CS6650 Homework 5

Product API with Terraform & Load Testing

Part II: Product API

API Endpoints

Based on the OpenAPI specification, the following two endpoints are implemented:

Method	Endpoint	Description	Status Codes
GET	/products/{productId}	Get product by ID	200, 404, 500
POST	/products/{productId}/details	Add/update product details	204, 400, 404, 500

Product Schema (all fields required)

Field	Type	Constraints
product_id	int32	≥ 1
sku	string	1-100 chars
manufacturer	string	1-200 chars
category_id	int32	≥ 1
weight	int32	≥ 0
some_other_id	int32	≥ 1

How to Run

Run Locally

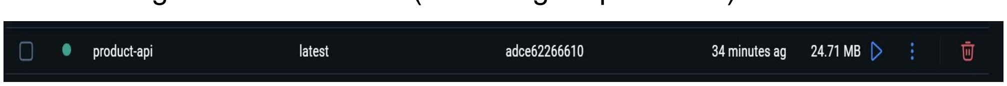
```
cd src
go mod tidy
go run main.go
```

Server starts on http://localhost:5173

Run with Docker

```
docker build -t product-api .
docker run -p 5173:5173 product-api
```

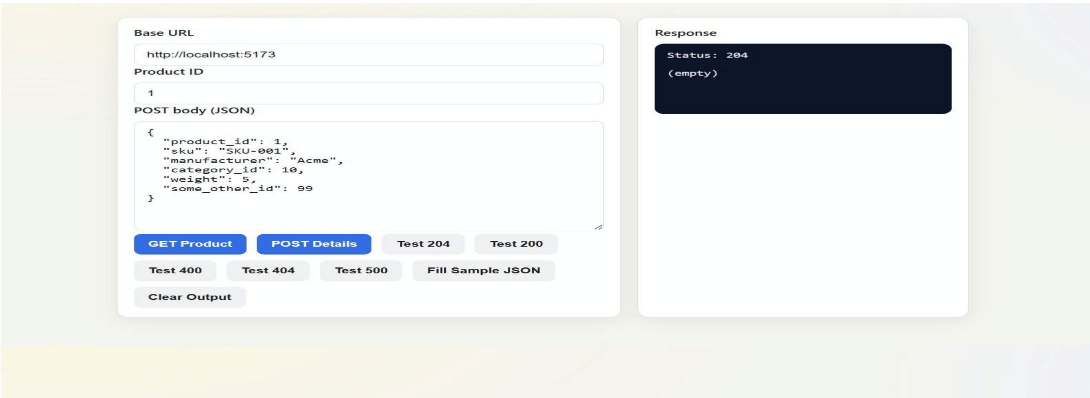
Docker image size: ~24.71 MB (multi-stage Alpine build)



Response Code Examples

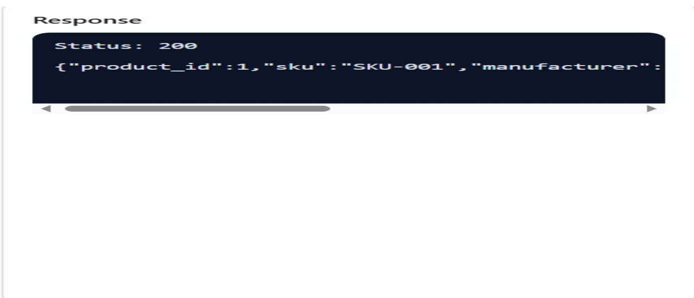
204 — Product Details Added (POST)

```
curl -i -X POST http://localhost:5173/products/1/details \
  -H "Content-Type: application/json" \
  -d '{"product_id":1,"sku":"SKU-
001","manufacturer":"Acme","category_id":10,"weight":5,"some_other_id":99}'
```



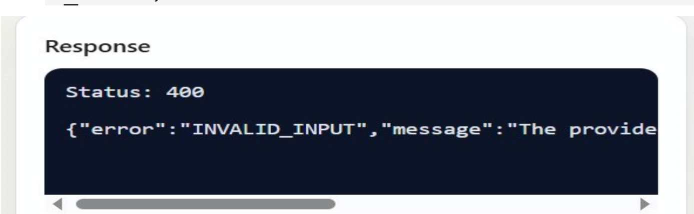
200 — Product Found (GET)

```
curl http://localhost:5173/products/1
```



400 — Invalid Input

```
curl -X POST http://localhost:5173/products/1/details \
  -H "Content-Type: application/json" \
  -d
'{"product_id":1,"sku":"","manufacturer":"Acme","category_id":1,"weight":100,"some_othe
r_id":1}'
```



404 — Product Not Found

```
curl http://localhost:5173/products/9999
```

500 — Internal Server Error

The server includes a panic recovery middleware. Any unexpected panic is caught and returns a 500 JSON response.

```
Status: 404

{"error":"NOT_FOUND","message":"Product not fou
```

Data Storage

Products are stored in-memory using a Go map[int]*Product protected by sync.RWMutex for thread-safe concurrent read/write access. Data does not persist across server restarts.

Part III: Terraform Deployment to AWS

Deployment Steps

- Configure AWS credentials: aws configure
- Initialize Terraform:

```
cd terraform && terraform init -upgrade
```

```
PS C:\Users\98999\Desktop\CS6650_2b_demo\terraform> terraform init -upgrade
Initializing the backend...
Upgrading modules...
- ecr in modules\ecr
- ecs in modules\ecs
- logging in modules\logging
- network in modules\network
Initializing provider plugins...
- Finding kreuzwerker/docker versions matching "~> 3.0"...
- Finding hashicorp/aws versions matching "~> 6.7.0"...
- Using previously-installed hashicorp/aws v6.7.0
- Installing kreuzwerker/docker v3.6.2...
- Installed kreuzwerker/docker v3.6.2 (self-signed, key ID BD080C4571C6104C)
If you'd like to know more about provider signing, you can read about it here:
https://developer.hashicorp.com/terraform/cli/plugins/signing
Terraform has made some changes to the provider dependency selections recorded
in the .terraform.lock.hcl file. Review those changes and commit them to your
version control system if they represent changes you intended to make.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
```

- Deploy: terraform apply
- Get public IP from ECS task network interface:

```
PS C:\Users\98999\Desktop\CS6650_2b_demo>
PS C:\Users\98999\Desktop\CS6650_2b_demo> $eniId = aws ecs describe-tasks `
>>   --cluster $cluster `
>>   --tasks $taskArn `
>>   --query "tasks[0].attachments[0].details[?name=='networkInterfaceId'].value" `
>>   --output text

usage: aws [options] <command> <subcommand> [<subcommand> ...] [parameters]
To see help text, you can run:

  aws help
  aws <command> help
  aws <command> <subcommand> help

Unknown options: Γöé , Γöé The state file either has no outputs defined, or all the defined outputs are empt
. Please define an, Γöé output in your configuration with the `output` keyword and run `terraform refresh` 
r it to become, Γöé available. If you are using interpolation, please verify the interpolated value is not 
pty. You can use, Γöé the `terraform console` command to assist., Γö, Γöé Warning: No outputs found

PS C:\Users\98999\Desktop\CS6650_2b_demo>
PS C:\Users\98999\Desktop\CS6650_2b_demo> aws ec2 describe-network-interfaces `
>>   --network-interface-ids $eniId `
>>   --query "NetworkInterfaces[0].Association.PublicIp" `
>>   --output text
34.213.40.13
```

- Test on AWS:

```
lhq5520@DESKTOP-PDSMRNB:/mnt/c/Users/98999/Desktop/CS6650_2b_demo$ curl -i -X POST http://34.213.40.13:5173/p
roducts/1/details \
H "Co>   -H "Content-Type: application/json" \
>   -d '{
>     "product_id": 1,
>     "sku": "SKU-1001",
>     "manufacturer": "Acme",
>     "category_id": 10,
>     "weight": 5,
>     "some_other_id": 123
>   }'
HTTP/1.1 204 No Content
Date: Sun, 15 Feb 2026 06:33:10 GMT
```

```
lhq5520@DESKTOP-PDSMRNB:/mnt/c/Users/98999/Desktop/CS6650_2b_demo$ curl -i http://34.213.40.13:5173/products/
1
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sun, 15 Feb 2026 06:33:21 GMT
Content-Length: 104

{"product_id":1,"sku":"SKU-1001","manufacturer":"Acme","category_id":10,"weight":5,"some_other_id":123}
```

- Tear down: terraform destroy

Part IV: Load Testing with Locust

Test Script Design

- GET:POST = 3:1 ratio — simulates real-world read-heavy e-commerce traffic
- on_start pre-populates 50 products — ensures GET requests hit valid data
- Both HttpUser and FastHttpUser included for comparison
- wait_time = between(1, 3) — realistic think time

How to Run Locust

```
pip install locust
locust -f locustfile.py --host=http://localhost:5173
```

Open http://localhost:8089 to configure users and spawn rate.

Test Results Summary

Environment	Users	Spawn Rate	Total Requests	Failures	Avg RT (ms)	RPS	Duration
Local	10	2	1,337	0 (0%)	1.1	8.0	2m 47s
Local	100	10	24,468	0 (0%)	2.1	62.0	6m 35s
Local	500	50	142,191	0 (0%)	25.8	295.3	8m 1s
AWS	10	2	1,592	0 (0%)	13.4	7.1	3m 43s
AWS	100	10	18,248	0 (0%)	12.9	67.1	4m 33s
AWS	500	50	89,546	0 (0%)	38.7	332.0	4m 30s

Key Observations

- Zero failure rate across all tests — the Go server handled all load levels without errors.
- Local vs AWS latency: Local tests show ~1ms average response time at low load, while AWS adds ~12ms of network latency due to the round-trip to AWS Fargate.
- Scaling behavior: As users increase (10 → 100 → 500), average response time grows gradually (1.1ms → 2.1ms → 25.8ms locally), showing graceful degradation.
- RPS scales well: RPS increased roughly proportionally with users (8 → 62 → 295 locally), indicating no bottleneck at these levels.

HttpUser vs FastHttpUser

In our tests, we did not observe a significant difference between HttpUser and FastHttpUser. Reasons:

- The bottleneck is the server, not the client. Our Go server responds in ~1ms locally. Both HTTP clients keep up easily at this speed.

- Network latency dominates. When testing against AWS, the ~12ms round-trip far outweighs any client-side overhead difference.
- FastHttpUser shines when generating extremely high RPS from a single worker, or when response times are sub-millisecond and client overhead becomes significant.

Read vs Write Ratio

In a real e-commerce system, reads (GET) vastly outnumber writes (POST). This is why our test uses a 3:1 GET:POST ratio. A sync.RWMutex-protected hashmap is ideal because it allows concurrent reads while only blocking for writes — much better throughput than a regular Mutex for read-heavy workloads.

Discussion Questions

Scalable Backend Design for the Full API

The complete api.yaml defines four services: Products, Shopping Cart, Warehouse, and Payments. A scalable design would use:

- Microservices architecture — separate each service for independent scaling.
- Database per service — Products: read-replica PostgreSQL; Cart: Redis; Warehouse: PostgreSQL with transactions; Payments: ACID-compliant DB.
- Message queue (SQS/Kafka) for async checkout, decoupling services.
- API Gateway for routing, rate limiting, and authentication.
- Caching layer (Redis/Memcached) for frequently accessed product data.

Terraform: Declarative vs Imperative

Declarative means you describe WHAT the desired end state should be, not HOW to get there. You write "I want an ECS cluster with this task" — Terraform figures out the steps.

Imperative languages (like shell scripts) require specifying every step and handling ordering/errors yourself.

Terraform's declarative approach helps by automatically handling dependency ordering, detecting drift, and previewing changes (terraform plan), making infrastructure safer and reproducible.