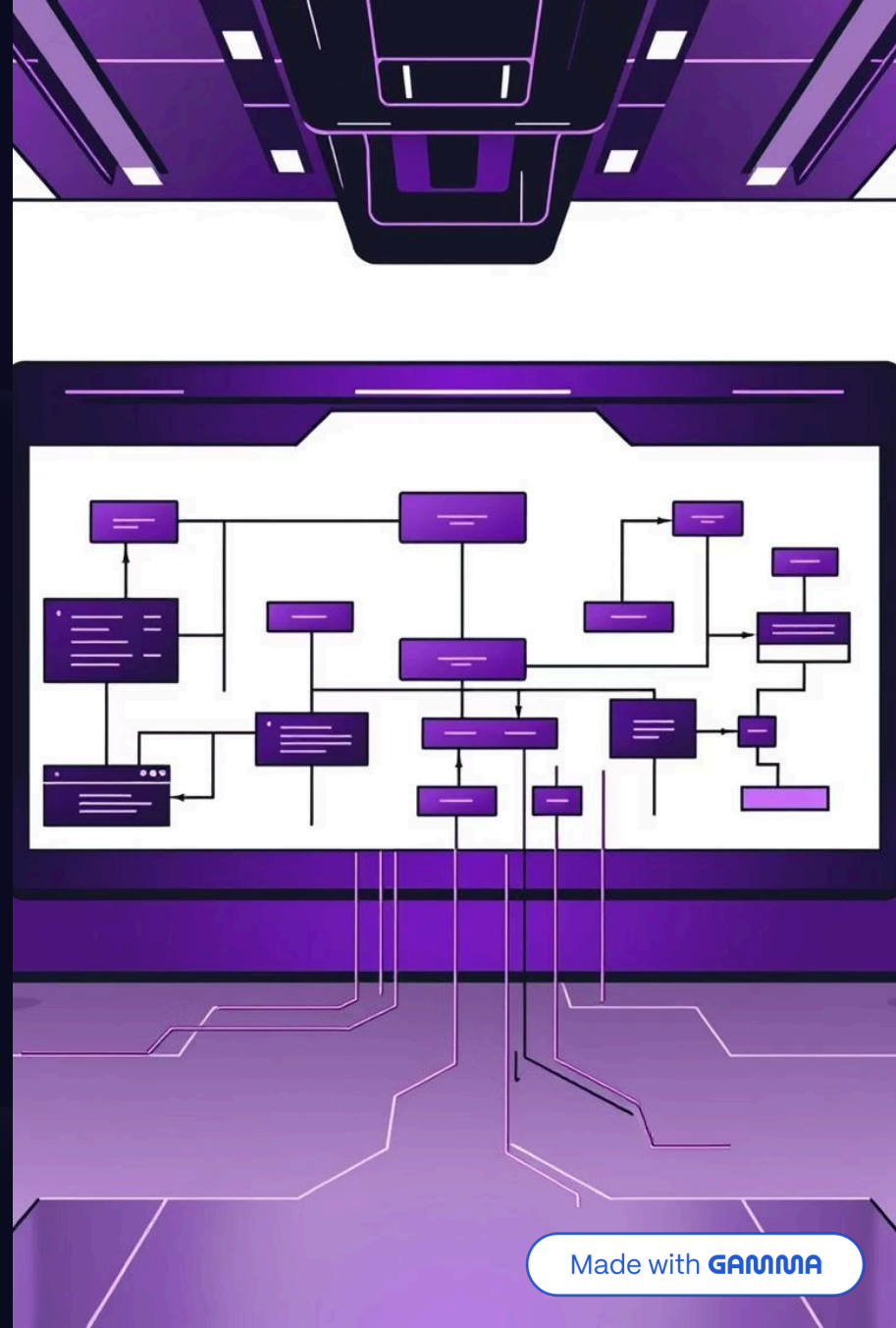
Design Patterns Identificados no Projeto

Uma análise completa dos padrões de design implementados em nossa arquitetura de sistema, demonstrando as melhores práticas para sistemas distribuídos, resilientes e escaláveis.



Padrões de Criação

1

Dependency Injection

Implementado extensivamente no NestJS através de decorators:

```
constructor(
  private readonly
batchProcessingService:
BatchProcessingService,
  private readonly configService:
ConfigService,
) {
  this.maxRetries =
this.configService.get('MAX_RET
RY_ATTEMPTS', 3);
}
```

Benefícios: Desacoplamento entre componentes, facilita testes unitários e inversão de controle.

2

Factory Pattern

Implementado através dos métodos forRoot(), forRootAsync() e forFeature():

```
TypeOrmModule.forRootAsync({
  inject: [ConfigService],
  useFactory: (configService:
ConfigService) => ({
    type: 'postgres',
    host:
configService.get('DB_HOST'),
    // outras configurações
  })
})
```

3

Singleton Pattern

Serviços NestJS são singleton por padrão:

```
@Injectable()
export class RabbitMQService
implements OnModuleInit,
OnModuleDestroy {
 // implementação
}
```

Padrões Estruturais

Module Pattern

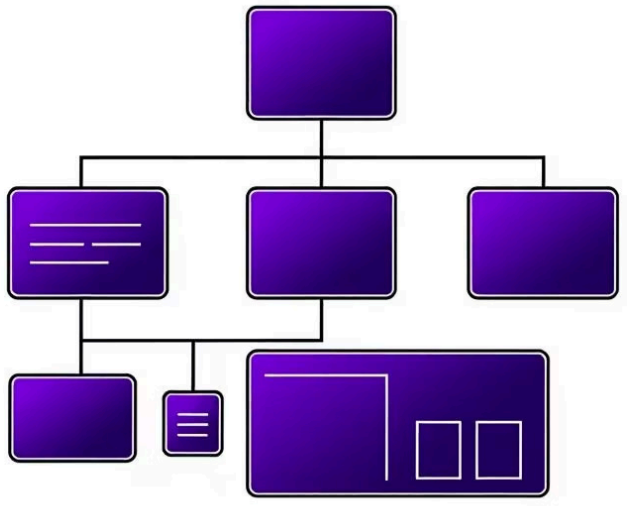
Organização em módulos NestJS:

```
@Module({
  imports: [RabbitMQModule,
BatchProcessingModule],
  controllers: [EventsController],
  providers: [EventsConsumer],
})
export class EventsModule {}
```

Facade Pattern

Controllers como fachada para a lógica de negócio:

```
@Post()
@HttpCode(HttpStatus.ACCEPTED)
async createEvent(@Body() createEventDto:
CreateEventDto) {
  // implementação que oculta complexidade
}
```



O padrão Repository também é implementado através do TypeORM Repository para acesso aos dados:

```
constructor(
 @InjectRepository(UserInteraction)
 private userInteractionRepository: Repository,
 private dataSource: DataSource,
 // ...
)
```

Padrões Comportamentais

Observer Pattern

Event Emitter para eventos internos:

```
this.eventEmitter.emit('batch.processed', {
  count: eventsToProcess.length,
  timestamp: new Date(),
  // outros dados
});
```

Strategy Pattern

Duas estratégias para processamento de lote:

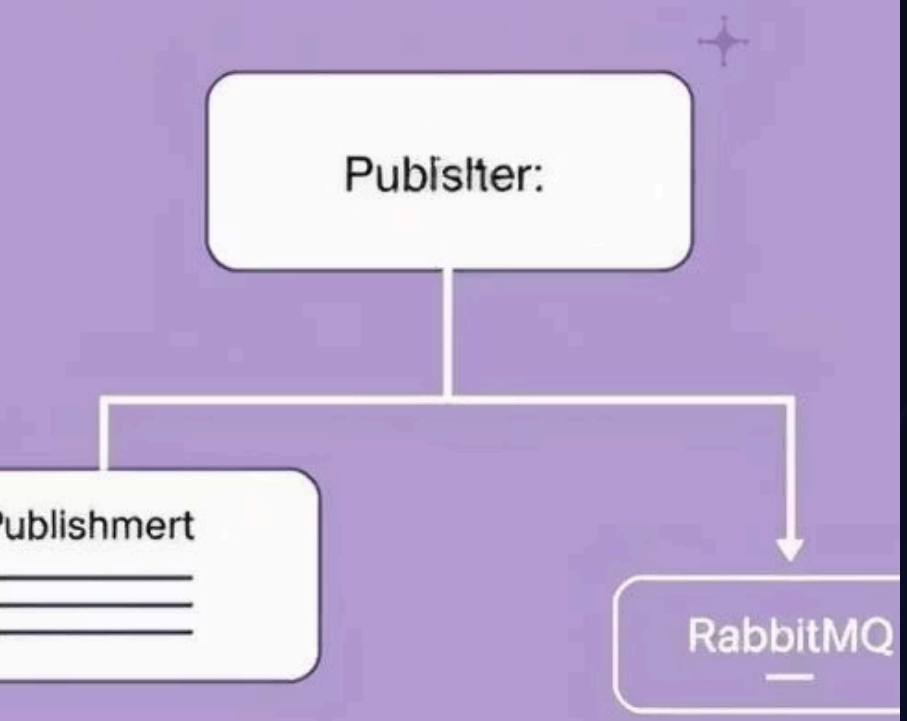
- Por tamanho máximo do buffer
- Por tempo (agendamento)

Template Method

Lifecycle hooks do NestJS:

```
async onModuleInit() {
  try {
    await this.connect();
    await this.startConsumer();
  } catch (error) {
    this.logger.error('Initialization failed', error);
  }
}
```

Outros padrões comportamentais identificados incluem Chain of Responsibility (middleware e interceptors do NestJS) e Command Pattern (DTOs como comandos).



Padrões de Mensageria

Publisher-Subscriber

RabbitMQ como message broker:

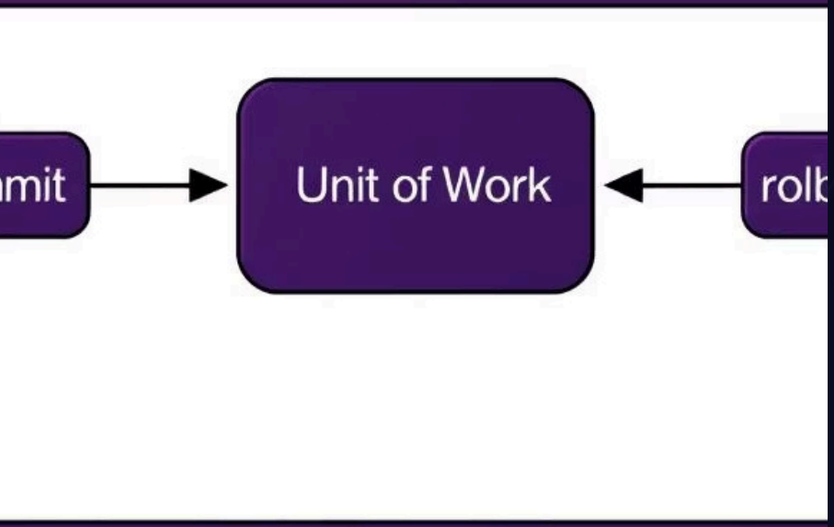
```
async publishEvent(event: any): Promise {
  try {
    await this.channelWrapper.sendToQueue(

`${this.configService.get('RABBITMQ_QUEUE_NAME')}`,

Buffer.from(JSON.stringify(event)),
    );
  } catch (error) {
    this.logger.error('Error when publish event', error);
    throw error;
  }
}
```

Consumer Pattern

EventsConsumer consumindo mensagens da fila, com confirmação manual (ack/nack) para garantir processamento.

Dead Letter Queue

Fila para mensagens com falha de processamento, permitindo análise posterior e evitando perda de dados.



Padrões de Persistência

Unit of Work Pattern

Implementado através de transações com QueryRunner:

```
const queryRunner =
this.dataSource.createQueryRunner();
await queryRunner.connect();
await queryRunner.startTransaction();
try {
 // operações de banco de dados
 await
queryRunner.manager.save(UserInter
action, userInteractions, {
   chunk: 50,
 });
 await
queryRunner.commitTransaction();
} catch (error) {
 await
queryRunner.rollbackTransaction();
} finally {
 await queryRunner.release();
}
```

Data Mapper Pattern

Implementado através de TypeORM Entities:

```
@Entity('user_interact
ions')
@Index(['userId',
'eventType'])
export class
UserInteraction {

@PrimaryGeneratedC
olumn()
 id: number;

 @Column()
 userId: string;

 // outros campos
}
```

Padrões de Resiliência

Retry Pattern

Implementação de retry com limite de tentativas:

```
const retryCount = (message.properties.headers?.['x-retry-count'] || 0) + 1;
if (retryCount <= this.maxRetries) {
// Rejeita e recoloca na fila para nova tentativa
channel.nack(message, false, true);
} else {
// Envia para Dead Letter Queue
channel.nack(message, false, false);
}
```

Bulkhead Pattern

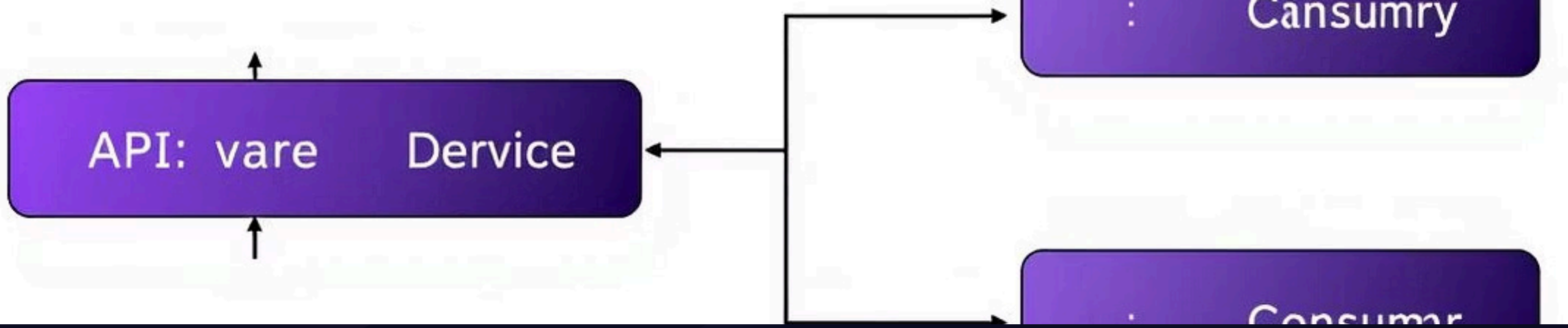
Isolamento de recursos com buffer para cada instância:

```
private eventBuffer: any[] = [];
private readonly maxBatchSize = 100;
private isProcessing = false;
```

Circuit Breaker (Implícito)

Flag isProcessing previne sobrecarga:

```
private async processBatch(): Promise {
 if (this.isProcessing || this.eventBuffer.length === 0) {
 return;
 }
 this.isProcessing = true;
 // processamento
}
```



Padrões de Integração e Concorrência

Padrões de Integração

- **Gateway Pattern:** EventsController como API Gateway
- **Message Channel Pattern:** RabbitMQ como canal de mensagens
- **Message Router Pattern:** Roteamento baseado no tipo de evento

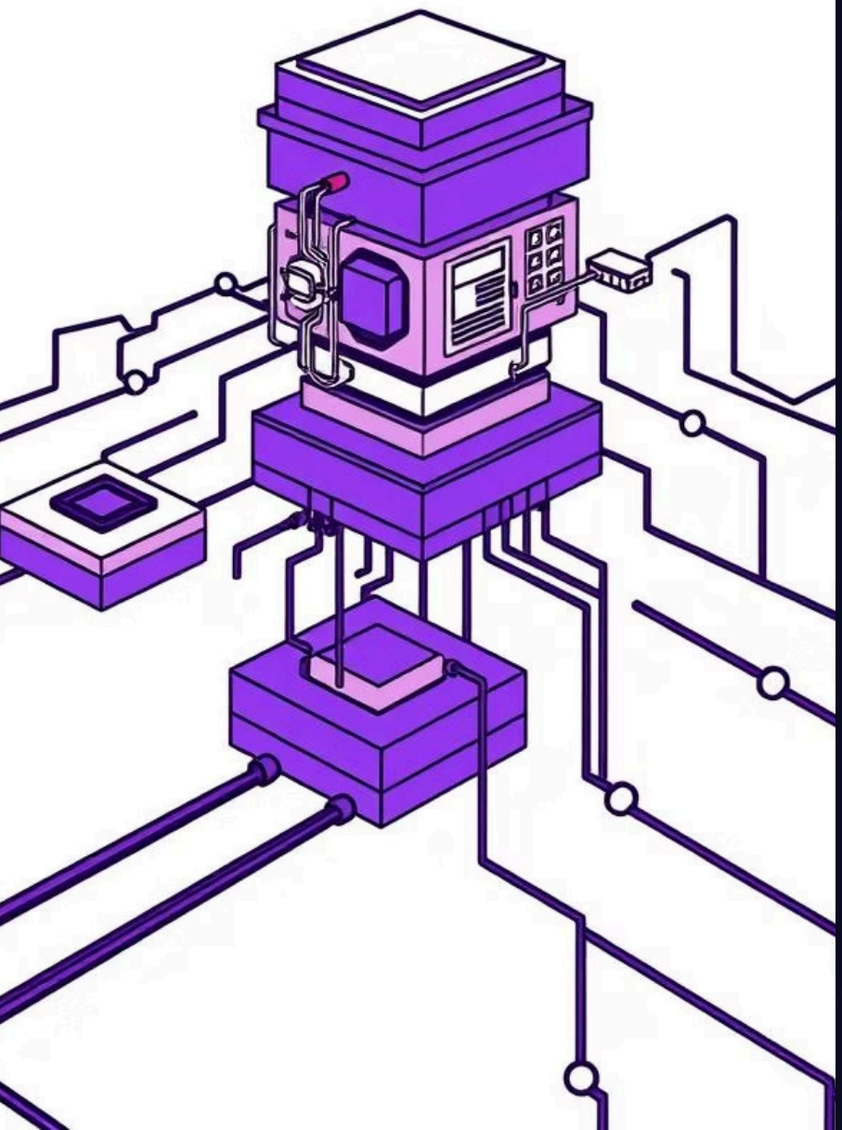
Padrões de Concorrência

- **Producer-Consumer Pattern:** API produz eventos, Consumer os processa
- **Batch Processing Pattern:** Agrupamento de eventos para processamento eficiente

Estes padrões trabalham em conjunto para criar um sistema distribuído que pode escalar horizontalmente e processar grandes volumes de eventos de forma eficiente.

Resumo dos Design Patterns

Categoria	Pattern	Implementação
Criação	Dependency Injection	NestJS DI Container
Criação	Factory	forRoot/forFeature methods
Criação	Singleton	NestJS Services
Estrutural	Module	NestJS Modules
Estrutural	Facade	Controllers
Estrutural	Repository	TypeORM Repository
Comportamental	Observer	EventEmitter
Comportamental	Strategy	Dual batch processing
Mensageria	Publisher-Subscriber	RabbitMQ
Persistência	Unit of Work	Database transactions
Resiliência	Retry	Message retry logic
Concorrência	Batch Processing	Event batching



Conclusão

Arquitetura Rica em Padrões

Este projeto demonstra uma implementação robusta das melhores práticas para sistemas distribuídos, resilientes e escaláveis. A combinação estratégica de padrões de design cria uma arquitetura que é:

Modular

Componentes desacoplados que podem ser desenvolvidos, testados e mantidos independentemente.

Resiliente

Capaz de lidar com falhas através de padrões como Retry, Dead Letter Queue e Circuit Breaker.

Escalável

Processamento em lote, isolamento de recursos e mensageria assíncrona permitem escalar horizontalmente.