SE325 Exam

Aiden Burgess - abur970 - 600280511

1.

a)

Caching can be used between the server and client, as well as between the server and the database. Caching improves scalability by storing requests, so that repeated requests can be executed faster and without putting additional load on the server. Therefore, the DB and service can acoomodate more users, which increases scalability.

b)

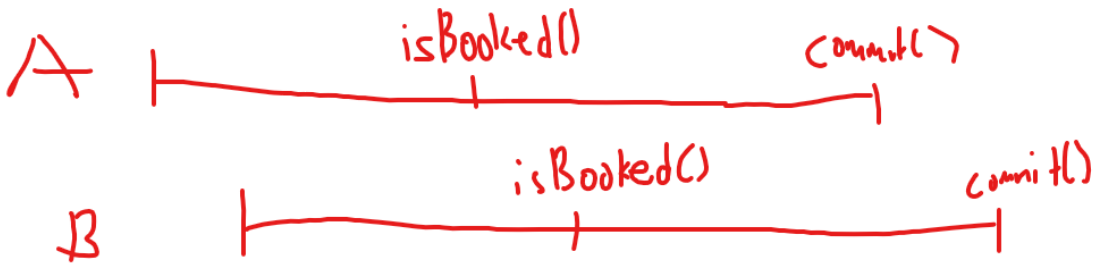
A DTO is an object which is used to transport data between different entities. An appropriate use of a DTO is to convert a database object into an object with is able to be used to send to a client over the network (e.g. Concert Service). The reason why the DB object can not be sent directly is that it is not serializable, and may contain irrelevant information that the client does not use.

c)

An open specification is a clear and detailed description of a system and its attributes, without specifying the exact implementation details. This allows for modifiability and portability, as different implementations can be used to substitute for each other, while having different characteristics.

2.

a)



If two transactions, A and B, were running concurrently as shown in the image, and both transactions wished to book the same seat. Both A and B would find that the seat is not currently booked, and set themselves as the User who has booked that seat. However, as B commits last, and there is no concurrency control, the seat would be booked by B even

though A tried to book first. This is a conflict as the user's A and B both believe they have the seat booked. This conflict is known as last commit wins.

b)

To introduce optimistic concurrency control, there must be a version field for the object being locked.

Insert into Seat (fig 1.) above line 7:

```
@Version
private long version;
```

The behaviour of fig 2. would change as follows:

- The transaction which commits first would have their changes saved to the DB and the version number for Seat would increment by 1.
- The transaction which commits second would try and commit, but would encounter a OptimisticLockException, as the version number on the Seat object has changed from the local copy being worked on.
- The second client could choose to try again, where they would find that the seat is booked and then abort the seat booking.
- Therefore, there are no consistency issues or conflicts in the DB.

c)

Pessimistic concurrency control works by utilising DB level locks on rows of the DB.

Therefore, at the start of the query, the Seat object being changed should be locked with a write lock..

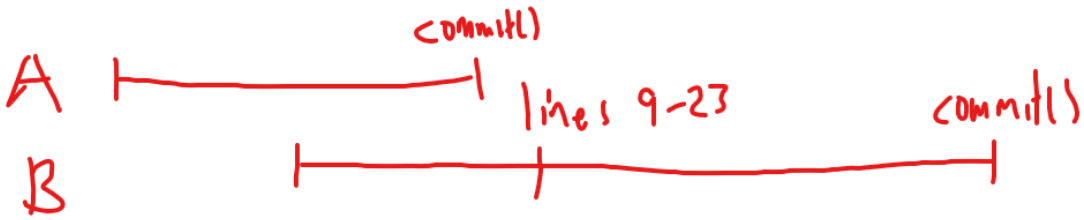
Replace line 7 (fig 2.) with the following:

```
Seat seat = em.find(Seat.class, seatId, LockModeType.PESSIMISTIC_WRITE);
```

The behaviour of fig 2. would change as follows:

- The transaction which executes line 7 (fig 2.) first would obtain the write lock on that DB row. Then, it would book that Seat and commit. At the commit(), the lock on that Seat is released.
- The transaction which goes to execute line 7 (fig 2.) would have to wait (blocked) until the lock obtained by the first transaction is released before obtaining the write lock on the Seat. Therefore, by the time it obtains the write lock, the seat is booked, so on line 9 (fig 2.) it would find that the seat is already booked and not book it.
- If the write lock takes too long for the second transaction to obtain, a PessimisticLockException
- Therefore, there are no consistency issues or conflicts in the DB.

d)



Although the Venue entity has optimistic concurrent control through the @Version annotation, the Concerts do not. Assume there are transactions A (fig 4.) and B (fig 5.). If A were to move around a Concert from one venue to another (lines 8-9 fig 4.) and commit() at the same time as B was performing its transaction, it is possible for a venue to be listed twice. The version number for Venue does not change for this transaction so there would be no OptimisticLockException thrown. This is a conflict as the information retrieved by B is not accurate to the actual DB state. This conflict is known as an phantom read.

e)

In order to make sure that changes are detected in Concert by fig 5., there must be a forced version increment when performing the transaction in fig 4.

Therefore, replace lines 4-5 (fig 4.) with:

```
Venue venue1 = em.find(Venue.class, 123,
LockModeType.OPTIMISTIC_FORCE_INCREMENT);
Venue venue2 = em.find(Venue.class, 456,
LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

The behaviour of fig 4. would change as follows:

- When the commit() occurs on line 11 (fig 4.), the version number of Venue 123 and Venue 456 would be incremented.

The behaviour of fig 5. would change as follows:

- When the commit() occurs on line 25 (fig 5.), a difference in the Venue version numbers for Venue 123 and Venue 456 would be detected between the DB and the local copy. Therefore, an OptimisticLockException would be thrown. This would stop the transaction and make sure that the inaccurate information is not recorded.
- The transaction can then be performed again to retrieve the information required.

This removes the conflict as the information obtained by the transaction in fig 5. is accurate to the DB, as it would be aborted if transaction 4 committed changes concurrently.

3.

a)

An asynchronous communication protocol would be beneficial for a notification option, e.g. app notifications, as the client does not have to constantly poll the server asking for new updates, which would reduce performance of the server, as it must use an extra thread for each of these requests.

b)

The AsyncResponse interface arranges for a response to be sent back to the client using a responder thread. The resume() call takes in a result to be passed back to the client. When the resume() method is called, the result being sent back is handled by a responder thread and sent back to the client using an HTTP response message.

c)

Asynchronous web methods require a single request/response pair. The AsyncResponse object is completed after resume() is called. So after each response, the client needs to resubscribe. This is a drawback as clients can miss methods by subscribing too late.

4.

a)

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "VTYPE")
public abstract class Vehicle {
    @Id
    @GeneratedValue
    protected long id;
    protected int year;
    protected String make;
    protected String model;
    protected long registrationNumber;
    protected int dailyPriceToRent;
    @ManyToOne
    protected Caryard caryard;
}

@Entity
@DiscriminatorValue("CR")
public class Car extends Vehicle {
    private boolean hasTowBar;
}

@Entity
@DiscriminatorValue("TK")
public class Truck extends Vehicle {
    private int cargoCapacity;
}

@Entity
public class Caryard {
    @Id
    @GeneratedValue
    private long id;
    @Column(nullable = false)
    private String name;
    @Column(nullable = false)
```

```
    private String address;
    @OneToMany (mappedBy = "caryard")
    private Set<Vehicle> vehicles;
}

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Person {
    @Id
    @GeneratedValue
    protected long id;
    @Column(nullable = false)
    protected String name;
    @Column(nullable = false)
    protected String username;
    @Column(nullable = false)
    protected String password;
}

@Entity
public class Customer extends Person {
    private String address;
    private String DOB;
    private String licenseNo;
}

@Entity
public class Employee extends Person {

}

@Entity
public class RentalAgreement {
    @Id
    @GeneratedValue
    private long id;
    @ManyToOne
    @Column(nullable = false)
    private Vehicle vehicle;
    @Column(nullable = false)
    private LocalDateTime startDate;
    @Column(nullable = false)
    private LocalDateTime endDate;
    @ManyToOne
    private Caryard returnCaryard;
    private int rentalCost;
    @ManyToOne
    @Column(nullable = false)
    private Customer customer;
    @ManyToOne
    private Employee employee;
    private boolean vehicleReturned;
}
```

Assumptions

A person must have a name, username, and password.

A caryard must have a name and address.

A customer does not need to have a address, DOB, and licenseNo.

A renrtal agreement must at least have a vehicle, start date, end date, and customer.

When a rental agreement is constructed, vehicle returned is initialised as false

Justification for Vehicle hierarchy

Table per class hierarchy

Good fit as majority of fields are in Vehicle, rather than spread across the inheritance hierarchy. Data integrity drawback exists, but is not a major issue. The registrationNo field would ideally be non nullable, although this is the tradeoff between

Justification for Person hierarchy

Table per class w/ joins

Only two subtypes, so performance hit from joining is not too impactful. Data structure is normalised so can enforce data integrity and non null columns

b)

HTTP Method	Resource	Required path, query, cookie params	Possible HTTP response status codes	Required HTTP request payload	Expected HTTP response payload	Other response data
POST	/users		201 - Created new resource 400 - Bad request (invalid username/password)	Username, password, name		Auth Cookie
POST	/signin		200 - OK 401 - Unauthorized	Username and password		Auth Cookie
GET	/vehicles	Auth cookie (Customer)	200 - OK 204 - No Content	search term caryard start date end date sort type sort direction	Vehicle array based on the search parameters, sorted based on sort type and sort direction	
POST	/rentals	Auth cookie (Customer)	201 - Created new resource 401 - Unauthorized	vehicle id start date end date	Link to rental agreement created	

HTTP Method	Resource	Required path, query, cookie params	Possible HTTP response status codes	Required HTTP request payload	Expected HTTP response payload	Other response data
GET	/rentals	Auth cookie (Employee)	200 - OK 204 - No Content 401 - Unauthorized	customer's username start date	Array of rental agreements based on the search critera	
PUT	/rentals/{id}	Auth cookie (Employee) Rental agreement id as path parameter	200 - OK 401 - Unauthorized	property to be updated and the value to assign	Link to rental agreement modified	

5.

No, a single picture is not enough to explain the architecture of even a very simple software system. Although the system is very simple, it still can not be fully described with just one picture. There are always at least two views: how the code is organised, and how the system looks when executed. These are shown through the static/module structures, and the runtime/component-and-connector structures.

The module structure is a code-based view of the system, and documents how the system is structured as a set of code or data units. This view can show modifiability, dependencies and functionality of the system.

The runtime view shows the security of the system at runtime through data flows. It also shows concurrency by showing the executing components at runtime.

Furthermore, there are also allocation structures, which can show elements of the system which are not software, such as hardware, development teams, and file systems. Even a very simple software system may have a team with each person having different responsibilities. If the team only had one person, it is still useful to document responsibilities so in the future these can be delegated or analysed.

So as can be seen, even for a simple system, a single picture does not show the entire architecture. For a complete view of the architecture, we need to know at least what the class structure is like, the runtime structure, the team and their responsibilities, the hardware that the system would be run on. These aspects can not be captured in a single picture.

6.

Overall, this depends **heavily** on the client and the system itself. Without knowing what the client wants, the quality attributes that are most important for software architecture are unknown. For example, when working with a bank, security and testability may be the most important quality attributes.

Therefore, the following discussion will be in **general terms**, as generally some quality attributes which have a large impact than others. These attributes are availability, performance, and security.

Availability

For most systems, the stability and availability is a significant design decision. If availability needs to be high, then a cloud or replica architecture would be useful. If availability is not important, then a single machine can be used to host the system.

Usually availability has a tradeoff with performance, as to increase availability, redundancy needs to be introduced.

Performance

To achieve other quality attributes, performance is usually sacrificed. For example, to increase security, extra layers may be introduced into the architecture which delay network traffic. Therefore, if a client needs performance to be high, then this would have a high impact on the system, as other attributes would need to be traded off. Furthermore, for the system to be usable, there is usually a baseline performance metric that must be met, so performance would be important to most clients. Therefore, performance is generally one of the most important attributes in choosing software architecture.

Security

Systems generally have security requirements, and there are many aspects of security which may be required by a client. A system usually needs to be safe from unauthorized access - whether that be internal or external. Although this is heavily dependent on whether there is sensitive information being used by the system. For example, a high frequency trading company is greatly dependent on external actors having access to their information. Security has a tradeoff with performance, as to increase security, the data being passed must be authorized and checked, which increases latency.

If the security requirements are low, this would also have a large impact on the system as the architecture could be greatly simplified. A simple personal desktop application could avoid the use of authorization and data protection.

7.

Which quality attribute scenarios

There would be interviews performed with the client to discover which aspects are most important to them (such as "Does the speed of the system matter as much as the uptime?"). However, even if one of the attributes is not a focus for the client, it should still be considered in the architecture design. For example, if the client is using the system personally and does not require any security, this should be noted as it may affect the design.

As the system is large, testability is likely to be important, as many different aspects are interacting with each other, it is important to ensure that no one change breaks the entire system.

How to identify exact scenarios

To identify scenarios for these quality requirements, the client/s should be communicated with to understand their expectations for the system. While discussing with the clients, common language should be used, instead of jargon. From this we can identify the general scenarios which the clients are interested in, and their basic expectations.

Then, these scenarios need to be precisely defined so that they can be measured. A concrete scenario should include: source, stimulus, artifact, environment, response, and response measure. By specifying these attributes, the architecture can be designed to meet these scenarios.

For example, a client may inform us: "The system should be able to handle many users, and we expect that the number of users could grow 100% within a day." This would be converted into a concrete scenario, and design decisions would be made to achieve this target. To meet this target, our architecture may be hosted on the cloud, to allow this scalability requirement.

8.

Pros/Cons of Broadcasting

Client will not notice if a server goes down.. If multiple servers go down, the client would still not notice as long as one server was still running. Therefore, degree of availability would depends on the number of servers (horizontal scaling). This means that there is very high availability.

Has high overhead, as each server communicates with each client. This leads to a scalability issue, where each client added would need to be handled by all servers simultaneously. Therefore, to increase performance and the number of users the system can handle, the system would need to be vertically scaled by continuously upgrading servers to have higher performance. However, this has a limit as hardware does not scale up in performance forever.

Pros/Cons of Dedicated Server

Can increase capacity for number of clients just by increasing number of servers (horizontal scaling). This increases scalability, as it is easier to increase the number of servers rather than increasing the performance of each individual server.

However, if each client is connected to a single server, then availability would be decreased. If the server crashes, then the client would immediately notice this downtime and would be negatively affected.

Given the same servers for broadcasting and dedicated server, we would expect that the dedicated server would have higher performance as the requests are not distributed to every server.

Circumstances

If a system requires high availability, such as critical infrastructure then broadcasting should be chosen. However, this would make the system difficult to scale. e.g. Stock Exchange or Bank

If a system does not require high availability, then a dedicated server would be preferable. This would increase the scalability greatly. So it should also be used if a high number of users is needed. e.g. Website Server

Another option is to combine the two to provide a hybrid approach which tries to balance the advantages of both approaches. For example, a client could connect to n servers at a time, where n can be adjusted for availability. It should be noted that this approach would require a load balancer to ensure that all servers are being utilised efficiently.