2 | Distributed Cloud Applications
Deploying service-oriented distributed applications on a cloud platform

Practical arrangements

The remaining six lab sessions are allocated to completing one large assignment around cloud applications. The assignment is divided as follows:

1. **Level 1** is **mandatory** and we expect a submission from *everyone* after *three* sessions (weeks).
2. **Level 2** is **optional**: you may decide yourself whether you want to make and submit the assignment after *six* sessions (weeks), in exchange for one fewer question at the exam.

This assignment must be carried out in groups of *two* people. Try to find a teammate who will complete the same levels of the assignment: either you both do only the mandatory level, or you both do the optional level(s) as well.

For each level, your deliverables are:

1. The **code** of your project, which implements all requirements up to the level that you choose to make.
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 include unnecessary files. 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`.

***Mandatory* submission:** Before Friday **November 26** at 19:00, **each** student must submit their deliverables (code and report) for **Level 1** to Toledo. Submit these to Toledo under `Assignments > Submission 2.1: Cloud: Mandatory part`.

***Optional* submission:** If you completed the second (optional) part, before Friday **December 17** at 19:00, you must submit their results for **Level 1** and **Level 2** to Toledo; i.e., ensure that **all** requirements (including the mandatory ones) are fulfilled. Then, submit your updated deliverables (a new zip file for the extended code, and the extended report) to Toledo under `Assignments > Submission 2.2: Cloud: Optional part`.

1 Introduction

After learning about the differences and (low-level) intricacies of remote communication protocols, we now shift to developing a large-scale enterprise application that captures many of the major concepts supporting a distributed system. In addition, we leverage cloud computing to outsource the required computing resources to a third party, allowing us to leverage their scale and complementary services.

Goal: In this assignment, you will familiarize yourself with building, deploying and running a Java web application that interacts with distributed independent servers. You will leverage indirect communication and distributed transactions to reliably perform operations across these different servers. Moreover, 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 'Assignments'. Make all your changes based on this application code: extend or modify this code when necessary, even if not explicitly described in the assignment. Use your preferred IDE or the command line.

In the **computer labs**, you can use the preinstalled IntelliJ IDE:

```
/localhost/packages/ds/intellij.sh
```

Unzip the file on Toledo, and (`File >`) `Open` the top-level folder to generate a project.

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 **11**.

If necessary, tell IntelliJ to use the Java 11 SDK on your machine, rather than another version (e.g. 1.8 or 16). In the IDE, go to: `File -> Project Structure -> Project Settings -> Project`. Set the Project SDK to 11 and Project language level to the SDK default.

2 Use case and domain model

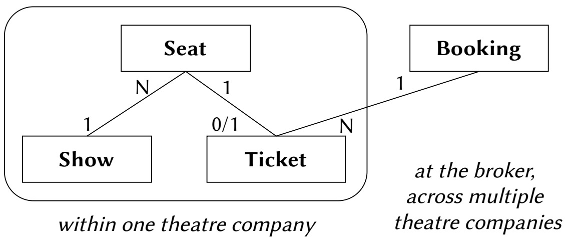
You will build the back end of a booking platform for making reservations across multiple *theatre companies*. Customers browse the *shows* that are playing across the different companies and select one or more *seats* if they want to attend shows, which will be stored in-memory as a *quote*. Once they have finalised their selection, they will confirm their selected seats and receive a *ticket* for each one. All these tickets combined constitute one single *booking* that potentially contains tickets from multiple companies. Meanwhile, managers for each company or for the platform as a whole can retrieve aggregate statistics on shows and customers. We provide a web app as the front end that allows customers to make reservations, and managers to generate reports.

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 shows, customers, and bookings. The system must be able to cope with many concurrent interactions, in terms of both remaining responsive and avoiding double reservations. These requirements as well as the client-server design of the application make a distributed system ideal for this use case.

Our domain model comprises the following entities and their relationships:

Show The event that a customer can attend, which is hosted by one specific theatre company. A show has a predetermined number of *seat*s available. A show has properties such as its name and location.

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


Seat
Booking
Show
Ticket
N
1
1
0/1
1
N
within one theatre company
at the broker, across multiple theatre companies

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 will evolve into an 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 shows across multiple theatre companies. 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 shows that are available across all registered theatre companies, and allows to make reservations for seats across these shows. It offers a web-based interface for customers to interact with, and a Java Spring Boot application in the backend. This is the application you will develop in these lab sessions.

Theatre company The theatre company is an individual entity that hosts a number of shows. The company is responsible for tracking and reporting the availability of seats across its shows. We provide two external theatre companies that your booking platform needs to interact with to offer shows to customers.

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 browsing other shows. 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 favourite shows. The platform lists four *show*s: 'Hamlet' at company 30CC, 'Macbeth' at 30CC, 'Macbeth' at STUK, and 'Othello' at OPEK. Sarah first select a *seat* for the show 'Hamlet' at 30CC: seat `5A` on Sunday 31 October 2021. This is a floor seat at a price of 25 euro. She adds a tentative allocation for this seat to her shopping cart. She continues selecting seats: another seat `5B` on Sunday 31 October 2021 for 'Hamlet' at 30CC (to attend with a friend), and seat `115` on Monday 1 November 2021 for 'Macbeth' at STUK.

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 blocks to get started quickly with implementing each requirement.

3.1 Level 1: Developing a locally deployed distributed application

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 companies who actually host shows. The business logic allows consumers to retrieve and reserve available shows and seats, and managers to discover the best performing shows 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. To improve scalability and guarantee high availability, you will use indirect communication through Cloud Pub/Sub to decouple reservation processing from other back-end tasks, all while maintaining all-or-nothing semantics. 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 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 (which you will need to do for Cloud Pub/Sub later), you can configure here whether they 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 send an OAuth Identity token containing the user's identity and attributes to the `/authenticate` endpoint. This endpoint will put the Identity Token in a cookie, so it will be included in all future requests. 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 `/manager` page** to only users with the manager role. This manager page is created in `controller.ViewController#viewManager`. 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`

3.1.2 Functional requirements: business logic

So far, when you start the web application, you see that there are no shows available yet. The booking platform has to contact the theatre companies to get a list of shows and seats. Therefore, **implement the business logic in the `Model` class** for contacting the Reliable Theatre company (see Section 5.3). Take into account that in a later part of this assignment, another company will be added to this booking platform, and that in the future, it should be easy to add other companies as well. The API for the theatre company has no documentation available, but uses REST Hypermedia to let developers discover the possible endpoints. Open the URL of the Reliable Theatre in your browser or use `curl https://reliabletheatrecompany.com/?key=wCIoTqec6vGJijW2meeqSokanZuqOL` for more info about which REST endpoints are available. As you learned during the SOAP/REST lab session, a RESTful API includes links to related resources and operations. Then, to access the data from this company 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, 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. You can store the bookings in-memory and assume only a single instance will be active at all times. After completing this section, the web application should work correctly and you should be able to make reservations.

> 5.3 Accessing the theatre companies

> **Spring WebClient**
> `https://docs.spring.io/spring-framework/docs/5.0.13.RELEASE/spring-framework-reference/web-reactive.html#webflux-client`

3.1.3 Indirect communication: Cloud Pub/Sub

While the system already works locally for a single user, we adapt this system to better utilize the scalability opportunities that would be available in a cloud environment. In our case, we leverage the possibility to use indirect communication for decoupling background processes. **Redesign the system to make use of Cloud Pub/Sub *push* subscriptions**. Think about which method in the `Model` class will benefit from using indirect communication. However, overusing indirect communication may also degrade the user experience, as in most cases prompt response is both feasible and more desirable. In **Level 1**, your platform is running locally and therefore must use the Cloud Pub/Sub emulator. Make sure to configure the client library to use the emulator, instead of trying to use the real service. Remember that Spring Security requires every request to be authenticated by default.

> **Cloud Pub/Sub Publisher**
> `https://cloud.google.com/pubsub/docs/publisher`
>
> **Cloud Pub/Sub Subscription (scroll down to push delivery)**
> `https://cloud.google.com/pubsub/docs/admin#creating_subscriptions`
>
> **Cloud Pub/Sub Emulator**
> `https://cloud.google.com/pubsub/docs/emulator#accessing_environment_variables`

3.1.4 All-or-nothing semantics

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 all-or-nothing semantics when confirming quotes**, i.e. either all quotes are succesfully reserved (= added to the booking), or none of them are reserved at all. For **Level 1** of the assignment, you may assume that the booking platform itself will not crash.

3.1.5 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 companies. For this, **add the Unreliable Theatre company** to your booking platform. Unfortunately, the Unreliable Theatre company 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.

3.2 Level 2: Deploying the distributed application to the cloud

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 the application data in a highly scalable manner supported by the NoSQL model, and will adapt your transaction demarcation accordingly. Moreover, you will implement a decoupled feedback channel to alert users about the progress of their reservations. The cloud deployment will also allow you to incorporate strategies to ensure availability and scalability even while uptake of the application is growing. *Reminder: this level is optional.*

3.2.1 Persistence: Cloud Firestore

Part 1. In the previous 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**.

Part 2. The booking platform also wants to extend their platform so they can host their own shows as well, instead of acting only as a broker. **Extend the booking platform to provide shows, seats and tickets** for the data located in `resources/data.json` (accessible from the classpath). The code for this "internal" company can be located within the same package as the booking platform. Of course, this company should persist all shows, 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 wether 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 3. **Make sure that the "internal" company 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_`
>
> `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 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`

3.2.4 Indirect communication: feedback channel

The process of creating a new booking uses indirect communication using Cloud Pub/Sub. However, this makes it very unclear for your customers what the current status is of a new (unconfirmed) booking. The indirect communication makes it impossible to give direct feedback via the browser request. Therefore, use SendGrid to **implement a feedback channel** to notify customers of a successful completion or failure of a new booking request.

> 5.5 Setting up SendGrid

> **Sending messages from App engine**
> `https://cloud.google.com/appengine/docs/standard/java11/sending-messages#sendgrid`
>
> **SendGrid**
> `https://github.com/sendgrid/sendgrid-java#quick-start`

3.2.5 Non-functional requirements: scalability

Check out the Google App Engine Cloud Console and see how many instances are running. Play with the application and see how the number of instances changes. The configuration for this auto scaling is located in `main/appengine/app.yaml`. If you notice your application doesn't scale well, try tweaking the configuration to improve it (this might not be necessary for your solution).

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. You are encouraged to use the LaTeX template on Toledo. Do not forget to put your names 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.

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

4.1 Level 1: Developing a locally deployed distributed application

1. Imagine you were to deploy your application to a real cloud environment, so not a lab deployment where everything runs on the same machine. Which hosts/systems would then execute which processes, i.e., how are the remote objects **distributed over hosts**? Clearly outline which parts belong to the front and back end, and annotate with relevant services. Create a component/deployment diagram to illustrate this: highlight where the client(s) and server(s) are.
2. Where in your application were you able to leverage **middleware** to hide complexity and speed up development?
3. At which step of the booking workflow (create quote, collect cart, create booking) would the **indirect communication** between objects or components kick in? Describe the steps that cause the indirect communication, and what happens afterwards.
4. Which kind of **data** is passed **between the application and the background worker** when creating a booking? Does it make sense to persist data and only pass references to that data?
5. How does your solution to **indirect communication** improve **scalability** and **responsiveness** (especially in a distributed/cloud context)?
6. Can you give an example of a user action where indirect communication would *not* be beneficial in terms of the **trade-off between scalability and usability**?
7. Is there a scenario in which your implementation of **all-or-nothing semantics** may lead to double bookings of one seat?
8. How does **role-based access control** simplify the requirement that only authorized users can access manager methods?
9. 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?
10. How does your application cope with failures of the **Unreliable Theatre company**? How severely faulty may that company become before there is a significant impact on the functionality of your application?

4.2 Level 2: Deploying the distributed application to the cloud

Next to the questions, **include your App Engine URL** where your application is deployed. Make sure that you keep your application deployed until after you have received your grades.

11. How does using a **NoSQL** database affect **scalability** and **availability**?
12. How have you structured your **data model** (i.e. the entities and their relationships), and why in this way?
13. Compared to a relational database, what sort of **query limitations** have you faced when using the Cloud Firestore?
14. How does your implementation of **transactional behaviour** for Level 2 compare with the all-or-nothing semantics required in Level 1?
15. How have you implemented your **feedback channel**? What triggers the feedback channel to be used, how does the client use the feedback channel and what information is sent?

16. Did you make any changes to the Google App Engine **configuration** to improve **scalability**? If so, which changes did you make and why? If not, why was it not necessary and what does the default configuration achieve?
17. What are the benefits of running an application or a service (1) as a **web service** instead of natively and (2) **in the cloud** instead of on your own server?
18. 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?
19. 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?
20. What is the estimated monthly cost if you expect 1000 customers every day? How would this cost scale when your customer base would grow to one million users a day? Can you think of some ways to reduce this cost?

5 Setup

5.1 Installing the required software locally

This is needed for **Level 1**.

The PC labs have the Google Cloud SDK and Firebase CLI already installed. If you are working on your own machine, follow the *Bring Your Own Device* (byod) guide on Toledo to install them.

The other dependencies (including Spring Boot) are managed via Maven. They will be installed automatically the first time you run the application. The PC labs have Maven installed, and IntelliJ has built-in support. Otherwise, please refer once again to the *Bring Your Own Device* guide.

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. However, from your local environment you do not always have or even want access to all different Cloud products you are using. As a solution, you can use a local emulator to emulate these cloud services. Run the following command from the terminal to start the Firebase Emulator suite:

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

In the PC labs, `firebase` is located inside `/localhost/packages/ds/firebase/latest/`.
This will start an emulator for Firebase Authentication, Cloud Pub/Sub 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.

5.3 Accessing the theatre companies

This is needed for **Level 1**.

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

The theatre companies are located at the following HTTP endpoints:

- **Reliable Theatre**: `https://reliabletheatrecompany.com/`
- **Unreliable Theatre**: `https://unreliabletheatrecompany.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 `wCIoTqec6vGJijW2meeqSokanZuqOL`. Append a request parameter `key` with the API key to every request: e.g., `https://reliabletheatrecompany.com/shows?key=wCIoTqec6vGJijW2meeqSokanZuqOL`

Be aware that the theatre companies 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 companies will be reset daily, 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.

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 replace the values in `resources/static/js/login.js` with the correct values you received when selecting "Use a <script> tag".

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

Cloud Pub/sub. Visit `https://console.cloud.google.com/cloudpubsub` and configure your topic. Make sure the names match those in your code. The documentation might help you: `https://cloud.google.com/pubsub/docs/quickstart-console`. You will not be able to create the push subscription yet, since you do not know the URL of your App Engine application.

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 on which your application is deployed. Use this to further configure your push subscription. After that, you can visit that URL in your browser and test your application. The first time you visit the url, the application is 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 real distributed system into the Google Cloud.

5.5 Setting up SendGrid

This is needed for $\boxed{\textbf{Level 2}}$.

You can setup the SendGrid Email API from within the Google Cloud Console. Visit `https://console.cloud.google.com/launcher/details/sendgrid-app/sendgrid-email` and create a new SendGrid account. The free plan is more than enough.

Once your account is created, verify an email address to send emails from here: `https://app.sendgrid.com/settings/sender_auth/senders`. You can use your student email address for this. Once verified, you can create an API key here: `https://app.sendgrid.com/settings/api_keys`. Now you are all set up, and you can use the SendGrid API to send emails.

6 Spring Boot Essentials

The booking platform uses Spring Boot to accelerate the development process. This section provides a brief overview of its functionality. As an example, you can take a look at how the provided code uses Spring Boot and these annotations.

`@SpringBootApplication` Tags the class as a source of `Bean` definitions for the application context; and tells Spring to look for other components in the same package.

`@Bean` An object that is going to be instantiated, assembled and managed by Spring. They can be injected in other components (*dependency injection*). The annotation is used on methods.

`@Component` Similar to a `Bean`, but also enables automatic wiring within the class. The annotation is used on classes.

`@Autowired` Marks a constructor, field, or setter method for automatic wiring by Spring dependency injection. This typically happens based on the type of the object.

`@Resource(name = ...)` Injects a `Bean` by using its name.

`@PostConstruct` The annotated method will be called right after the component is created and all its dependencies are injected.

`@RestController` Enables the component to handle HTTP requests. This also implies the `@Component` annotation.

`@GetMapping(...)` Maps HTTP GET requests to the path given to the annotated method.

`@PostMapping(...)` Maps HTTP POST requests to the path given to the annotated method.

`@PutMapping(...)` Maps HTTP PUT requests to the path given to the annotated method.