UNIVERSIDADE FEDERAL DE PERNAMBUCO

Milton José Vieira Souto Maior

Análise Comparativa de Performance de Frameworks para APIs Rest

Recife,

2023

UNIVERSIDADE FEDERAL DE PERNAMBUCO

Sistemas de informação

Milton José Vieira Souto Maior

Análise Comparativa de Performance de Frameworks para APIs Rest

TCC apresentado ao Curso de Sistemas de Informação da Universidade Federal de Pernambuco, como requisito para a obtenção do título de bacharel em Sistemas de informação.

Orientador(a): Breno Alexandro Miranda

Recife

2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Maior, Milton José Vieira Souto.
Análise comparativa de performance de frameworks para APIs Rest / Milton José Vieira Souto Maior. - Recife, 2023.
55 : il., tab.

Orientador(a): Breno Alexandro Ferreira Miranda
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Sistemas de Informação - Bacharelado, 2023.

1. framework. 2. API. 3. performance. I. Miranda, Breno Alexandro Ferreira. (Orientação). II. Título.

000 CDD (22.ed.)

Milton José Vieira Souto Maior

Análise Comparativa de Performance de Frameworks para APIs Rest

TCC apresentado ao Curso de Sistemas de Informação da Universidade Federal de Pernambuco, como requisito para a obtenção do título de bacharel em Sistemas de informação..

Aprovado em: ___/___/_____.

BANCA EXAMINADORA

_____
Profº. Dr. Breno Alexandro Ferreira Miranda (Orientador)
Universidade Federal de Pernambuco

_____
Profº. Dr. Juliano Manabu Iyoda (Examinador Interno)
Universidade Federal de Pernambuco

RESUMO

O desenvolvimento de software, sobretudo de Application Programming Interfaces (APIs) vem evoluindo e mudando de maneira muito rápida devido à alta quantidade de tecnologias no mercado. Diversas linguagens de programação e frameworks são utilizados para implementar essas interfaces, cada uma com suas peculiaridades, vantagens e desvantagens, o que dificulta tomar decisões, como qual tecnologia utilizar para fazer uma API? Qual framework mostraria mais desempenho? Nesse sentido, esse trabalho apresenta um comparativo de desempenho de uma mesma implementação de API desenvolvida nos em diferentes frameworks utilizados por programadores no mercado.

Palavras-chave: framework ; API; performance.

ABSTRACT

Software development, especially Application Programming Interfaces (APIs) has been evolving and changing very quickly due to the high number of technologies on the market. Several programming languages and frameworks are used to implement these interfaces, each with its own peculiarities, advantages and disadvantages, which makes it difficult to make decisions, such as which technology to use to make an API? Which framework would show more performance? In this sense, this work presents a performance comparison of the same API implementation developed in different frameworks used by programmers in the market.

Keywords: framework ; API; performance.

SUMÁRIO

1 INTRODUÇÃO

No cenário atual, com a tecnologia da informação utilizada em diversos contextos diferentes, fez surgir uma alta demanda de serviços informatizados e online. Com diversas linguagens de programação surgindo com o tempo e sendo utilizadas no mercado e em pesquisas em inúmeros contextos.

No contexto mais específico de um dos ramos da programação, os serviços de Application Programming Interfaces (APIs) são serviços utilizados para enviar ou receber informações de um cliente (sistema).

Devido a isso surgiram diversos frameworks que auxiliam a programar essas interfaces, cada uma com suas qualidades e peculiaridades, o que leva a questão de qual escolher? Qual linguagem? Qual framework utilizar? Qual seria mais performático? A escolha de uma linguagem para resolver esses problemas de criar APIs vai, muitas vezes, muito além de qual seria mais performático, mas também observando-se outras características específicas, como afinidade das pessoas relacionadas com tal tecnologia, capacidade de recursos de hardware e até mesmo funcionalidades específicas em dados contextos que seriam melhores aproveitadas se fossem feitas em dada linguagem.

É possível assim demonstrar que seria difícil fazer uma comparação abstrata e genérica que mostraria que tal linguagem se sobressairia sobre outras de maneira incontestável em todos os cenários.

No entanto, ao definir um problema real que utiliza-se de uma API, em que todos esses frameworks fossem implementados da mesma forma e utilizassem as mesmas configurações de infraestrutura e mesma arquitetura, seria possível mostrar qual conseguiria ser sobressair sobre o outro e demonstrar também que características dos frameworks corroboram para ele ser mais performático nesse caso.

Esse trabalho mostra uma comparação de desempenho entre os 4 frameworks mais utilizados por programadores para esse fim no ano de 2022, conforme as pesquisas da comunidade do Stackoverflow.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção pretende detalhar os tópicos abordados nesta pesquisa de maneira mais teórica, explicando os princípios e conceitos, detalhando o que são web frameworks, APIs Rest. Como também, explorar as características das linguagens e dos frameworks utilizados neste estudo.

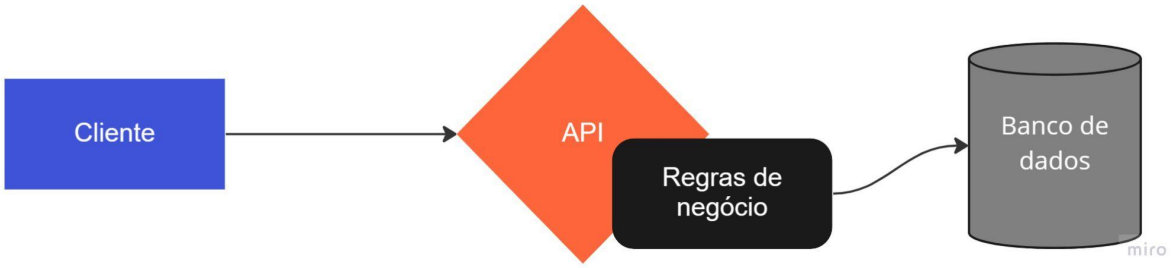
2.1 O que são web frameworks

Web framework é um framework de software desenvolvido para facilitar o processo de desenvolvimento web, fazendo com que seja mais fácil de se fazer um site ou sistema. Um web framework inclui todo o processo de desenvolvimento de apresentação de conteúdo no navegador como também nas APIs.

2.1.1 Application Programming Interface

Application Programing Interface(API) é um conjunto de definições e protocolos para construir e integrar aplicações de softwares. É utilizada como meio de comunicação entre aplicações.

Figura 1: Diagrama de um cliente para uma API.

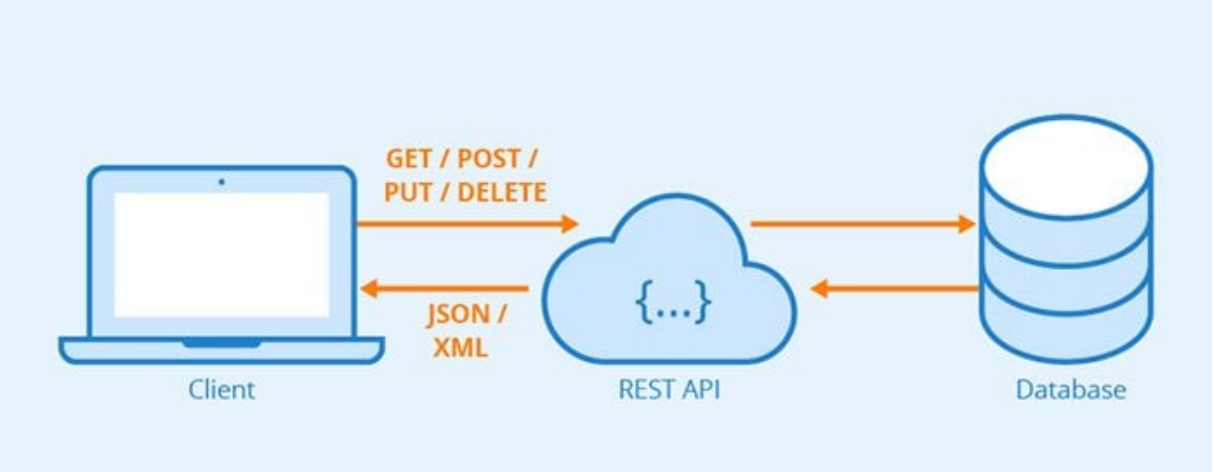


Fonte: O autor (2023)

Na figura 1, pode-se ver o fluxo de uma aplicação monolítica simples que utiliza um cliente que consome uma API de instância única para ter acesso a dados, em questões de leitura e escrita, respeitando as regras que estão estabelecidas dentro da API.

2.1.2 Rest e API RestFul

Figura 2: Ilustração do fluxo Rest API.



Fonte: Seobility - licença CC BY-SA 4.0

A arquitetura Rest foi criada pelo cientista da computação Roy Fielding nos anos 2000. No intuito de padronizar os protocolos de comunicação e desenvolvimento na internet, Fielding com um time de especialistas desenvolveu as características da Representational State Transfer (REST), que foi definida na sua tese de PhD.

Podendo-se definir como um conjunto de princípios e restrições, REST segue um padrão de transferência do estado do recurso ao solicitante ou endpoint, onde essa transferência é feita pelo protocolo HTTP, utilizando diversos formatos como JSON, HTML, XLT e plaintext, sendo o JSON o mais utilizado.

Uma API Restful, é definida por uma API que está conforme os critérios estabelecidos pela arquitetura REST, nos quais podem-se destacar os seguintes critérios:

- **Arquitetura cliente-servidor**: Dividir a interface do usuário da persistência de dados com pelo menos dois serviços para o sistema. Um responsável pela interface (cliente) e outro pelo servidor (API).
- **Comunicação stateless**: A comunicação feita entre cliente e servidor não deve armazenar nenhuma informação entre as solicitações. Em uma REST API, cada solicitação contém todos os dados necessários para ser atendida, não dependendo de informações já armazenadas em outras sessões.

- **Cache**: A API deve conseguir salvar em cache recursos e dados para melhorar o desempenho.
- **Interface uniforme**: Uma REST API deve conter uma interface uniforme, pois ela oferece uma comunicação padronizada entre o usuário e o software. A manipulação de recursos é feita por meio de representações (como JSON ou XML).
- **Sistema de camadas**: cada camada do sistema deve possuir uma funcionalidade específica (como segurança ou carregamento). Assim, cada camada é responsável por uma etapa diferente dos processos de requisição de usuário e de resposta do servidor.

2.2 - Stack Overflow

O *StackOverflow* é um plataforma e comunidade *online* de perguntas e respostas mais utilizada por programadores profissionais e não profissionais em diversas áreas de tecnologia da informação, sobretudo na área de engenharia de software. A comunidade dissemina conhecimento mediante discussões entre as pessoas da comunidade. Devido à comunidade ser muito forte, é feita uma pesquisa anualmente entre a comunidade, respondendo a questionários que vão desde localização geográfica da comunidade a linguagens de programação e frameworks que a comunidade mais utilizou naquele período.

2.3 - Linguagens de programação

As linguagens de programação utilizadas foram escolhidas selecionando as 4 primeiras mais utilizadas no contexto de desenvolvimento web para APIs segundo as pesquisas do *StackOverflow* em 2022, essas demonstradas na tabela 1.

Tabela 1: Comparativo entre linguagens de programação.

Linguagens	**Java**	**JavaScript**	**Python**	**C#**
Execução	Compilada	Interpretada	Interpretada	Compilada
Modelo de concorrência	Multi threads	Single thread	Single thread e Multi threads	Multi threads

Tipagem	Estática	Dinâmica	Dinâmica	Estática e Dinâmica
Paradigma	Orientada a objetos, imperativa e funcional	Orientado a eventos, funcional e procedural	Orientado a objetos, procedural e funcional	Orientado a objetos, baseada em componentes e imperativa
Garbage collection	Sim	Sim	Sim	Sim
Suporte a async	Sim	Sim	Sim	Sim
Tipagem	Forte	Fraca	Fraca	Forte
Tipo de compilação	JIT ou AOT	JIT	JIT	JIT ou AOT
Gerenciamento de pacotes	maven, Gradle e Ant	npm	pip	nuGet

Fonte: O autor(2023)

A tabela 1 mostra as principais características a respeito das linguagens utilizadas pelos frameworks escolhidos. De maneira geral, múltiplas threads podem ajudar a lidar com concorrências, o que pode ser um fator a ser considerado quando a API lida com múltiplas requisições de vários usuários ao mesmo tempo.

2.4 ORMs

ORM (Object-Relational Mapping) é uma técnica de programação que permite desenvolvedores interagirem com o banco de dados usando paradigmas da programação objeto relacional para muitas vezes suprir as necessidades de escrever queries nativas SQL. ORM mapeia uma tabela e suas relações com outras tabelas para classes, usando conceitos como herança, encapsulamento para manipular os dados no banco de dados.

2.5 Frameworks

Tabela 2: Comparativo entre os frameworks escolhidos.

Framework	Linguagem	Multithread	ORM	Async	Suporte a cache	Versão
Spring Boot	Java	suportado	Hibernate	Sim	Sim	3.0.1
Express.js	JavaScript	Não suportado	sequelize	Sim	Sim	4.18.2
ASP.NET Core	c#	suportado	dapper	Sim	Sim	7.0.101
Django	Python	suportado	djangoORM	Sim	Sim	4.1.7

A tabela 2 mostra as principais características dos frameworks a serem utilizados para a comparação.

2.5.1 Spring boot

Criado pelo time spring na Pivotal Software, é o framework de código aberto Java mais utilizado para implementações de serviços de API, com uma comunidade muito forte e diversas bibliotecas para aumentar a robustez do framework. Sua qualidade mais atrativa é “Convenção sobre a configuração” que reduz a quantidade de código que seria necessário para configurar uma aplicação.

2.5.2 ExpressJs

Criado por TJ Holowaychuk em 2010, se tornou o mais popular framework do Node.js. É um framework de código aberto que provê simplicidade e um conjunto leve de recursos utilizados para fazer as aplicações de REST. Possui uma grande comunidade ativa e abundante de plugins e módulos que estendem suas capacidades além de ser implementado em JavaScript, que até a data que esse estudo está sendo feito, é considerado a linguagem de programação mais utilizada no mundo.

2.5.3 ASP.NET Core

Criado por um time de desenvolvedores da Microsoft, é um framework de código aberto, múltiplas plataformas, e é o sucessor do ASP.NET framework original lançado em 2016. Ele foi feito para ser modular, leve, flexível e feito no topo do runtime do .NETCore que é um runtime de múltipla plataforma de código aberto para desenvolver aplicações web, como API. Possui múltiplas features sendo lançadas constantemente, com uma comunidade forte e o apoio da Microsoft no suporte de suas tecnologias.

2.5.4 Django

Criado em 2003 por Adrian Holovaty and Simon Wilson, Django é um framework de aplicação web baseado em Python e de código aberto. Foi feito no intuito de construir aplicações web complexas de maneira rápida e fácil. Com diversas features como ORMs, roteamento de URLs, middlewares e vários outros que fizeram ele ser popular, possui uma comunidade forte e engajada, sendo também bastante utilizado pela indústria.

3 METODOLOGIA

3.1- O Problema

A escolha das linguagens foi baseada na pesquisa anual do *stackoverflow,* na qual a comunidade de desenvolvedores pelo mundo responde essa pesquisa.
Consoante a pesquisa, com 58.743 respostas, os mais utilizados na área de web frameworks foram *ExpressJS (22.99%), ASP.NET Core (18,59%), Spring (16.3%) e Django (14,65%).*

Esse trabalho fará uma análise comparativa entre esses frameworks da forma mais justa possível, dado que eles funcionam de maneiras diferentes e suas linguagens têm diferentes limitações e features implementadas de maneira diferente. Para isso, vamos utilizar a mesma arquitetura de camadas para implementação de uma API para tentar colocar esses frameworks nas mesmas condições e na mesma complexidade. Assim podemos observar o comportamento de cada um com a mesma implementação diante dos mesmos cenários de testes.

A escolha dos frameworks

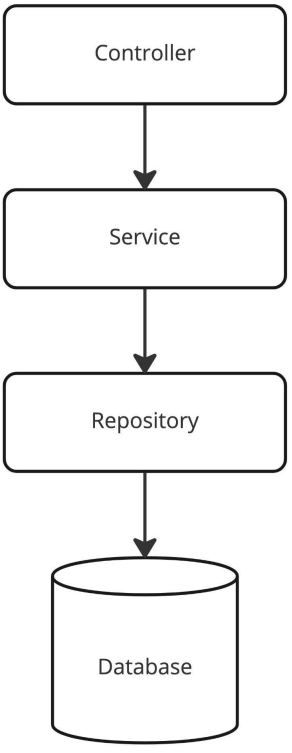
Os frameworks foram selecionados a partir da pesquisa anual do *stackoverflow*, na qual a comunidade responde um questionário com as tecnologias mais utilizadas, questões salariais e geográficas. Foi selecionado o top 4 das tecnologias mais utilizadas na área de desenvolvimento web de *backend.*

3.2 Definição da arquitetura e do modelo relacional

3.2.1 - A arquitetura

Para podermos nivelar os frameworks e tentar fazer um comparativo com condições iguais, o autor propõe um modelo de arquitetura para ser utilizado na implementação da API dos frameworks a serem testados. A arquitetura foi feita com base no conhecimento do autor derivado de seu tempo no curso de bacharelado e no mercado de trabalho trabalhando como desenvolvedor.

Figura 3: Modelo de arquitetura definido pelo autor



Fonte: O autor (2023)

A arquitetura foi elaborada conforme a figura 3, com 3 camadas de abstração e o acesso a banco de dados.

Controller: Camada responsável por receber os requests e tratar os erros, status e retorno para o cliente que fez a requisição.

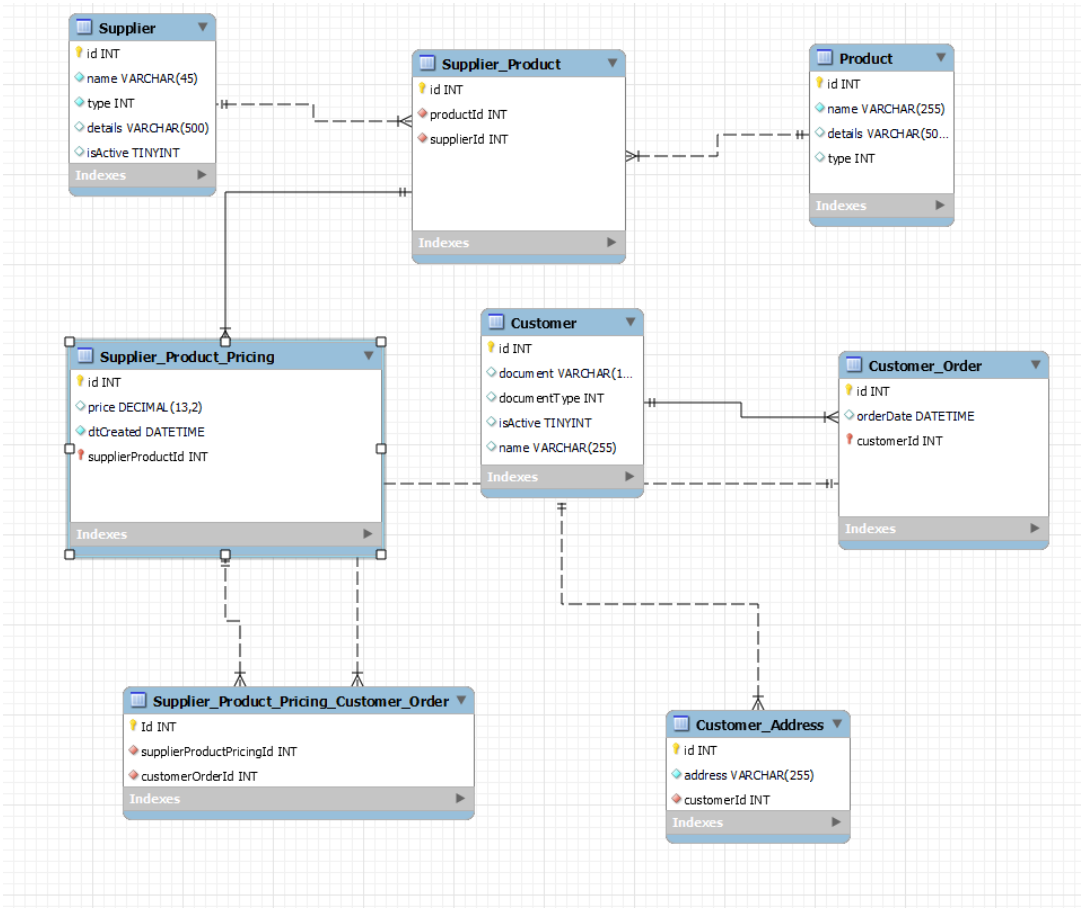
Service: Camada responsável pelas regras de negócio da aplicação, contendo a maioria da lógica e do pós-processamento dos dados.

Repository: Camada responsável por ter acesso ao banco de dados, com única responsabilidade de estabelecer a conexão com o banco e fazer operações de leitura e escrita.

3.2.1 - O modelo relacional

O modelo relacional foi feito pelo autor no intuito de forçar os frameworks a fazerem consultas não triviais e que requeiram pós-processamento desses dados para se obter um retorno desejado, esses que serão descritos mais adiante neste trabalho.

Figura 4: Modelo de arquitetura definido pelo autor



Fonte: O autor(2023)

O banco de dados mostrado na figura 4, é uma esquematização para uma operação de E-commerce, na qual existem fornecedores, produtos, clientes e pedidos. Toda a lógica sobre o modelo será melhor explicada e detalhada mais adiante.

3.3 O Modelo de testagem

Para fazer testes de performance de APIs é necessário separar critérios para serem pontuados e avaliados. Considerando apenas o desempenho dos frameworks, os seguintes critérios serão avaliados.

- Tempo de request
  - tempo médio
  - tempo mínimo
  - tempo máximo
  - vazão de request / tempo
- Percentual de erro (erros que podem acontecer ao se fazer uma request em modos de estresse)
- Análise de comportamento nos diferentes cenários de teste

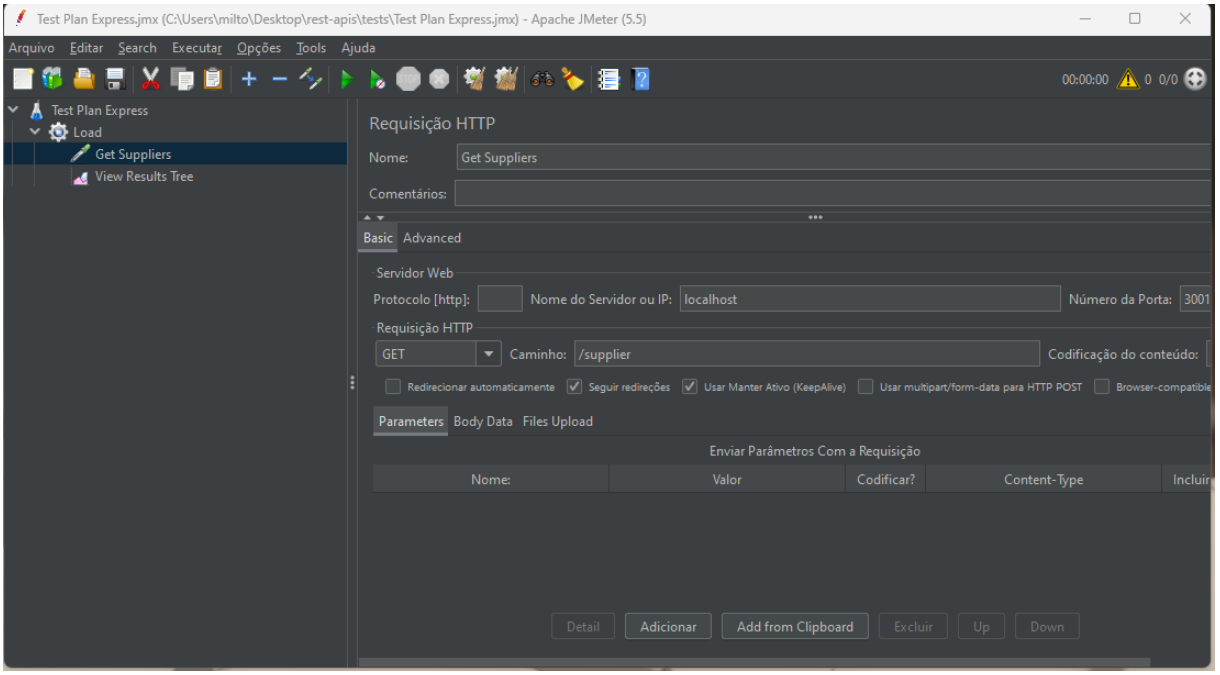
A ferramenta de testagem

Para testar as APIs fazendo requisições de maneira a simular mais usuários ou até mesmo simular picos de requisições, a ferramenta *apache Jmeter* foi utilizada nesse trabalho.

Para realizar testes de desempenho entre as APIs é preciso fazer testes de estresse para medir o tempo de respostas para as requisições do cliente para a API.

Esse trabalho utilizará chamadas HTTP para fazer os testes nas 4 aplicações implementadas. Como todos os testes foram realizados numa única máquina local e devido ao tempo de pesquisa foi optado o uso do HTTP ao invés do HTTPS.

Figura 5: Ambiente do apache Jmeter



Fonte: O autor (2023)

O *Jmeter* (Figura 5) consegue simular, múltiplos usuários fazendo requisições, com configurações de intervalo de tempos, assim permitindo ser explorado vários casos de testes.

4. TRABALHOS RELACIONADOS

Há alguns trabalhos na literatura que fizeram avaliações de desempenho de *frameworks*, alguns que compararam frameworks diferentes de uma mesma linguagem, outros que compararam o desempenho de um *framework* em um problema específicos como de persistência de dados. Por fim, a literatura também tem pesquisas que comparavam de 2 a 3 frameworks diferentes, realizando análise de desempenho entre eles.

O trabalho relacionado mais próximo da pesquisa deste trabalho foi feito por *DALBARD, Axel; ISACSON, Jesper (2021)*. Eles compararam Dotnet Core com ExpressJs em uma API que tinha uma ferramenta que, dadas as dimensões de um produto e as dimensões de um container padronizado, calculava quantos produtos poderiam ser armazenados. O teste de desempenho mensurou o consumo de recursos e capacidade das APIs em responder a inúmeros requests. A pesquisa concluiu que o Dotnet core conseguiu ter um desempenho melhor do que o ExpressJs.

Na pesquisa de *Dhalla, Hardeep Kaur.(2021),* Dhalla faz uma comparação entre o Spring e Asp.Net Core de uma API Restful utilizando como critérios o tempo de resposta médio de uma requisição e o percentual de erro nas requisições feitas para os dois frameworks para operações de Create, Retrieve, Update, Delete (CRUD) Com o banco do MySQL. O trabalho mostrou que, no aspecto geral, o .Net Core conseguiu ter um tempo médio menor, consumindo menos recursos, e quando a carga de requisições foi para 64.000 usuários, a faixa de erro nas requisições chegou a 85%.

5. SISTEMA DESENVOLVIDO PARA TESTES

O sistema proposto para ser desenvolvido de base para essas API se trata de uma aplicação de e-commerce, contendo fornecedores, produtos e clientes e compras, cada um com seus atributos e relações.

A escolha do modelo relacional para mapear o sistema foi feita para forçar os sistemas a fazerem buscas complexas e terem pós-processamento para conseguir retornar resultados específicos. Bancos de dados não relacionais como o mongoDB poderiam ser utilizados para fazer testes, porém a abordagem seria um pouco diferente da proposta deste trabalho.

5.1 VISÃO GERAL

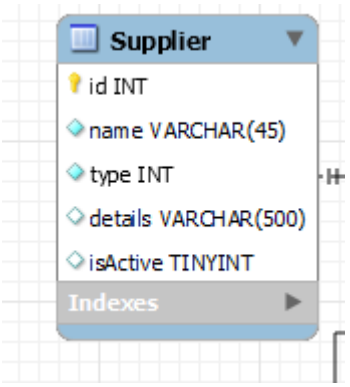
O sistema representa um modelo de e-commerce, feito com práticas de modelagem utilizadas no mercado. O sistema foi modelado pensando unicamente em benchmarking e não para ser um banco de dados que resolveria um problema da realidade, apesar de estar bem próximo disso.

Como requisitos vários fornecedores vendem produtos que podem ser fornecidos ou não por fornecedores diferentes. O produto terá um histórico de preços, onde o preço atual será o mais recente, definido por sua data de criação.

Os clientes desse e-commerce podem ter mais de um endereço e realizar compras armazenadas no banco de dados.

Supplier

Figura 6: Representação ER Supplier



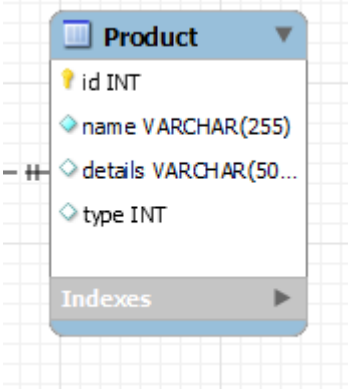
Fonte: O autor (2023)

A entidade *Supplier* representa um fornecedor, contendo propriedades como:

Id: Inteiro sequencial autogerado pelo banco para representar um atributo único.
Name: Representa o nome do fornecedor.
type: Um identificador variando de 0 ou 1 para representar se o fornecedor é do tipo big ou small.
isActive: Binário com 0 ou 1 representando se o fornecedor está ativo ou não.

Product

Figura 7: Representação ER do Product



Fonte: O autor(2023)

A entidade *Product* representa um produto, contendo propriedades como:

Id: Inteiro sequencial autogerado pelo banco para representar um atributo único

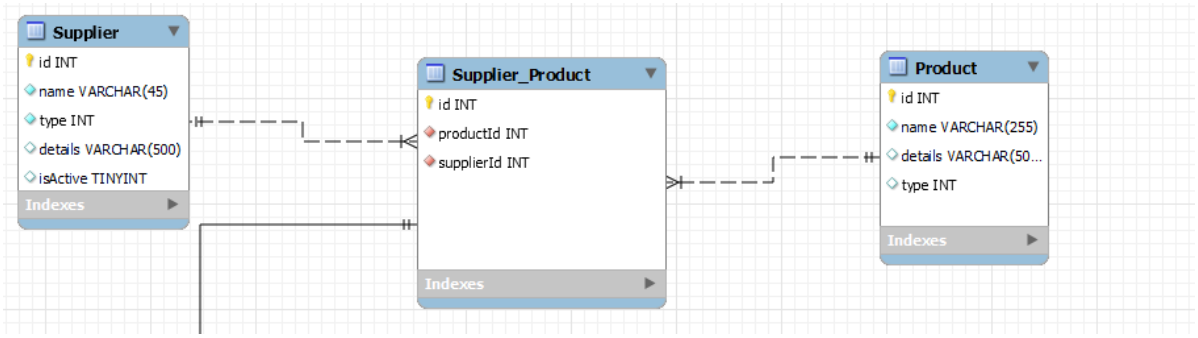
Name: Representa o nome do produto

details: Detalhes do produto

type: Um identificador para saber se o produto é físico ou digital

Supplier_Product

Figura 8: Representação ER de produto e fornecedor



Fonte: O autor (2023)

A entidade *Supplier_Product* representa a relação MxN entre *Supplier* e *Product.* Essa tabela contém atributos como:

Id: Inteiro sequencial autogerado pelo banco para representar um atributo único

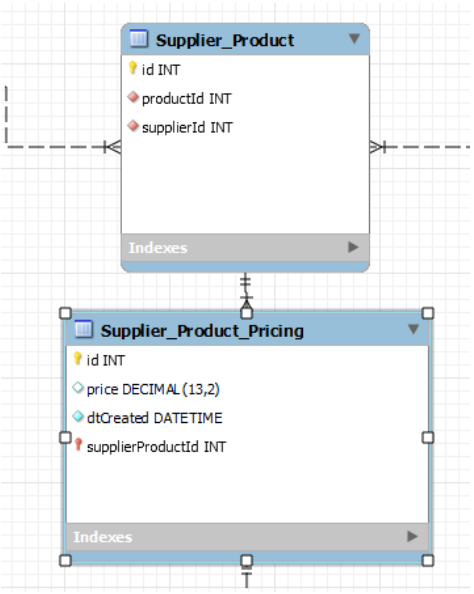
productId: Chave estrangeira da entidade Supplier

supplierId: Chave estrangeira da entidade Product.

A Relação *Supplier* e *Product* foi mapeada dessa forma para poder ter o id autogerado como chave primária da tabela.

Supplier_Product_Princing

Figura 9: Representação ER do supplier_Product e Supplier_Product_Pricing



Fonte: O autor(2023)

A Entidade *Supplier_Product_Pricing* representa os preços do produto oferecidos pelo fornecedor, estabelecendo a relação entre *Supplier_Product* e *Supplier_Product_Pricing.*

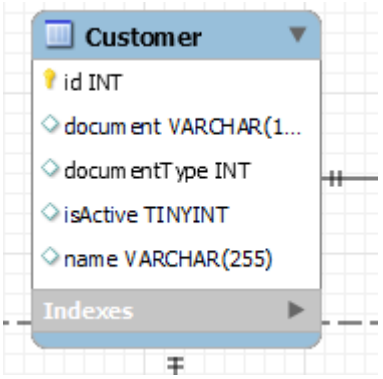
A relação estabelecida é que, para cada **1** produto oferecido pelo fornecedor, haverá **N** preços. Para se obter o preço atual será definido pelo preço com o campo **dtCreated** com a data mais atual.

Os atributos do Supplier_Product_Pricing são definidos como:

Id: Inteiro sequencial autogerado pelo banco para representar um atributo único.
price: Decimal que representa o preço do produto.
supplierProductId: Chave estrangeira da entidade *Supplier_Product*.
dtCreated: Campo date que representa a data que o preço foi criado.

Customer

Figura 10: Representação do modelo ER do Customer



Fonte: O autor(2023)

A tabela *Customer* representa o cliente do sistema, com propriedades como:

Id: Inteiro sequencial autogerado pelo banco para representar um atributo único.

document: String que representa o CPF ou CNPJ do cliente.

documentType: Identificador que representa o tipo do documento, sendo 1 para CPF e 0 para CNPJ.

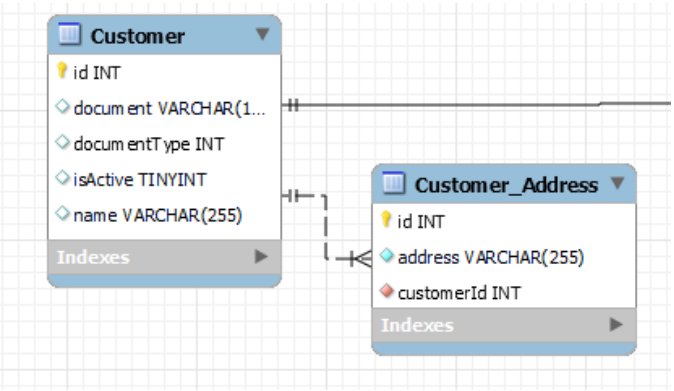
isActive: Booleano para representar se o cliente está ativado ou desativado.

name: String que representa o nome do cliente.

Customer_Address

Essa entidade modela representação do endereço de um cliente, tendo a relação de 1 cliente tendo N endereços.

Figura 11: Representação ER do Customer Address



Fonte: O autor(2023)

A tabela *Customer_Address* possui atributos como:

Id: Inteiro sequencial autogerado pelo banco para representar um atributo único.

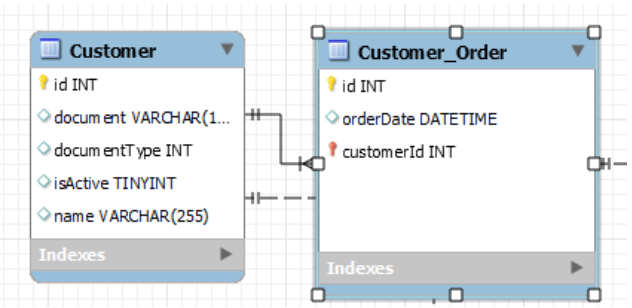
address: String que representa o endereço completo do cliente

documentType: Identificador que representa o tipo do documento, sendo 1 para CPF e 0 para CNPJ.

Customer_Order

A entidade *Customer_Order* representa uma compra realizada por um cliente, tendo relação de 1 cliente podendo ter N pedidos.

Figura 12: Representação ER do Customer_Order



Fonte: O autor(2023)

A entidade possui atributos como:

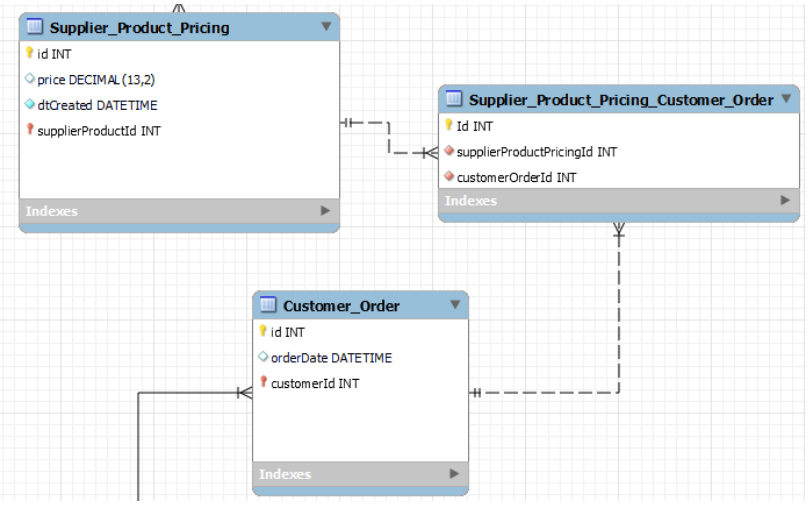
Id: Inteiro sequencial autogerado pelo banco para representar um atributo único.

orderDate: Data que o pedido foi feito.
customerId: Chave estrangeira que identifica o cliente que fez o pedido.

Supplier_Product_Pricing_Customer_Order

Essa relação representa o relacionamento MxN da tabela que representa o preço dos produtos e da tabela que representa a compra feita pelo cliente. Nesse modelo, a compra pode ter vários produtos (nesse caso a referência é feita pelo id do preço do produto comprado).

Figura 13: Representação ER do Supplier_Product_Pricing_Customer_Order



Fonte: O autor (2023)

A entidade possui atributos como:
Id: Inteiro sequencial autogerado pelo banco para representar um atributo único.
supplierProductPricingId: Chave estrangeira que identifica o preço do produto que foi comprado.
customerOrderId: Chave estrangeira que identifica o pedido.

5.2 Arquitetura de implementação

A Arquitetura desenvolvida pelo autor se baseia em camadas com responsabilidades únicas como mostradas na figura 3.

A camada de serviços contém a parte lógica, as quais foram feitas utilizando funções com mesma lógica e estrutura de dados para todas as implementações nos diferentes frameworks.
A camada de acesso ao banco de dados tem a responsabilidade de estabelecer a conexão com o banco e executar a query, fechando a conexão assim que a consulta retorna os resultados.

6 TESTAGEM

6.1 Configuração do ambiente de testes

Todos os testes foram realizados no mesmo computador, com as configurações destacadas a seguir.

Processador: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz - 1.80 GHz
Memória ram: 16gb ddr4
Sistema operacional: Windows 11 Home Single Language, versão 22H2
Banco de dados: MySQL server 8.0

Para os testes foi utilizado um banco de dados que possui 10 mil instâncias para as tabelas principais e 20 mil instâncias para as tabelas que representam relacionamento. O banco de dados foi povoado por um script feito pelo autor.

6.2 Endpoints

Foram feitos dois endpoints nas implementações das aplicações de backend.

6.2.1 - GET Suppliers

Esse endpoint faz a consulta para trazer todos os suppliers e os produtos relacionados a eles. Para fazer essa consulta em um banco relacional que possui as configurações de modelagem descrita nas seções anteriores, é preciso consultar 3 tabelas e processar os dados para serem retornados segundo a figura 14.

O retorno é um JSON que contém o fornecedor e uma lista com outro JSON contendo os produtos que estão relacionados a ele.

Figura 14: Representação dos dados que serão retornados na requisição


```
{
   "name":"name_1",
   "isActive":1,
   "id":1,
   "products":[
      {
         "name":"name_1",
         "id":1,
         "details":"details details"
      },
      {
         "id":2,
         "details":"details details"
      }
   ]
}
```


Todos os frameworks utilizam a mesma query SQL para consultar as informações no banco de dados. A query na figura 15 retorna todas as instâncias de fornecedores com seus produtos que representam um total de 19.999 instâncias a serem retornadas e posteriormente feito um pós-processamento para ficarem no padrão da figura 14.

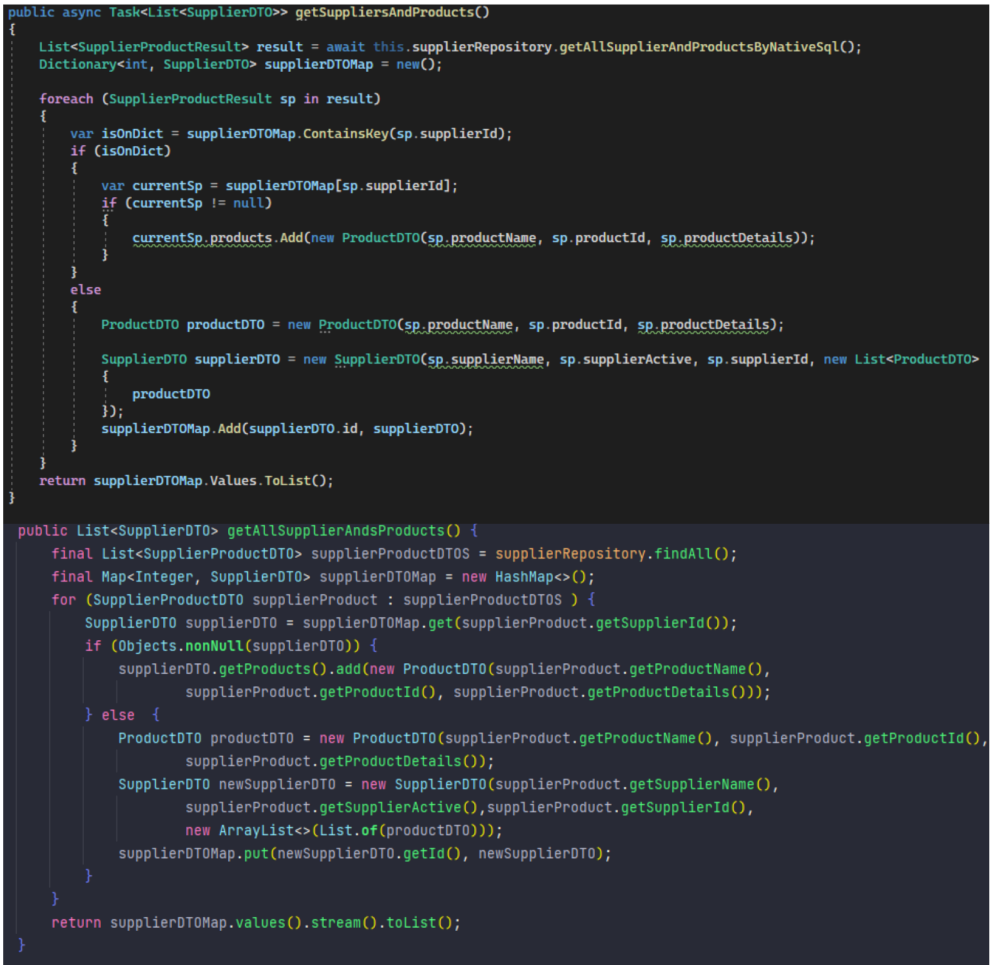
Figura 15: Query SQL feita pelo autor para retornar todos os fornecedores e seus produtos

```
SELECT s.name AS supplierName, s.isActive AS supplierActive,
s.id AS supplierId, p.name AS productName, p.id AS productId, p.details AS productDetails
FROM supplier s
INNER JOIN supplier_product sp ON sp.supplierId = s.id
INNER JOIN product p ON sp.productId = p.id
```

Fonte: O autor (2023).

A implementação da parte lógica de cada framework é mostrada abaixo:

Figura 16: Implementação da parte logica no ASP.NET Core e Spring respectivamente



```
public async Task<List<SupplierDTO>> getSuppliersAndProducts()
{
    List<SupplierProductResult> result = await this.supplierRepository.getAllSupplierAndProductsByNativeSql();
    Dictionary<int, SupplierDTO> supplierDTOMap = new();

    foreach (SupplierProductResult sp in result)
    {
        var isOnDict = supplierDTOMap.ContainsKey(sp.supplierId);
        if (isOnDict)
        {
            var currentSp = supplierDTOMap[sp.supplierId];
            if (currentSp != null)
            {
                currentSp.products.Add(new ProductDTO(sp.productName, sp.productId, sp.productDetails));
            }
        }
        else
        {
            ProductDTO productDTO = new ProductDTO(sp.productName, sp.productId, sp.productDetails);

            SupplierDTO supplierDTO = new SupplierDTO(sp.supplierName, sp.supplierActive, sp.supplierId, new List<ProductDTO>
            {
                productDTO
            });
            supplierDTOMap.Add(supplierDTO.id, supplierDTO);
        }
    }
    return supplierDTOMap.Values.ToList();
}
```

```
public List<SupplierDTO> getAllSupplierAndsProducts() {
    final List<SupplierProductDTO> supplierProductDTOS = supplierRepository.findAll();
    final Map<Integer, SupplierDTO> supplierDTOMap = new HashMap<>();
    for (SupplierProductDTO supplierProduct : supplierProductDTOS ) {
        SupplierDTO supplierDTO = supplierDTOMap.get(supplierProduct.getSupplierId());
        if (Objects.nonNull(supplierDTO)) {
            supplierDTO.getProducts().add(new ProductDTO(supplierProduct.getProductName(),
                    supplierProduct.getProductId(), supplierProduct.getProductDetails()));
        } else  {
            ProductDTO productDTO = new ProductDTO(supplierProduct.getProductName(), supplierProduct.getProductId(),
                    supplierProduct.getProductDetails());
            SupplierDTO newSupplierDTO = new SupplierDTO(supplierProduct.getSupplierName(),
                    supplierProduct.getSupplierActive(),supplierProduct.getSupplierId(),
                    new ArrayList<>(List.of(productDTO)));
            supplierDTOMap.put(newSupplierDTO.getId(), newSupplierDTO);
        }
    }
    return supplierDTOMap.values().stream().toList();
}
```

Figura 17: Implementação da parte lógica no DJANGO e no EXPRESSJS respectivamente

```
def getAllSuppliers(self):
    repository = SupplierRepository()
    sqlResults = repository.fetchAllSuppliers()
    supplierProductDict = dict()
    for sp in sqlResults:
        supplier = supplierProductDict.get(sp.get('supplierId'))
        if (supplier is not None):
            supplier.get('products').append({
                'name': sp.get('productName'),
                'id': sp.get('productId'),
                'details': sp.get('productDetails'),
            })
        else:
            newSupplier = {
                'name': sp.get('supplierName'),
                'isActive': sp.get('supplierActive'),
                'id': sp.get('supplierId'),
                'products': [
                    {
                    'name': sp.get('productName'),
                    'id': sp.get('productId'),
                    'details': sp.get('productDetails'),
                    },       You, 2 weeks ago • Added logic to get su
                ],
            };
            supplierProductDict[newSupplier.get('id')] = newSupplier
    return supplierProductDict.values()
```

```
async getAllBySql() {
  const sqlResults = await this.repository.getAllSuppliersAndProduct
  const supplierProductMap = new Map();

  for (const supplierProduct of sqlResults) {
    const supplier = supplierProductMap.get(supplierProduct.supplier
    if (supplier) {
      supplier.products.push({
        name const supplierProduct: any e,
        id: supplierProduct.productId,
        details: supplierProduct.productDetails,       You, 2 months
      });
    } else {
      const newSupplier = {
        name: supplierProduct.supplierName,
        isActive: supplierProduct.supplierActive,
        id: supplierProduct.supplierId,
        products: [
          {
            name: supplierProduct.productName,
            id: supplierProduct.productId,
            details: supplierProduct.productDetails,
          },
        ],
      };
      supplierProductMap.set(newSupplier.id, newSupplier);
    }
  }
  return Array.from(supplierProductMap, ([_, value]) => value);
}
```

Basicamente, é feito um for para percorrer os resultados e processá-los, utilizando uma estrutura de dados como um HashMap para lidar com o retorno da query. Por fim é feito um parse para serem retornados os valores do HashMap em uma lista para ficar igual à figura 14.

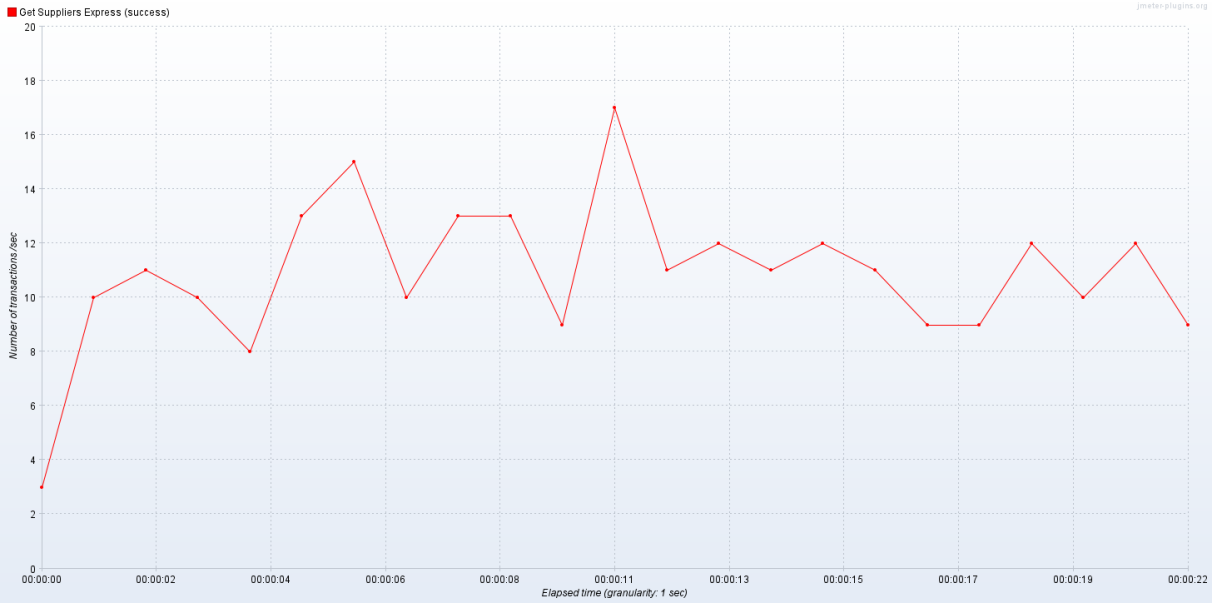
Cenário 1: 50 requisições por segundo, durante 5s, totalizando 250 requisições

Utilizando conexão com banco de dados e executando a mesma query para todos os frameworks, foram obtidos os seguintes resultados.

ExpressJs

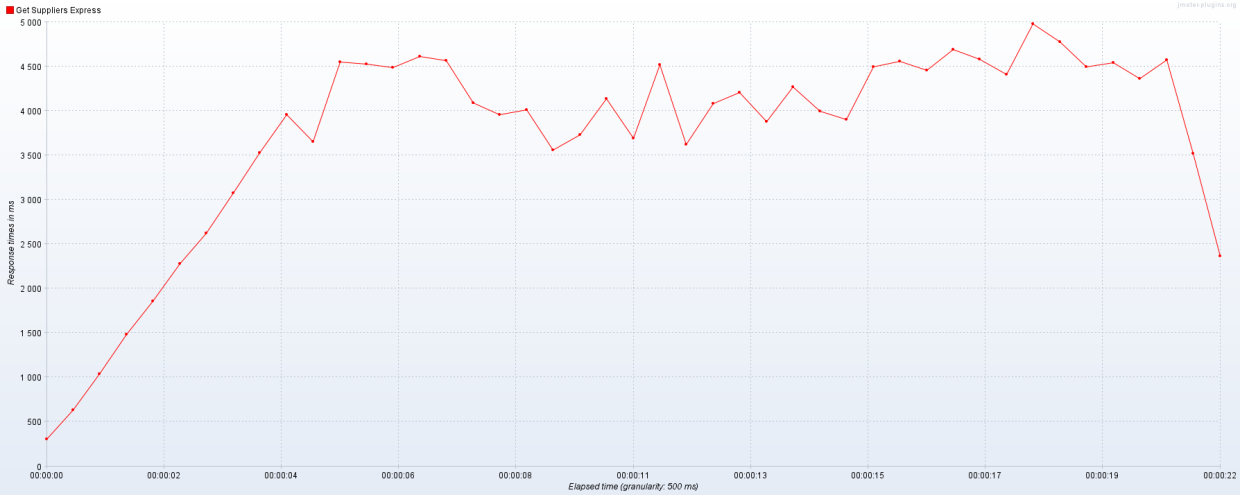
Para executar todas as requisições, foi levado um tempo total de 22 segundos (figura 18).

Figura 18: Número de transações por tempo total de teste



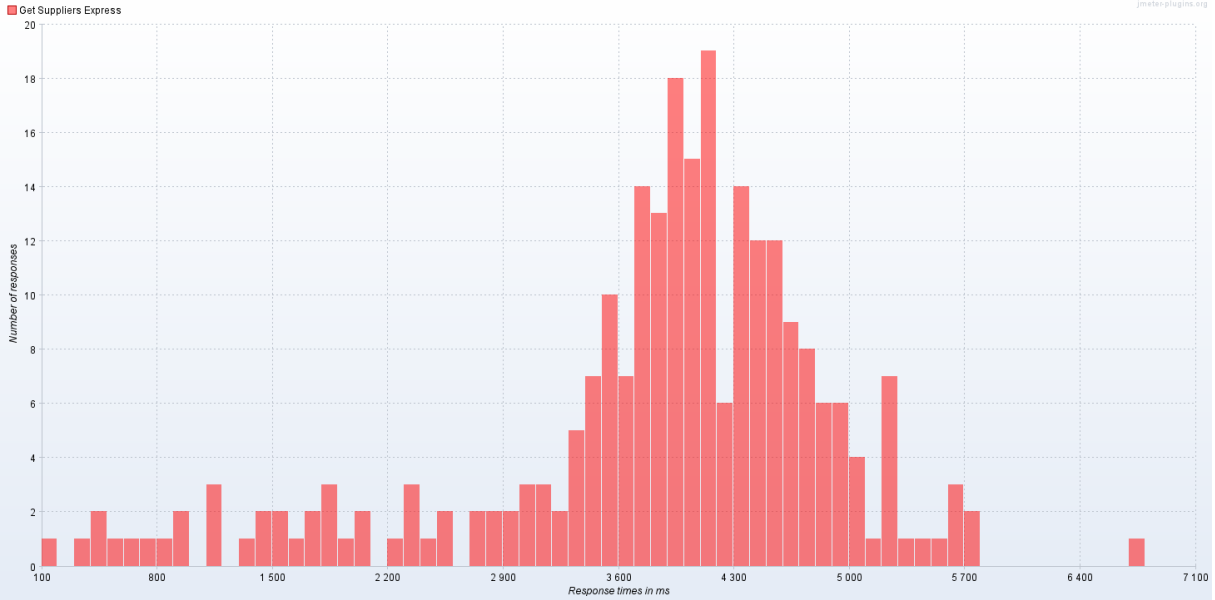
É possível perceber que, ao atingir a metade do tempo total, foi-se atingido o melhor número de requisições por milissegundo.

Figura 19: Tempo de resposta de uma request x tempo total do teste



O tempo de resposta da requisição (figura 19) foi subindo com o tempo, muito possivelmente por ter que esperar uma requisição acabar para poder começar a outra.

Figura 20: Distribuição do número de requisições com seu tempo de resposta total



A distribuição (figura 20) mostra que a maioria das requisições estão entre 3,6 segundos e 4,3 segundos.

Tabela 3: Representação dos resultados do expressJs

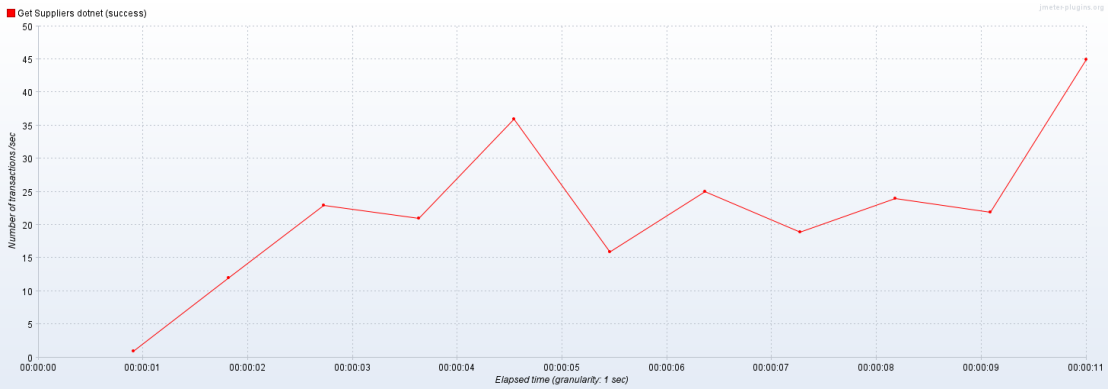
Total	Média	Mínimo	Max	Vazão (requisição/tempo)	Percentual de erro
250	3818 ms	198 ms	6720 ms	11,2 req/sec	0%

No resultado geral (tabela 3), teve um tempo mínimo bom e sem nenhuma requisição com falha.

ASP.NET CORE

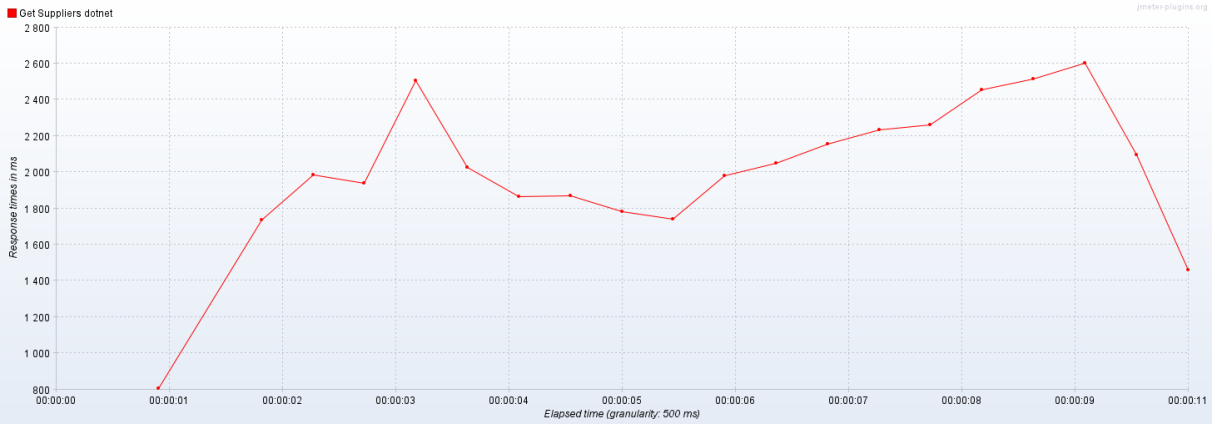
Para executar esses testes, foi levado um total de 11 segundos, sendo a metade do tempo do expressJS.

Figura 21: Transações por segundo durante o tempo de teste



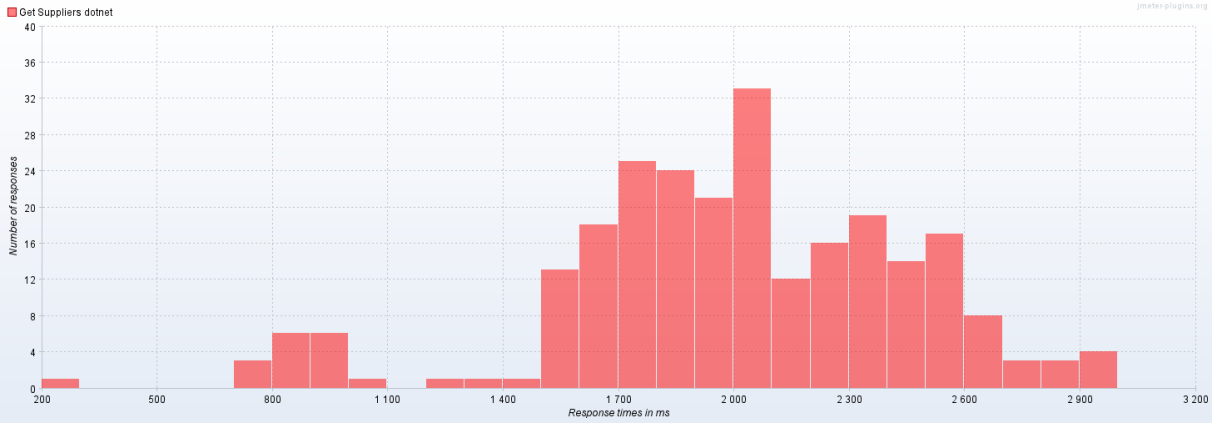
Nesse caso, a maior quantidade de transações por segundo foi alcançada bem próximo às últimas requisições a serem feitas (ver figura 21).

Figura 22: Tempo de resposta de uma request x tempo total do teste



O tempo de resposta (figura 22) oscilou menos que o do Express.js, tendo valores menores com uma queda ao chegar no tempo final do teste.

Figura 23: Distribuição do tempo de resposta das requisições



A maioria do tempo das requisições se encontram entre 1700 a 2000 milissegundos (ver figura 23). Uma diferença de comportamento interessante é que, mesmo tendo um tempo de resposta menor que o Expressjs, o tempo mínimo do dotnet é maior e a quantidade de requests que levaram menos tempo é menor que a do framework de Javascript devido ao comportamento do dotnet de lidar com maior número de requisições no mesmo momento.

Tabela 4: Representação dos resultados do ASP NET CORE

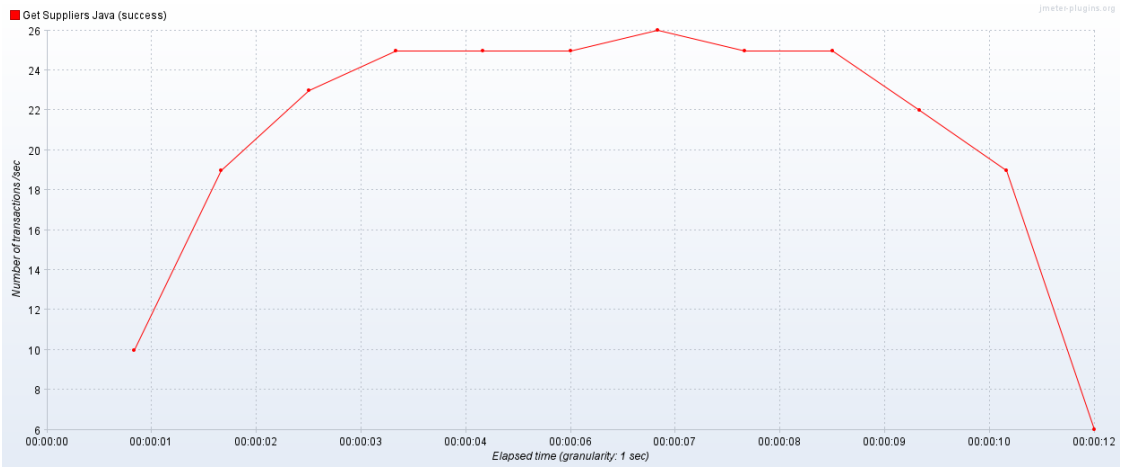
Total	Média	Mínimo	Max	Vazão (requisição/tempo)	Percentual de erro
250	1994 ms	294 ms	2991 ms	23,1 req/sec	0%

A vazão do ASP .NET Core chegou a ser pouco mais de 2 vezes maior que a do ExpressJS. Um dos fatos que pode colaborar nisso é o suporte nativo a multi threads que melhoram a execução concorrente.

Spring Boot

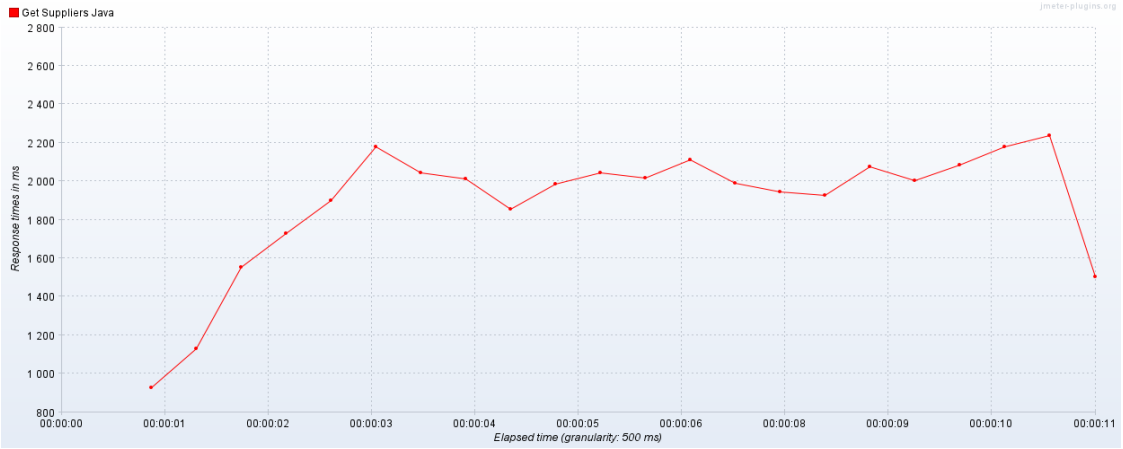
O framework do Java levou 12 segundos para responder todas as requisições.

Figura 24: Transações por segundo durante o tempo de teste



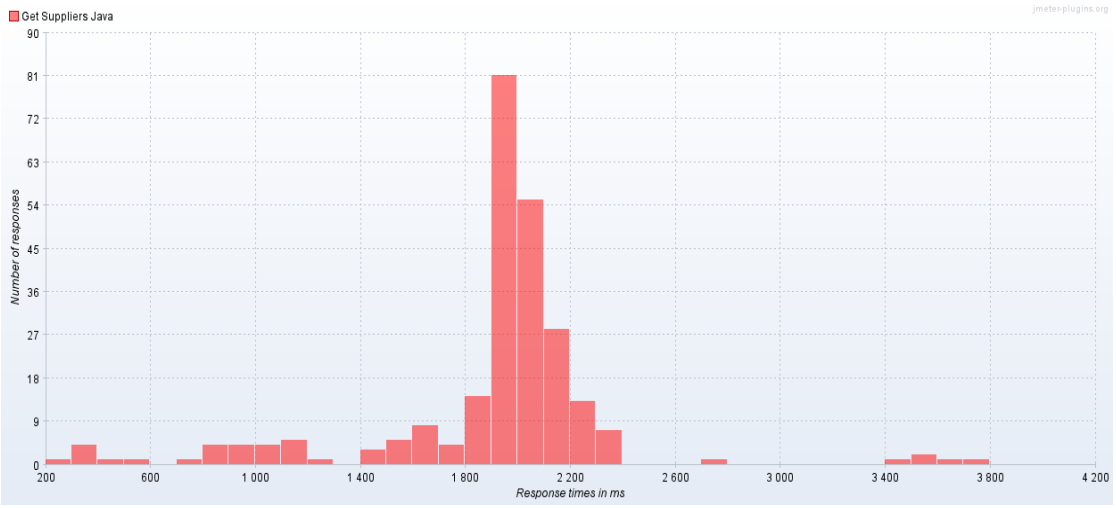
As transações por segundo se mantiveram muito mais estáveis no pico do que os outros dois citados anteriormente.(ver figura 24)

Figura 25: Tempo de resposta durante o tempo do teste



O tempo de gasto total do teste foi de 12 segundos, com oscilações do tempo de resposta (ver figura 25).

Figura 26: Distribuição do tempo de resposta das requisições



O desempenho foi muito parecido com o do framework do C#, mas manteve o tempo mais estável por mais tempo no meio do tempo do teste, conforme a figura 26.No entanto, o Dotnet ainda conseguiu manter um tempo menor, oscilando mais.

Tabela 5: Representação dos resultados do Spring boot

Total	Média	Mínimo	Max	Vazão (requisição/tempo)	Percentual de erro
250	1908 ms	261 ms	3702 ms	22,1 req/sec	0%

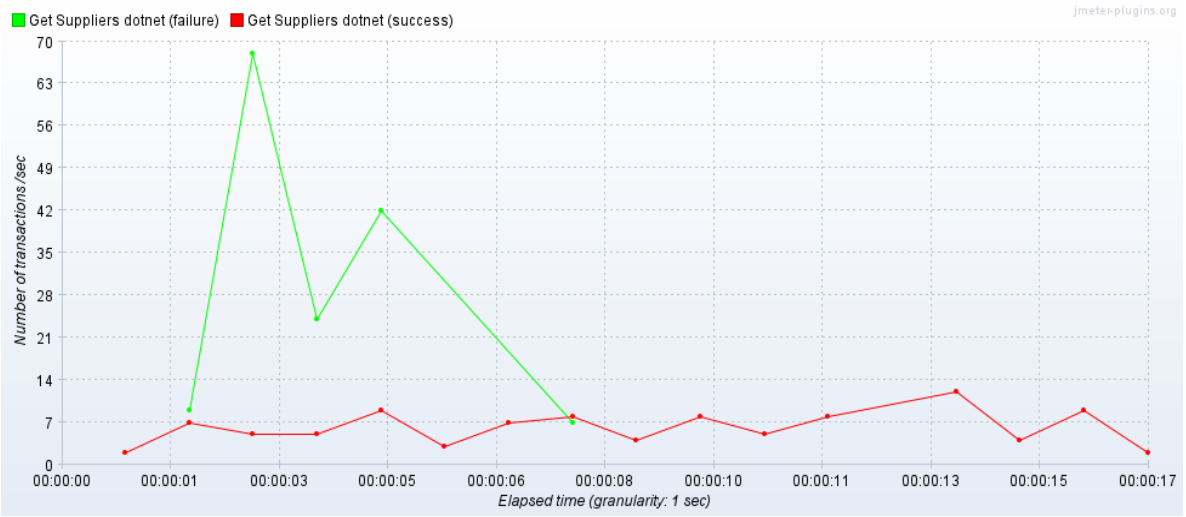
A vazão do ASP .NET CORE e do Spring Boot se mantiveram bem parecidas, com o dotnet conseguindo fazer em média uma requisição a mais por segundo (ver tabela 5). Outro fato a ser considerado é que no tempo máximo do Spring Boot foi pior que o ASP .NET CORE, mesmo os dois possuindo suporte a multi thread nativamente.

Django

O tempo de execução do teste durou um total de 17 segundos, porém grande parte das requisições falharam por motivos do Django no comportamento default com as configurações do ambiente de teste descritas acima não aguentar um pouco mais que 30 requisições por segundo. O número de requisições limitadas foi pequeno devido a limitações de hardware de testes.

No entanto, isso pode ocorrer em produção e devido a isso a documentação do Django recomenda utilizar alguns middlewares na requisição http que melhoram performance por cache, ou até mesmo utilizar o Django com PyPy que funcionaria como um novo compilador para Python, tendo resultados 4,2 vezes mais rápidos em performance de acordo com sua documentação. Como a ideia desse trabalho é comparar os frameworks de maneira mais “nativa” e justa possível, foi utilizado o padrão de todos os frameworks, não utilizando nenhum plugin ou biblioteca a mais além do framework e seu respectivo ORM.

Figura 27: Transações por segundo durante o tempo de teste



O Django começa com as primeiras requisições dando falha (requisições em vermelho) pelo fato de ao se utilizar multithreads, sofrer com problemas de alcançar um limite de conexões abertas no protocolo HTTP (ver figura 27).

A figura 28 mostra o gráfico don tempo de respostas não separando requisições falhadas das de sucesso.

Figura 28: Tempo de resposta durante o tempo do teste

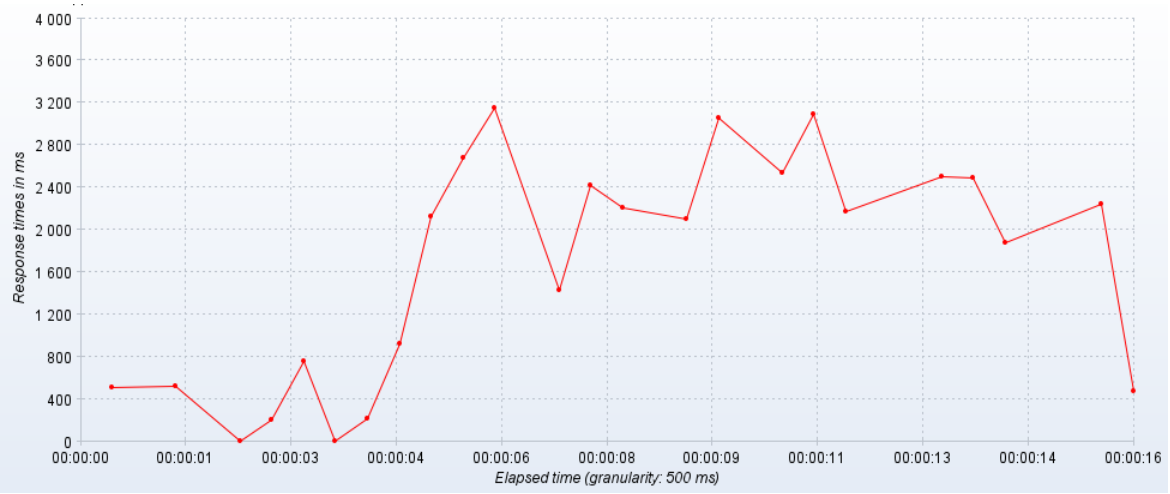
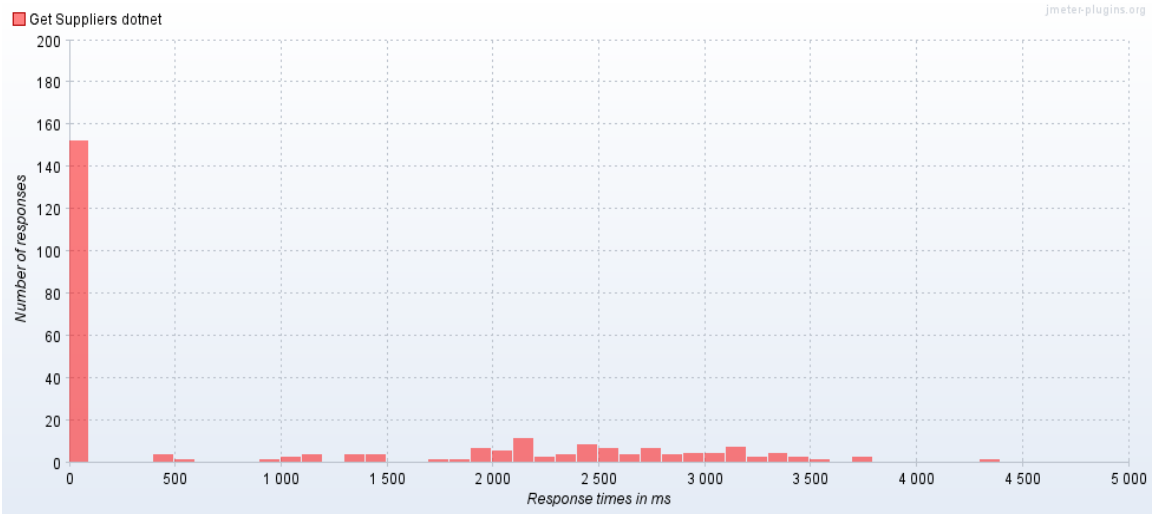


Figura 29: Distribuição do tempo de resposta das requisições



A maioria da distribuição ficou em 0 segundos pelo fato do comportamento ter sobrecarregado a quantidade de requisições ativas (ver figura 29).

Quadro 6: Representação dos resultados do Django

Total	Média	Mínimo	Max	Vazão (requisição/tempo)	Percentual de erro
250	936 ms	0 ms	4337 ms	15,1 req/sec	60,80%

A tabela 6 mostra o resultado geral do teste, com destaque de 60,80% das requisições com falha.

Cenário 2: 250 requisições feitas em 1s, simulando 250 usuários tentando consumir a mesma API ao mesmo tempo.

Express.js

O Express.js levou um total de 30 segundos para responder todas as requisições (ver figura 30).

Figura 30: Tempo de resposta durante o tempo do teste

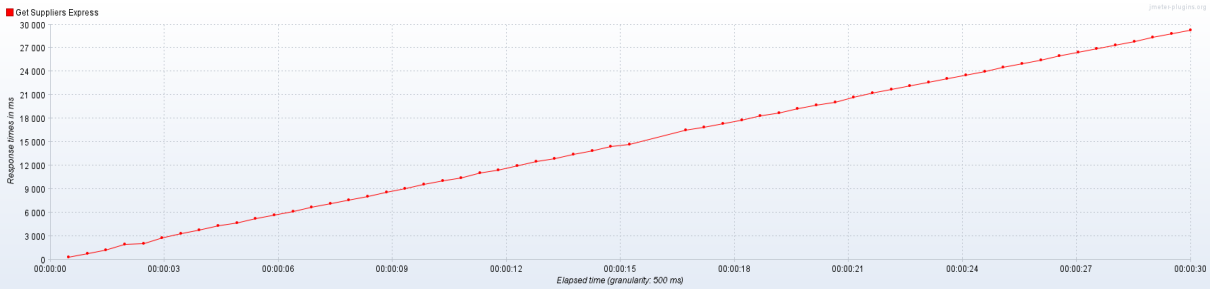
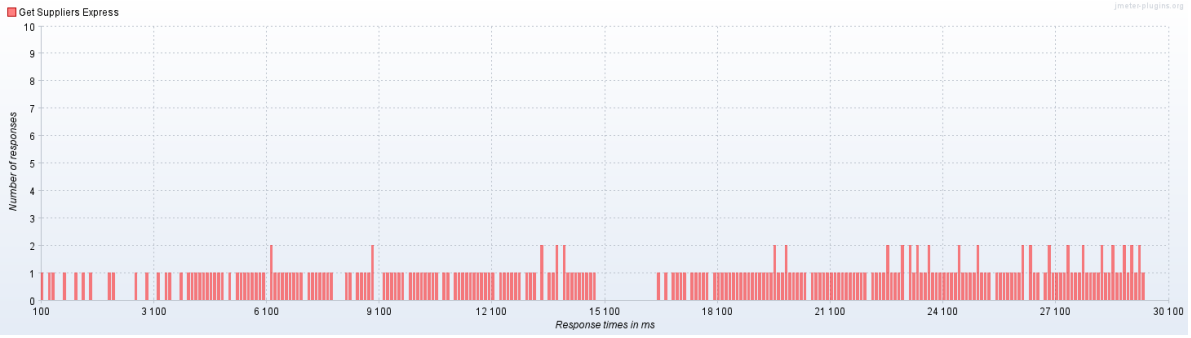


Figura 31: Distribuição do tempo de resposta das requisições



O tempo de resposta subiu conforme o tempo de teste, não apresentando nenhuma queda ou oscilação para baixo, diferente do cenário que teve que lidar com as requisições feitas com intervalo de tempo (figura 31).

Tabela 7: Representação dos resultados do ExpressJs

Total	Média	Mínimo	Max	Vazão (requisição/tempo)	Percentual de erro
250	16821 ms	188 ms	29460 ms	8,2 req/sec	0%

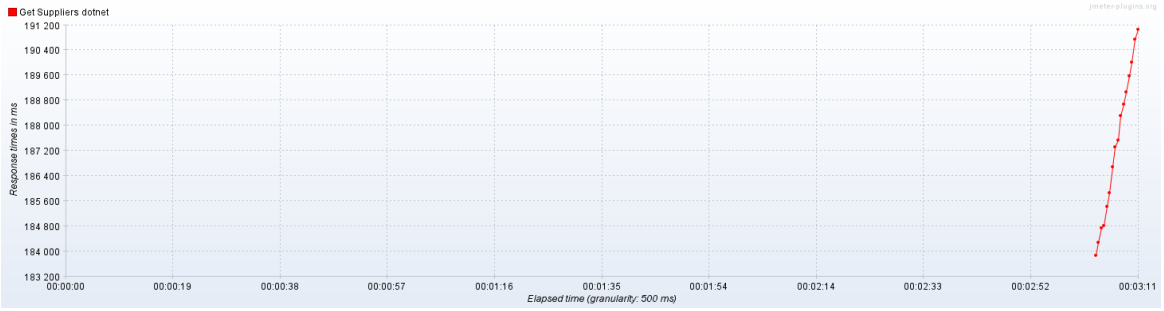
A tabela 7 mostra o resultado geral, com destaque da vazão estar a 8,2 requisições por segundo, mostrando-se ser menor do que o primeiro caso de teste.

ASP.NET CORE

Ele levou 3 minutos e 11 segundos para realizar o teste e teve 20% das requisições retornadas com erro de limite de conexão de pool no banco.

O que aumentou incrivelmente o tempo de resposta quando comparado ao fazer 50 requisições por segundo durante 5 segundos (figura 32).

Figura 32: Tempo de resposta durante o tempo do teste



Ao ver os resultados, é um tanto quanto assustador e contraintuitivo ver que ele demorou praticamente o mesmo tempo para retornar todas as requisições.

O problema está no multi-thread que tenta acessar o banco de dados ao mesmo tempo, em várias threads diferentes. Isso sobrecarrega o banco de dados, podendo até retornar erros de conexão devido ao limite de pool que está atrelada a quantidade de conexões abertas que um banco de dados pode ter sem retornar exceções.

Tabela 8: Representação dos resultados do ASP NET Core

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	187216ms	183396ms	191413ms	1,3 req/sec	20%

O tempo mínimo e o tempo máximo estão muito próximos (tabela 8), isso aconteceu porque o framework utilizou o poder de concorrência para fazer a maioria das consultas do request ao mesmo tempo, no banco de dados.

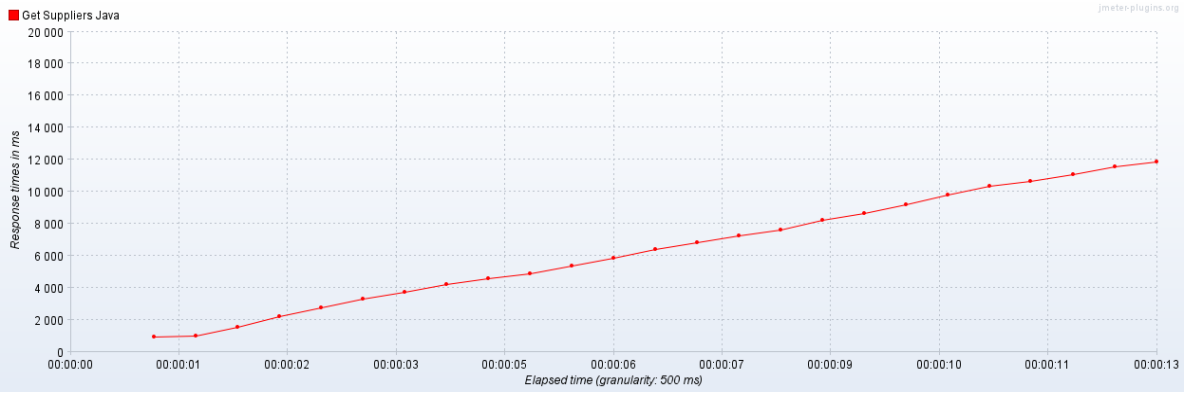
O desempenho foi influenciado pelo banco de dados, que não suportou tantas consultas e conexões ao mesmo tempo. Isso mostra o poder do dotnet em seu multi-thread, pois foi o único a conseguir sobrecarregar o banco de dados no mesmo

cenário. Em uma operação que não dependesse de APIs externas ou de banco de dados, o multithreading do dotnet seria muito mais útil para realizar operações de processamento.

Spring Boot

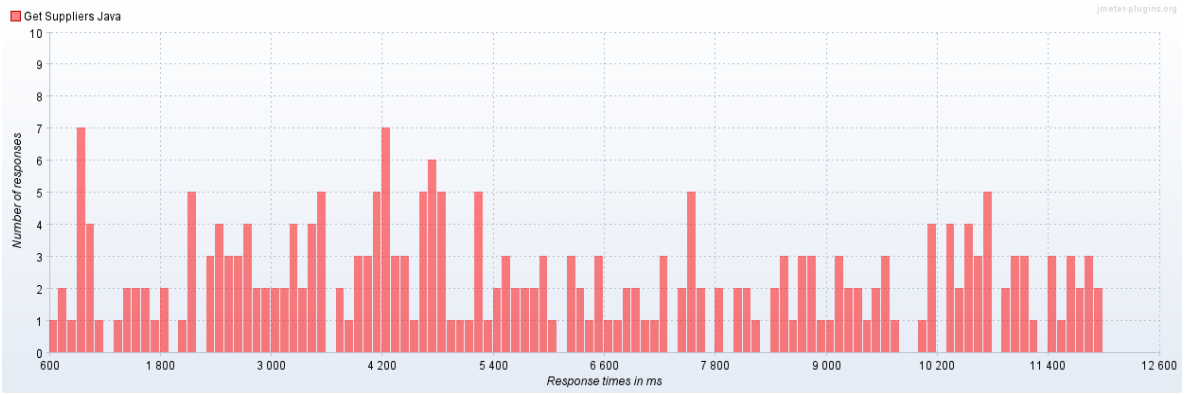
Foi levado um tempo total de 13 segundos para terminar os testes.

Figura 33: Tempo de resposta durante o tempo do teste



Da mesma forma que o Express, o Spring Boot também teve seu tempo de resposta só aumentado, sem nenhuma oscilação para diminuir e depois aumentar (figura 33).

Figura 34: Distribuição do tempo de resposta das requisições



A figura 34 mostra a distribuição do tempo de resposta das requisições do cenário 2.

Tabela 9: Representação dos resultados do Spring boot

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	6019 ms	687 ms	11937 ms	19,3 req/sec	0%

A vazão foi mais lenta do que do cenário anterior, porém ainda conseguiu um bom desempenho para retornar às requisições sem erros (tabela 9).

DJANGO

O tempo do teste durou 5s, tendo o pior desempenho dos 4 frameworks nesse cenário também (figura 34).

Figura 34: Tempo de resposta pelo tempo de teste no django com 250 requisições

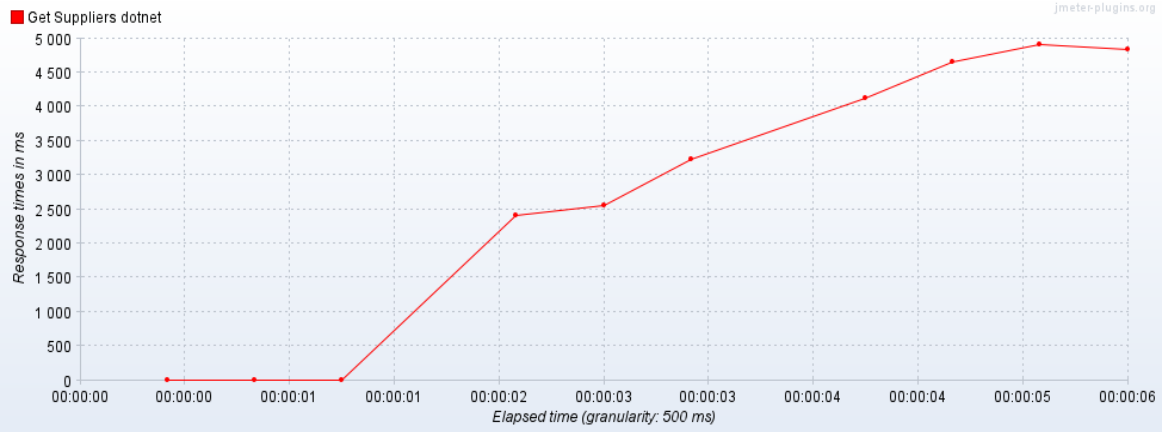
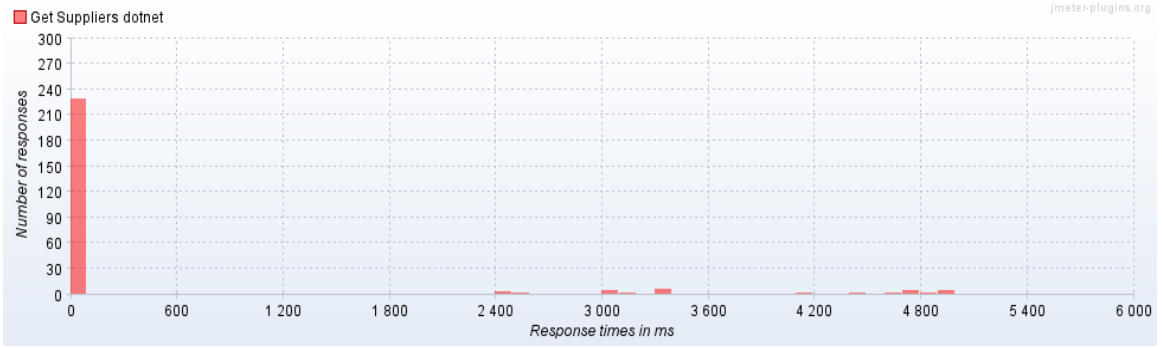


Figura 35: Distribuição do tempo de resposta das requisições



O tempo de distribuição se manteve majoritariamente no 0s devido ao grande número de erros por perda de conexão com cliente que o django teve ao receber as 250 requisições ao mesmo tempo (figura 35).

Quadro 10: Representação dos resultados do Django

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	336 ms	1 ms	4944 ms	42,8 req/sec	91.20%

O percentual de erro aumentou mais ainda nesse cenário mais difícil de lidar, com apenas 22 requisições retornadas com sucesso (tabela 10).

6.2.2 - Resultados GET Suppliers

Com a execução dos testes foi possível entender que cada um tem sua estratégia para lidar com possíveis estresses, o que faz com que resultados diferentes sejam alcançados. Mesmo com resultados diferentes e indicadores que mostrariam qual foi melhor, há outros fatores que podem influenciar nos resultados.

Por exemplo, se o ASP .NET Core estivesse em um ambiente em produção sem limitar a quantidade de conexões abertas com o banco de dados, ele teria o melhor desempenho para esse número de requisições. No entanto, haveria um número maior de requisições que chegariam no mesmo problema.

Para todos os efeitos, em um cenário ideal não é interessante sobrecarregar uma API desse modo sem um tipo de estratégia como orquestração de pods com múltiplas instâncias e um proxy para dividir o estresse ou até mesmo uma “fila” de espera para não haver sobrecarga de infraestrutura externas como outras APIs externas e bancos de dados.

6.2.3 - GET Customers

Esse endpoint irá retornar todos os clientes e seus pedidos, contendo o nome do produto e o preço do produto no momento da compra. Essa query utilizará 6 tabelas diferentes para pegar a informação necessária, tornando assim o endpoint mais pesado. O banco de dados possui um total de 10 mil registros.

Figura 36: Query SQL feita pelo autor para buscar todos os clientes e seus respectivos pedidos com produtos

```
"SELECT DISTINCT c.name as customerName, c.isActive as customerActive, c.id as customerId, " +
"co.orderDate as orderDate, co.id as orderId, spp.price as productPrice, p.name as productName, p.id as productId, " +
"p.details as productDetails " +
"FROM customer c " +
"INNER JOIN customer_order co ON co.customerId = c.id " +
"INNER JOIN supplier_product_pricing_customer_order sppco ON sppco.customerOrderId = co.id " +
"INNER JOIN supplier_product_pricing spp ON spp.id = sppco.supplierProductPricingId " +
"INNER JOIN supplier_product sp ON sp.id = spp.supplierProductId " +
"INNER JOIN product p ON p.id = sp.productId ";
```

Fonte: O autor (2023)

Tendo como base no retorno das requisições, uma coleção desse JSON abaixo:

Figura 37: JSON de retorno na requisição de customers


```
[
  {
    "customerName":"teste_1",
    "customerId":1,
    "customerActive":true,
    "orders":[
      {
        "orderDate":"2023-01-13T01:09:21",
        "orderId":1,
        "products":[
          {
            "name":"name_1",
            "id":1,
            "details":"details details",
            "price":20.89
          },
          {
            "name":"name_2",
            "id":2,
            "details":"details details",
            "price":20.89
          }
        ]
      }
    ]
  }
]
```


A parte lógica implementada para esse endpoint nos 4 serviços é mostrada nas figuras 37, 38, 39 e 40.

Figura 37: Implementação parte logica no ASP.NET CORE

```
public async Task<List<CustomerOrderReport>> getAllCustomerAndOrders()
{
    List<CustomerFullReport> result = await this.customerRepository.getFullCustomerData();
    Dictionary<int, CustomerOrderReport> customerHashMap = new();

    foreach (CustomerFullReport customerReport in result)
    {
        var isOnDict = customerHashMap.ContainsKey(customerReport.customerId);
        if (isOnDict)
        {
            var currentCustomerReport = customerHashMap[customerReport.customerId];
            if (currentCustomerReport != null)
            {
                ProductDTO newProduct = this.createProduct(customerReport);
                OrderDTO currentOderDTO = currentCustomerReport.orders.Find(ord => ord.orderId == customerReport.orderId);

                if (currentOderDTO != null)
                {
                    currentOderDTO.products.Add(newProduct);
                }
                else
                {
                    OrderDTO newOrder = this.createOrderDTO(customerReport);
                    newOrder.products.Add(newProduct);
                    currentCustomerReport.orders.Add(newOrder);
                }
            }
        }
        else
        {
            CustomerOrderReport newCustomerOrderReport = new CustomerOrderReport(customerReport.customerName,
                customerReport.customerId, customerReport.customerActive, new List<OrderDTO>());

            ProductDTO productDTO = this.createProduct(customerReport);
            OrderDTO orderDTO = this.createOrderDTO(customerReport);
            orderDTO.products.Add(productDTO);

            newCustomerOrderReport.orders.Add(orderDTO);

            customerHashMap.Add(newCustomerOrderReport.customerId, newCustomerOrderReport);
        }
    }
    return customerHashMap.Values.ToList();
}
```

Figura 38: Implementação parte lógica no Spring

```
public List<CustomerOrderReport> getAllCustomerAndOrders() {
    List<CustomerFullReportDTO> reportDTOS = this.customerRepository.getFullCustomerData();
    final Map<Integer, CustomerOrderReport> customerOrderDTOHashMap = new HashMap<>();
    for (CustomerFullReportDTO customerReport : reportDTOS) {
        CustomerOrderReport currentCustomerReport = customerOrderDTOHashMap.get(customerReport.getCustomerId());
        if (Objects.nonNull(currentCustomerReport)) {
            ProductDTO newProduct = this.createProductByCustomerReport(customerReport);
            OrderDTO currentOrderDTO = currentCustomerReport.getOrders().stream()
                    .filter(orderDTO -> orderDTO.getOrderId().equals(customerReport.getOrderId())).findFirst().orElse( other: null);
            if (Objects.nonNull(currentOrderDTO)) {
                currentOrderDTO.getProducts().add(newProduct);
            } else {
                OrderDTO newOrder = this.createOrderDTO(customerReport);
                newOrder.getProducts().add(newProduct);
                currentCustomerReport.getOrders().add(newOrder);
            }
        } else  {
            CustomerOrderReport newCurrentCustomerReport = new CustomerOrderReport();
            newCurrentCustomerReport.setCustomerId(customerReport.getCustomerId());
            newCurrentCustomerReport.setCustomerActive(customerReport.getCustomerActive());
            newCurrentCustomerReport.setCustomerName(customerReport.getCustomerName());
            newCurrentCustomerReport.setOrders(new ArrayList<>());

            ProductDTO newProduct = this.createProductByCustomerReport(customerReport);
            OrderDTO newOrderDto = this.createOrderDTO(customerReport);
            newOrderDto.getProducts().add(newProduct);
            newCurrentCustomerReport.getOrders().add(newOrderDto);

            customerOrderDTOHashMap.put(newCurrentCustomerReport.getCustomerId(), newCurrentCustomerReport);

        }
    }
    return customerOrderDTOHashMap.values().stream().toList();
}
```

Figura 39: Implementação parte lógica no ExpressJS

```
async getAllCustomerAndOrders() {
  const sqlResults = await this.repository.getAllCustomerReport();
  const customerOrderMap = new Map();

  for (const customerReport of sqlResults) {
      const currentCustomerReport = customerOrderMap.get(customerReport.customerId);
      if (currentCustomerReport) {
          const newProduct = this.createProductByCustomerReport(customerReport);
          const currentOrder = currentCustomerReport.orders.find(order => order.orderId === customerReport.orderId);
          if (currentOrder) {
              currentOrder.products.push(newProduct);
          } else {
              const newOrder = this.createOrderDTO(customerReport);
              newOrder.products.push(newProduct);
              currentCustomerReport.orders.push(newOrder)
          }
      } else {
          const newCurrentCustomerReport = {
              customerName: customerReport.customerName,
              customerId: customerReport.customerId,
              customerActive: customerReport.customerActive === 1 ? true : false,
              orders: [],
          }

          const newProduct = this.createProductByCustomerReport(customerReport);
          const newOrder = this.createOrderDTO(customerReport);
          newOrder.products.push(newProduct)

          newCurrentCustomerReport.orders.push(newOrder)

          customerOrderMap.set(newCurrentCustomerReport.customerId, newCurrentCustomerReport);
      }
  }
  return Array.from(customerOrderMap, ([_, value]) => value);
}
```

Figura 40: Implementação parte lógica no Django

```
def getAllCustomerAndOrders(self):
    repository = CustomerRepository()
    sqlResults = repository.getAllCustomerReport()
    customerOrderMap = dict()
    for customerReport in sqlResults:
        currentCustomerReport = customerOrderMap.get(customerReport.get('customerId'))
        newProduct = self.__createProduct(customerReport)
        if (currentCustomerReport is not None):
            currentOrder = self.__findOrder(currentCustomerReport, customerReport.get('orderId'))
            if(currentOrder is not None):
                currentOrder.get('products').append(newProduct)
            else:
                newOrder = {
                    'orderId': customerReport.get('orderId'),
                    'orderDate': customerReport.get('orderDate'),
                    'products': [],
                }
                newOrder.get('products').append(newProduct)
        else:
            newCurrentCustomerReport = {
                'customerName': customerReport.get('customerName'),
                'customerId': customerReport.get('customerId'),
                'customerActive': customerReport.get('customerActive') == 1,
                'orders': [
                    {
                        'orderId': customerReport.get('orderId'),
                        'orderDate': customerReport.get('orderDate'),
                        'products': [newProduct],
                    }
                ],
            };
            customerOrderMap[newCurrentCustomerReport.get('customerId')] = newCurrentCustomerReport
    return customerOrderMap.values()
```

Cenário 1: 50 requisições por segundo, durante 5s, totalizando 250 requisições

Express.js

Com essa query mais pesada, o ExpressJS levou 1 minuto e 11 segundos para responder às requisições (figura 41).

Figura 41: Tempo de resposta pelo tempo de teste

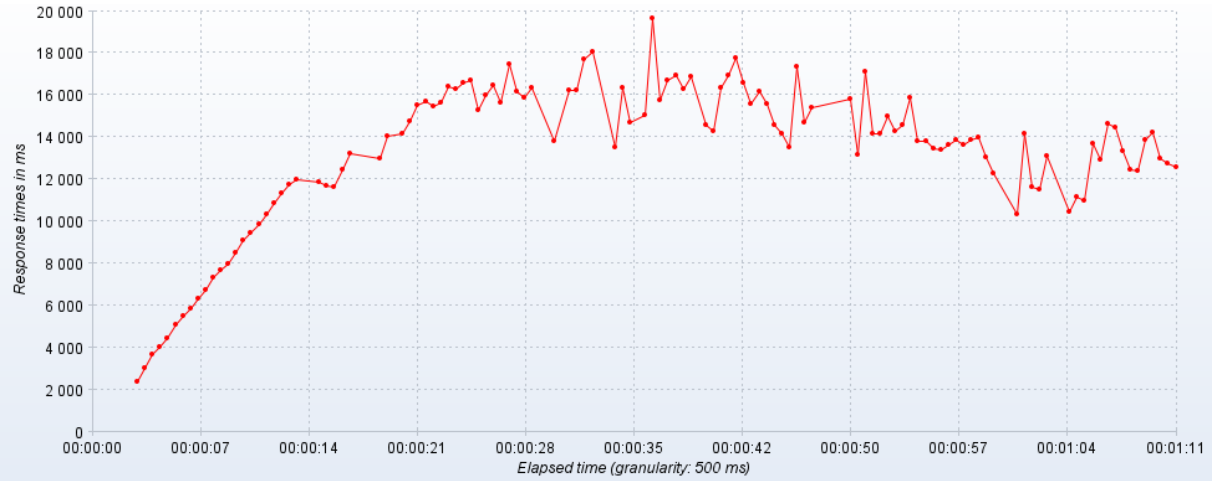


Figura 42: Distribuição do tempo de resposta das requisições

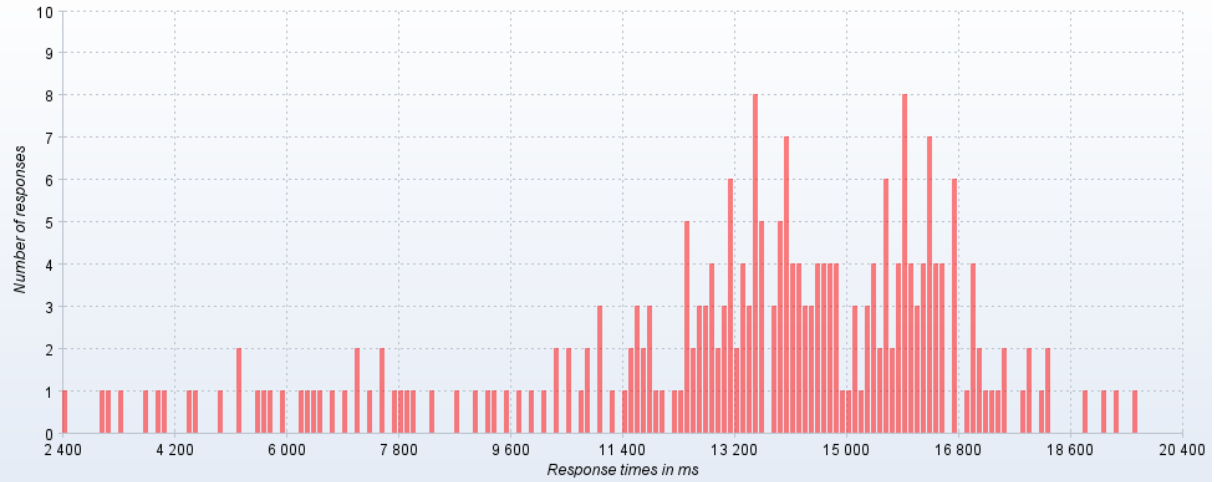


Tabela 11: Representação dos resultados do Expressjs

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	13288 ms	2410 ms	19646 ms	3,5 req/sec	0%

A figura 42 mostra a distribuição do tempo de resposta e a tabela 11 mostra o resultado geral. O tempo express demorou consideravelmente mais no endpoint customer do que no supplier, devido à quantidade de tabelas a serem processadas. No entanto, ainda conseguiu retornar todas as requisições sem erros.

Spring Boot

O tempo total do teste foi de 1 minuto (figura 43), com todas as requisições realizadas com sucesso.

Figura 43: Tempo de resposta pelo tempo de teste

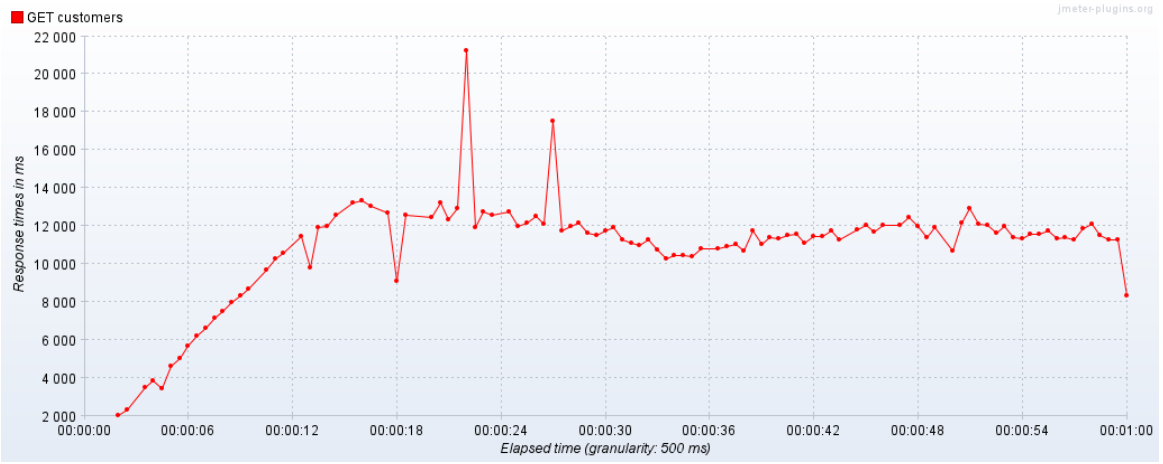
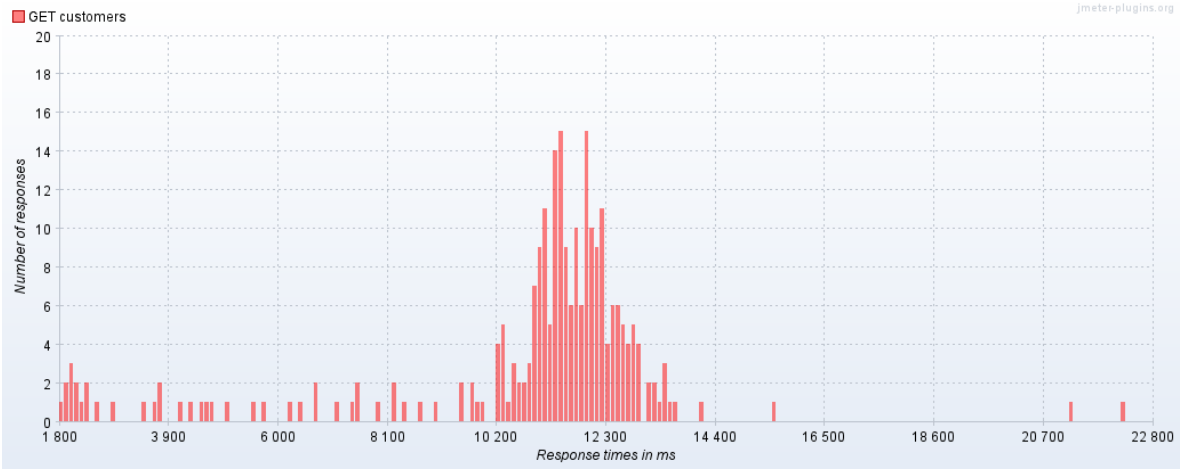


Figura 44: Distribuição do tempo de resposta das requisições



A figura 44 mostra a distribuição do tempo de respostas das requisições, e podemos observar a maior concentração em algo em torno de 1200 milisegundos.

Quadro 12: Representação dos resultados do Spring boot

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	10739 ms	1861 ms	22298 ms	4,2 req/sec	0%

O tempo de teste foi 11 segundos mais rápido que o Express.js, com sua vazão de requisições sendo um pouco maior (tabela 12). O tempo médio, mínimo e máximo também foram melhores.

ASP.NET CORE

Ele levou 55 segundos para responder todas as 250 requisições, com todas as requisições retornadas com sucesso (figura 45).

Figura 45: Tempo de resposta pelo tempo de teste

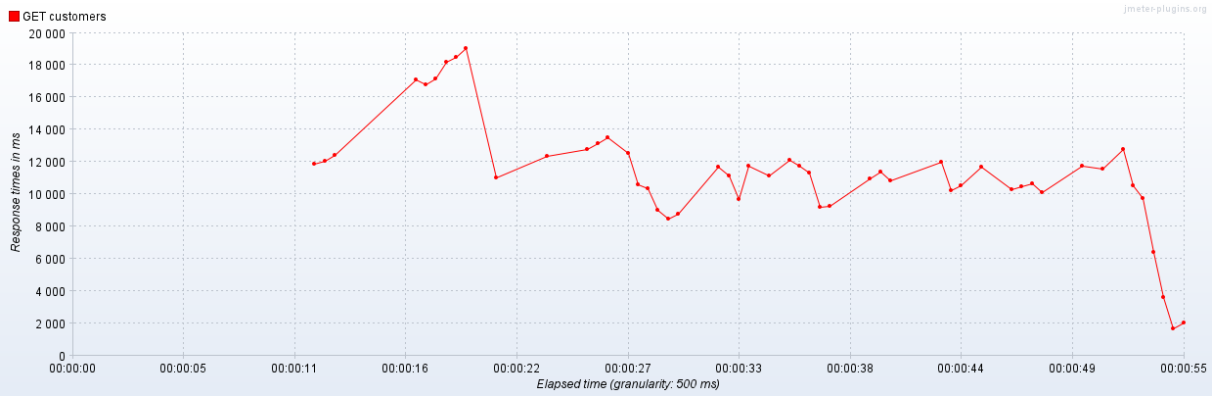
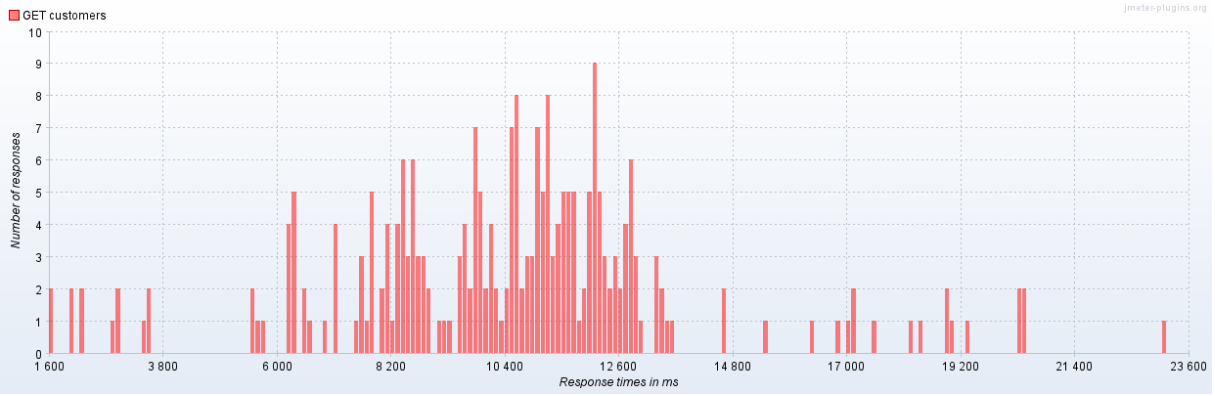


Figura 46: Distribuição do tempo de resposta das requisições



A figura 46 mostra o comportamento da distribuição de tempo de resposta, mostrando que todas as distribuições dos 4 frameworks se mostram bastante diferentes.

Quadro 13: Representação dos resultados do ASP.NET CORE

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	10527 ms	1687 ms	23132 ms	4,5 req/sec	0%

Até agora o mais performático em questões de vazão (tabela 13), o comportamento desse framework consegue ser bom ao não conseguir quebrar o limite de conexões ativas no banco de dados. Porém, esse comportamento padrão é perigoso mesmo com um limite de conexões maiores, não é interessante deixar esse tipo de sobrecarga acontecer se porventura a quantidade de pessoas mandando requisições consiga ser maior que a quantidade máxima de conexões ativas com o banco.

DJANGO

O tempo total do teste foi de 47 segundos (figura 47), porém com um percentual de erro de 64,2%. Sendo um resultado não ótimo, pois mais da metade das requisições retornaram sem sucesso.

Figura 47: Tempo de resposta pelo tempo de teste

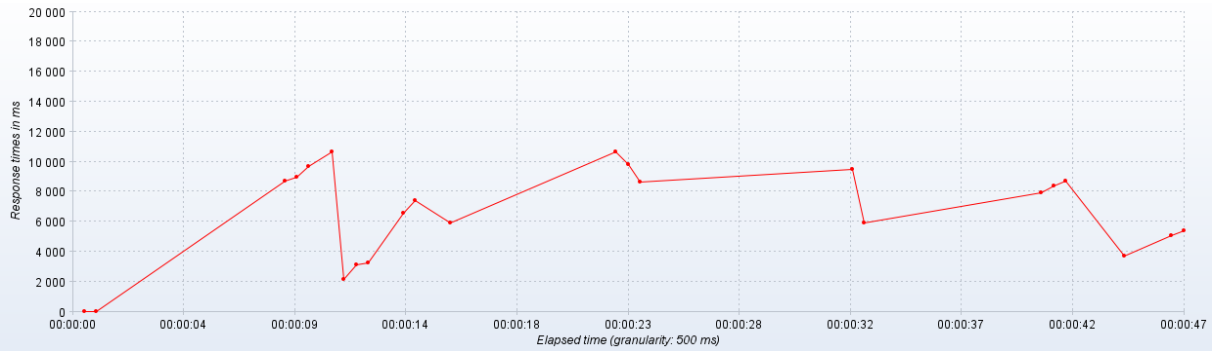
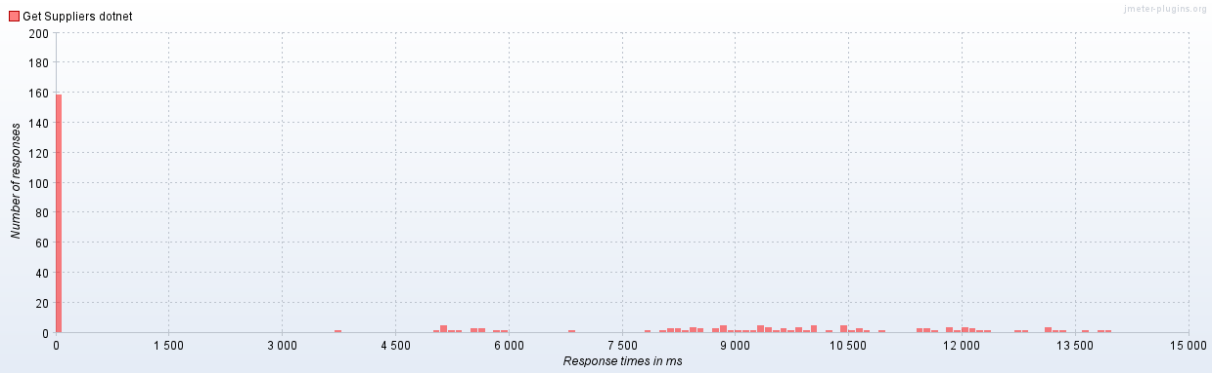


Figura 48: Distribuição do tempo de resposta das requisições



A maioria das requisições teve um tempo de resposta de 0 segundos (figura 48) devido ao fato do framework, nessa implementação e no ambiente de testes do autor não aguentar segurar um pouco mais de 35 requisições por segundo. Tendo no seu comportamento as primeiras requisições retornando com erro e só após ter um número de requisições menor para lidar, o framework começa a retornar algumas com sucesso. O erro não se dá por limitações de conexões com o banco de dados, mas sim de limitações de requisições ativas.

Tabela 14: Representação dos resultados do Django

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	3531 ms	0ms	13971 ms	5,3 req/sec	63,2%

A tabela 14 mostra o percentual de erro de 63,2% com as requisições falhadas devido aos problemas discutidos.

Cenário 2: 250 requisições feitas em 1s, simulando 250 usuários tentando consumir a mesma API ao mesmo tempo.

Express.js

O tempo total de testes foi de 1 minuto e 6 segundos (figura 49), com o comportamento do tempo de resposta seguindo aumentando conforme o tempo e com um percentual de 3,2% de requisições com erro.

Figura 49: Tempo de resposta pelo tempo de teste

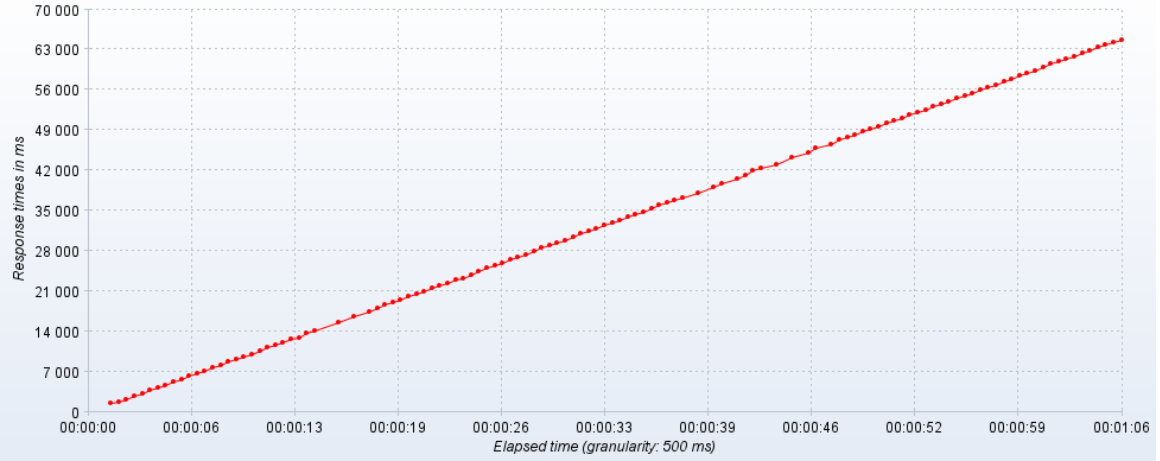
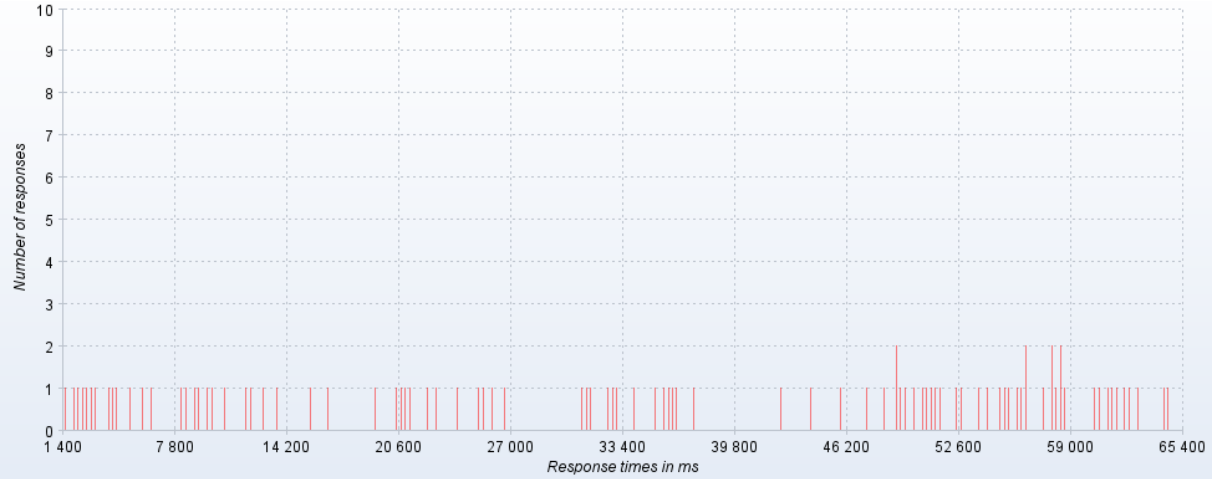


Figura 50: Distribuição do tempo de resposta das requisições



A figura 50 mostra que a distribuição desse deste ficou praticamente com um tempo de resposta para cada requisição.

Tabela 15: Representação dos resultados do Expressjs

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	36926 ms	1455 ms	64642 ms	3,8 req/sec	3,2%

Esse número de requisições feitas no mesmo segundo alcança o ponto em que o Express.js nessas condições começaria a ter problemas de limites de conexões ativas com o banco de dados com 3,2% das requisições falhadas (tabela 15).

Spring Boot

O tempo total do teste foi de 40 segundos (figura 51), com 26,8% (tabela 16) das requisições atingindo o limite de conexões ativas com o banco de dados e assim tendo retornado como falha.

Figura 51: Tempo de resposta pelo tempo de teste

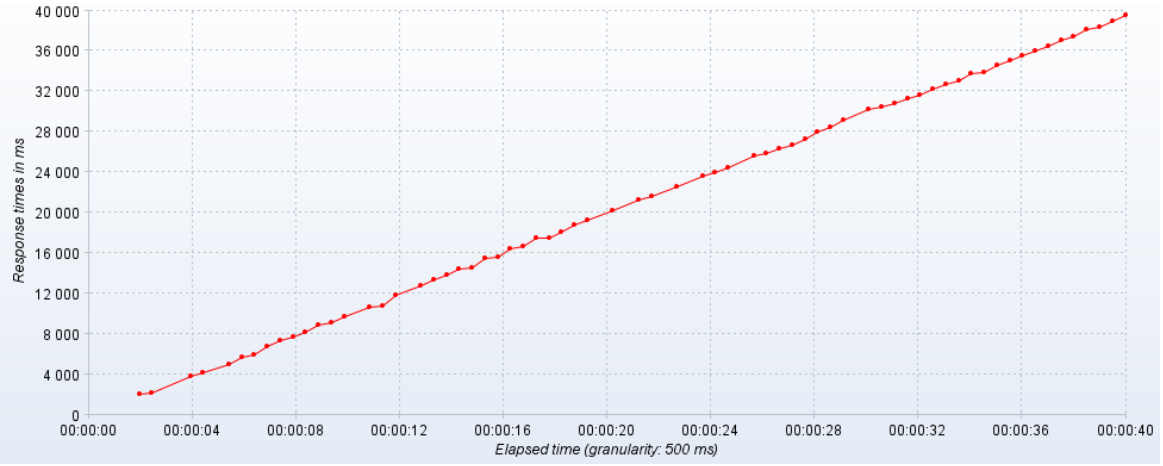
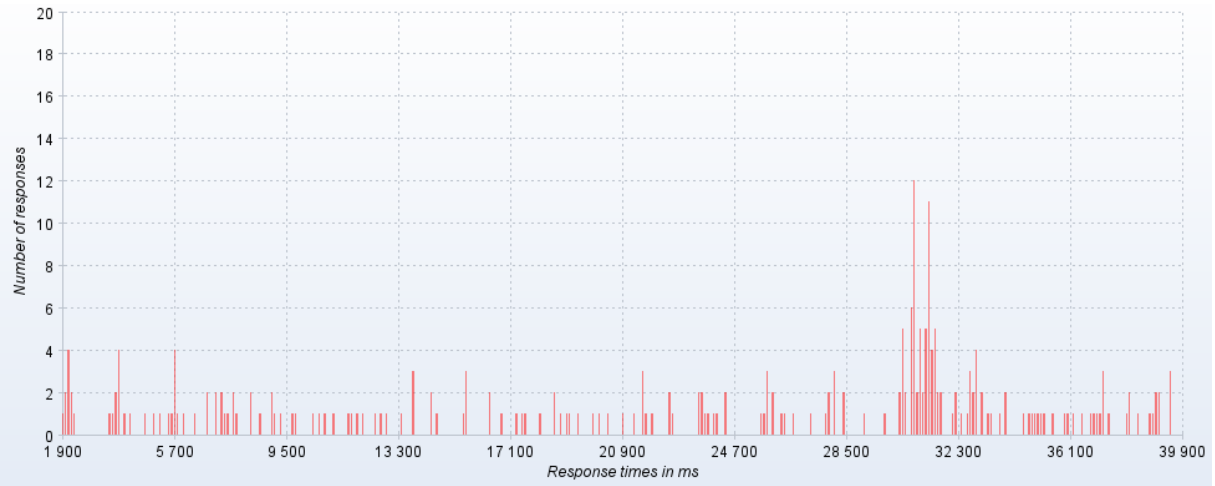


Figura 52: Distribuição do tempo de resposta das requisições



A distribuição (figura 52) foi totalmente diferente da distribuição do Express.js (figura 50).

Quadro 16: Representação dos resultados do Expressjs

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	23377 ms	1943 ms	39542 ms	6,2 req/sec	26,8%

ASP.NET CORE

O tempo total do teste foi de 3 minutos e 36 segundos (figura 53), tendo alcançado o maior tempo para realizar todas as requisições.

Figura 53: Tempo de resposta pelo tempo de teste

Tabela17: Representação dos resultados do ASP .NET Core

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	194206 ms	178465 ms	216216 ms	1,2 req/sec	19,2%

Como esperado com os testes anteriores, o tempo mínimo e máximo ficam um pouco próximos e as requisições demoram para serem retornadas devido ao sobrecarregamento do banco de dados feitas pelo comportamento das múltiplas threads desse framework, tendo seu desempenho influenciado por limitações externas, nesse caso, o banco de dados (tabela 17).

DJANGO

O tempo total do teste foi de 14 segundos (figura 54), sendo o menor tempo entre os frameworks, porém isso aconteceu devido à maioria das requisições falhar quase instantaneamente, com um percentual de erro de 88,8%. Esse cenário aconteceu devido a limitações do framework com o hardware do ambiente de testes usado pelo autor. As requisições foram descartadas sem mesmo antes de chegar a ativar conexão com o banco de dados.

Figura 54: Tempo de resposta pelo tempo de teste

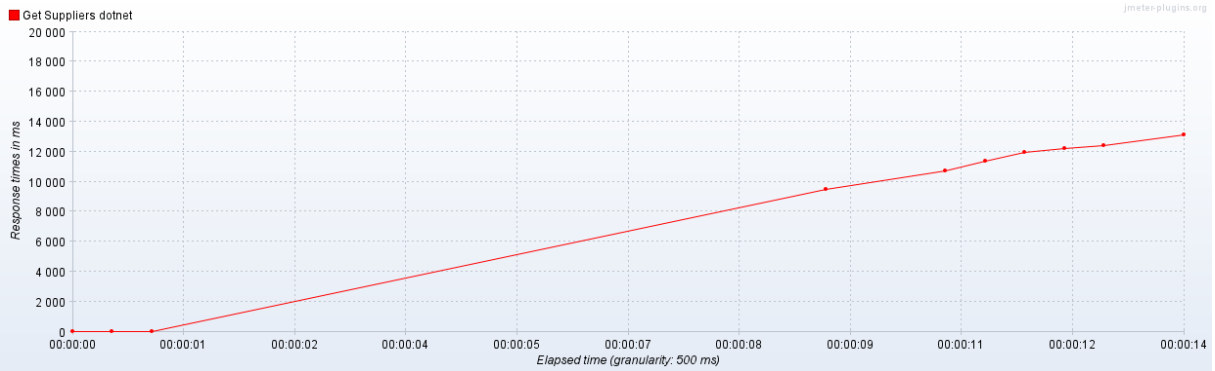


Tabela 18: Representação dos resultados do Expressjs

Total	Média	Mínimo	Max	Vazão(requisição/tempo)	Percentual de erro
250	1319 ms	0ms	13165 ms	17,9 req/sec	88,8%

A tabela 18 mostra que o Django tem um percentual de erro de 88,8%, muito mais alto do que os outros frameworks.

6.2.4 - Resultados GET Customers

Os resultados foram parecidos com o do endpoint suppliers, porém o tempo de resposta de todos, nos que obtiveram mais requisições de sucesso do que falha, foi aumentado devido a query ser mais pesada. Deste modo, este endpoint conseguiu alcançar o ponto de estresse mais cedo em questões em número de requisições para alcançar algum tipo de falha na resposta das requisições.

7 LIÇÕES APRENDIDAS

7.1 Diferenças

Ao tentar comparar os frameworks, percebi o quanto eles são diferentes, não só por questões da sintaxe ou dos paradigmas da linguagem, mas sim da forma que eles lidam com as entradas e saídas. Esse fato influencia completamente no tempo de respostas das requisições e de possíveis erros que podem acontecer ao levar o sistema a testes de estresse.

7.2 Por que usar queries nativas?

Durante o desenvolvimento foi percebido que os ORMs dos frameworks tinham filosofias diferentes para lidar com relacionamento e mapeamento, podendo ter seu

desempenho influenciado pela maneira como o banco de dados foi modelado. Além disso, os retornos de objetos do ORM eram bastante diferentes, o que poderia levar a ter diferentes pós-processamentos para chegar ao resultado esperado para ser retornado na request. Portanto, foi decidido usar queries nativas SQL, pois assim as únicas dependências externas são a conexão com o banco de dados e a execução da query, que retorna o mesmo objeto para todos os frameworks.

7.3 Comportamentos diferentes

A maneira padrão que os frameworks lidam com múltiplas requests simultâneas é diferente, o que faz com que existam resultados muito diferentes, até mesmo de erros por exceder o limite de conexões ativas com o banco de dados aconteça mais rapidamente ou até mesmo não conseguindo manter a request viva antes de conseguir concluí-la.

Se ao receber múltiplas requisições ao mesmo tempo, o framework tentar acessar o banco de dados ao mesmo tempo (utilizando-se de múltiplas threads), o tempo de resposta pode ser muito maior por não ter um gargalo de espera devido ao fator limitante da capacidade do banco de dados que foi utilizado nesta pesquisa. O que é mostrado é que ele poderia trabalhar com concorrência muito bem se não depender de fatores externos como outras APIs ou conexões com banco de dados.

Por exemplo, Express.js obteve desempenho geral melhor que o Django, mesmo não contando com múltiplas threads para afunilar a quantidade de requisições e consequentemente não teve problemas de estourar o limite de conexões ativas no banco de dados.

8 CONCLUSÃO

Escolher um framework vai além de desempenho, mas também é considerado fatores como intimidade com a linguagem de programação, suporte a bibliotecas, e de melhor desempenho em cenários específicos que seriam o caso de uso de algum sistema.

Com os resultados dos testes feitos no ambiente mencionado com as configurações padrões de cada framework e tendo tentado fazê-lo de maneira justa com a mesma implementação feita pelo autor, foram alcançados resultados bastantes diferentes, os quais não conseguem afirmar qual é o mais performático de maneira geral.

Porém, pode-se dizer que nos testes feitos, nesse sistema, o ASP.NET Core e Spring Boot tiveram o melhor desempenho em termos de processamento nos tempos de resposta das requisições nos cenários de teste. Certamente seriam boas escolhas ao se olhar para problemas em que se teria um banco de dados robusto ou que fosse necessário velocidade de processamento com concorrência.

Para casos mais simples que não precisassem se preocupar tanto com desempenho, o Express.js se sairia muito bem por possuir uma configuração simples, ser em javaScript que é a linguagem de programação mais utilizada no mundo nos últimos anos ou por ter diversas bibliotecas para diferentes casos de usos, dos mais simples aos complexos.

O Django seria uma boa opção para programadores Python e para casos de uso que as bibliotecas existentes sanaram algum tipo de problema que o caso de uso do sistema teria. E caso seja necessário mais desempenho, a documentação tem um “workaround” sobre isso com mais dependências para serem instaladas e configurações a serem feitas.

Para trabalhos futuros, seria interessante tentar fazer uma comparação mais justa com um ambiente de testes externo, banco de dados de produção, docker para ter múltiplas instâncias da API e talvez utilizar implementações que favoreçam cada framework para mostrar o máximo de desempenho que cada um pode entregar com suas características distintas.

Outrossim, seria interessante também expandir a metodologia de testes, fazendo testes em mais cenários para poder obter mais detalhes de comportamento em níveis de estresse.

REFERÊNCIAS

APACHE SOFTWARE FOUNDATION. Apache JMeter - Apache JMeterTM. Disponível em: <https://jmeter.apache.org/>. Acesso em: 1 mar 2023.

DHALLA, H. K. A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core. Journal of Physics: Conference Series, v. 1933, n. 1, p. 012041, 1 jun. 2021.

Dalbard, Axel, and Jesper Isacson. "Comparative study on performance between ASP. NET and Node. js Express for web-based calculation tools." (2021).

What is a REST API? Definition and Principles - Seobility Wiki. Disponível em: <https://www.seobility.net/en/wiki/REST_API>.Acesso em: 1 mar 2023.

Databases | Django documentation | Django. Disponível em: <https://docs.djangoproject.com/en/4.1/ref/databases/>.Acesso em: 1 mar 2023.

TEAM, T. P. PyPy. Disponível em: <https://www.pypy.org/>.Acesso em: 1 mar 2023.