ALTICE LABS

OCT 2025

Relatório Técnico

labseq sequence.

Yanis Marina Faquir

ALTICE LABS

SUMÁRIO EXECUTIVO

Este projeto implementa uma API REST em Java (Quarkus) para calcular valores da sequência matemática LabSeq, com uma interface web em Angular para interação do utilizador. A solução atende todos os requisitos especificados, incluindo:

- Cálculo da sequência: l(n) = l(n-4) + l(n-3)
- Performance otimizada: l(100.000)
- Cache inteligente para cálculos intermediários
- Documentação OpenAPI completa (Swagger UI)
- Interface web interativa com histórico e gráficos
- Testes unitários

INSTALAÇÃO E EXECUÇÃO

Repositório Git:
https://github.com/yanisfaquir/FullStack_Exercise_YanisFaquir

---

Run: docker-compose up -d

Acesso:

- **Frontend (Angular): http://localhost:4200**
- **Backend API: http://localhost:8080/labseq/{n}**
- **Swagger UI: http://localhost:8080/swagger-ui**
- **Health Check: http://localhost:8080/labseq/health**

RESULTADOS

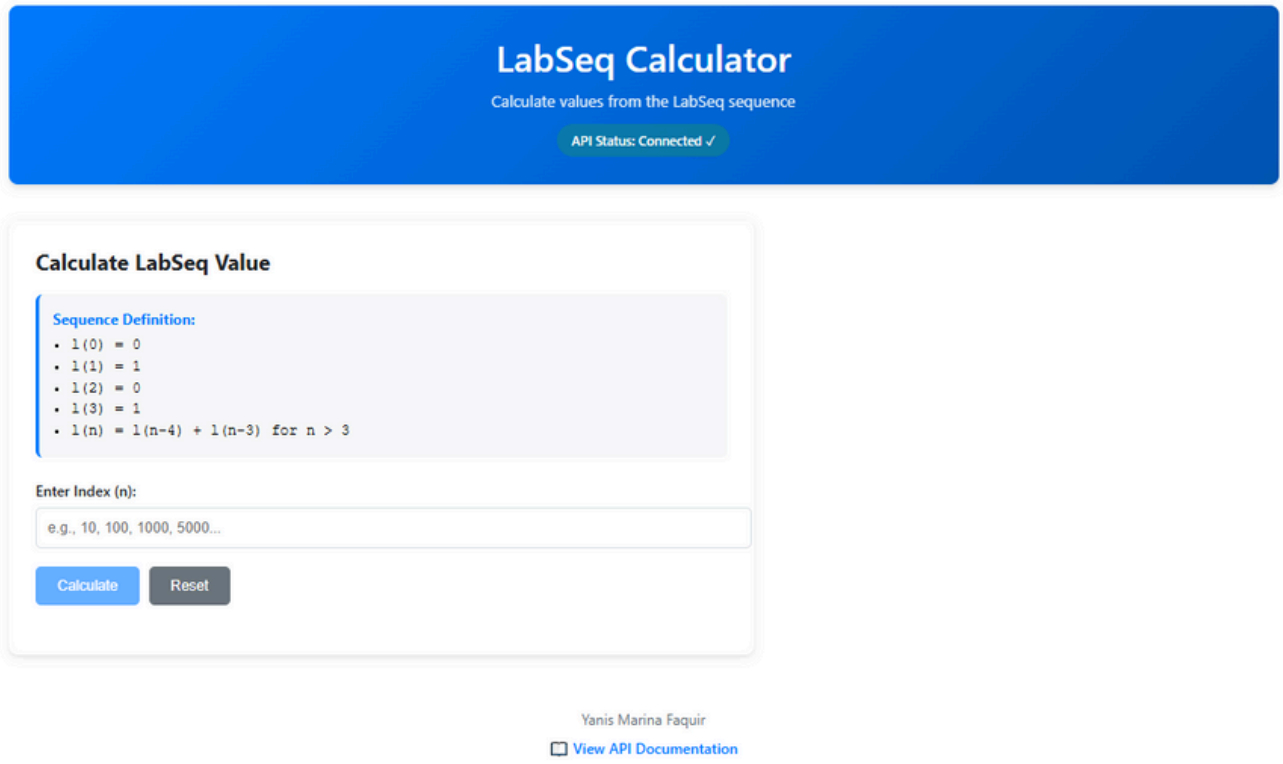


Figure 1 – Home Page

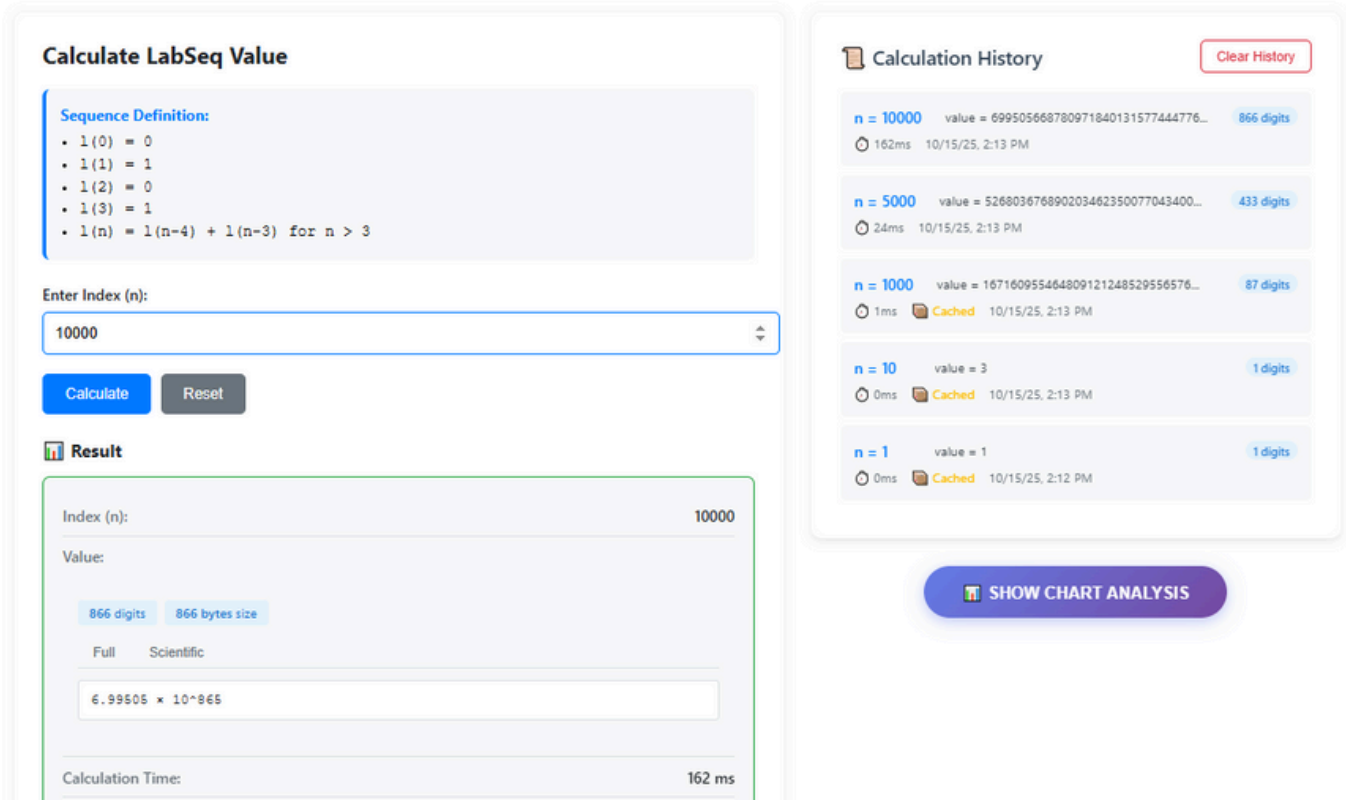


Figure 2 – Calculation History

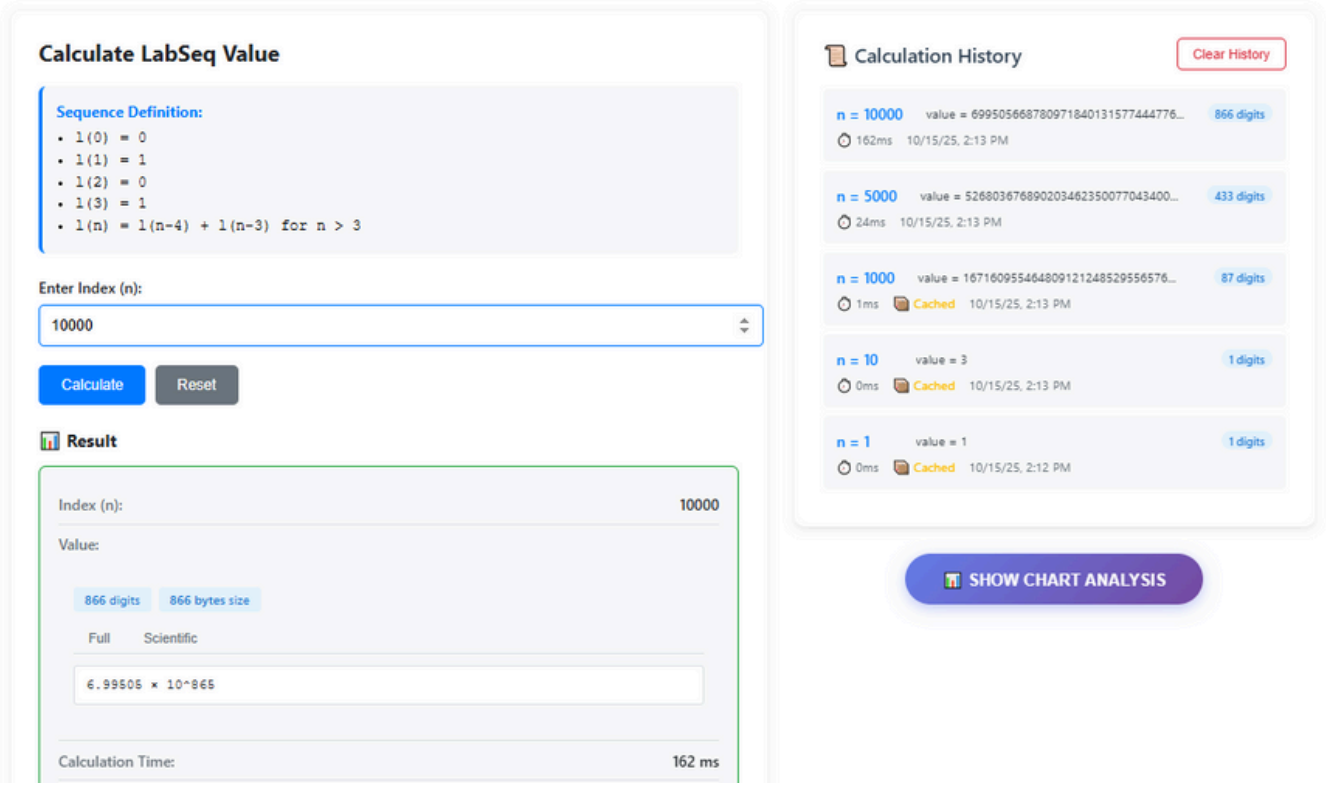
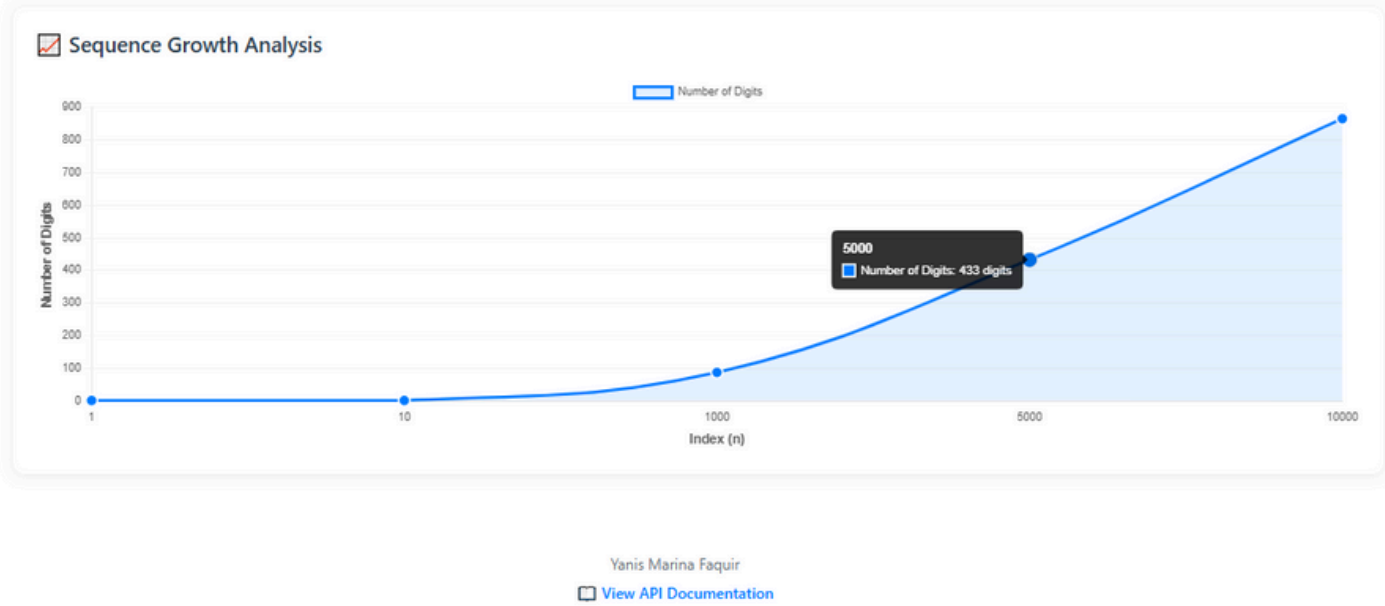


Figure 2 - Result



Yanis Marina Faquir

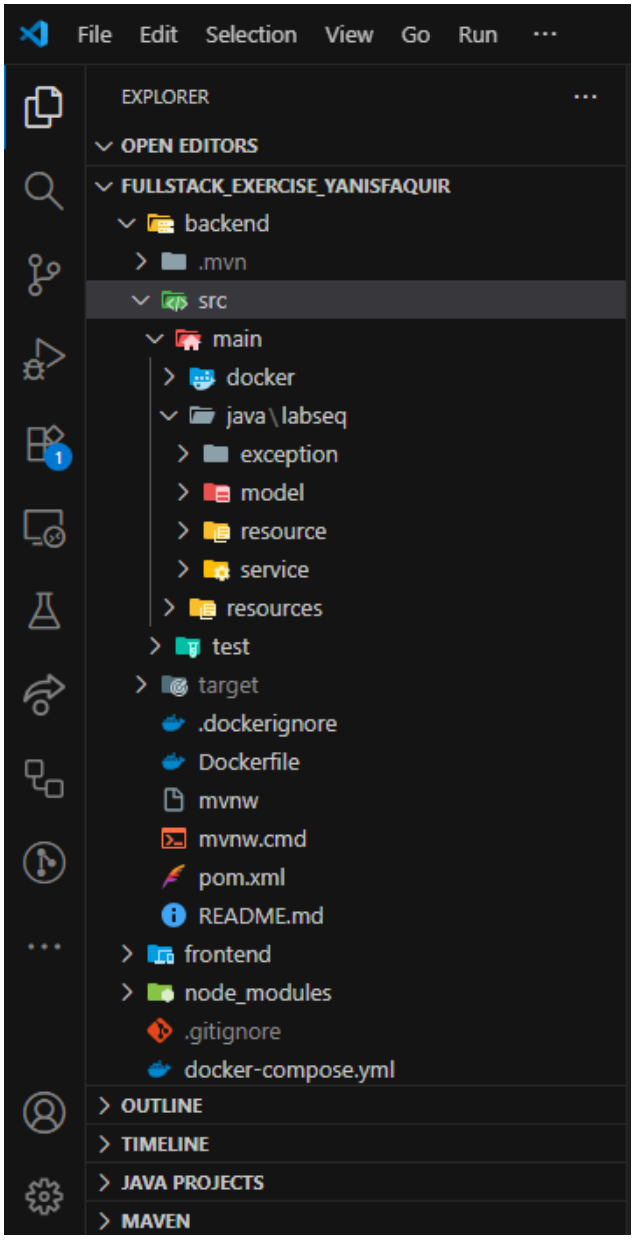
View API Documentation

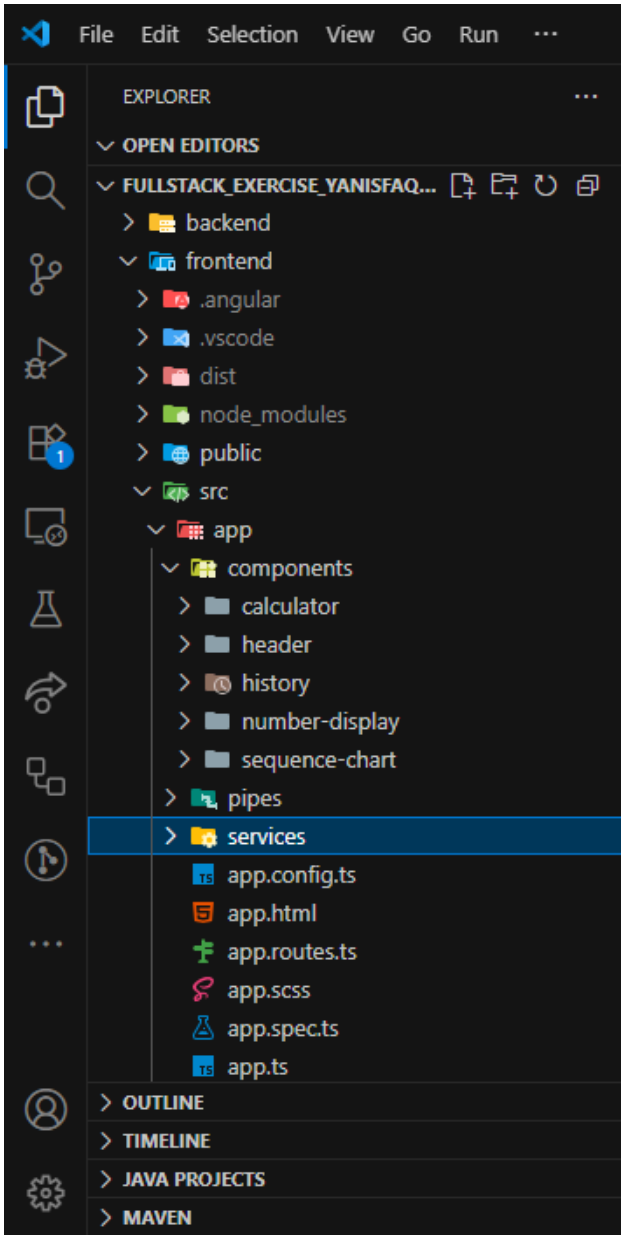
Figure 3 - Gráfico

STACK TECNOLÓGICO

Camada	Tecnologia	Justificativa
Backend	Quarkus + Java	Supersônico, baixo consumo de memória, hot reload
Frontend	Angular 20	Framework completo, TypeScript, RxJS
Cache	Quarkus Cache (Caffeine)	Cache distribuído, LRU eviction
Build	Maven + Angular CLI	
Testes	JUnit 5, Jasmine	Testes unitários (backend + frontend)
Documentação	OpenAPI (Swagger)	-

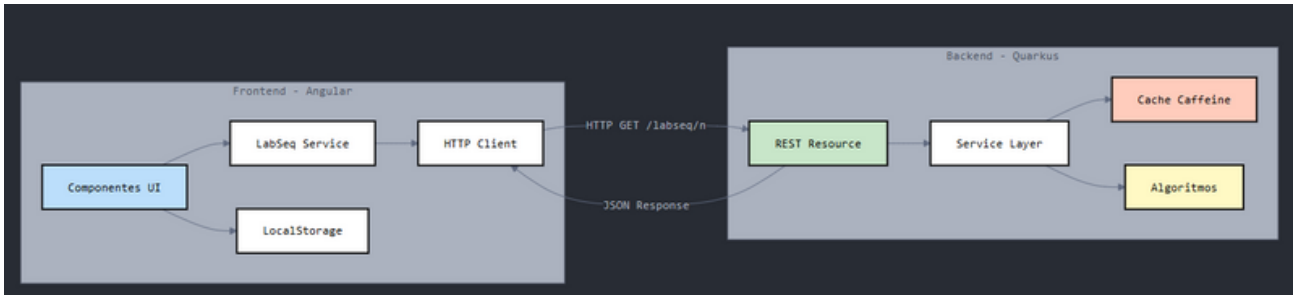
ESTRUTURA DO PROJETO & ARQUITETURA





ALTICE LABS

ESTRUTURA DO PROJETO & ARQUITETURA



SOLUÇÕES TÉCNICAS IMPLEMENTADAS

1. **Algoritmo Híbrido de Cálculo**

A solução implementa uma estratégia híbrida baseada em threshold (n = 1000). Para valores até 1000, utiliza o método recursivo com cache automático (@CacheResult), armazenando resultados intermediários e permitindo reutilização em requisições futuras com tempos inferiores a 1ms. Para valores acima de 1000, emprega o método iterativo com buffer circular de 4 posições, mantendo complexidade de espaço O(1) e evitando stack overflow. Esta abordagem combina as vantagens do cache para índices pequenos com a robustez necessária para índices grandes como n = 100.000.

```
@CacheResult(cacheName = "labseq-cache")
public BigInteger calculateRecursive(int n) {
 if (n == 0) return BigInteger.ZERO;
 if (n == 1) return BigInteger.ONE;
 if (n == 2) return BigInteger.ZERO;
 if (n == 3) return BigInteger.ONE;
 return calculateRecursive(n - 4).add(calculateRecursive(n - 3));
}
```

Vantagens:

- Cache automático de cálculos intermediários
- Beneficia requisições repetidas (cache hit < 1ms)
- Código elegante e fácil de entender

Para n > 1000: Método Iterativo

O método iterativo utiliza um buffer circular de apenas 4 posições para armazenar os valores base da sequência (0, 1, 0, 1). Como a fórmula l(n) = l(n-4) + l(n-3) depende apenas dos 4 valores anteriores, não é necessário manter todo o histórico de cálculos em memória. A cada iteração de 4 até n, calcula-se o valor atual somando as posições (i-4) % 4 e (i-3) % 4 do buffer, e o resultado sobrescreve a posição i % 4, reutilizando ciclicamente as mesmas 4 posições. Esta técnica garante complexidade de espaço constante O(1) independentemente do valor de n,

```
public BigInteger calculateIterative(int n) {
 // Buffer circular de 4 posições (O(1) memória)
 BigInteger[] last4 = new BigInteger[4];
 last4[0] = BigInteger.ZERO;
 last4[1] = BigInteger.ONE;
 last4[2] = BigInteger.ZERO;
 last4[3] = BigInteger.ONE;
 for (int i = 4; i <= n; i++) {
 BigInteger current = last4[(i - 4) % 4].add(last4[(i - 3) % 4]);
 last4[i % 4] = current;
 }
 return last4[n % 4];
}
```

Vantagens:

- Memória constante O(1): apenas 4 BigIntegers em memória
- Sem risco de stack overflow

2. Suporte a Números Gigantes

O cálculo de l(100.000) gera aproximadamente 21.000 dígitos, ultrapassando em muito a capacidade de tipos primitivos como long (limitado a ~19 dígitos). Para resolver isso, o backend utiliza BigInteger do Java, que suporta números de tamanho arbitrário limitado apenas pela memória disponível. No entanto, como JSON não suporta precisão arbitrária em números, a resposta da API serializa o valor como String ("value": "123...890"), preservando todos os dígitos sem perda de precisão. No frontend, implementei um sistema de formatação inteligente através do BigNumberPipe que adapta a visualização conforme o tamanho: números até 50 dígitos são exibidos completos com separadores de milhares, entre 50-100 dígitos mostra início e fim com reticências no meio, e acima de 100 dígitos utiliza notação científica. O usuário pode alternar entre três modos de visualização (compact, scientific, full) e copiar o valor completo para a área de transferência com um único clique.

Backend: BigInteger

```
@JsonProperty("value")
private String value; // String preserva precisão total
public LabSeqResponse(int n, BigInteger value, ...) {
 this.value = value.toString();
 this.digits = value.toString().length();
}
```

Frontend

```
@Pipe({ name: 'bigNumber' })
export class BigNumberPipe implements PipeTransform {
 transform(value: string, mode: 'compact' | 'scientific' | 'full') {
 const digits = value.length;
 if (digits <= 50) {
 return this.addThousandsSeparators(value); // "1 234 567 890"
 } else if (digits <= 100) {
 return `${value.substring(0, 40)}...${value.slice(-20)}`;
 } else {
 return this.toScientific(value); // "1.23456 × 10^20956"
 }
 }
```

Modos de visualização:

- Compact: Números até 50 dígitos completos
- Scientific: Notação científica para números enormes
- Full: Exibição completa para copiar/colar

3. Sistema de Cache em Múltiplas Camadas

Configuração (application.properties)

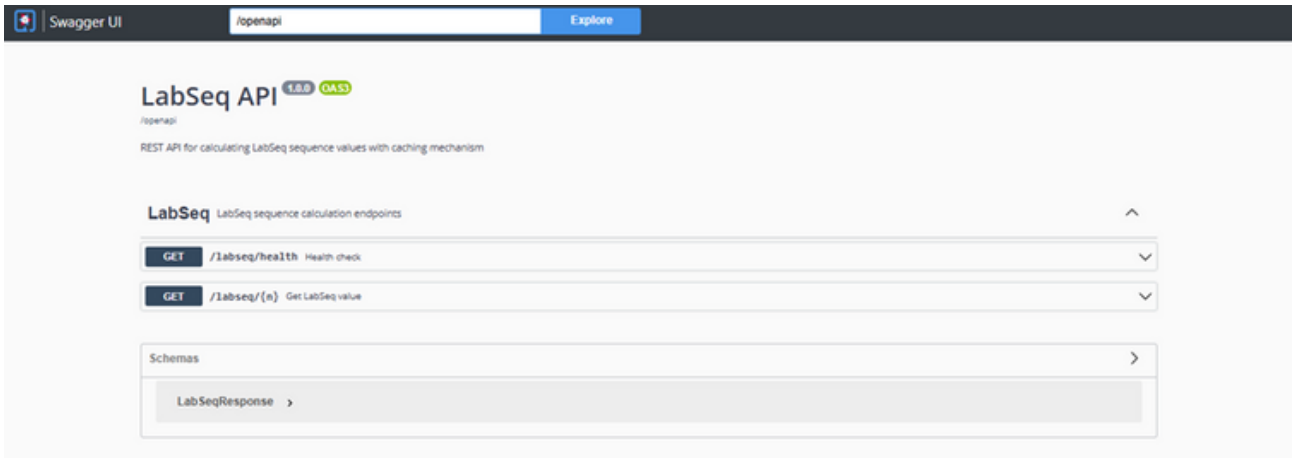
quarkus.cache.caffeine."labseq-cache".maximum-size=10000

quarkus.cache.caffeine."labseq-cache".expire-after-write=1H

Eviction Policy

- LRU (Least Recently Used): remove valores menos acessados
- Expiração após 1 hora (evita crescimento infinito)
- Máximo 10.000 entradas

4. API REST com OpenAPI/Swagger



Acesso: http://localhost:8080/swagger-ui

5.Interface Web Interativa (Angular)

Funcionalidades Implementadas

1. Calculadora Principal
   - Input com validação em tempo real
   - Feedback visual
   - Exibição de resultados com metadados
2. Histórico Persistente
   - Armazena últimos 10 cálculos
   - LocalStorage
   - Recarregar valores anteriores com um clique

3. Visualização de Números Grandes
   - Três modos de exibição (scientific, full)
   - Informação de tamanho (dígitos, bytes)
4. Gráfico de Crescimento
   - ng2-charts, chart.ts para visualização
   - Mostra crescimento exponencial da sequência
   - Eixo X: índice n, Eixo Y: número de dígitos
5. Health Check Visual
   - Verifica conectividade com backend
   - Indicador "Connected ✓" / "Disconnected ✗"
   - Executado automaticamente ao iniciar

6. Testes e Qualidade de Código

Backend: mvn test

- Casos base: Validação de l(0)=0, l(1)=1, l(2)=0, l(3)=1
- Sequência completa: Verificação dos primeiros 12 valores da sequência
- Performance: l(100.000) executado em menos de 10 segundos
- ·Funcionalidade do cache: Chamadas repetidas até 200x mais rápidas
- Consistência algorítmica: Métodos recursivo e iterativo retornam resultados idênticos
- Validação de entrada: Rejeição de índices negativos com erro HTTP 400
- API REST: Status HTTP corretos, estrutura JSON e headers CORS

Frontend: ng test --include='**/labseq.service.spec.ts'

·Criação do serviço: Instanciação correta do LabseqService

· Requisições básicas: Testes para n=0, n=10 validando estrutura da resposta

·Números grandes: Teste com n=5000 (1523 dígitos) e n=100000 (30000 dígitos)

·Tratamento de erros HTTP: Erros 500 (Internal Server Error) e 400 (Bad Request)

·Erros de rede: Simulação de falhas de conexão (ProgressEvent)

·Retry automático: Verifica tentativa de reconexão (falha → retry → falha)
·Retry com sucesso: Primeira tentativa falha, segunda bem-sucedida
·Health check: Status 'UP' com serviço disponível
·Health check failure: Tratamento de erro 503 (Service Unavailable)
·Edge cases: n=0 (caso base), n=100000, fromCache=true
·Validação de método HTTP: Confirma uso correto de GET requests
·Mocks HTTP: Todos os testes isolados com HttpClientTestingModule