Week 8 Assignment: Serialization for the Parking System Application: Client/Server for

ICT 4315-1 Object-Oriented Methods & Paradigms II

Jack Hermanson

University of Denver University College

May 25, 2025

Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

Introduction

This paper discusses the basic implementation of a client/server relationship for the parking application, using HTTP requests, serialization/deserialization, and command line arguments. This sets the stage for the application to run and maintain its own state while another actor interacts with the application using HTTP requests. As part of this assignment, I also created a small program (executed from the command line) that handles all of the HTTP requests heavy lifting for the user, giving them a simple interface to work with. Other options might include a front-end web application with an HTML form that the user can interact with intuitively, using CURL from the command line, or using an API testing tool like Postman or Insomnia (Raunak 2025).

Properly installing and configuring Gson, researching and implementing Java HTTP servers, and testing everything took an extremely long time, requiring great focus and time dedicated to this assignment. As such, the codebase is at a good "stopping point" now, where a basic HTTP client/server relationship is operational and all tests pass, but more effort will be needed to fully implement each command and ensure proper function.

Server Setup and Functionality

I have never set up a Server in Java before; my only experience comes from pre-built solutions or working with frameworks like Spring Boot. This server's needs are relatively simple; it needs to do the following things:

1. Run on some port (like 8000) and listen for requests/connections
2. Accept a POST request to a "/command" endpoint

3. Deserialize the request JSON into a ParkingRequest instance
4. Perform some logic using that instance, dependening on the command
5. Create an appropriate ParkingResponse instance
6. Serialize the ParkingResponse instance to JSON
7. Send the serialized response back in an HTTP response to the request

This server was implemented using the HTTPServer class in the com.sun.net.httpserver package (Paul, 2022). It creates a "context" for the "/command" URL, then sets a "handler," which is a method that will be called when a request is sent to that endpoint. The handler method is responsible for the following actions:

1. Ensure that the request method is POST, returning a "405" error status code if the wrong request method is used
2. Read the request body bytes into a string
3. Deserialize that string from JSON into a ParkingRequest using the static ParkingRequest.fromJson() method
4. Create a ParkingResponse instance using the data from the ParkingRequest
5. Serialize the ParkingResponse instance into a JSON string using the toJson() method on the instance
6. Correctly set headers, specify a "200" status code, and return the response with the serialized ParkingResponse instance as a JSON string in the response body

The Server class has a "main" method, allowing it to be executed from the command line. It expects the first argument to be the name of a command, and it expects subsequent arguments to be properties for that command, which we set using the Properties class.

Serialization and Deserialization

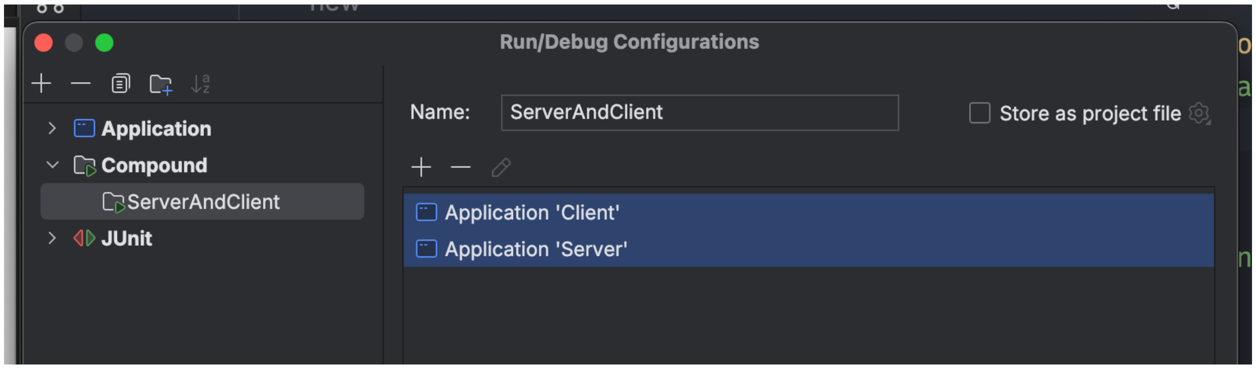
For this assignment, JSON was a good choice for serialization. It is human-readable, and it is easy for me to mentally picture a Java class as a JSON string. I used the Gson library and decided to create a single instance with all of the options I wanted to set in a single JsonSerializer class with a static Gson instance that any other class can use. I also had to tell Gson which properties to serialize, and there were a few ways to do that, but the simplest was to put the @Expose decorator above properties that I wanted to serialize. The ParkingRequest and ParkingResponse classes both used this class to serialize themselves to JSON using Gson's .toJson() method in their own .toJson() methods.

To deserialize from a JSON string into an instance of a class, I used that same Gson instance's .fromJson() method, passing it the JSON string and the type of class it should deserialize to. The ParkingRequest and ParkingResponse classes both have fromJson() methods that do this, and I used JUnit to test that they worked as expected.

Challenges

This assignment was very challenging. There are several ways to write a server, so I tried to follow methods that were simple and easy to understand without having to add external dependencies. Many online sources expect you to use a framework like Spring Boot. I took several approaches before settling on the final one.

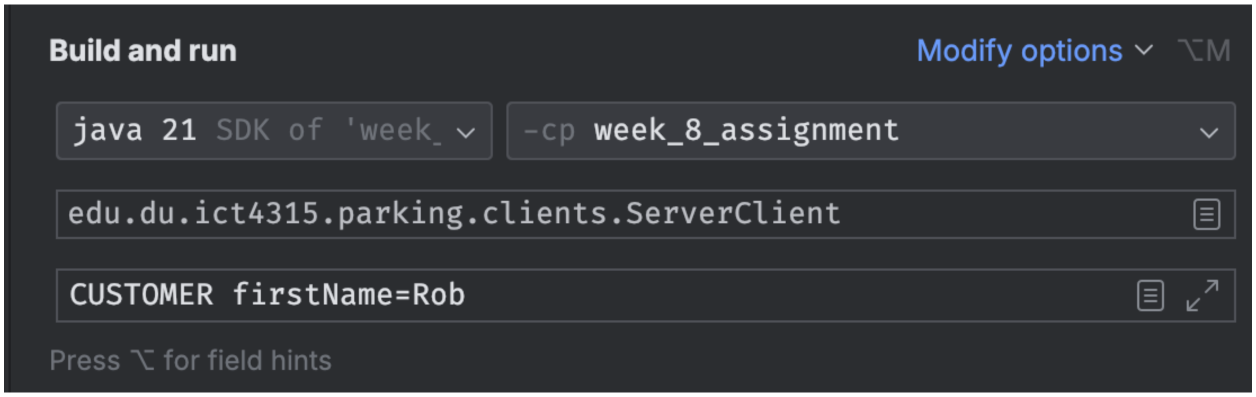
Another challenge was running two main methods at the same time and dealing with class path issues. The instructions on the assignment mostly seemed to work, but I kept having issues with Gson not being found. To work around this, I used my IDE's run/debug configurations to run the client and server simultaneously while passing in any command line arguments that make their way to the main method. See below:



Name: ServerAndClient

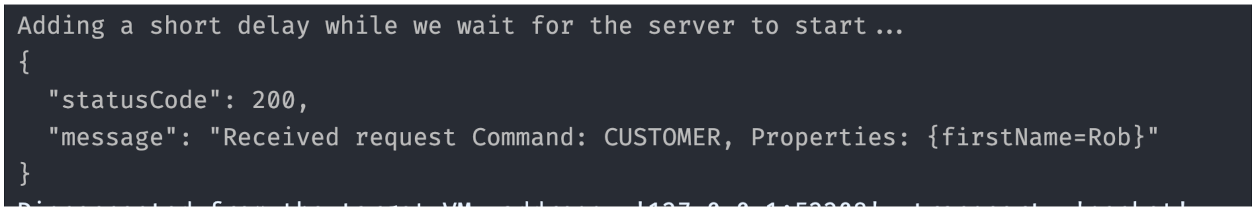
Application 'Client'
Application 'Server'

This is the Client part of the configuration, and you can see that I am passing in CUSTOMER as the command along with firstName=Rob:



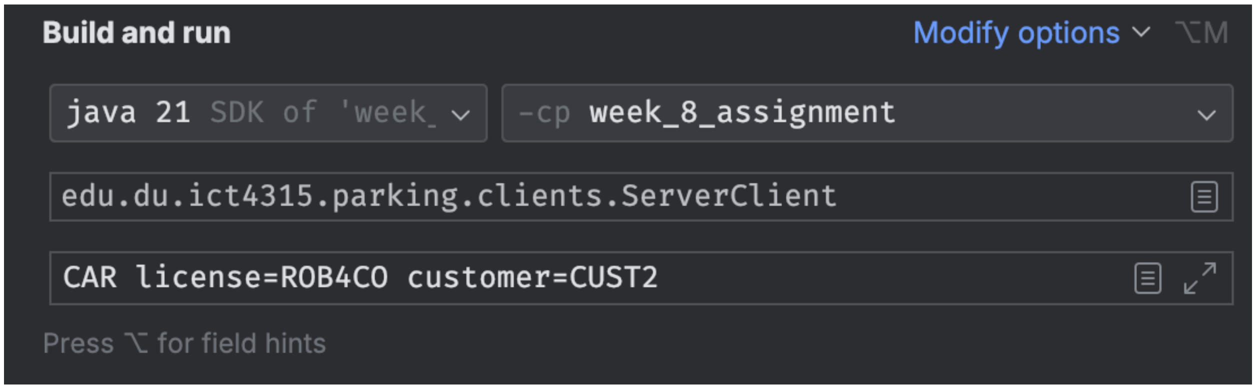
```
java 21 SDK of 'week_
-cp week_8_assignment
edu.du.ict4315.parking.clients.ServerClient
CUSTOMER firstName=Rob
```

That configuration produces this output:



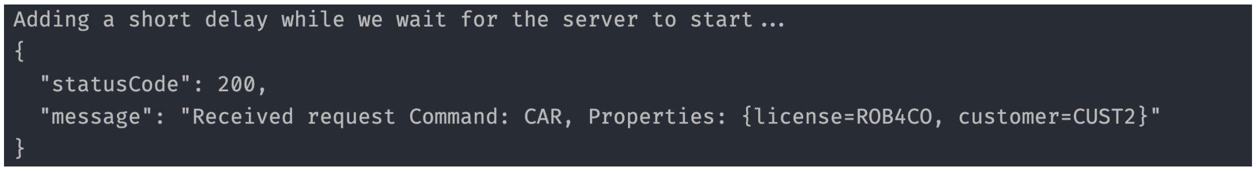
```
Adding a short delay while we wait for the server to start...
{
  "statusCode": 200,
  "message": "Received request Command: CUSTOMER, Properties: {firstName=Rob}"
}
```

For the other command, CAR, this is what the configuration looks like:



```
java 21 SDK of 'week_
-cp week_8_assignment
edu.du.ict4315.parking.clients.ServerClient
CAR license=ROB4CO customer=CUST2
```

That configuration produces this output:

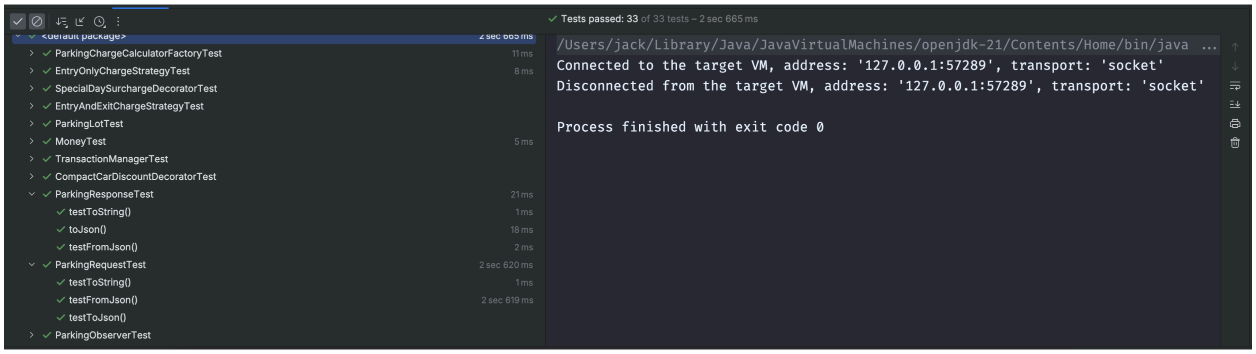


```
Adding a short delay while we wait for the server to start...
{
  "statusCode": 200,
  "message": "Received request Command: CAR, Properties: {license=ROB4CO, customer=CUST2}"
}
```

You may also notice the “Adding a short delay” line, which was necessary because the client would attempt to connect to the server before the server had gotten to a point where it was listening for requests. I worked around that by adding a 2 second delay. Using the IDE’s configuration also allowed me to place breakpoints and use the IDE’s debugger, which was helpful.

Program Compiles and Tests Pass

The program builds successfully and all unit tests pass.



```
<default package>                       2 sec 665 ms
  ParkingChargeCalculatorFactoryTest    11 ms
  EntryOnlyChargeStrategyTest           8 ms
  SpecialDaySurchargeDecoratorTest
  EntryAndExitChargeStrategyTest
  ParkingLotTest
  MoneyTest                             5 ms
  TransactionManagerTest
  CompactCarDiscountDecoratorTest
  ParkingResponseTest                   21 ms
    testToString()                      1 ms
    toJson()                            18 ms
    testFromJson()                      2 ms
  ParkingRequestTest                    2 sec 620 ms
    testToString()                      1 ms
    testFromJson()                      2 sec 619 ms
    testToJson()
  ParkingObserverTest
```

```
Tests passed: 33 of 33 tests – 2 sec 665 ms
/Users/jack/Library/Java/JavaVirtualMachines/openjdk-21/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:57289', transport: 'socket'
Disconnected from the target VM, address: '127.0.0.1:57289', transport: 'socket'

Process finished with exit code 0
```

References

Jain, Raunak. 2025. “Insomnia vs Postman: Which is Better for Your Project?” *Testsigma*. https://testsigma.com/blog/insomnia-vs-postman/.

Paul, Sayan. 2022. “Develop an HTTP Server in Java.” *Medium*, February 5, 2022. https://medium.com/@sayan-paul/develop-an-http-server-in-java-2137071a54a1.