Google App Engine: assignment 1

Deploying web-based distributed applications on a cloud platform

Practical arrangements

There will be 3 exercise sessions on Google App Engine:

1. Tuesday 24 November 2020:
    - Assignment 1 *(this assignment)*: Introduction to Google App Engine (GAE) and GAE datastore.
    - **Submit** your solution on Toledo **before Friday 27 November 2020, 19:00**.
2. Tuesday 1 December 2020:
    - Assignment 2: Extend Google App Engine application with task queues and worker processes.
    - No submission.
3. Tuesday 8 December 2020:
    - Assignment 2: Extend Google App Engine application with task queues and worker processes (continued).
    - Submit *(1) the report* and *(2) the code* on Toledo before Friday 11 December 2020, 19:00.

In total, **each** student must submit their solutions to the two GAE assignments to Toledo.

Submission: Before Friday 27 November 2020 at 19:00, **each** student must submit their results to the Toledo website. To do so, enter the base directory of your project and zip your solution using the following command:

```
mvn assembly:single
```

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 any additional libraries used in your solution are also included in the zip file.* Submit the zip file to the Toledo website (under Assignments) before the deadline stated in the overview above.

Important:
- This session must be carried out in groups of *two* people. Team up with the same partner you collaborated with in the previous sessions.
- When leaving, make sure your server is no longer running.

Good luck!

1 Introduction

Recent years have seen a major shift towards cloud computing: outsourcing required computing services to a third party. Three models exist: IaaS platforms (e.g. Amazon Web Services[1]) that provide bare computing power, PaaS platforms (e.g. Google App Engine) that provide a middleware, and SaaS platforms (e.g. Salesforce[2]) that run a specific application. In this assignment, we focus on Google App Engine as a PaaS platform for running custom web applications in a highly scalable manner.

Goal: The goal of this session is to familiarize yourself with building and running a Java web application on Google App Engine (GAE) [1]. The high-scalability properties of GAE also put some restrictions on the programming model, and limit 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; our focus is on the changes in data persistence.

2 The Car Rental Application

2.1 Rental agency

We continue with a rental agency that centralizes the data from all rental companies and acts as the sole point of interaction for car renters. The only change is that data is now stored in the GAE datastore. In order to reduce implementation effort, we have discarded the manager session: data is therefore loaded statically when the agency server starts.

2.2 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 together with the business logic is then deployed on GAE.

As web technologies haven't been introduced in the lectures and are not the focus of this course, we do not expect you to have an in-depth understanding of these 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.

2.2.1 Web Applications versus Native Client Software

With the rise of the cloud computing paradigm for client-server architectured applications, we observe that the client parts tend to be implemented as web applications, instead of 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.

2.2.2 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

[1]`https://aws.amazon.com/`
[2]`https://www.salesforce.com/`

of a request is lost after the request processing has been finished, i.e. 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 multiple HTTP requests. While techniques exist for client-side management of conversational state (e.g. data stored in cookies), we will use server-side `HttpSessions` [7]. In this model, the server stores the data contained within the `HttpSession`, for which the client receives an identifier that it then sends alongside each request.

In our car rental application, we want to enable clients to make tentative reservations that 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 `HttpSession`. A simple example of using these sessions can be found at [6]. **Note**: it is a good practice to only store Serializable Java objects in a `HttpSession`, as this allows sessions to be written to disk or migrated to another JVM for performance or availability reasons.

2.2.3 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). Java EE supports both technologies. 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, JSPs are processed, *rendering 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 Assignment

3.1 Implementation

The goal of this assignment is to change the persistence strategy in the car rental application so that data entities, being the car rental companies including their car types, cars, reservations, etc., are stored in the Google Cloud Datastore (Cloud Firestore in Datastore mode).

- **Adapt the entities so they can be stored in the Cloud Datastore**. Unlike JEE, there is no support for automatic loading objects from the database and populating all the fields. You will have to do load and save them yourself by using the Cloud Datastore API [3].
- Query support is limited. However, make sure to **use the datastore API [4] as much as possible**, as this is more efficient than retrieving all objects from the database, loading them in memory and processing them separately.
- 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 Cloud Datastore. In the methods `isDummyDataAvailable` and `loadRental`, remove the code that accesses static data entities and replace it with persistence queries.
- **Modify and implement the methods of `CarRentalModel` so that they properly use persistence.** `CarRentalModel` is the interface to the application model accessed by the JSPs. It is possible that this requires modifications in other classes, too. You may leave out the implementation for `confirmQuotes` for this assignment; however, it will be required in the next assignment.
- **Use transactions [5] where synchronization is necessary**, *but* only use transactions when you really need them.

When all modifications are done, deploy the application (as described in Section 4.1) and check whether it starts without problems. Now, you should be able to call `http://localhost:8080` again and receive the same view as in Section 4.1.

Hints:

- Make sure you import the APIs from `com.google.cloud.datastore` and *not* the deprecated ones from `com.google.appengine.api.datastore`.
- You can get a reference to the Datastore using `DatastoreOptions.getDefaultInstance().getService();`
- Cloud Datastore is a NoSQL database where relations are limited to parent ↔ child relationships. 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.
- You are allowed to change the signature of the given methods or add new methods. You will also need to remove some variables from the Entity classes. For example, you will need to remove the `cars` variable from `CarRentalCompany` and instead model this with a relationship.

4 Getting up and running

In this section, you first setup and test a Google App Engine project with the given code from Toledo. You can then complete the assignment of modifying the car rental application so that it uses the GAE datastore to store its application state (Section 3).

4.1 Preliminaries

You can develop and upload Java applications for Google App Engine using the Google Cloud 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.

Maven. Apache Maven[3] is a tool for automating the build and execution process of your application, similar to Apache Ant used in the RMI sessions. In contrast to Ant, Maven also includes dependency management, which allows a developer to declare all components (libraries) that a project needs, where Maven can then automatically download the required packages. We provide a Maven configuration to set up the GAE development server (i.e., automatically download and install all dependencies).

Maven is based around the concept of a *build lifecycle*, which contains several lifecycle phases such as `validate`, `compile`, `test`, `package`, `verify`, `install`, and `deploy`. Next to the default phases, plugins can also define *goals* (similar to Ant tasks) which can either be associated with a specific phase, or executed independently. An example of such a goal is `assembly:single` which you can use to zip your solution: `assembly` is the name of the Maven plugin, and `single` is the goal, defined by the plugin. You can execute this goal separately; it is not linked to any specific phase in the build lifecycle.

Getting started.

1. Download `ds_gae1_src.zip` from Toledo.
2. Import the source files from `ds_gae1_src.zip` into your IDE as a Maven project. In Eclipse, go to `File ▹ Open Projects from File System`. Click `Archive` and select the `ds_gae1_src.zip` file. Eclipse will automatically recognize it as a Maven project and use the `pom.xml` file to fetch all dependencies.
    - **Computer labs**. In the computer labs, you can use the preinstalled Eclipse IDE:

      ```
      /localhost/packages/ds/eclipse-gae/eclipse
      ```

    - **Your own PC**. If you want to develop your code on your own machine, check the 'Bring Your Own Device Guide' on Toledo. We require you to use Java SE *Development Kit* (JDK) version 8; verify that you use the JDK instead of the JRE. In Eclipse, go to `Window ▹ Preferences`. Go to `Java ▹ Installed JREs`. Ensure that there is an entry referring to `jdk1.8` (*not* `jre1.8`) and that it is selected. Otherwise, add a new entry, pointing to the location of the correct JDK on your computer. Click `Apply and close`.
3. Execute the `clean` Maven goal. You only have to run this once to install the GAE Maven plugin.

[3] `http://maven.apache.org/`

- **Eclipse**. In Eclipse, you can right-click on the `pom.xml` file and use `Run ▹ Run as ▹ Maven build...`, and specify `clean` as goal. Now click `Run` to execute the goal.
- **IntelliJ**. In IntelliJ, you can open the Maven tool in the sidebar and double-click on the `clean` goal under `Lifecycle`.
- **CLI**. If you have installed Maven, you can also run it from the command line using the following command: `mvn clean`.

Deploy and run the application. Execute the `package` Maven goal to package your Google App Engine application into a WAR file. This goal will execute the build lifecycle phase `package` and all preceding phases (`validate` and `compile`). Next, execute the `gae:run` Maven goal to run the Google App Engine application. The `gae:run` goal will start a local Cloud Datastore emulator and run your application in a local development server.

You can execute the goals in the same way you ran the `clean` goal. In Eclipse, you can also bundle these goals by specifying both in the same `Run Configuration` separated by a space. Note that the first time you run these commands this might take a while, as all dependencies including the Google Cloud SDK will be installed.

After executing the `gae:run` goal, you can now open a browser and go to `http://localhost:8080/`. 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 and 78 cars respectively, 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. To inspect the data in the datastore, go to `http://localhost:8080/_ah/admin`.

Execute the `package` goal each time you change the code, and execute `gae:run` so it uses the newly built WAR file. If you only want to restart the App Engine server, without any code changes, you only have to execute `gae:run`.

4.2 Practical Information

Important:

- Check if you have enough storage space in your home directory via `quota` and `du | sort -n`. **Lack of storage space results in strange errors!**
- If Eclipse does not shut down cleanly, there may be lingering server processes. You may get `java.io.IOException: Failed to bind to /0.0.0.0:8080`. Use a task manager to kill old `java` processes running `com.google.appengine.tools.development.DevAppServerMain`.

References

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

[2] Google Inc. *Google Cloud Tools for Eclipse*. `https://cloud.google.com/eclipse/docs/quickstart`.

[3] Google Inc. *Entities, Properties, and Keys*. `https://cloud.google.com/datastore/docs/concepts/entities`.

[4] Google Inc. *Datastore Queries*. `https://cloud.google.com/datastore/docs/concepts/queries`.

[5] Google Inc. *Transactions*. `https://cloud.google.com/datastore/docs/concepts/transactions`.

[6] JavaTPoint. *JSP Tutorial, section JSP sessions*. `https://www.javatpoint.com/session-implicit-object`.

[7] Oracle. `HttpSession`. `https://javaee.github.io/javaee-spec/javadocs/javax/servlet/http/HttpSession.html`.