also the use of session variables that could be stored in memcache could improve in-session data access to try and reduce the number of Datastore transactions made on each page load.

Another big improvement could be the use of dynamic front-end frameworks, such as AngularJS or even self-defined Vanilla Javascript, to reduce the amount of data and rendering the server has to process before returning a response to the client. A dynamic front-end would also reduce the number of Datastore transaction per page content change because the server would only have to process and return the data for the content that has been changed instead of having to reload the whole page and all of its data. This will most definitely be one of my future developments at UniMOOC.
Bibliografía


