# Decision Register

## Structure

1. Coding challenge specific files such as decision registers will be included in the repo, although they are not project files; so they can all be accessed easily from a Git pull

## Code

1. Metadata such as the number of floors in the building will be stored in config, not the database
2. Elevator details will be stored in the database, since this is not volatile data and should be preserved even if there is a system failure. State data such as an elevator’s current floor and load will not be written to DB each time, as this would significantly slow down the application and increase data transport and storage costs. In the event of a system failure/reboot, elevators could be reset to a default state (eg. returned to ground floor) as a way to deal with the loss of state; other low risk options could also be explored.
3. Although console “ReadLine” operations aren’t strictly async, operations in the master elevator control as async to allow for future extensions with different UI types, or multiple elevator calls

## Database Design

1. Since data size is not a primary concern, strings will be stored in the DB as nvarchar rather than varchar, for better compatibility
2. Since elevator capacity could be specified as an integer (eg. number of passengers) or a floating point number (eg. weight), the max capacity field in the DB will be of type float. Having different DB structures for different elevator types is unnecessarily complicated and does not scale easily, and integers can be represented as floats without boxing.
3. Model and serial number are not specified in the requirements but it’s very likely to be relevant; it is easier in the long run to implement the fields in the models now and not use them than update all the models and CRUD operations later. In a real life scenario, this business rule would be verified with the client or Product Owner during grooming prior to implementation.

## User Experience

1. Although user messages are pulled in from a list of phrases for future localization, severe error messages and exceptions don’t support translation. These are aimed at admins, technical operators and developers rather than front end users.

## Future Features

1. A reset function that could return all physical elevators to a default state, e.g. ground floor, to deal with a system failure/reboot or other scenario in which the software and hardware could get out of sync.
2. Security features such as role based authentication and authorization, to ensure only users with the required credentials can perform different levels of actions (e.g. passengers should be able to call an elevator; admins should be able to interrogate the status of all elevators; superusers should be able to perform a reset).
3. A graphical observability interface to represent the elevators with their states visually.
4. Dictionaries of phrases can be added for different languages to enable localization.
5. Since there is no requirement for database management such as archiving, deleting an elevator currently does a hard delete and removes it permanently from the database. In future, they could be soft deleted instead (moved to an archiving table, marked deleted etc).
6. The database should be indexed on those fields likely to be used most often for lookups, such as ElevatorID.
7. For this MVP version, the only setting related to floors is in the config, to indicate how many floors there are (N). The application assumes a range of 0 – (N-1), with all floors accessible to the elevators. DB structures could be added to specify all floors with their friendly name, if the elevators should stop there, mobility/accessibility criteria, etc.
8. An accessibility mode could be added that would let users call only lifts that are wheelchair friendly, have braille buttons, etc. A screenreader interface could be added, since the console interface is loosely coupled from the business logic.