Automatização de Geração de Relatórios em Sistemas Web[1]

Eduardo Nascimento Pessoa, Rodrigo de Almeida Vilar de Miranda

Departamento de Ciências Exatas (DCX) – Universidade Federal da Paraíba (UFPB)
Rua da Mangueira, s/n, Companhia de Tecidos Rio Tinto
CEP 58.297-000 – Rio Tinto – PB – Brazil

{eduardo.nascimento,rodrigovilar}@dcx.ufpb.br

Abstract. *Code generators were conceived to automate the development of repetitive and generic functionalities, CRUD operations being an example. There is a demand for reporting features, as they provide a streamlined view of information and greater assertiveness in decision making. In this work, a solution is presented for generating code to report functionalities on web systems using fine-grained code generators based on the Scaffolding technique.*

Resumo. *Os geradores de código foram idealizados com o intuito de automatizar o desenvolvimento das funcionalidades repetitivas e genéricas, por exemplo as operações de CRUD. Existe uma demanda por funcionalidades de relatórios, uma vez que, proporcionam uma visualização dinamizada de informações e uma maior assertividade na tomada de decisões. Neste trabalho, é apresentada uma solução para geração de código para funcionalidade de relatórios em sistemas web, utilizando geradores de código de granularidade fina baseados na técnica de Scaffolding.*

1. Introdução

No ciclo de vida de um software de sucesso, existem eventualmente demandas para alterações em função da evolução de requisitos, de tecnologias e do conhecimento dos *stakeholders* [Rajlich 2014], contudo, a habilidade de fazer evoluir um software de forma rápida, possibilitando o reuso é um dos maiores desafios para um engenheiro de software [Mens 2008]. Visando a redução desses esforços, foram idealizados geradores de código para a criação das funcionalidades mais comuns, denominadas CRUD (*create - read - update - delete*). Esses geradores produzem, em um curto espaço de tempo, código equivalente a horas de desenvolvimento de um profissional qualificado. Uma das técnicas disponíveis no mercado e utilizadas por estes geradores é a *scaffolding,* que consiste na criação automática de estruturas significativas de um projeto, tais como: banco de dados, mapeamento entre objetos e interfaces CRUD [Magno 2015].

No que se refere ao esforço no desenvolvimento de software, uma das tarefas mais repetitivas no desenvolvimento de aplicações web com fluxo intenso de dados é a implementação de operações de CRUD [Rodriguez-Echeverria 2016], e os geradores de código são utilizados para automatizar a implementação dessas operações, melhorando em termos de eficiência o desenvolvimento de software. Porém, um sistema não é

[1] "Trabalho de conclusão de curso, sob orientação do professor <nome do professor> submetido ao Curso de Licenciatura em Ciência da Computação do Centro de Ciências Aplicadas e Educação (CCAE) da Universidade Federal da Paraíba, como parte dos requisitos necessários para obtenção do grau de LICENCIADO EM CIÊNCIA DA COMPUTAÇÃO."

composto apenas por CRUDs, existem diversas outras funcionalidades vitais que são necessárias para atender o domínio da aplicação.

Mediante dessa perspectiva, destacam-se os relatórios como uma das funcionalidades não contempladas pelos geradores de código. Diante disto, é válido destacar que relatórios computacionais são conjuntos de dados, com finalidades específicas, que podem ser coletados com determinada periodicidade e exibidos instantaneamente ou armazenados para exposições posteriores [IBM 2013 apud Eliziario, 2014]. Vistos em empresas como pontos cruciais na tomada de decisões, os relatórios levam a uma maior assertividade, tendo em vista que, o usuário consegue visualizar de forma dinamizada e clara as informações acerca dos dados produzidos pela operação de cada setor da empresa. Diante disto, pode-se afirmar que:

> A falta de informação em tempo hábil pode causar um problema à empresa, uma vez que as tomadas de decisão serão empíricas ao invés de embasadas em dados reais. Torna-se assim, imprescindível a utilização de tecnologias capazes de resgatar os dados e transformá-los em informações [Zimmermann 2006, p.12].

Diante do exposto, a geração automatizada de relatórios adquire importância em muitas situações. Entretanto, as limitações dos geradores dificultam a automatização desse trabalho, tendo em vista que, os geradores de código em sua maioria são direcionados a gerar funcionalidades genéricas e repetitivas, não atendendo a demandas específicas de sistemas.

Atualmente, no mercado é possível encontrar diversas alternativas e abordagens para geração de código, no entanto, este trabalho se atém aos geradores de código de granularidade fina propostos por Vilar *et al.* (2015), pois permitem o desacoplamento dos *templates* de geração de código e facilitam a sua evolução.

Neste sentido, o presente trabalho objetiva descrever o processo de implementação de uma solução para a geração automatizada de código para a funcionalidade de relatórios em sistemas Web, utilizando gerador de código de granularidade fina. O restante do conteúdo está dividido nas seguintes Seções: 2. Descrição do processo, onde se detalha o processo de desenvolvimento deste trabalho; 3. Descrição das tecnologias utilizadas durante o processo de desenvolvimento; e 4. Solução, onde são descritos os artefatos e resultados do trabalho.

2. Descrição do Processo

O processo de desenvolvimento da solução foi dividido nas seguintes etapas: *i)* escolha do gerador de código baseada em pesquisas por artigos acadêmicos e ferramentas do mercado; *ii)* elaboração de um cenário para desenvolvimento da solução; *iii)* implementação manual do código de referência para geração de relatórios, inicialmente de forma não automatizada; *iv)* criação de *templates* na ferramenta escolhida para a geração de código para a funcionalidade de relatórios; *v)* validação do código gerado automaticamente.

2.1. Escolha do Gerador de Código

O gerador de código escolhido para este trabalho foi o *Potter*[2], devido a sua capacidade de fragmentar os *templates* de geração de código em grãos finos. Um *template* é uma parte customizável do gerador de código que modela genericamente o código fonte de uma série de funcionalidades do sistema a ser gerado. Os *templates* são arquivos de texto que utilizam como modelo códigos-fonte de projetos bem sucedidos [Magno 2015], nesse contexto, podem ser compreendidos como códigos bem implementados, refatorados, testados e validados por arquitetos de software. Com a fragmentação, os *templates* passam a ser desacoplados, permitindo assim um maior reuso e facilidade na evolução do desenvolvedor visando sua melhoria. E este é exatamente o cenário almejado: melhorar os *templates* de CRUD a fim de que também possam gerar relatórios.

O *Potter* é um projeto código aberto e surgiu a partir da evolução da abordagem de granularidade fina proposta por Vilar (2017)[3]. Diferentemente de outros geradores de sua classe, baseados em *scaffolding*, que efetuam a geração da base do projeto e cabe ao desenvolvedor fazer evoluir o código após a geração, no Potter dá-se a integração o desenvolvedor na melhoria dos *templates* existentes, levando a um melhor refinamento dos artefatos finais. Com essa integração, o desenvolvedor efetua melhorias nos templates do *Potter,* possibilitando que outros desenvolvedores e usuários utilizem estes templates na geração de outras aplicações.

2.2. Elaboração de um cenário para o desenvolvimento da solução

Para o desenvolvimento da solução proposta, fez-se necessário selecionar um cenário real para aplicação. Nessa etapa, buscou-se identificar a demanda e quais os possíveis cenários seriam capazes de atendê-la.

Foi escolhido o cenário de Vendas, que possui três entidades, sendo elas: Venda, ItemVenda e Produto. A entidade Venda é composta pela data da venda, um cliente e uma lista de ItensVenda que é resultado do relacionamento de um para muitos com a ItemVenda. A ItemVenda é composta pela quantidade de produtos e um produto que é fruto do relacionamento com a entidade Produto. Já um Produto é composto pelo valor, o nome e a categoria. Tal cenário pode ser melhor visualizado na Figura 1.

Nesse cenário, vários relatórios podem ser gerados, por exemplo: quantidade e valor das vendas por cliente, Quantidade de itens vendidos por Produto, Quantidade de itens vendidos por Categoria, Valor médio de venda, Histórico de vendas por Produto, Categoria e Cliente, etc. Ao final deste trabalho, espera-se que parte desses relatórios de exemplo sejam geradas automaticamente pelo *Potter*.

---

[2] Disponível em: <https://github.com/potterjs/pot> Acesso em: 05 set. 2021.
[3] Disponível em: <http://dspace.sti.ufcg.edu.br:8080/jspui/handle/riufcg/583> Acesso em: 05 set. 2021.

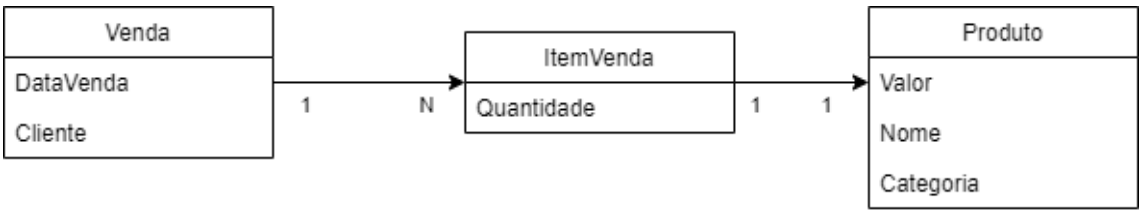

Figura 1. Cenário de Vendas
Fonte: O autor

2.3. Implementação manual de Código para geração de relatórios

Após a escolha do cenário, foi iniciada a etapa de implementação manual do código de referência para geração de relatórios. Para tal, foi gerado um projeto utilizando o *Potter*, contemplando apenas os CRUDs do cenário escolhido e a partir deste, foi efetuada a implementação de forma manual dos relatórios, com o objetivo de identificar as diferenças para que assim fosse possível projetar os *templates* necessários para efetuar a geração dos relatórios automaticamente. Portanto, essa etapa foi necessária para mensurar o esforço para a implementação das funcionalidades, possibilitando comparar a eficácia da sua criação na geração do projeto, podendo assim, demonstrar os benefícios obtidos pela solução.

A princípio, na etapa de implementação, houve a necessidade de buscar por ferramentas que possibilitam a obtenção de dados para a montagem dos relatórios. Após pesquisar por ferramentas para criação de *API*s analíticas, foi encontrado o *Cube.js*[4], que tem o propósito de fornecer dados e atuar como um *back-end* analítico, sendo o mesmo melhor descrito na Seção 3 deste artigo.

Sendo o *Potter* um projeto que vem sendo aperfeiçoado paulatinamente, já é possível encontrar *templates* desenvolvidos para *Spring Boot* juntamente com *Angular*, destinados a geração de CRUDs simples. Desta forma, este trabalho teve foco na implementação da geração de uma API analítica para o fornecimento de dados e nas funcionalidades para geração de relatórios em um *front-end* Angular.

Como citado anteriormente, foi utilizado o *Potter* para a geração de um projeto inicial, baseado no cenário escolhido na etapa anterior e a partir deste foi efetuada a evolução. Essa evolução foi iniciada com a criação de um projeto *Cube.js*, e para tal fez-se o uso do comando disponibilizado em sua documentação para criá-lo de forma automática, gerando assim todos os arquivos iniciais necessários. Ainda no *Cube*, foram efetuadas as configurações para viabilizar a conexão com a base de dados do projeto, sendo assim possível sua execução. Em segundo momento, no projeto *Angular* gerado pelo *Potter*, foi instalada a biblioteca *CubeClient*, o que possibilitou a comunicação do *client* com o *server*.

Para a montagem dos relatórios, foi utilizada a biblioteca *Chart.js*[5], que facilita sua inserção e a configuração, além de disponibilizar uma vasta gama de gráficos de

[4] Disponível em: <https://cube.dev/> Acesso em: 05 set. 2021
[5] Disponível em: <https://www.chartjs.org/> Acesso em: 05 set. 2021

fácil implementação. Outro ponto importante a ressaltar na escolha da ferramenta para essa etapa, é o suporte nativo dado a ela pelo *Cube.js*, uma vez que, este utiliza a *Chart.js* para efetuar a geração dos gráficos que são exibidos em seu *playground.* A biblioteca *Chart.js* está descrita na Seção 3 deste artigo.

Visando a montagem de relatórios referentes ao cenário escolhido na Seção 2.2, decidiu-se por utilizar relatórios em formato de tabelas, para propiciar uma melhor visualização dos dados de cada produto. Além disso, foram implementados também, relatórios com uso de gráficos, objetivando a exibição da quantidade de produtos separados pela categoria.

2.4. Criação de *templates* no *Potter* para fornecer código para geração dos relatórios.

Para a criação dos *templates*, fez-se necessário a criação prévia de um código de referência. Uma vez que este foi criado, precisou ser seccionado em partes menores para atender os princípios da granularidade fina. A decisão de como realizar a quebra do código foi feita com o objetivo de identificar em quais partes do código existe a possibilidade de adaptação para reuso. Um exemplo disso seria em um formulário, onde os campos compartilham estruturas semelhantes, como *labels*, *placeholders* e etc, permitindo que com uma leve mudança nos metadados ocorra uma flexibilização no estado do *template* responsável por estes campos. No ecossistema do *Potter,* cada parte do código é individual e se comunica com as demais partes através de portas. As portas são em suma as chamadas que um *template* faz para para outros *templates*. Um exemplo prático disso seria em um sistema de formulários, onde o *template* principal realiza a abertura de diversas portas, uma para cada campo que deve ser criado no formulário. As portas podem ter comportamentos condicionais, definidos com uma estrutura chamada de seletores, que funcionam como chamadas polimórficas a uma hierarquia de *templates*, baseadas na estrutura dos metadados.

2.5. Validação do código gerado pelo *Potter*.

Uma vez que o código de referência foi devidamente segmentado e inserido na plataforma, fez-se necessário realizar a validação, que consistiu em gerar o código utilizando o *Potter*. Para a confirmação de que o código está devidamente desacoplado do domínio da aplicação, realiza-se a geração utilizando dois metadados distintos, um contendo o domínio do cenário escolhido, e outro contendo um cenário de teste.

Ao final desse processo, os artefatos gerados precisam funcionar corretamente e de forma individual. Caso isso não aconteça, faz-se necessário uma melhor reestruturação do código de referência ou dos *templates*, visto que esses não estariam devidamente desacoplados. Os artefatos são testados de acordo com suas tecnologias, variando de iniciar um serviço até o uso de testes automatizados. Para este trabalho, os artefatos serão testados utilizando os seguintes passos: execução do serviço *back-end*, execução do serviço de *back-end* analítico *Cube.js* e execução do serviço *front-end*, de ambas as aplicações geradas. Ao executar os serviços de back-end das aplicações geradas, estas devem funcionar corretamente sem precisar de ajustes diretos no código. Os serviços *Cube.js* devem ser executados de forma a servir as *APIs* para consulta,

sendo imprescindível o sucesso da conexão com o banco de dados das aplicações. Já os serviços de *front-end* , devem ser executados e entregar ao usuário as funcionalidades de relatórios, servindo gráficos com dados reais obtidos através das requisições feitas ao serviço *Cube.js*.

3. Descrição das Tecnologias

Para o desenvolvimento da solução, foram utilizadas diversas ferramentas, descritas ao longo desta Seção, sendo elas, *Potter*, *Cube.js* e *Chart.js*. A arquitetura da solução pode ser visualizada na Figura 2 que demonstra como as tecnologias se interconectam nos componentes da Solução proposta (descrita na Seção 4).

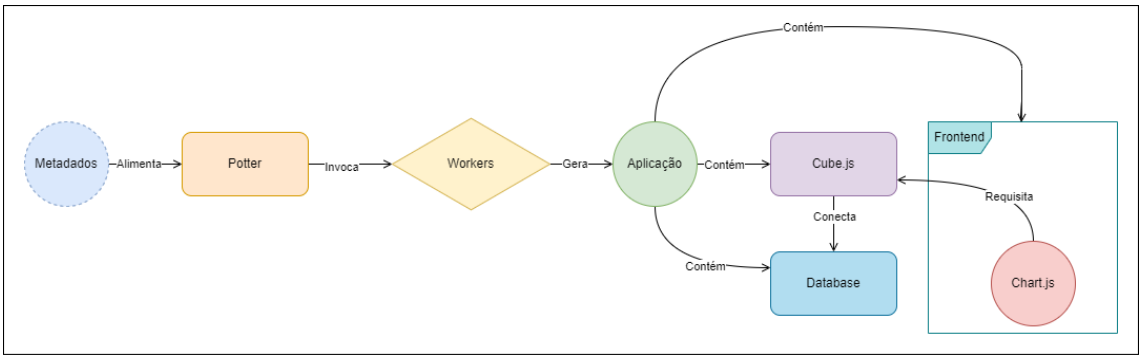

Figura 2. Arquitetura da solução.
Fonte: O autor

3.1. Potter

O *Potter* é um gerador de código baseado na técnica de *Scaffolding* que se diferencia pelo fato de utilizar os padrões para granularidade fina. Esta ferramenta faz uso de *templates* e permite que o desenvolvedor os crie ou os modifique para que seja possível a geração de novas funcionalidades no *software*, possibilitando a integração do desenvolvedor na melhoria dos *templates*, além disso, viabilizando a componentização de partes do sistema devido a sua granularidade fina.

Para gerar a aplicação, o gerador é alimentado com um metadado, para que assim o mesmo invoque os *workers* necessários para a geração da aplicação. Os *workers*, são estruturas que carregam os modelos e regras de geração de código, por exemplo quais *templates* deverão ser utilizados.

Os *Workers* são subdivididos em três tipos, sendo eles, *Designs*, *Templates* e *Shapes*. Os *Designs* contêm as regras para conexão dos templates e utilizam conceitos como: portas e seletores e possibilitam o uso de comandos como: *mkdir*, *zip* e *output*. Os *Templates* são fragmentos textuais de código e utilizam a sintaxe mustache (“{{ }}”) para acessar os metadados ou delegar a geração para outro *Template* através das portas. Os *Shapes* são processadores de metadados, que quando utilizados possibilitam a simplificação dos metadados em tempo de execução a fim de facilitar a geração da aplicação.

3.2. Cube.js

O *Cube.js* é uma plataforma de API Analítica utilizada para criação de ferramentas para Business Intelligence (*BI*). Uma aplicação em *Cube.js*, atua como um serviço que media a comunicação da aplicação *client* com os dados do banco de dados. Utilizando o *CubeClient*, é possível efetuar requisições, de modo a consumir os dados disponibilizados pela API do *Cube.js*. Os dados fornecidos pela *API*, são recebidos em formato de *ResultSet<>*, e podem ter seus valores extraídos e utilizados na montagem dos relatórios no *front-end* da própria aplicação gerada pelo *Potter*. Na Figura 4 é possível visualizar melhor a arquitetura do *Cube.js*.

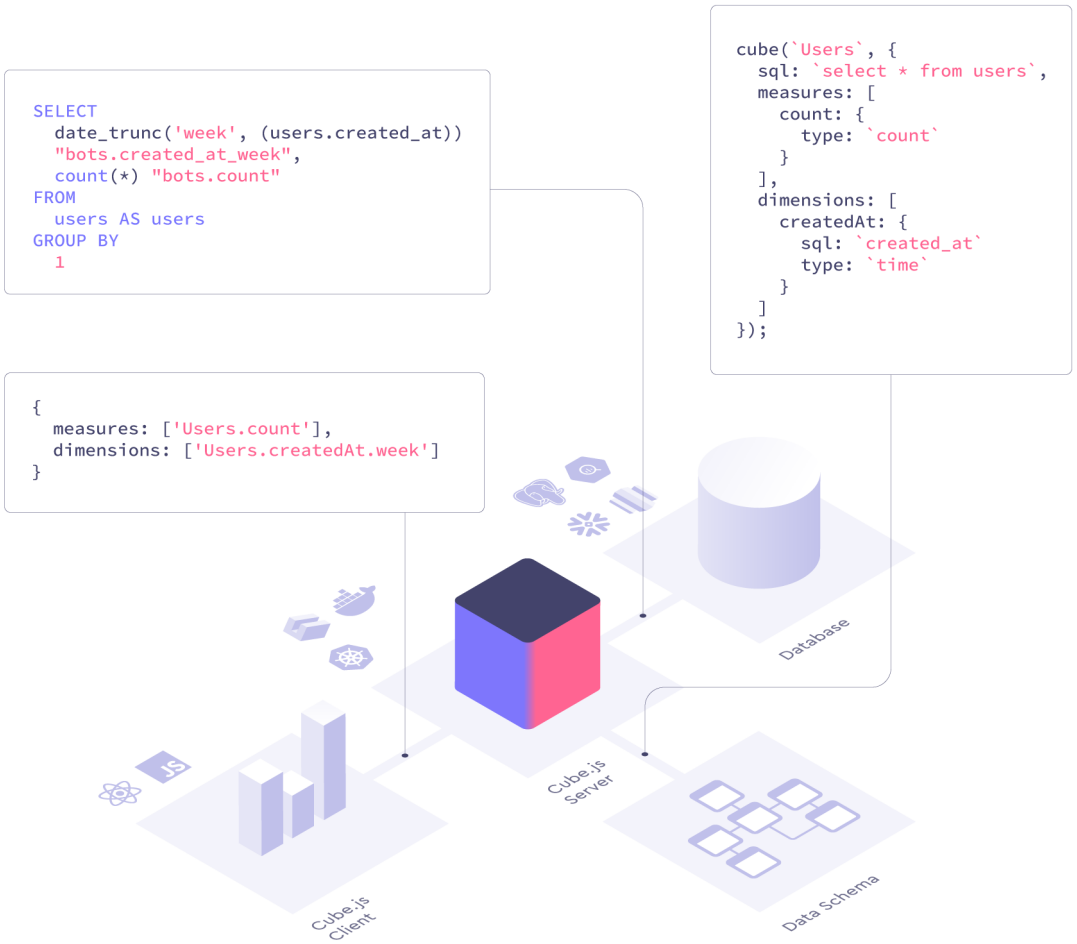

Figura 3. Arquitetura do *Cube.js*
Fonte: Documentação oficial do *Cube.js*[6]

Com base na figura acima, pode-se destacar que o *Cube.js client* efetua uma requisição enviando como parâmetro uma *cube query*, que é interpretada pelo *Cube.js server*. Por conseguinte, esse *server* faz uso dos *data schemas*, efetua a tradução da lógica de negócio para SQL, bem como a consulta no banco de dados e processa o envio

[6] Disponível em: <https://cube.dev/docs/introduction> Acesso em: 12 nov. 2021

do resultado para o *client*. Ademais, é válido ressaltar que a *Cube Query* diz respeito a um objeto *Javascript* simples, que descreve uma consulta analítica, sendo composta principalmente por *measures*, que representam um conjunto de pontos de dados no cubo e *dimensions*, que representam as propriedades de um único dado no cubo. No que diz respeito a *Data Schemas*, estes são arquivos textuais criados no projeto com a finalidade de traduzir a lógica de negócios em código SQL, compostos por cubos, que representam conjuntos de dados em *Cube.j*s e são geralmente declarados separadamente de modo que cada arquivo detenha apenas um cubo.

A utilização do *Cube.js* soluciona problemas que ocorrem com frequência no desenvolvimento de aplicações para *BI*, como problemas no desempenho e na infraestrutura agilizando assim o processo de desenvolvimento de soluções para este fim e reduzindo o tempo e esforço necessários para criação desses sistemas.

3.3. Chart.js

O *Chart.js* é uma biblioteca de visualização de dados *open source*. Sua utilização possibilita a visualização de dados pelo cliente de um sistema *web*. Ela facilita a implementação de gráficos e simplifica o processo de obtenção de gráficos, tendo em vista que a mesma oferece uma gama de diferentes tipos com a vantagem de ter fácil aplicação e não necessitar de configurações complexas ou da produção do gráfico do zero. É necessário apenas a instalação da biblioteca no projeto *Angular*, instalação essa que pode ser efetuada utilizando os gerenciadores de pacotes nativos da linguagem, a criação de um <*canvas*> e a inserção dos dados para popular o gráfico. Na Figura 4 é possível ver alguns exemplos de gráficos feitos com *Chart.js.*

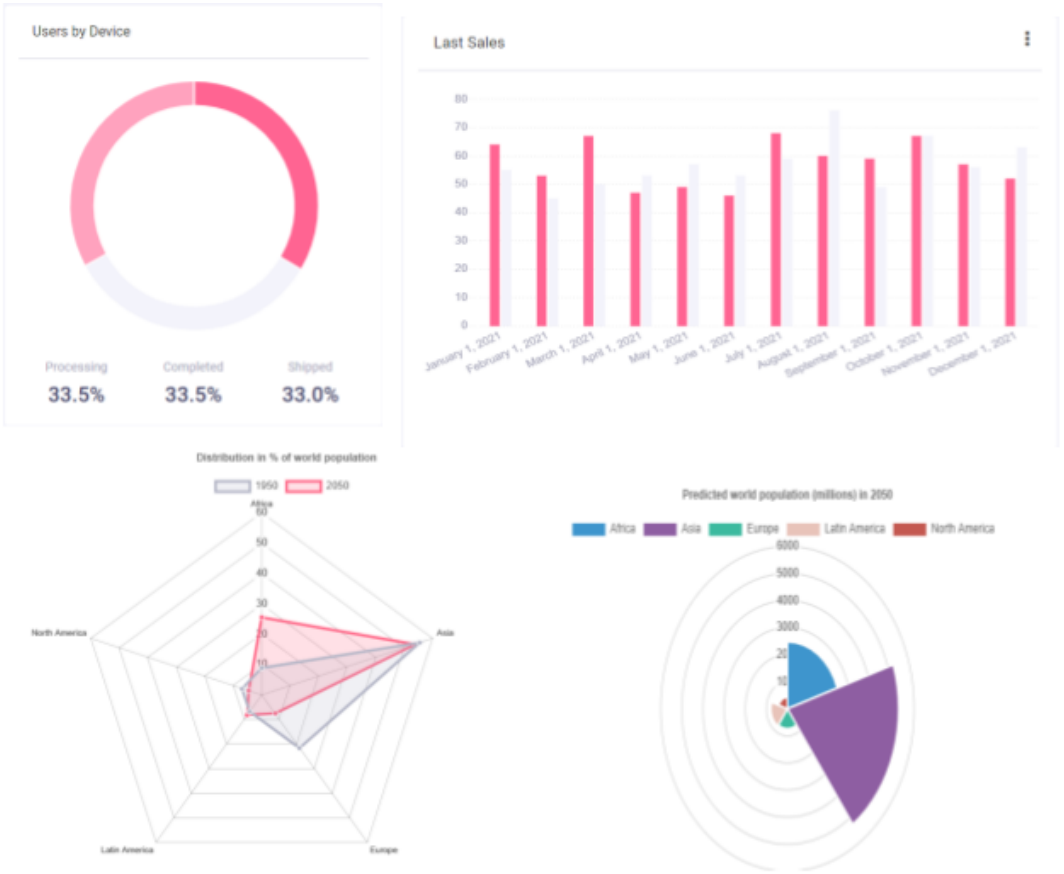

Figura 4. Exemplos de gráficos feitos com *Chart.js*
Fonte: Compilação do autor[7]

4. Solução

A solução implementada foi arquitetada como apresentada na Figura 2, de modo a se obter a solução proposta ao gerar um sistema *web*. O processo de implementação da solução será detalhado nesta Seção, dividindo-se no detalhamento da arquitetura da solução, templates criados ou modificados para gerar o código de relatórios e resultados obtidos.

4.1. Detalhamento da arquitetura da solução

Assim como é possível visualizar na Figura 2, o processo de desenvolvimento da solução se iniciou com a criação de um metadado[8], que é estruturado como um arquivo do tipo *JavaScript Object Notation* (*JSON)* e representa o domínio da aplicação, baseada no cenário escolhido na Seção 2.2.

---

[7] Compilação feita a partir de imagens do site da documentação do *Cube.js* e do repositório de Yong Beom Kim

[8] Metadado criado para validação encontra-se disponível em: <https://gist.github.com/eduardopessoa89/1c64cb585886e9c1a56e25fba96b00b9>.

O gerador foi alimentado com os metadados e utiliza os *workers* a fim de gerar a aplicação final. Essa aplicação foi composta por um *front-end*, um projeto *cube.js* e um *back-end*.

O *front-end* contém todos os componentes necessários para efetuar as operações de *CRUD* e possui em sua composição componentes implementados utilizando o *framework Chart.js* que possibilita a geração de gráficos de forma simplificada. Tais componentes têm o intuito de viabilizar a criação de gráficos de pizza e barras, possibilitando assim uma visualização dinamizada dos dados. Para que os relatórios gráficos sejam gerados, existe uma comunicação entre o projeto *Cube.js* e o *front-end* da aplicação, que dá-se por meio de requisições, para que assim, o *front-end* possa consumir os dados providos pelo *Cube.js*.

O projeto *Cube.js,* por sua vez, contém os *schemas* necessários para contemplar todas as entidades utilizadas no sistema, como também está configurado para conectar-se à base de dados da aplicação. O projeto *back-end* foi configurado para atender às necessidade do sistema e efetuar as operações CRUD estando também conectado a uma base de dados.

O *back-end* da aplicação foi gerado com todas as funcionalidades para atender às necessidades das funcionalidades CRUDs. Além disso, também possui autenticação e comunicação com o banco de dados.

4.2. *Templates* criados ou modificados para gerar o código de relatórios

Visando seguir o fluxo do processo de desenvolvimento da solução descrito na Seção 2, após obter o código de referência[9], efetuou-se a adaptação e a quebra deste código em *templates* a serem utilizados no gerador. Tal quebra foi feita baseada nos preceitos da granularidade fina, visando atingir um menor acoplamento entre os *templates*, e na premissa de isolar componentes que mudam de forma conjunta, buscando viabilizar o reuso destes componentes.

Inicialmente foi criado um *worker* chamado *Design:CubeJSProject:1,* que contém as regras para a geração de projetos *Cube.js*. Nele foi declarado o *Worker Design:CubeJS:SchemaFile:1*, que é responsável pelas regras da geração dos arquivos de *schemas* que são incubidos da responsabilidade de gerar as *queries* SQL, como também dos demais arquivos de um projeto *Cube.js*.

A Figura 5, representa o fluxo de geração do código dos projetos *Cube.js*. A mesma está dividida, de modo que, à esquerda estão os arquivos do projeto e à direita estão os *workers* responsáveis pela geração de cada um deles.

---

[9] Disponível em: <https://github.com/eduardopessoa89/bi-tcc/compare/1.0.0...master> Acesso em: 12 dez. 2021

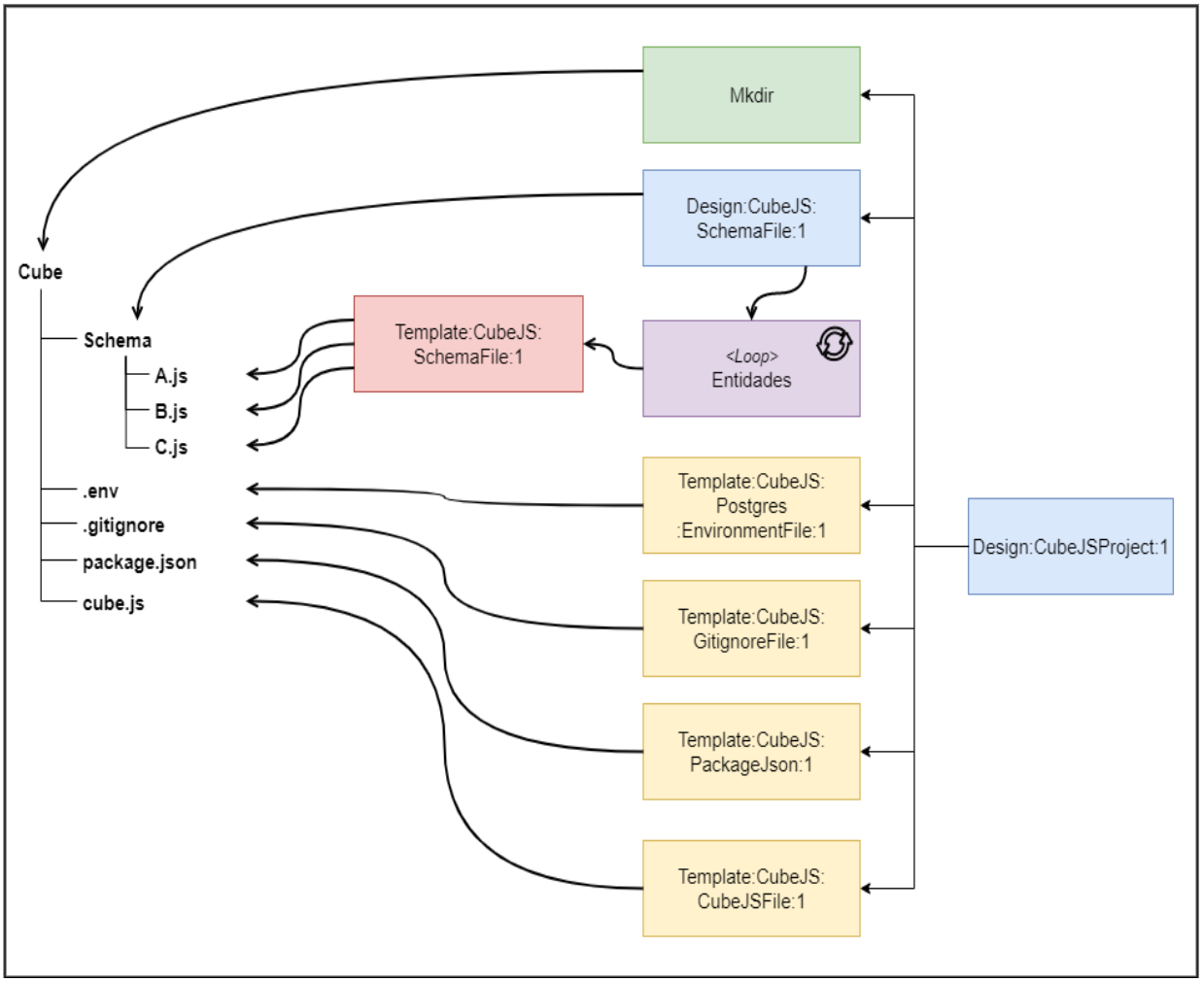

Figura 5. Diagrama da geração do projeto *Cube.js*
Fonte: O autor.

Destacado na cor vermelha na Figura 05, está o *Template:CubeJS:SchemaFile:1,* este *worker* contém a estrutura base para a geração dos arquivos *schema*, seu conteúdo pode ser visualizado no código a seguir. Nas linhas 10 e 14 do Código 1, estão as portas *measures-model* e *dimensions-model* que são responsáveis respectivamente pela criação das *measures* e *dimensions.* Ambas recebem como contexto a lista de campos da entidade em questão, denominada *fields* e provida pelo metadado.

```
1.  cube(`{{capitalize name.singular}}`, {
2.      sql: `SELECT * FROM public.{{name.singular}}`,
```

```
3.
4.      measures: {
5.          count: {
6.            type: `count`,
7.            drillMembers: []
8.          },
9.
10.         {{measures-model fields}}
11.     },
12.
13.     dimensions: {
14.         {{dimensions-model fields}}
15.     },
16. });
```

Código 1. Conteúdo do Worker Template:CubeJS:SchemaFile:1
Fonte: O autor

Ao utilizar esse template na geração de código, é gerado um arquivo de estrutura semelhante à mostrada no Código 2. É possível observar que, todas as notações que utilizam a sintaxe mustache foram substituídas no ato da geração pelos dados cabíveis.

```
1.  cube(`Disciplina`, {
2.    sql: `SELECT * FROM public.Disciplina`,
3.
4.    measures: {
5.      count: {
6.        type: `count`,
7.        drillMembers: [],
8.      },
9.
10.     nomeCount: {
11.       sql: `nome`,
12.       type: `count`,
13.     }
14.   },
15.
16.   dimensions: {
17.     nome: {
18.       sql: `nome`,
19.       type: `string`,
20.     },
21.     creditos: {
22.       sql: `creditos`,
23.       type: `string`,
24.     },
25.     descricao: {
26.       sql: `descricao`,
27.       type: `string`,
```

```
28.     },
29.   },
30. });
```

Código 2. Exemplo de Conteúdo do arquivo gerado com o worker Template:CubeJS:SchemaFile:1
Fonte: O autor

Além das modificações feitas para possibilitar a geração do projeto *Cube.js*, foram necessárias efetuar modificações nos *workers* responsáveis pela geração do *front-end*.

Para facilitar a criação dos componentes de relatórios, foram criados componentes genéricos para cada tipo de relatório a ser gerado, estes componentes contêm o código necessário para a criação de gráficos e podem ser utilizados livremente em todo o projeto conforme a necessidade. Os gráficos escolhidos para serem implementados foram os gráficos de pizza e de barras, tendo em vista que estes propiciam uma melhor visualização dos dados.

Para automatizar a geração destes componentes, *workers* foram criados e modificados. Na Figura 6, é possível visualizar o fluxo de geração dos componentes. À esquerda estão os arquivos referentes aos componentes visuais utilizados no projeto e à direita os respectivos *workers* responsáveis por sua geração.

Destacados em vermelho na Figura 6, estão os *workers* do tipo *design* que foram criados para conduzir a geração dos componentes genéricos de gráficos. Em amarelo estão os *templates* que dispõe dos modelos a serem utilizados na geração.

Um projeto *front-end*, em geral, é gerado com o intuito de atender às demandas de CRUDs, no entanto, com a adição da funcionalidade de relatórios houve a necessidade da criação de *workers* para a geração de componentes referentes a esta funcionalidade, na Figura 7 está representado o fluxo da geração destes componentes.

No *worker Design:Angular2+:EntityComponents:3,* destacado na cor vermelha na Figura 7, foram adicionadas as regras para geração do códigos dos componentes de relatórios. Com esta adição, os projetos que utilizarem deste *worker* contarão não somente com os componentes de CRUD, mas também com os componentes referentes a funcionalidade de geração de relatórios.

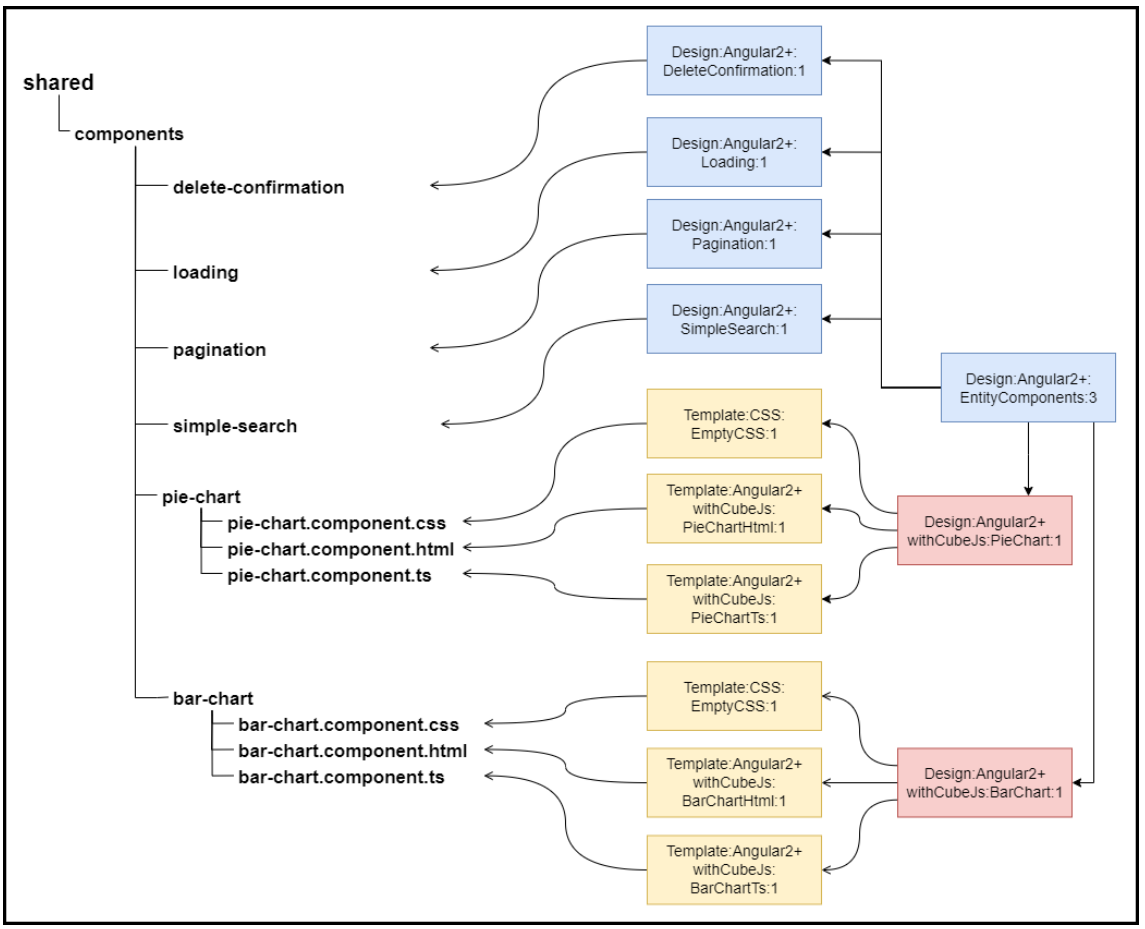

Figura 6. Diagrama da geração dos componentes gráficos
Fonte: O autor.

Para que os componentes de relatórios fossem gerados, foram criados *workers* do tipo *Template*, contendo os modelos a serem seguidos na geração dos códigos. Na Figura 7, destacados em amarelo, estão os *workers* criados para a geração destes componentes, são eles: *Template:Angular2+WithCubejs:EntityReportComponentHTML:1* e *Template:Angular2+WithCubejs:EntityReportComponentTs:1*, os conteúdos de ambos podem ser respectivamente encontrados no Código 2 e Código 3.

```
1.  <div class="row">
2.    <div class="col-6">
3.      <app-pie-chart id="pie" [cubeQuery]="cubeQuery" [titleText]="textPieTitle"
    ></app-pie-chart>
4.    </div>
5.    <div class="col-6">
6.      <app-bar-chart id="pie" [cubeQuery]="cubeQuery" [titleText]="textBarTitle"
    ></app-bar-chart>
7.    </div>
8.  </div>
```

Código 3. Conteúdo do Worker Template:Angular2+WithCubejs:EntityReportComponentHTML:1
Fonte: O autor

O código em *Hypertext Marking Language* (HTML) acima está contido em *Template:Angular2+WithCubejs:EntityReportComponentHTML:1*, e é usado como modelo para a geração do componente visual da funcionalidade de relatórios.

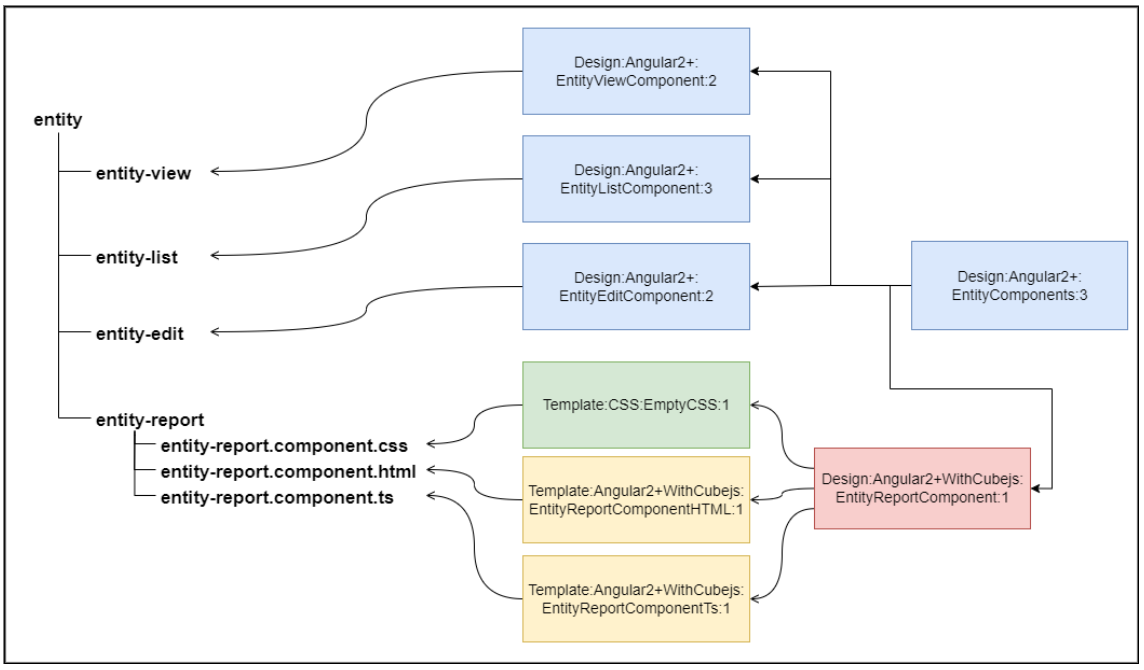

Figura 7. Diagrama de geração dos componentes de relatórios por entidade.
Fonte: O autor.

No Código 3, é possível encontrar o conteúdo do *worker* *Template:Angular2+WithCubejs:EntityReportComponentTs:1,* que tem como objetivo ser usado como modelo para a geração de arquivos em *TypeScript.* Estes arquivos são responsáveis pela lógica dos componentes.

```
1.  import { Component, OnInit } from '@angular/core';
2.  import { ChartDataSets } from 'chart.js';
3.  @Component({
4.    selector: 'app-{{lower name.singular}}-report',
5.    templateUrl: './{{lower name.singular}}-report.component.html',
6.    styleUrls: ['./{{lower name.singular}}-report.component.css']
7.  })
8.  export class {{capitalize name.singular}}ReportComponent implements OnInit {
9.    cubeQuery = {
10.     "measures": [
11.       "{{capitalize name.singular}}.count"
12.     ],
13.     "dimensions": [
```

```
14.       "{{capitalize name.singular}}.nome"
15.     ]
16.   }
17.   textPieTitle = "Gráfico de Pizza de {{capitalize name.plural}} por Nome (%)";
18.   textBarTitle = "Gráfico de Barras de {{capitalize name.plural}} por Nome";
19.   constructor() {}
20.   ngOnInit() {}
21. }
```

Código 4. Conteúdo do Worker
Template:Angular2+WithCubejs:EntityReportComponentTs:1
Fonte: O autor

Além dos *Workers* mostrados anteriormente, houveram outros que sofreram modificações pontuais às quais não cabem a exibição de código, como por exemplo, a adição de um botão na listagem para permitir o acesso aos *dashboards*, a adição de dependências no *package.json* e a adição da conexão com *cube.js* no *front-end*.

4.3. Resultados

Para fins de avaliação da solução, foi efetuada a geração de um projeto fim a fim. De fato, uma aplicação foi gerada utilizando metadados diferentes do cenário inicialmente projetado para, dessa forma, demonstrar a eficácia da solução implementada em um cenário de controle.

O cenário escolhido foi o de Turmas, que possui três entidades, são elas: Turma, Disciplina e Avaliação. A Turma possui nome, código e uma lista de disciplinas que é resultante do relacionamento um para muitos com Disciplina. A Disciplina, por sua vez, é composta pelo nome, créditos, descrição e uma lista de avaliações advindas de um relacionamento um para muitos com a entidade Avaliação. Já a Avaliação é formada por nome, peso e questões. Este cenário é melhor representado na Figura 8.

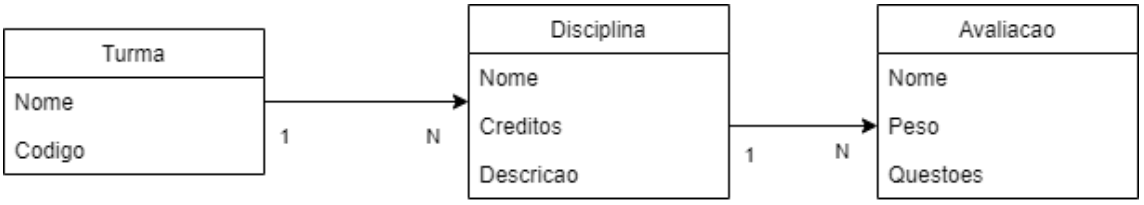

Figura 8. Cenário de controle
Fonte: O autor.

Como resultado da geração, foi obtida a aplicação composta por todas as funcionalidades planejadas. A tela de listagem das entidades (Figura 9), possui um botão que ao ser clicado dá acesso a tela de relatórios (Figura 10).

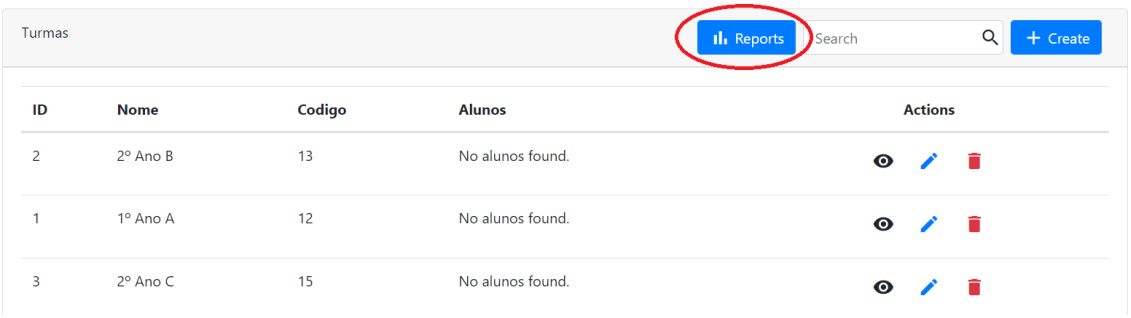

Figura 9. Listagem de Turmas
Fonte: O autor.

Foram gerados dois relatórios, sendo um em barras e outro em pizza, ambos baseados na quantidade de turmas e dimensionado pelo nome de cada uma (Figura 10). A funcionalidade de relatórios está disponível e funcional em todas as entidades da aplicação gerada podendo ser acessada pela listagem, portanto, é possível afirmar que a geração da aplicação foi bem sucedida.

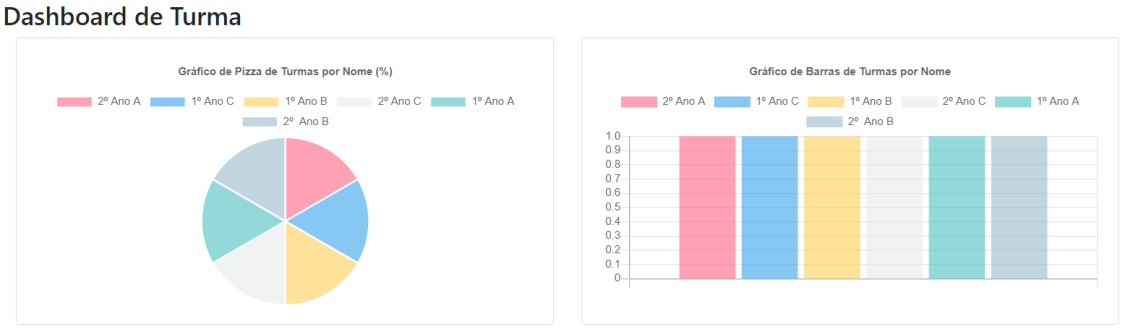

Figura 10. Relatórios de turma
Fonte: O autor.

5. Conclusão

Mediante os resultados apresentados, pode-se afirmar que a solução proposta viabiliza a geração de código para a funcionalidade de relatórios para sistemas web utilizando gerador de código de granularidade fina baseado na técnica de *Scaffolding*.

Para o desenvolvimento da solução, foram necessárias diversas etapas a fim de solucionar as dificuldades encontradas e gerar um código mais completo. Dentre estas dificuldades, pode-se destacar a transformação do código de referência em templates como sendo a mais desafiadora, haja vista a necessidade de um nível de conhecimento avançado e boas práticas de codificação.

Dessa forma, este trabalho poderá ser utilizado como base para a futura criação de diversos tipos de relatórios a partir dos metadados de uma aplicação, principalmente ao se tratar metadados de relacionamentos entre entidades distintas.

Outras evoluções possíveis para este trabalho são: a geração automatizada de processos de ETL (*Extract-Transform-Load*) para auxiliar a alimentação dos dados dos relatórios; e a geração de *pipelines* de eventos para relatórios em tempo real.

Referências Bibliográficas

Eliziario, W. D. S. (2014) "Framework Gerador de Relatórios".

Magno, D. G. (2015), "Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD".

Mens, T. (2008) "Introduction and roadmap: History and challenges of software evolution", In: Software evolution. Springer, Berlin, Heidelberg. p. 1-11.

Rajlich, V. (2014) "Software evolution and maintenance", In: Future of Software Engineering Proceedings. p. 133-144.

Rodriguez-Echeverria, R. et al. (2016) "AutoCRUD-automating IFML specification of CRUD operations", In: **International Workshop on Avanced practices in Model-Driven Web Engineering**. SCITEPRESS. p. 307-314.

Vilar, R., Oliveira, D. e Almeida, H. (2015) "Rendering patterns for enterprise applications", in: Proceedings of the 20th European Conference on Pattern Languages of Programs. p. 1-17.

Vilar, R., Oliveira, D. e Almeida, H. (2017) "Decomposição e reúso de componentes baseados em metadados para interfaces gráficas do usuário em aplicações corporativas web".

Zimmermann, T. R. (2006) "Desenvolvimento de um sistema de apoio à decisão baseado em business intelligence".