Ministerul Educatiei al Republicii Moldova

Universitatea Tehnica a Moldovei

Facultatea Calcuatoare Informatica si Microelectronica

Departamentul Ingineria Software si Automatica

RAPORT

Lucrarea de laborator nr.2

Disciplina: Programarea in retea

Tema: HTTP Client with concurrency superpowers

A efectuat
st.gr. TI-153 **Popusoi Victor**

A controlat
asis., univ. **Gavrisco Alexandru**

Chisinau 2018

1 Scopul lucrarii de laborator

Study OSI model, HTTP and implement a client app which can do multiple HTTP requests concurrently.

2 Obiectivele lucrarii de laborator

- Read about OSI model (definition, basic info)
- Read about (de)serialization
- Concurrency (definition, primitives etc)

3 Efectuarea lucrarii de laborator

3.1 Sarcinile propuse pentru efectuare lucrarii de laborator

Link-ul la repozitoriu *here*
Report Generator

Theres a legacy system your new client want to extend (without access to its source code). The existing system allows to get a list of orders made by client's customers and a list of categories. The orders are quite simple - createdat, total, userid, categoryid. And a category isn't complicated as well - id, name, categoryid¡nullable¿. Category is a simple recursive data structure (what means that some categories can have as parent another category, and there can be "root" categories which doesn't have a parent).

Your client wants from you a tool which can generate a new type of report - total per categories (including data from child descending categories).

There is some info about the legacy system you managed to get from your client:

- Categories URL https://evil-legacy-service.herokuapp.com/api/v101/categories/
- Orders URL https://evil-legacy-service.herokuapp.com/api/v101/orders/
- The client said that he had found a mysterious "key" 55193451-1409-4729-9cd4-7c65d63b8e76 for the legacy system **Note:** The legacy system isn't documented properly (all you know about it - URL and that it must return CSV), so you need to read about HTTP and discover what additional info you may need to supply to get a response with requested data.

The application must offer next functionality:

- retrieve the list of orders (since it's a legacy system, it exports data in CSV format :( ) within a date interval
- retrieve the list of available categories (also CSV :( )
- parse and validate received data
- aggregate data
- display results to the user
- cache received data locally

There are some requirements, though:

- The application should have a friendly UI
- When the application is opened, it must load cached data if it's available
- All I/O operations mustn't block UI
- Requests must be performed concurrently whenever it's possible
- Aggregation of the data must performed concurrently
- Application must display results and cache them concurrently

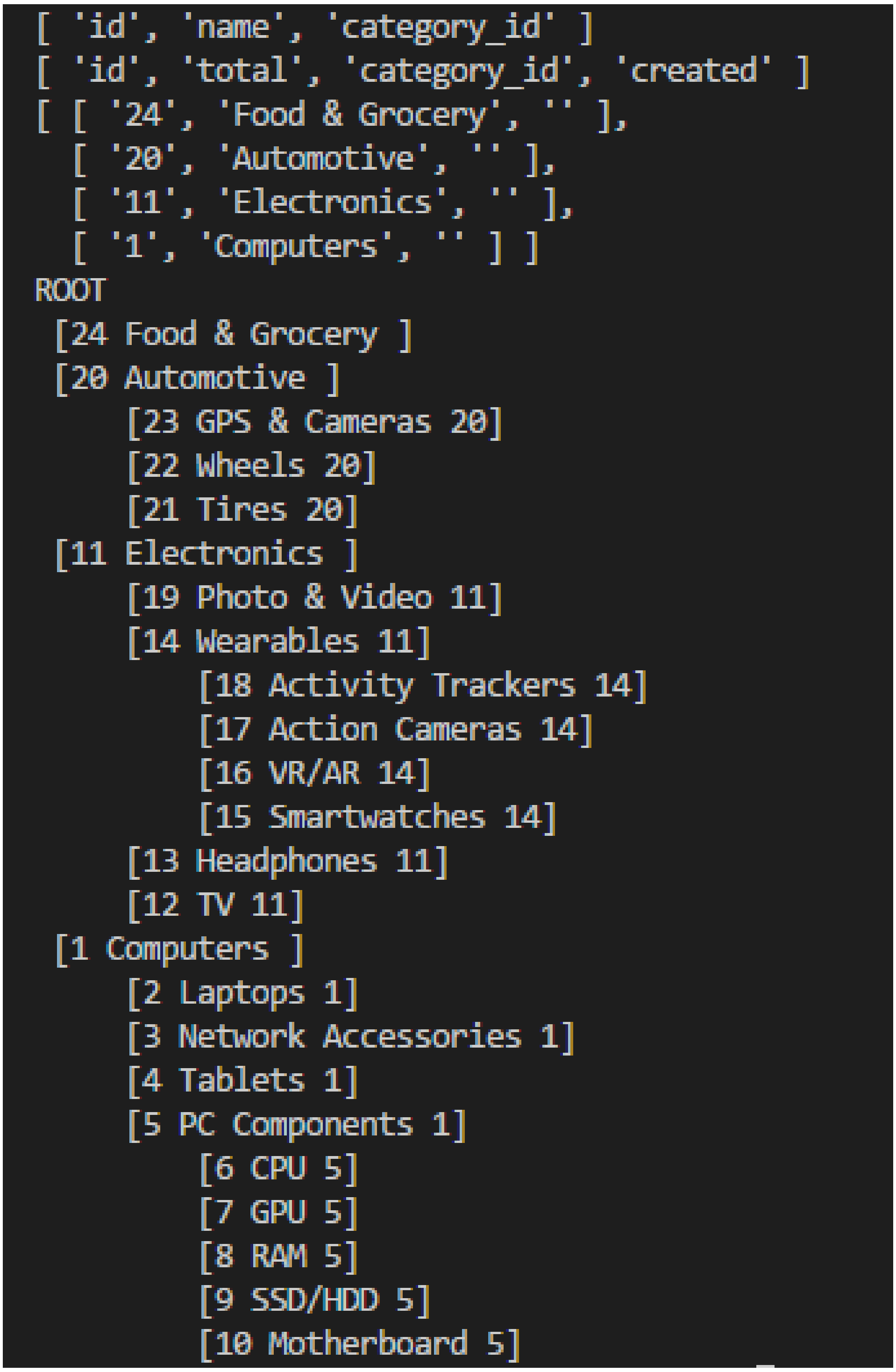
3.2 Realizarea lucrarii de laborator

Implementarea task-ului poate fi vizualizata in programul de mai jos

```
const {exec} = require('child_process')
var fs = require('fs')
var count = 0,
  nr = 0
var categories = [],
  orders = []
var done = function (next) {
  console.log('All done')
  next()
}
function fireFileRead () {
  nr++
  if (2 == nr)
    doneFileRead()
}
function fired (err, stdout, stderr) {
  console.log(stdout)
  count++
  if (2 == count)
    done(function () {
      fs.readFile('./orders.csv', function (e, data) {
        data = data.toString()
        var csv_data = data.split(/\r?\n|\r/)
        csv_data = csv_data.map(k => k.split(','))
        orders = csv_data
        fireFileRead()
      })
      fs.readFile('./categories.csv', function (e, data) {
        data = data.toString().trim()
        var csv_data = data.split(/\r?\n|\r/)
        csv_data = csv_data.map(k => k.split(','))
        categories = csv_data
        fireFileRead()
      })
    })
}
function cat () {this.orders = [];}
cat.prototype.toString = function catToString () {
  return '[' + this.categoryId + ' ' + this.name + ' ' + this.parentId + `]`
}
function doneFileRead () {
  var arr = [categories, orders]
  var filterResult = categories.filter(n => !n[2])
  var k = new Tree()
  var x = filterResult
  k.add('ROOT')
  var groupe = []
  for (var m in x) {
    var nx = new cat()
    nx.categoryId = x[m][0]
    nx.name = x[m][1]
    nx.parentId = x[m][2]
    nx.level = 1
    groupe[x[m][0]] = nx
    k.add(nx, 'ROOT')
  }
  categories.forEach(c => {
    var p_id = c[2]
    if (groupe[p_id]) {
      var nx = new cat()
      nx.categoryId = c[0]
      nx.name = c[1]
      nx.parentId = c[2]
      nx.level = groupe[p_id].level + 4
      groupe[c[0]] = nx
      k.add(nx, groupe[p_id])
    }
  })
  categories.filter(g => !groupe[g[0]]).forEach(c => {
    var p_id = c[2]
    if (groupe[p_id]) {
      var nx = new cat()
      nx.categoryId = c[0]
      nx.name = c[1]
      nx.parentId = c[2]
      nx.level = groupe[p_id].level + 4
      groupe[c[0]] = nx
      k.add(nx, groupe[p_id])
    }
  })
  categories.filter(g => !groupe[g[0]]).forEach(c => {
    var p_id = c[2]
    if (groupe[p_id]) {
      var nx = new cat()
      nx.categoryId = c[0]
      nx.name = c[1]
      nx.parentId = c[2]
      nx.level = groupe[p_id].level + 4
      groupe[c[0]] = nx
      k.add(nx, groupe[p_id])
    }
  })
  console.log(categories[0])
  console.log(orders[0])
  orders.forEach(n => {
    if (!groupe[n[2]]) {
      return
    }
    groupe[n[2]].orders.push(n)
  })
  console.log(x)
  // k.printByLevel()
  k.traverseDFS(function (r) {
    console.log('  '.repeat(r.data.level) + r.data.toString())
  })
  return
  for (var c in categories) {
    for (var n in x) {
      if (categories[c][2] == n[1]) {
        filterResult.push(categories[c])
```

```
        k.add(categories)
      }
    }
  }
  console.log(k)
}
exec('node reqOrders.js', fired)
exec('node reqCategories.js', fired)
function Node (data) {
  this.data = data
  this.children = []
}
function Tree () {
  this.root = null
}
Tree.prototype.add = function (data, toNodeData) {
  var node = new Node(data)
  var parent = toNodeData
    ? this.findBFS(toNodeData)
    : null
  if (parent) {
    parent
      .children
      .push(node)
  } else {
    if (!this.root) {
      this.root = node
    } else {
      return 'Root node is already assigned'
    }
  }
}
Tree.prototype.findBFS = function (data) {
  var queue = [this.root]
  while (queue.length) {
    var node = queue.shift()
    if (node.data === data) {
      return node
    }
    for (var i = 0; i < node.children.length; i++) {
      queue.push(node.children[i])
    }
  }
  return null
}
Tree.prototype._preOrder = function (node, fn) {
  if (node) {
    if (fn) {
      fn(node)
    }
    for (var i = 0; i < node.children.length; i++) {
      this._preOrder(node.children[i], fn)
    }
  }
}
Tree.prototype._postOrder = function (node, fn) {
  if (node) {
    for (var i = 0; i < node.children.length; i++) {
      this._postOrder(node.children[i], fn)
    }
    if (fn) {
      fn(node)
    }
  }
}
Tree.prototype.traverseDFS = function (fn, method) {
  var current = this.root
  if (method) {
    this['_' + method](current, fn)
  } else {
    this._preOrder(current, fn)
  }
}
```

3.3 Imagini



```
[ 'id', 'name', 'category_id' ]
[ 'id', 'total', 'category_id', 'created' ]
[ [ '24', 'Food & Grocery', '' ],
  [ '20', 'Automotive', '' ],
  [ '11', 'Electronics', '' ],
  [ '1', 'Computers', '' ] ]
ROOT
 [24 Food & Grocery ]
 [20 Automotive ]
     [23 GPS & Cameras 20]
     [22 Wheels 20]
     [21 Tires 20]
 [11 Electronics ]
     [19 Photo & Video 11]
     [14 Wearables 11]
         [18 Activity Trackers 14]
         [17 Action Cameras 14]
         [16 VR/AR 14]
         [15 Smartwatches 14]
     [13 Headphones 11]
     [12 TV 11]
 [1 Computers ]
     [2 Laptops 1]
     [3 Network Accessories 1]
     [4 Tablets 1]
     [5 PC Components 1]
         [6 CPU 5]
         [7 GPU 5]
         [8 RAM 5]
         [9 SSD/HDD 5]
         [10 Motherboard 5]
```

Figure 3.1 – Final result

Concluzie

HTTP este metoda cea mai des utilizată pentru accesarea informațiilor în Internet care sunt păstrate pe servere. Conform sarcinii lucrarii de laborator a fost implementat un program care permite obținerea unei liste de comenzi făcute de clienți și o listă de categorii. Pe parcursul efectuarii lucrarii de laborator am descoperit conceptele de bază referitoare la protocolul HTTP precum si programarea concurentă. Cu ajutorul metodelor GET și POST am primit răspunsuri de la server și am procesat rezultatele.

Rezultatele obtinute au fost agregate și afișate. Programarea concurenta are un avantaj care consta in utilizarea eficientă a resurselor si utilizarea eficienta a timpului de calcul

4 Bibliografie

1 https://stackoverflow.com/questions/247483/http-get-request-in-javascript

2 https://developer.mozilla.org/ro/docs/Web/JavaScript/Reference/Global_Objects/Object/toString

3 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/repeat

4 https://stackoverflow.com/questions/2734025/is-javascript-guaranteed-to-be-single-threaded

5 https://medium.com/@jotadeveloper/abstract-syntax-trees-on-javascript-534e33361fc7

6 https://code.tutsplus.com/articles/data-structures-with-javascript-tree–cms-23393

7 https://stackoverflow.com/questions/18017869/build-tree-array-from-flat-array-in-javascript

8 https://www.quora.com/How-exactly-do-you-build-a-tree-data-structure-in-JavaScript

9 https://codereview.stackexchange.com/questions/47932/recursion-vs-iteration-of-tree-structure