

CS6650 Assignment 2: Load Testing Client-Server with Persistent Data

Rahul Arun Kumar [<https://github.com/rahulleoak/CS6650>]

→ Client Description

Class *Client2* and its associated *RequestHandler* class, are designed to perform load testing on web servers. *Client2* is responsible for generating concurrent *HTTP* requests, both *GET* and *POST*, to a specified endpoint. The *RequestHandler* manages the actual sending and receiving of these requests. The system is built to simulate high-traffic conditions to test server responsiveness and robustness.

Load testing is a critical component of software quality assurance, ensuring that applications can handle the anticipated load in production. The *Client2* and *RequestHandler* classes are developed to conduct automated load testing by sending multiple *HTTP* requests to a web server and recording the server's responses.

The *Client2* class is engineered to simulate multiple users by using multi-threaded execution, allowing for parallel request processing. It manages a set number of threads that each sends a predefined number of *API* calls. These are further broken down into *ThreadGroups*, the *NumberOfThreadsPerGroup* and 1000 *API* calls. The *RequestHandler* class serves as the communication bridge between *Client2* and the web server, executing *GET* and *POST* requests and handling the server's responses.

Key Notes of the Design

Concurrency Management: *Client2* leverages Java's *ExecutorService* framework to handle multiple threads, each representing a virtual user sending requests to the server. This simulates a real-world scenario where multiple users interact with the server simultaneously.

Configurable Load Parameters: The client is configured with specific constants to control the volume and frequency of requests, allowing testers to adjust the load based on testing requirements.

Request Record-Keeping: *Client2* maintains a concurrent queue to log details of each *HTTP* request, essential for subsequent analysis of the server's performance under load.

Resiliency and Retries: The code incorporates a retry mechanism that uses *exponential backoff*, which is crucial for testing the server's behavior under suboptimal network conditions.

Metrics and Reporting: After the load test is completed, *Client2* can output the request logs to a CSV file for further analysis. It also computes various latency statistics, such as *mean*, *median*, and *99th percentile*, which are vital metrics in load testing.

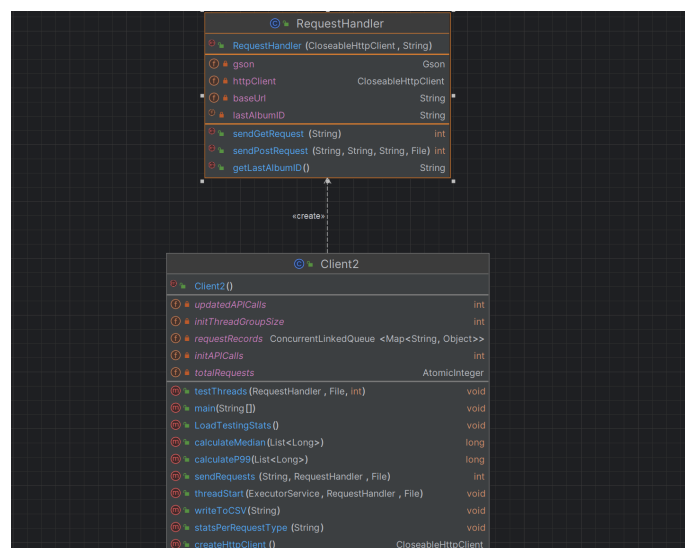
Technical Specifications

CloseableHttpClient Initialization: A custom-configured *HTTP* client is used with timeout settings and connection pooling to optimize network resource usage during the test.

Request Handling: The *RequestHandler* class encapsulates the logic for making *HTTP GET* and *POST* requests. It dynamically handles responses, particularly extracting and storing the *albumID* from *POST* requests to maintain state between calls.

Content Assembly: For *POST* requests, *RequestHandler* creates *multipart* entities combining both *JSON* and *binary data* (such as images), which reflects a common requirement in web applications to handle mixed content types.

The *Client2* and *RequestHandler* constitute an effective load testing tool, capable of simulating a variety of conditions to assess the performance and stability of web servers. The architecture of the system allows for flexible test configuration, robust error handling, and detailed performance metrics, which are essential features for a load testing suite.



→ Server Description

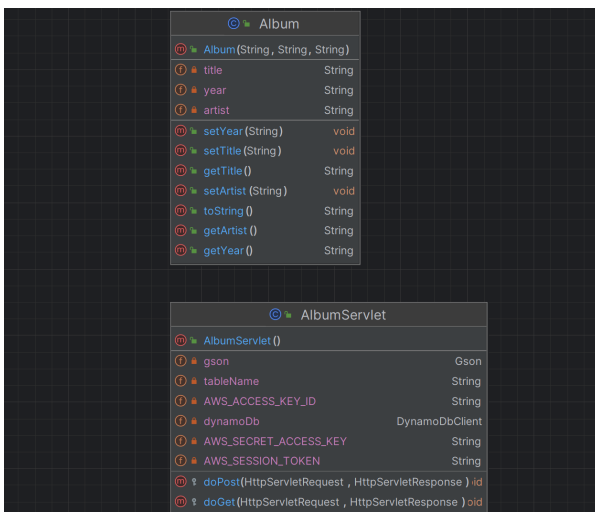
The server code represents a *Java* servlet that serves as a backend for managing *album data*, interfacing with *Amazon DynamoDB* to store and retrieve *album* information. The *servlet* is configured to handle *multipart* file uploads up to *50 MB* and support requests up to *100 MB*.

The *AlbumServlet* class extends *HttpServlet* and is annotated with *@WebServlet*, mapping it to the */albums* URL pattern. The *servlet* handles two types of *HTTP* requests

HTTP GET: Retrieves *album* data from *DynamoDB*. It expects an *albumID* as part of the URL path, which it uses to query the *DynamoDB* table for an *album*'s profile. If found, it sends back the *album* data in *JSON* format to the client.

HTTP POST: Stores new *album* data in *DynamoDB*. It generates a new *albumID* through *UUID* and expects *multipart* form data containing the *album* profile and an image. The *profile* is read as a *string* and converted to an *Album* object using *Gson* for *JSON* processing. The image's size is determined from the uploaded file part. The *album* data, including the *albumID*, *profile*, and *image size*, is then stored in a *DynamoDB* table.

The *servlet* uses *AWS SDK for Java* to create a *DynamoDbClient*. The client is configured to connect to the *DynamoDB* service in the *US_WEST_2* region. The *servlet* provides *JSON*-formatted responses and handles errors by setting appropriate *HTTP* status codes and printing error messages.



Security Note:

The code contains hardcoded *AWS* access keys, which is a bad security practice. These credentials can potentially grant access to *AWS* resources. Typically, credentials should be stored securely, for example, in environment variables, *AWS Secrets Manager*, or *IAM* roles, especially when deploying to *AWS infrastructure* like *EC2* where *IAM* roles are the recommended approach. It is crucial to never expose *AWS* credentials in source code, especially in public repositories or shared codebases. However since our *AWS Learner Lab* doesn't provide *IAM* Roles, the credentials were obtained through the *AWS CLI* using the command “*cat .aws/credentials*”

```
eee_W_2405195@runweb100797:~$ cat .aws/credentials
```

→ DynamoDB Schema

The database chosen for this homework is *Amazon's DynamoDB*. *DynamoDB* is a fully-managed *NoSQL* database service provided by *AWS*, designed for high performance and scalability. Its integration in a client-server load testing scenario provides numerous advantages, such as managed scalability, low latency, and high throughput. The album table, with albumID as the *partition key* and configured for *on-demand capacity*, serves as the data store in the examined load testing scenario, where the server component interacts with *DynamoDB* to fetch and store album information. The role of this database is to manage and persist album data, which includes the storage (and retrieval) of *album ID's*, *album profiles* and *associated images*.

Advantages of Using DynamoDB

Managed Scalability: *DynamoDB* handles scaling automatically, which is beneficial during load testing, as it can accommodate sudden spikes in traffic without manual intervention.

Performance: It offers consistent, single-digit millisecond response times at any scale, ensuring that the load tests reflect the performance clients can expect in production.

Maintenance-Free: As a fully-managed service, it requires no routine maintenance, allowing the focus to remain on application development and testing.

Flexible Data Model: Being a *NoSQL* database, it supports varied data structures, which is advantageous for diverse datasets like *albums* and *images*.

Integrated with AWS: It offers tight integration with other *AWS* services, providing a comprehensive environment for deploying and managing applications.

On-Demand vs Provisioned Capacity Modes

On-Demand Capacity: This mode requires no capacity planning. Costs are based on the actual reads and writes the application performs. It is suitable for unpredictable workloads and is preferred for new applications with unknown usage patterns. Given the nature of load testing, which can have variable and unpredictable read/write rates, on-demand capacity can handle the load without any provisioning.

Provisioned Capacity: In this mode, the capacity must be specified in terms of read and write units. It is appropriate for applications with predictable traffic and can be more

cost-effective if usage patterns are well understood. However, it requires monitoring and potentially manual scaling or auto-scaling setup to handle traffic changes.

The *album* table in *DynamoDB*, using the *on-demand* capacity mode, is leveraged to store album data. In the load testing scenario, where multiple virtual users created by *Client2* may simultaneously send requests, *DynamoDB*'s on-demand mode allows for seamless performance without the need to manage throughput capacity settings, them being Read Capacity Units (RCU) and Write Capacity Units (WCU). This ensures that the *database* does not become a *bottleneck* during tests and can handle the *throughput* required by the client application without throttling.

<input type="checkbox"/>	Name ▲	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode	Write capacity mode	Total size	Table class
<input type="checkbox"/>	album	Active	albumID (S)	-	0	Off	On-demand	On-demand	0 bytes	Standard

DynamoDB > Tables > album

Tables (1)

Any tag key

Any tag value

Find tables by table name

album

album

OverviewIndexesMonitorGlobal tablesBackupsExports and streamsAdditional settings

Protect your DynamoDB table from accidental writes and deletes

When you turn on point-in-time recovery (PITR), DynamoDB backs up your table data automatically so that you can restore to any given second in the preceding 35 days. Additional charges apply. [Learn more](#)

Edit PITR

General information

Partition key

albumID (String)

Sort key

-

Capacity mode

On-demand

Table status

Active

Alarms

No active alarms

Point-in-time recovery (PITR)

Off

Additional info

Items summary

DynamoDB updates the following information approximately every six hours.

Item count

0

Table size

0 bytes

Average item size

0 bytes

Get live item count

Running Tests on a Single Instance with a Database:

Table Showing the effect on Throughput by with configurations of 10 ThreadGroups as a constant on a single servlet instance connected to DynamoDB

	Threads	Wall Time	Mean	Median	P99	M i n	Max	Successful Request	Failed Requests	Throughput
POST	10	189	99	73	893	3	1583	200,000	0	1058.2
GET			66	43	851	2	1607			
POST	20	383	108	80	896	3	1737	400,000	0	1044.39
GET			72	47	862	2	1674			
POST	30	724	164	81	1265	3	5069	600,000	0	828.73
GET			63	41	682	2	3465			

It is evident to see that on low workloads of 10 threads with 10 thread groups, the system is able to hold up somewhat. As we increase the workload however, we can see a dropoff in performance, with higher average latencies, lower but high P99 numbers, and high max values. Throughput as a result of all of this is greatly affected and some change to the system needs to take place. Even as we are not at risk of failed requests, mostly due to the fact that the server exists on a t2.large instance, this performance is considerably lower than what we achieved in the first homework, even after the consideration of the addition of a database and it's respective operations.

For screenshots, please refer to the end where they are attached

Running Tests on Two Instances with a Database and a Load Balancer:

Table Showing the effect on Throughput by with configurations of 10 ThreadGroups as a constant and 2 instances connected to AWS ALB

	Threads	Wall Time	Mean	Median	P99	M i n	Max	Successful Request	Failed Requests	Throughput
POST	10	106	57	52	148	4	1142	200,000	0	1886.79
GET			30	27	92	2	1092			
POST	20	202	60	50	177	3	5035	400,000	0	1980.12
GET			32	22	109	2	1838			
POST	30	297	59	50	166	3	1455	600,000	0	2020.20
GET			31	22	102	2	1440			

We seem to be achieving better throughput given our workloads as the requests are being sent into two different servers. The application load balancer (ALB) is helping distribute the incoming requests across multiple instances, preventing any single instance from being overwhelmed, hence achieving the increased throughput. By utilizing the combined processing power of the instances, we can ensure to some extent the optimal use of resources and minimizing response time, thus enhancing the application's ability to handle more concurrent requests efficiently. One thing I did find peculiar is the recorded latencies for get seem to be lower when comparing the 30 threads to the 20, which is not following the pattern of an increase in latency, it could be an edge case of where the number of users on AWS was low or tomcat was able to handle the requests better given the higher load and the help of the load balancer.

For screenshots, please refer to the end where they are attached

Optimizing Performance To Increase Throughput By Identifying Bottlenecks:

In the context of web applications, bottlenecks can significantly impact throughput and overall performance. Two common types of bottlenecks are: Database bottlenecks and Servlet bottlenecks.

Database bottlenecks occur when the demand for database operations exceeds the database's ability to handle them efficiently. Some causes could be insufficient indexing, leading to slow query performance, inadequate hardware, such as limited CPU/Memory or a poorly designed database schema. Servlet Bottleneck occurs when a web server or servlet container like Tomcat cannot handle the number of incoming HTTP requests. Common causes include limited number of threads to handle incoming requests, resource-intensive computations or blocking operations within servlets or just inadequate server resources, leading to high CPU or memory utilization.


With respect to DynamoDB, throughput bottlenecks can occur if the allocated read/write capacity units are exceeded, leading to throttled requests. Increasing these units or opting for on-demand capacity can alleviate these issues. Hence not many changes can be made here, since my configuration is already on-demand. For my java servlets, bottlenecks might arise from limited compute resources or insufficient instance scaling. This can be mitigated by scaling out where we add more instances behind a load balancer or scaling up where we upgrade to instances with more CPU/Memory. Since my servlets already run on the largest option available to me (t2.large), one option to help alleviate bottlenecks may be to increase the number of instances behind my load balancer.

I decided to test by adding 3 and 4 instances to my target group for my load balancer to see how it may affect the throughput.

Table Showing the effect on Throughput by adding 2,3,4 more instances to AWS ALB on configuration 10 ThreadGroups and 30 Threads:

	# of Instance	Wall Time	Mean	Median	P99	Min	Max	Successful Request	Failed Requests	Throughput
POST	2	297	59	50	166	3	1455	600,000	0	2020.20
GET			31	22	102	2	1440			
POST	3	300	60	53	161	3	1301	600,000	0	2000.00
GET			31	22	103	2	1328			
POST	4	266	54	43	148	4	5026	600,000	0	2255.64
GET			27	21	93	2	1865			

Again a peculiar result on 3 instances, where I expected to see a rise in Throughput, but instead see it decrease by 20 units of throughput, which can be negligible but nonetheless odd. It could also be due to the fact that AWS reported errors on the learners platform on the day of testing that skewed some/all the results here:



AWS Academy Lab Outage - 5 Nov 2023

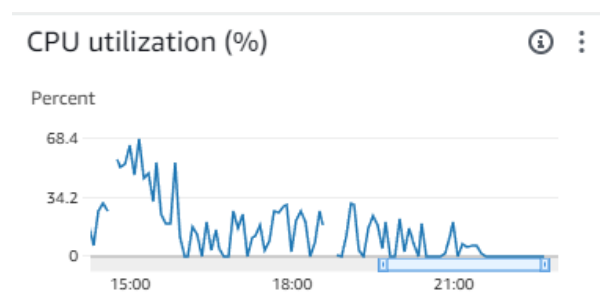
AWS Academy has received reports of an outage affecting lab environments in most courses. This error will prevent labs from launches. Please do not attempt to open a lab as we work to resolve this issue. Thank you for your patience.

This is a message from AWS Academy



However, we can observe that at 4 instances, the throughput increases to 2255.64, which is an 11% increase, which is small but noticeable, it is here where I decided adding more instances to the system is not beneficial as we are seeing small percent changes and 4 seems to be optimum for performance. Given that from our initial test on a single instance our throughput was 828.73 to getting a throughput now of 2255.64, a considerable 172.18% increase in performance, the system is now way better optimized at handling loads in the form of requests.

It is more noticeable when you see the change in CPU utilization, where we can see around 3pm when we ran initial tests the



CPU usage was at nearly 70% and it gradually came down with the addition of ALB and more instances to below 34%.

Some other changes I made was to tomcats config <servers.xml> file:

```
<Connector port="8080" protocol="HTTP/1.1"
  connectionTimeout="50000"
  keepAliveTimeout="10000"
  maxKeepAliveRequests="100"
  redirectPort="8443"
  maxThreads="1000"
  maxParameterCount="1000"
/>
```

I increased the maxThreadCount to 1000, specified timeouts for connections and introduced a keep alive time for requests. These were done in the hopes that it would allow Tomcat to handle more concurrent requests, which can improve throughput up to the point where hardware resources (CPU, memory) are fully utilized, help in freeing up threads that would otherwise be stuck waiting for slow clients, making them available for new incoming requests, which can improve throughput and keeping connections alive for less time can free up server resources more quickly for new connections.

Some additional notes:

- The client and the server both run on different AWS instance, this was done due to:
 - My internet speed is so erratic that sometimes it works perfectly, while other times, the requests take too long to send
 - My internet base speed is too slow to get good throughput
 - Running them on the instances of the same definition, allows for a more concrete set of test results as the environment remains the same.
- I occasionally ran into some errors of “java.net.BindException: Cannot assign requested address” but I chalked it up due to the fault message issued by AWS as on some runs I had this and the other runs it was ok. I verified by getting a live count on my database, to ensure that the expected number of entries on the DB exist.

For screenshots, please refer to the end where they are attached

Combined Table with Results

	Threads	Wall Time	Mean	Median	P99	M i n	Max	Successful Request	Failed Requests	Throughput
POST	10	189	99	73	893	3	1583	200,000	0	1058.2
GET			66	43	851	2	1607			
POST	20	383	108	80	896	3	1737	400,000	0	1044.39
GET			72	47	862	2	1674			
POST	30	724	164	81	1265	3	5069	600,000	0	828.73
GET			63	41	682	2	3465			

	Threads	Wall Time	Mean	Median	P99	M i n	Max	Successful Request	Failed Requests	Throughput
POST	10	106	57	52	148	4	1142	200,000	0	1886.79
GET			30	27	92	2	1092			
POST	20	202	60	50	177	3	5035	400,000	0	1980.12
GET			32	22	109	2	1838			
POST	30	297	59	50	166	3	1455	600,000	0	2020.20
GET			31	22	102	2	1440			

	# of Instance	Wall Time	Mean	Median	P99	M i n	Max	Successful Request	Failed Requests	Throughput
POST	2	297	59	50	166	3	1455	600,000	0	2020.20
GET			31	22	102	2	1440			
POST	3	300	60	53	161	3	1301	600,000	0	2000.00

[illegible]

Screenshots

- Testing with a Single Instance
 - Java Servlet on EC2 Instance (t2.large) with DynamoDB

10 Threads

```
Total Requests: 200000
Successful requests: 200000
Failed Requests: 0

Wall Time: 189 seconds
Throughput: 1058.2010582010582 requests per second
-----

Statistics for GET requests:
Mean latency: 66 ms
Median latency: 43 ms
99th percentile latency: 851 ms
Min latency: 2 ms
Max latency: 1607 ms
-----

Statistics for POST requests:
Mean latency: 99 ms
Median latency: 73 ms
99th percentile latency: 893 ms
Min latency: 3 ms
Max latency: 1583 ms
-----
```

20 Threads	<pre>Total Requests: 400000 Successful requests: 400000 Failed Requests: 0 Wall Time: 383 seconds Throughput: 1044.3864229765013 requests per second ----- Statistics for GET requests: Mean latency: 72 ms Median latency: 47 ms 99th percentile latency: 862 ms Min latency: 2 ms Max latency: 1674 ms ----- Statistics for POST requests: Mean latency: 108 ms Median latency: 80 ms 99th percentile latency: 896 ms Min latency: 3 ms Max latency: 1737 ms ----- [ec2-user@ip-172-31-17-125 ~]\$</pre>
30 Threads	<pre>Total Requests: 600000 Successful requests: 600000 Failed Requests: 0 Wall Time: 724 seconds Throughput: 828.7292817679559 requests per second ----- Statistics for GET requests: Mean latency: 63 ms Median latency: 41 ms 99th percentile latency: 682 ms Min latency: 2 ms Max latency: 3465 ms ----- Statistics for POST requests: Mean latency: 164 ms Median latency: 81 ms 99th percentile latency: 1265 ms Min latency: 3 ms Max latency: 5069 ms -----</pre>

- Testing with on Two Instances linked via a Load Balancer
 - Java Servlet on 2 EC2 Instance (t2.large) connected via Application Load Balancer with DynamoDB

10 Threads	<pre>Total Requests: 200000 Successful requests: 200000 Failed Requests: 0 Wall Time: 106 seconds Throughput: 1886.7924528301887 requests per second ----- Statistics for GET requests: Mean latency: 30 ms Median latency: 27 ms 99th percentile latency: 92 ms Min latency: 2 ms Max latency: 1092 ms ----- Statistics for POST requests: Mean latency: 57 ms Median latency: 52 ms 99th percentile latency: 148 ms Min latency: 4 ms Max latency: 1142 ms -----</pre>
------------	---

20 Threads

```
Total Requests: 400000
Successful requests: 400000
Failed Requests: 0

Wall Time: 202 seconds
Throughput: 1980.1980198019803 requests per second
-----

Statistics for GET requests:
Mean latency: 32 ms
Median latency: 22 ms
99th percentile latency: 109 ms
Min latency: 2 ms
Max latency: 1838 ms
-----

Statistics for POST requests:
Mean latency: 60 ms
Median latency: 50 ms
99th percentile latency: 177 ms
Min latency: 3 ms
Max latency: 5035 ms
-----
```

30 Threads

```
Total Requests: 600000
Successful requests: 600000
Failed Requests: 0

Wall Time: 297 seconds
Throughput: 2020.20202020202 requests per second
-----

Statistics for GET requests:
Mean latency: 31 ms
Median latency: 22 ms
99th percentile latency: 102 ms
Min latency: 2 ms
Max latency: 1440 ms
-----

Statistics for POST requests:
Mean latency: 59 ms
Median latency: 50 ms
99th percentile latency: 166 ms
Min latency: 3 ms
Max latency: 1455 ms
-----
```


- Optimizing Throughput
 - Different Number of Instances tested on ec2 Instances (t2.large) connected via Application Load Balancer

3 Instances	<pre>Total Requests: 600000 Successful requests: 600000 Failed Requests: 0 Wall Time: 300 seconds Throughput: 2000.0 requests per second ----- Statistics for GET requests: Mean latency: 31 ms Median latency: 22 ms 99th percentile latency: 103 ms Min latency: 2 ms Max latency: 1328 ms ----- Statistics for POST requests: Mean latency: 60 ms Median latency: 53 ms 99th percentile latency: 161 ms Min latency: 3 ms Max latency: 1301 ms -----</pre>
-------------	--

4 Instances

```
Total Requests: 600000
Successful requests: 600000
Failed Requests: 0

Wall Time: 266 seconds
Throughput: 2255.6390977443607 requests per second
-----

Statistics for GET requests:
Mean latency: 27 ms
Median latency: 21 ms
99th percentile latency: 93 ms
Min latency: 2 ms
Max latency: 1865 ms
-----

Statistics for POST requests:
Mean latency: 54 ms
Median latency: 43 ms
99th percentile latency: 148 ms
Min latency: 4 ms
Max latency: 5026 ms
-----
```

- Database after a Test:

DynamoDB > Explore items > album

album

Autopreview View table details

► Scan or query items
Expand to query or scan items.

This table has more items to retrieve. To retrieve the next page of items, choose **Retrieve next page**.

Items returned (100)

	albumID (String)	imageSize	profile
<input type="checkbox"/>	c8a42834-51f2-4087...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	5b3203ce-6235-4a8b...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	19c37342-03e8-4526...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	53ac25a7-ec70-46a7...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	d74a940d-fe2e-4772...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	a324c1fb-d437-45dc...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	639dc8c9-b87d-4361...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	2c35f1a6-3994-4f1a...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}
<input type="checkbox"/>	74178c9a-a00f-4f03...	3475	{"artist": "Rock Pistols", "title": "Never Mind The Bollocks!", "year": "1977"}

- ALB/ELB Setup:

modServerLB

▼ Details

Load balancer type	Status	VPC	IP address type
Application	Active	vpc-07a74b455c75d79ef	IPv4
Scheme	Hosted zone	Availability Zones	Date created
Internet-facing	Z1H1FLSHABSF5	subnet-011f4b0bbf131966e us-west-2a (usw2-az2)	November 5, 2023, 11:32 (UTC-08:00)
		subnet-030e7736a5f2d8c35 us-west-2d (usw2-az4)	
		subnet-0b2322553317f8270 us-west-2c (usw2-az3)	
		subnet-0a565f7e9073e1a66 us-west-2b (usw2-az1)	
Load balancer ARN	DNS name		
arn:aws:elasticloadbalancing:us-west-2:122604473639:loadbalancer/app/modServerLB/566ac1e2d82da12f	modServerLB-899373939.us-west-2.elb.amazonaws.com (A Record)		

Listeners and rules

Network mapping

Security

Monitoring

Integrations

Attributes

Tags

Listeners and rules (1) [Info](#)

Manage rules

Manage listener

Add listener

Filter listeners

< 1 > ⚙

<input type="checkbox"/>	Protocol:Port	Default action	Rules	ARN	Security policy	Default SSL/TLS certificate	Tags
<input type="checkbox"/>	HTTP:80	<div>Forward to target group<ul style="list-style-type: none">modServerGroup 1 (100%)Group-level stickiness: Off</div>	1 rule	ARN	Not applicable	Not applicable	0 tags