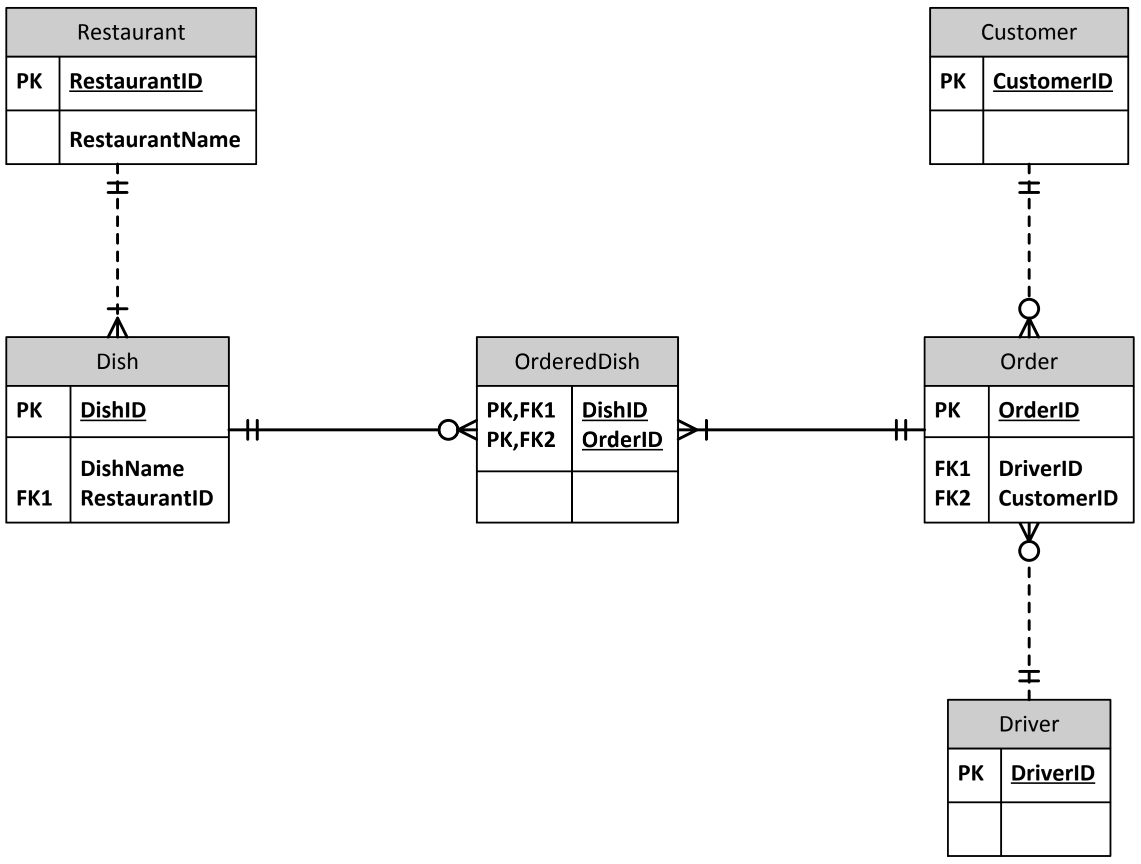
FineFoods4U ERD

This is the basic structure of the ERD. I've only shown primary keys and foreign keys for simplicity here (and so as not to give you a complete solution ☺), but I'll discuss some of the attributes later.

This is the simplest interpretation of the case:

- Restaurants have potentially many Dishes on offer. They must offer at least one dish, otherwise they won't be in the system. Each Dish would be from a single restaurant, as even though different places might offer similar dishes (fish and chips) they are unlikely to be identical.
- A Dish can appear in many Orders, and each Order would include at least one and possibly many dishes. We represent this many-to-many relationship with another entity, DishOrdered.
- A Customer may make many Orders over time. They can register without making an order. An order is for only a single Customer.
- A Driver might deliver many Orders over time, but each Order is delivered by a single Driver. There may be drivers in the system who haven't made any deliveries yet.

I've used IDs for primary keys throughout. There aren't any single attributes that are likely to be unique: two restaurants may have the same name, and two dishes may have the same name. There may be chains of restaurants so that even the combination of RestaurantName+DishName isn't unique.



Variations:

- Recording multiple addresses for a Customer. The customer would have several addresses registered, and would choose one of them for an order.
- Assuming that a driver would take orders for different customers on a single delivery run. Ask if any additional information would need to be stored if this was the case.
- Separate Order and Delivery entities. Several of you did this, with Order and Delivery related in a 1:1 mandatory relationship. Although this would work, there is nothing to be gained by separating the entities as both would always be completed for a completed order delivery.

Entities that SHOULDN'T be included:

- Booklet – the booklet referred to is similar to a report, which can be assembled dynamically from the information in the other entities when needed.
- Website – this is an implementation decision. At this stage, we are modelling the data requirements of the system.
- Anything to do with payment – we are told it is out of scope.

Multivalued attributes

Ask whether each attribute you put in an entity will be single valued or could have several values. For example, if you had an attribute 'Certifications' in Restaurant then it's likely that it could take more than one value – e.g. it could be certified both Vegan AND Organic. You can't have a multivalued attribute in a relation, so you need to do something else with it.

- Assume it can only take one value? No, really not a solution.
- Assume it can only take a maximum of 2/3/4 values, and have attributes Cert1, Cert2, Cert3, … Not a solution. Why 2/3/4? Not extendible to more values without changing the structure of the table.
- Have a long text string for the field and try to search within it? … Gets very messy. Not a solution.
- Have a weak entity Restaurant_Certification in a 1:N relationship with Restaurant? Restaurant_Certification would have attributes RestaurantID, CertificationType, DateCertified [etc]). This could work, as it would allow you to add as many certifications as you like for each restaurant. The main issue with this solution is that it doesn't enforce any consistency in the CertificationType attribute – there is nothing to stop typos such as Orgnanic, resulting in incorrect information returned from queries.

- Have an entity Certification in a M:N relationship with Restaurant, with an intersection entity Restaurant_Certification? Certification would have attributes CertificationID(PK), CertificationType. Restaurant_Certification would have PK RestaurantID+CertificationID, and other attributes could be included in this entity. This is the best solution, as it enforces consistency of CertificationType through the foreign key. This comes at the expense of increased complexity and more joins, so Bill will have to weigh up better data against possibly reduced performance.

Constraints on attributes

If you've got an attribute that takes a single value (such as Suburb in Restaurant) but which has a fixed set of allowable values, you need to be able to enforce that constraint. This isn't strictly part of a conceptual ERD, but it is worth thinking ahead to the options for the logical design and implementation.

- Use a CHECK constraint. This is best for a small, limited set of values that isn't going to change, or you will need to alter the constraint every time a new value comes along.
- Use a 'lookup table'. This is a table containing only the attribute for the values plus a PK attribute, e.g. (<u>SuburbID</u>, Suburb). The attribute Suburb in Restaurant is then replaced with SuburbID as FK. The two tables are joined to find the restaurant suburb name. Again, this enforces consistency at the expense of increased complexity, but is a good solution where there may be a large or increasing set of values. It's also a good solution if the same constraint applies to more than one table in the database.
- The lookup table solution can be simplified to include only the attribute (in this example, Suburb), in the lookup table. Suburb is PK of the lookup table, and the attribute Suburb in Restaurant is defined as FK referencing it. Consistency is still enforced through referential integrity, but no joins are required.