

# PARCIAL 1

# DESARROLLO

# DE SOFTWARE

API REST - MAGNETO

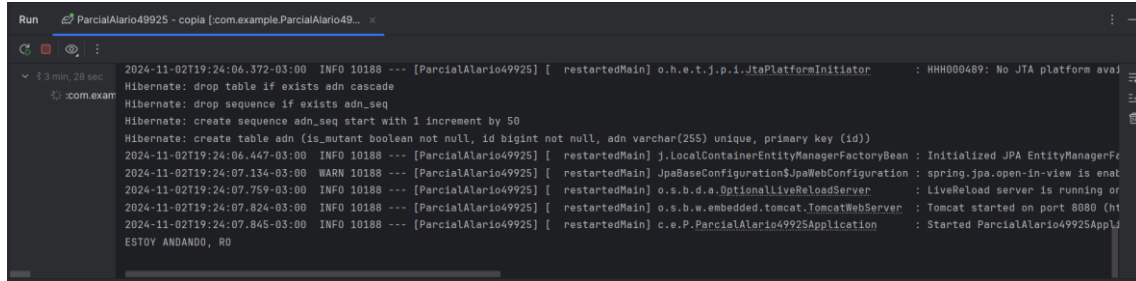
Rocio Alario

Legajo: 49925

2024

### CONFIGURACION GENERAL:

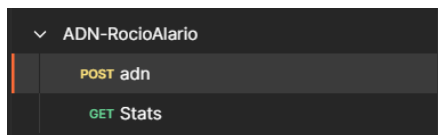
1. Primero debe abrir la carpeta en IntelliJ
2. Luego debe apretar “ParcialAlario49925Application.java” y ejecutar el proyecto
3. Una vez lo ejecute le debe aparecer un cartel que diga “Estoy Andando” como en la siguiente imagen:



The screenshot shows the Run console of IntelliJ IDEA. The logs indicate that the application has started successfully. Key messages include: "Hibernate: drop table if exists adn cascade", "Hibernate: drop sequence if exists adn\_seq", "Hibernate: create sequence adn\_seq start with 1 increment by 50", "Hibernate: create table adn (is\_mutant boolean not null, id bigint not null, adn varchar(255) unique, primary key (id))", "WARN 10188 --- [ParcialAlario49925] [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'", "INFO 10188 --- [ParcialAlario49925] [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 34567", "INFO 10188 --- [ParcialAlario49925] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http)", and "ESTOY ANDANDO, RD".

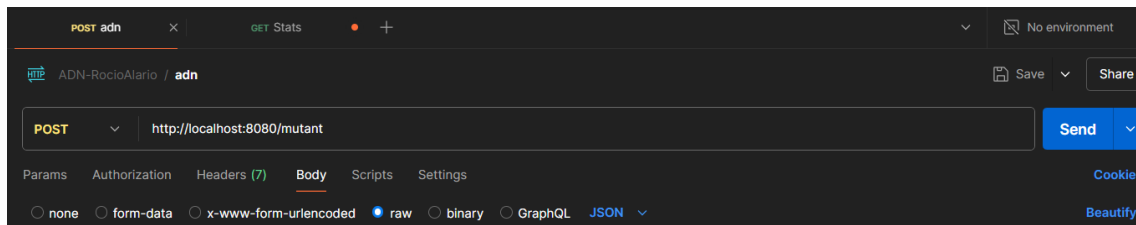
### CONFIGURACION POSTMAN:

4. Después de esto, deberá abrir Postman
5. Luego, dentro de Postman debe tener dos instrucciones: POST y GET



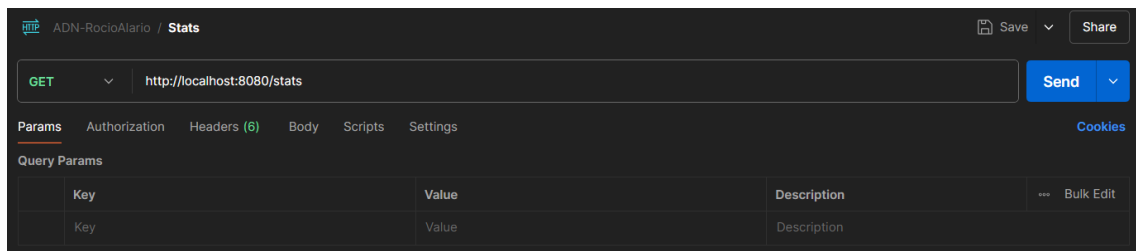
6. La instrucción POST sirve para cargar ADNs a la base de datos H2
7. Para configurar la opción de POST primero debe insertar la siguiente URL:

<http://localhost:8080/mutant>



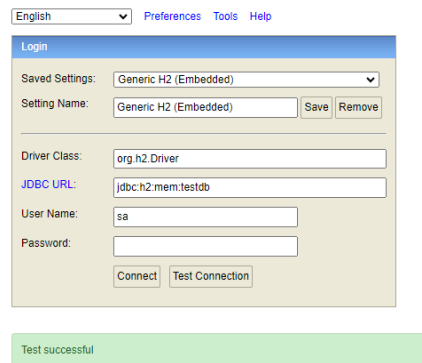
8. La instrucción GET sirve para obtener las estadísticas de cuantos ADNs mutantes y humanos hay cargados en la base de datos H2.
9. Para configurar la opción GET primero debe insertar la siguiente URL:

<http://localhost:8080/stats>

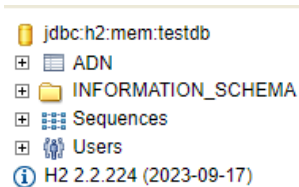


## CONFIGURACION H2:

10. Luego de esto, pasaremos a conectar la base de datos H2:
11. Primero debe insertar la siguiente URL en su navegador (yo usé brave):  
<http://localhost:8080/h2-console/>
12. Una vez dentro de H2, deberá completar con los siguientes datos:
  - Driver Class: org.h2.Driver
  - JDBC URL: jdbc:h2:mem:testdb
  - User Name: sa
  - Password: (nada)
13. Luego aprete en Test Conection y debería aparecerle Test succesful



14. Una vez dentro, le va a aparecer una tabla que se llama ADN



## ¿COMO USAR LA API?

15. Hay que tener en cuenta que debe estar ejecutándose  
“ParcialAlario49925Application.java” para que funcione
16. Una vez tengamos todo configurado, podemos comenzar a usarla:
17. Primero vamos a postman y vamos a usar la instrucción POST:
18. Hay que posicionarse en la parte que dice “Body” debemos tener la opción de “raw” marcada y asegurarnos de estar trabajando con JSON.
19. Vamos a poder enviarle un array de Strings con un ADN y nos dirá si es mutante o no. Si tiene mas de una secuencia de 4 de las letras permitidas iguales, nos dirá que es mutante y nos mostrará 200OK. Caso contrario nos dirá que no es mutante y nos mostrara 403 Forbidden.
20. Una consideración a tener en cuenta es que el ADN debe empezar con la palabra “adn”, si no lo verificará y lo tomará como válido.
21. A continuación dejo algunos ejemplos que puedes usar:

**Mutante1:**

```
{  
  "adn": [  
    "AAAATG",  
    "TAATGT",  
    "GCATCC",  
    "TTGATG",  
    "GTAGTC",  
    "AGTCAA"  
  ]  
}
```

```
"AAAATG",  
"TAATGT",  
"GCATCC",  
"TTGATG",  
"GTAGTC",  
"AGTCAA"
```

**Mutante2:**

```
{  
  "adn": [  
    "AAAAA",  
    "CAGTG",  
    "TTATG",  
    "AGAAG",  
    "CCCCT"  
  ]  
}
```

```
"AAAAA",  
"CAGTG",  
"TTATG",  
"AGAAG",  
"CCCCT"
```

**Mutante3:**

```
{  
  "adn": [  
    "AAAAAAA",  
    "CAGTGCC",  
    "TTATGAA",  
    "AGAAGTT",  
    "CCCCTAA",  
    "TCACTGG",  
    "AGTCAAA"  
  ]  
}
```

```
"AAAAAAA",  
"CAGTGCC",  
"TTATGAA",  
"AGAAGTT",  
"CCCCTAA",  
"TCACTGG",  
"AGTCAAA"
```

#### Mutante4:

```
{
  "adn": [
    "AAAAAT",
    "AAGTGC",
    "ATATGT",
    "AGAAGG",
    "CCCCTA",
    "TCACTG"
  ]
}
```

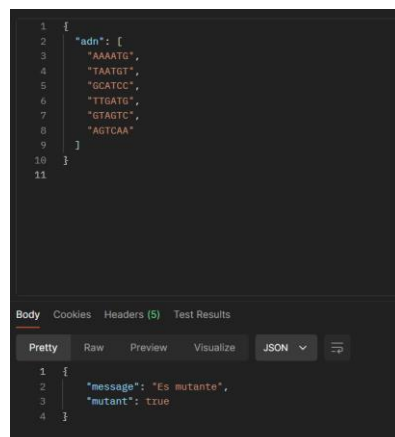
```
"ATAAT",
"AAGTGC",
"ATATGT",
"AGAAGG",
"CCCCTA",
"TCACTG"
```

#### Mutante5:

```
{
  "adn": [
    "ATGCCGA",
    "CAGTGC",
    "TTATGT",
    "AGAAGG",
    "CCCCTA",
    "TCACTG"
  ]
}
```

```
"ATGCCGA",
"CAGTGC",
"TTATGT",
"AGAAGG",
"CCCCTA",
"TCACTG"
```

A continuación, un ejemplo de cómo nos muestra si es mutante:



The screenshot shows a web application interface with a dark theme. At the top, there is a text area containing a JSON object with an 'adn' array of six DNA sequences. Below the text area, there are tabs for 'Body', 'Cookies', 'Headers (5)', and 'Test Results'. The 'Body' tab is selected, and it shows a 'Pretty' JSON view of the response. The response is a JSON object with 'message' and 'mutant' fields.

```
1 {
2   "adn": [
3     "AAAAAT",
4     "TAATGT",
5     "GCATCC",
6     "TTGATG",
7     "GTAGTC",
8     "AGTCAA"
9   ]
10 }
11
```

```
1 {
2   "message": "Es mutante",
3   "mutant": true
4 }
```

**NoMutante1:**

```
{  
  "adn": [  
    "ATGCG",  
    "CTGTG",  
    "TTATG",  
    "AGAAG",  
    "CGCCT"  
  ]  
}
```

**NoMutante2:**

```
{  
  "adn": [  
    "ATGCGA",  
    "CAGTAC",  
    "TTATGT",  
    "AGAAGG",  
    "CCTCTA",  
    "TCACTG"  
  ]  
}
```

**NoMutante3:**

```
{  
  "adn": [  
    "AATATG",  
    "TAATGT",  
    "GCATCC",  
    "TTGATG",  
    "GTAGTC",  
    "AGTCAA"  
  ]  
}
```

**NoMutante4:**

```
{  
  "adn": [  
    "ATGC",  
    "CAGT",  
    "TTAT",  
    "AGAA"  
  ]  
}
```

**NoMutante5:**

```
{  
  "adn": [  
    "AAATTAA",  
    "CAGAGCC",  
    "TTATGAA",  
    "AGTAGTT",  
    "CCACTAA",  
    "TCACTGG",  
    "AGTCATA"  
  ]  
}
```

**Invalido1: ADN NULO**

```
{  
  "adn": []  
}
```

**Invalido2: matriz NxM**

```
{  
  "adn": [  
    "ATGCGA",  
    "CAGTGC",  
    "TTATGT",  
    "AGAAGG",  
    "CCCCTA"  
  ]  
}
```

**Invalido3: números entre comillas**

```
{  
  "adn": [  
    "123456",  
    "654321",  
    "112233",  
    "332211",  
    "445566",  
    "665544"  
  ]  
}
```

**Invalido4: recibir null**

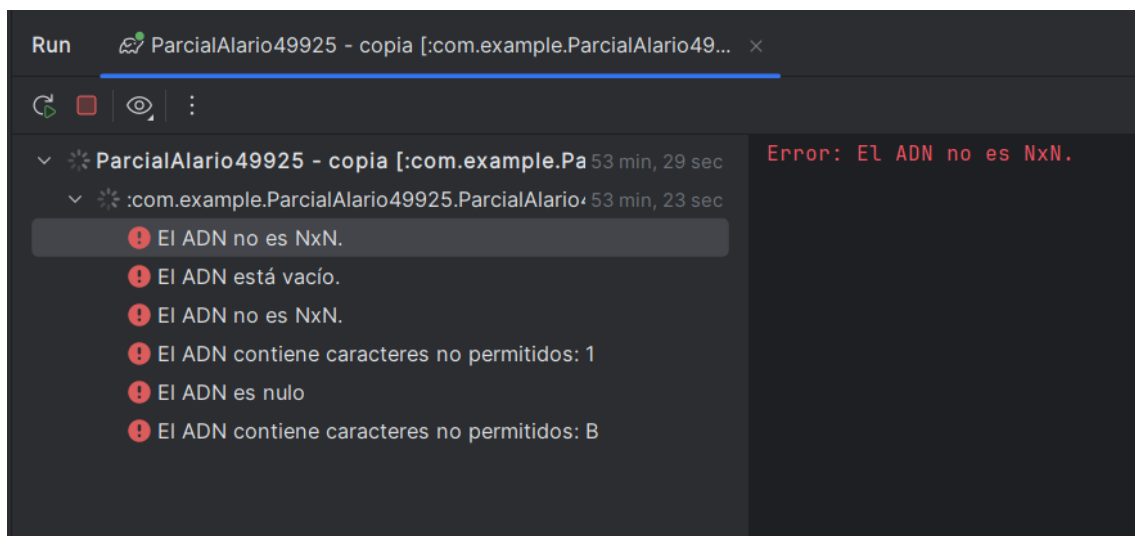
```
{  
  "adn": null  
}
```



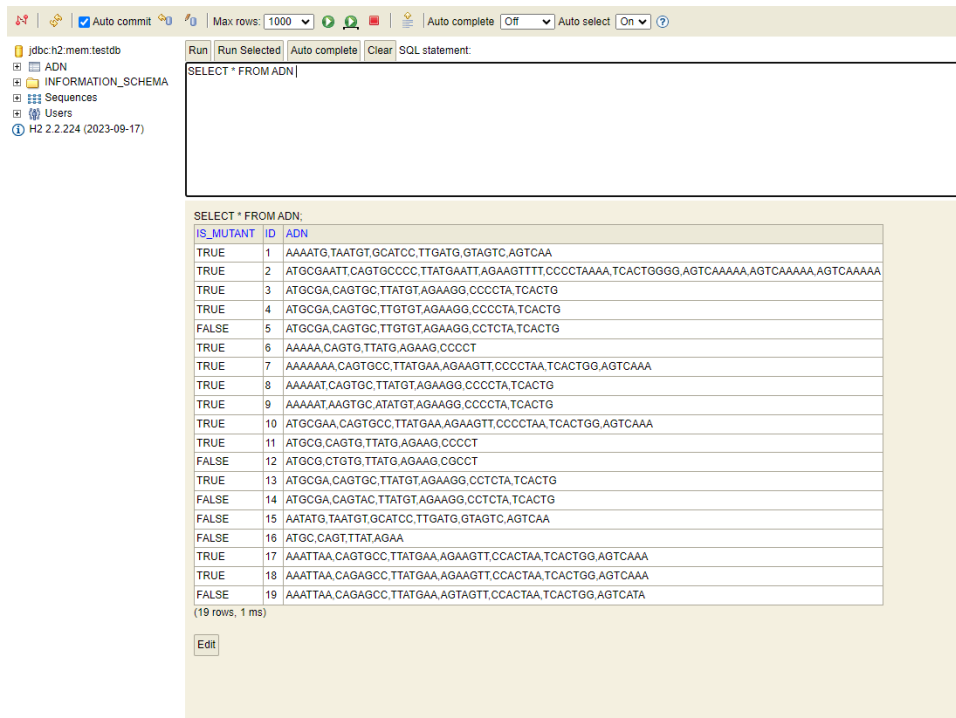
### Invalido5: caracteres inválidos

```
{  
  "adn": [  
    "ATGCCGA",  
    "CAGTGC",  
    "TTATGT",  
    "AGAAGG",  
    "CCCCTA",  
    "TCACTB"  
  ]  
}
```

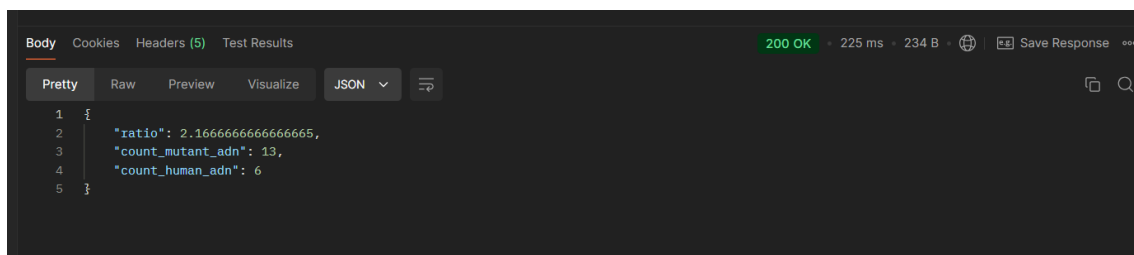
22. Cabe aclarar que en el caso de que sean inválidos, nos mostrará “400 Bad request” en postman y podremos ir a la consola de IntelliJ para ver por qué no es válida.



23. A medida que vayamos enviando los ADN, se nos irán cargando en la base de datos H2, en donde podremos ver el ADN enviado, si es mutante o no (TRUE→ mutante, FALSE→ humano) y su id, y se mostrará de la siguiente forma:



24. Para mostrarlos todos debemos actualizar con las flechas de arriba, escribir `SELECT * FROM ADN` y apretar “run” y nos mostrará todos los ADN.
25. Ahora vamos con el GET stats para saber cuantos de los que metimos eran mutantes y cuantos humanos:
26. Primero vamos a la sección de GET, y apretamos “send”
27. Luego, como vemos en la imagen, nos dirá cuantos de los que cargamos fueron humano, cuantos fueron mutantes y el ratio de estos:

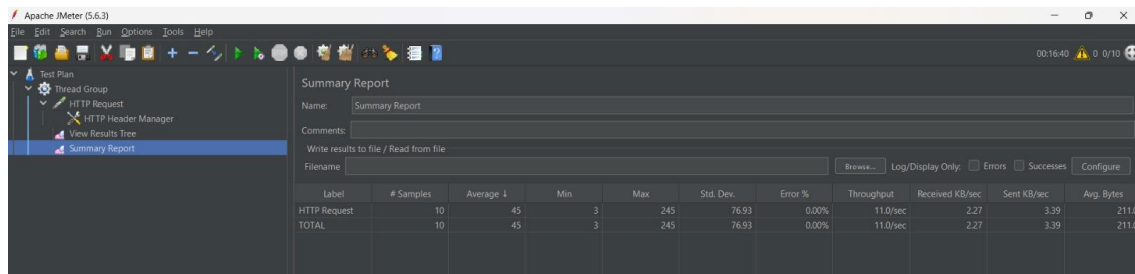


## FLUCTUACIONES:

Para calcular las fluctuaciones de tráfico que puede recibir la API use Jmeter:

Se probó con diferentes números:

## 10 peticiones por segundo:



Summary Report

Name: Summary Report

Comments:

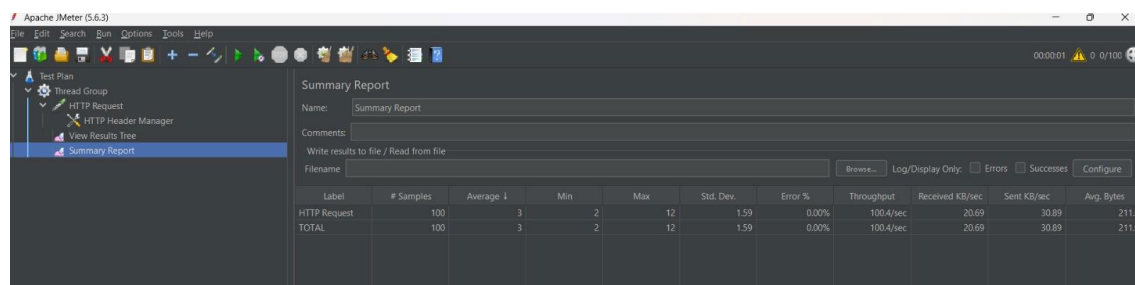
Write results to file / Read from file

Filename:  Browse... Log/Display Only: ☐ Errors ☐ Successes ☐ Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10	45	3	245	76.93	0.00%	11.0/sec	2.27	3.39	211.0
TOTAL	10	45	3	245	76.93	0.00%	11.0/sec	2.27	3.39	211.0

Como se puede ver, con 10 peticiones por segundo se demoró un tiempo promedio de 45 milisegundos y dio un error del 0%

## 100 peticiones por segundo:



Summary Report

Name: Summary Report

Comments:

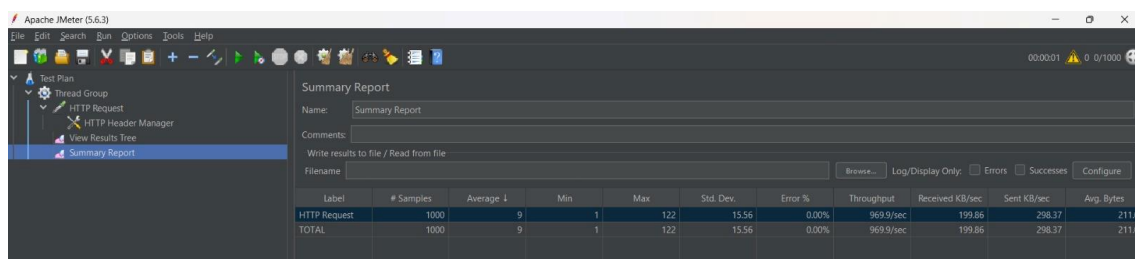
Write results to file / Read from file

Filename:  Browse... Log/Display Only: ☐ Errors ☐ Successes ☐ Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	100	3	2	12	1.59	0.00%	100.4/sec	20.69	30.89	211.0
TOTAL	100	3	2	12	1.59	0.00%	100.4/sec	20.69	30.89	211.0

Como se puede ver, con 100 peticiones por segundo se demoró un tiempo promedio de 3 milisegundos y dio un error del 0%

## 1000 peticiones por segundo:



Summary Report

Name: Summary Report

Comments:

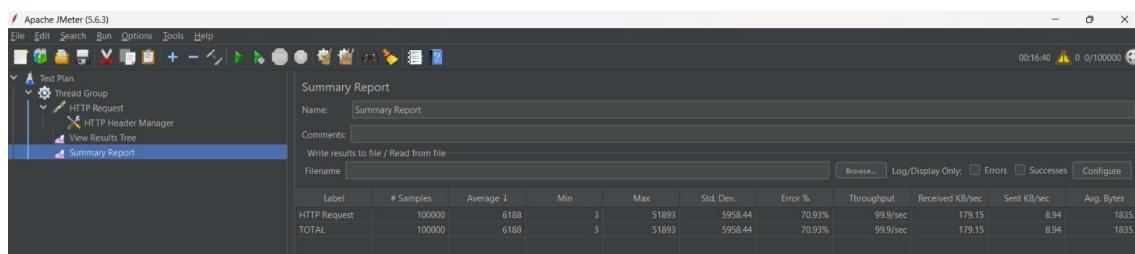
Write results to file / Read from file

Filename:  Browse... Log/Display Only: ☐ Errors ☐ Successes ☐ Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	9	1	122	15.56	0.00%	969.9/sec	199.86	298.37	211.0
TOTAL	1000	9	1	122	15.56	0.00%	969.9/sec	199.86	298.37	211.0

Como se puede ver, con 1000 peticiones por segundo se demoró un tiempo promedio de 9 milisegundos y dio un error del 0%

## 100.000 peticiones por segundo:



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename:  Browse... Log/Display Only: ☐ Errors ☐ Successes ☐ Configure

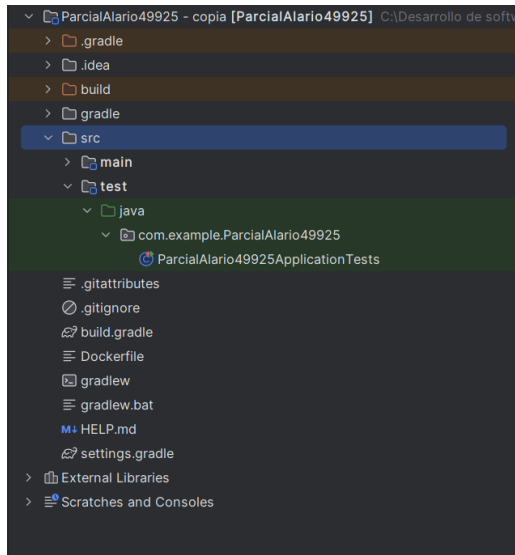
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	100000	6188	3	51893	5958.44	70.93%	99.9/sec	179.15	8.94	1835.6
TOTAL	100000	6188	3	51893	5958.44	70.93%	99.9/sec	179.15	8.94	1835.6

Como se puede ver, con 100k peticiones por segundo se demoró un tiempo total de 16 minutos con 40 segundos y dio un error del 70.93%. Esto sugiere que para un flujo tan grande de peticiones por segundo, no es tan eficiente el algoritmo y podría mejorarlo.

## TEST AUTOMATICOS:

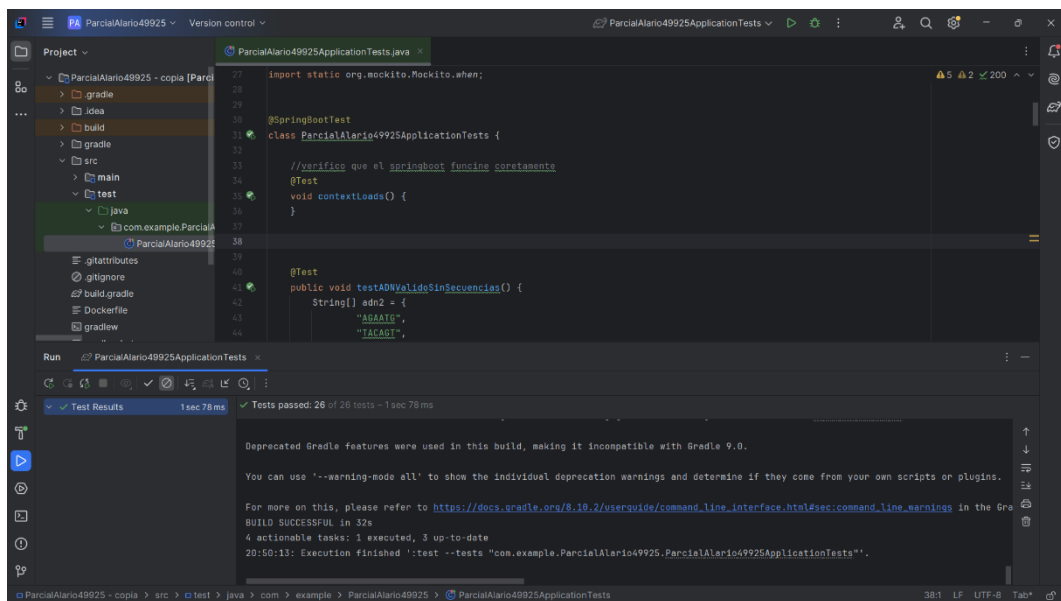
Se probaron un total de 26 test unitarios con Junit para corroborar el funcionamiento de la api.

Para probarlos, primero debe ir a la carpeta de test, ahí podrá visualizar todas las pruebas que se hicieron.



Luego deberá correr “ParcialAlario49925ApplicationTests”

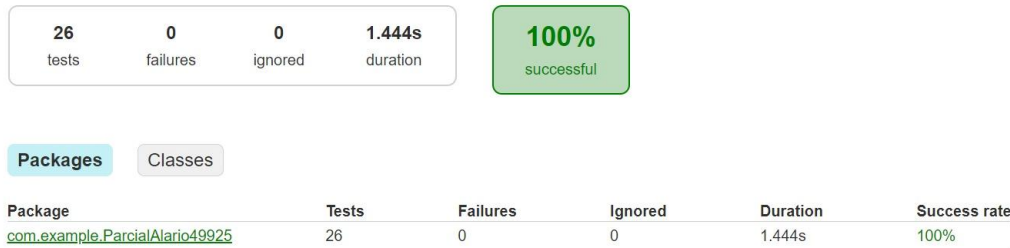
Una vez termine la ejecución, mostrará la cantidad de test que se ejecutaron (26) y cuantos fallaron. En este caso todos pasaron las pruebas. También le saldrá una notificación como en la imagen 2



## CODE COVERAGE:

Finalmente, se calculó el total de cobertura que tiene el código con las pruebas unitarias realizadas y se llegó a la siguiente conclusión:

### Test Summary



Se hicieron un total de 26 test, la prueba total duró 1.44 segundos y se tuvo una tasa de éxito del 100%, es decir, no falló ninguna prueba.

## Cobertura total:

### ParcialAlario49925

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.ParcialAlario49925.Services	<div><div></div></div>	81 %	<div><div></div></div>	85 %	5	32	17	103	2	11	0	2
com.example.ParcialAlario49925.Validators	<div><div></div></div>	88 %	<div><div></div></div>	95 %	4	15	5	24	3	4	0	1
com.example.ParcialAlario49925	<div><div></div></div>	27 %	<div><div></div></div>	n/a	1	2	3	4	1	2	0	1
com.example.ParcialAlario49925.Controllers	<div><div></div></div>	81 %	<div><div></div></div>	50 %	2	5	2	12	1	4	0	2
com.example.ParcialAlario49925.dto	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	3	0	5	0	2	0	2
Total	96 of 535	82 %	8 of 68	88 %	12	57	27	148	7	23	0	8

Finalmente, se muestra detalladamente la cobertura total de las pruebas, lo mas importante que podemos concluir de acá es que el código tiene una cobertura de código total del **82%** para instrucciones y una cobertura total del **88%** para las validaciones.

## ARQUITECTURA DE SOFTWARE:

