**Documentation**

Before getting into the details of the code and how things run. I would like to give the list of the framework/software’s I used to build this solution.

Instead of sending only the code I thought It would be better if I could send you the image of the entire dev environment, so that the solution can be testable. So I used vagrant to configure a VM that runs all the softwares (mysql, php, apache etc.) and can be started with a very little effort.

Following are the frameworks/software’s I used to build the solution

**VirtualBox**

VirtualBox is a powerful x86 and AMD64/Intel64 [virtualization](https://www.virtualbox.org/wiki/Virtualization) product for enterprise as well as home use.

**Vagrant**:

Vagrant provides easy to configure, reproducible, and portable work environments and controlled by a single consistent workflow to help maximize the productivity and flexibility. Vagrant stands on the shoulders of giants. Machines are provisioned on top of VirtualBox, VMware, AWS, or [any other provider](http://docs.vagrantup.com/v2/providers/)

**Chef** solo:

chef-solo is an open source version of the chef-client that allows using cookbooks with nodes without requiring access to a server. chef-solo runs locally and requires that a cookbook (and any of its dependencies) be on the same physical disk as the node.

Recipes used: Apache2, Mysql, Php, Openssl, and My custom recipes for db setup and web server setup

**Kohana**

An elegant HMVC PHP5 framework that provides a rich set of components for building web applications.

**Setting up the environment:**

This is gonna take a few minutes of your time to set it up.

1)Download/Install Virtual box.

<https://www.virtualbox.org/wiki/Downloads>

2) Download/Install Download/Install Vagrant

<http://downloads.vagrantup.com/tags/v1.3.5>

3) Clone repo from github

<https://github.com/furion-np/uber.git>

4) Go into the folder Uber and run

**Uber kedari$** vagrant up

5) SSH into VM

**Uber kkedari$** vagrant ssh

6) Log into app folder

vagrant@precise64:~$ cd app

7) run sql script for sample data

vagrant@precise64:~$ mysql –uroot –proot < Uber.sql

**API:**

With the earlier setup the server would be hosted on

<http://localhost:8080>

Following are the API available

GET <http://localhost:8080/api/cabs> - retrieves all cabs

GET [http://localhost:8080/api/cabs ?latitude=12.23&longitude=1.23](http://localhost:8080/api/cabs%20?latitude=12.23&longitude=1.23)  -retrieves cabs nearer sorted by distance ascending order

GET <http://localhost:8080/api/cabs/cabid> - Retrieves cab data for that id.

PUT <http://localhost:8080/api/cabs/cabid> - update/create for cab id

DELETE <http://localhost:8080/api/cabs> - deletes all cabs

DELETE <http://localhost:8080/api/cabs/cabid> - deletes one cab record

FOR UX use

GET [http://localhost:8080/api/cabs ?latitude=12.23&longitude=1.23&type=html](http://localhost:8080/api/cabs%20?latitude=12.23&longitude=1.23&type=html)

**Design:**

The solution is based on a MVC pattern using Kohana framework.

**Modules:**

modules/cabs

modules/core

are the modules that were built for the cabs API.

Core module is for setting up filters for the restful route processing and also abstract model that any model in the system can make use of.

Cabs module has the following components

**init.php** – The php file where the RESTful Routes are configured.

**Classes/Controller** – The controller for the cabs API.

**Classes/Uber** - Place for models and Mappers.

**Classes/Task** - Any CLI tasks required for the solution. Currently none but stubbed it out.

The sample task I creted can be run as

**vagrant@precise64**:~$ ./app/modules/minion cabs:sampledata

**Classes/Unit-tests** - Diretory for unit tests. Stubbed out but not implemented.

Classes/Functional-tests - Directory for functional tests. Stubbed out but not implemented.

**Dependency Injection and Dependency Container:**

I am huge fan of this design pattern. I implemented the dependency container.

application/config/dependencies.php

application/config/dependencies-cabs.php

these are the DI containers for mappers, databases. I configured the mapper in the DI container and the mapper object cab be instantiated as

Kohana::$di\_container->get('mappers.cabs');

That would also inject db instance into the constructor, This would make it easy for me to mock the objects and unit test the classes and get 100% coverage. I feel unit tests are really great stuff to maintain a clean code base.

**Views:**

**modules/cabs/views**

This is the directory for the views and I created one simple view using jQuery and Php .

Please pass &type=html (I used the same controller but switched views).

to the GET api.