Axiomatic + inferred invariants (would normally be clarified via BA, instead assumed + inferred by dev here):

1. No guest login/registration is mentioned and no guest-booking list function is required.
2. We have strict knowledge of hotel room-count being uniform across all hotels - this number is very low.
3. This app is for multi-hotel use, due to "Find a hotel based on its name." requirement.
4. Via document - The API requires no authentication.
5. (Via 1, 2, 3) This API concerns a specific hotel franchise.
6. (Via 1, 4, 5): This is a back-office API, for use by staff and not the public.
7. (Via 2, 5, 6): Booking throughput will be low.

-> Max 6 bookings per day, per hotel (due to 6 rooms)

1. Availability searches may not be low.

-> Proportional to franchise size and consumer demand.

1. The ability to identify and contact the guest is desirable.

-> Guest details will be collected, but will be held as a booking attribute - utilising absent criteria (#1) to speed up development.

-> Guest info constitutes PID under GDPR so it must be minimised and for legitimate processing use only.

-> Assume a data processing policy to be written and made available to users to this API, and consent gathered via T&C.

-> Guest details will contain a lead-contact number - this is for a practical purpose - guest contact. It is also a good candidate to derive a PK from for a Guest entity if a requirement for this emerges in future development.

1. No mention was made of this being an international franchise, so time-zone considerations will not be included (YAGNI).

-> Project will still use time-agnostic or offset-capable date objects because it will not add extra dev overhead.

DB design notes:

* Using a string "name" for room numbers - ordering is not expected to be important and non-numeric names may exist (named suites etc)
* Booking ID is of GUID type rather than int - this allows the ID to be output as the booking reference while defending against booking enumeration/volume estimation - useful if these references are made public (sent back to guests).
* Address info added to hotels, otherwise "Find a hotel based on its name" requirement doesn't have much meaning outside of just recovering its ID via API.

API design notes:

* I would normally add in-line documentation to every element in the codebase. I cannot do this due to time constraints.
* I would normally create a software guidebook containing design info via C4 model + UML and discussion around technical requirements and constraints. I cannot do this due to time constraints.
* I would typically include a "Domain" project, and use it to abstract Controllers and DAL: I would use the mediator pattern (via MediatR) to decouple ASP from the domain, and Repository classes to separate EF from the domain. Such a design should end up conforming to ports-and-adapters/hexagonal and would reap those benefits. Because ASP and EF are stated inseperable requirements (and also to save time), I am eschewing this architecture in favour of something more quick and dirty. Domain logic will be held in the top-level project and a reference to the Data project will be made directly.
* Domain logic currently resides within controller code. This violates SRP but massively speeds up development. It should be considered tech debt and rectified.
* I would normally work via TDD where feasible – the design compromises in this project make it harder to do so (e.g. can’t unit test domain objects). I won’t be standing up an SIT environment to run automated tests against the API either, so I have gone forward with a manual testing approach as implicitly endorsed by the doc - this means there’s no automated backstop against regression.
* I will implement "Find a hotel based on its name." via partial name search, as in my experience that is more natural for a user than typing out the full name.
* Due to #7 and #8, there's a discrepancy between read and write volumes. I'm going to implement this solution as a monolithic API app because it's going to be significantly quicker to do and should still satisfy the expected load. Future upgrades may involve a conversion to microservices so they can be differently provisioned and lends well to an implementation of CQRS to take advantage of that discrepancy.
* I'm not going to implement caching due to time constraints and also due to low expected load - if expected load exceeds design it could be added in. Hotel data should remain static so it can be held in cache with a long TTL and no explicit invalidation. Booking data is variable so needs active invalidation on write - writes will be quite infrequent (max 6 per location per day) so significant hit-rate expected.