# Ski Resort Client Documentation

GitHub Repo: https://github.com/zz39/SkiResort-CS6650

## Configuration

### Server Configurations

1. Deploy the `Assignment1.war` file to your EC2 instance running Tomcat.
2. Ensure that your EC2 instance is properly configured to allow traffic on port 8080.
3. Start the Tomcat server and verify that the application is running by accessing `http://{server-public-ip}:8080/Assignment1/`.
4. Make sure the API endpoints are accessible and functioning correctly.

### Client Configurations

1. Update API endpoint in `PostingSkiInfo.java`:

```
private SkiersApi createApiClient() {
    ApiClient apiClient = new ApiClient();
    apiClient.setBasePath("http://{server-public-ip}:8080/Assignment1/");
    return new SkiersApi(apiClient);
}
```

2. Adjust thread (Optional) and request settings in `MultithreadedClient.java`:

```
// example:
private static final int INITIAL_THREADS = 32;        // Phase 1 threads
private static final int PHASE2_THREADS = 100;        // Phase 2 threads
private static final int TOTAL_REQUESTS = 200000;    // Total requests to
send
```

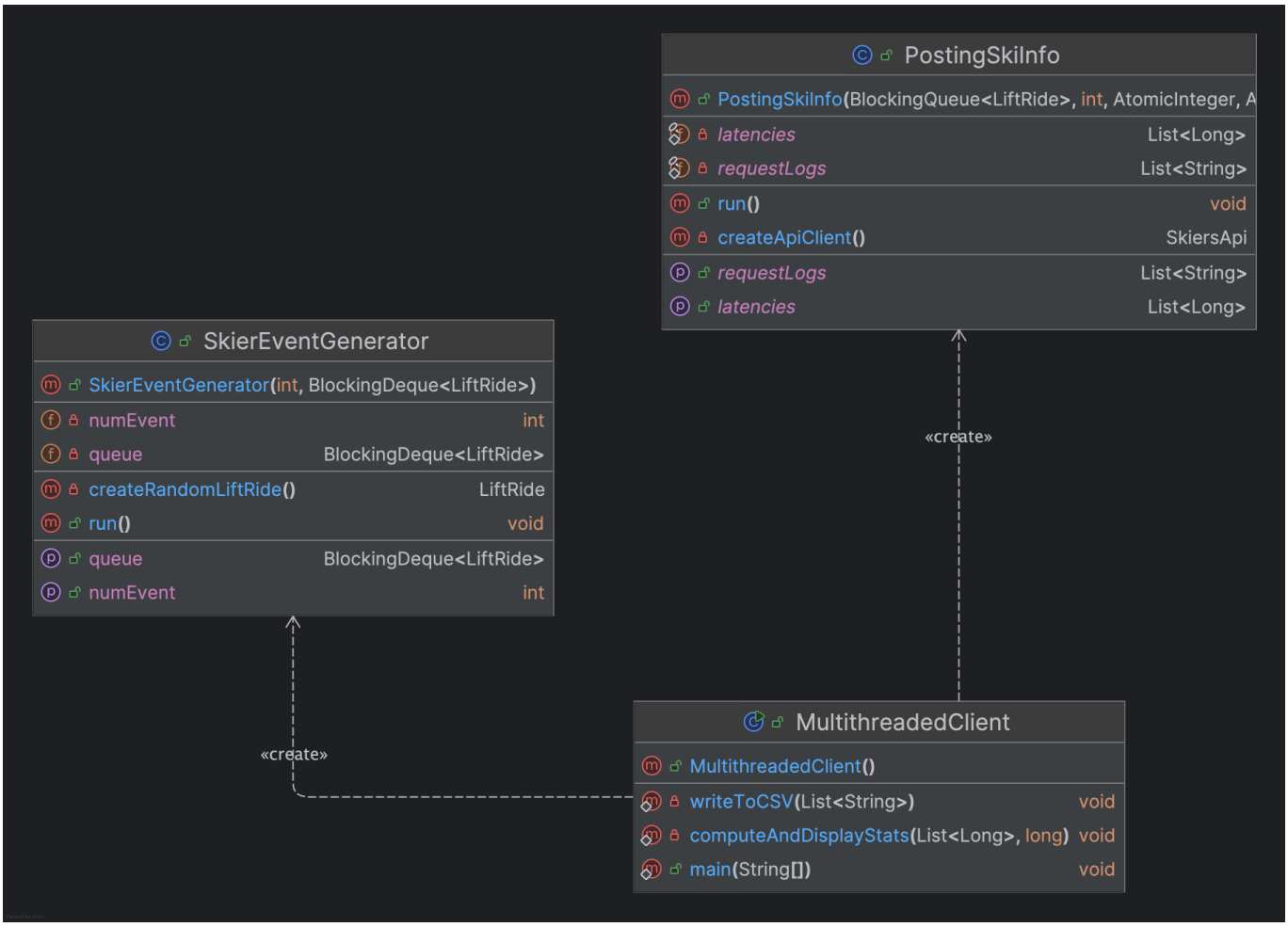
3. Simply run `MultithreadedClient.java`:

```
javac MultithreadedClient.java
java MultithreadedClient
```

## Design Documentation

### Client Design Overview

The client implements a multi-threaded architecture to efficiently send POST requests to a ski resort API. It uses a two-phase approach to handle 200,000 POST requests with optimal throughput.




## Key Components

1. **MultithreadedClient (Main)**
    - Manages overall execution flow
    - Handles statistics collection and reporting
    - Coordinates between event generation and request processing
2. **SkierEventGenerator (Producer)**
    - Generates random lift ride events
    - Populates shared blocking queue
    - Ensures thread-safe event distribution
3. **PostingSkiInfo (Consumer)**
    - Processes POST requests to API
    - Implements retry logic
    - Records latency metrics
    - Manages request success/failure tracking

## Thread Safety Considerations

- Uses `BlockingQueue` for safe event distribution
- Atomic counters track successful/failed requests
- CountDownLatch ensures phase synchronization

## Threading Strategy

- Phase 1: 32 threads × 1000 requests = 32,000 requests
- Phase 2: 100 threads × ~1680 requests = 168,000 requests
- ThreadPoolExecutor manages execution

## Performance Analysis

### 32-100 Thread Approach

- Success Rate: 100%
- Mean Response Time: 98.61 ms
- 99th Percentile: 190 ms
- Throughput: 291.57 requests/second

### Experiment: 200 Threads × 100 Requests

- Success Rate: 100%
- Mean Response Time: 98.0 ms
- 99th Percentile: 196 ms
- Throughput: 1946.03 requests/second

### Key Insights

- Increasing threads to 200 significantly boosted throughput (~6.6x improvement).
- Response times remained stable, indicating efficient concurrency handling.
- The setup effectively balances performance and resource utilization.

## Client Outputs

### Client-1 (initial configuration)

```
Successful requests: 200000
Failed requests: 0
Total time spent: 672502 ms
Throughput: 297.39688506502586 requests/second
```

### Client-2 (initial configuration)

```
All requests completed
Successful requests: 200000
Failed requests: 0
Collected 200000 latency measurements
Statistics:
Mean response time: 98.60778 ms
Median response time: 96.0 ms
99th percentile response time: 190 ms
Min response time: 79 ms
Max response time: 513 ms
Throughput: 291.5715414512682 requests/second
Latency data saved to latencies.csv
Client shutdown complete
```

**Client-2 (optimized configuration)**

```
Collected 200004 latency measurements
Statistics:
Mean response time: 101.30705385892283 ms
Median response time: 98.0 ms
99th percentile response time: 196 ms
Min response time: 81 ms
Max response time: 512 ms
Throughput: 1946.0364103412376 requests/second
Latency data saved to latencies.csv
Client shutdown complete
```

## RESTful API Screenshots

To show clients actually send requests to the server on EC2 instance - using Postman testing page showing URL