Project Task (C++): Efficient Key-Value-Store

Challenge

The challenge is to create an efficient, performance optimized key-value-store by making use of modern C++. With this application, users should be able to add/update values by key, retrieve values by key and delete values by key.

For demonstration purposes the application should be network-accessible. Please also provide a client for which you can use any library or language.

Requirements

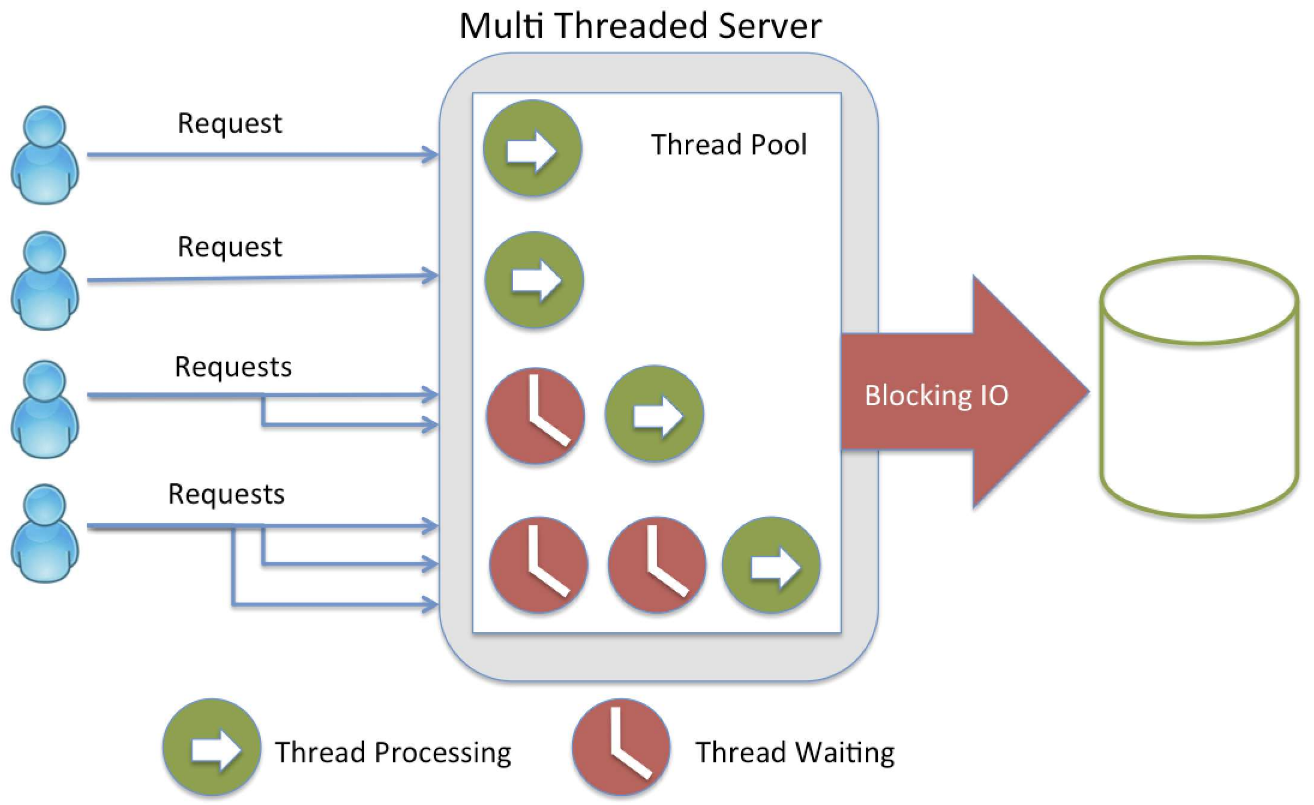
The following functionalities should be implemented:

Interface:

- Keys are strings, values are strings as well. Both have variable length
- A “PUT” call which adds a pair to the store
- A “DELETE” call which deletes a pair from the store by key
- A “GET” call which retrieves a pair from the store by key

Technology overview and reasoning behind choosing those technologies

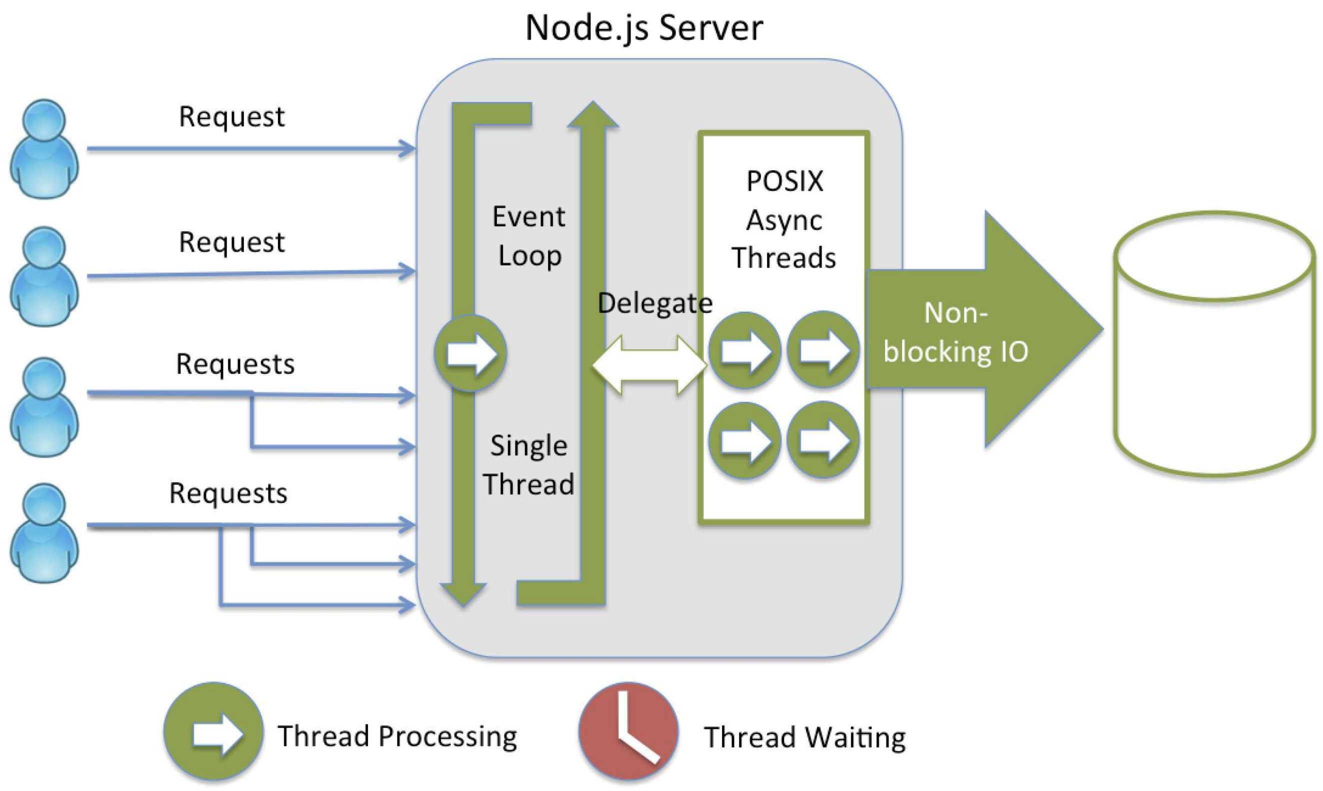
The classic approach



Picture source: https://strongloop.com/strongblog/node-js-is-faster-than-java/

The Node.JS approach

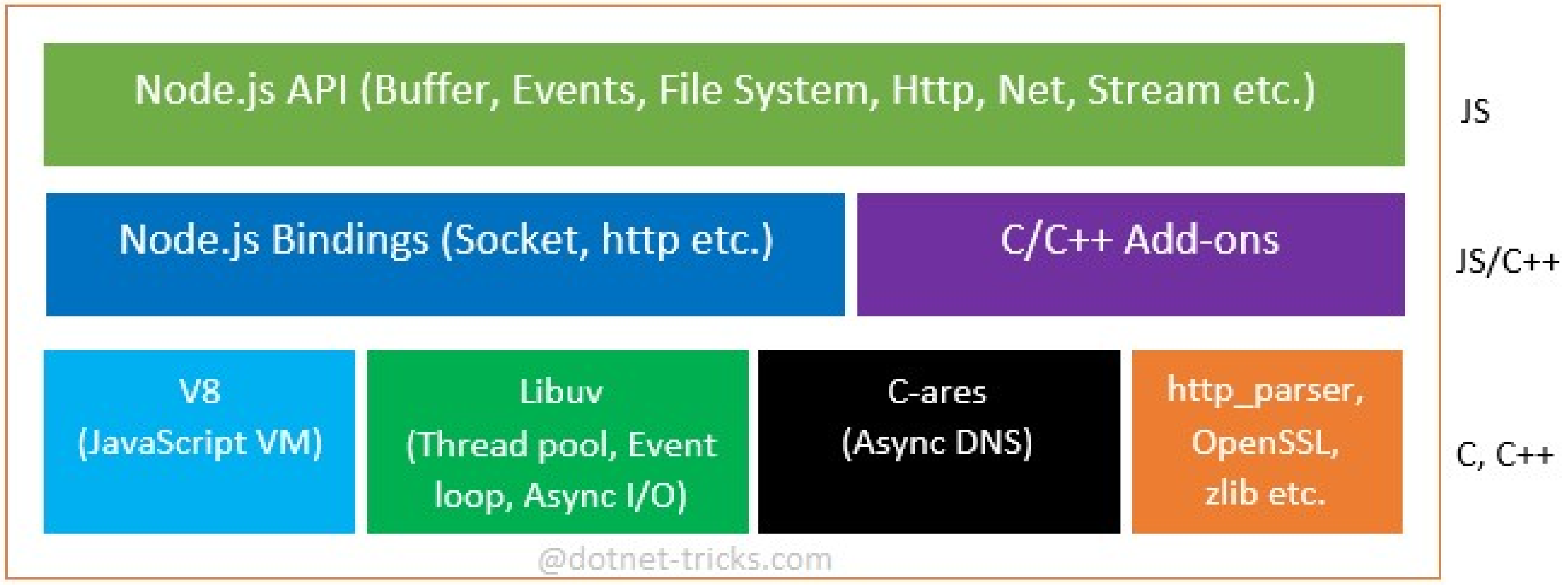
"Node.js is a JavaScript runtime built on Chrome's V8 JavaScript runtime."



Node.js server lifecycle.

Picture source: https://strongloop.com/strongblog/node-js-is-faster-than-java/

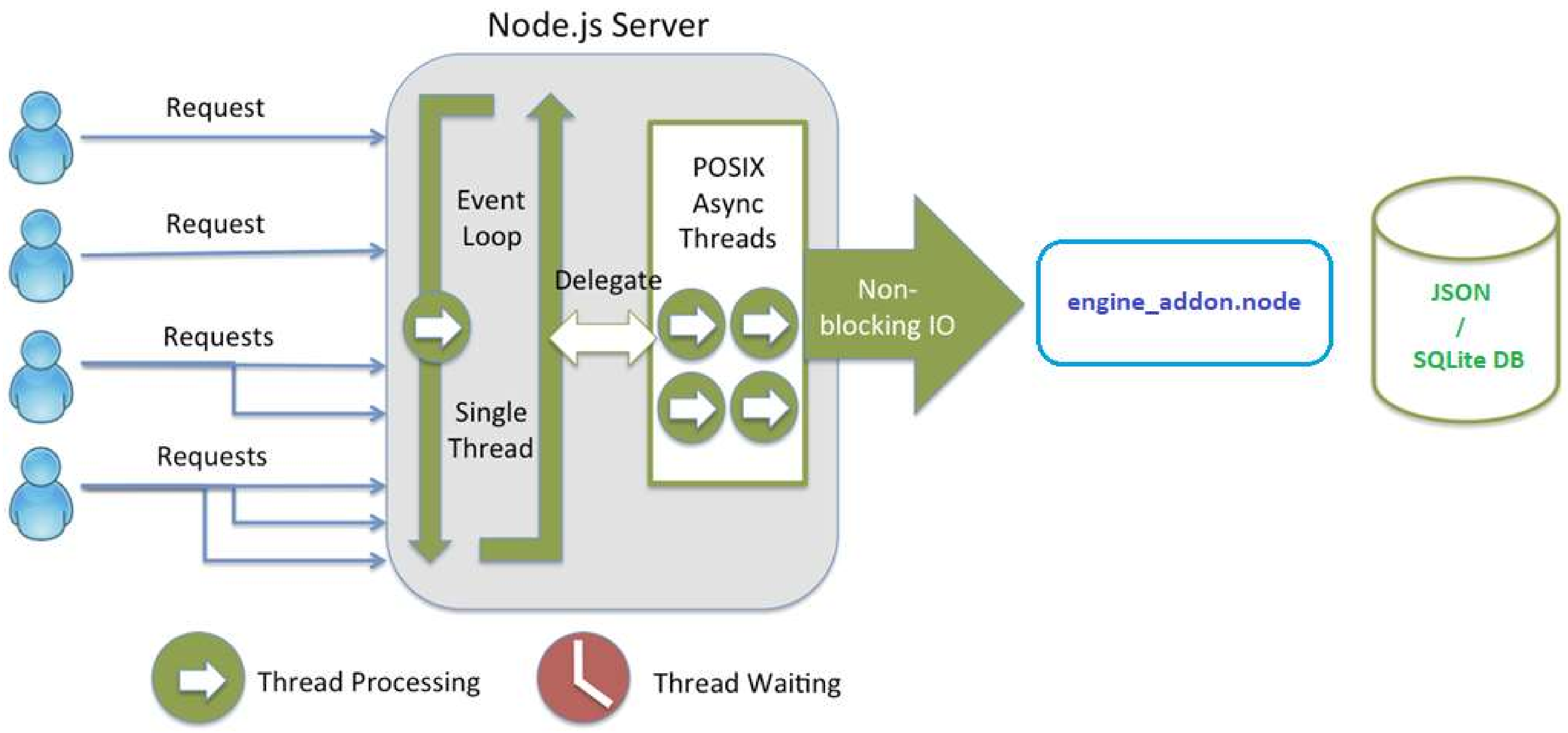
The Node.JS architecture


Node.js API (Buffer, Events, File System, Http, Net, Stream etc.)
JS
Node.js Bindings (Socket, http etc.)
C/C++ Add-ons
JS/C++
V8 (JavaScript VM)
Libuv (Thread pool, Event loop, Async I/O)
C-ares (Async DNS)
http_parser, OpenSSL, zlib etc.
C, C++
@dotnet-tricks.com

Node.js 4.2.4 Architecture

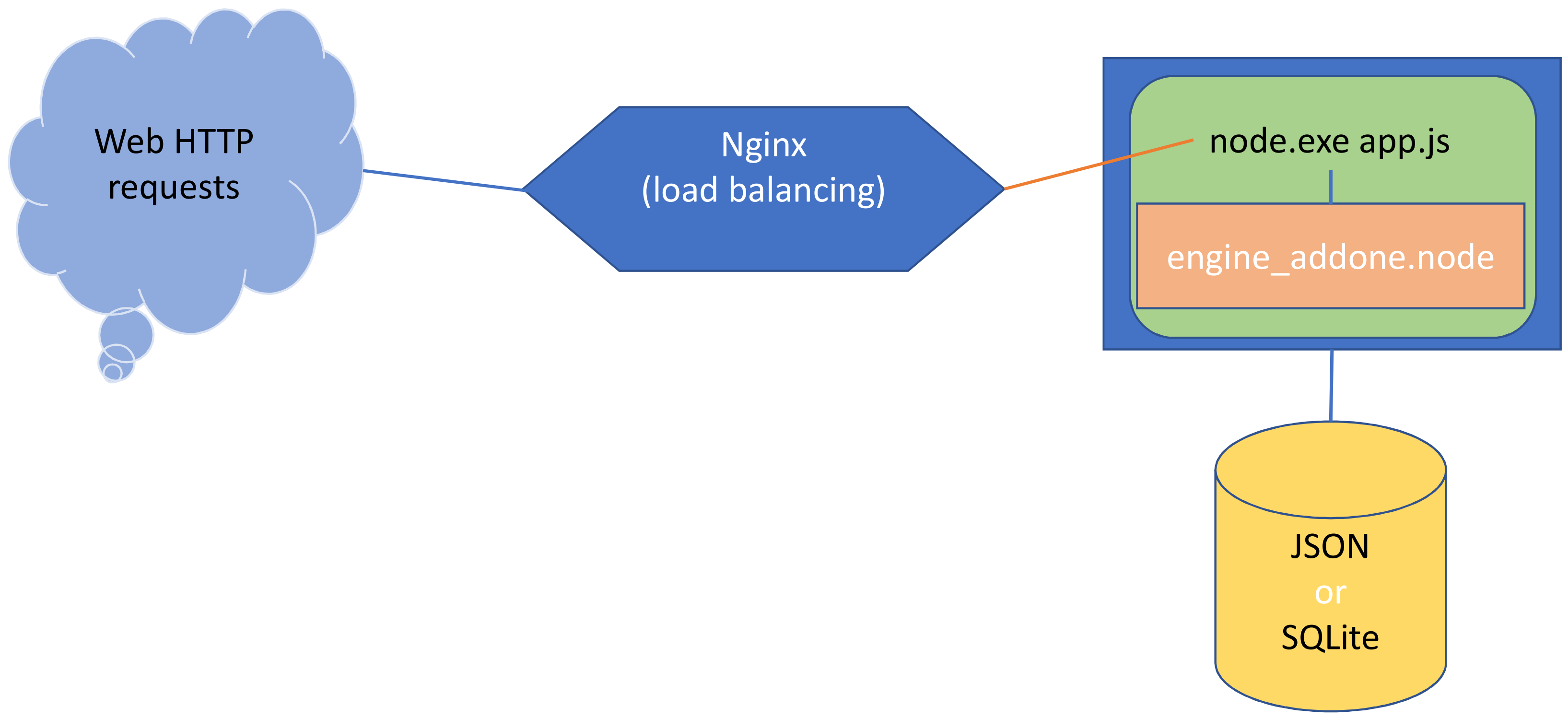
Picture source: https://www.dotnettricks.com/learn/nodejs/exploring-nodejs-architecture

The Node.JS C++ addin – engine_addon.node



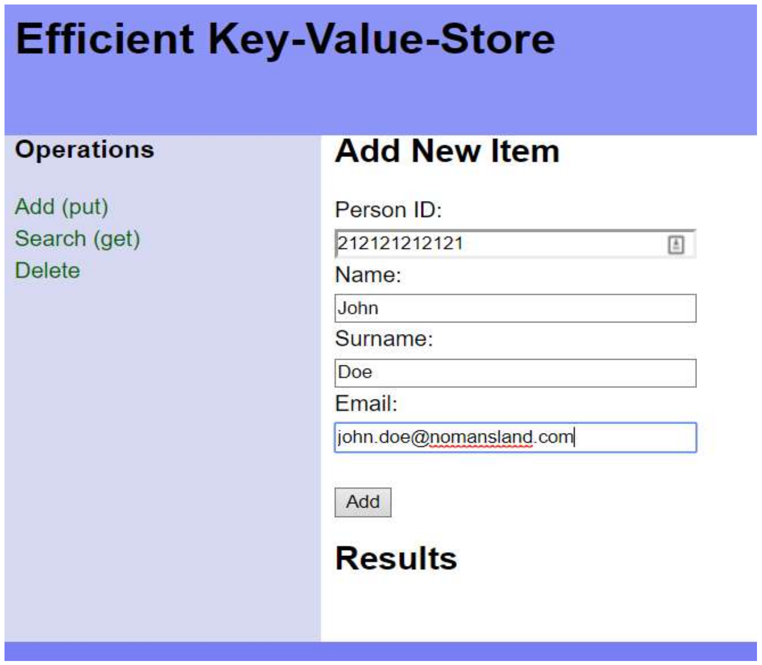
Picture source: https://strongloop.com/strongblog/node-js-is-faster-than-java/

Real scenario C++ addin – engine_addon.node



Picture source: https://strongloop.com/strongblog/node-js-is-faster-than-java/

The Challenge User Interface



Efficient Key-Value-Store

Operations

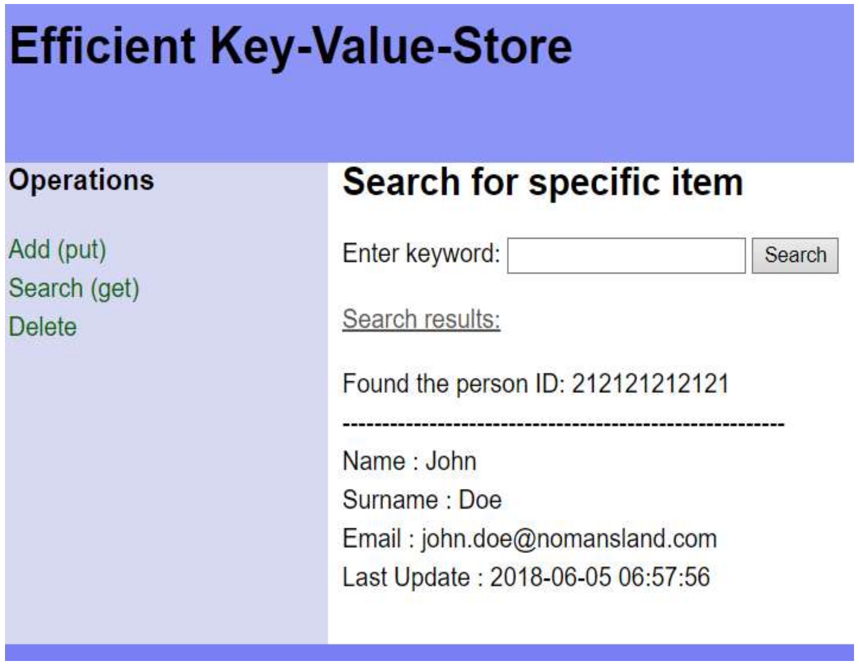
Add (put)
Search (get)
Delete

Add New Item

Person ID:
212121212121
Name:
John
Surname:
Doe
Email:
john.doe@nomansland.com

Add

Results



Efficient Key-Value-Store

Operations

Add (put)
Search (get)
Delete

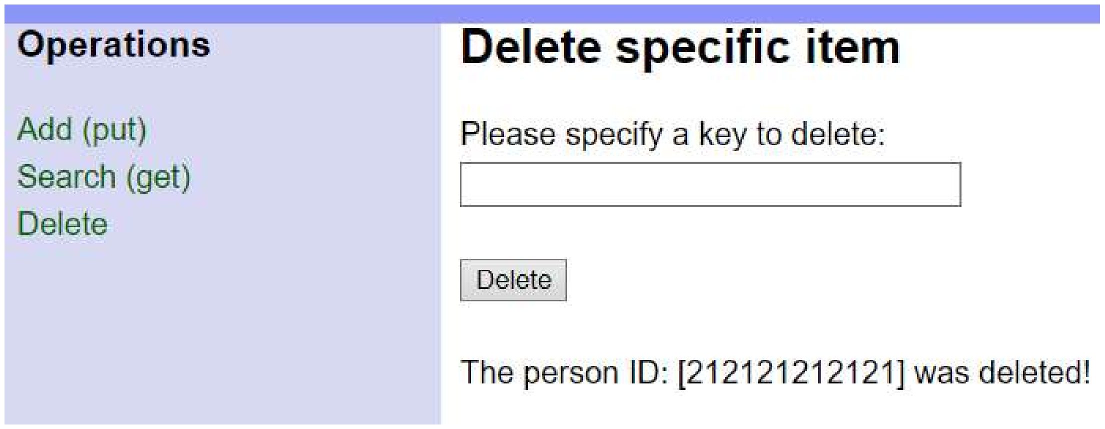
Search for specific item

Enter keyword: Search

Search results:

Found the person ID: 212121212121

Name : John
Surname : Doe
Email : john.doe@nomansland.com
Last Update : 2018-06-05 06:57:56



Operations

Add (put)
Search (get)
Delete

Delete specific item

Please specify a key to delete:

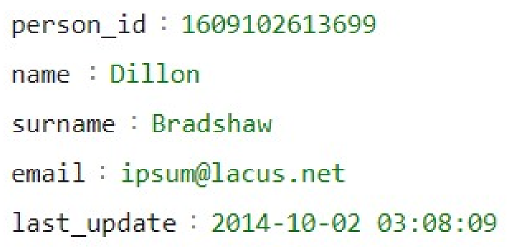
Delete

The person ID: [212121212121] was deleted!

http://silviuardelean.ro

The effective C++ approach

To exemplify I started with the idea of a Person data including: person id, name, surname, email.



```
person_id : 1609102613699
name : Dillon
surname : Bradshaw
email : ipsum@lacus.net
last_update : 2014-10-02 03:08:09
```

The **last_update** attribute is internal established and it’s used to sort the data on data manager but also to store the latest items within the cache.

All these properties are stored within the instances of the Person class.

For the first iteration **engine_addon.node** exposes 4 methods:

- **initEngine** – initialize the internal data structures
- **addItem** – add new item to cache and data manager, later to the JSON file (SQLite future version).
- **deleteItem** – delete an item from data manager (including the JSON) and from the cache if exists
- **searchItem** – search for an existing item based on the person ID

The wrapper data class

```
class Person
{
public:

  Person(const std::string& id, const std::string& name,
             const std::string& surname, const std::string& email, const std::string& lastupdate);
  virtual ~Person();

  Person(const Person& rhs);
  Person& operator = (const Person& rhs);

  Person(Person&& rhs);
  Person& operator = (Person&& rhs);

  bool operator > (const Person& rhs) const;
  bool operator < (const Person& rhs) const;

  template <typename Writer>
  void Serialize(Writer& writer) const;

  std::string getPersonalID() const { return id_; }
  std::string getName() const { return name_; }
  std::string getSurname() const { return surname_; }
  std::string getEmail() const { return email_; }
  std::string getLastUpdate() const { return lastupdate_; }

protected:
  std::string id_;
  std::string name_;
  std::string surname_;
  std::string email_;
  std::string lastupdate_;
};
```

For simplicity, this class wraps over the JSON data.

Nice to have:
This class should became abstract and extended by more specific types (students, workers, etc)