***Page Caching***

***Page caching is a Rails mechanism which allows the request for a generated page to be fulfilled by the webserver (i.e. Apache or NGINX) without having to go through the entire Rails stack***

***it can't be applied to every situation (such as pages that need authentication).***

***Also, because the webserver is serving a file directly from the filesystem you will need to implement cache expiration***

***Action Caching***

***Page Caching cannot be used for actions that have before filters - for example, pages that require authentication***

***Action Caching works like Page Caching except the incoming web request hits the Rails stack so that before filters can be run on it before the cache is served***

***This allows authentication and other restrictions to be run while still serving the result of the output from a cached copy***

***Fragment Caching***

***Dynamic web applications usually build pages with a variety of components not all of which have the same caching characteristics. When different parts of the page need to be cached and expired separately you can use Fragment Caching.***

***Fragment Caching allows a fragment of view logic to be wrapped in a cache block and served out of the cache store when the next request comes in.***

***Filters vs Callbacks in Rails***

**Filters: Filters are methods that are run before, after or “around” a *controller action***

***before\_action Before filters are run on requests before the request gets to the controller’s action eg:validating a user’s authentication before granting them access***

***after\_action After filters are run after the action completes.***

***around\_action Around filters may have logic before and after the action being run. You can use around filters for exception handling, setup and teardown, and a myriad of other cases.***

***Callbacks: Callbacks allow you to trigger logic before or after an alteration of an object's state in model.***

***It is possible to write code that will run whenever an Active Record object is created, saved, updated, deleted, validated, or loaded from the database.***

***You use git rebase if you want your changes in the feature branch to always the latest. And you use git merge if you want to reflect the true ordering of commits.***

***What is a Merge?  
When you merge one branch into another branch the branch being merged into receives a single commit that basically brings it up to the current state of the source branch.***

***This commit is created automatically by git and essentially represents all of the differences between the branches stuffed into one change. If you were to `git log` in the branch that was merged into, you wouldn’t see any of the commits that had been made in the source branch.***

***What is a Rebase?  
With rebase you basically say “use another branch as new base for my work”.  Behind the scenes what appears to be happening is that git takes the current branch and rewinds any updates you have made on it.  It then moves the starting point to the head of the branch you’re rebasing against.  Finally all of the rewound changes are then applied on TOP of the updated/new starting point***

**SOAP (Simple Object Access Protocol) is a messaging protocol that allows programs that run on disparate operating systems (such as Windows and Linux) to communicate using Hypertext Transfer Protocol (HTTP) and its Extensible Markup Language (XML)**

**What is Mixin?**

**Multiple Inheritance**

**Difference between include and extend?**

As a general rule of thumb, you use extend for class level methods, and include for instance level methods. This isn’t always the case, as we’ll see in a later example. For now, let’s just assume this to be the true. Here’s an example that demonstrates this:

module SayHello

def hello

puts 'hello'

end

end

module SayBye

def bye

puts 'bye'

end

end

class Greeter

include SayHello

extend SayBye

end

Greeter.bye # class level

bye

=> nil

Greeter.new.hello # instance level

hello

=> nil

This demonstrates the simple way to handle the difference between instance and class level methods. More often than not, you may want a module to be able to have instance AND class level methods, so you can just mix that module in, and have the class gain both. This is possible with this slightly more advanced example:

common mixin usage

module SayHello

def self.included(base)

base.extend SayBye

end

module SayBye # all class level methods

def bye # class level method

puts 'bye'

end

end

def hello # instance level method

puts 'hello'

end

end

class Greeter

include SayHello

end

Greeter.bye # class level

bye

=> nil

Greeter.new.hello # instance level

hello

=> nil

This acheives the same effect as the first example, but with a more simple interface. Instead of having to extend and include 2 seperate modules when you include SayHello, it automatically extends the sub-module SayBye. This is accomplished using the self.included method, which rails calls whenever a module is included. From inside this method, we capture the base object and tell it to extend our class level methods in the SayBye module. In our example the base object is the Greeter class.

**Object Relational Mapping**

Object Relational Mapping, commonly referred to as its abbreviation ORM, is a technique that connects the rich objects of an application to tables in a relational database management system. Using ORM, the properties and relationships of the objects in an application can be easily stored and retrieved from a database without writing SQL statements directly and with less overall database access code.

**Active Record as an ORM Framework**

Active Record gives us several mechanisms, the most important being the ability to:

• Represent models and their data.

• Represent associations between these models.

• Represent inheritance hierarchies through related models.

• Validate models before they get persisted to the database.

• Perform database operations in an object-oriented fashion.