Distributed Systems (H0N08A)

Report: JAVA EE

Robin Geelen *(r0664431)*
Jochem Hoes *(r0666420)*

November 20, 2020

Outline the different tiers of your application, and indicate where classes are located. Our application consists of three tiers. A Client tier which provides the necessary functionalities to a client. The Business tier contains our session beans (`ReservationSession` and `ManagerSession`) and all other business logic (`CarRentalCompany`, `Reservation`, ...). Finally we have a basic EIS tier for our Java DB database. Figure 1 shows a diagram of our application.

Why are client and manager session beans stateful and stateless respectively? For the client session, we want to remember the state between requests. For example, a client browses the rental agency's website and creates some quotes. These quotes are remembered during the whole session of the client. At the end, he/she can then create reservations for all the quotes he/she has obtained. For the manager session, we don't need this functionality. The manager session focuses on requesting statistical information and hence does not need to remember a state.

How does dependency injection compare to the RMI registry of the RMI assignment? Both can be used for retrieving sessions. However, dependency injection is a lot easier to use for the programmer because a JNDI lookup is performed automatically. With Java RMI, it was necessary to manually lookup the rental agency in the RMI registry based on its name.

JPQL persistence queries without application logic are the recommended approach for retrieving rental statistics. Can you explain why this is more efficient? The JPQL queries are translated to SQL queries. These are in turn executed by the database engine. This means that we can automatically use indexes and years of research regarding SQL optimisation. If we use application logic instead, we would need to optimise the code ourselves. Furthermore, if we write a loop in application logic, we need to fetch many entries from the database instead of the result only.

How does your solution compare with the Java RMI assignment in terms of resilience against server crashes? The current solution is more resilient to server crashes, largely due to the added persistence. When the Java RMI application server crashes, all data was lost because it was only stored in the server's objects. Another disadvantage of RMI is that the registry will contain a dangling reference to the rental agency when the server crashes.

How does the Java EE middleware reduce the effort of migrating to another database engine? All accesses to a database are abstracted by the JPA. This means that migrating can be as easy as changing the jdbc connection string in *persistence.xml*. The Java code does not need any changes. The queries defined in the code also need no adaptation because of the use of JPQL instead of actual SQL. The only thing that has to be converted is the database structure, as this depends on the database itself.

How does your solution to concurrency prevent race conditions? In the manager session bean, the method `addRentalCompany` is a transaction because it updates the database. Furthermore, all methods for statistics retrieving are transactions as well. This is to avoid phantom reads when multiple quotes are confirmed simultaneously in one reservation session. In the reservation session, only `confirmQuotes` is a transaction. All other methods do not require transactional support, because they involve tentative reservations only. If the result of `getAvailableCarTypes` is incorrect, as a result of an execution without a transaction, the effect is still minor. A client cannot distinguish between this situation and a situation where some other person reserves a car while the client holds the quote. Both situations will result in a `ReservationException` when he tries to confirm his quote. As we are also not directly modifying the database, the lack of a transaction on `getAvailableCarTypes` doesn't cause inconsistencies.

How do transactions compare to synchronization in Java RMI in terms of the scalability of your application? With Java RMI, it was not possible to do an automatic rollback at the end of the `confirmQuotes` method. Therefore, it was necessary to lock the whole car rental company before confirming quotes. With Java EE, rollbacks are possible and therefore it is not necessary to lock the whole car rental company when confirming quotes. This can make the application more scalable. The disadvantage of Java EE is that rollbacks might need to cascade. This effect is minor because we don't expect it to happen very often.

How do you ensure that only users that have specifically been assigned a manager role can open a ManagerSession and access the manager functionality? The security is enforced using the *RolesAllowed* and *DeclareRoles* annotations on the `ManagerSession` class. The accounts and their link to the defined roles are set in the GlassFish console, or alternatively in the deployment descriptor. We used the GlassFish console with default mapping because of its simplicity.

Why would someone choose a Java EE solution over a regular Java SE application with Java RMI? Java EE has several high-level features that Java SE doesn't provide. This makes it possible to develop a complex system in a relatively easy way. For instance, Java EE has support for security, persistence and transactions. If you want to use the same functionality in Java SE, you have to implement it yourself which is time-consuming and error-prone.

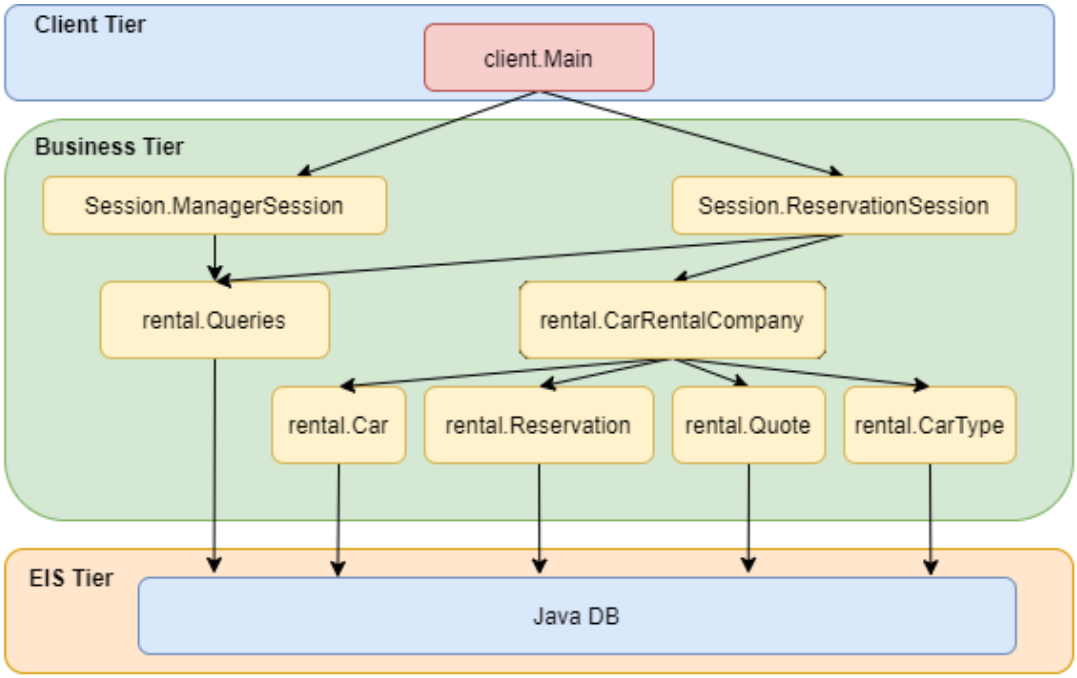

Figure 1: The different tiers of our JEE application.