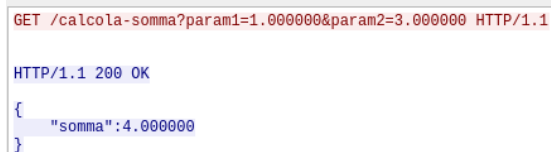


## Esercizi Web Services - Pagine 63, 64, 65 e 68

### Esercizio 1 (pagina 63)

1. Alla linea di comando vanno passati `tipofunzione op1 op2`.



```
GET /calcola-somma?param1=1.000000&param2=3.000000 HTTP/1.1  
  
HTTP/1.1 200 OK  
{  
  "somma":4.000000  
}
```

Figure 1: Scambio HTTP GET verso `/calcola-somma`

2. Si può notare che è una normale richiesta HTTP GET verso l'API `/calcola-somma`, mentre la risposta è in formato JSON.
3. La signature di `calcoloSomma` è:

```
float calcoloSomma(float, float)
```

Averle uguali serve a mantenere lo stesso tipo e ordine dei parametri, evitando errori di interpretazione.

### Esercizio 2 (pagina 64)

È un'opportunità, poiché:

- Lo strato HTTP fa da “collante”, disaccoppiando completamente i binari: il contratto è solo sull'URL, sui parametri e sul payload.

- Libertà evolutiva: puoi riscrivere il server in Rust domani o aggiungere un client mobile in Kotlin senza toccare il resto.

## Esercizio 3 (pagina 65)



Figure 2: Tempo di calcolo dei numeri primi

- 1.
2. No, non è stato necessario tradurre l'algoritmo dei numeri primi in Java, perché tutta la parte algoritmica e di calcolo è svolta interamente dal processo server. L'unica aggiunta in Java è la funzione che invia la richiesta.

## Esercizio Finale (pagina 68)

1. Basterebbe passare tre indirizzi diversi (che puntano a macchine con lo stesso servizio).
2. No, purché ciascuna macchina esponga lo stesso servizio REST sulla porta 8000.
3. Dividi lo spazio in N spezzoni uguali, ad esempio:

$$serverA \rightarrow [1..333333], \quad serverB \rightarrow [333334..666666], \quad serverC \rightarrow [666667..1000000].$$

4. No, la parte logica del server fa già quello che deve fare: basta lanciare più istanze (su host o porte diverse).