# TMC API Service Challenge

## Introduction

Your company has gone SOA, all the way. It started off with so much promise: microservices as far as the eye could see, fully decoupled teams, and reusability out the wazoo.

Slowly, chaos begin to creep in. These services were not quite similar enough to make sense. Service A was written in Go and UTF-8 friendly, while Service B was written in Node and ASCII only. You had n services and 2n + 1 databases. It’s time to add some sanity!

Your job is to bring up a simple caching service. Your service will exist solely to get, put, and delete data into and out of a cache. All these methods will support a clean and transparent contract to make cache interactions as simple as possible.

Your clients will push you different types of data (all valid JSON), they’ll do it concurrently, and they expect a fast response, so choose an appropriate language.

## Deliverables

Please provide an archive file with:

- Your source code
- Build/deploy instructions needed to spin up your solution
- README.txt that details how you tested your solution, as well as any other explanation tasks requested in this challenge
- This document to avoid ambiguity where multiple versions of this document may exist

We will be looking at idiomatic consistency, cleanliness, documentation and implementation.

## Endpoints

Your caching service will operate at http://localhost:8088/. Please provide the following endpoints which handle the following HTTP verbs:

- /cache/
    - POST - creates a new item in the cache. The body of the POST should match the contract specified in the Contract section.
    - GET - gets the entire cache.
    - DELETE - deletes the cache.
- /cache/{key}
    - GET - gets the fully-specified key / value pair.
    - PUT - updates the cache item with matching key. The body of the PUT should match the contract specified in the Contract section.
    - DELETE - deletes the cache item with matching key.

# Contract

Your cache service will expect input in valid JSON and emit output in valid JSON. A single cache item will always be described thusly:

```
{
    "key": "{key}",
    "value": "{value}"
}
```

An example payload:

```
{
    "key": "problem_free_philosophy",
    "value": "Hakuna Matata"
}
```

Keys and values may be any string, boolean, integer, or decimal value.

Any service call that returns more than one cache item will return a “cache” key with an array of the JSON object above:

```
{ "cache": [{
            "key": "foo",
            "value": 3.9999
        },
        {
            "key": "bar",
            "value": true
        }]
}
```

## Additional Specifications

Implement at least 2 of the following 4 requirements:

## Requirement #1

- A successful POST should return status code 201 with a link to the new cache item’s key in the Location header
- A POST should return 409 if a resource with that key already exists
- GET, PUT, and DELETE should return 404 if passed in a cache item key that does not exist in the cache
- If invalid JSON is passed in to a PUT or POST, return a status code 406
- What HTTP status codes are missing from this list? Implement them and document them in the README file

## Requirement #2

Ensure the contents of your cache survive a clean restart of the machine hosting the cache service.

Discuss a scenario where your service may not shut down cleanly. To what degree and in what ways can you handle these scenarios? What are the potential risks/complications when your application unexpectedly shuts down?

## Requirement #3

When the HTTP verb is POST or PUT a cache item should be able to accept an expires field (in number of seconds) which defaults to 0 (never expire). The item will remain cached for the duration of N seconds after a successful POST or PUT completes and MUST NOT be returned beyond N seconds.

Furthermore, subsequent PUT requests for a cache item that do not explicitly set an expires value should cause the cache to reset the expiration of the item based on its previous expires value. A new expires value in a PUT request will take precedence over any previously set value.

```
{
    "key": "{key}",
    "value": "{value}",
    "expires": {seconds}
}
```

The return value from any POST, PUT or GET on a cache item that has an expires value should display a human readable datetime string for when that cache item will expire. If a cache item is not set to expire, the “expires” field SHOULD NOT be returned.

```
{
    "key": "session_token",
    "value": "cf23df2207d99a74fbe169e3eba035e633b65d94",
    "expires": "2015-11-10 23:00:00 +0000 UTC"
}
```

## Requirement #4

Implement an additional endpoint that provides some basic usage metrics for the caching service—system uptime, for example. Identify 5 possible metrics and implement 2 of them. Explain why the two metrics you implemented took precedence over the other three.