CPPServer

Dev-platform for high performance JSON microservices and modern Apps with C++

Contact

https://www.cppserver.com

Phone/WsApp:
+58 (424) 2686639

Email:
cppserver@martincordova.com

Social

Facebook: @cppserver

#cppserver
#simplicityworks

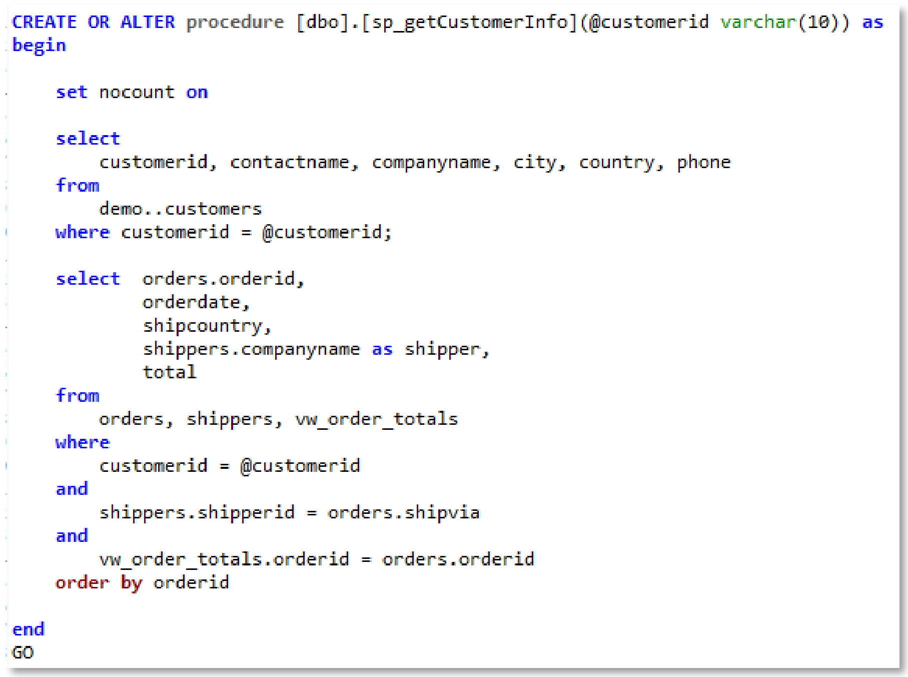
Created by:
Martín Córdova y Asociados, C.A.
Caracas – Venezuela

Anatomy of a C++ microservice

Our CPPServer platform allows fast development of microservices that return JSON datasets from SQL databases without writing any code, just by declaring tasks using a simple JSON configuration file. In this particular case we are going to export a multi-array response, given a customer ID the microservice will return a customer dataset and the customer's orders dataset, all in one roundtrip.

The SQL backend

We will use SQLServer as the database for this example, and a stored procedure to return multiple resultsets in one roundtrip to the database server, looking for efficiency, pre-compiled optimized queries and conscious usage of network resources by returning all data in one shot. SQLServer's TransactSQL makes this task very easy to accomplish.

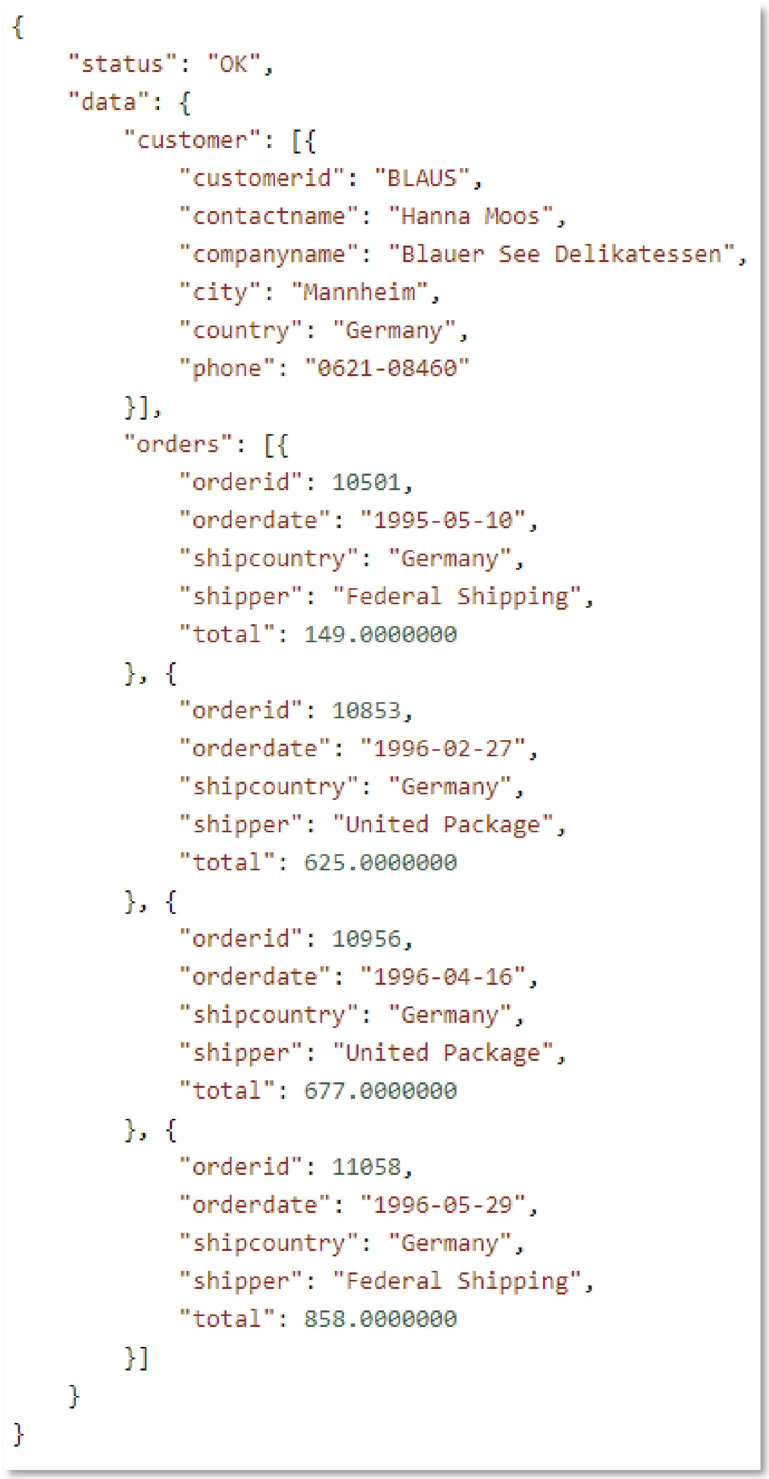


```sql
CREATE OR ALTER procedure [dbo].[sp_getCustomerInfo](@customerid varchar(10)) as
begin

    set nocount on

    select
        customerid, contactname, companyname, city, country, phone
    from
        demo..customers
    where customerid = @customerid;

    select  orders.orderid,
            orderdate,
            shipcountry,
            shippers.companyname as shipper,
            total
    from
        orders, shippers, vw_order_totals
    where
        customerid = @customerid
    and
        shippers.shipperid = orders.shipvia
    and
        vw_order_totals.orderid = orders.orderid
    order by orderid

end
GO
```

This stored procedure will return two resultsets given a customer ID. Our database backend is ready and follows best practices, we are only returning the columns required to reduce the size of the data packets to be transferred over the network. Tables are properly indexed and statistics are up to date. So far so good.

JSON output

According to CPPServer JSON specification, if there is no error processing the request, the output will look like this:



```
{
    "status": "OK",
    "data": {
        "customer": [{
            "customerid": "BLAUS",
            "contactname": "Hanna Moos",
            "companyname": "Blauer See Delikatessen",
            "city": "Mannheim",
            "country": "Germany",
            "phone": "0621-08460"
        }],
        "orders": [{
            "orderid": 10501,
            "orderdate": "1995-05-10",
            "shipcountry": "Germany",
            "shipper": "Federal Shipping",
            "total": 149.0000000
        }, {
            "orderid": 10853,
            "orderdate": "1996-02-27",
            "shipcountry": "Germany",
            "shipper": "United Package",
            "total": 625.0000000
        }, {
            "orderid": 10956,
            "orderdate": "1996-04-16",
            "shipcountry": "Germany",
            "shipper": "United Package",
            "total": 677.0000000
        }, {
            "orderid": 11058,
            "orderdate": "1996-05-29",
            "shipcountry": "Germany",
            "shipper": "Federal Shipping",
            "total": 858.0000000
        }]
    }
}
```

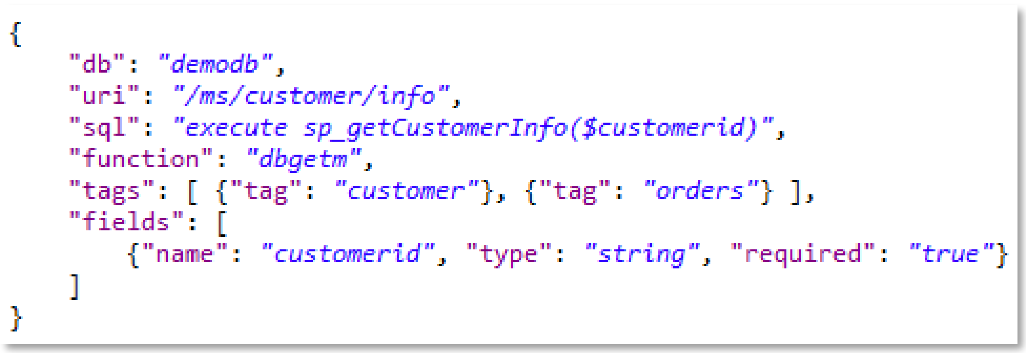
The complete specification of the JSON output supported by CPPServer:

https://cppserver.com/docs/json_response_spec.pdf

The *modern* C++ Backend

Are we going to write C++ code to create the microservice? No, we don’t need to write any code, we’ll just declare our intention in a JSON configuration file, CPPServer program loads this configuration and executes its generic functions that interact with the database server and produce JSON output at very high speed (if the DBMS allows it).

The JSON configuration for this particular example looks like this:

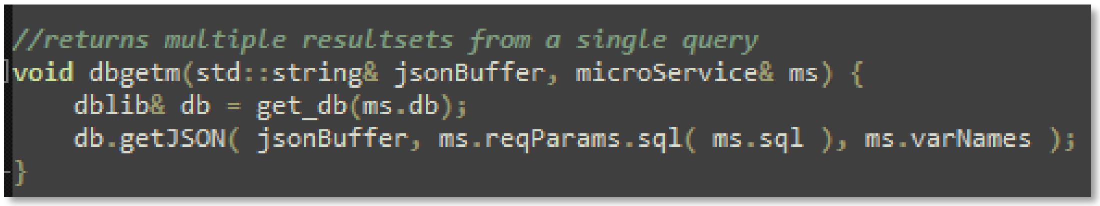


```
{
    "db": "demodb",
    "uri": "/ms/customer/info",
    "sql": "execute sp_getCustomerInfo($customerid)",
    "function": "dbgetm",
    "tags": [ {"tag": "customer"}, {"tag": "orders"} ],
    "fields": [
        {"name": "customerid", "type": "string", "required": "true"}
    ]
}
```

The configuration contains the *URI* for this service, the *SQL* to execute the stored procedure shown in page 1, the *$customerid* is a placeholder that will be replaced by the corresponding input parameter, CPPServer ensures that there is no SQL injection attack in the input parameters, the *tags* array indicates the name of each JSON array (SQL resultset) as shown in page 2, and the *fields* array indicates the input fields required by this microservice in the URL, via GET or POST, including validation information such as data type and if it’s required or not. With this configuration CPPServer will do all the work for you. The attribute *function* is the name to identify the internal CPPServer generic function (the actual microservice) that will be used for this case, “dbgetm” stands for “get multiple resultsets”, there is also “dbget” for single-resultset cases. Internally, CPPServer will have a pointer to the function for a fast thread-safe invocation.

This JSON definition is loaded and parsed once only and transformed into a memory data structure for very fast lookups.

The actual C++ code, which you don't have to write, is actually very easy to write, thanks to the efficient high-level abstractions of C++ and CPPServer:


```
//returns multiple resultsets from a single query
void dbgetm(std::string& jsonBuffer, microService& ms) {
    dblib& db = get_db(ms.db);
    db.getJSON( jsonBuffer, ms.reqParams.sql( ms.sql ), ms.varNames );
}
```

This is the native C++ code, 2 lines of code, the dblib type is one of the main CPPServer abstractions, encapsulates the database server native API in a powerful and pragmatic way. Each HTTP request runs on a separate thread, by using C++ *thread_local* variables the threads share almost nothing to minimize concurrency bottlenecks.

The function will return a JSON response according to the specification, it can be OK, INVALID or ERROR response, the microservice does not care about it, detailed error description will be printed to STDERR in case of error.

The C++ code was written to minimize allocations on each request by pre-allocating buffers for json output and database connections only once when the server starts.

It's not hard to write a C++ microservice using CPPServer facilities, but for most of the common tasks this is not required, you can achieve it using the declarative way and enjoy the benefits of highly-optimized native machine code instead of using interpreted languages for your server-side applications. Queries, data modification (insert/update/delete) and blob upload/download, are all covered by pre-built microservices.

Testing the microservice

It's just a matter of sending a GET request using a browser or a command line utility such as CURL (available on Linux and Windows):

https://cppserver.com/ms/customer/info?customerid=BERGS

You can test other customer IDs like: ALFKI, ERNSH, BOTTM… also invalid input or no input at all. There will be a JSON response for every case.

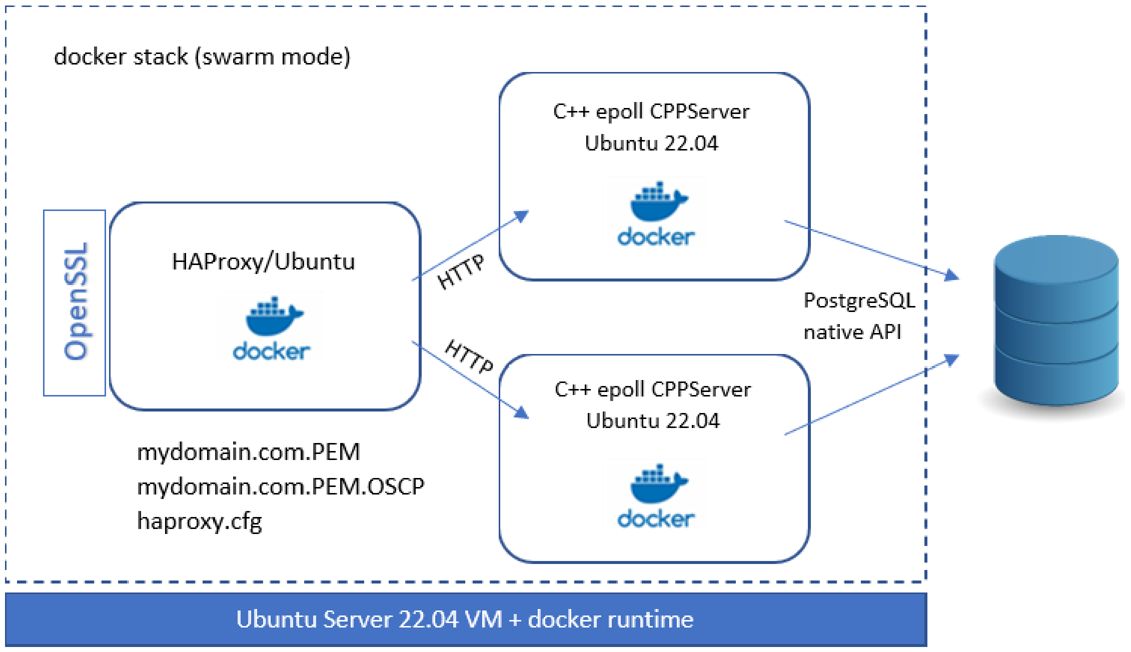
The execution of the microservice is subject to security restrictions, like an existing security session started by a login (LDAP/custom DB security model) and specific roles constraints, the restrictions can be disabled for development and demos, like the example above. If you have a database for managing logins, we can build an efficient CPPServer login adapter using your DBMS native API, a generic LDAP adapter that uses the standard LDAP native API is available too, works with OpenLDAP and ActiveDirectory.

Dockerized from birth

CPPServer was designed and built for containers, by default it runs on Docker, in swarm mode (stack deploy) behind HAProxy load balancer, it does use Docker secrets to read database connection information and a single configuration file that can be read from the host file system or from a docker configuration. There is separation of concerns, the docker Admin creates and manages the secrets without our intervention.

HTTP logs are printed to STDOUT, diagnostics and error logs are printed to STDERR, just as Docker expects, making it easy to manage the logs using docker facilities and the preferred customer configurations for this purpose, like log driver, rotation, size of files, etc.

We provide all the necessary tech-support to configure the docker environment in very short time with optimized settings, including the load balancer and the backend servers. The whole stack can be executed in a single Linux virtual machine, or spread to several VMs, depending on your workload. On the cloud or on-premises.



Native Code

CPPServer is written in C++, compiled to platform-specific native code using GNU G++ v12.2, its executable weights about 230K. It was designed specifically for Linux using the kernel's epoll non-blocking I/O facilities and one thread per core, saving system resources and achieving fast response time and scalability.

It was designed to be executed in cluster mode behind a load balancer inside a container, it's stateless, no need for session affinity, which alleviates the task of the load balancer. The HTTPS/TLS protocol is managed by the load balancer, HAProxy LB uses OpenSSL for this purpose, it's an industrial-grade open-source product which excels at this task. The connection to the backends is plain HTTP over a closed docker network to avoid overhead, the backend servers are not visible outside this network, only the load balancer can reach them.

The C++ promise and our goal

Direct mapping to hardware, efficient high-level abstractions, type-safety and resource management, we built over these fundamental benefits of using *Modern* C++ and achieved our goal of having the productivity and ease of use of platforms like NodeJS with the raw power of C++ and the Linux Kernel.

We offer free trials of CPPServer technology for your company, we will use your database and LDAP server in the way you decide, so you can test the performance of this platform on your own terms, no compromises, no strings attached.