Ruby Tests

1.
Show the code you would use for a database migration that added a field “permalink” to an existing table called “post”. The migration is to also create a permalink for every existing Post record based on the “title” field. You can use any method you want to convert title to become permalink, the only requirements being permalink must be based on title, must be constructed of only URL valid characters, and must be unique (even if another Post has the same title and converts to the same permalink).

2.
Credit cards use a check digit algorithm called the Luhn algorithm. Write a function that takes one parameter, the credit card number, and returns true or false depending on whether the number is a valid number. Similarly, write a second function that takes a number of any length and calculates the Luhn check digit and returns the original number with the check digit appended on the end.
For information on the Luhn algorithm:
http://en.wikipedia.org/wiki/Luhn_algorithm

3.
Find or create an object by a unique key and update it without introducing any extra objects.

Let's assume that an external API is being polled for payments. The actual external API is irrelevant to this task,
what really matters is that each payment has a line_item_id and belongs to a service.
A line_item_id is unique in the scope of a service.

Assume that you are already given an ActiveRecord object (and underlying table) for representing a payment.
It was already designed by another team member, it has been deployed and is being actively used in production.

```
class Payment < ActiveRecord::Base
belongs_to :service
end
```

Initially the team of developers had ignored the possibility of races and did not provide any means of guarding against concurrent access to payments.
However, then our app grew up to a point where multiple processes routinely poll this external API and might simultaneously receive data for the same payment.
The naive existing solution results in sometimes multiple copies of the same payment, which ruins balances, etc.

Please design the solution that would ensure that no duplicate Payment (with the same line_item_id, service_id) objects can exist in our system.
You can assume that we use PostgreSQL as our database and you can use all its features to help you in the task.

Ideally, your solution should be re-usable, so that the actual logic of updating the Payment could be separated from all the details of how you
implement serializability.
Here's an example of how your code could be included into the older (non-race free) code:

```
# we received a payment with line_item_id and service_id from external API
# if such payment has already been seen, we should access the existing object
# if no such payment exists in our db yet, create it
# in both cases the block parameter payment should be the payment object
# which the block code will update with data received or whatever
Payment.with(:line_item_id => line_item_id, :service_id => service_id) do |payment|
# the bulk of old code here - just updating the payment data here
end
```

The idea is that Payment.with should just incapsulate any logic required to prevent multiple objects creation/finding,
and the block code does more mundane tasks such as actually updating the payment's other fields or determining if they need to be updated, etc

You can suggest other way too - as long as the access to a Payment is serialized and the following invariant holds:
If two or more processes started to update the same logical payment (line_item_id, service_id) simultaneously then
there should not be 2 or more payments afterwards, instead each process should wait its turn, update the object and let the next one(s) do its(their) job.
If the payment did not exist, one of the processes will create it and other processes should wait their turn and use that exact payment object for update.