# CIT Assignment 3 – Network

This assignment concerns the development of a network service using the CJTP – CIT JSON Transport Protocol. The protocol is defined below. The task is to create a network service that provides the functionality defined by the CJTP. There are some similarities between the service to implement and a web server providing a web service; the primary difference is the protocol used to define requests and responses.

### How and when to hand in

Submit a 1-2 page document that includes the following information:

- List the members of your group.
- Provide the URL to a GitHub repository where the source code can be found.
- Include a table or a screenshot from your testing environment showing the status of running all tests in the test suite attached to this assignment.
- Please upload the document to Moodle under "Assignment 3" no later than October 16 at 23:55.

### Important

Submit one submission from your group, preferably from one of the members, but ensure that you include ALL NAMES of participants in your group at the top of the file.

## CJTP – CIT JSON Transport Protocol

Below, you find the description of the protocol to be implement.

Elements marked with an asterisk (*) is mandatory**.**

**Request Format:**

```
{
          method: <METHOD*>,
          path: <PATH*>,
          date: <DATE*>,
          body: <BODY>
}
```

Note: The `path` is mandatory for all methods except the “echo” method.

**Response Format:**

```
{
        status: <STATUS*>,
        body: <BODY*>
}
```

**<METHOD>**
can be one of the following: “create”, “read”, “update”, “delete”, “echo”

**<PATH>**
is the path to the resource in the format /foo/resource, i.e., standard path where elements are separated by slashes

**<DATE>***
in Unix time (i.e., number of seconds that have elapsed since 1970-01- 01T00:00:00Z[1])

**<BODY>**
is the payload of the message if it contains data. If available, the payload must be in JSON format.

**<STATUS>**
can be one of the following status codes followed by their reason phrase:

| Status Code | Reason phrase |
|---|---|
| 1 | Ok |
| 2 | Created |
| 3 | Updated |
| 4 | Bad Request |
| 5 | Not found |
| 6 | Error |

The level of description is not specified, except for bad requests with missing or erroneous elements which should be reported in the form

```
<STATUS CODE> reason
```

where reason has the form

[1] You can get the current unixtime with this statement: DateTimeOffset.Now.ToUnixTimeSeconds()

missing <element>

or

illegal <element>

where `<element>` is the name of the protocol element (e.g., method, date, path).
If more than one problem is found, return the status code followed by a list of reasons, e.g.,

```
“4 missing date, missing body, illegal method”
```

The order of reasons is not important.

## Question a)

Implement a web service by use of the `TcpListener`[2]/`TcpClient`[3] classes from the .NET framework. The service must adhere to the client/server model[4] and the request-response design pattern[5] Your service must listen on port 5000[6] and the implementation could benefit from being multithreaded: in other words, every connection to your service should ideally be handled by a new thread, although this is not a strict required.

The common pattern is for clients connect to your service, send a request, and have your service handles that request and send a response back. Nevertheless, you must be prepared for connections that do not send a request, and should simply ignore such connections.

Your service must implement the CJTP (CIT JSON Transport Protocol) and be capable of verifying all the constraints given in the description, concerning requirements on structure and content. This means that you must perform verification on the input requests from the clients and respond with error messages if the request is not valid.

The structural constrains pertain to the correct usage of the fields in requests and responses. For example:

```
{
    method: “update”,
    path: “/api/products/1”,
    date: 1507318869
}
```

This is an example of a violation of the constraint that specify that the `update` method requires a `body`.

The content constraints are related to the format of values. For instance:

[2] https://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener(v=vs.110).aspx
[3] https://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient(v=vs.110).aspx
[4] https://en.wikipedia.org/wiki/Client%E2%80%93server_model
[5] http://www.servicedesignpatterns.com/clientserviceinteractions/requestresponse
[6] Very import, since the test suite expect a service on this port

```
{
    method: “update”,
    path: “/api/categories/1”,
    date: “06-10-2017 19:41:09”,
    body: “{cid: 1, name: "NewName"}”
}
```

Here, the structure is correct, but the content of the date is not in the Unix time format.

The test suite will tell you if you missed some constraints :-)

## Question b)

Provide an API to the following data model:

```
category(cid, name)
```

And the following data:

| category | |
|---|---|
| **cid** | **name** |
| 1 | Beverages |
| 2 | Condiments |
| 3 | Confections |

You do not need to implement any real database functionality. Just keep the data in memory. Changes to the data on create, update and delete must be reflected but not persisted, meaning they are not stored and will be lost on restart.

Note: The test suite expects this data to be available on your CJTP service. It will query exactly this data and fail if not found or different.

The API must provide the following path: **`/api/categories`**

Just like the requests and responses, all data transported between client and server must be in JSON format, except for the payload (body) on the `echo` method, which is in plain text format. To send, for instance, the category with id 1 over the protocol, you must transform it into JSON

```
{cid: 1, name: "Beverages"}
```

Here are some examples on how to use the service:

| Method | Path | Example input | Status code | Example output |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| read | /api/categories/1 | | 1 Ok | {cid: 1, name: “Beverages”} |
| read | /api/categories | | 1 Ok | [{cid: 1, name: “Beverages”}, {cid: 2, name: “Condiments”} {cid: 3, name: “Confections”}] |
| update | /api/categories/3 | { cid: 3, name: “Test”} | 3 Updated | |
| update | /api/categories | { cid: 3, name: “Test”} | 4 Bad Request | |
| create | /api/categories | { name: “Seafood”} | 2 Created | {cid: 4, name: “Seafood”} |
| delete | /api/categories/3 | | 1 Ok | |
| delete | /api/categories/123 | | 5 Not Found | |
| read | /api/categories/123 | | 5 Not Found | |

**read**
The list of all elements can be retrieved using the path to the resource, here **`/categories`**. Individual elements from a resource can be retrieved by adding the **`“/<id>”`** to the resource path (the id of the element to be retrieved without the <>), e.g., **`/categories/2`**. Requesting individual elements must return status “5 Not found” if the requested element is not in the database, otherwise the status is “1 Ok”.

**create**
New elements can be added by use of the path to the resource and with the new element in the body of the request. Using path and an id with the create method is invalid and should return “4 Bad request”. On successful creation return the “2 Created” status plus the newly created element in the response body.

**update**
All elements can be updated by use of the path extended with the id and the updated element in the body. Updates without an id in the path is not allowed and should return “4 Bad request”. On successful update return the “3 Updated” status.

**delete**
All elements can be deleted by use of the path extended with the id. If the element is not in the database “5 Not found” should be returned otherwise “1 Ok”.

**echo**
This method does not take any path, it is just ignored if provided, and will simply return the body of the request as the body of the response with status “1 Ok”.

# Appendix

## Converting to and from JSON

If you use .NET Core 3.0(or later) JSON support is included as part of the framework[7]. Include the necessary `using` statements for it.

```
using System.Text.Json;
```

conversion can be done like this:

```
var category = new Category();
// from objects to JSON
var catgoryAsJson = JsonSerializer.Serialize<Category>(category);
// from JSON text to object
var categoryFromJson = JsonSerializer.Deserialize<Category>(catgoryAsJson);
```

Note: That you need to specify if the serialization should use camel case. This is done by this statement:

```
// from objects to JSON in camel case
var catgoryAsJson = JsonSerializer.Serialize<Category>(
    catgory,
    new JsonSerializerOptions {
      PropertyNamingPolicy = JsonNamingPolicy.CamelCase
    }
);
```

If you are using previous versions of .NET Core, or prefer to use an external package, you can perform JSON conversion with the Newtonsoft.Json[8] package, which can be obtained from NuGet. Simply add a `using` statement to your file.

```
using Newtonsoft.Json;
```

Conversion can be done like this:

```
var category = new Category();
// from objects to JSON
var categoryAsJson = JsonConvert.SerializeObject(category);
// from JSON text to object
var categoryFromJSON = JsonConvert.DeserializeObject<Category>(categoryAsJson);
```

`Note: In Newtonsoft.Json the camel case problem is handled like this:`

```
JsonConvert.SerializeObject(
  category,
  new JsonSerializerSettings
  {
    ContractResolver = new CamelCasePropertyNamesContractResolver()
  });
```

[7] https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to
[8] https://www.newtonsoft.com/json