PUT and DELETE operations

INTRODUCTION TO FASTAPI

Matt Eckerle
Software and Data Engineering Leader

datacamp

PUT vs. DELETE

PUT Operations

- Traditional use: update an existing object
- Parameters sent via query string as well as request body
- Requires an application or framework
  - e.g. `cURL`, `requests`

```
api = "http://moviereviews.co/reviews/1"
body = {"text": "A fantastic movie!"}
response = requests.put(api, json=body)
```

DELETE Operations

- Traditional use: delete an existing object
- Parameters sent via query string as well as request body
- Requires an application or framework
  - e.g. `cURL`, `requests`

```
api = "http://moviereviews.co/reviews/1"
response = requests.delete(api)
```

Referencing Existing Objects

- No ORM, so app must map object to ID
- Database ID - unique identifier
- `_id` convention for database IDs
  - `review_id` : Table `reviews` , column `id`
  - Same convention in frameworks with ORM

```python
from pydantic import BaseModel

class DbReview(BaseModel):
    movie: str
    num_stars: int
    text: str
    # Reference database ID of Reviews
    review_id: int
```

Handling a PUT Operation

PUT endpoint to update an existing movie review:

- Endpoint: `/reviews`
- Input: `DbReview` (from previous slide)
- Output: `DbReview`

```
@app.put("/reviews", response_model=DbReview)
def update_review(review: DbReview):
    # Update the movie review in the database
    db_review = crud.update_review(review)
    # Return the updated review
    return db_review
```

Handling a DELETE Operation

DELETE endpoint to remove an existing movie review:

- Endpoint: `/reviews`
- Input: `DbReview`
- Output: `{}`

```python
@app.delete("/reviews")
def delete_review(review: DbReview):
    # Delete the movie review from the database
    crud.delete_review(review)
    # Return nothing since the data is gone
    return {}
```

Let's practice!

INTRODUCTION TO FASTAPI

Handling errors

INTRODUCTION TO FASTAPI

Matt Eckerle
Software and Data Leader

Two Main Reasons To Handle Errors

User error

- Invalid or outdated URI
- Missing or incorrect input

```python
@app.delete("/items")
def delete_item(item: Item):
    if item.id not in item_ids:
        # Return an error
    else:
        crud.delete_item(item)
        return {}
```

Server error

- Something else happened

```python
@app.delete("/items")
def delete_item(item: Item):
    try:
        crud.delete_item(item)
    except Exception:
        # Return an error
    return {}
```

HTTP Status Codes: "Levels of Yelling"

- Enables API to provide status in response
  - Success, failure, error, etc.
- Specific codes defined in HTTP protocol
- Range: `100` - `599`
- Categorize by first number (`1` - `5`)

1. Informational responses (`100` - `199`)
2. Successful responses (`200` - `299`)
3. Redirection messages (`300` - `399`)
4. Client error responses (`400` - `499`)
5. Server error responses (`500` - `599`)

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Common HTTP Status Codes

Success (`200` - `299`)

- `200 OK`
  - Default success response
- `201 Created`
  - Specific to POST operation
- `202 Accepted`
  - Noncommittal. "Working on it"
- `204 No Content`
  - Success! Nothing more to say

Other responses

- `301 Moved Permanently`
  - URI changed permanently
- `400 Bad Request`
  - Client error
- `404 Not Found`
  - Server cannot find the requested resource
- `500 Internal Server Error`
  - Server has encountered a situation it does not know how to handle

Handling Errors With Status Codes

```python
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.delete("/items")
def delete_item(item: Item):
    if item.id not in item_ids:
        # Send response with status 404 and specific error message
        raise HTTPException(status_code=404, detail="Item not found.")
    else:
        delete_item_in_database(item)
        return {}
```

Let's practice!

INTRODUCTION TO FASTAPI

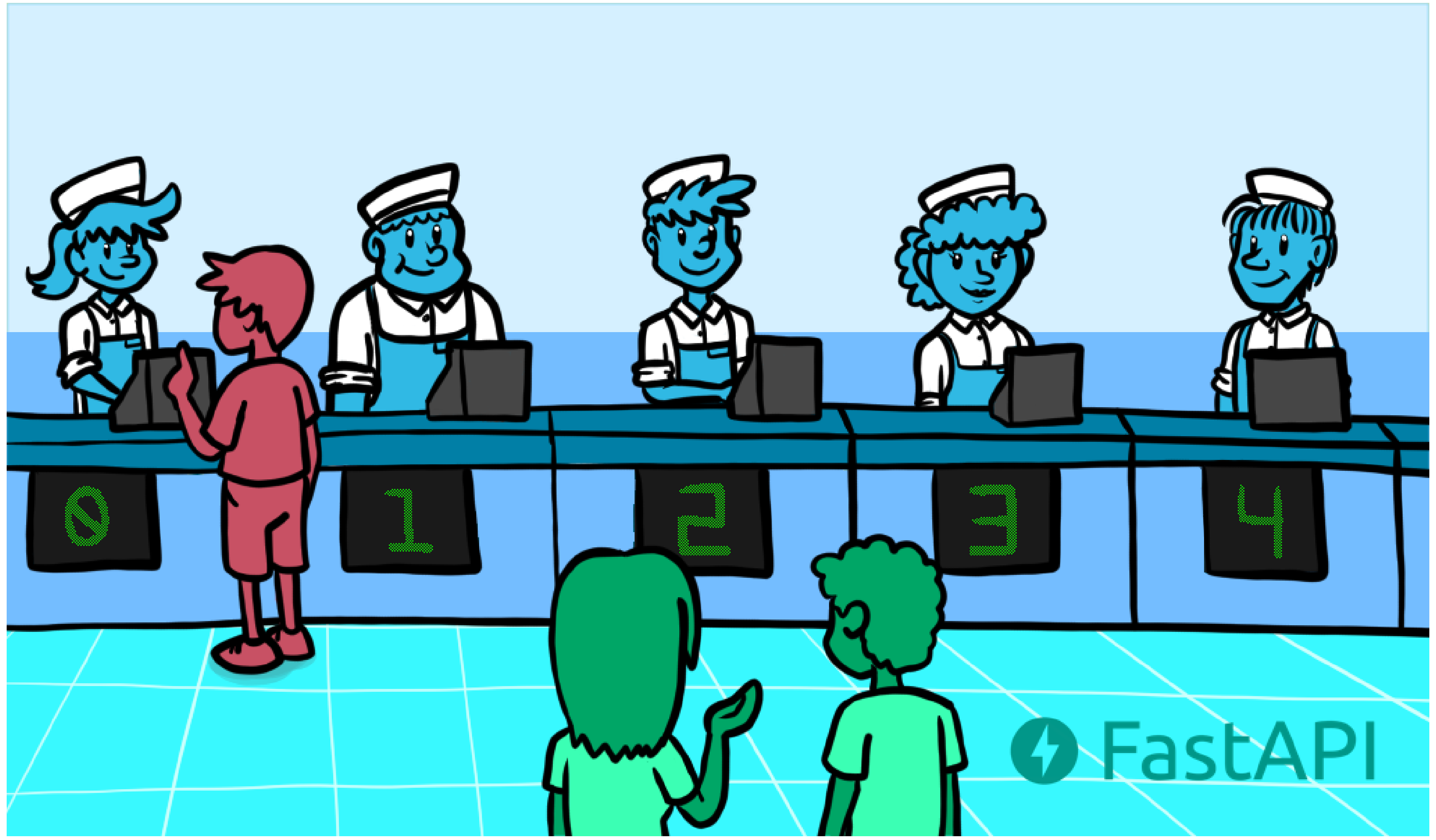
Using async for concurrent work

INTRODUCTION TO FASTAPI

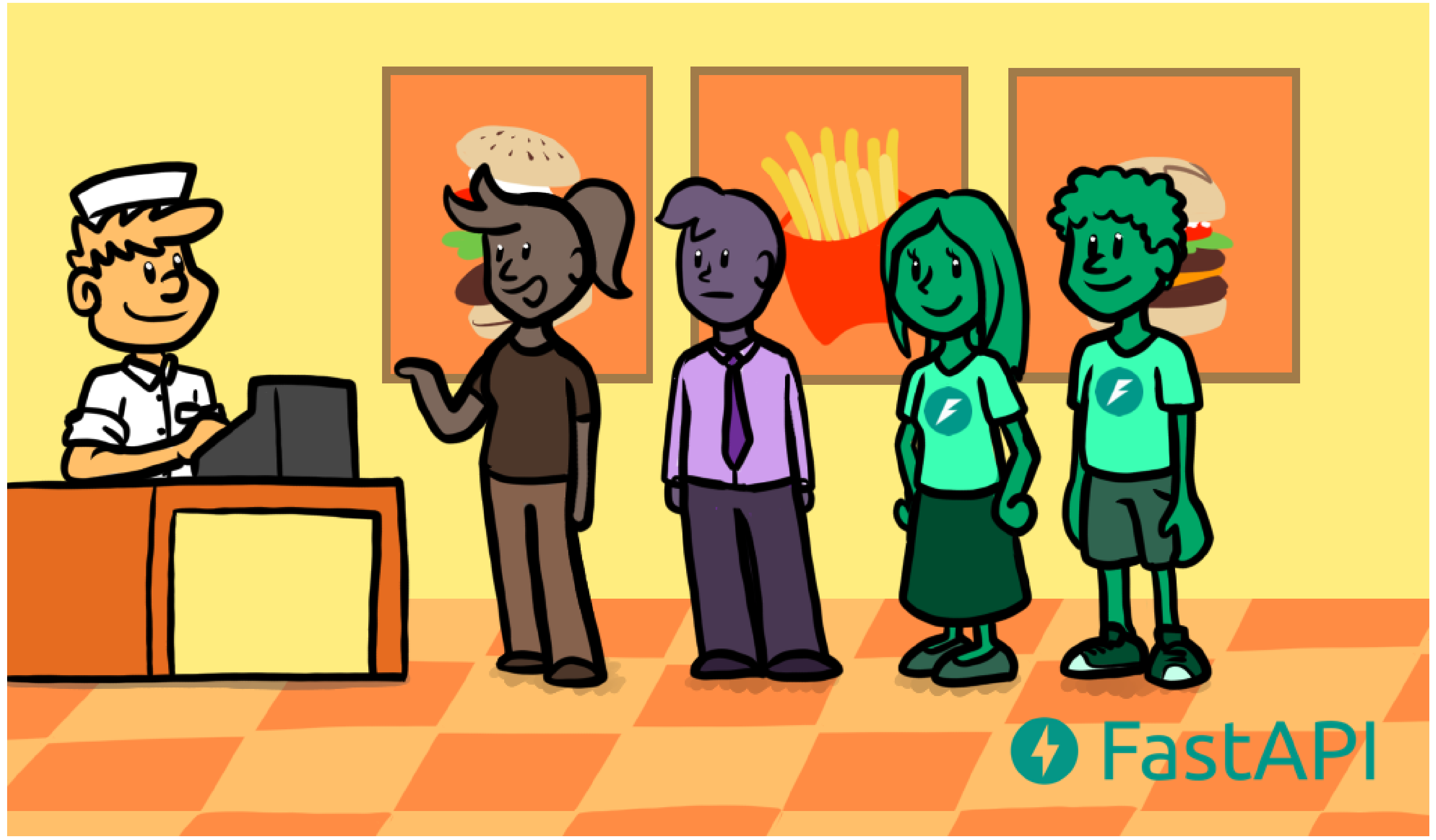
Matt Eckerle
Software and Data Engineering Leader

Why use async? Concurrent Burgers!

Sequential Burgers



Concurrent Burgers



[1] https://fastapi.tiangolo.com/async/

async in practice

Sequential Burgers

Defining a function to get burgers

```
# This is not asynchronous
def get_sequential_burgers(number: int):
    # Do some sequential stuff
    return burgers
```

Calling the function sequentially

```
burgers = get_burgers(2)
```

Concurrent Burgers

Defining a function to get burgers

```
async def get_burgers(number: int):
    # Do some asynchronous stuff
    return burgers
```

Calling the function asynchronously

```
burgers = await get_burgers(2)
```

FastAPI with async

If we can:

```
results = await some_library()
```

Then use `async def` :

```
@app.get('/')
async def read_results():
    results = await some_library()
    return results
```

> **Note** Only use `await` inside of functions created with `async def`

When to use async

Use async

If our application doesn't have to communicate with anything else and wait for it to respond

Examples

- Audio or image processing
- Computer vision
- Machine Learning
- Deep Learning

Don't use async

If our application has to communicate with:

- File system
- Database
- Another server

If we aren't sure

Let's practice!

INTRODUCTION TO FASTAPI