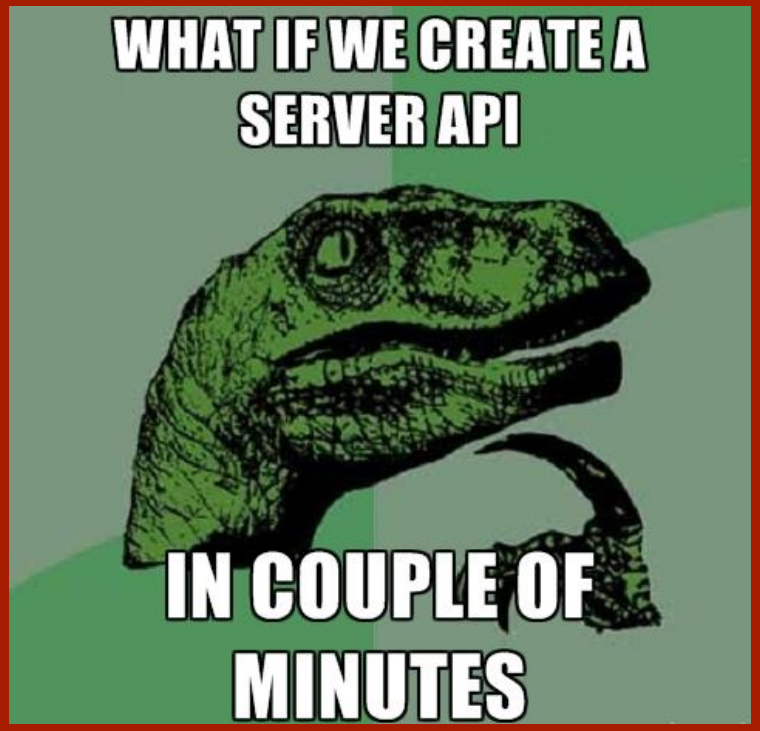
Enabling Technologies for Data Science

Lesson 6 - Server-side APIs



TINKERCADEMY

An Infocomm Club Appointed Vendor

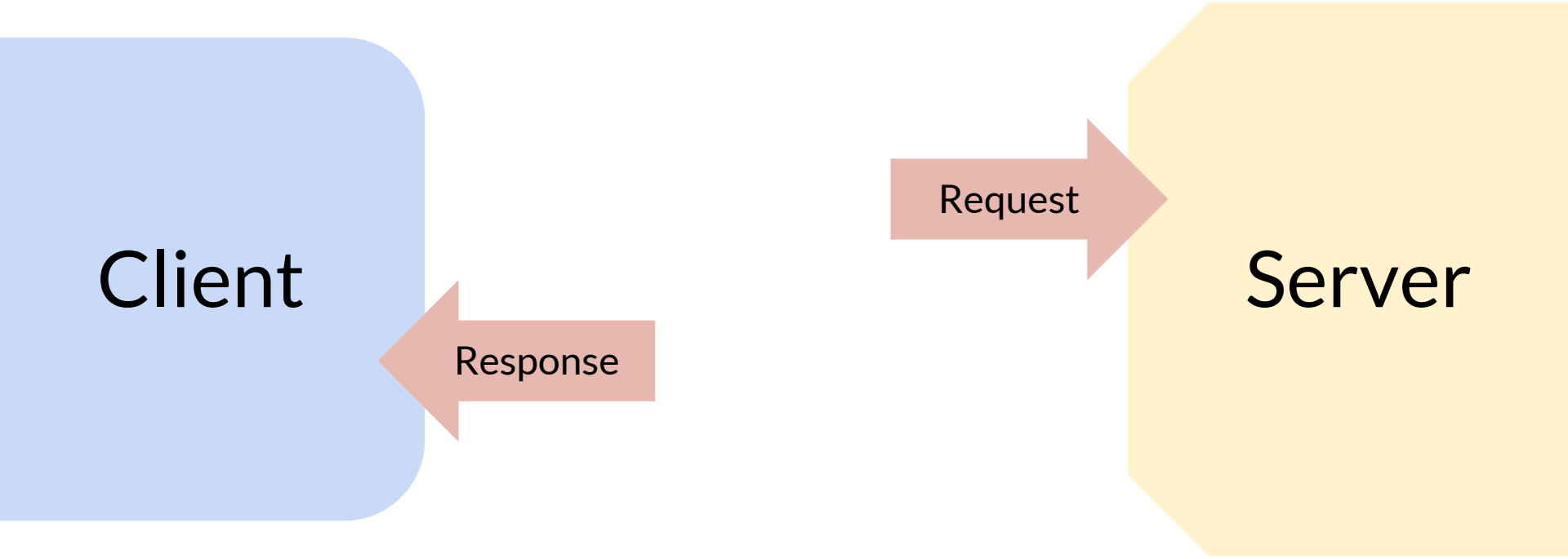
The Other Side

So far, we have learned how to send HTTP requests from a client and handle the server's response.

Now, we are going to learn how to write the server code that **listens** for HTTP requests from clients and **sends** responses back to them.

When we are done, you will be able to run your own server, very similar to the APIs that we have used.

The Other Side



Recap – HTTP Methods

GET - retrieves information

POST - sends information

PUT - updates information

DELETE - removes information

Many others - OPTIONS, HEAD, TRACE, CONNECT, and also nonstandard methods

Recap – JSON

JSON is simply a data structure. It is written in name:value pairs and it can be nested:

```
{
   "name": "John",     # String value
   "age": 15           # Number value
   "siblings": [       # Array value
      {"name": "Jane", "age": 17},
      {"name": "Jill", "age": 13}
   ]
}
```

Creating a Simple Server

Steps we need to take:

1. Create a way to handle requests (GET, POST, etc)
    a. i.e. when our server receives a GET/POST request, we need to send a response back and perform any actions needed
2. Make sure the server is constantly listening for requests
3.

Creating a Simple Server

Let's build a server now.

Python 3 has a built-in HTTP server class named `HTTPServer` and a request handler class named `BaseHTTPRequestHandler`.

```
from http.server import HTTPServer, BaseHTTPRequestHandler
```

Creating a Simple Server

Let's say we want to send a GET request to the server. Unfortunately, the `BaseHTTPRequestHandler` class does not have any HTTP method handlers, so it doesn't know what to do with our GET request.

Fortunately, we can **subclass** `BaseHTTPRequestHandler` and provide our own method handlers.

```
class DemoHandler(BaseHTTPRequestHandler):
```

Sidenote on Classes

See http://www.jesshamrick.com/2011/05/18/an-introduction-to-classes-and-inheritance-in-python/ for a full length tutorial if you'd like more info

Creating a Simple Server

```
# Create a subclass to attach HTTP method handlers
class DemoHandler(BaseHTTPRequestHandler):

  # GET method handler (this will run every time it receives a GET request)
  def do_GET(self):

    # Add a "200 OK" response code to the headers buffer
    self.send_response(200)

    # Add the Content-type HTTP header to the headers buffer
    self.send_header('Content-type','text/html')

    ...
```

Creating a Simple Server

```
...

# Send headers from the buffer
self.end_headers()

# The output to send back in the response
response = "Hello world!"

# Write the response into the output stream
self.wfile.write(bytes(response, "utf8"))

return
```

Creating a Simple Server

When our `DemoHandler` receives an incoming GET request, it will run the code in the `do_GET()` method.

Now we have a working HTTP request handler that can handle GET requests. When the server receives an incoming request, it will hand it over to our `DemoHandler` subclass, which will decide what to do with the request.

Instead of adding handlers for POST (and other) requests now, we'll first write the server code to **listen** for incoming requests on a certain **port**.

Creating a Simple Server

```
def run(server_class=HTTPServer, handler_class=DemoHandler):
    print('starting server...')

    # Server settings
    # Run server in localhost on port 8080
    server_address = ("localhost", 8080)
    httpd = server_class(server_address, handler_class)

    print('running server...')
    httpd.serve_forever()
```

Note: this should NOT be part of the DemoHandler class!

Creating a Simple Server

Now, when we call `run()` and the server will listen for incoming GET requests on `localhost:8080`.

Web browsers (IE, Chrome, Firefox) send a GET request to port 80 by default. You can change this by specifying the port number after the host address. Since our server is listening on port 8080, we need to specify this new port.

Try it out:

Once your code is running, open up a browser and enter `localhost:8080` in the address bar.

If “Hello world!” appears, congratulations! Your server successfully handled and responded to the GET request.

JSON Response

So now we know how to send a simple response from a server. Examine the following line in the code:

```
# Add the Content-type HTTP header to the headers buffer
self.send_header('Content-type','text/html')
```

We have specified that the content that we are sending in the response is of type `text/html`. There are other content types, and in particular we are interested in `application/json`.

JSON Response

Try it out:

Currently, the code sends a “Hello World” message as a response. Now, try to modify the code to send a JSON response. You can create your own JSON data or you can use this one:

```
response_dict = {
    "business": "Pizza Hut",
    "founded": 2010, # Bogus date
    "locations": [
        "Pasir Ris",
        "Tampines",
        "Bishan",
        "Toa Payoh",
        "Orchard" # ...and everywhere else
    ]
}
```

Hint : The `json` package provides `json.dumps()`, which can convert Python dicts into JSON-formatted strings.

JSON Response

Answer:

First, remember to

```
import json
```

Also, change the content type:

~~`self.send_header('Content-type','text/html')`~~

```
self.send_header('Content-type','application/json')
```

Then…

JSON Response

```
response_dict = {...}

# Convert dict to a JSON-formatted string
response_json = json.dumps(response_dict)

# Then convert that into a bytes object with UTF-8 encoding
message_bytes = bytes(response_json, "utf8")

# Then write the bytes into the output stream
self.wfile.write(message_bytes)
```

JSON Response

Don’t forget, we can only send our response in ‘bytes’ format! So we need to convert our response:

`Python Dict > JSON > bytes`

JSON Response

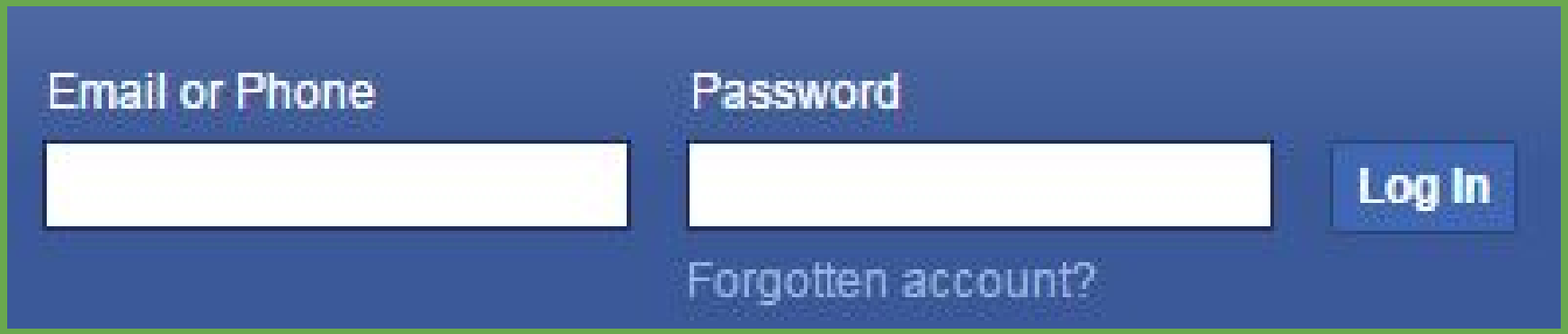
```
# Shorthand:

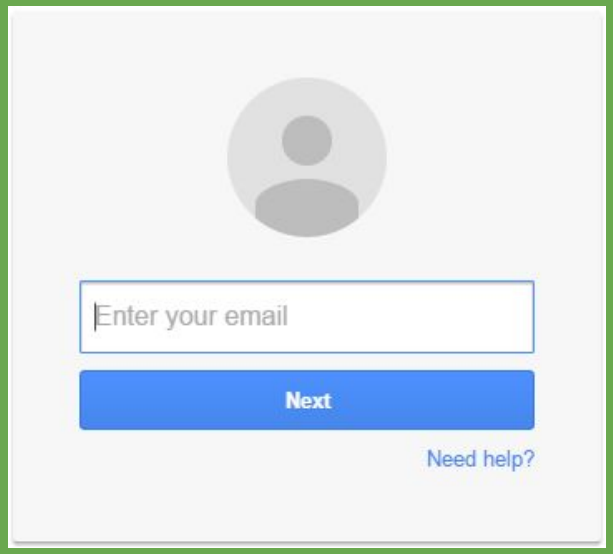
# self.wfile.write(bytes(json.dumps(response_dict), "utf8"))
```

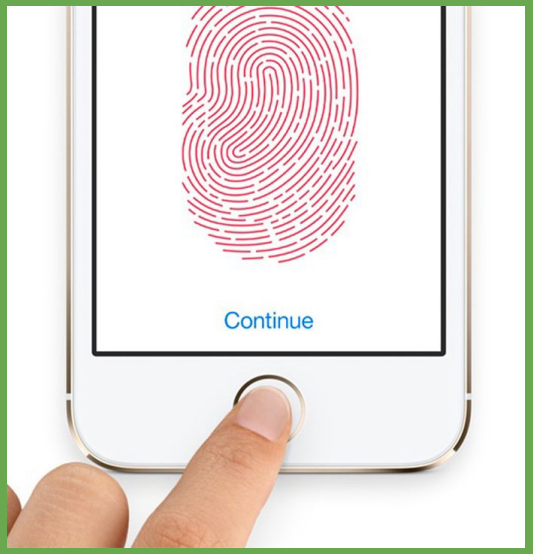
Now if you go to `localhost:8080`, you should see something like:

```
{"locations": ["Pasir Ris", "Tampines", "Bishan", "Toa Payoh", "Orchard"], "founded": 2010, "business": "Pizza Hut"}
```

Fantastic. Now save your code file as `json_server.py` and create a new code file.







Authentication

Authentication

Do you remember how the APIs we used needed API keys?

API keys are a form of **authentication**, the process of identifying internet users. The usual username + password combination is also a form of authentication.

We need authentication to ensure that API users are **authorized** to access the resources. An example of when we might want to prevent access is in cases of abuse.

Authentication allows us to track individual users of our API, and if they are abusing our API, it provides a convenient means to withdraw access privilege from that specific user.

Authentication

In practice, implementing proper authentication requires these additional things in our server application:

- **Random character** generator to generate API keys
- **Database** to store the API keys and track usage
- Server code to **interface** with the database
- Server code to **check for validity** of API keys

Authentication - Database

We are going to be using **SQLite**, which is conveniently packaged with Python 3 in `sqlite3`.

To use databases, we need to have an understanding of database operations and the SQL commands to achieve them.

For now, we'll just provide the SQL code that you can use to run the SQL operations which you need for authentication. We'll learn more about SQL after.

If you want to learn more, you can do so at: https://docs.python.org/3.5/library/sqlite3.html

Authentication - Database

```
import sqlite3


# Initialise db at the start of program
def db_init():
    db = sqlite3.connect("auth.db")
    db.execute("""CREATE TABLE IF NOT EXISTS keys
                  (key text, status integer,
                  quota integer, used integer)""")
    db.commit()
    db.close()
...
```

Authentication - Database

```
...
# Insert new key into db
def db_keygen():
    db = sqlite3.connect("auth.db")
    db.execute("INSERT INTO keys VALUES ('ASDF123', 1, 10, 0)")
    db.commit()
    for row in db.execute("SELECT * FROM keys"):
        print(row)
    db.close()

db_init()
db_keygen()
```

Authentication - Database

Your code should output `('ASDF123', 1, 10, 0)`.

Now, to insert from variables the right way (to prevent **SQL injection** attacks), we have to use **parameter substitution** (this automatically escapes everything inside the tuple):

```
newkey = "QWER456"

t = (newkey,) # t is a tuple

db.execute("INSERT INTO keys VALUES (?, 1, 10, 0)", t)
```

Note: here we know what newkey is, but we should always practice good safety in case we don't know what newkey is, someone makes a mistake, or we take user input.

We have to use `t = (key,)` to convert our key into a **tuple** because parameter substitution only accepts tuples.

Authentication - Database

Update db_keygen( ) to match the following:

```
# Generate a new key, insert into db, and return key
def db_keygen():
    newkey = "QWER456"
    t = (newkey,)
    db = sqlite3.connect("auth.db")
    db.execute("INSERT INTO keys VALUES (?, 1, 10, 0)", t)
    db.commit()
    for row in db.execute("SELECT * FROM keys"):
        print(row)
    db.close()
    return newkey
```

Authentication - Key Generation

Now let's generate a random string for our API key.

The `random` module (be sure to import it!) contains **`random.choice(seq)`** which chooses a random element out of `seq`.

The `string` module contains **`string.ascii_letters`** and **`string.digits`** which contain the ASCII letters (a–z + A–Z) and digits (0–9) respectively.

Authentication - Key Generation

Try it out:

Write a function **`keygen(length)`** that lets you generate a random string containing (a–z + A–Z + 0–9) of a specified `length`.

Authentication - Key Generation

```
import string
import random

def keygen(length=32):
    charspace = string.ascii_letters + string.digits
    key= ""
    for _ in range(length):
        key += random.choice(charspace)
    return key
```

Authentication – Key Generation

Now that we have written `keygen()`, incorporate it into the previous code for data entry into the database, replacing the `"QWER456"` string with a string generated with `keygen()`.

Make sure it works by running the code a few times and seeing that there are new keys which are randomly generated in the database:

Note: your database is stored in an auth.db file.

```
('ASDF123', 1, 10, 0)
('QWER456', 1, 10, 0)
('t2wAvWKqYju3LNld0Wm88lD9iZXsOsZ2', 1, 10, 0)
('qC0vL9QfKlDbFNgHre7DsSgEgQwb5nkr', 1, 10, 0)
('CBfwpFgZM1ighhW0UXbeuJFXSL4BI5nO', 1, 10, 0)
```

Databases and SQL

Now, let's take a closer look at this `sqlite3` module that we have been using.

SQL stands for Structured Query Language, a special-purpose programming language used to manage data in a database.

`sqlite3` is a Python module that uses the lightweight SQLite library, which is based on SQL.

Basically, it allows us to perform database operations with easy-to-understand commands right from within our Python program.

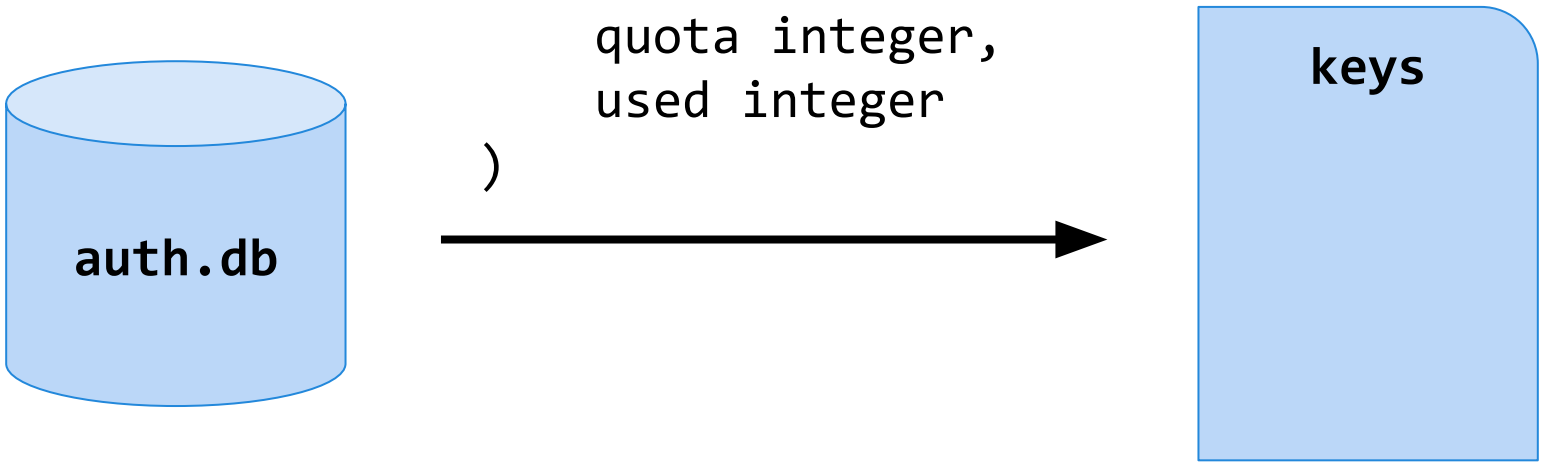
Databases and SQL

SQL operations:	SQL commands:
Create new table	**CREATE**
Insert records into a table	**INSERT**
Retrieve data from a table	**SELECT**
Update information in a table	**UPDATE**
Delete information from a table	**DELETE**
Delete tables	**DROP**

Databases and SQL

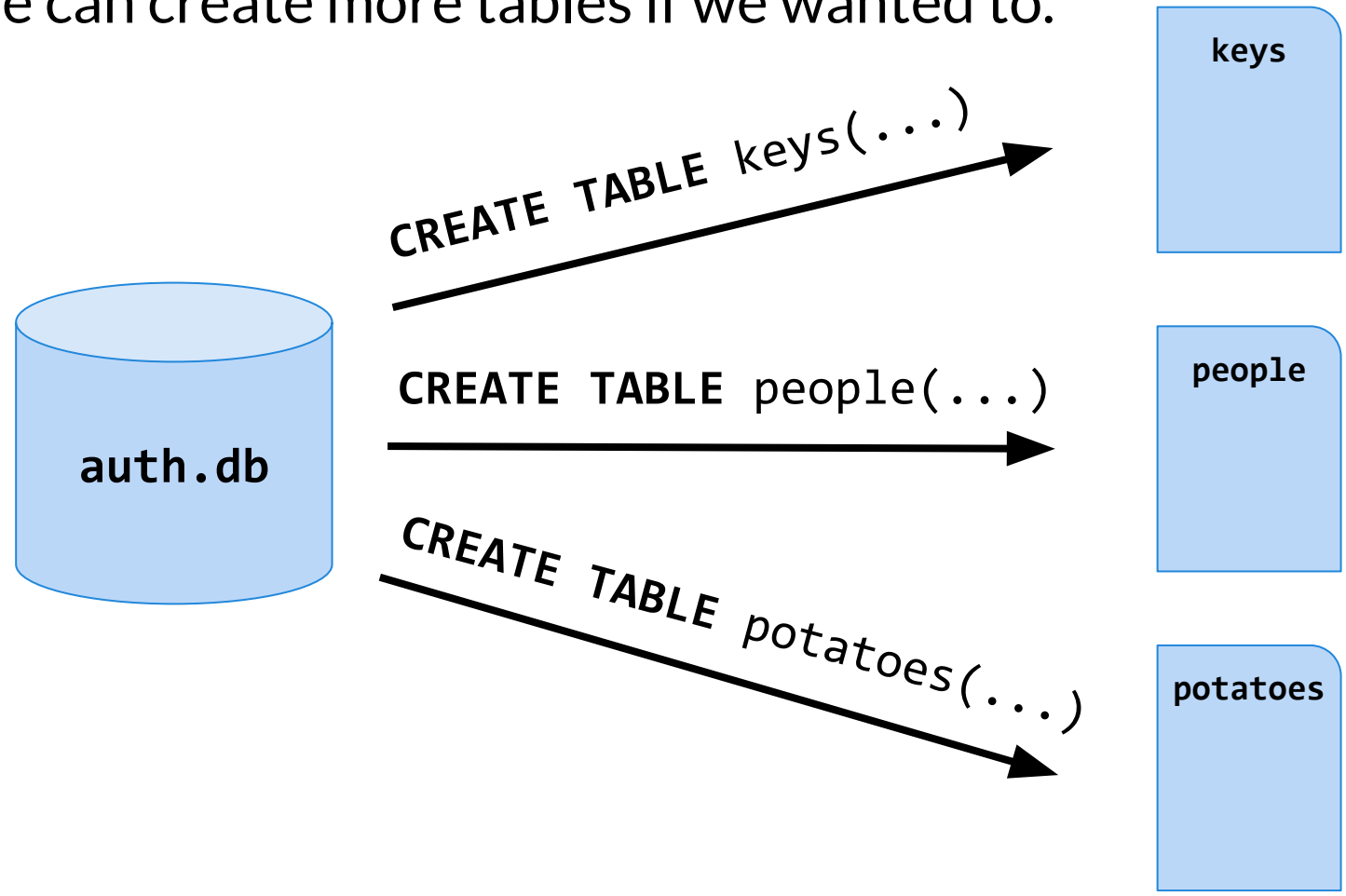
To put data into a database, first we must create a table.

```
CREATE TABLE keys (
    key text,
    status integer,
    quota integer,
    used integer
)
```



Databases and SQL

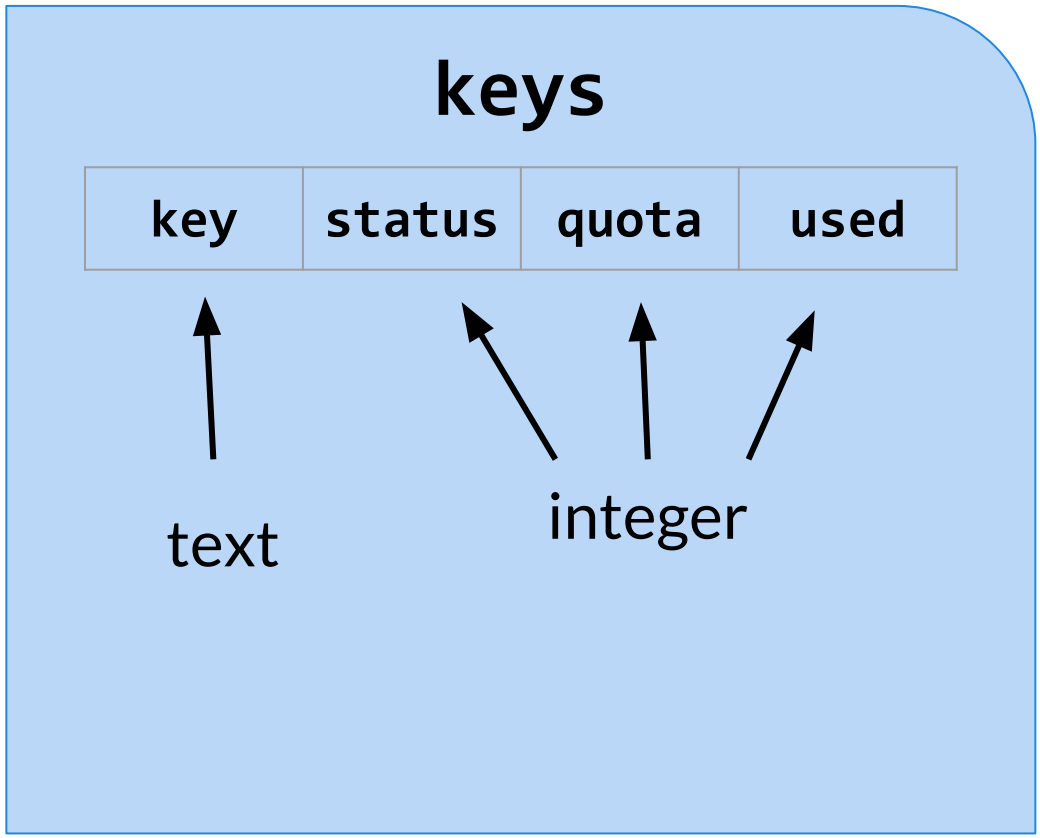
We can create more tables if we wanted to.



Databases and SQL

The CREATE command lets you specify the column **names** and **data types**.

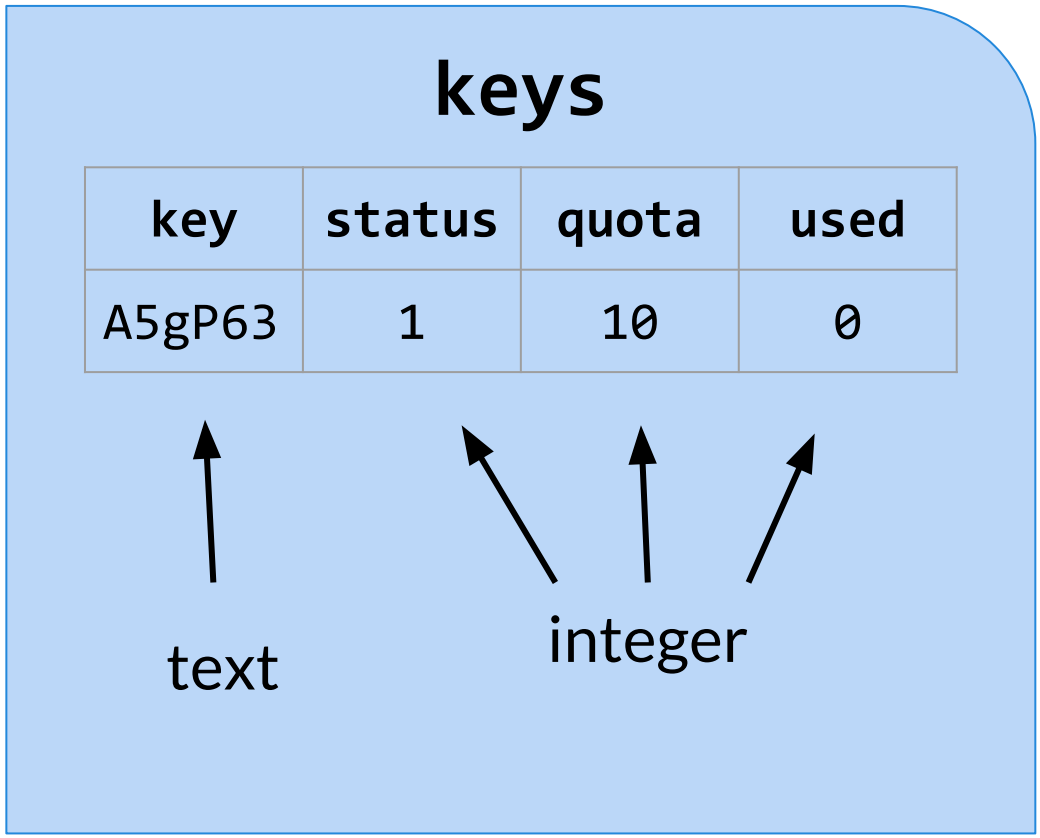
```
CREATE TABLE keys (
    key text,
    status integer,
    quota integer,
    used integer
)
```



Databases and SQL

The INSERT command lets you insert records, but they must match the data types.

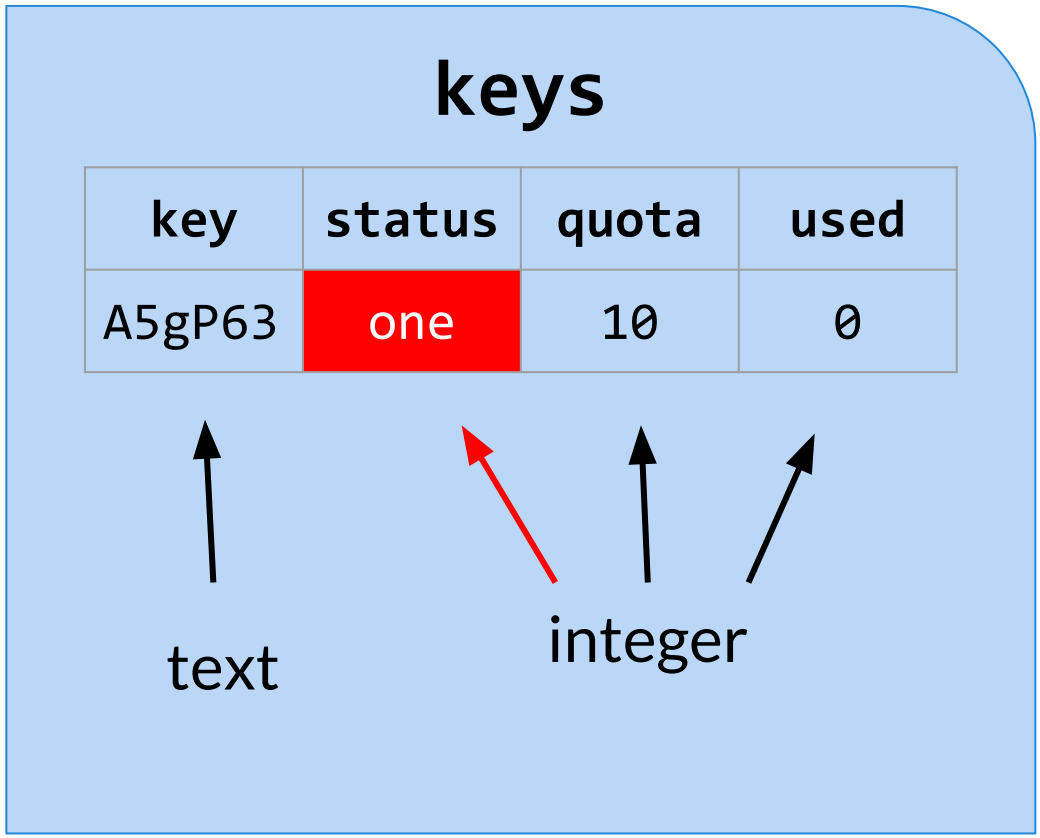
```
INSERT INTO keys
VALUES (
    A5gP63,
    1,
    10,
    0
)
```



Databases and SQL

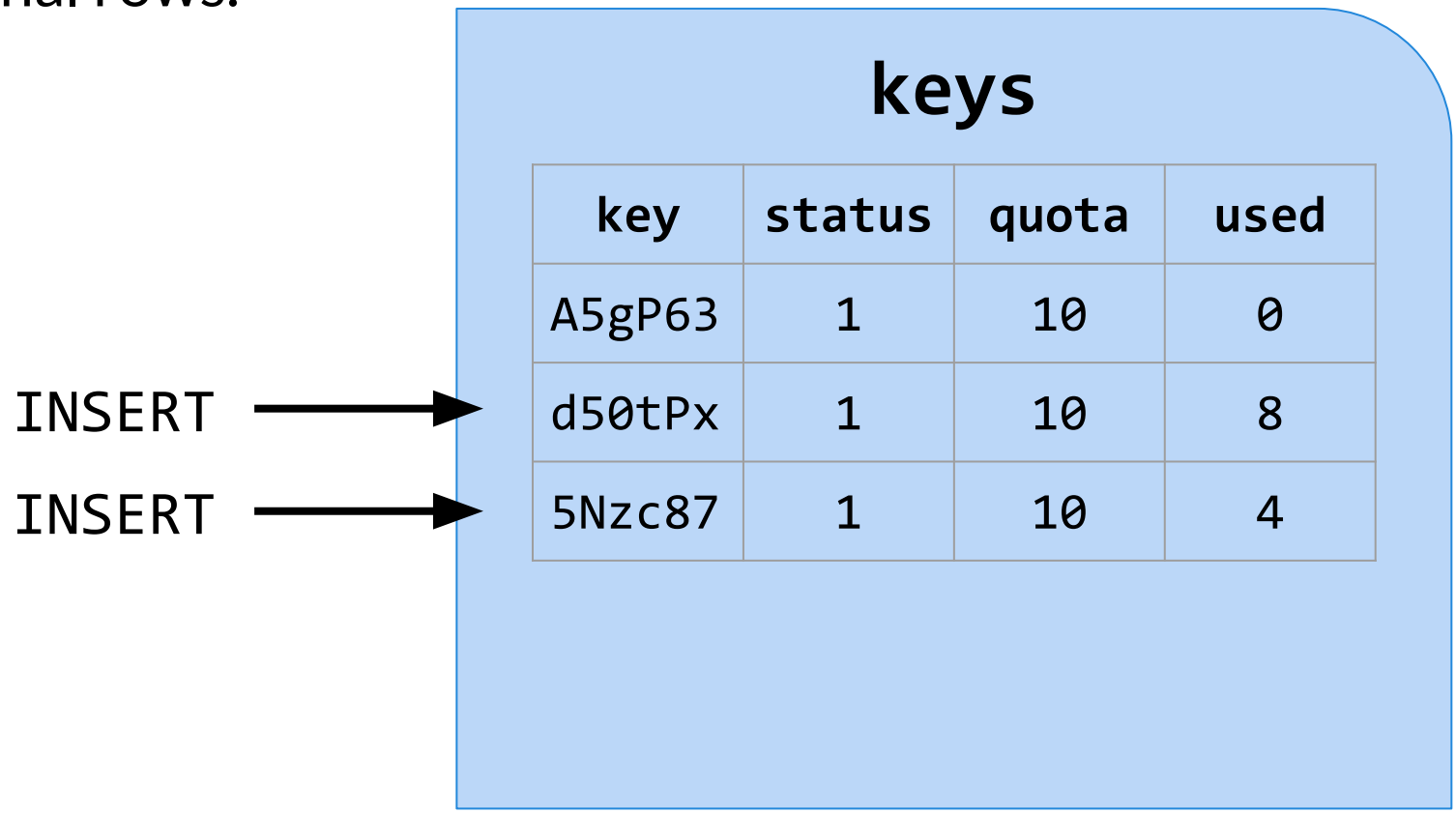
If you try to INSERT a record with an invalid data type, an error will occur.

```
INSERT INTO keys
VALUES (
    A5gP63,
    one,
    10,
    0
)
```



Databases and SQL

When you INSERT more data, it will be appended as additional rows.



Databases and SQL

When you have several records, you can try to SELECT the data from your table.

```
SELECT * FROM keys

>>> 3 rows {...}
```

keys

key	status	quota	used
A5gP63	1	10	0
d50tPx	1	10	8
5Nzc87	1	10	4

Databases and SQL

You can SELECT with a WHERE constraint to check for the existence of data.

```
SELECT * FROM keys
WHERE key = d50tPx

>>> 1 row {...}
```

keys

key	status	quota	used
A5gP63	1	10	0
d50tPx	1	10	8
5Nzc87	1	10	4

Databases and SQL

You can SELECT specific columns like this.

```
SELECT key, quota
FROM keys

>>> 3 rows {...}
```

keys

key	status	quota	used
A5gP63	1	10	0
d50tPx	1	10	8
5Nzc87	1	10	4

Databases and SQL

You can even SELECT a specific column with a WHERE constraint like this.

```
SELECT quota, used
FROM keys
WHERE key = 5Nzc87

>>> 1 row {...}
```

keys

key	status	quota	used
A5gP63	1	10	0
d50tPx	1	10	8
5Nzc87	1	10	4

Databases and SQL

Learn more about how to use Python with databases at https://docs.python.org/3.5/library/sqlite3.html

There, you can learn more about the UPDATE, DELETE, and DROP commands.

Authentication – Server

Try it out: Heavy coding ahead!

We will provide you with:

1. `api_client_get.py`, containing code to send a GET request for an API key from the server
2. `api_client_post.py`, containing code to send a POST request for the API data using the API key for authorisation. (note: we're using POST to request data here because we need to send over the API key in the body of the request)

Your job is to write `api_server.py`, which will take the respective requests and give the expected response.

Authentication – Server

`api_server.py` should be able to:

1. Handle a GET request from `api_client_get.py`, by storing a randomly generated API key into the database and responding to the client with the key string.
2. Handle a POST request from `api_client_post.py`, by extracting the API key from the data field and checking for its validity. If valid, respond to the client with the JSON data. If invalid, respond with a `403-Forbidden` error.

To test this out, use one instance of IDLE to run your server, and open a second instance of IDLE to run your client GET/POST requests.

Authentication – Server

Hints:

- You have learned how to write a simple json server and you saved your code in `json_server.py`. You need this framework to create `api_server.py`.
- You can run two IDLE console processes at once; just open another instance of the IDLE console. Use one to run the server and another to run the client.
- You have already written `db_init()`, `db_keygen()`, and `keygen()`. You need these methods in `api_server.py`.
- You should put `db_init()` right before `httpd.serve_forever()`.
- You should write the method `auth_key(`*`key`*`)` that returns a boolean, to verify that *`key`* is indeed valid and in the database.

Authentication – Server

Hints (2):

- You should use `self.rfile.read(size)` and `urllib.parse.parse_qs(query-string)` from the `urllib.parse` module to handle POST data from the client.
- You must supply ***size*** to `self.rfile.read(`***size***`)`.
- ```
  data = self.rfile.read(
      int(self.headers.get("Content-length")))
      .decode("utf-8")
  ```
- `data = urllib.parse.parse_qs(data)`
- `key = data["key"][0]`
- How do we send a `403 Forbidden` error back to the client?

  ```
  self.send_response(403, "API key is invalid")
  ```
  ```

Authentication – Server

To see how everything works together:

Make sure the server code is running. Then, in a separate IDLE console, run `api_client_get.py` and copy the key in the response. Then, replace the placeholder key in `api_client_post.py` and run it. It should return the server's API data in JSON format.

Stuck? View our example code at: tk.sg/apiserver

Wrap-Up

The server that we have made is a very simple API server. We have not implemented:

- Authorisation levels (e.g. admin/user)
- Usage tracking and abuse detection
- A banning policy (e.g. ban if >100 requests/min)
- E-mail verification
- Actual data
- Data selectors (e.g. only return data for past week)
- Data uploading

You'll be adding some of these features in the Midterm project!

Midterm Project

Objective: Add features to api_server.py

Distributed via Slack

Due 24 Aug, 11:59pm