1.	Start MongoDB
1.1	start up the vagrant box - navigate to the redis/vagrant folder: **cd mongodb/vagrant** - enter the command: **vagrant up**
This will now download and install the vagrant box, this might take a while	
1.2	start up the mongodb shell. - enter the command: **vagrant ssh** - you will now go the the vagrant box - correct a local setting via: **export LC_ALL=C** - start the shell via entering: **mongo** You should see >`>` If you see an error: Failed global initialization: BadValue Invalid or no user locale set. Please ensure LANG and/or LC_* environment variables are set correctly Then first run: `export LC_ALL=C` Followed by **mongo**
1.3	let us first switch to the test database via the command: **use test** this will represent the message: **switched to db test** If a database does not exist - mongodb will automatically create it - it is very flexible.
2.	Insert a document
2.1	Insert a document into a collection restaurants **db.restaurants.insert(** **{** **"address" : {** **"street" : "2 Avenue",** **"zipcode" : "10075",**

	"building" : "1480",
	"coord" : [-73.9557413, 40.7720266],
	},
	"borough" : "Manhattan",
	"cuisine" : "Italian",
	"grades" : [
	{
	"date" : ISODate("2014-10-01T00:00:00Z"),
	"grade" : "A",
	"score" : 11
	},
	{
	"date" : ISODate("2014-01-16T00:00:00Z"),
	"grade" : "B",
	"score" : 17
	}
	],
	"name" : "Vella",
	"restaurant_id" : "41704620"
	}
	)
	
	The method returns a WriteResult object with the status of the operation:
	WriteResult({ "nInserted" : 1 })
	
	If the _id field is not passed in as an argument, mongodb will automatically add it.

2.2	To find documents, just use the find command: **db.restaurants.find()** **db.restaurants.find({ "borough": "Manhattan" })** **db.restaurants.find({ "address.zipcode": "10075" })** Via the dot annotation you can query on properties of properties **db.restaurants.find({ "grades.grade": "B" })**

	At the .pretty() function to make the output more readable **db.restaurants.find({ "grades.grade": "B" }).pretty()**
2.3	The greater then operator $gt **db.restaurants.find({ "grades.score": { $gt: 3 } })** You also have a less then $lt operator
2.4	AND clause Just separate the properties by a , in the query: db.restaurants.find({ "cuisine": "Italian", "address.zipcode": "10075" })
2.5	Logical OR Via the $or operator you can specify an OR clause **db.restaurants.find(** **{ $or: [{ "cuisine": "Italian" }, { "address.zipcode": "10075" }] }** **)**
2.6	Sort query results To specify an order for the result set append the sort() method to the query. Pass the fields to sort by and the sort type (1 for ascending and -1 for descending) **db.restaurants.find().sort({ "borough": 1, "address.zipcode": 1 })**
2.7	Update fields: The following operation updates the first document with name equal to "Juni", using the $set operator to update the cuisine field and the $currentDate operator to update the lastModified field with the current date. **db.restaurants.update(** **{ "name" : "Juni" },** **{** **$set: { "cuisine": "American (New)" },** **$currentDate: { "lastModified": true }** **}**

	) Embedded fields can be updated via the “dot” operator: db.restaurants.update({ "restaurant_id" : "41156888" }, { $set: { "address.street": "East 31st Street" } })
2.8	By default update only updates a single document. In order to update more then 1 document the “multi” option must be set to true db.restaurants.update({ "address.zipcode": "10016", cuisine: "Other" }, { $set: { cuisine: "Category To Be Determined" }, $currentDate: { "lastModified": true } }, { multi: true})

2.9	To replace an entire document by another => pass an entirely new document as 2nd parameter in the update method.
	
	Important: after the update the document will only contain the fields of the provided document and the id.
	
	db.restaurants.update(
	{ "restaurant_id" : "41704620" },
	{
	"name" : "Vella 2",
	"address" : {
	"coord" : [-73.9557413, 40.7720266],
	"building" : "1480",
	"street" : "2 Avenue",
	"zipcode" : "10075"
	}
	}
	)
	
	You can check this via
	db.restaurants.find({"name" : "Vella 2"})

2.10	Remove documents via remove. **db.restaurants.remove({ "borough": "Manhattan" })** By default the remove function removes all documents matching to the condition. You can pass the justOne option to make sure that only 1 documents gets deleted: **db.restaurants.remove({ "borough": "Queens" }, { justOne: true })**
2.11	With an empty remove you can remove all documents of a collection: **db.restaurants.remove( )**
2.12	Via drop you can drop a collection: **db.restaurants.drop( )** If the collection to drop does not exist, the operation will return false.
3.	Aggregate queries
3.1	Load in the complete test dataset: Open a new vagrant ssh shell: - Open a command prompt - Navigate to the vagrant folder - Enter "**vagrant ssh**" - Go to the vagrant folder via **cd vagrant** - execute the following command: **mongoimport --db test --collection restaurants --drop --file primer-dataset.json**
3.2	Group by a field and calculate a count Via $group you can group by a specified key. In the $group stage, specify the group by key in the _id field. $group accesses fields by the field path, which is the field name prefixed by $ The $group stage can use accumulators to perform calculations on each group Over here we will use the $sum accumulator

	db.restaurants.aggregate(**[** **{ $group: { "_id": "$borough", "count": { $sum: 1 } } }** **]** **);** The results consists of the following documents: { "_id" : "Staten Island", "count" : 969 } { "_id" : "Brooklyn", "count" : 6086 } { "_id" : "Manhattan", "count" : 10259 } { "_id" : "Queens", "count" : 5656 } { "_id" : "Bronx", "count" : 2338 } { "_id" : "Missing", "count" : 51 }
3.3	Use the $match to filter documents. $match uses the query syntax. **db.restaurants.aggregate(** **[** **{ $match: { "borough": "Queens", "cuisine": "Brazilian" } },** **{ $group: { "_id": "$address.zipcode" , "count": { $sum: 1 } } }** **]** **);** The result will be { "_id" : "11368", "count" : 1 } { "_id" : "11106", "count" : 3 } { "_id" : "11377", "count" : 1 } { "_id" : "11103", "count" : 1 } { "_id" : "11101", "count" : 2 } Where _id contains the zipcode value - which was used to group by.
3.4	There exist some common aggregation methods. count: **db.restaurants.count()** will return the total number of restaurant documents in the collection count also works after a find(): **db.restaurants.find({ "cuisine": "Italian", "address.zipcode": "10075" }).count()** => 15

	distinct: The distinct() function allows you to retrieve the distinct values of a single field. db.restaurants.distinct("cuisine")
Indexes	

4.1	Create a single field index
	
	Create an ascending index on the cuisine field of the restaurants collection
	
	db.restaurants.createIndex({ "cuisine": 1 })
	
	This returns:
	{
	"createdCollectionAutomatically" : **false**,
	"numIndexesBefore" : 1,
	"numIndexesAfter" : 2,
	"ok" : 1
	}

4.2	Compound indexes are also possible **db.restaurants.createIndex({ "cuisine": 1, "address.zipcode": -1 })**
5. Map Reduce	
	MongoDB can use the map - reduce type of working to handle more complex tasks
5.1	Data setup: **db.sessions.save({ userid: "a", ts: ISODate('2011-11-03 14:17:00'), length: 95 });** **db.sessions.save({ userid: "b", ts: ISODate('2011-11-03 14:23:00'), length: 110 });** **db.sessions.save({ userid: "c", ts: ISODate('2011-11-03 15:02:00'), length: 120 });** **db.sessions.save({ userid: "d", ts: ISODate('2011-11-03 16:45:00'), length: 45 });** **db.sessions.save({ userid: "a", ts: ISODate('2011-11-04 11:05:00'), length: 105 });** **db.sessions.save({ userid: "b", ts: ISODate('2011-11-04 13:14:00'), length: 120 });**

	db.sessions.save({ userid: "c", ts: ISODate('2011-11-04 17:00:00'), length: 130 }); db.sessions.save({ userid: "d", ts: ISODate('2011-11-04 15:37:00'), length: 65 });

5.2	Define a map function to map the userid to an object that contains the fields userid, total_time, count and avg_time
	var mapFunction = function() {
	var key = this.userid;
	var value = {
	userid: this.userid,
	total_time: this.length,
	count: 1,
	avg_time: 0
	};
	
	emit(key, value);
	};

	Define the corresponding recude function with 2 arguments, keys and values to calculate the total time and the count.
	Key = the userid while values is an array with properties.
	
	var reduceFunction = function(key, values) {
	
	var reducedObject = {
	userid: key,
	total_time: 0,
	count:0,
	avg_time:0
	};
	
	values.forEach(function(value) {
	reducedObject.total_time += value.total_time;
	reducedObject.count += value.count;
	}
	);
	return reducedObject;
	};

	Define the finalize function with two arguments key and reducedValue. The function modifies the reducedValue document to add another field average and returns the modified document. var finalizeFunction = function (key, reducedValue) {

```
                if (reducedValue.count > 0)
                   reducedValue.avg_time = reducedValue.total_time / reducedValue.count;

                return reducedValue;
             };
```

Now perform the map reduce operation.

The results will be written out to the collection "session_stat"
If that collection already exists, it will be emptied and overwritten

```
db.sessions.mapReduce( mapFunction,
                reduceFunction,
                {
                  out: "session_stat",
                  finalize: finalizeFunction
                }
              )
```

Via **db.session_stat.find().pretty()** you can see its contents