Rate Limit

Source code of released version | Source code of development version *** === A Rate Limiter is a general rate limiting object that stores rules and determines whether inputs are allowed based on the rules. There is also a general structure of Rules which contain all the internal state of a rule.

Rate limiters analyze a series of "inputs" (which are POJOs) by running them against a set of "rules." Rules specify which inputs they match by running configurable "matcher" functions on keys in the input object. A `check` method returns whether this input should be allowed, the time until the rate limit is reset and the number of calls remaining for this input. The count of processed inputs are kept in a dictionary of counters stored inside each rule, keyed by a unique string composed of the input that matched to the rule.

Rule Structure

Each rule is composed of an `id`, an options object that contains the `intervalTime` in milliseconds after which the rule is reset and `numRequestsAllowed` in the specified interval time, a dictionary of `matchers` whose keys are searched for in the input to determine if there is a match. If the values match, then the rule's counters are incremented. Values can be objects or they can be functions that return a boolean of whether the provided input matches. For example, if we only want to match all even ids, plus any other fields, we could have a rule that included a key-value pair as follows:

```
{
  ...
  id: function (id) {
    return id % 2 === 0;
  },
  ...
}
```

A rule is only said to apply to a given input if every key in the matcher matches to the input values. There is also a dictionary of `counters` that store the current state of inputs and number of times they've been passed to the rate limiter. Each rule defines a domain of keys and values that it applies to, and we want to have a unique way of recording each input provided to the Rate Limiter that matches to the rule. Say a rule inspects a methodName property and a username property. We want to count how many times each user called a certain method and restrict them to a certain number of calls per user defined time frame. So we generate a unique string key (to be used as keys in a counters object) to represent each specific methodName + user combination. Since this rule applies to multiple user, we need to concatenate the different input key names with their values. For example, if we had a rule with matchers as such:

```
{
  username: function(username)  {
    return true;
  },
  methodName: 'hello'
}
```

and we were passed an input as follows:

```
{
  username: 'meteor'
  methodName: 'hello'
}
```

The key generated would be 'usernamemeteormethodNamehello'. This is guaranteed to be unique for this username+methodName combination. These keys are cleared every time the intervalTime is passed, at which point we delete the current dictionary of counters we store. Every time a rule matches to an input, we determine the unique key string and check if it's counters have exceeded the allowed amounts, returning an error to the user letting them know that a rate limit has been reached.