1. **How have you made an effort to improve the scalability of your web service?**

In the interests of scalability on the service layer, the concert service is mostly stateless. This allows for relatively simple replication as multiple servers do not have to share any specific data between them. This is because it is stored in a database which all servers could query meaning that each server would have replicability between them (as in your request will perform the same no matter what server your request goes to).
The only caveat to this is the subscription function, as currently each server would have a separate list of subscriptions. This would be problematic as the server would never know to send the notification out if it never received a booking for the concert that its users wanted notifications for because it will never check to see if it must send out those notifications. To work around this while the Async response itself can't be persisted, each sever could send a request to an update endpoint every time a successful booking was executed so each server would know to recheck its subscriptions.
To help with a potential database bottleneck, the database could be fragmented by its tables. For example, one database handles login and authentication only, one database handles concerts, performers and seats, etc. This would mean that the database would have to handle fewer requests and it would perform better than one database under the same heavy load.

2. **Identify (implicit and explicit) uses of lazy loading and eager fetching within your web service, and justify why those uses are appropriate in the context of this web service?**

Implicitly all fetches are lazy except for concerts_performers and concerts_dates. Concert was the only table I felt with enough data which is repetitively queried for the eager fetch type to be more appropriate. Most of the time when constructing the DTO's, not all the corresponding non DTO information was used so using lazy fetching to only get what you need seemed appropriate. While each concert usually has very few performers and could possibly use lazy fetching for it, I felt that in the off chance that there is a concert with a lot of performers, it may prove to be better performance wise(since it would also be a very popular concert). Concert dates made sense to be fetched eagerly as the server must check a date is in a concert's dates, which seemed like it would be heading to a N+1 problem.

3. **How have you made an effort to remove the possibility of issues arising from concurrent access, such as double-bookings?**

The issue with double bookings arises when multiple server would make a booking request, not see that either is making a booking and then have both bookings succeed. To try and prevent this problem I have used pessimistic concurrency control. When the server first looks at the seats to see if they are not booked it will acquire pessimistic write locks on the data base and after updating it will release those locks. Therefore, in a double-booking scenario one of the processes will be blocked until the other one is done so when it then checks the data base it will see the booking of the other and will not make that booking.

4. **How would you extend your web service to add support for the following new features?:**
    - **Support for different ticket prices per concert (currently ticket prices are identical for each concert)**

If the webservice had to support different ticket prices per concert, the only changes would have to be in the concert_init file where the prices for each seat is set. The code would have to be altered to create different prices based on the concert id. It may also be practicable to have each concert store the price of each of their seating bands for this purpose.

- **Support for multiple venues (currently all concerts are assumed to be at the same venue with an identical seat layout)**

Support for multiple venues would require a little bit more variation. Seats would now have to store their venue (probably as an Enum type) as would concerts. Concerts would now have to make sure that each concert had a unique combination of date and venue instead of just date and seats would still have a composite key of all its column values. Bookings would also need to know what venue they have been made for, and subscriptions

- **Support for "held" seats (where, after a user selects their seats, they are reserved for a period of time to allow the user time to pay. If the user cancels payment, or the time period elapses, the seats are automatically released, able to be booked again by other users). would also need to know.**

Support for held seats could be implemented by having the seats store a local date time variable availableAfter. When holding the seats, the request would write to the database the value it would be available after. When a booking would be requested by another server, in addition to checking the booking status, it would check that it is available by checking the current time relative to when it is available from to see whether it could make a booking for it. Then if need be it could hold those seats by overwriting the availableAfter column.