Scalable event-driven applications with NestJS

A modern framework for building back-end Node.js

Author

DMITRY KHOREV

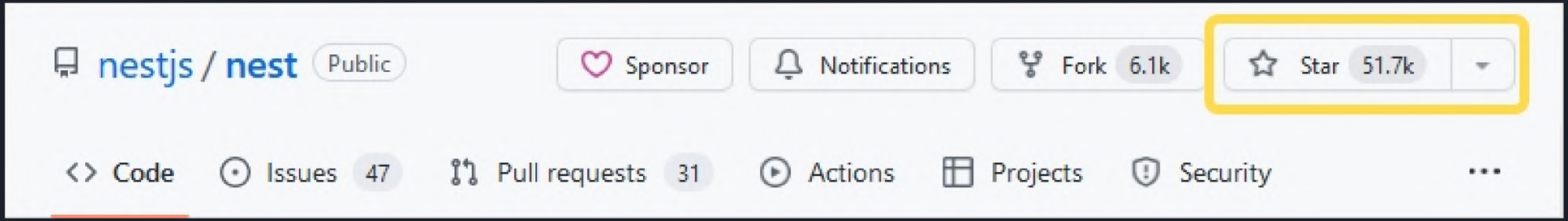
Agenda

- What is NestJS?
- How does it help build scalable applications?
- Demo app and tools
- Demo in action

What is NestJS?

NestJS is a **framework** for building **Node.js** applications.

- **Inspired by Angular**
- **TypeScript**



Why use another framework?

- **Dependency Injection**
- Abstracted integration with **databases**
- Abstracted **common use cases**: *caching, config, API versioning and documentation, task scheduling, queues, logging, cookies, events, and sessions, request validation, HTTP server (express or fastify), auth.*
- **TypeScript** (and decorators)
- Other design elements for great applications: **Middleware, Exception filters, Guards, Pipes**, and so on.
- And some more which I will talk about later...

How does NestJS help?

WITH A SHORT REMINDER ON ARCHITECTURE PARADIGMS

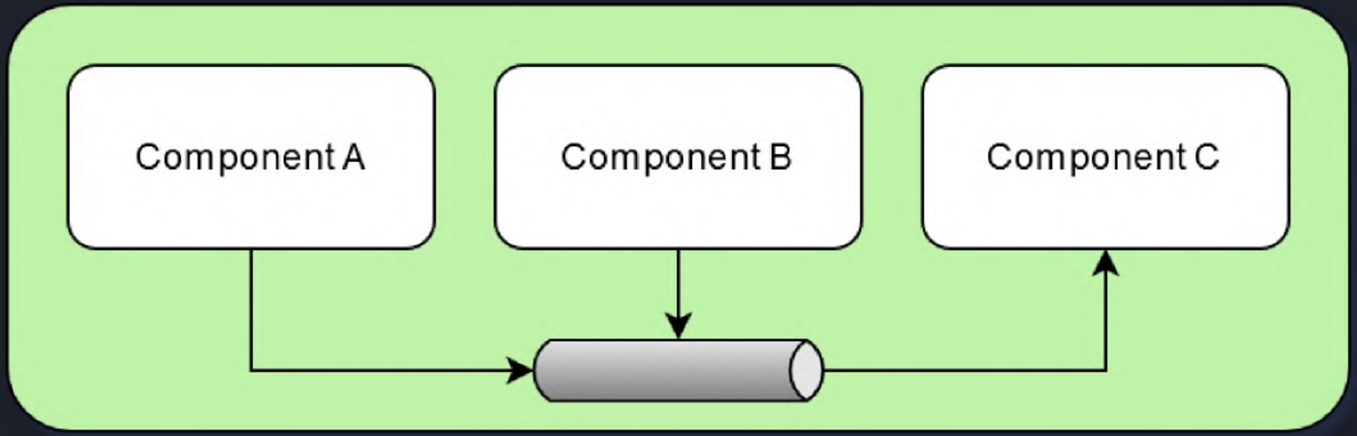
Building highly scalable apps

OPTIONS

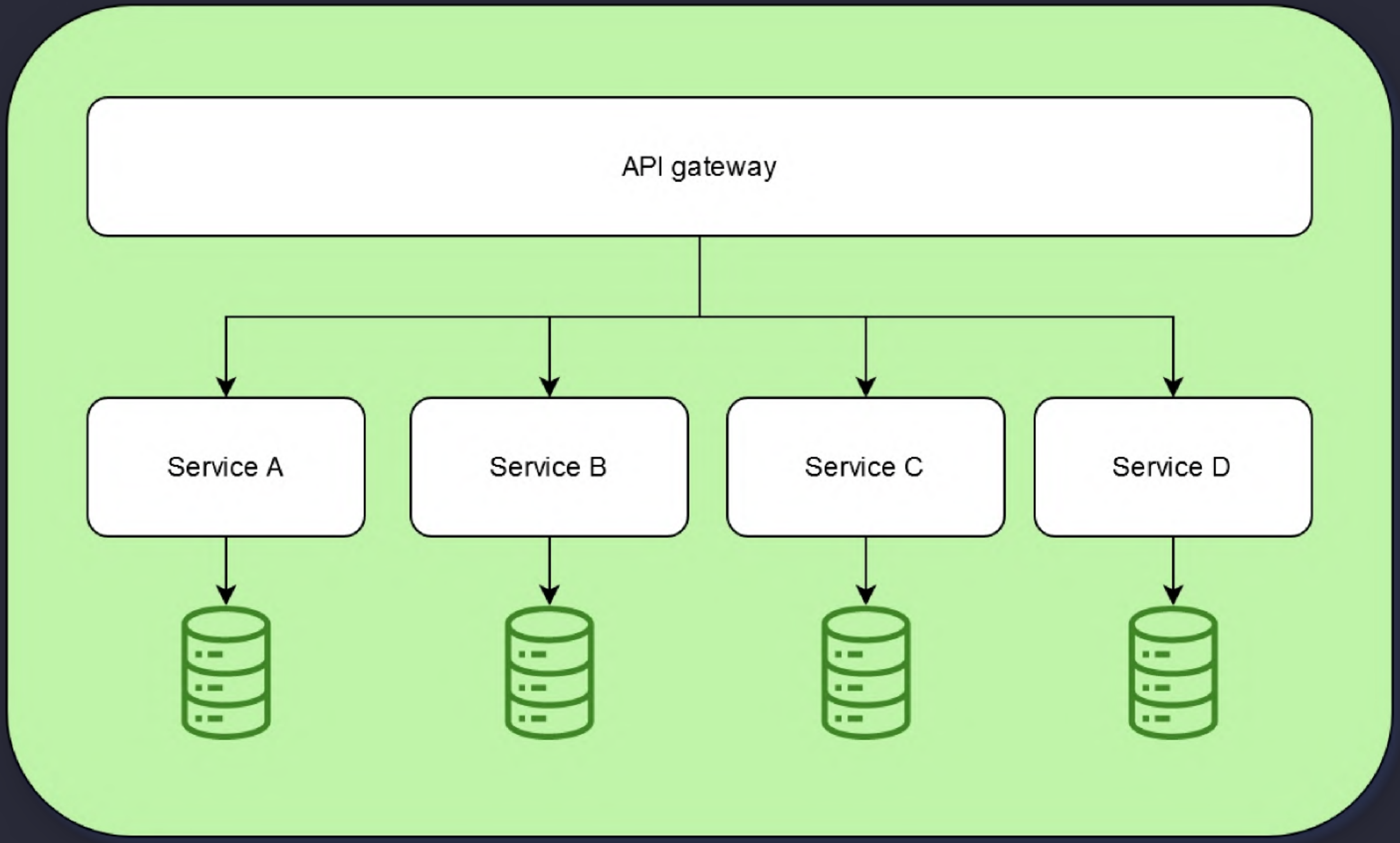
- Monolith (modular)
- Microservices
- Event-driven
- Mixed

Software development is all about trade-offs.

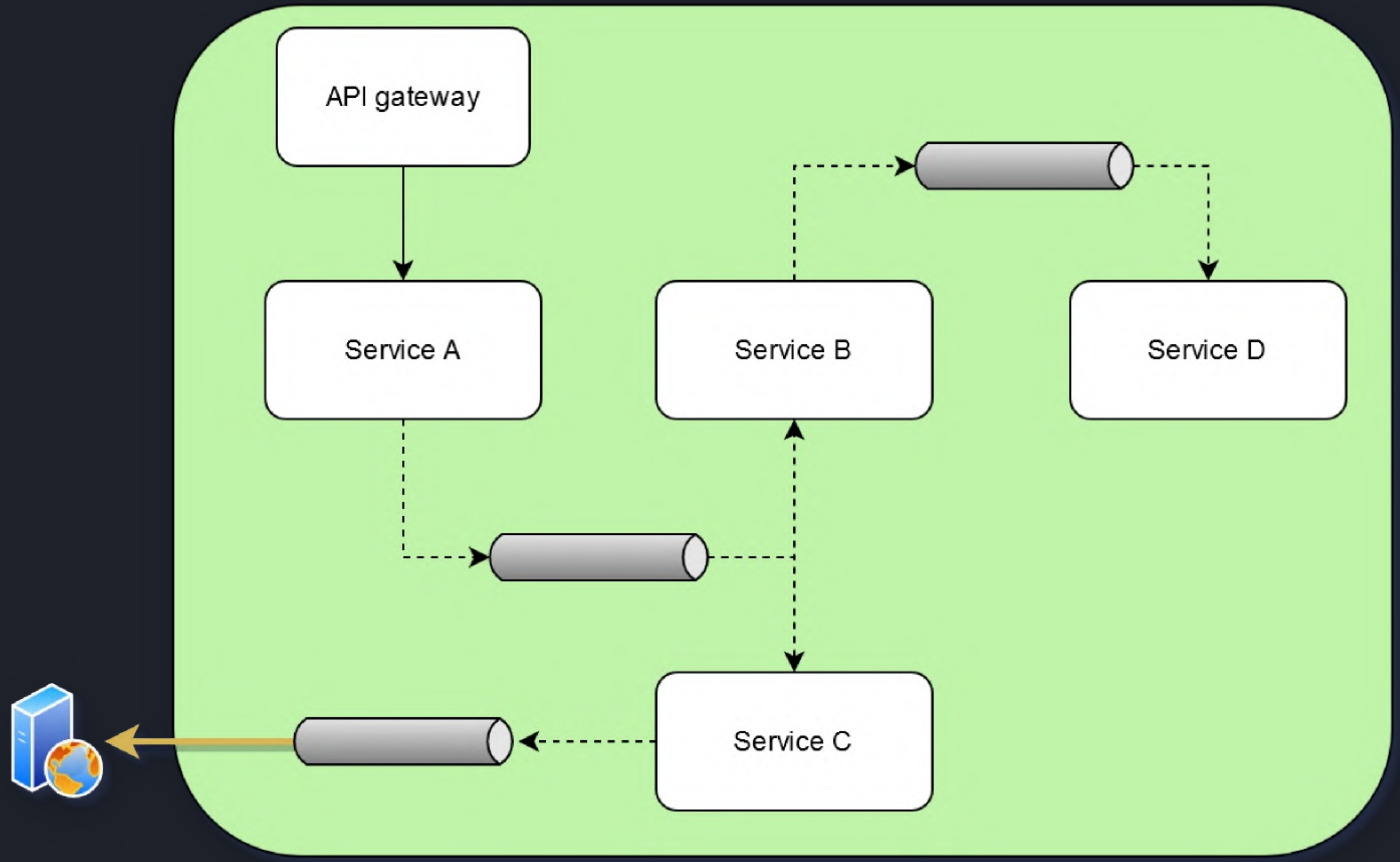
Monolith



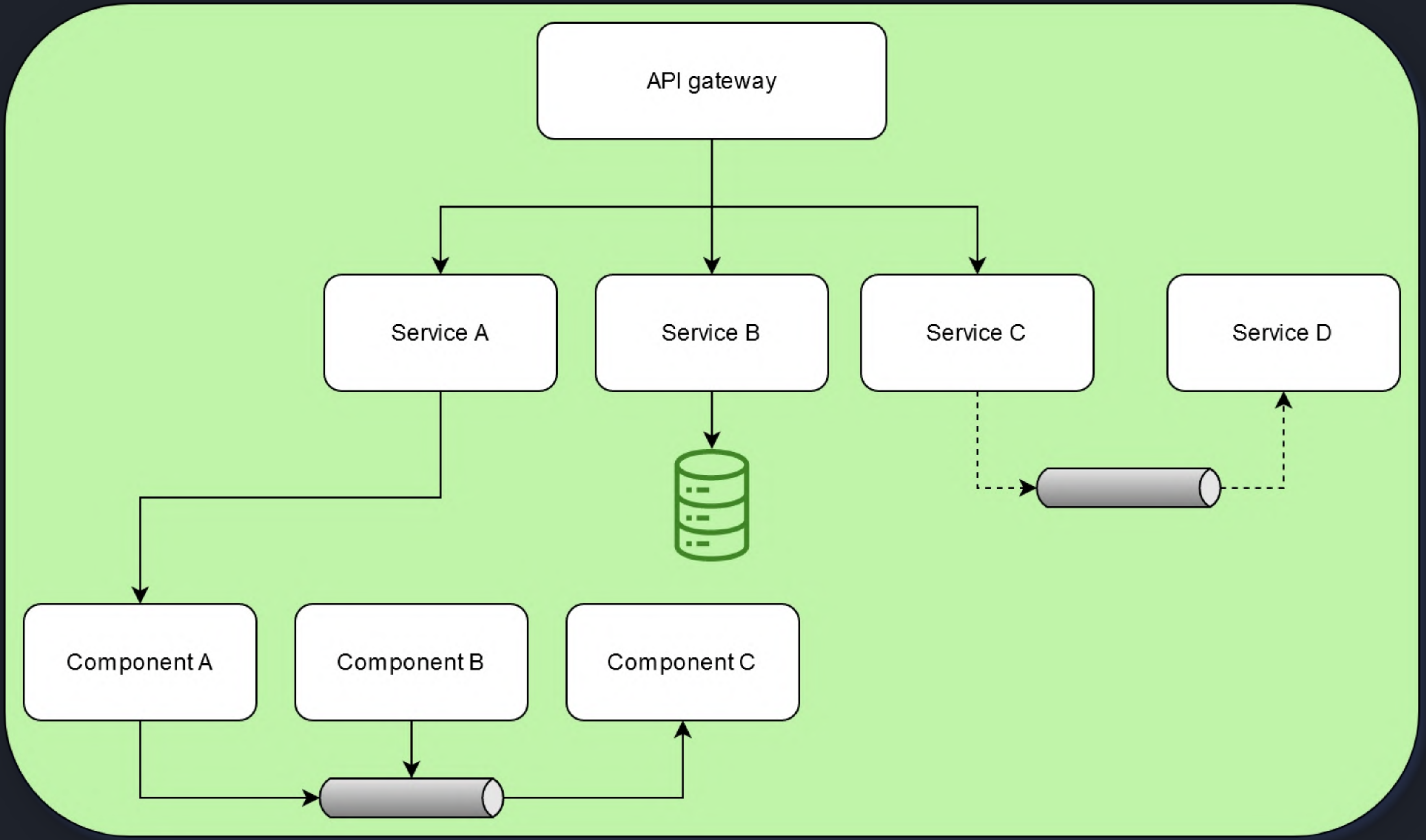
Microservices



Event-driven



Mixed architecture



NestJS = Easy EDA development*

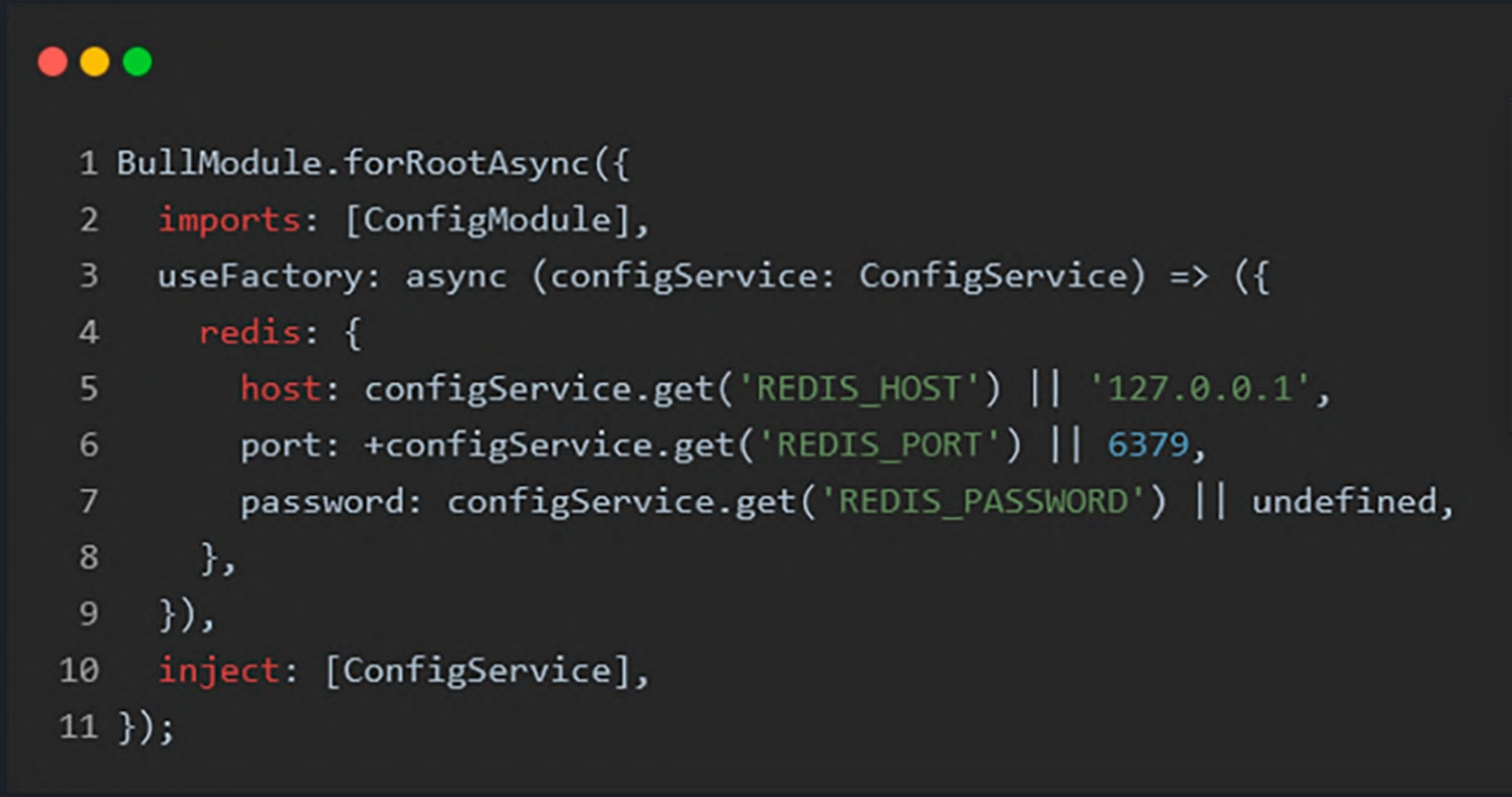
Redis, Kafka, RabbitMQ, MQTT, NATS

*event-driven architecture

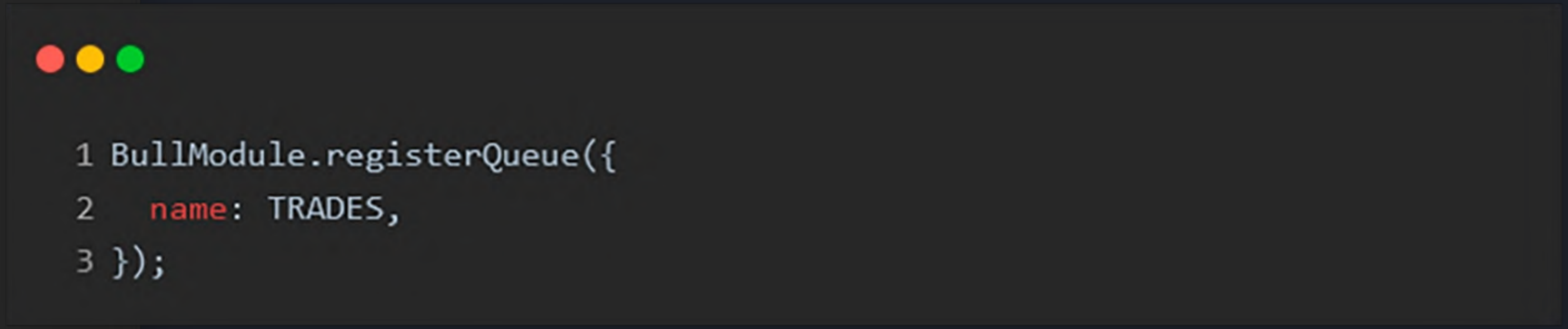
- Integrates with Redis/Bull for queue management github.com/OptimalBits/bull
- Integrates with messaging brokers
- Promotes modular development
- Great documentation and examples
- Unit and integration testing is bootstrapped (DI, Jest)

Queues: adding the connection (npm/bull)

```
$ npm install --save @nestjs/bull bull
$ npm install --save-dev @types/bull
```

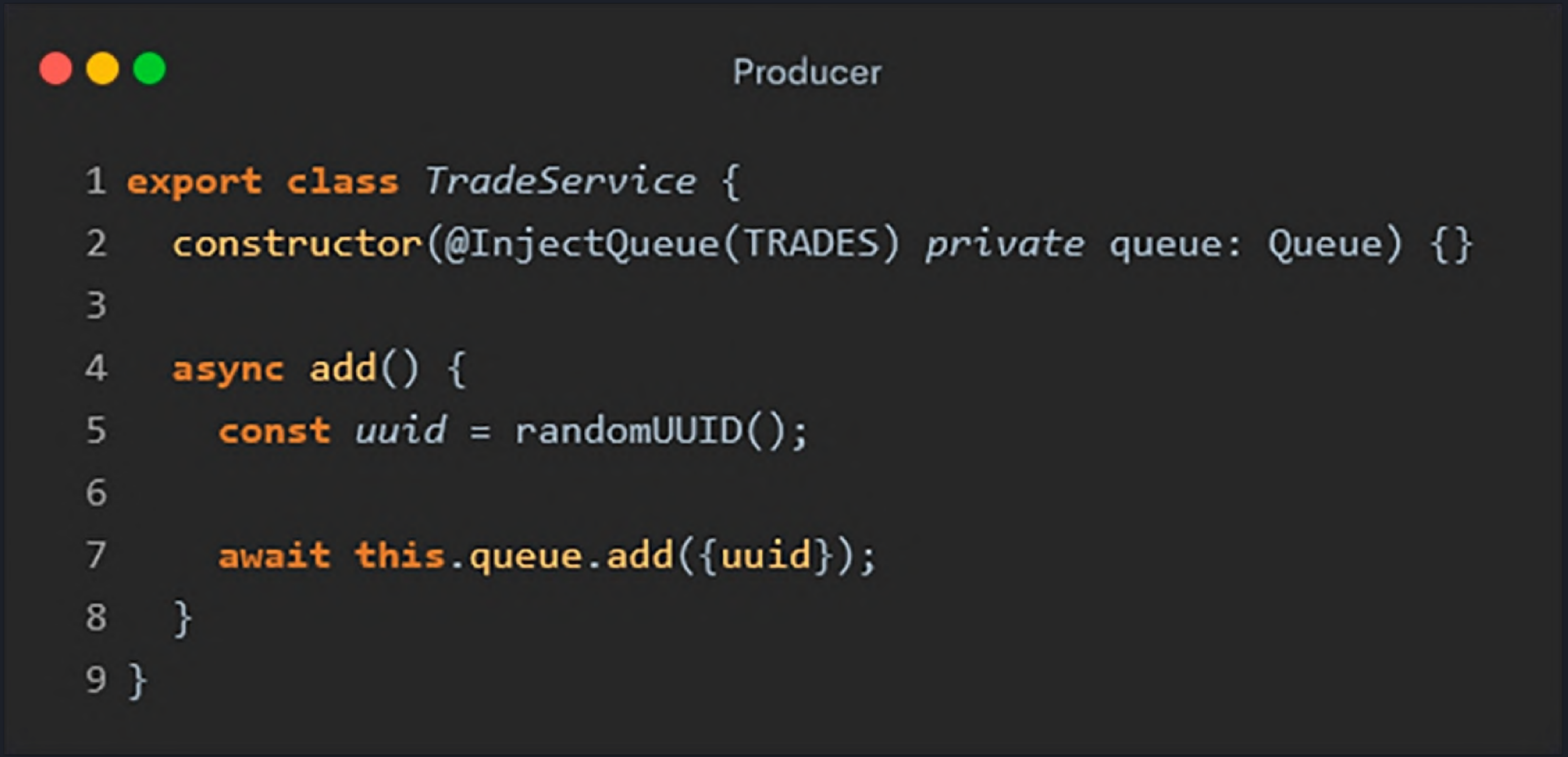


```
BullModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    redis: {
      host: configService.get('REDIS_HOST') || '127.0.0.1',
      port: +configService.get('REDIS_PORT') || 6379,
      password: configService.get('REDIS_PASSWORD') || undefined,
    },
  }),
  inject: [ConfigService],
});
```



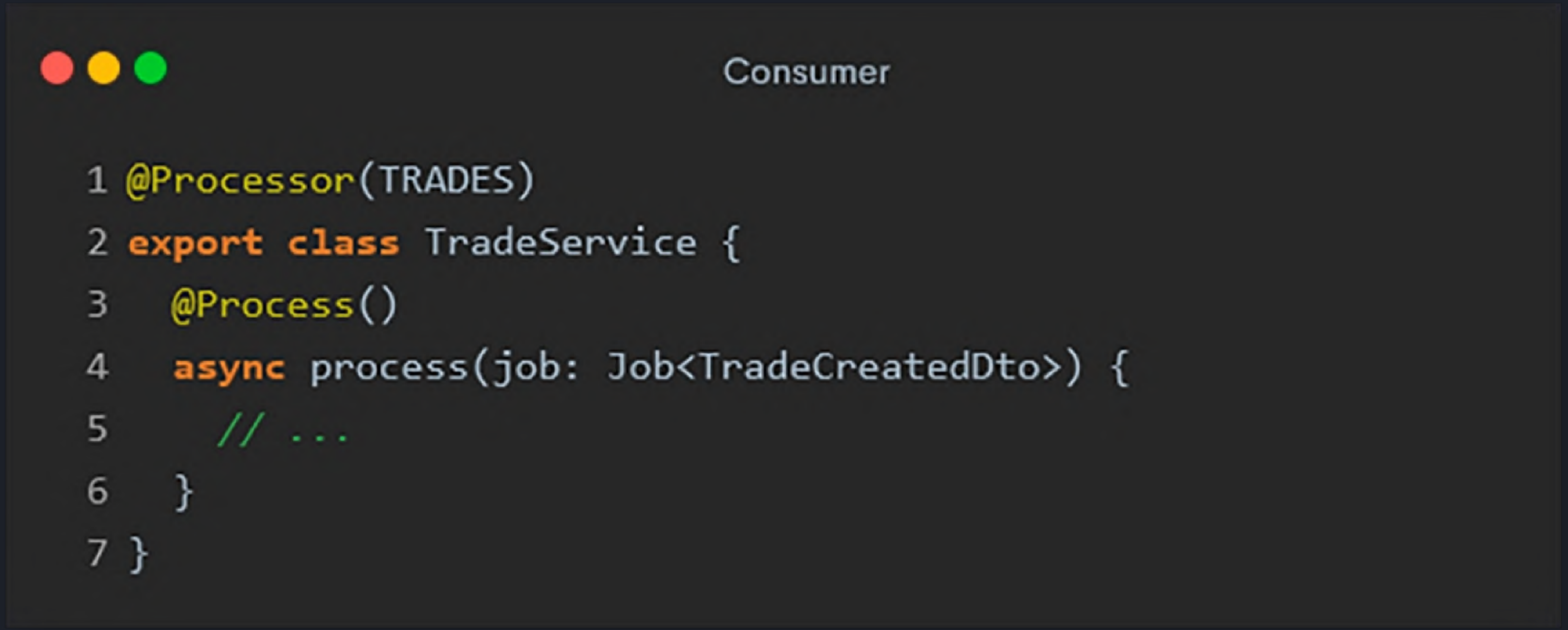
```
BullModule.registerQueue({
  name: TRADES,
});
```

Queues: event producer injects a queue



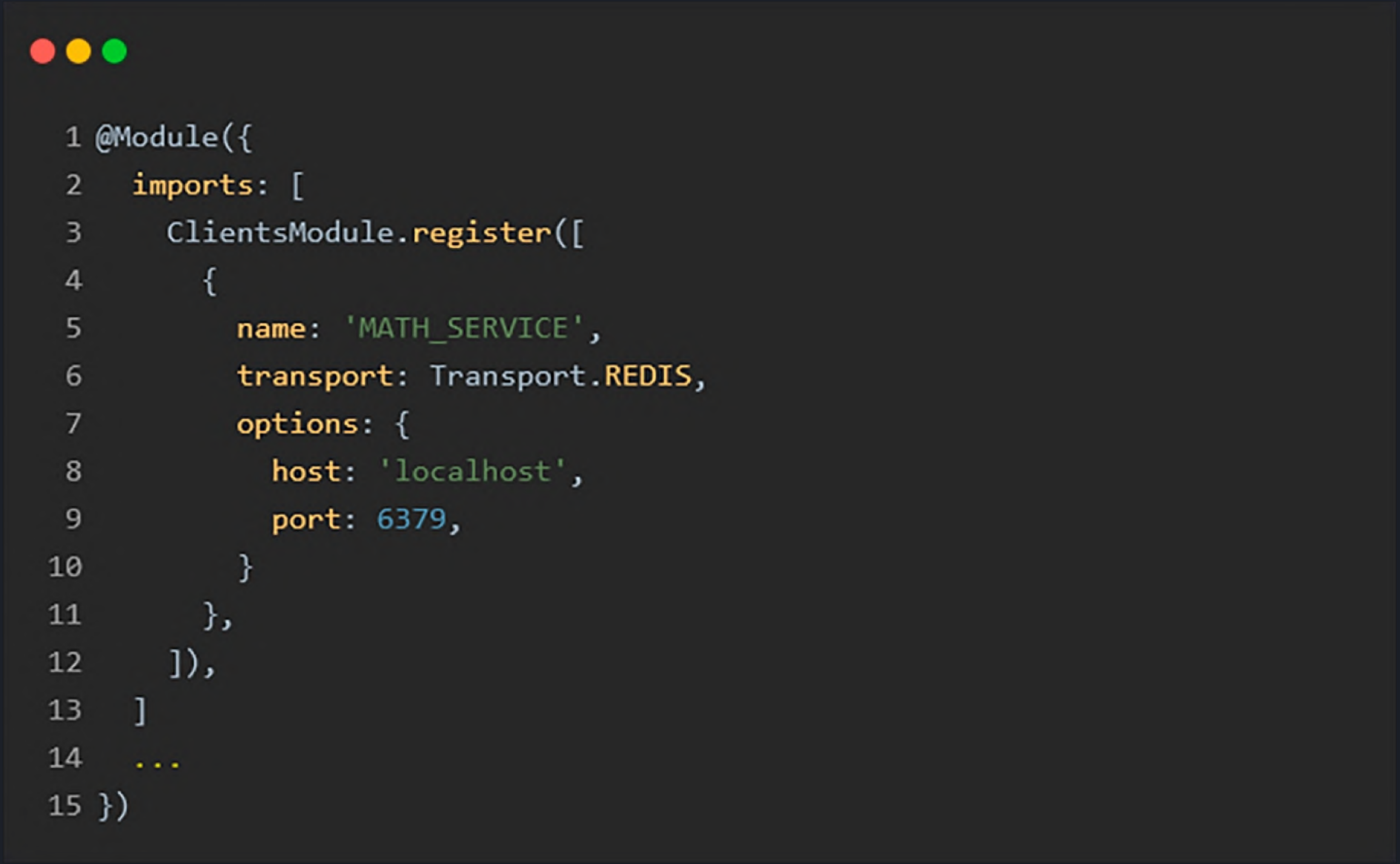
```
export class TradeService {
  constructor(@InjectQueue(TRADES) private queue: Queue) {}

  async add() {
    const uuid = randomUUID();

    await this.queue.add({uuid});
  }
}
```

Queues: event consumer processes the queue



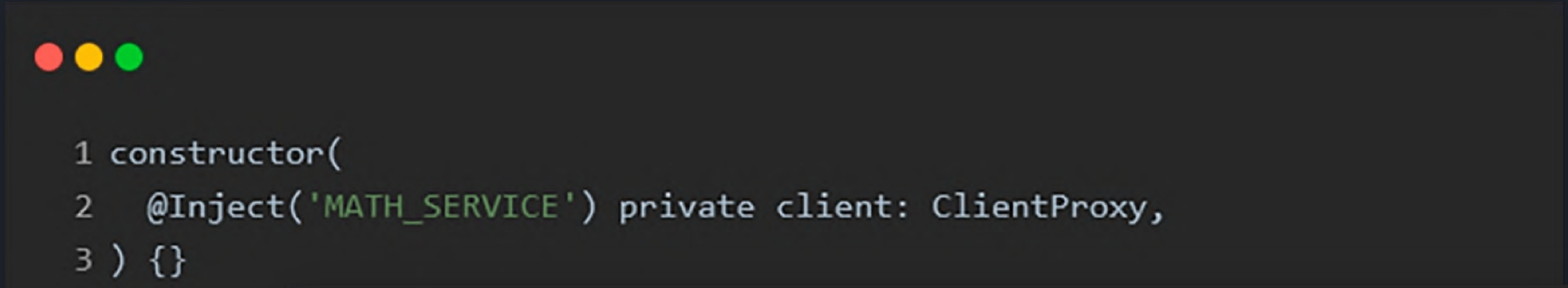
```
@Processor(TRADES)
export class TradeService {
  @Process()
  async process(job: Job<TradeCreatedDto>) {
    // ...
  }
}
```

Messaging integration - connection

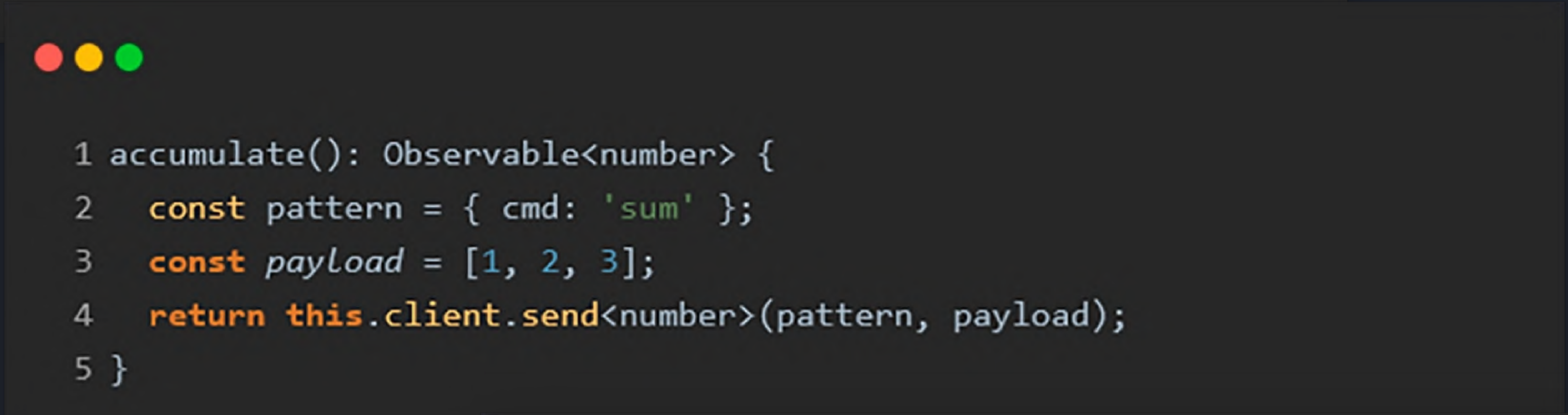


```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.REDIS,
        options: {
          host: 'localhost',
          port: 6379,
        }
      },
    ]),
  ]
  ...
})
```

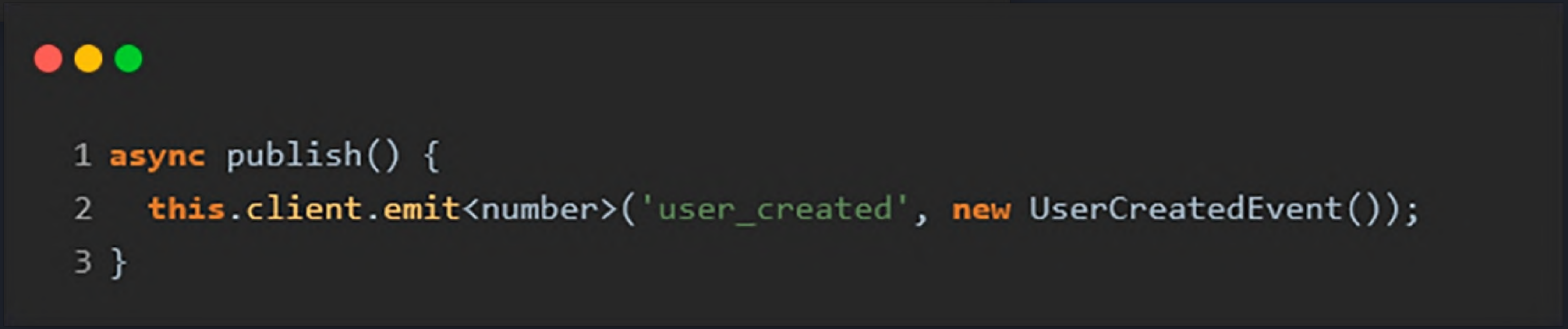
Messaging integration - producer



```
constructor(
  @Inject('MATH_SERVICE') private client: ClientProxy,
) {}
```

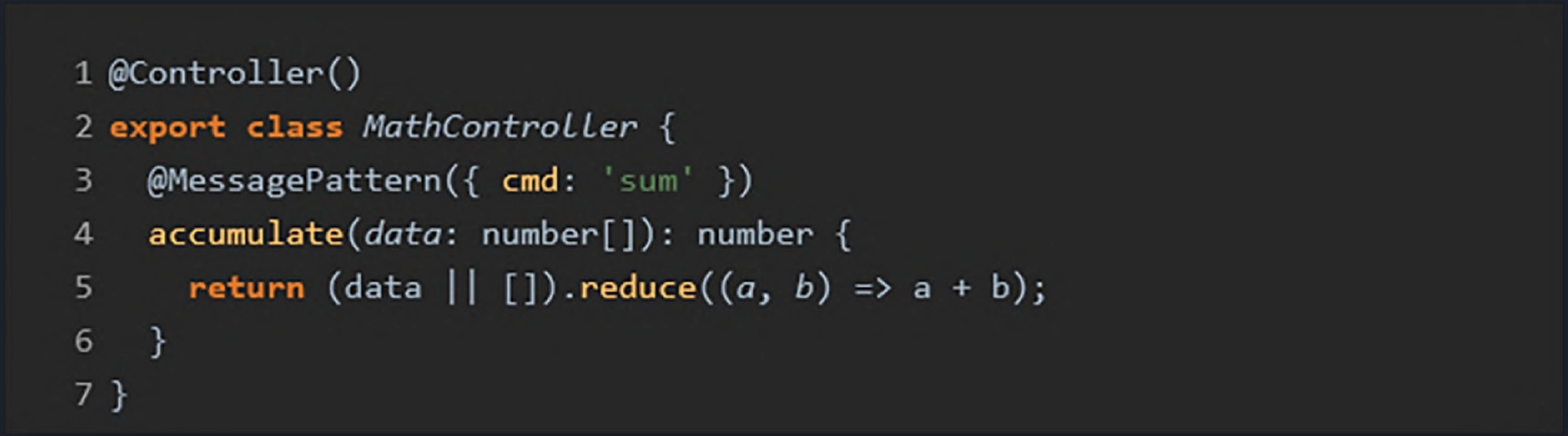


```
accumulate(): Observable<number> {
  const pattern = { cmd: 'sum' };
  const payload = [1, 2, 3];
  return this.client.send<number>(pattern, payload);
}
```

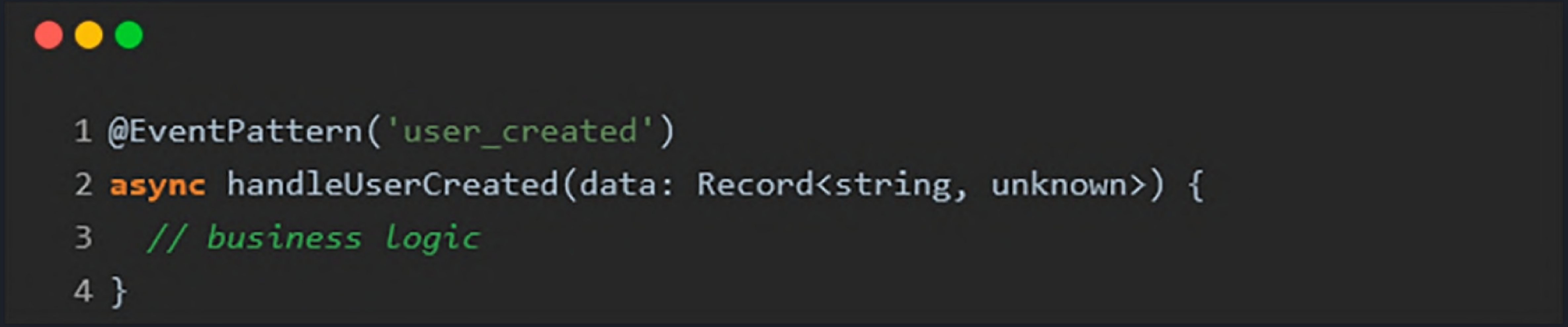


```
async publish() {
  this.client.emit<number>('user_created', new UserCreatedEvent());
}
```

Messaging integration - consumer

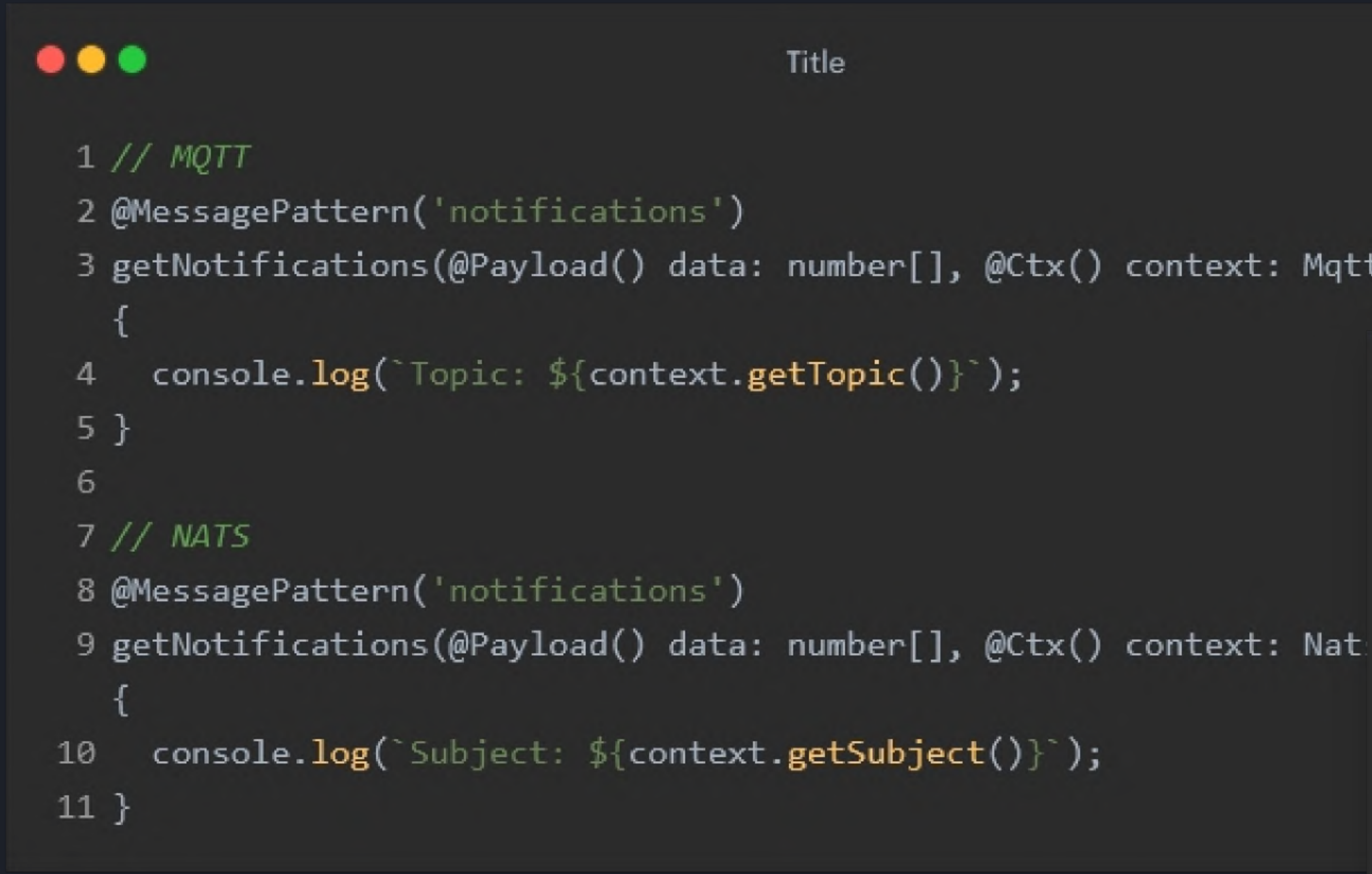


```
@Controller()
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  accumulate(data: number[]): number {
    return (data || []).reduce((a, b) => a + b);
  }
}
```

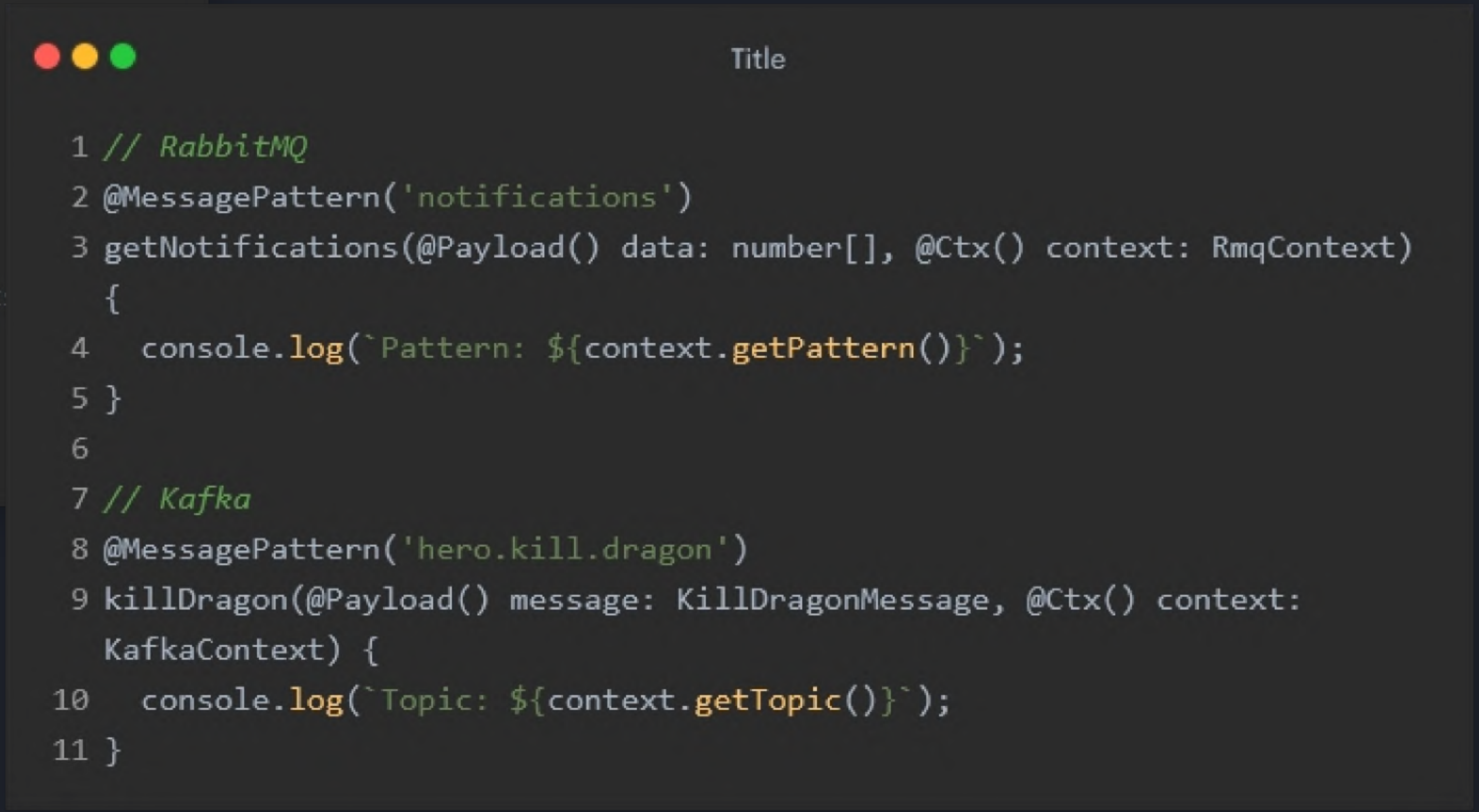


```
@EventPattern('user_created')
async handleUserCreated(data: Record<string, unknown>) {
  // business logic
}
```

Same for all other brokers



```
// MQTT
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: MqttContext)
{
  console.log(`Topic: ${context.getTopic()}`);
}

// NATS
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: Nat
{
  console.log(`Subject: ${context.getSubject()}`);
}
```



```
// RabbitMQ
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext)
{
  console.log(`Pattern: ${context.getPattern()}`);
}

// Kafka
@MessagePattern('hero.kill.dragon')
killDragon(@Payload() message: KillDragonMessage, @Ctx() context:
KafkaContext) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

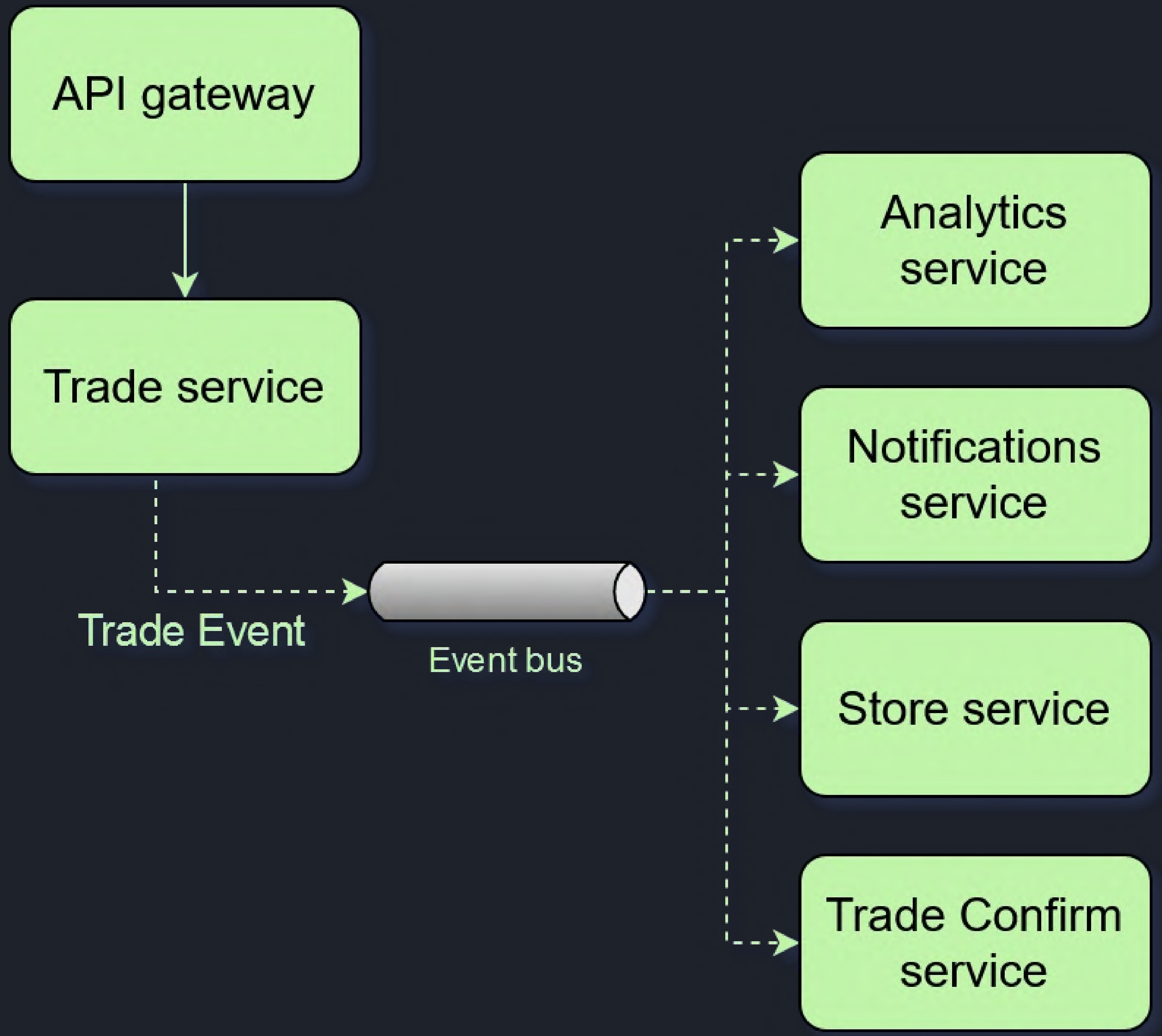
Pitch

Demo app and tools

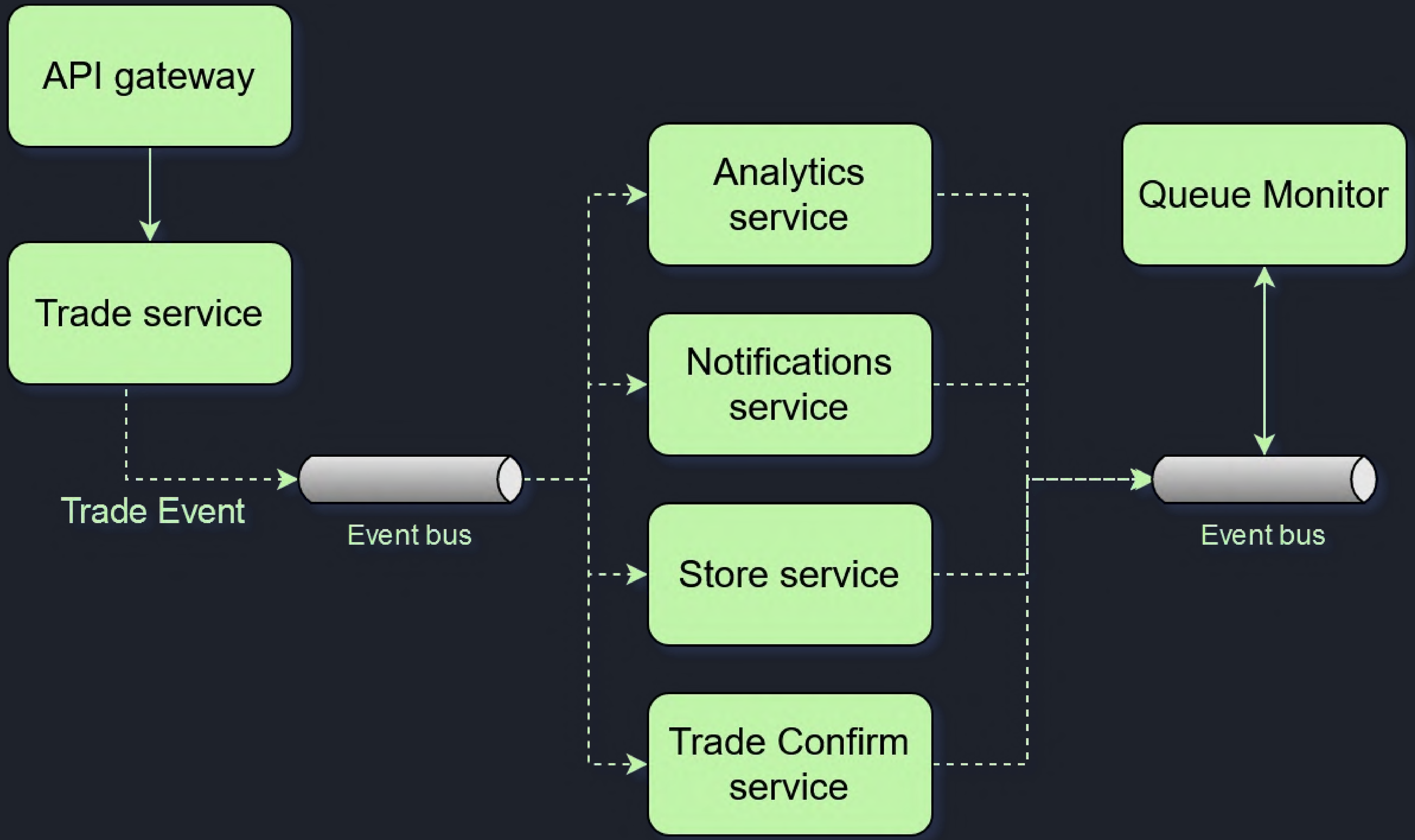
AVAILABLE AT GITHUB

github.com/dkhorev/conf42-event-driven-nestjs-demo

Demo app overview



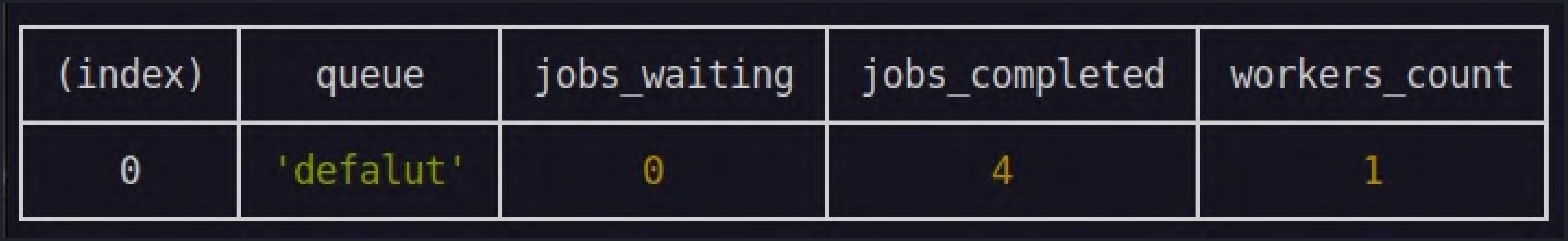
Demo app overview



Pitch

Demo app in action - normal conditions

```
$ make start
$ make monitor
```

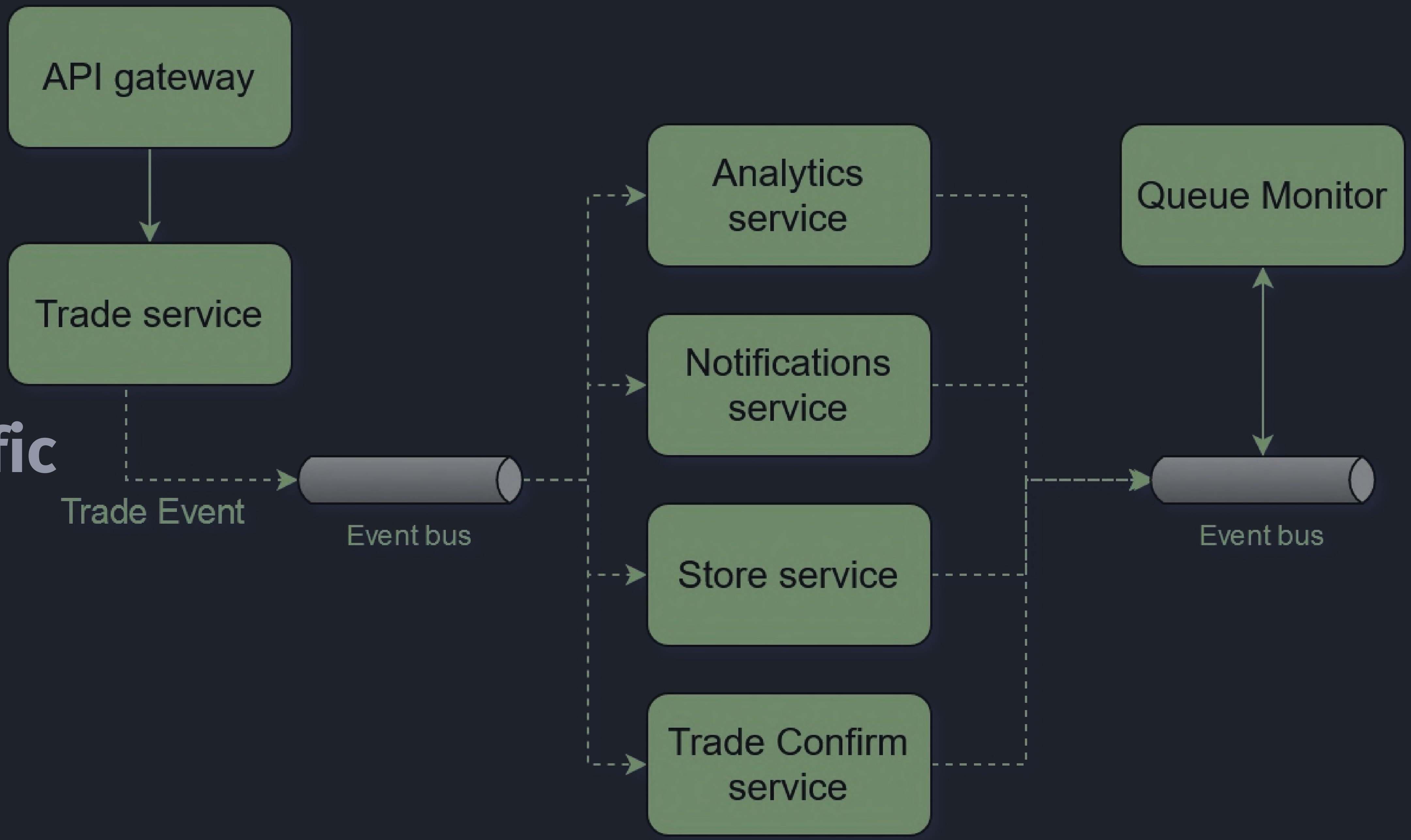


(index)	queue	jobs_waiting	jobs_completed	workers_count
0	'defalut'	0	4	1

Demo app in action - normal conditions

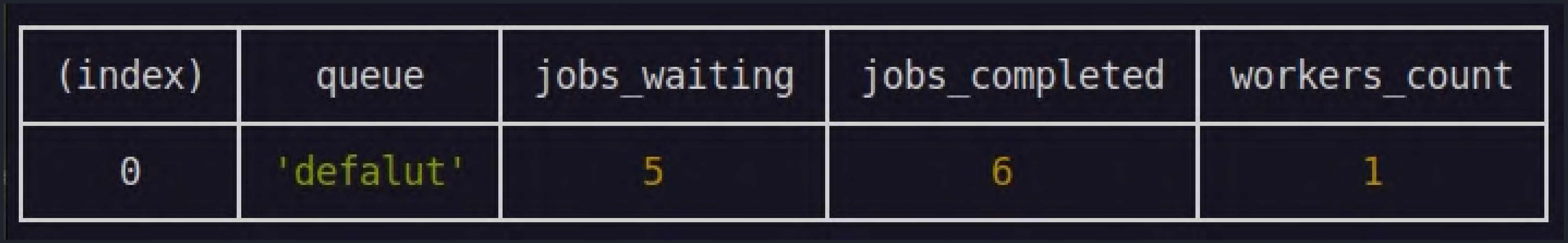
Works fine with low trades/minute.

But what if suddenly traffic to our app increases?

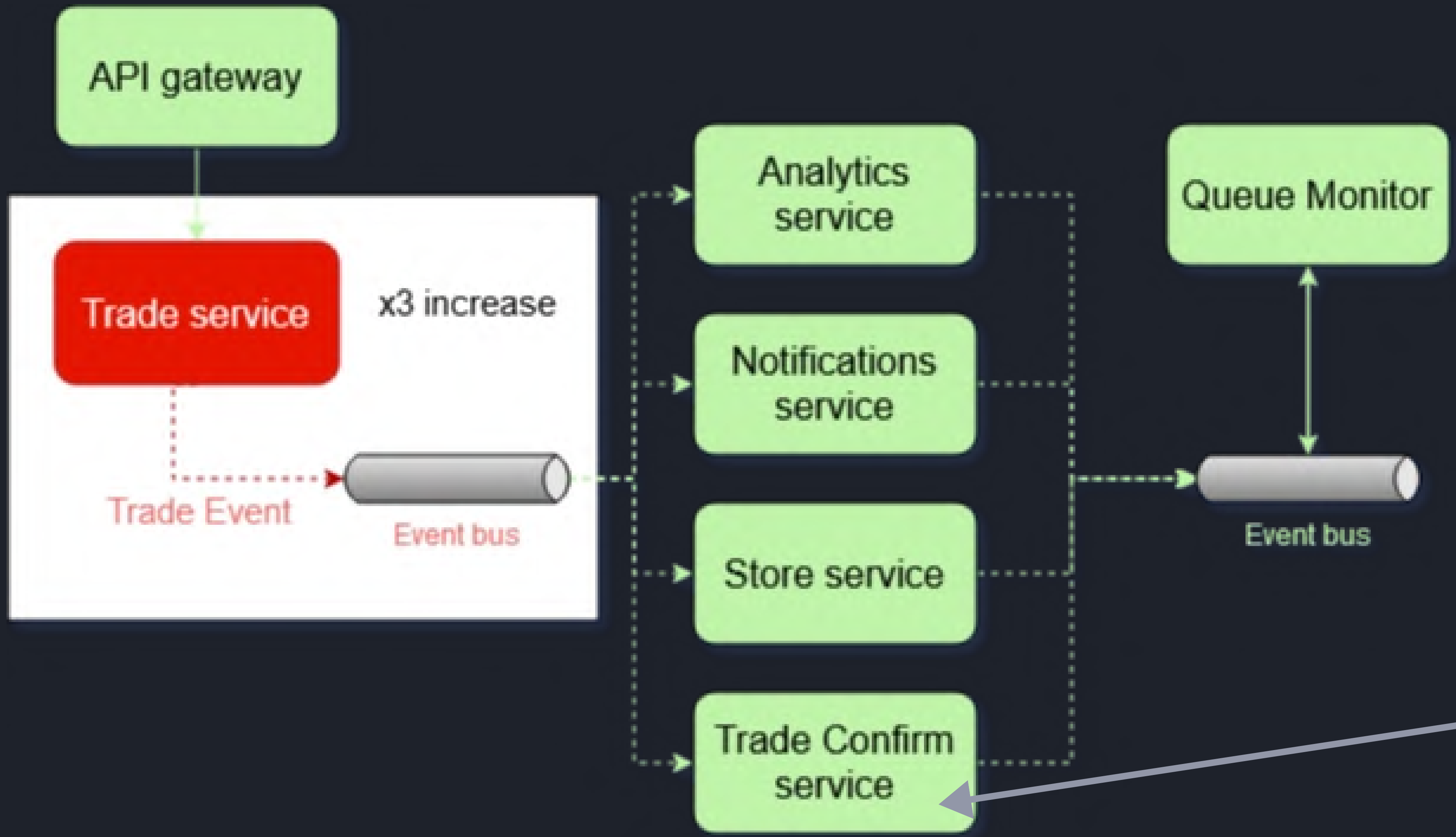


Demo app in action - traffic spike

```
$ make start-issue1
$ make monitor
```



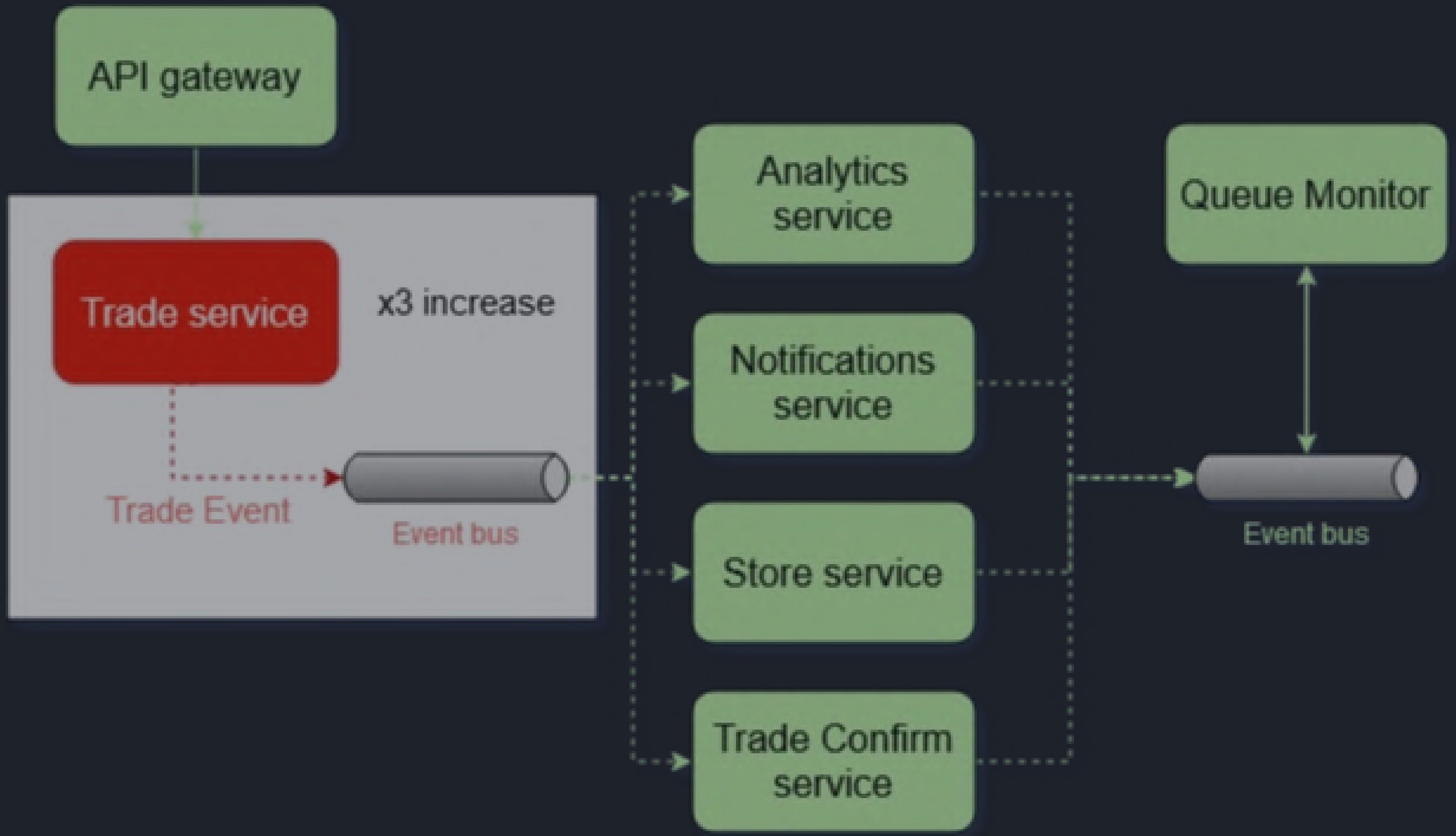
```
┌─────────┬───────────┬──────────────┬────────────────┬───────────────┐
│ (index) │   queue   │ jobs_waiting │ jobs_completed │ workers_count │
├─────────┼───────────┼──────────────┼────────────────┼───────────────┤
│    0    │ 'defalut' │      5       │       6        │       1       │
└─────────┴───────────┴──────────────┴────────────────┴───────────────┘
```



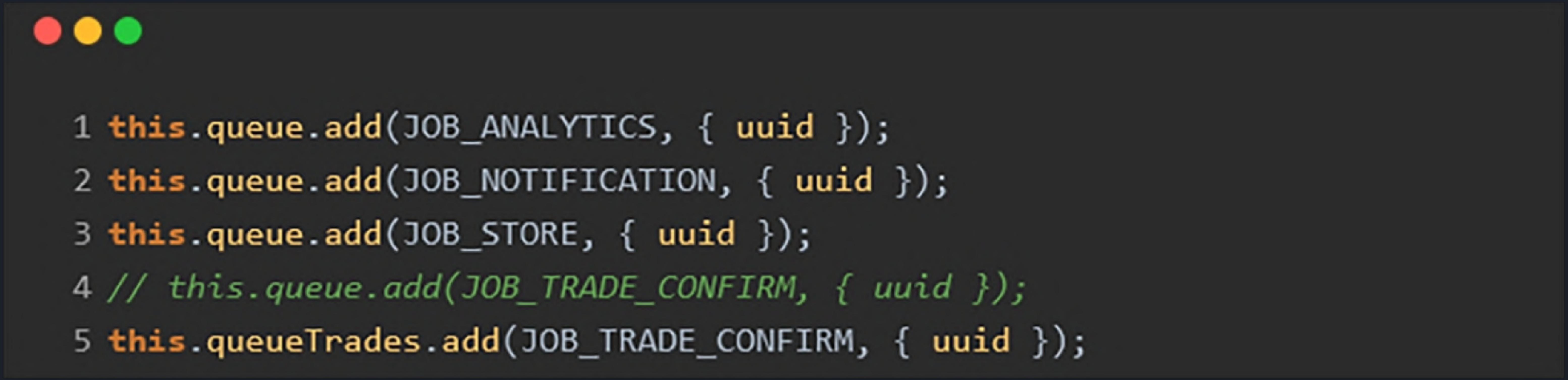
mission-critical

Solutions?

- Scale the worker instance so it will process queue faster
- Increase worker instance count
- Application optimizations
- Separate the queues
- Prioritize events

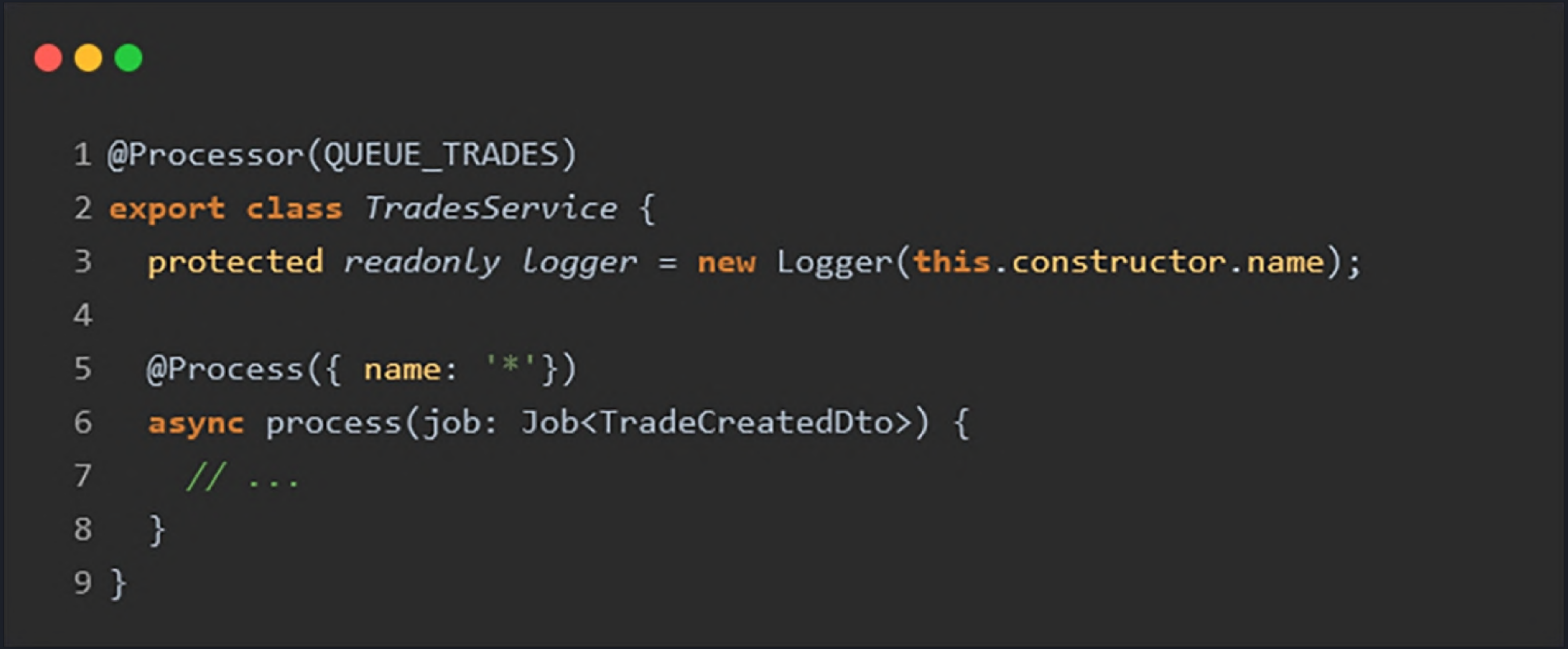


Step 1 - separate the queues (producer)



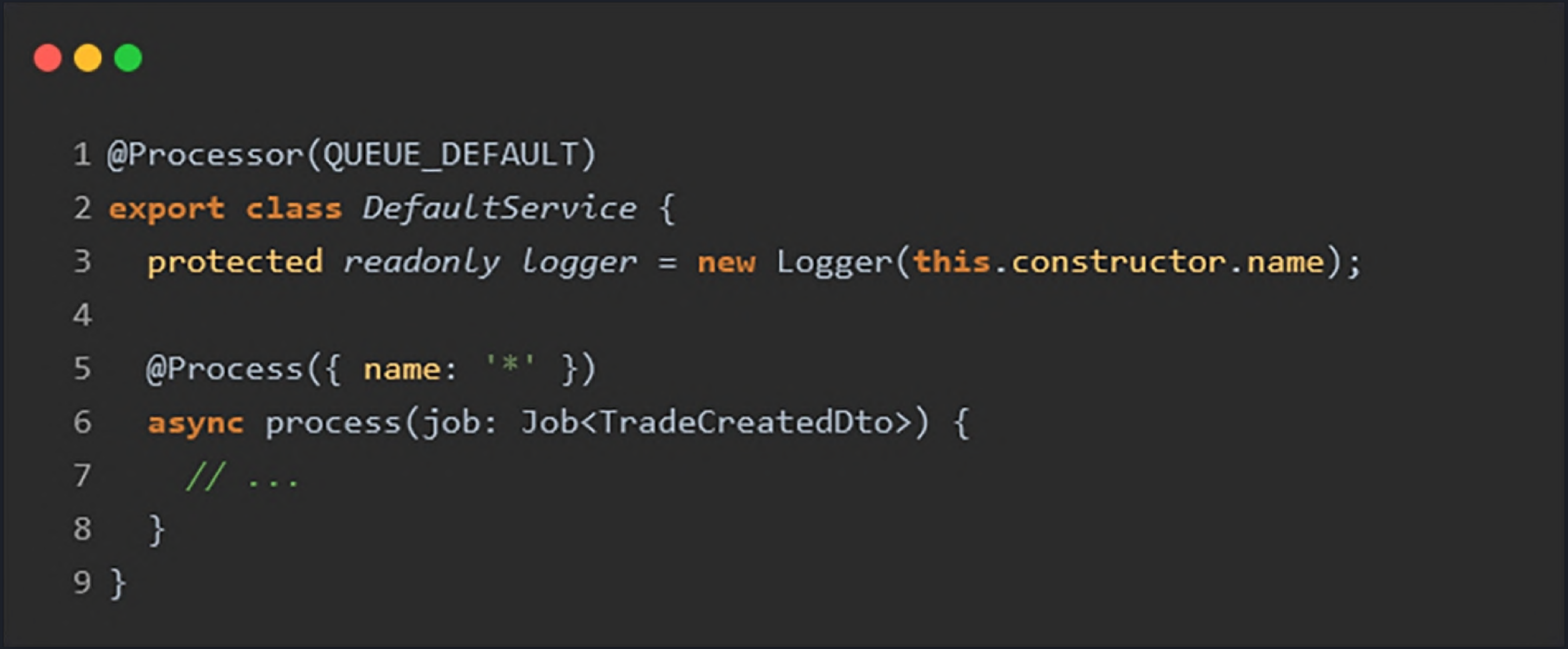
```
this.queue.add(JOB_ANALYTICS, { uuid });
this.queue.add(JOB_NOTIFICATION, { uuid });
this.queue.add(JOB_STORE, { uuid });
// this.queue.add(JOB_TRADE_CONFIRM, { uuid });
this.queueTrades.add(JOB_TRADE_CONFIRM, { uuid });
```

Step 1 - separate the queues (consumer)



```
@Processor(QUEUE_TRADES)
export class TradesService {
  protected readonly logger = new Logger(this.constructor.name);

  @Process({ name: '*'})
  async process(job: Job<TradeCreatedDto>) {
    // ...
  }
}
```

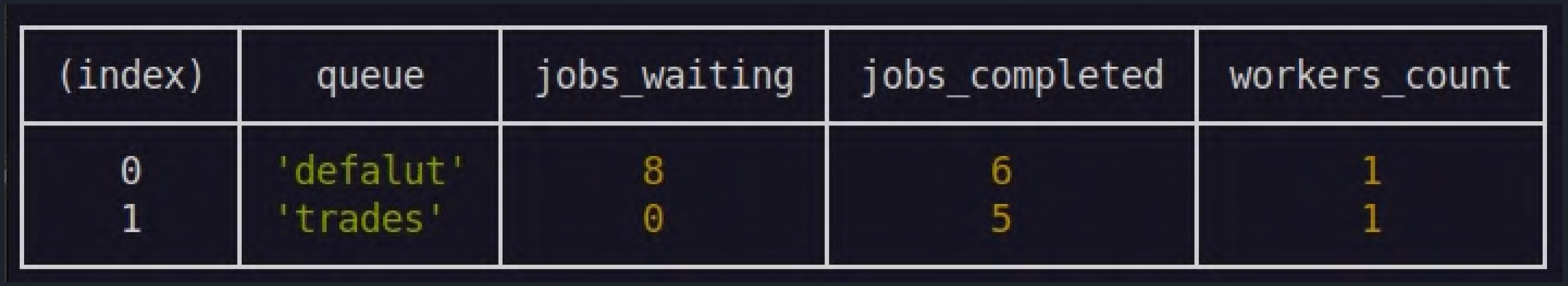
Step 1 - separate the queues (consumer)



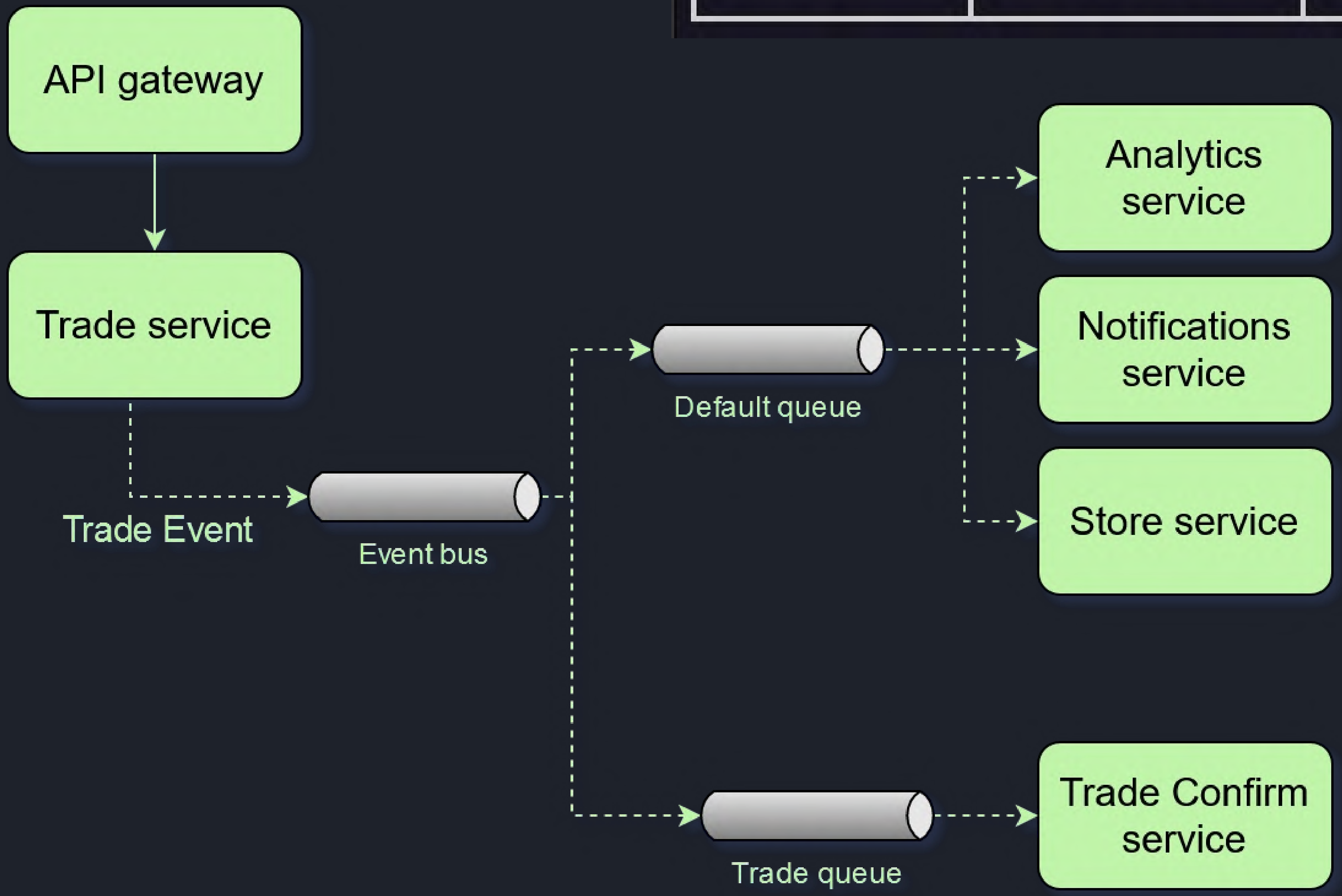
```
@Processor(QUEUE_DEFAULT)
export class DefaultService {
  protected readonly logger = new Logger(this.constructor.name);

  @Process({ name: '*' })
  async process(job: Job<TradeCreatedDto>) {
    // ...
  }
}
```

Step 1 - separate the queues

```
$ make start-step1
$ make monitor
```

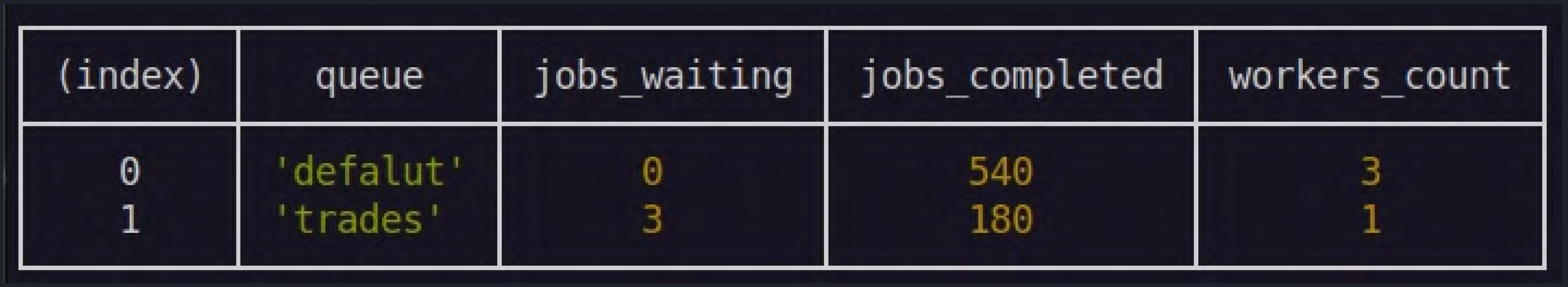


```
┌─────────┬───────────┬──────────────┬────────────────┬───────────────┐
│ (index) │   queue   │ jobs_waiting │ jobs_completed │ workers_count │
├─────────┼───────────┼──────────────┼────────────────┼───────────────┤
│    0    │ 'defalut' │      8       │       6        │       1       │
│    1    │ 'trades'  │      0       │       5        │       1       │
└─────────┴───────────┴──────────────┴────────────────┴───────────────┘
```

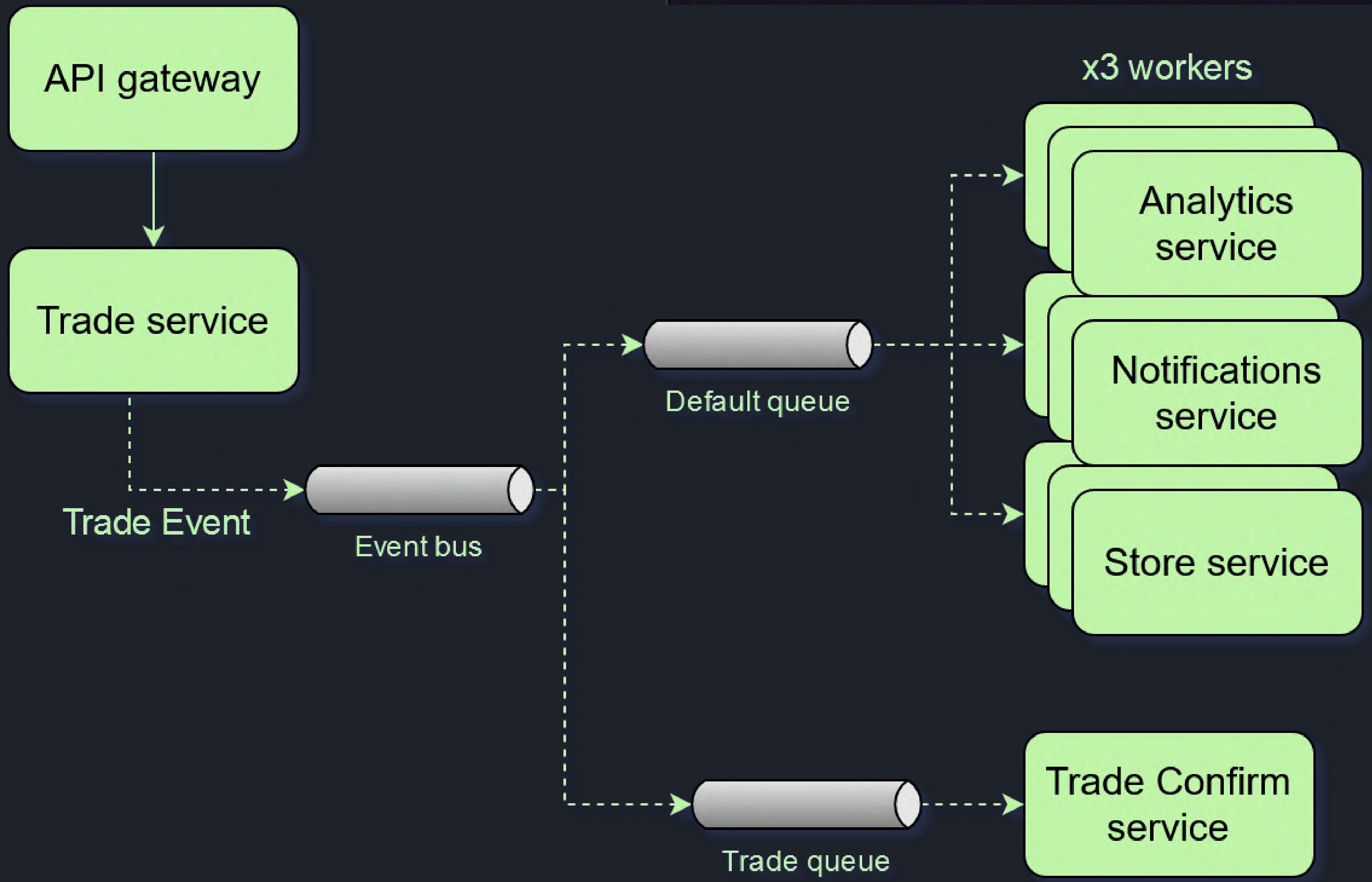


Step 2 - scale workers

```
$ make start-step2
$ make monitor
```

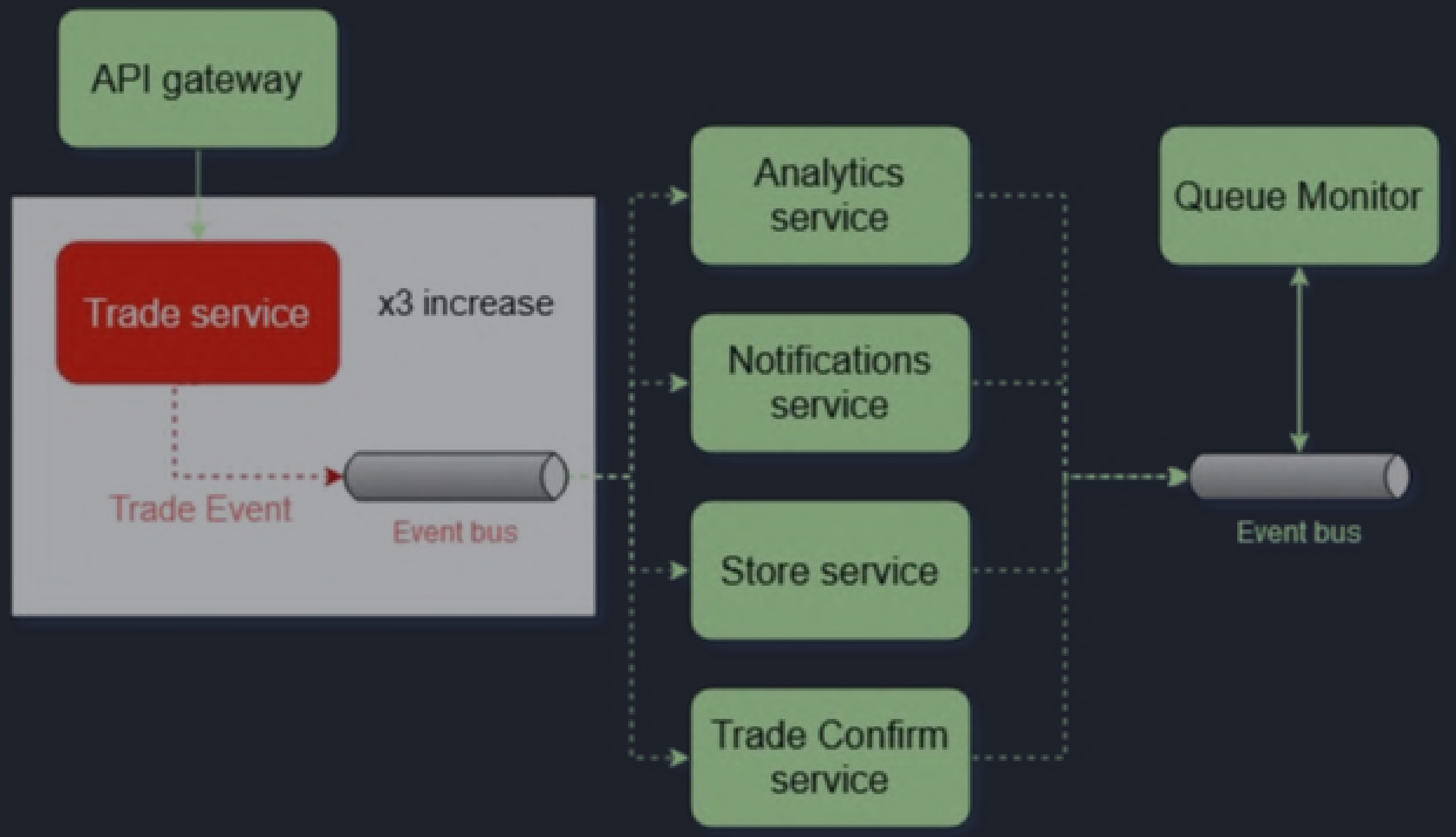


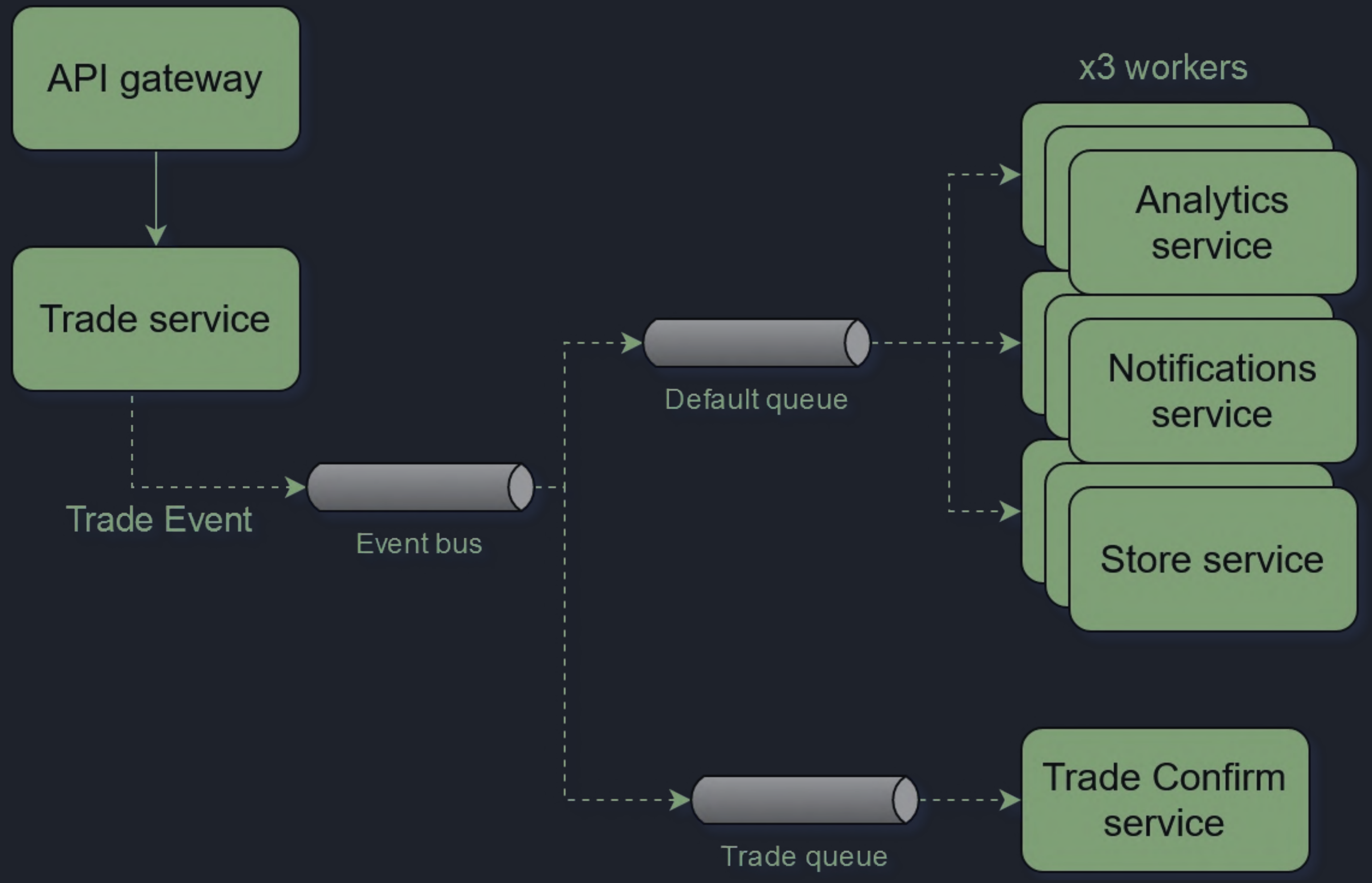
```
┌─────────┬───────────┬──────────────┬────────────────┬───────────────┐
│ (index) │   queue   │ jobs_waiting │ jobs_completed │ workers_count │
├─────────┼───────────┼──────────────┼────────────────┼───────────────┤
│    0    │ 'defalut' │      0       │      540       │       3       │
│    1    │ 'trades'  │      3       │      180       │       1       │
└─────────┴───────────┴──────────────┴────────────────┴───────────────┘
```



Solutions - recap

- Scale the worker instance so it will process queue faster
- **Increase worker instance count**
- Application optimizations
- **Separate queues**
- Prioritize events





Thank you

AND HAPPY CODING!

Demo app repo

github.com/dkhorev/conf42-event-driven-nestjs-demo

Feel free to connect via LinkedIn

linkedin.com/in/dmitry-khorev