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 7 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 (as mentioned above).

Assignment Overview

This assignment consists of two main parts:

1. **Tool Setup**
   Prepare and configure all necessary tools and components required for the project.
2. **Web Server Development**
   Build a lightweight web server capable of handling HTTP requests and responding according to a defined specification.

The goal is to implement a functional server that interprets incoming requests, validates them, and returns appropriate responses based on the protocol rules.

Testing the Assignment

The assignment is accompanied by two test suites, each corresponding to one part of the project:

- Part I – Unit Tests
  These tests verify the correctness of all classes and functionalities developed in the first part, including data handling, request validation, and path parsing.

- Part II – Integration Tests
  These tests evaluate the behavior of the web service by sending HTTP requests and checking the responses.
  Important: You must start your server before running these tests.

If the number of tests feels overwhelming or if some tests cover functionality you haven't implemented yet, you can temporarily disable them using comments or conditional compilation (#if, #endif). This allows you to enable and run tests incrementally without encountering compile-time errors.

CJTP – CIT JSON Transport Protocol

CJTP is a JSON-based protocol used for communication between clients and servers. It is a simplified protocol designed for educational purposes. It mimics the structure of HTTP-based APIs but uses custom formatting to help students focus on core concepts like request validation, routing, and response handling. CJTP operates through structured request and response objects, both formatted in JSON. The server must validate incoming requests and respond accordingly—either with error details if the request is invalid, or with the requested data if the request is valid.

Request Object Structure

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

- method (required)

  Specifies the operation to perform. Valid values are:

  - "create"
  - "read"
  - "update"

- "delete"
- "echo"

The first four are standard CRUD operations. The "echo" method is used for testing, allowing simple text to be sent to the server.

- path (required)

Defines the resource to operate on, similar to a URL path (e.g., "/api/categories"). Combined with the method, it determines the action. For example:

```
{
  "method": "read",
  "path": "/api/categories"
}
```

This would request all categories from the service.

- date (required)

A Unix timestamp (64-bit integer) representing the number of seconds since January 1, 1970. It must be a valid long data type. Optionally, it can be validated to fall within a specific time range.

- body (optional)

Contains the data needed for the operation. The `body` field is required only for certain methods:

- For `"create"` and `"update"`, it must be a valid JSON object representing the data to be stored or modified.
- For `"echo"`, it must be plain text, which will be returned as-is in the response.

Note: All JSON keys and string values must be enclosed in double quotes ("), as per standard JSON syntax.

Request Validation

Before processing, the server must validate the request:

- Ensure all required fields (method, path, date) are present and correctly formatted.
- Confirm that method is one of the allowed values.
- Validate that date is a valid Unix timestamp.

- Check that body is present and correctly structured when required.

If validation fails, the server should return a response detailing the errors. If successful, it should interpret the request and return the appropriate response.

Examples of Valid Requests

```
{
  "method": "read",
  "path": "/api/categories",
  "date": 1633036800
}
```

```
{
  "method": "create",
  "path": "/api/products",
  "date": 1633036800,
  "body": {
    "name": "New Product",
    "price": 19.99
  }
}
```

```
{
  "method": "echo",
  "path": "/test",
  "date": 1633036800,
  "body": "Hello, server!"
}
```

Response Format

Responses from the server follow this JSON structure:

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

- **status (required)**

The `status` field indicates the outcome of the request. It consists of a **status code** and a **reason phrase**, chosen from the following:

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

- Error Reporting for Bad Requests

For status code 4 (Bad Request), the response must include a reason describing the validation issues. The format is:

4 missing <element>, illegal <element>, ...

Where <element> refers to the name of the protocol field (e.g., method, date, path).
Examples:

- Single issue:
  "4 missing date"
- Multiple issues:
  "4 missing date, illegal method, missing body"

The order of the listed reasons is not important.

Part 1 – Supporting Classes for the Web Service

In this part of the assignment, you will implement several foundational classes that support the development of a web service.

Data Service: Category Management

You are provided with a simple in-memory database containing categories:

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

Each Category object has two attributes:

- cid (integer)
- name (string)

You must implement a **CategoryService** class that provides full CRUD functionality through the following interface:

```
public List<Category> GetCategories();
public Category? GetCategory(int cid);
public bool UpdateCategory(int id, string newName);
public bool DeleteCategory(int id);
public bool CreateCategory(int id, string name);
```

Request Validation Service

Create a **RequestValidator** class responsible for validating incoming request objects. It should implement the following method:

```
public Response ValidateRequest(Request request);
```

This method checks whether the request meets all protocol requirements. It returns a Response object with a status indicating either:

- "1 Ok" if the request is valid
- Or an appropriate error code and reason(s) if the request is invalid

URL Parsing Utility

Implement a UrlParser class that can parse and interpret the path string from a request. This class helps interpret the structure of the request path and extract any embedded identifiers (e.g., /api/categories/3 → ID = 3). This is essential for routing requests to the correct data operations. The class should expose the following properties:

```
public bool HasId { get; set; }
public int Id { get; set; }
public string Path { get; set; }
```

And the following method:

```
public bool ParseUrl(string url);
```

This method takes a URL-like path string and validates it. If the path is valid, it returns true; otherwise, false.

When valid, the properties should be populated as follows:

- HasId is true if the path includes an ID
- Id contains the extracted ID
- Path contains the base path without the ID

Example:

Input: "api/categories/1"
Output:

- Path = "api/categories"
- HasId = true
- Id = 1

By completing Part I, you will have built the core components needed to support a structured web service, including data management, request validation, and path parsing.

Part II – Web Service Implementation

In this part of the assignment, you will build a web service using the **TcpListener**[1] and **TcpClient**[2] classes from the .NET framework. The service must follow the **client/server model** and implement the **request-response pattern**[3].

Requirements

- The server must listen for incoming connections on **port 5000**[4].
- Each client connection should be handled independently. While multithreading is not required, handling each client connection in a separate thread is recommended to improve responsiveness and scalability.
- The server must be capable of:
    - Accepting connections from clients
    - Receiving and interpreting requests
    - Sending appropriate responses

Connection Handling

Not all connections will send valid requests. Your server must be able to detect and ignore connections that do not provide a usable request, e.g., malformed JSON, missing fields, or empty or no input.

Protocol Compliance

Your service must fully implement the **CJTP (CIT JSON Transport Protocol)** as described in Part I. This includes:

- Validating the structure and content of incoming requests
- Responding with appropriate status codes and error messages for invalid requests
- Executing CRUD operations on the category data when requests are valid

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

Using Helper Tools in the Web Service

The web service must integrate all the helper tools developed in Part I to handle incoming requests effectively. Each tool plays a specific role in the request lifecycle:

- Category Service: Handles all data operations (Create, Read, Update, Delete) related to category objects. When a valid request targets category data, this service performs the necessary action and returns the result.
- Request Validator: Ensures that incoming requests conform to the CJTP protocol. Before any processing occurs, the request is passed to this validator. If the request is invalid, the validator returns a response with the appropriate error status and reason(s).
- Path Parser: Interprets the path string from the request to determine the target resource and whether an ID is included. This helps route the request to the correct operation in the Category Service.

By combining these tools, the web service can reliably validate, interpret, and respond to client requests according to the CJTP specification.

API Specification

The web service must expose the following endpoint:

/api/categories

All communication between client and server must use JSON format, including both requests and responses. The only exception is the echo method, where the request body is plain text and the response body mirrors that text.
To transmit a category object over the protocol, it must be serialized as JSON. For example:

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

Usage Examples

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	(none)
update	/api/categories	{ cid: 3, name: “Test”}	4 Bad Request	(none)
create	/api/categories	{ name: “Seafood”}	2 Created	{“cid”: 4, “name”: “Seafood”}
delete	/api/categories/3		1 Ok	(none)
delete	/api/categories/123		5 Not Found	(none)
read	/api/categories/123		5 Not Found	(none)

Method Behavior

- **read**
  - **List all categories**: Use /api/categories.
  - **Get a specific category**: Use /api/categories/<id>.
  - If the requested category does not exist, return:
    5 Not Found.
- **create**
  - Use /api/categories with the new category data in the request body.
  - Including an ID in the path is **invalid** and should return:
    4 Bad Request.
  - On success, return:
    2 Created and the newly created category.
- **update**
  - Use /api/categories/<id> with the updated category data in the body.
  - If the path does **not** include an ID, return:
    4 Bad Request.
  - On success, return:
    3 Updated.
- **delete**
  - Use /api/categories/<id> to delete a category.
  - If the category does not exist, return:
    5 Not Found.
  - On success, return:
    1 Ok.
- **echo**
  - The path is ignored.
  - The request body must be plain text.
  - The response body will mirror the input text.
  - Always return:
    1 Ok.

By completing Part II, you will have implemented a functional network service using TCP in C#, capable of handling structured requests, validating input, and responding according to a custom protocol. This builds foundational skills for working with web services and networked applications.

Appendix

Converting to and from JSON

If you use .NET Core 3.0(or later) JSON support is included as part of the framework[5]. 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 categoryAsJson = 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 categoryAsJson = JsonSerializer.Serialize<Category>(
    category,
    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[6] 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(

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

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