Distributed Cloud Applications

Final project: service-oriented distributed applications on a cloud platform

---

Practical arrangements

The remaining weeks are allocated for the final exam project on cloud applications. You will complete this project in groups of four students. The deadline for submission on Toledo is June 18th, 23:59 CET. You will need to defend your solution with your full team during the exam period at the end of June (June28th-29th). Register your team on Toledo in the Distributed Applications course as soon as possible.

The defense will take place in the days after your written theory exam, such that you have processed the learning material of the lectures before the oral defense. This defense will consist of a 20 minute presentation and 10 minutes of questions. Make sure each team member takes an active role in the presentation. Some questions might ask for clarifications of your solution based on concepts from the lectures. However, the oral defense will not include exam questions about the concepts from the lectures.

During the remaining lab sessions you can ask further clarification questions about the assignment. However, this project is the basis for an oral exam for which your team must work independently to complete the following learning goals:

- Developing, deploying and running a complex distributed multi-tiered cloud application;
- Learning how to implement a REST service based on a specification;
- Independently learn a more complex middleware platform based on public documentation on the web;
- Learning how to interact with existing badly documented REST services based on the HATEOAS principles;
- Learning to inspect and debug multi-tiered distributed applications.

The project is divided as follows:

1. **Level 1** are **the basic requirements** to pass the exam with a sufficient score, assuming you complete these requirements correctly and you can motivate your design choices. We expect each team to complete at least these requirements.
2. **Level 2** are **the advanced requirements**: you may decide yourself whether you want to complete these requirements depending on the time you have left and your ambitions for a high score.

This assignment must be carried out in groups of *four* people. Try to find teammates who will complete the same level of the project: either your team works together to complete the basic level, or your team works together to complete (some of) the advanced requirements. Your deliverables are:

1. The **code** of your project, which implements all requirements up to the level that you choose to complete.
2. A **report** in which you discuss design decisions and show additional insights into deploying a cloud application. This report should answer the questions in section 4.

To submit the code and the report, you should generate a zip file of your implementation. 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 make your submission unnecessarily large. Make sure additional libraries used in your solution are also included

in the zip file. Make sure that you place a PDF file named `report-cloud.pdf` with your report in the base directory of your project for it to be added to the zip file. You can change the name of this zip file in `pom.xml` under `properties/zip.filename`. We wil run your application without any additional external configuration (e.g. environment variables), so make sure that your application works with only `mvn spring-boot:run`.

Project submission: Before Sunday **June 18th** at 23:59, **each** student must submit the deliverables on Toledo. Submit these on Toledo under `Assignments > Final Project`. Include the name of your team members and team number in the report and the submission comments.

1 Introduction

After learning about the differences and (low-level) intricacies of remote communication protocols, we now shift to developing a cloud application that captures many of the major concepts supporting a distributed system.

Goal: In this project, you will learn building, deploying and running a Java web application that interacts with distributed independent servers. You will use the Spring Boot framework to experience how middleware accelerates the development of enterprise-grade applications. Optionally, you will further support high availability and scalability requirements by deploying your application on Google App Engine (GAE), a PaaS cloud platform. There, you will also learn to work with the NoSQL model for persisting data scalably while maintaining consistency.

Loading the project. The code from which you start can be found on Toledo, under 'Course Documents', together with this assignment document. Make all your changes based on this application code: extend or modify this code when necessary, even if not explicitly described in the assignment. We strongly advise to use IntelliJ for developing and debugging the application. In addition, some commandline tools will be needed as explained further.

Unzip the file from Toledo, and (`File >`) `Open` the project folder to generate an IntelliJ project. We require you to use Java SE Development Kit (JDK) version **17**. If necessary, tell IntelliJ to use the Java 17 SDK on your machine, rather than another version (e.g. 1.8 or 18). In the IDE, go to: `File -> Project Structure -> Project Settings -> Project`. Set the Project SDK to 17 and Project language level to the SDK default.

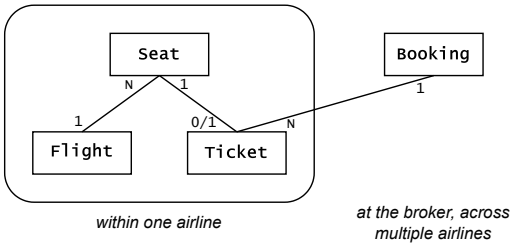
What is provided in the project. The provided project contains a Maven project and the start code for a Spring Boot application. More info is provided in Section 5 where we discuss the practical setup.

2 Use case and domain model

You will build the back-end of an airline broker's booking platform for making flight reservations across multiple *airlines*. Customers browse the *flights* that are available across the different airlines and select one or more *seats* for their itinerary, which will be stored in-memory as a *quote*. Once they have finalized their selection, they will confirm their selected seats and receive a *ticket* for each seat. All these tickets combined constitute one single *booking* that potentially contains tickets from multiple airlines. Meanwhile, managers of the platform can retrieve aggregate statistics on flights and customers.

This system must conform to several functional and non-functional requirements. Customers must be able to discover which seats are still available, book and cancel tickets, and retrieve details on these bookings. Managers must be able to compute metrics across all flights, bookings, and customers. The system must be able to cope with many concurrent interactions, in terms of both remaining responsive and avoiding double seat reservations.

Our domain model comprises the following entities and their relationships:



Flight The flight that a customer can take, which is provided by one specific airline. A flight has a predetermined number of *seat*s available. A flight has properties such as its number (id) and departure/arrival airports.

Seat A specific instance within a flight, characterised by (at least) a concrete date and time, and seat number. Additionally, a seat has properties such as a type (e.g., economy/business) and price. A seat can only be booked by one customer at a time, i.e., belong to exactly one ticket, or belong to zero tickets if the seat has not yet been booked.

Quote A temporary allocation of one seat. The quote is held by one specific customer, but contains no reference to that customer. A quote is a volatile object, i.e., it is not persisted and has no unique identifier. A quote does not block a seat from other quotes or booking reservations for that seat. A quote will evolve into a ticket during the booking process.

Ticket A final allocation of one seat to one specific customer. Not every seat has a corresponding ticket: if no ticket is present, the seat remains available. A ticket has a unique identifier. A ticket can be deleted if the customer cancels it.

Booking A collection of one or more tickets booked at the same time by one customer. Tickets within a booking may belong to multiple flights across multiple airlines. A booking contains references to the identifiers of its constituent tickets, and additionally has its own reference ID and properties such as the customer who made it and a timestamp.

Additionally, the following concepts exist in the domain model, but are not necessarily represented by separate entities:

Booking platform The booking platform is the central point with which customers interact. The platform lists all the flights that are available across all registered airlines, and allows to make reservations for seats across these flights. It offers a web-based interface for customers to interact with, and a Spring Boot application in the back end. This is the application you will develop in these lab sessions.

Airline The airline is an individual entity that operates a number of flights. The airline is responsible for tracking and reporting the availability of seats across its flights. Externally, we host two external airlines that your booking platform needs to interact with to offer flights to customers. The URLs are specified in Section 5.

Customer The person who receives tentative allocations and ultimately makes bookings. A customer can have multiple bookings. While a customer could have their own properties (address, login credentials, …), for simplicity we model them as an email address in the `customer` field of a booking.

Shopping cart Before finalizing a reservation, a customer can select a seat to already receive a tentative allocation to put in their 'shopping cart', while they are still searching other flights. We model a shopping cart as a booking that has not yet been persisted, where no tickets have been generated yet, and that does not have a reference number yet. Once the customer has selected all the seats they want, they can complete the transaction, after which the shopping cart will morph into a booking.

Example. Sarah visits the booking platform to reserve seats for her next flight. The platform lists three *flight*s: 'Dubai' at Airline A, 'Hong Kong' at Airline A, and 'Singapore' at Airline B. Sarah first selects a *seat* for the flight to 'Dubai' at Airline A: seat `5A` on Sunday 31 October 2022. This is a seat in first class at a price of 2500 euro. She adds a tentative allocation for this seat to her shopping cart. She continues selecting seats: another seat `5B` on Sunday 31 October 2022 for 'Dubai' at Airline A (to travel with a friend).

Having selected these seats, she proceeds to the checkout (where in a real application, she would for example fill out billing details and pay). Having completed the checkout, her reservation of these seats is now finalised: for each seat, a separate *ticket* is generated, and the *booking* with reference number `#211015023` links these tickets to Sarah.

3 Requirements

We now outline the requirements that your implementation of the assignment must fulfill. These depend on the level that you choose to complete; as a reminder, only **Level 1** is mandatory, while **Level 2** is optional. The latter level builds upon and extends the former, so we recommend you to first implement all requirements of **Level 1** before moving onto those of **Level 2**.

> Read the documentation (setup sections and external links) marked by these grey blocks to get started quickly with implementing each requirement.

3.1 Level 1: Basic Requirements

Target: You will build a fully functional distributed application that serves as the back end for the booking platform. The booking platform in **Level 1** acts only as a broker for other airlines who actually operate flights.

The business logic allows consumers to retrieve and reserve available flights and seats, and managers to discover the best performing flights and most loyal clients. Through the Spring Boot middleware, you can quickly set up and launch your application. You will enforce the elevated privileges of managers by implementing authentication and authorization through Firebase Authentication. You will use Cloud Firestore to persist and query the application data. You will ensure that the platform is fault tolerant even when relying on an unreliable service. Finally, while your application runs primarily locally for test purposes, the interaction with external services and the division of tasks between different processes and services already makes it a distributed system.

> 5.1 Installing the required software locally
> 5.2 Running the web application locally

3.1.1 Functional requirements: business logic

When you start the web application and login, you see that there are no flights available yet. The booking platform has to contact the airlines to get a list of flights and seats.

Therefore, **implement the API defined in the `api.yaml` specification** to allow the web application to contact the Reliable Airline via your booking platform (see Section 5.3). This specification is defined in the OpenAPI format, the most broadly adopted industry standard for describing HTTP-based APIs. Make use of the Spring Boot middleware to accelerate the development process, i.e., create a new `RestController` to serve API requests (as you have seen in the previous lab session on REST). Take into account that in a later part of this assignment, another airline will be added to this booking platform, and that in the future, it should be easy to add other airlines as well.

The API for the airlines has no documentation available, but uses REST Hypermedia to let developers discover the possible endpoints. Open the URL of the Reliable Airline in your browser or use `curl` for more info about which REST endpoints are available. As you learned during the SOAP/REST lab session, a RESTful API can include links to related resources and operations. Some of these links might be templated, and require that you replace the template argument with an actual value.

Then, to access the data from this airline within your application, you can use the `webClientBuilder` Bean to instantiate a Spring `WebClient` which allows you to make REST requests in Java. Make sure to use the Bean we provide, via dependency injection, as this one is already configured to understand REST Hypermedia. The Spring `WebClient` can also automatically convert the JSON response to an object or a list of objects of a class you can specify. For now, you can store the bookings in-memory and assume only a single instance of the booking platform will be active at all times.

Note that representing time and dates as strings in URLs or JSON data has always been a challenge. Investigate how dates and times are represented by the airlines and adopt the right DateTimeFormat in your implementation.

After completing this section, the web application should work correctly and you should be able to make reservations.

> 5.3 Accessing the airlines

OpenAPI Editor (you can paste the `api.yaml` file here)
`https://editor.swagger.io/`

Spring WebClient
`https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-client`

3.1.2 Security: Firebase Authentication

Users of the booking platform have to authenticate themselves before they can make reservations. We use *Spring Security* to integrate security on the server. Its configuration is defined in `auth.WebSecurityConfig`. This configuration is already provided, but if you add new endpoints, you can change that configuration to set whether these endpoints should have authentication enabled or not. The web application will authenticate the user by contacting the Firebase Authentication server. Once authenticated in the browser, the web application will attach an OAuth Identity token containing the user's identity and attributes to every request sent to the `/api/` endpoint. Requests that require authentication (defined in `auth.WebSecurityConfig`) will be intercepted by the security filter (`auth.SecurityFilter`), which can verify the token and make its properties available to the request. Decode the Identity Token in `auth.SecurityFilter#doFilterInternal` and **assign the correct user attributes** to the security context. Note that the local emulator does not sign the tokens, therefore you do not need to verify them.

Certain business logic methods should only be available to managers. **Restrict access to the `/api/getAllBookings` and `/api/getBestCustomers` endpoints** to only users with the manager role. A link to the manager page will appear once you are logged in as a manager. You can configure the role for your user in the Firebase Emulator Suite dashboard (`http://localhost:8081`).

Firebase Authentication
`https://firebase.google.com/docs/auth/admin/verify-id-tokens#verify_id_tokens_using_a_third-party_jwt_library`

Java JWT
`https://github.com/auth0/java-jwt`

Firebase Emulator Suite
`https://firebase.google.com/docs/emulator-suite`

3.1.3 Non-functional requirements: fault tolerance

You now have a fully working distributed system with several running components. However, when one component or one connection between two components fails, the whole system fails as well. **Modify the system to be more fault tolerant** and handle failures appropriately. You can focus on the communication with the airlines. For this, **add the Unreliable Airline** to your booking platform. Unfortunately, the Unreliable Airline did not create a very good system, and it will frequently, but undeterministically, respond with an error. Make sure this does not cause a total failure of the booking platform.

5.3 Accessing the airlines

3.1.4 Persistence: Cloud Firestore

In the first part of the assignment, bookings were stored in-memory, but of course this gives problems when an instance crashes or if you want to scale up the number of instances. For this reason, we will use the Cloud Firestore database to persist all bookings. **Modify the booking platform to store all bookings in Cloud Firestore**.

Cloud Firestore in Native Mode
`https://cloud.google.com/firestore/docs/quickstart-servers`

Cloud Firestore Emulator
`https://cloud.google.com/java/docs/reference/google-cloud-firestore/latest/com.google.cloud.firestore.FirestoreOptions.Builder#com_google_cloud_firestore_FirestoreOptions_Builder_setEmulatorHost_java_lang_String_`

3.1.5 ACID properties

Users might not want bookings to be partial, e.g., if they want to make sure a whole group of friends has tickets. Therefore, **implement Atomicity when confirming quotes**, i.e. either all quotes are succesfully reserved (= added to the booking), or none of them are reserved at all. You may assume that the airlines will correctly handle concurrent reservations: i.e., if an airline creates a ticket succesfully, it will not create another ticket for the same seat. Your application can also crash during operation, e.g., due to a hardware failure of the underlying machine. Make sure your application correctly recovers from such a crash. And that the **Atomicity** property is still satisfied. **Make sure the Consistency, Isolation and Durability properties are also satisfied**.

3.2 Level 2: Advanced Requirements

Target: You have the opportunity to experience a real-world cloud setup by deploying your (extended) application to a real Google App Engine instance, leveraging services available (only) on a real Google App Engine deployment. You will use Cloud Firestore to persist and query extended application data for a new airline, and will adapt your transaction demarcation accordingly. *Reminder: this level is optional.*

3.2.1 Persistence: Cloud Firestore (extended)

Part 1. The booking platform wants to extend their platform so they can provide their own flights as well, instead of acting only as a broker. **Extend the booking platform to provide flights, seats and tickets** for the data located in `resources/data.json` (accessible from the classpath). The code for this "internal" airline can be located within the same package as the booking platform. Of course, this airline should persist all flights, seats and tickets in Cloud Firestore. **Think about your data model** to be able to make performant queries, and avoid manually querying and processing data in Java. Every time your application starts, check whether the data is already initialized. If not, read `data.json` and initialize the database. Putting this data into the Firestore can take a while.

Part 2. **Make sure that the "internal" airline uses correct transactional behaviour** when necessary, to make sure no seat can get reserved twice. However, only use transactions when you really need them.

Cloud Firestore in Native Mode
`https://cloud.google.com/firestore/docs/quickstart-servers`

Cloud Firestore Emulator
`https://cloud.google.com/java/docs/reference/google-cloud-firestore/latest/com.google.cloud.firestore.FirestoreOptions.Builder#com_google_cloud_firestore_FirestoreOptions_Builder_setEmulatorHost_java_lang_String_`

Cloud Firestore Data Model
`https://cloud.google.com/firestore/docs/data-model`

3.2.2 Deployment to Google App Engine

Deploy your application to Google App Engine, and use the real production services in Google Cloud instead of the emulators. Make sure your application still works locally as well. You can use the `isProduction`

Bean to check whether your code is running locally or in the cloud.

> 5.4 Deploying on Google Cloud

3.2.3 Security: Firebase Authentication (extended)

The real cloud environment does sign the Identity Tokens. Adapt your solution in `auth.SecurityFilter` to not only decode the token, but to also **verify the signature**. Do not rely on the Firebase Admin SDK, instead dynamically request the public keys from the endpoint provided by Google and use the Java JWT library from Auth0 to verify the token manually. The real Firebase console does not support adding custom claims from the UI. You can add claims programmatically using the Admin API, however, you are not required to do this for this assignment.

> **Firebase Authentication**
> `https://firebase.google.com/docs/auth/admin/verify-id-tokens#verify_id_tokens_using_a_third-party_jwt_library`
> **Java JWT**
> `https://github.com/auth0/java-jwt`

4 Reporting

We would like you to report on your design and the decisions you have made (design choices, possible alternatives and trade-offs), and on your understanding and implementation of distributed concepts. Provide your answers in English, and keep them short, concise and to the point: a few lines is often enough.

Do not forget to put your names and team number in the report, and store the report (as one PDF called `report-cloud.pdf`) in the main directory of your solution (next to `pom.xml`), such that the final zip creation process includes the report.

For each team member, mention the role and tasks of the team member as well as the amount of hours that the team member contributed to the project. Be honest and fair.

Compile a report that answers the following questions, for the levels that you have chosen to complete:

4.1 Level 1: Mandatory basic requirements

1. Is there a scenario in which your implementation of **ACID properties** may lead to double bookings of one seat?
2. How does **role-based access control** simplify the requirement that only authorized users can access manager methods?
3. Which components would need to change if you switch to **another authentication provider**? Where may such a change make it more/less difficult to correctly enforce access control rules, and what would an authentication provider therefore ideally provide?
4. How does your application cope with failures of the **Unreliable Airline**? How severely faulty may that airline become before there is a significant impact on the functionality of your application?

4.2 Level 2: Optional advanced requirements

1. How have you structured your **data model** (i.e. the entities and their relationships), and why in this way?
2. Compared to a relational database, what sort of **query limitations** have you faced when using the Cloud Firestore?

3. How does your implementation of **transactional behaviour** for **Level 2** compare with the all-or-nothing semantics required in **Level 1**?
4. What are the pitfalls when **migrating** a locally developed application to a real-world cloud platform? What are the restrictions of Google Cloud Platform in this regard?
5. How extensive is the **tie-in** of your application with Google Cloud Platform? Which changes do you deem necessary for migrating to another cloud provider?
6. Include the **App Engine URL** where your application is deployed. Provide a login and password for the teaching team and mention it in your report. Make sure that you keep your application deployed until after you have received your grades.

5 Setup

This section outlines the setup of the project.

5.1 Installing the required software locally

This is needed for **Level 1**.

You will need to have Google Cloud SDK and Firebase CLI installed.

- Follow the Quickstart guide at `https://cloud.google.com/sdk/docs/install-sdk` to install the Google Cloud SDK appropriate for Linux. We advise to use the Linux installation archive x86_64.tar.gz. Then, follow the guide at `https://cloud.google.com/appengine/docs/standard/java-gen2/setting-up-environment` to set up the Java development environment (crucially, install the app-enginejava component).
- You'll also need to install the Firebase CLI, see `https://firebase.google.com/docs/cli`. Using the auto-install script is advised.

The other dependencies (including Spring Boot) are managed via Maven. They will be installed automatically the first time you run the application. You should have Maven installed, and IntelliJ has built-in support. Otherwise, please refer once again to the homework to install your development environment.

5.2 Running the web application locally

This is needed for **Level 1**.

Step 1. Running the booking platform locally makes it easy to test your application during development. You can use a local emulator to emulate the cloud services. Run the following command from the terminal in your project root to start the Firebase Emulator suite:

```
firebase emulators:start --project demo-distributed-systems-kul
```

This will start an emulator for Firebase Authentication and Cloud Firestore, which are needed for this assignment. The emulator suite also has a dashboard where you can debug and configure some of the emulators. Visit `http://localhost:8081` in your browser to view this dashboard. All data in the emulators is only stored in-memory, so if you want to start from a clean state, just quit the emulators and start them again.

Step 2. Once the emulators are running, you can start your Spring application using Maven.

- **IntelliJ**: In IntelliJ, you can open the Maven tool in the sidebar and double-click on the `spring-boot:run` goal under `Plugins`.
- **CLI**: If you have installed Maven, you can also run it from a terminal using the following command: `mvn spring-boot:run`.

You can now open a browser and go to `http://localhost:8080` to use the application.

We provide a JavaScript-based web app as the graphical front end that allows customers to make reservations, and managers to generate reports. There is no need to change this. However, the developer of this graphical front has already implemented the remote calls from the GUI to the REST endpoints of the back end. Your implementation of the back end will need to obey and adhere to these remote calls. Using the developer tools of your Browser (F12 on FireFox and Chrome) you can observe the outgoing calls to the (not yet implemented) REST services.

Step 3: Debugging your code. You can also run the Spring Boot Application from IntelliJ by navigating to the Application.java class, and using the Run button to run the current file. IntelliJ will then automatically create a run configuration. You can use this configuration to run and/or debug the application using the appropriate button or shortcut. This method will also allow you to halt the server on certain breakpoints that you have set in the code (which is not possible with the Maven approach above in Step 2). This should avoid println-debugging. When your server runs in debugging mode, you can still use the Web GUI to call the REST operations hosted by your server, and trigger the breakpoints.

5.3 Accessing the airlines

This is needed for **Level 1**.

We provide two different airlines that your booking platform has to interact with to offer flights to customers. One airline is the Reliable Airline and this airline is very reliable: API calls made to this airline will (normally) not fail, unless some application-level constraint is not satisfied, i.e., a seat is already booked. The other airline is the Unreliable Airline: this airline is trying their best, but their servers are not always that reliable. You might get an error, even though your request was correct.

The airlines are located at the following HTTP endpoints:

- **Reliable Airline**: `http://reliable.westeurope.cloudapp.azure.com/`
- **Unreliable Airline**: `http://unreliable.eastus.cloudapp.azure.com/`

These API endpoints are protected with an API key to prevent outsiders from accessing these endpoints. You are free to hardcode the API key in your source code, just do not share it publicly on the internet. The API key this year is `Iw8zeveVyaPNWonPNaU0213uw3g6Ei`. Append a request parameter `key` with the API key to every request: e.g., `http://reliable.westeurope.cloudapp.azure.com/flights?key=Iw8zeveVyaPNWonPNaU0213uw3g6Ei`

Be aware that the airlines are shared with all students, so it is a real possibility that two students try to reserve the same ticket concurrently. The data stored at these airlines will be reset daily at night, so there should always be plenty of free seats available to work with.

5.4 Deploying on Google Cloud

This is needed for **Level 2**.

5.4.1 Obtaining Google Cloud credits

A deployment to the cloud is not free, therefore, we arranged a $50 Google Cloud credit coupon for everyone making the optional part of the assignment. Check Toledo (under 'Assignments') to request a coupon. Please note that you can only request a coupon once, so only use those credits for this assignment until you know you passed the course.

5.4.2 Configuring your Google Cloud project

Create a new project on `https://console.cloud.google.com/`. If asked about a billing account, choose the "Billing Account for Education" you received after you redeemed your Google Cloud credits. Take note of the project ID and put this inside your report.

Firebase Authentication. Visit `https://console.cloud.google.com/marketplace/product/google-cloud-platform/firebase-authentication` and choose “Get started for free”. You will be redirected to the Firebase console and asked to select a project. Add a new project and link it to the Cloud Project you just created. If you get the option to "Add Firebase to one of your existing Google Cloud projects", you can choose this instead of creating a new project. If Firebase suggests to add other products (e.g. Analytics), you can safely skip these as we will not use them.

Once created, visit `https://console.firebase.google.com/project/` and go to the Authentication tab and click on "Get started", and then enable the “Email/password” sign-in provider.

Follow these steps to add a new web app: `https://firebase.google.com/docs/web/setup#register-app`. Make sure to select "Use a <script> tag" and replace the values in `resources/static/js/index.js` with the correct values you receive there.

App Engine. Visit `https://console.cloud.google.com/appengine` and create an empty App Engine application. Put the correct `projectId` in the configuration of the `appengine-maven-plugin` in your `pom.xml` file.

Cloud Firestore. Visit `https://console.cloud.google.com/firestore/` and choose `Native mode`. If a database is already created for you, check that the `Native mode` is used, otherwise choose "Switch to Native mode".

5.4.3 Deploying to the Cloud

Build your application using the `package` Maven goal, then execute the `appengine:deploy` Maven goal (under `Maven` > `Plugins` if you are using IntelliJ).

In the log messages (in the `Run` tab in IntelliJ), you will see the URL at which your application is deployed. After that, you can visit that URL in your browser and test your application. The first time you visit the URL, the application still needs to be started. This might take a while, especially the first time when your application must load the data into the Firestore. If you get any errors, visit `https://console.cloud.google.com/logs/` and `https://console.cloud.google.com/appengine` to debug your application running on App Engine.
If it works, CONGRATULATIONS, you’ve now deployed a distributed application on the Google Cloud Platform.