**Distributed Systems**

*Trip booking application tutorial using Java language and Spring Boot framework*

# Setup

1. Install version 22 of OpenJDK
2. Install the latest version of IntelliJ Community Edition
3. Create a new private GitHub repository and name it last-second
4. Use Spring Initializer to create a new Spring Boot project
   * <https://start.spring.io/#!type=maven-project&language=java&platformVersion=3.3.3&packaging=jar&jvmVersion=22&groupId=com.bagdouri&artifactId=lastsecond&name=lastsecond&description=Demo%20project%20for%20Spring%20Boot&packageName=com.bagdouri.lastsecond&dependencies=lombok,configuration-processor,devtools,web>
5. Extract the Spring project into your local GitHub repository
6. Configure SSH access to your GitHub account
   * Generate a new SSH key on your computer (using ssh-keygen command)
   * Copy the content of your public key (<user-folder>/.ssh/id\_rsa.pub)
   * Add the public key to your GitHub account (<https://github.com/settings/ssh/new>)
7. From inside the extracted folder, commit and push the content of your local repository by following the instructors indicated on your project’s GitHub page
8. Import the project into IntelliJ
   * Here are some useful IntelliJ shortcuts

|  |  |  |
| --- | --- | --- |
| Action | Windows shortcut | Mac shortcut |
| Optimize/Fix imports | Ctrl + Alt + O | Ctrl + Options + O |
| Go to class | Ctrl + N | Cmd + O |
| Go to file | Ctrl + Shift + N | Cmd + Shift + O |
| Navigate to previous location | Ctrl + Alt + Left | Cmd + Options + Left |
| Navigate to next location | Ctrl + Alt + Right | Cmd + Options + Right |
| Search in files | Ctrl + Shift + F | Cmd + Shift + F |
| Generate code | Alt + Ins | Cmd + N |

1. Build the project using Maven actions clean & package

A screenshot of a computer

Description automatically generated

1. Once the Maven build is successful, reload the Maven project (build & reload must be done after each time you make a modification to the Maven file pom.xml)

A screenshot of a computer

Description automatically generated

1. Rename the *application.properties* file inside the *resources* folder to *application.yml* file
   * The YAML hierarchical syntax is more readable than PROPERTIES key=value syntax
   * Make any necessary changes to make the file YAML compliant

# REST endpoints

1. Create the following package structure

A screenshot of a computer

Description automatically generated

1. Create a DTO class *FlightBookingDto* representing a flight booking instance (make sure Lombok plugin is installed)
   * ID, customer first name, customer last name, customer ID number, departure airport, departure date/time, arrival airport, arrival date/time, price
2. Create a service FlightBookingService that manages flight bookings
   * Create flight booking (save in a list or map for now)
   * Get flight booking by ID
   * List all flight bookings
   * Update flight booking
   * Delete flight bookings
3. Create a controller FlightBookingController that provides endpoints to manage flight bookings
4. Add the *springdoc-openapi-starter-webmvc-ui* dependency to your maven POM file to generate an OpenAPI (Swagger) documentation (don’t forget to re-build and re-load Maven project)
5. Run the application on your local environment
   * Make sure that the application starts successfully
   * Make sure that the swagger is accessible on <http://localhost:8080/swagger-ui/index.html>
   * Make sure that all of your endpoints work as expected
   * Invoke your endpoints using *Postman* and make sure that they're all working as expected

# Error handling

1. Create a *FlightBookingNotFoundException* class and throw this exception when the user attempts to update a non-existing booking
2. Try to update a non-exisiting booking and make sure that a 500 HTTP error is returned
3. Create a *ControllerAdvice* that returns a proper HTTP response whenever *FlightBookingNotFound* is thrown
4. Try to update a non-exisiting booking and make sure that a proper HTTP response is returned

# Security

1. Follow the tutorial below to secure the web application with TLS (HTTPS) using a self-signed certificate
   * <https://www.baeldung.com/spring-boot-https-self-signed-certificate>
2. Make sure that the server doesn't accept unsecured HTTP requests
3. Make sure that the server accepts HTTPS requests
   * You should notice that the browser doesn't trust the connection to your application's server since it doesn't trust your self-signed certificate

# Database

1. Add *spring-boot-starter-data-jpa* dependency to your maven POM file to be able to use Spring Data
2. Create a *FlightBooking* entity class
3. Create a *FlightBookingRepository* repository class
4. Use the *FlightBookingRepository* inside the *FlightBookingService* to read/write flight bookings to the DB instead of memory
5. Install and start PostgreSQL server on your local machine
6. Create a new PostgreSQL database called lastsecond\_flight (using *pgAdmin*)
7. Add *postgresql* dependency to your maven POM file to be able to connect to your PostgreSQL server
8. Add DB connection configuration to your *application.yml* file

|  |
| --- |
| spring:  datasource:  url: jdbc:postgresql://localhost:5432/lastminute  username: postgres  password: yourPassword |

1. Add the *liquibase* dependency to your Maven POM file to be able to automatically create the DB at startup (if not already created)
2. Enable *Liquibase* via the configuration file

|  |
| --- |
| spring:  liquibase:  enabled: true |

1. Create a Liquibase script in the resources folder that creates a *FLIGHT\_BOOKING* table with the same structure as the *FlightBooking* entity
2. Create a Liquibase master script that imports the script mentioned above
3. Add the following property to verify that the DB schema matches the entity classes at the startup of the application

|  |
| --- |
| spring:  jpa:  hibernate:  ddl-auto: validate |

1. Run the application and make sure that the endpoints behave as expected and that data is correctly inserted in the DB
   * Observe and investigate SQL queries sent to the PostgreSQL server : <https://stackoverflow.com/questions/30118683/how-can-i-log-sql-statements-in-spring-boot>

# Multiple Applications

1. Create a new Maven sub-module *lastsecond-flight*
   * You should now have two pom.xml files : one at the root level and one at the level of the *lastsecond-flight* sub-module
   * The *lastsecond-flight* sub-module pom.xml extends the root module using the *<parent>* tag
   * The root level pom.xml imports the *lastsecond-flight* sub-module using the *<module>* tag
   * When you build the root level Maven project, the sub-module *lastsecond-flight* will be implicitly built as well
2. Move all of the Java and resources file to the *lastsecond-flight* sub-module
3. Create a new sub-module *lastsecond-hotel* that is responsible for managing hotel reservations
4. Similarly to *lastsecond-flight*, implement all the necessary hotel booking features in the *lastsecond-hotel* sub-module (REST, security, error handling, DB, …)
5. To avoid port conflict between the two applications, override the default port in the *lastsecond-hotel* module
6. Run both applications and make sure that they can run simultaneously and that they both behave as expected

# Coordinator Application

1. Create a new Maven sub-module *lastsecond-trip*
   * This module will be exposed to customers to be able to book a trip composed of a flight and a hotel
2. Create a new package *client* that will contain the client definitions that will be used by the trip module to invoke endpoints of the flight and hotel applications.
3. Import the *retrofit2* dependency
   * Retrofit is a type-safe HTTP client library
4. Create a *RetrofitConfiguration* class that defines two *Retrofit* beans that will be used to create HTTP clients for flight and hotel applications
5. Inside the *client* package create *FlightBookingClient* and *HotelBookingClient* Retrofit interfaces that define the HTTP endpoints
6. Declare *FlightBookingClient* and *HotelBookingClient* beans inside *RetrofitConfiguration*.
7. Create a *TripBookingService* that can be used to manage trip bookings (create, get, list, update, delete).
   * The service will invoke the flight and hotel applications to perform the requested action
   * The service must guarantee consistency (for example, when a trip booking is created, both the flight booking and the hotel booking must be created, or none)
8. Create a *TripBookingController* that customers can use to perform trip booking actions (assume that customers will only have access to the trip booking application)
9. Run all three applications and make sure they behave as expected
   * Perform actions on the trip booking application and make sure that your action get reflected on the flight and booking applications
   * Simulate flight and/or hotel application crash and make sure that the trip application is able to return adequate HTTP errors and that it guarantees consistency

# Failure Detection

To avoid putting additional stress on the applications, the trip application should not invoke the flight booking and hotel applications if they’re not available.

1. Implement a ping mechanism that will allow the trip booking application to assess the availability of the flight booking application
2. Implement a heartbeat mechanism that will allow the trip booking application to assess the availability of the hotel booking application
3. Add a logging interceptor to the Retrofit clients of your trip booking application to log HTTP requests and responses
4. Run all three applications and make sure the failure detection mechanism works as expected
   * Enable debug logs on the *TripBookingService* to determine which endpoints are being invoked
   * *TripBookingService* should not invoke an endpoint if the concerned application is considered unavailable

# Caching

To reduce the volume of requests made to the applications and the DB, we will put in place in-memory caches.

1. Add Guava dependency to the trip application POM.
2. Create a *LoadingCache* object in the *TripBookingService* for flight bookings. This cache will be used whenever a customer requests trip information.
   1. The cache should query the information from the flight application when it receives a request for a given trip for the first time.
   2. The following queries for the same trip must be served directly from the caches.
   3. To avoid serving stale data, the cache must have a timeout of 5 minutes.
   4. To avoid running out of memory, the cache must be limited in size.
   5. Whenever the customer updates or deletes a trip, the corresponding flight object must be invalidated from the cache.
3. With the help of logs, make sure that the caching mechanism behaves as expected.
4. Create a similar cache in the *HotelBookingService* in order reduce requests made to the DB.

# Load Balancing

To be able to support a high number of requests, we need to create a second instance of the flight booking application, and balance load between the two instances.

1. Duplicate the flight booking run configuration instance and set a different port for the second instance.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

1. Declare a second bean of type *FlightBookingClient* in the trip application configured with the port of the second instance.
2. Inject the second *FlightBookingClient* bean in the *TripBookingService*. The two *FlightBookingClient* instances must be used in a round-robin manner.
3. Run both flight booking instances and, with the help of logs, make sure that the load balancing mechanism behaves as expected.

# Downstream Resilience

1. Configure a time-out of 1 minute for requests made by the trip application to hotel and flight applications.
2. If a request made by the trip application fails, it must be retried with exponential backoff.
3. Simulate long running requests and failures on the hotel and flight applications to test the downstream resilience of the trip application.

# Upstream Resilience

1. Implement load shedding on the flight booking application. The application must not accept more than 100 concurrent requests. If the limit is exceeded, the application must return a 503 error without processing the request.
2. Implement rate-limiting on the hotel booking application. The application must only accept requests that have a header containing the API key. For each API key, the application should reject requests if we exceed the limit of 3 requests per second or 10 requests per minute.
3. Simulate a huge stream of requests from the trip application and make sure that upstream resilience on both flight and hotel booking applications works as expected.