Distributed Systems: Google App Engine session 1

Fatih Gey, Stefan Walraven, Arnaud Schoonjans,
Wouter De Borger and Wouter Joosen

November 29, 2016

Overview

There will be 3 exercise sessions on Google App Engine:

- 29/11/16 in pc lab:
  - Introduction to Google App Engine (GAE) and GAE datastore.
  - **Submit** your solution on Toledo **before Friday, December 2, 19:00**.
- 06/12/16 and 13/12/16 in pc lab:
  - Extend Google App Engine application with task queues and worker processes.
  - **Submit** your solution on Toledo **before Friday, December 16, 19:00**.

1 Introduction

Goal: The goal of this session is to familiarize yourself with building and running a Java application on Google App Engine (GAE) [1]. As a cloud platform, GAE provides high-scalability properties but also puts restrictions on the programming model and limits the set of statements and libraries that can be used. You will learn how to migrate a legacy Java application (the car rental application from previous exercise sessions) to GAE. Therein, the focus will be set on data persistence. The concepts and principles mentioned in the lesson will be the subject of this session.

Approach: This session must be carried out in groups of *two* people. You will have to team up with the same person for each of the sessions in this course.

Submitting: Before the deadline mentioned above, **each** student must submit his or her results to the Toledo website. To do so, close your Eclipse, enter the base directory of your project and zip your solution using the following command:

```
ant -Dzip.filename=firstname.lastname.zip
```

Please do not use a regular zip application to pack your entire project folder, as this would include the entire Google App Engine SDKs (about 40 MB). *Make sure that your reports (i.e. they should be placed in the base directory of your project), and addition libraries you solution made use of are also included in the zip file.* Submit the zip file to the Toledo website (under Assignments) before Friday, 19:00.

Important: When leaving, make sure your server is no longer running.

2 Web Applications

In this exercise session, we are going to deploy the car rental application (which has been introduced in the previous sessions) as a web application. This section briefly introduces topics that are relevant to build a web tier for the car rental application in Java. This web tier is deployed together with the business logic on GAE.

As web technologies haven't been introduced in the lessons and are not the focus of this course, we do not expect you to have in-depth understanding of those technologies. We will provide all necessary code that is related to the web tier of the application. However, *a basic understanding of web applications supports you in understanding the big picture* that this exercise wants to convey.

2.1 Web Applications versus Native Client Software

With the arise of the cloud computing paradigm for client-server architectured applications, we observe that the client parts tend to be implemented as web applications, instead or in addition to native client software.

The major benefit of a web application is that users need nothing more than a browser to access the application: no software installation is needed and the same version of the application is ensured for all users. However, such a web application is limited in functionality to what a web browser supports and permits. It also brings additional load to the server, e.g. converting output to common Web standards. In comparison, native client software (cf. Java RMI and Java EE sessions) is specifically suited for the user's operating system environment, is not limited to the browser's features, and runs on the client side which moves the overhead to the client.

2.2 HTTP Sessions

Web sites and web applications use the HTTP protocol to communicate between client and server. HTTP is a stateless protocol that employs a request-reply message pattern. Using a stateless protocol means that a web application can maintain *application state* but no *conversational state*. That is, all non-persisted data of a request is lost after the request processing has been finished, thus when the server has sent back the reply to the client.

Often, this absence of conversational state at the server side stands in conflict with application use cases in which data from a user request is needed along several HTTP requests[1]. For such cases, a workaround called *HTTP sessions* is known, that exists in various variations.

In our car rental application, we want to enable clients to make tentative reservations which need to be stored between client requests but do not affect the application state, i.e. a tentative reservation does not block a certain car from being booked by other clients. Thus, tentative reservations are perfect candidates to be put into a HTTP session. A simple example of using HTTP sessions can be found at [6].

Note: In order to store a Java object in a `HTTPSession`, it needs to be serializable.

2.3 Servlets and JSPs

The building blocks for web applications are web pages. The most basic way of building dynamic web pages in Java is by using Java Servlets and JavaServer Pages (JSPs). Both technologies are supported by Java EE. They complement each other and are meant for two different concerns of processing a HTTP request.

An application server (which serves the web pages) that receives a HTTP request, first triggers the Java Servlets. These are essentially ordinary Java classes whose purpose is to *control the application logic*, i.e. update application state or generate data to be sent back to the client. Next,

[1] For the sake of simplicity, here, we ignore techniques for client-side management of conversational state, i.e. data stored in Cookies

JSPs are processed, *rending the response* to be returned to the client. They are basically HTML-formatted files extended with embedded Java code. The embedded Java code should only serve the purpose of preparing the HTML output, e.g. by iterating through a `Collection` in order to create a HTML-viewable list such as a table (`<TABLE>`), an enumeration (`<OL>`) or a bullet list (`<UL>`).

3 Car Rental Application on Google App Engine

In this section, you first setup and test a Google App Engine project with the given code from Toledo. Next, the goal is to modify the car rental application so that it uses the GAE datastore to store its application state.

3.1 Preliminaries

On top of Google App Engine you can develop and upload Java applications for Google App Engine using the App Engine Java software development kit (SDK). This SDK includes software for a web server that you can run locally to test your applications. The server simulates all of the App Engine services, including a local version of the datastore and caching service.

GAE plugin for Eclipse. The GAE plugin [2] includes everything you need to build, test and deploy your application (including the App Engine SDK), entirely within Eclipse. We use version 1.9.34 of the App Engine SDK which requires Java 7.

1. Start the Eclipse IDE (with the GAE plugin pre-installed) by executing the following command: `/localhost/packages/ds/eclipse-gae/eclipse`

2. Go to `Window ▷ Preferences` in Eclipse. Verify the following settings:

   (a) Go to `Java ▷ Installed JREs`. Ensure that there is an entry called “java-7-openjdk-amd64” and that it is selected. Otherwise add it by clicking `Add... ▷ Standard VM`. Set `JRE home` to “/usr/lib/jvm/java-7-openjdk-amd64” and click `Finish`. Select the newly added entry.

   (b) Go to `Java ▷ compiler`. Ensure that the *compiler complience level* is set to 1.7

   (c) Go to `Google ▷ App Engine`. Ensure that the entry with version 1.9.34 is selected.

   (d) Click `OK`.

Extract source files to a new GAE project in Eclipse.

1. In Eclipse, create a new Google “Web Application Project” by using the Google icon in the toolbar, or by selecting `File ▷ New ▷ Web Application Project`.

2. The project name is `CarRentalGAE`, the package name is `ds.gae`.

3. Use Google App Engine (default SDK), but deselect “Use Google Web Toolkit”.

4. Deselect “Generate project sample code”. Finish the wizard.

5. Close the project.

6. Extract the source files from the `GAE1_src.zip` file into the new project’s base directory (i.e. merge folders and replace already existing files) using system tools. Do *not* drag-and-drop unzipped files *into Eclipse*. Do *not* use the Eclipse’s import functionality.

7. Open the project, hit refresh on the project’s context menu.

Deploy and run the application.

1. In Eclipse, run your project as "Web Application" by using the run icon in the toolbar, or by selecting `Run ▷ Run As ▷ Web Application`.
2. Open a browser and go to `http://localhost:8888/`.
3. You should see a web page listing all data entities of the car rental application. These are: 2 car rental companies (Hertz and Dockx), 7 different car types each, 46 (resp. 78) cars, and one or more reservations which are booked at Hertz.
   This view is a testing instrument for you to see whether all entities are accessible.

3.2 Assignment: Run Car Rental Application on GAE datastore

The goal of this assignment is to change the persistence strategy in the car rental application so that data entities, namely, the car rental companies including their car types, cars, reservations, etc. are stored in the GAE datastore.

- **Adapt the data entities to comply with the JPA API provided by GAE datastore [3].** Remember that the underpinning persistence system of GAE is the Megastore storage system, which relies on the Bigtable datastore. Although an abstraction layer is provided on top of this NoSQL datastore, this means that *entities, entity relations and queries will differ from those defined in the Java EE sessions.*
- To access data entities of the GAE datastore, use the singleton class `ds.gae.EMF` that is provided: It contains the `EntityManagerFactory` and should be used to create an *EntityManager* instance when needed.
- Modify the methods in `CarRentalServletContextListener`. This listener will be invoked just before the first user request. We will use it to load dummy data into the GAE datastore. In the methods `isDummyDataAvailable` and `loadRental`, remove the code that accesses static data entities and replace it with persistence queries (use JPQL as much as possible!). **Adapt the function `loadData` so that it applies to *your* entity structure.**
- **Modify and implement the methods of `CarRentalModel` so that they properly use persistence. Use JPQL as much a possible!** `CarRentalModel` is the interface to the application model accessed by the JSPs. It is possible that this requires modifications in other classes, too.

When all modifications are done, deploy the application (as described in Section 3.1) and check whether it starts without problems. Now, you should be able to call `http://localhost:8888` again and receive the same view as in Section 3.1.

Hints:

- Bigtable stores an entire entity group within a single row, in order to ensure serializable and yet fast transactions (as introduced in previous lecture). For that, the relations in an entity group must form a tree-structure-like hierarchy [4]: Each entity must be owned by at most one entity (the *parent entity*) such that a root entity (i.e. entity without any parents) contains all children recursively. Only that way, any changes in a entity can unambiguously be mapped to an entity group (and, by extension, to a Bigtable row).
- Google App Engine supports so-called cross-group transactions which provide transactional guarantees *across* entity groups. However, they are (i) expensive w.r.t. performance overhead and therefore (ii) limited to 25 entity groups.

- Bigtable on the one hand inherently provides transactional guarantees (ACID) for changes that affect a single entity group, even without employing JPA transactions. On the other hand, using JPA transactions does not overcome the limitations set by Bigtable.
- Megastore's support for JPQL JOINs is rather limited.

4 Assignment: Discussion

Provide answers to the following questions in English and store them in 1 PDF file called `session1.pdf`. Place that file in the main directory of your code (i.e. next to `build.xml`)

1. Motivate your solution that provides the all-or-nothing semantics when confirming quotes. For example, why did you (not) use cross-group transactions (XGT)? Did you rely on JPA transactions? What are limitations of your design? Justify your choice, assuming the following dimensions of a car rental agency service in practice: 32 registered car rental companies, and holding together more than 95 car types.
2. Let another, independent part of that rental agency application on Google App Engine manage data related to client profiles. These profiles –whose data model is shown in Figure 1– store data that is exclusively related to a single client, such as bills and car-rental preferences. Would you apply the same design decision in this case as in the previous question, i.e. how would you ensure data modification on client profiles? Provide a short motivation.
3. Consider the following two alternatives to implement transactional semantics of the first question: (i) relying on transactions only (i.e. this requires XGT) (ii) using application logic to role back if the quote confirmation (partially) fails. Discuss their difference in *locking* (concurrency control): How does each one prevent concurrent changes to a particular set of data?
    - Cloud applications typically aim at large-scale operation, facing many concurrently accessing requests. Yet, as a crucial success factor, high responsiveness must be ensured at all times. Which alternative of those you discussed above would you prefer?
    - **Bonus:** Can you link your arguments to (i) those type of trade-offs that you observed at actual internet application (i.e. please present "real-world" examples), and (ii) the popularity of NoSQL databases?

Hints:

- Be precise and to the point: Quantity does neither replace completeness nor correctness or quality of your answers. As a rule of thumb, 100 - 200 words should suffice to answer one of the bullet point above.
- To ensure that an answer is covering sufficient aspects, it is always a good idea to consult the slides of the lecture or to "google" on the subject. To that end, showing insights of the underlying concepts is a plus. However, avoid relying on keywords only and show that you understand those that you use.

5 Practical Information

Important:

- Check whether you have enough free storage space in your home directory (`quota`). **Lack of storage space results in strange errors!** To investigate why you are over quota, use the command `baobab`.
- If the console gives a warning about classes that are not enhanced, refresh your project (F5) and try again.

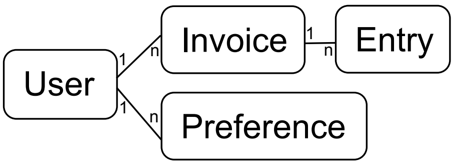

Figure 1: Data model to store client profiles.

Datastore:

- To inspect the data in the datastore, start the application and go to `http://localhost:8888/_ah/admin`.
- To restart your application with an empty development datastore, remove the files under `war/WEB-INF/appengine-generated/`. Refresh your project first!
- The JPA API provided by GAE only supports one object fetching strategy: *lazy loading*. That is, when loading a persisted object that contains persisted child objects, the child objects are not loaded at the same time as the parent is (eager loading) but on the first access to that child (lazy loading). The programmer has to ensure that at the time of the first "child object access" the parent object is still attached to the entity manager and that the entity manager connection has not been closed yet.

References

[1] Google, Inc. *Google App Engine.* `https://cloud.google.com/appengine/docs/java/`. October 2016

[2] Google, Inc. *Google App Engine SDK and Plugin for Eclipse.* `https://developers.google.com/eclipse/docs/getting_started`. October 2016

[3] Google, Inc. *Google App Engine Datastore, Java, JPA.* `https://cloud.google.com/appengine/docs/java/datastore/jpa/overview-dn2`. October 2016

[4] Google, Inc. *Google App Engine's Entities and EntityGroups.* `https://cloud.google.com/appengine/docs/java/datastore/entities`. October 2016

[5] Google, Inc. *Google App Engine Datastore, Java, Transactions.* `https://cloud.google.com/appengine/docs/java/datastore/transactions`. October 2016

[6] Desiderata Software. *JSP Tutorial, section JSP sessions.* `http://www.jsptut.com/sessions.jsp`. October 2016

Good luck!