Intro to Backend Development

Lecture 2 · Databases

Aayush Agnihotri
Tony Matchev

Announcements

Assignments

- Demos are **very** helpful for understanding/completing the assignment! Take advantage of them and rewatch them as necessary
- Use the test cases file
- Save your Postman requests/collections
- Write documentation
- Follow API specification EXACTLY

Miscellaneous

- Please fill out the weekly feedback form. They’re worth 5% of your final grade.

Review

Client-Server Model

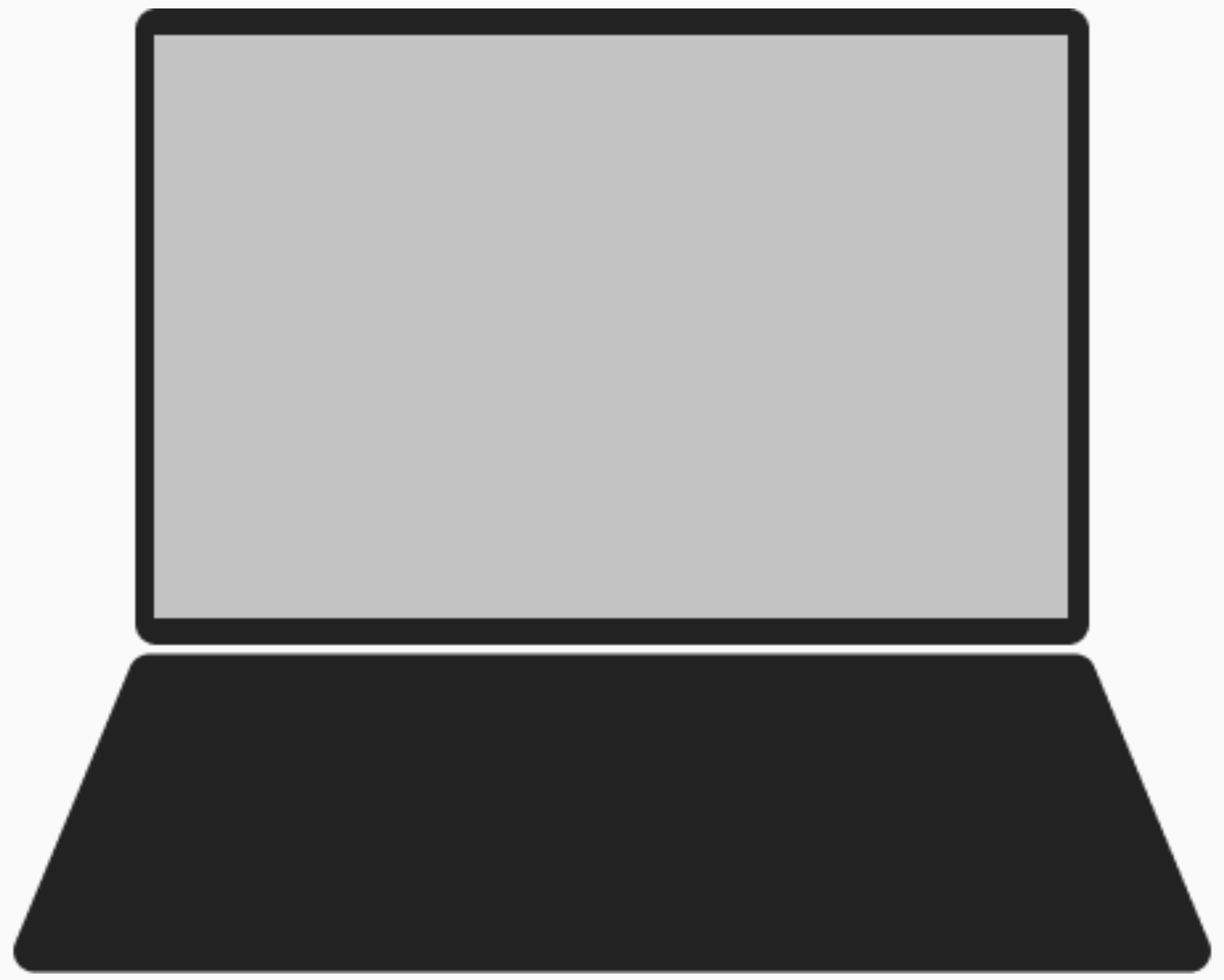
- Clients are computers (phones, laptops, tablets, etc.)
- Servers are also computers (think like a desktop with no screen)
- Clients send **requests** to servers
- Servers reply with **responses**

Requests

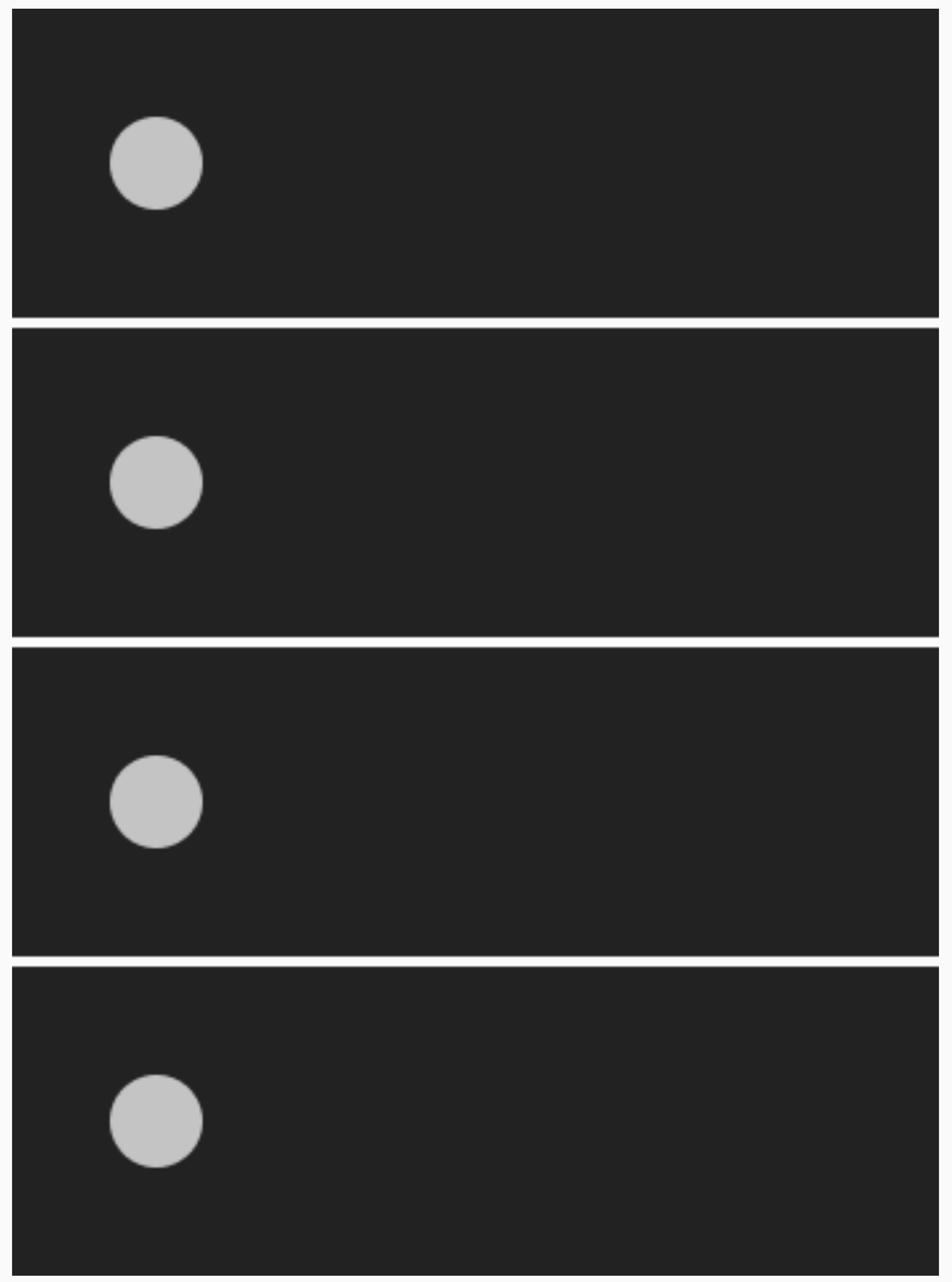
- Initiated by **client**
- Sent to a specific **URL** (i.e. *www.google.com/search*)
- URL contains domain, route **path**, and URL parameters (optional)
  - **Method** indicates nature of request (i.e. CRUD)
- Meaningful data sent in **body** of request (we use JSON)

Responses

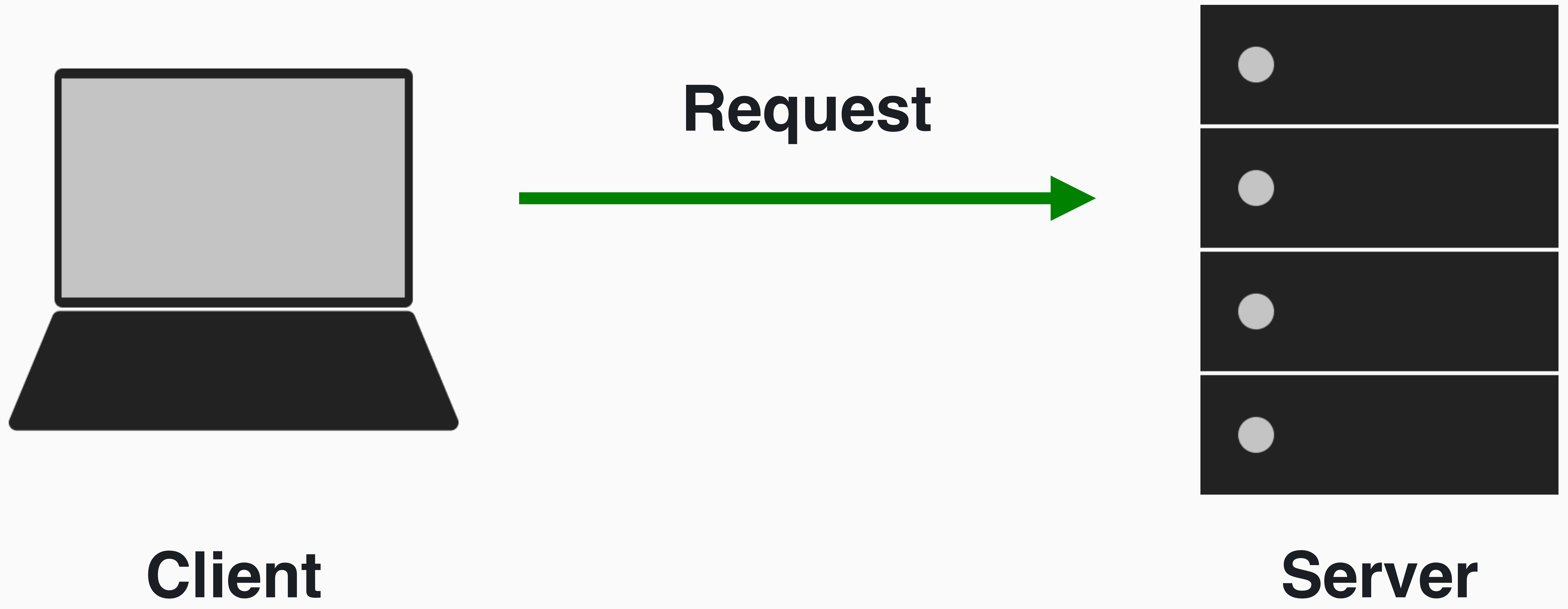
- Triggered by client **request**
- Returns confirmation of requested operation
- **Response codes** used to indicate success at abstract level
- Sends JSON data in a **body** just like a request

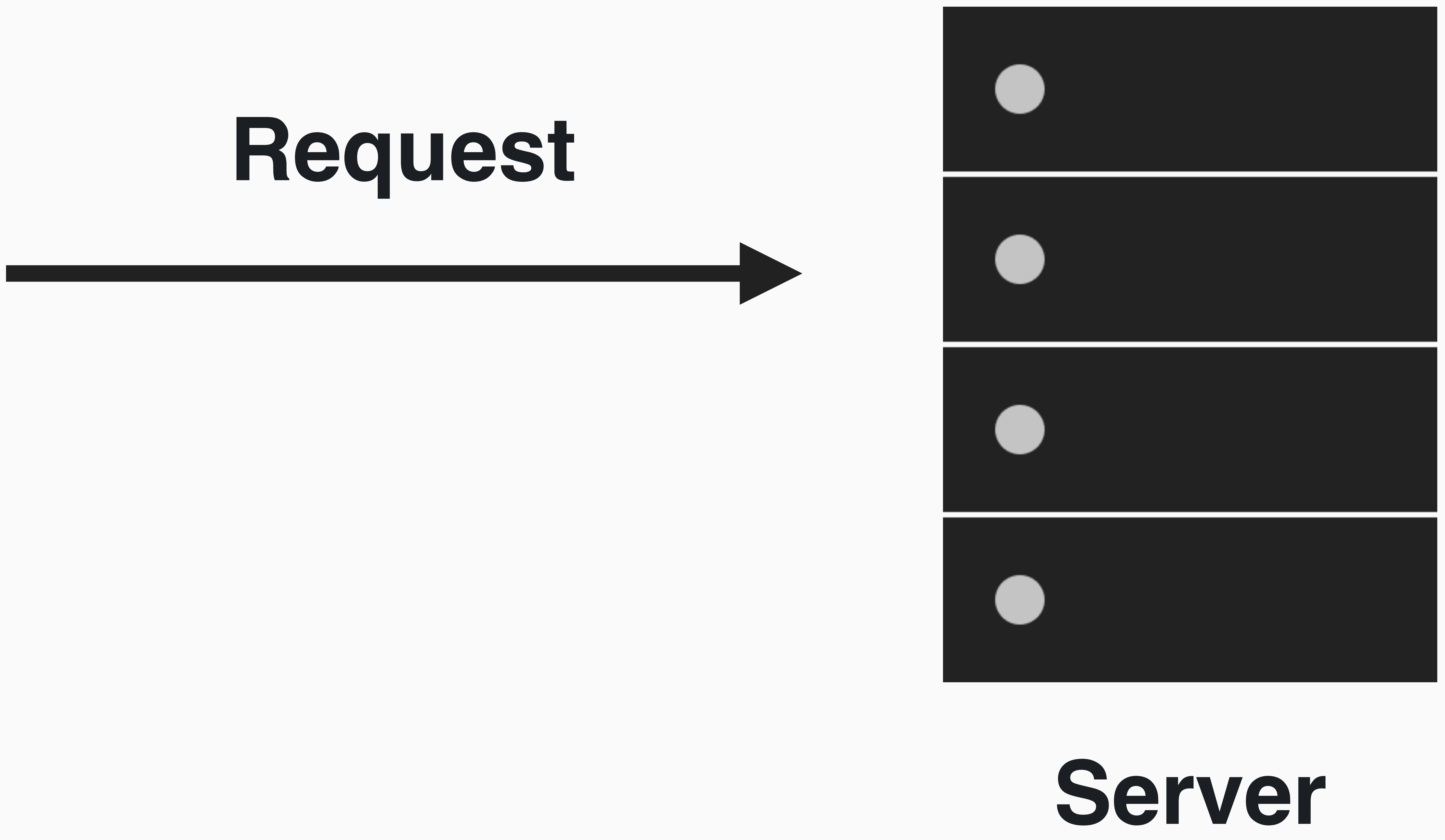


Client

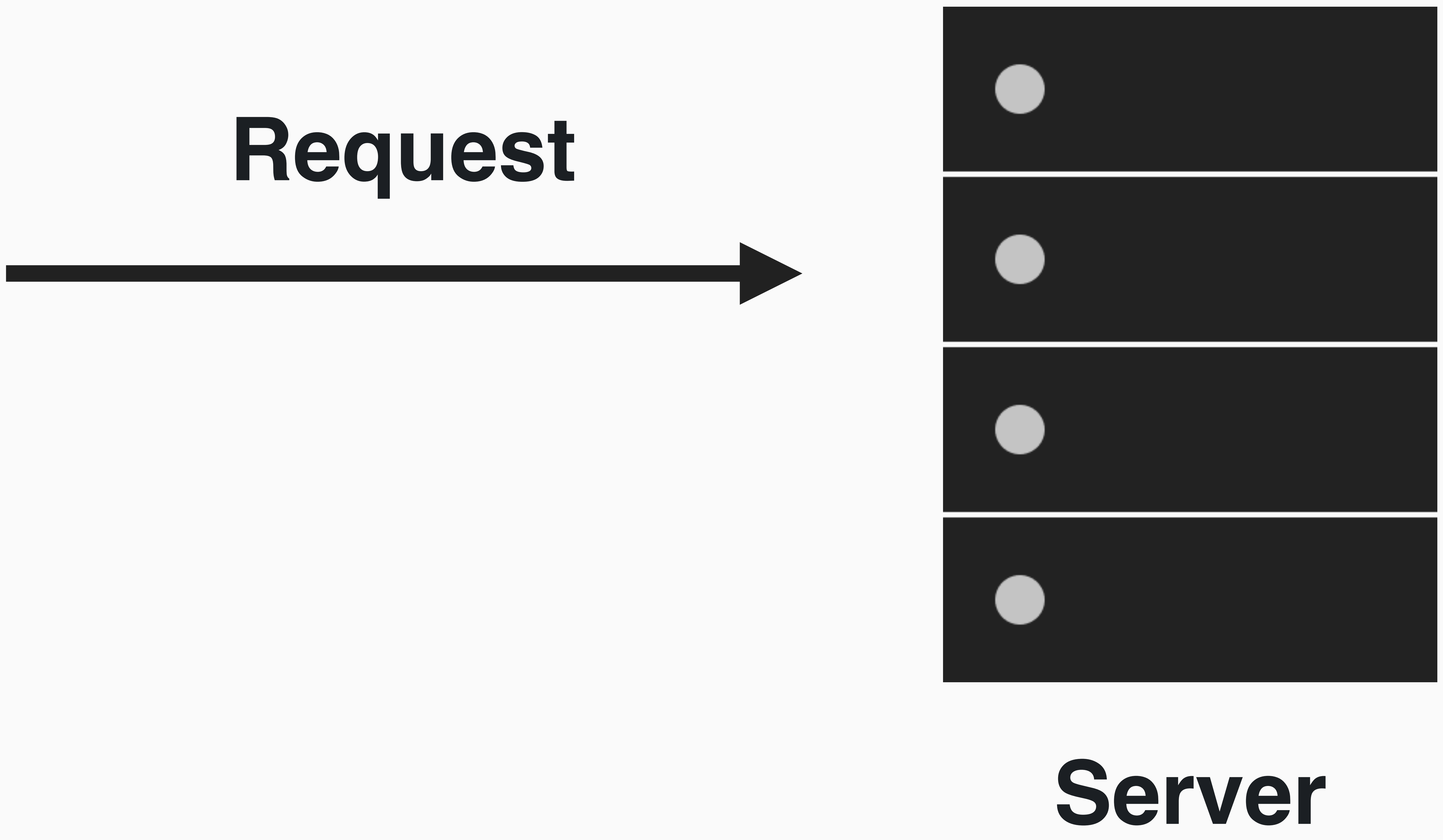


Server


Request
Client
Server



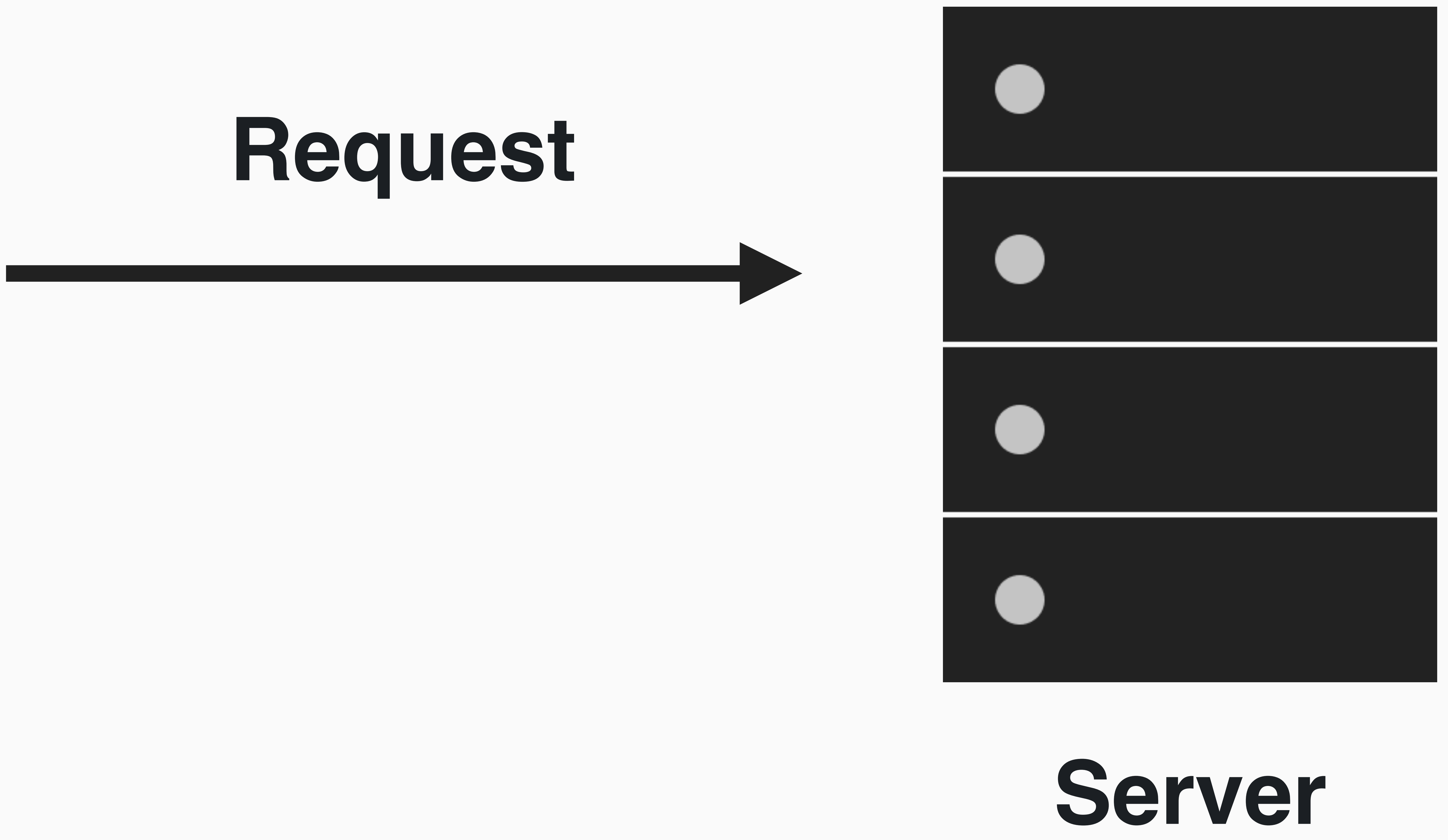
→ Route triggered
Data operation
Return response



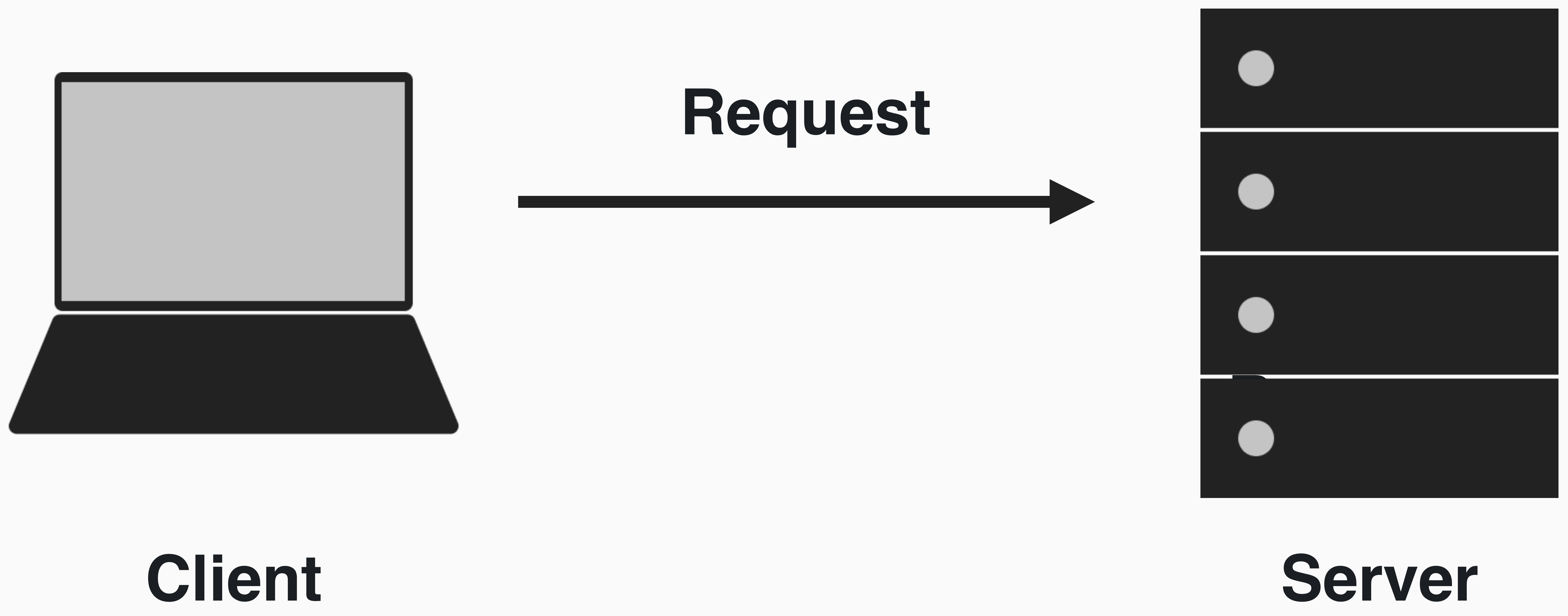
Route triggered

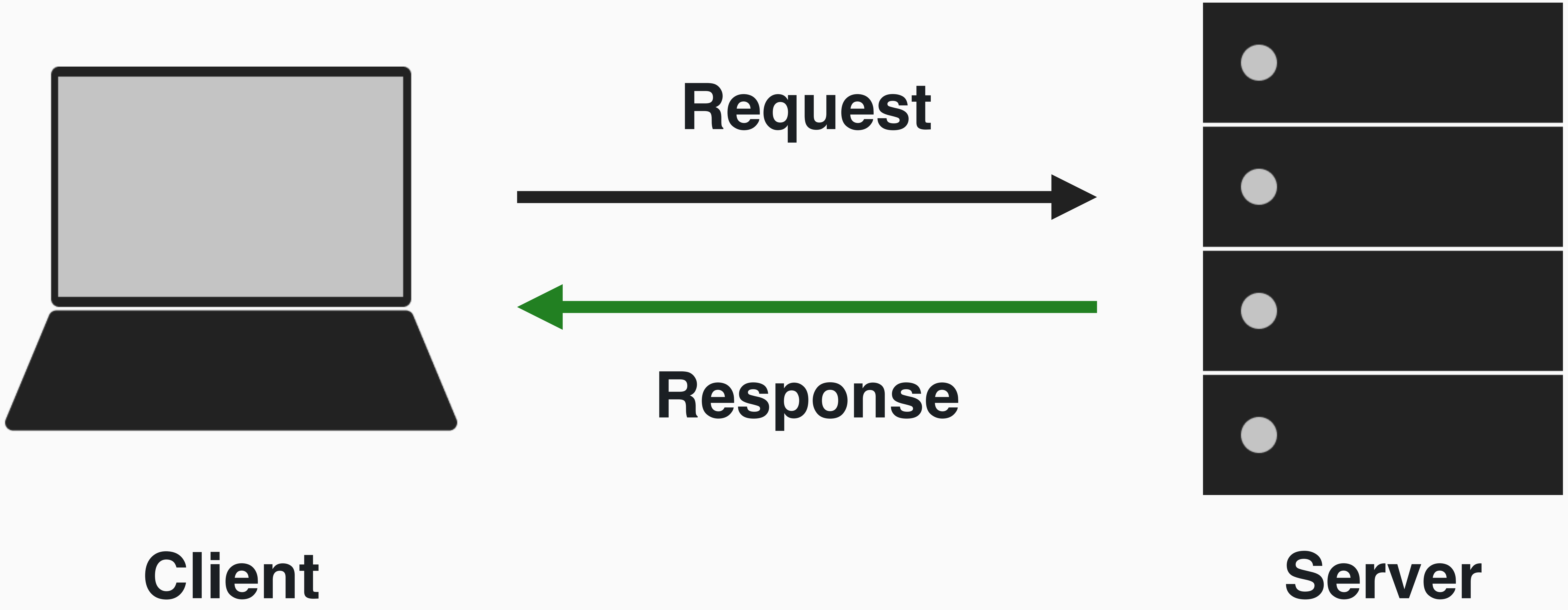
→ Data operation

Return response

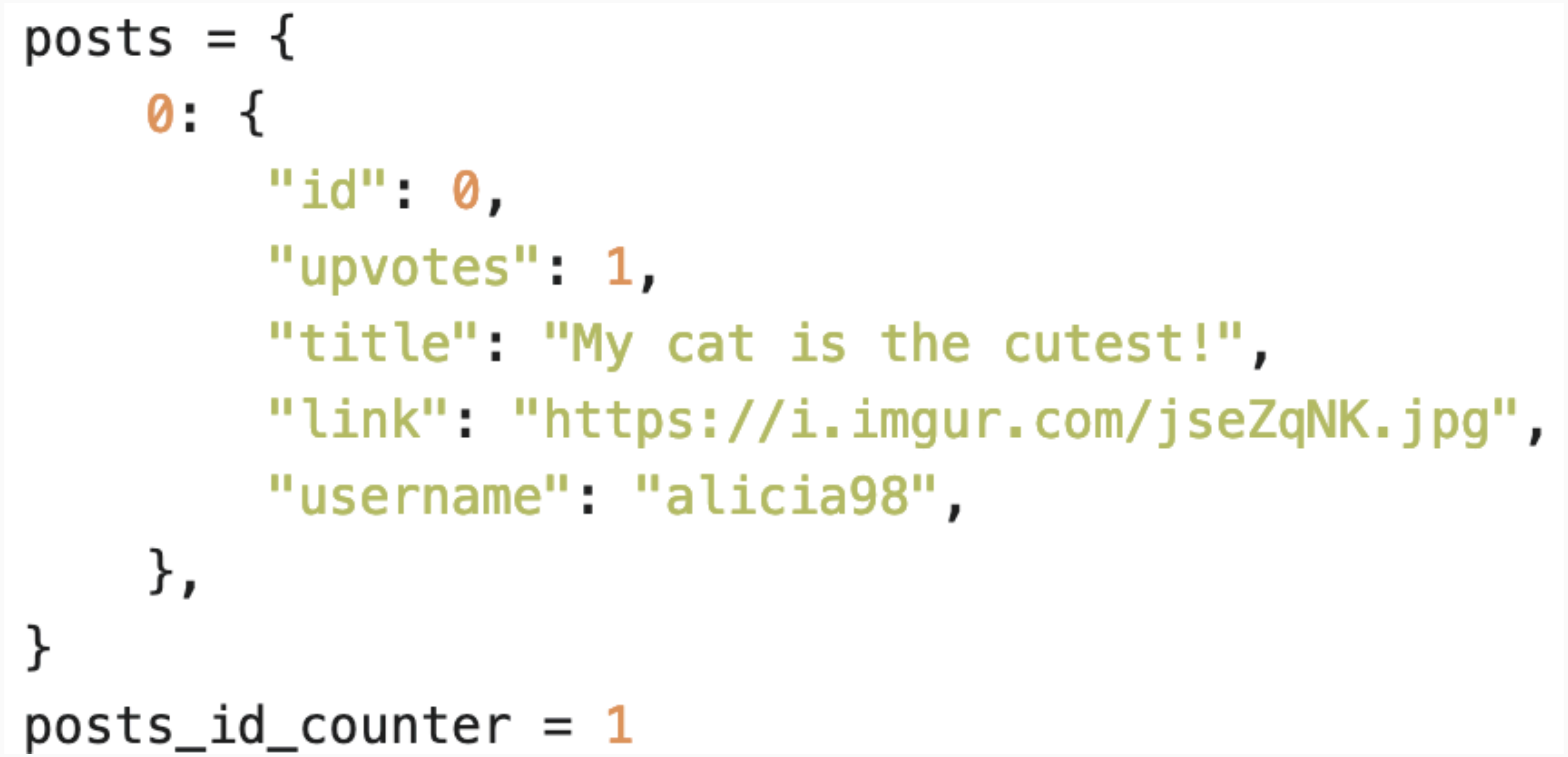


Route triggered
Data operation
→ Return response


Request
Client
Server


Request
Response
Client
Server

Data Persistence



```
posts = {
    0: {
        "id": 0,
        "upvotes": 1,
        "title": "My cat is the cutest!",
        "link": "https://i.imgur.com/jseZqNK.jpg",
        "username": "alicia98",
    },
}
posts_id_counter = 1
```

Deficiencies

- Stopping the server = Data lost
- Does not scale efficiently
- No formatting enforcement

Databases

Fundamentals

- Database = collection of organized information
- Easily access, update, and manage data
- Implemented as a collection of tables

id	name	age	hobby
1	“aayush”	19	“music”
2	“tony”	19	“running”
3	“kidus”	21	“gaming”

Columns have a descriptive **name** and specific **data type**

Rows contain a set of column values and each **represents one item**

id	name	age	hobby
1	“aayush”	19	“music”
2	“tony”	19	“running”
3	“kidus”	21	“gaming”

- Unique identifiers required for every item in table

id	name	age	hobby
1	“aayush”	19	“music”
2	“tony”	19	“running”
3	“kidus”	21	“gaming”

- Unique identifiers required for every item in table
- Id’s automatically increment

id	name	age	hobby
1	“aayush”	19	“music”
2	“tony”	19	“running”
3	“kidus”	21	“gaming”

- Unique identifiers required for every item in table
- Id’s automatically increment

id	name	age	hobby
2	“tony”	19	“running”
3	“kidus”	21	“gaming”
4	“archit”	20	“playing with dogs”

- Unique identifiers required for every item in table
- Id’s automatically increment
- Id’s of removed items do not get reused

Database != Table

Database = Tables

id	name	age	hobby
1	“aayush”	19	“music”
2	“tony”	19	“running”
3	“kidus”	21	“gaming”

Users Table

id	name	description	likes
1	“tony”	“running w my HS friend!”	30
2	“aayush”	“backend is so valid”	80
3	“kidus”	“Don’t be late to subteam meeting!”	60

Posts Table

id	name	age	hobby
1	“aayush”	19	“music”
2	“tony”	19	“running”
3	“kidus”	21	“gaming”

Users Table

id	name	description	likes
1	“tony”	“running w my HS friend!”	30
2	“aayush”	“backend is so valid”	80
3	“kidus”	“Don’t be late to subteam meeting!”	60

Posts Table

id	name	age	hobby
1	“aayush”	19	“music”
2	“tony”	19	“running”
3	“kidus”	21	“gaming”

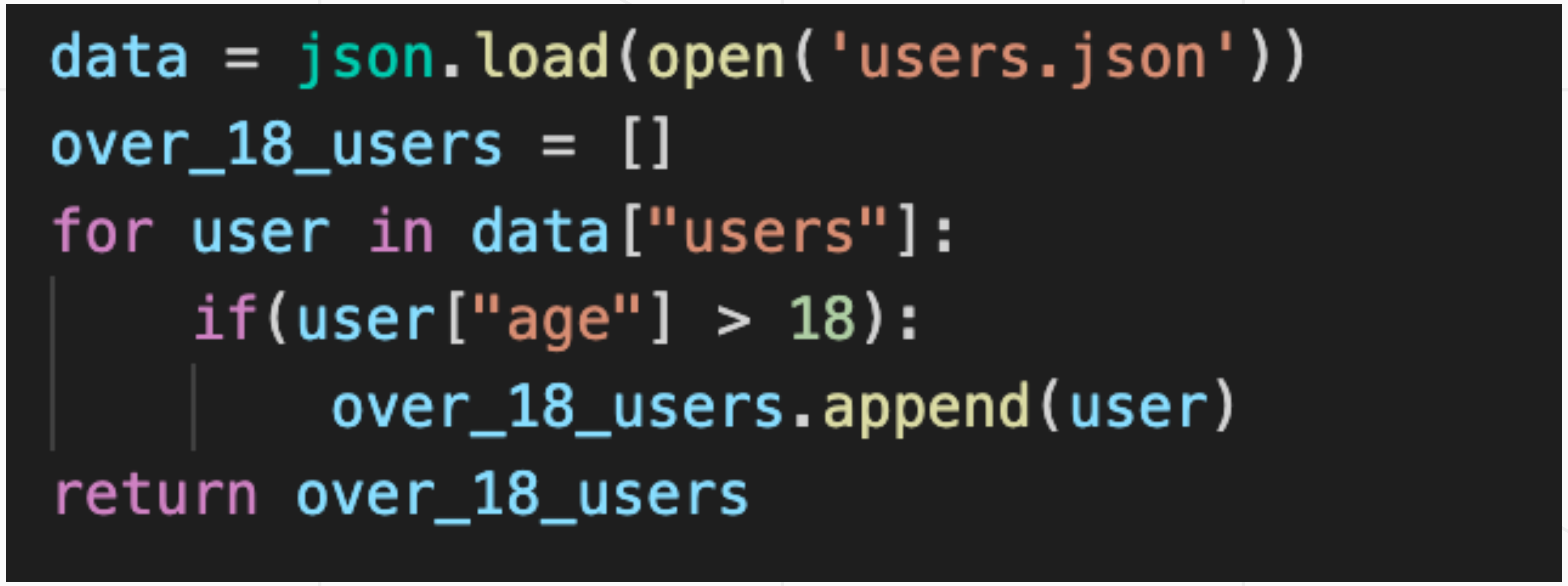
vs.

```
{
  “Users”: [
    {
      “id”: 1,
      “name”: “aayush”,
      “age”: 19,
      “hobby”: “music”
    },
    {
      “id”: 2,
      “name”: “tony”,
      “age”: 19,
      “hobby”: “running”,
    },
    {
      “id”: 3,
      “name”: “kidus”,
      “age”: 21,
      “hobby”: “gaming”,
    }
    ...
  ]
}
```

Database Benefits

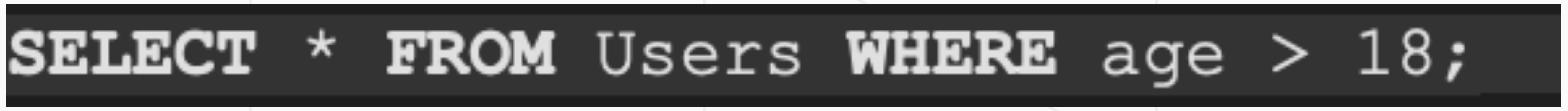
1. All data is structured
2. Scales well
3. Querying data
   - Asking the databases questions
   - Extremely helpful for larger datasets

Which users are over 18?



```python
data = json.load(open('users.json'))
over_18_users = []
for user in data["users"]:
    if(user["age"] > 18):
        over_18_users.append(user)
return over_18_users
```

Which users are over 18?



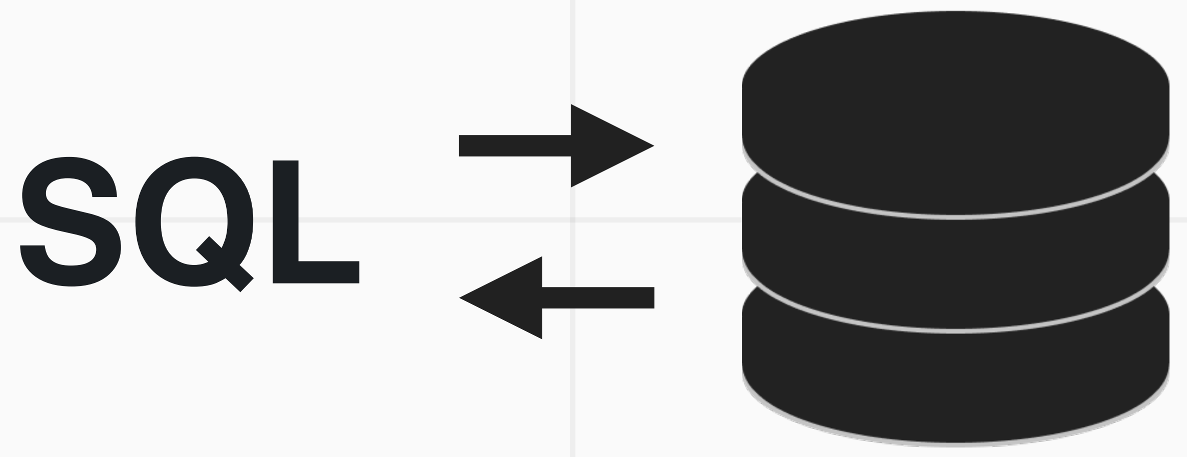
```
SELECT * FROM Users WHERE age > 18;
```

Structured Query Language

SQL

Overview

- Language for communicating with databases
- Executes create, retrieve, update, delete operations



Creating a Table

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  ...
);
```

Creating a Table

```
CREATE TABLE user (
  id    INTEGER PRIMARY KEY AUTOINCREMENT,
  name  TEXT NOT NULL,
  age   TEXT NOT NULL,
  email CHAR(50)
);
```

Inserting Data

```
INSERT INTO table_name (
  (column1, column2, ...)
VALUES (value1, value2, …)
);
```

Inserting Data

This is a bad practice!!

* **Bad for readability**
* **Changes in column order breaks it**

```
INSERT INTO table_name (
  ~~(column1, column2, ...)~~
VALUES (value1, value2, ...)
);
```

* Can omit columns if inserting values into **all** columns

Inserting Data

```
INSERT INTO user (
  (name, age, email)
VALUES (‘John’, 21, ‘js123@cornell.edu’)
);
```

Retrieving Data

```
SELECT column1, column2, ...
FROM   table_name;
```

Retrieving Data

```
SELECT *
FROM   table_name;
```

* = selecting all columns

Retrieving Data

```
SELECT column1, column2, ...
FROM   table_name
WHERE  condition;
```

Retrieving Data

```
SELECT *
FROM   users
WHERE  age >= 18;
```

Updating Data

```
UPDATE table_name
SET    column1=value1, column2=value2,
...
WHERE  condition;
```

Updating Data

```
UPDATE user
SET    email=‘jsmith@gmail.com’
WHERE  id=1;
```

Deleting Data

```
DELETE FROM table_name
WHERE condition;
```

Deleting Data

```
DELETE FROM user
WHERE id=1;
```

Demo