GERAÇÃO DE CÓDIGO PARA APLICAÇÕES COM ARQUITETURA *SERVERLESS*[1]

Diógenes Ataíde Chagas, Rodrigo de Almeida Vilar de Miranda

Departamento de Ciências Exatas - Universidade Federal da Paraíba (UFPB)
Rio Tinto - Paraíba, Brasil

{diogenes.ataide, rodrigovilar}@dcx.ufpb.br

Abstract: *New Serverless architectures reduce the cost of running and maintaining cloud services, so their use has grown steadily. The most commonly used code generation tools, such as Rails, Grails and jHipster, are designed to reduce the cost of software development, but do not have the ability to generate code for Serverless architectures. This paper presents an investigation into code generation for Serverless architectures, specifically AWS Lambda, and a proposal to evolve the state of practice in order to increase the percentage of activities that can be automated for the entire development, deployment, execution and management cycle of a simple CRUD service with AWS Lambda and DynamoDB. During this work, the percentage of automated lines of code increased from 24% to 73%.*

Keywords: *Serverless, AWS Lambda, Code Generation*

Resumo: As novas arquiteturas Serverless reduzem os custos de execução e manutenção de serviços em nuvem; portanto seu uso tem crescido continuamente. As ferramentas para geração de código mais utilizadas pelo mercado, como Rails, Grails e jHipster, buscam reduzir o custo de desenvolvimento de software, todavia não possuem a capacidade de gerar código para arquiteturas Serverless. Este trabalho apresenta uma investigação sobre geração de código para arquiteturas Serverless, especificamente AWS Lambda, e a proposta de evolução no estado da prática a fim de aumentar o percentual de atividades que pode ser automatizado para todo o ciclo de desenvolvimento, implantação, execução e gestão de um serviço de CRUD simples com AWS Lambda e DynamoDB. Durante este trabalho, o percentual de linhas de código automatizadas subiu de 24% para 73%.

Palavras-chave: Serverless, AWS Lambda, Geração de código

[1]Trabalho de conclusão de curso, sob orientação do professor Rodrigo de Almeida Vilar de Miranda submetido ao Curso de Bacharelado em Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE) da Universidade Federal da Paraíba, como parte dos requisitos necessários para obtenção do grau de Bacharel em Sistemas de Informação.

1. Introdução

Constantemente surgem no mercado de Tecnologia da Informação (TI), novas tecnologias, que são utilizadas para aumento da produtividade e redução de custos. A arquitetura *Serverless* (que em uma tradução livre quer dizer arquitetura *sem servidor*) tem ganhado espaço em eventos e fóruns que tratam de soluções em Computação na nuvem, pois esse conceito busca eliminar a necessidade de manter servidores em máquinas virtuais na nuvem, cobrando apenas pelo tempo de processamento realmente necessário para executar os serviços[2]. Além disso, é automatizada a distribuição e gestão dos serviços em escala global.

Nesse contexto, pode-se conectar a arquitetura *Serverless* à geração de código, como uma forma de propor um ciclo completo de redução de custo no mercado de software, desde o desenvolvimento até a implantação, execução e gestão de software.

A técnica de geração de código tem como propósito automatizar as tarefas repetitivas do processo de desenvolvimento de software, eliminando o gasto desnecessário de tempo e a falta de padronização do código (Spinelli, 2015, p.14).

O objetivo deste trabalho é aumentar o percentual de atividades que pode ser automatizado para o ciclo de desenvolvimento, implantação, execução e gestão de um serviço de CRUD[3] simples baseado em arquitetura *Serverless*. Mais especificamente:

- Estudar o estado da prática em geração de código para arquitetura Serverless;
- Elencar as atividades repetitivas para o desenvolvimento, configuração e implantação de serviços AWS Lambda com banco de dados DynamoDB;
- Implementar uma ferramenta para automatizar as tarefas repetitivas;
- Comparar o percentual de automação antes e depois da ferramenta.

Para realizar esse aumento na automação, foi desenvolvida parte de uma aplicação demonstrativa de gerenciamento de um *e-commerce* com as funções de pesquisar produto por meio de um identificador, adicionar, listar e remover produtos do estoque. Com isso, foi realizado o levantamento de quais artefatos podem ser gerados automaticamente através de *templates* de geração de código.

Complementando o artigo, a Seção 2 se refere ao referencial teórico, apresentando conceitos das tecnologias utilizadas; a Seção 3, descreve uma ferramenta para desenvolvimento de aplicações *serverless* de maneira automatizada; a Seção 4 mostra os resultados obtidos através da solução desenvolvida; a Seção 5 mostra a relação de outros trabalhos com este e, por fim, a Seção 6 apresenta as conclusões extraídas a partir dos resultados.

---

[2] https://medium.com/@BoweiHan/an-introduction-to-serverless-and-faas-functions-as-a-service-fb5cec0417b2

[3] As quatro operações básicas utilizadas na aplicação em base de dados: cadastrar, listar, editar e remover

2. Fundamentação Teórica

Nesta seção são apresentados os conceitos básicos de uma Arquitetura *Serverless*, bem como fundamentos de *FaaS*[4], obtidos a partir da investigação do estado da prática. Além disso, será apresentada uma revisão bibliográfica sobre geração de código geral e para arquiteturas *Serverless*.

2.1. *Serverless*

A arquitetura *Serverless* é um modelo que possui servidores, contudo, não se atribui ao cliente as atividades de gerência dos servidores, dado que o provedor de nuvem se encarrega de tal função, como uma forma de terceirizar os serviços de gerência de infraestrutura.

Essa arquitetura reduz o custo com o processamento de dados em nuvem, pois, em vez de manter um servidor virtual continuamente para executar os serviços, o cliente precisa apenas enviar o código do serviço para o provedor da nuvem. A execução do serviço ocorrerá de fato apenas quando ele for requisitado, e será cobrado apenas pelo tempo real da execução. Além disso, o provedor cuida de outros aspectos importantes, como escalabilidade flexível e alta disponibilidade, automaticamente**.**

Além de apresentar essas características, *Serverless* destaca-se por ser uma arquitetura orientada a eventos (Figura 1). Esse paradigma é definido por Gifrin e Ferreira (2017, n.p. tradução nossa) da seguinte forma:

> [...] Esse paradigma pode ser aplicado para automatizar fluxos de trabalho ao desacoplar os serviços que trabalham coletiva e independentemente para atender a esses fluxos [...].

Sendo assim, as funções *Serverless* atuam como um "observador" que recebe eventos de algumas fontes como: mudanças de estado na base de dados (se os dados estão sendo lidos, escritos ou deletados) ou requisições feitas através de *endpoints*[5]. A partir disso, todos os dependentes de um objeto são notificados ao mudar seu estado dinamicamente; além de permitir a adição de novos dependentes, definindo assim um modelo de relacionamento "um para muitos".

Com isso, nota-se que aplicações *Serverless* apresentam com frequência, "maior flexibilidade e são passíveis de mudança" (Newman, 2015) ou atualizações, tanto como um todo, quanto por meio de atualizações independentes de seus componentes; isso é possível pelo fato da arquitetura ser distribuída e permitir que enquanto seus componentes são atualizados, não seja comprometido o funcionamento da aplicação.

Atualmente, algumas organizações disponibilizam soluções voltadas para aplicações *Serverless*, dentre as que se destacam nesse cenário estão *Google*[6], *Amazon*[7] e *Microsoft*[8].

---

[4] *Function as a Service*
[5] URL onde o serviço pode ser acessado por uma aplicação cliente.
[6] https://aws.amazon.com/pt/serverless/
[7] https://cloud.google.com/serverless/

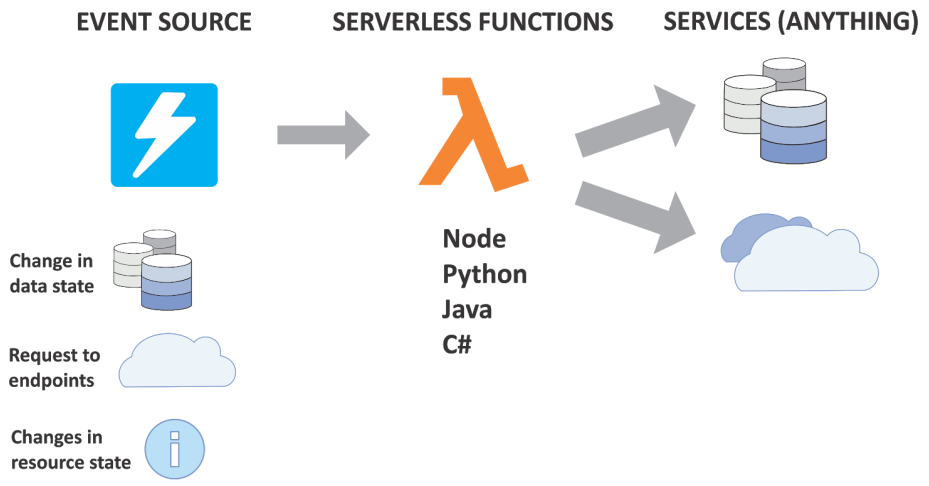

Figura 1. Exemplo do funcionamento da arquitetura orientada a eventos utilizando funções *Lambda*. Fonte: Slide share (2019)[9].

Entretanto, apesar da arquitetura Serverless trazer benefícios, assim como qualquer outra tecnologia, precisa-se de uma análise crítica, entendendo bem suas vantagens e desvantagens antes de implementar esse modelo de forma incorreta.

Por exemplo, em alguns casos, o *AWS Lambda* pode apresentar problemas com a latência[10] na hora de iniciar a execução de uma *FaaS*. Isso acontece por que uma aplicação que não está em uso pode ser completamente desativada, e quando é novamente necessária, é natural que leve um pouco mais de tempo para iniciar. Outro contraponto é o fato de tornar-se fortemente dependente de apenas um provedor, gerando altos riscos durante a migração para outro. Pois, será necessário modificar itens como, por exemplo, controle de acesso, banco de dados, o modelo de armazenamento e também pode ser necessário ajustar o código da sua aplicação.

2.2. *FaaS (Function as a Service)*

Basicamente, *Function as a Service* trata da execução do código *back-end* da aplicação sem a necessidade de gerenciar os serviços e aplicações do servidor, pois não requer nenhuma biblioteca específica ou codificação para algum ambiente de desenvolvimento ou linguagem de programação, já que o provedor de serviços gerencia e disponibiliza os recursos necessários para a execução do código.

No que tange ao processo de implantação, o código é carregado para o provedor *FaaS* e o mesmo se encarrega de instanciar *máquinas virtuais* ou *containers*, gerenciar processos e provisionar os demais recursos necessários para a execução da aplicação, algo que antes era da responsabilidade do desenvolvedor. Roberts (2018, n.p., tradução nossa) destaca que:

---

[8] https://azure.microsoft.com/en-us/overview/serverless-computing/

[9] Imagem retirada de: https://image.slidesharecdn.com/serverless-architecture-patter-e9abbe75-5b3f-4c5e-99bf-1d972843de5f-814659895-170927134900/95/serverless-architecture-patterns-4-638.jpg?cb=1506694774

[10] Diferença de tempo entre o início de um evento e o momento em que os seus efeitos se tornam perceptíveis.

> O escalonamento horizontal[11] é totalmente automático, elástico e gerenciado pelo provedor. Se o seu sistema precisar processar 100 solicitações em paralelo, o provedor lida com isso sem qualquer configuração extra.

2.3. Geração automática de código baseada em templates

A técnica de geração de código, além de fomentar a reusabilidade e manutenibilidade, tem como atribuição produzir funcionalidades semelhantes às que são implementadas manualmente, a partir de templates e parâmetros de entrada, reduzindo assim o esforço do desenvolvimento de software (Fowler 2002, p.13-17).

O funcionamento dos geradores de código, de maneira geral, consiste em: fornecimento de parâmetros de entrada: nome da aplicação, linguagem de programação, metadados do domínio, etc. Em seguida, haverá o processo de criação dos arquivos da aplicação e o gerador devolve o código-fonte da aplicação com os arquivos já alocados em pastas devidas.

Um exemplo de gerador de código é o *Ruby on Rails*[12] um *framework* que fornece uma infraestrutura para construir aplicações corporativas, recebendo como dados de entrada nome das entidades e das propriedades e o seu tipo; o código-fonte gerado consiste na arquitetura MVC[13]. Além de gerar o código com as operações CRUD, é possível executar a aplicação em um servidor web. A Figura 2 ilustra como funciona a geração de código para aplicações em *Ruby on Rails*.

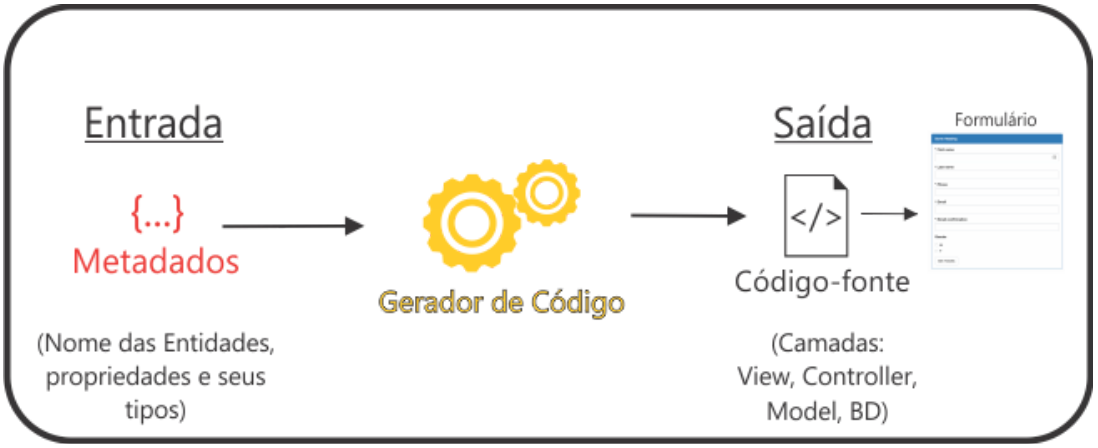

Figura 2. Exemplo do funcionamento do gerador de código *Ruby on Rails*. Fonte: Próprio autor (2019).

Outra vantagem notável, é a padronização percebida no código gerado, visto que esse pode se basear em um *template* padrão, o que também confere ao código-fonte um estilo único de programação.

O uso de *templates* fomenta o reuso, mas apresenta limitações devido à complexidade do código baseado em metadados. Assim sendo, é uma prática comum utilizar os geradores apenas para se obter uma versão inicial das aplicações; as adaptações que forem necessárias são efetuadas diretamente no código gerado, em vez de evoluir os templates de geração de

[11] Atividade de adicionar um novo servidor para uma aplicação de software distribuída
[12] https://rubyonrails.org/
[13] Model View Controller

código. Vilar et al. (2014) identificaram esse problema e propuseram uma linguagem de padrões para diminuir a granularidade dos geradores de código e assim, fomentar o seu reuso e evolução.

Para este trabalho, foi realizada uma investigação de ferramentas que pudessem gerar código para aplicações *serverless*; foi encontrado apenas o *Serverless Framework*[14]. Um gerador que, ao inserir os parâmetros de entrada (Figura 3), gera o código-fonte de uma aplicação *serverless*. Portanto, partindo do que foi encontrado na ferramenta verificou-se até que ponto ela gera o código-fonte da aplicação CRUD em *serverless* e o percentual que poderia ser automatizado, como pode ser visto no restante deste trabalho.

```
$ serverless create --template aws-java-maven --name application-api
  -p aws-java-application-api
```
[15]

Figura 3. Comandos utilizados para criar uma aplicação (*Serverless Framework*)

3. Metodologia

As atividades realizadas, com o objetivo de aumentar o percentual de atividades que podem ser automatizadas para o ciclo de desenvolvimento Serverless, foi:

I) investigação sobre *Serverless* e como gerar código para essa arquitetura;

II) desenvolver um backend *Serverless* para realização de CRUD, em uma aplicação com a temática de *e-commerce*;

III) identificar, no código e nos processo de desenvolvimento, as partes que poderiam ser automatizadas;

IV) modelar e implementar uma solução para gerar esse código e automatizar os passos identificados a partir do Tópico III;

V) medir o aumento percentual da automação em relação ao estado da prática. Esses passos serão detalhados nas próximas Subseções.

3.1. Investigação sobre *Serverless*

Neste trabalho foi realizada uma investigação através de buscas no portal de pesquisa *Google* utilizando palavras-chaves como *Serverless, AWS Lambda Functions e Function as a Service* durante o período de dois meses (agosto e setembro de 2018), sobre arquitetura *serverless*, investigando inicialmente seu conceito e funcionalidade; além disso, foi realizada uma pesquisa em busca de ferramentas que pudessem gerar código automaticamente para aplicações em *serverless*.

Sobre o método de pesquisa exploratório:

> *Este tipo de pesquisa tem como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a construir hipóteses[...]*. Gerhardt e Silveira (2009, p.35)

[14]Ferramenta de código aberto, disponível em: https://serverless.com/

[15] Referência aos comandos: 1) https://aws.amazon.com/pt/; 2) https://go.java/index.html; 3) https://maven.apache.org/; 4) nome dado à aplicação.

Os resultados da pesquisa encontram-se nas Seções 2 e 5 onde foram abordados os conceitos e fundamentos de *serverless* e trabalhos relacionados, respectivamente.

3.2. Desenvolvimento de uma aplicação *Serverless*

Após a fase de investigação, foi implementada uma aplicação *serverless* com a finalidade de servir de referência empírica para quantidade de código e atividades que podem ser automatizados. Assim, foi utilizada a plataforma ***AWS Lambda,*** da *Amazon Web Services,* que disponibiliza suporte para arquitetura *Serverless.* Essa ferramenta tem como característica a possibilidade de implementação do código que permite que a plataforma gerencie todos os recursos de infraestrutura.

Uma aplicação *back-end* foi desenvolvida utilizando a linguagem de programação Java, valendo-se de *REST API*[16], com auxílio da ferramenta *Serverless Framework*[17]. A aplicação[18] é um exemplo de gerenciamento de um *e-commerce*, que consiste na criação dos módulos CRUD possibilitando a criação, listagem, edição e exclusão de itens do banco de dados por meio da execução das funções *Lambda*.

Na Figura 4, tem-se uma representação gráfica de como a aplicação desenvolvida em *Serverless* funciona, a partir de onde será possível a realização do próximo passo: identificar pontos no código que podem ser automatizados.

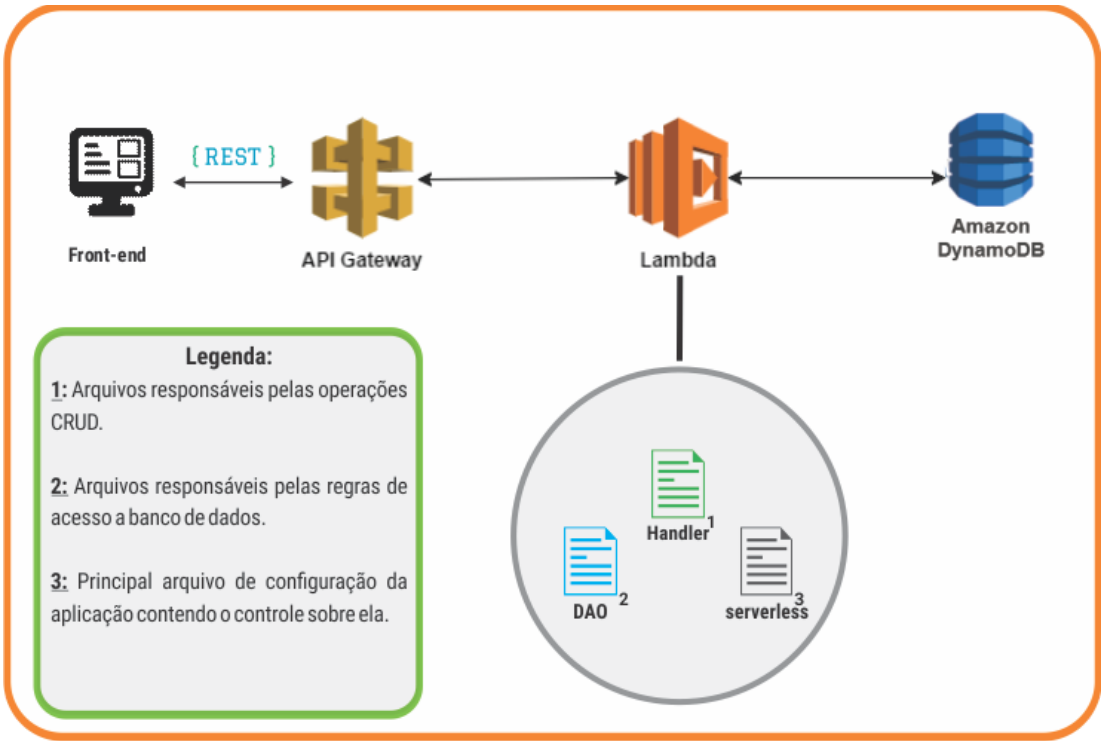

Figura 4. Como funciona a aplicação desenvolvida. Fonte: Internet (2019, adaptada) [19]

[16] *API* que usa solicitações **HTTP** para dados *GET, PUT, POST* e *DELETE*, frequentemente usadas no desenvolvimento de serviços da Web.
[17] https://serverless.com/framework/
[18] https://github.com/DioChagas/aws-java-tcc-api
[19] Imagem adaptada de:
https://cdn-ak.f.st-hatena.com/images/fotolife/a/acro-engineer/20160923/20160923092042.png

De acordo com a Figura 4, de forma sucinta percebemos que aplicação irá se comunicar, através de requisições REST, com o serviço ***API Gateway***; na ocasião, esse serviço identifica qual operação foi requisitada (*GET, PUT, POST* ou *DELETE*), para acionar o código *Lambda* que corresponde ao que foi requisitado; sendo que esse, quando executado possibilita que através das funções *Lambda* haja interação com o banco de dados por meio de eventos (*Handlers*) definidos na aplicação (cadastrar, listar, editar ou remover); os arquivos DAOs[20] que ficam por conta de gerenciar as regras e permissões de acesso ao Banco de Dados; e por fim a escolha de um Banco de Dados, *serverless,* utilizado neste trabalho, que é o *DynamoDB*.

3.3. Modelo de uma solução para gerar código e automatizar os passos

Nesta subseção, será demonstrado quais arquivos são estáticos ou dinâmicos na aplicação desenvolvida. Os arquivos estáticos independem dos metadados fornecidos pelo programador, diferentemente dos arquivos dinâmicos. Em seguida será apresentado o conjunto de templates e metadados desenvolvidos para gerar o código da aplicação.

A identificação de pontos estáticos e dinâmicos além de ser um mecanismo para cumprir o objetivo deste trabalho, também corrobora para que seja considerada a possibilidade de utilizar um gerador de código com *templates* de granularidade fina logo no início do projeto, a fim de que seja valorizado o tempo evitando repetição de atividades de desenvolvimento.

Para aumentar a eficiência e o percentual de geração de código, foi utilizada a linguagem de padrões proposta por Vilar et. al. (2014) através do *Potter Generator*[21], que consiste no uso de *templates* de granularidade fina e metadados (Figura 5), que são *templates* com poucas operações, divididas em vários "grãos" (*templates* menores), com funcionalidades bem específicas e muitas vezes independentes, facilitando a customização dos *templates*, além de permitir que o código-fonte fique minimamente acoplado, além de gerar o código idêntico ao que foi produzido manualmente.

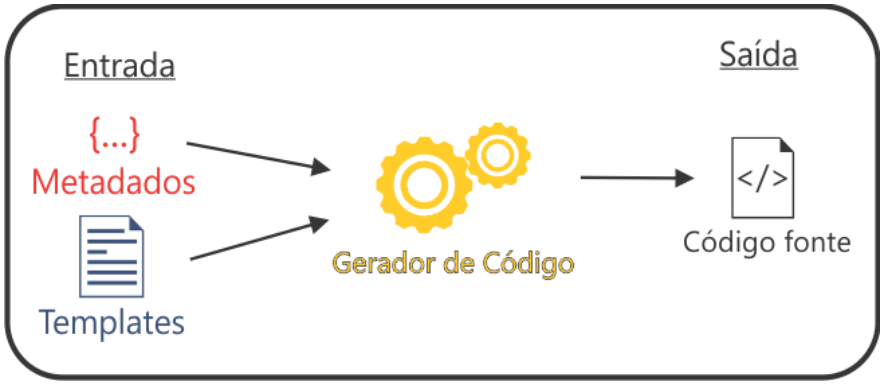

Figura 5. Exemplo do funcionamento do gerador de código proposto por Vilar et. al. (2014). Fonte: Próprio autor (2019).

[20] *Data Access Object*: São os arquivos que abstraem tudo que for referente ao acesso a dados da aplicação. Por exemplo, são nesses arquivos onde ficam as permissões e acesso ao DynamoDB na aplicação desenvolvida neste trabalho.

[21] https://github.com/potterjs/potter-project

A seguir, serão demonstrados quais arquivos possuem características estáticas e dinâmicas.

3.3.1. Identificação de pontos estáticos e dinâmicos no código e desenvolvimento da solução

Para constatar quais arquivos poderiam ser estáticos ou dinâmicos, foi analisado em cada um deles, trechos que teriam em comum, para o uso de *templates*.

Na Figura 6 é possível observar os artefatos que foram identificados como dinâmicos (na cor **amarela**) para a geração automática de código com uso de *templates*. Além desses há os artefatos que permanecem estáticos (na cor **azul**), para os quais não há necessidade de *templates*; o código desses arquivos não é alterado, independentemente do que é feito na aplicação.

Nesse procedimento foram criados *templates* de geração automática de código baseada em metadados. Conforme Vilar (2017, p.16), é através desses *templates* que há uma redução significativa na dependência entre os componentes de software otimizando o processo de desenvolvimento.

Outro ponto sobre os metadados é que, além de reduzir o acoplamento, os mesmos oferecem uma maneira de classificar, organizar e caracterizar os dados para a geração automática de código mantendo-o legível.

A solução desenvolvida neste trabalho utiliza o gerador de código baseado em metadados proposto por Vilar (2017), onde os metadados são utilizados para gerar novas entidades com suas respectivas propriedades no banco de dados, conforme a Figura 7.

As atividades de desenvolvimento contempladas neste trabalho foram as seguintes:

a. Criação dos arquivos das entidades;
   i. inserção de suas respectivas propriedades e atributos;

b. Criação dos arquivos responsáveis pelas operações CRUD (*Handles*);

c. Preenchimento do arquivo de configuração com as seguintes propriedades:
   i. Configuração de acesso ao Banco de dados (*DynamoDB*) e suas operações (consulta, inserção e remoção de itens);
   ii. Eventos das funções *Lambda* (representadas pelos arquivos *Handlers*)
   iii. Propriedades das tabelas (definição da chave primária, por exemplo)

d. Implantação da aplicação nos servidores da *AWS*;

Das atividades anteriormente listadas, todas foram contempladas quanto à automatização de seus procedimentos. Apenas a última (atividade de implantação) que não está dentro do escopo do trabalho. Portanto, para realizar ela, utilizou-se a como apoio a ferramenta *Serverless Framework*, que possui as ferramentas necessárias para a implantação dos arquivos nos servidores de *Cloud*.

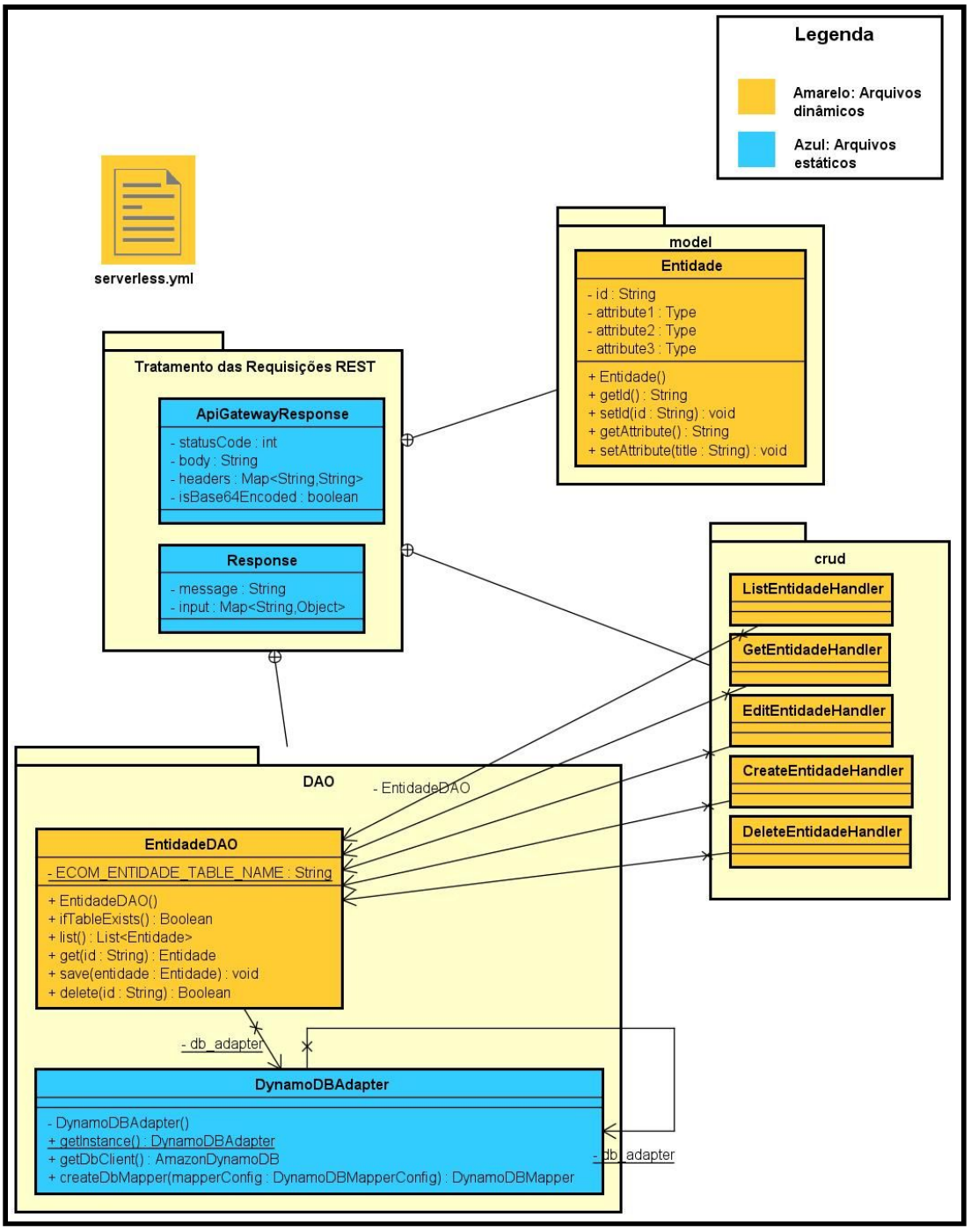

Figura 6. Diagrama de classes com os arquivos estáticos e dinâmicos

Embora o trabalho desenvolvido por Vilar (2017) proponha em seu escopo um gerador que possibilite o reúso de componentes para Interfaces Gráficas, no contexto de aplicações corporativas web foi possível adaptar seu uso na geração de código para aplicações em *Serverless* a partir da criação dos templates baseados nos metadados, que são inseridos como parâmetros demonstrados na Figura 7 e no Quadro 1 (localizado na subseção 4.1), respectivamente.

```
10    "entities": [
11      {
12        "name": "Livro",
13        "labels": {
14          "singular": "Livro",
15          "plural": "Livros"
16        },
17        "id": "id",
18        "properties": [
19          { "name": "titulo", "label": "Titulo", "type": "string" },
20          { "name": "autor", "label": "Autor", "type": "string" },
21          { "name": "edicao", "label": "Edicao", "type": "int" }
22        ]
23      }
24
```

Figura 7. Exemplo de uma entidade e suas respectivas propriedades

4. Resultados

Como resultado da pesquisa foi desenvolvido um gerador de código para aplicações em *Serverless* que recebe como parâmetros de entrada os metadados (demonstrado na Figura 7) e os *templates*.

Assim, a solução desenvolvida possibilita a automatização de procedimentos como: criação de entidades no banco de dados *serverless* (DynamoDB); criação dos arquivos Java com os respectivos atributos e propriedades das entidades, além de gerar o código semelhante ao que é produzido manualmente.

Como complemento neste trabalho, a ferramenta *Serverless Framework* terá o papel de automatizar a implantação da aplicação. A partir da sua instalação, por meio da linha de comando do Sistema Operacional[22] é possível, com um comando[23], implantar a aplicação no ambiente *serverless*.

A seguir, nas subseções **4.1** e **4.2** temos a demonstração do código-fonte sem o uso dos templates no Quadro 1 (A) e com sua utilização (B), utilizando uma das entidades criadas no banco de dados. As responsabilidades foram divididas da seguinte forma: os dados referentes ao projeto como um todo, por exemplo, o caminho onde se encontram os pacotes, estão marcados de **laranja**; os metadados da entidade estão marcados em **vermelho**; e os metadados das propriedades da entidade são mapeadas em **verde**.

[22] Através do comando (grifo nosso): ***npm install serverless -g***
[23] Comando: ***sls deploy***

4.1. Trecho de código produzido manualmente (A)[24] e código do template de granularidade fina (B)[25]

Nesta subseção, está o quadro com o código da aplicação *Serverless* desenvolvido manualmente (coluna A) e o código do template de granularidade fina (coluna B), respectivamente. Esta demonstração visa esclarecer em quais pontos no código as propriedades podem ser substituídas por parâmetros a fim de que haja melhor aproveitamento no uso do tempo para desempenhar atividades de desenvolvimento.

(A) Trecho produzido manualmente

```
1 package br.com.ufpb.dcx.si.model;
…
9 @DynamoDBTable(tableName =
"PLACEHOLDER_ECOM_LIVROS_TABLE_NAME")
10 public class Livro {
11      private String idLivro;
12      private String titulo;
13      private String autor;
14      private int edicao;
        …
17      public Livro() {
18      // TODO Auto-generated constructor
19      }
20      @DynamoDBHashKey(attributeName =
        "idlivro")
21      @DynamoDBAutoGeneratedKey
22      public String getIdLivro() {
23              return this.idLivro;
24      }
25
26      public void setIdLivro(String
idLivro) {
27              this.idLivro = idLivro;
28      }
        …
}
```

(B) Código do template de granularidade fina

```
1 package <%-$.invoke('make-package')%>.model;
…
9 @DynamoDBTable(tableName = "PLACEHOLDER_<%=
metadata.project.name.toUpperCase()
%>_<%=entity.name.toUpperCase()%>_TABLE_NAME")
10 public class <%=entity.name%>{
11      private String id<%=entity.name%>
12      <%-$.invokeLoop('property-entity',
entity.properties)%>
13      public <%=entity.name%>() {
14              // TODO Auto-generated constructor
15      }
16      @DynamoDBHashKey(attributeName="id<%=
entity.name.toLowerCase() %>")
17      @DynamoDBAutoGeneratedKey
18      public String getId<%=entity.name%>(){
19         return this.id<%=entity.name%>;
20      }
21      public void setId<%=entity.name%>(String
        id<%=entity.name%>){
22      this.id<%=entity.name%>=id<%=entity.name%>;
        }
...
25 <%-$.invokeLoop('property-get',entity.properties)%>
26 <%-$.invokeLoop('property-set',entity.properties)%>
}
```

Quadro 1: Na coluna (A), código desenvolvido manualmente; na coluna (B), código do template de granularidade fina.

Nota-se que em alguns pontos do código, algumas propriedades da entidade foram substituídas por parâmetros utilizados através da linguagem de padrões do *Potter Generator*.

[24] https://github.com/DioChagas/aws-java-tcc-api
[25] https://github.com/DioChagas/potter-serverless

O nome da entidade, por exemplo, ao invés de inseri-lo diretamente, usa-se o seguinte parâmetro: ***public <%=entity.name%>;*** que em seguida será substituído pelo nome da entidade que fora passado no arquivo de metadados (Figura 7).

Isso demonstra que ao invés do desenvolvedor criar as entidades uma-a-uma, ele poderá informar no arquivo de metadados quais serão utilizadas na aplicação, e assim, serão gerados os respectivos arquivos das entidades do projeto.

4.2. Código do template referindo-se às propriedades e atributos da Entidade

Tomando como exemplo os atributos e as propriedades dos *gets* e *sets* da Entidade, pode-se observar o código que é invocado pelos trechos das linhas 25 e 26 presentes no Quadro 1 (B), a seguir, no Quadro 2, refletindo a interação entre os *templates* da Entidade e dos *gets* e *sets* das suas respectivas propriedades.

Desse modo, o código apresenta-se com granularidade fina e as classes ficam independentes dos *gets* e *sets*. Permitindo que o desenvolvedor tenha sua atenção voltada para a inserção dos metadados ao invés de preocupar-se com com cada arquivo das Entidades para inserir seus atributos e propriedades.

```
   /**
    * Gets the <%= property.name %>.
    ...
    */
    @DynamoDBAttribute(attributeName = "<%= property.name.toLowerCase() %>")
    public <%- $.invoke('property-type', property) %> get<%=
property.name[0].toUpperCase() %><%= property.name.substring(1) %>() {
        return <%= property.name %>;
    }

   /**
    * Sets the <%= property.name %>.
    ...
    */
    public void set<%= property.name[0].toUpperCase() %><%= property.name.substring(1)
%>(<%- $.invoke('property-type', property) %> <%= property.name %>) {
        this.<%= property.name %> = <%= property.name %>;
    }
```

Quadro 2: Código dos *gets* e *sets* das propriedades da entidade que será invocado no template da Entidade no Quadro 1 (B).

Após o uso dos *templates* notou-se que os (templates) que possuem granularidade fina reduzem significativamente o trabalho repetitivo no desenvolvimento, dado que eles proporcionam que seja gerado um código idêntico ao que foi desenvolvido manualmente, semelhante ao Quadro 1 (A).

A seguir, na Tabela 1, contém informações das métricas e das ferramentas utilizadas para mensurar o percentual de automatização do código. Na tabela, nota-se as duas ferramentas utilizadas neste trabalho para geração de código na arquitetura *serverless* (*Serverless Framework e Potter Generate*); os resultados obtidos por cada uma - tanto em número de arquivos como o percentual de automatização - em relação à aplicação final, representada pela coluna (**Artefatos funcionais completos**).

Métrica	**Artefatos funcionais completos**	**Serverless framework (%)**	**Potter Generate (%)**
Quantidade de arquivos	35	6 (17%)	27 (77%)
Quantidade de linhas de código	1224	298 (24%)	896 (73%)

Tabela 1: Resultados obtidos a partir das métricas estabelecidas: quantidade de arquivos e linhas dos mesmos. Fonte: Próprio autor (2019)

As métricas que este trabalho empenhou-se em de avaliar, foram: **quantidade total de linhas de código** e **de arquivos gerados**; para que a partir disso, fosse mensurado o ganho na automatização da geração do código da solução desenvolvida.

Como demonstrado na Tabela 1, além das métricas já mencionadas, percebe-se o percentual de automatização que as ferramentas obtiveram em cada uma das métricas. Assim, nota-se um aumento significativo do percentual de automatização em relação à aplicação final, com o uso do *Potter Generate* para a criação da aplicação em *Serverless*. Destacando a importância do uso do gerador nas atividades de desenvolvimento de software, neste caso, para aplicações em *Serverless*.

5. Trabalhos relacionados

Durante o período de pesquisa para o desenvolvimento deste trabalho foram verificadas algumas ferramentas de geração de código com o intuito de que as mesmas suprissem automatização no desenvolvimento de aplicações *Serverless*.

A primeira, *JHipster*[26] , é uma plataforma de desenvolvimento para gerar, desenvolver e implantar aplicativos *Spring Boot*, *Angular* / *React Web* e arquitetura de microsserviços além de gerar *workflow* para *build* das aplicações utilizando ferramentas automação de compilação. Esta por sua vez, apesar de gerar código e automatizar processos, não dispõe estes recursos para aplicações *Serverless*.

[26] https://www.jhipster.tech/

Entretanto, apenas uma ferramenta encontrada a *Serverless Framework*[27], supre os recursos de geração de código e automatização no desenvolvimento em aplicações *Serverless* utilizando linguagens de programação como: *Java*[28], *Python*[29], *Node.js*[30], entre outras linguagens para diversos provedores de nuvem, além de disponibilizar de guias e documentação para cada provedor. Contudo, quanto à abrangência dos serviços de geração de código, esta se limita a gerar apenas os artefatos demonstrados na Figura 8.

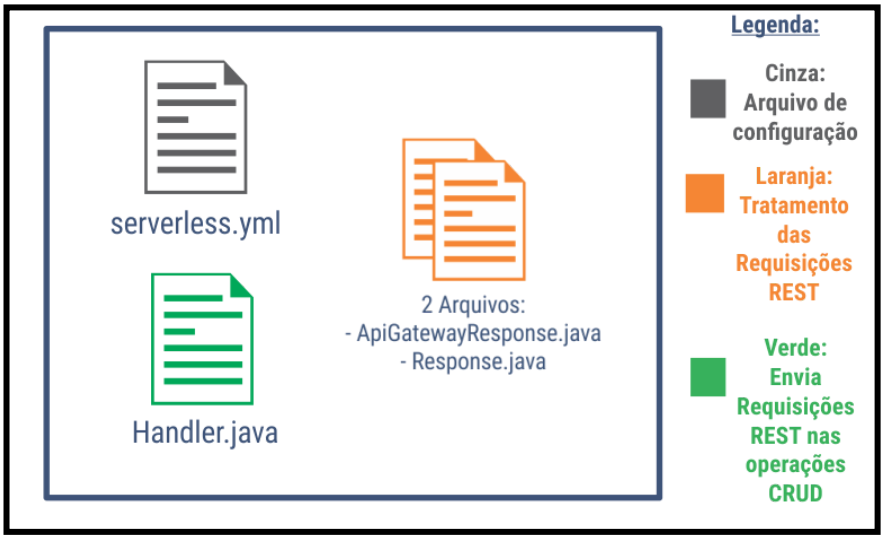

Figura 8. Arquivos gerados automaticamente pela ferramenta *Serverless Framework*. Fonte: Próprio autor (2019)

Como observado na Figura 8, cada cor representa um tipo de arquivo. Por exemplo, o arquivo *serverles.yml* (**cinza**) está contido as configurações de acesso ao Banco de dados (DynamoDB) e suas operações (consulta, inserção e remoção de itens); no arquivo *Handler.java* (**verde**) são responsáveis pelos eventos das funções *Lambda*; e por fim, nos arquivos *ApiGateway.java* e *Response.java* (**laranja**) estão contidos os padrões de tratamento das requisições e suas respostas mediante as situações de: indisponibilidade do serviço ou sucesso na conexão.

Ainda de acordo com a Figura 8 foi necessária criação de outros artefatos para que assim os templates pudessem ser definidos; esses são apresentados na Figura 6, nos pacotes **DAO, CRUD e Model**. Portanto, o gerador utilizado neste trabalho possibilitou a criação de mais artefatos, inclusive os de banco de dados, otimizando o desenvolvimento da aplicação na arquitetura *Serverless*.

6. Conclusões

Este trabalho apresenta a utilização de um gerador de código baseado em templates de granularidade fina com o intuito de otimizar o processo de desenvolvimento de aplicações na

[27] https://serverless.com/
[28] https://www.java.com/pt_BR/
[29] https://www.python.org/
[30] https://nodejs.org/

arquitetura *Serverless*. Foi criada uma aplicação *back-end* em *Serverless*, na linguagem de programação Java para que assim fosse realizado o levantamento do percentual do código que poderia ser automatizado.

Também foram analisadas ferramentas e aplicações que geram código baseado em parâmetros como dados de entrada para gerar os artefatos necessários para o desenvolvimento da aplicação.

No entanto, foram encontradas limitações das ferramentas levantadas; seja por não haver suporte à arquitetura *Serverless* ou, a quantidade limitada de artefatos fazendo com que o desenvolvedor tenha que criar novos arquivos. A partir disso, este trabalho buscou suprir as limitações dos demais geradores de código já existentes proporcionando a utilização de uma ferramenta de maior abrangência quanto a geração de código para aplicações em *Serverless*.

Dessa forma, a ferramenta *Serverless Framework* gerou 6 arquivos (representando 17% dos artefatos funcionais completos) com um total de 298 linhas (24% em relação ao total de linhas dos artefatos funcionais); a ferramenta *Potter Generate* aumentou esse número para 27 arquivos (77% do total de artefatos funcionais completos), o que representou um ganho de 60% em relação ao *Serverless Framework*, além de um total de 896 linhas (73% em relação ao total de linhas dos artefatos funcionais), representando um ganho de 49%.

Entretanto, este trabalho apesar de proporcionar o desenvolvimento de uma ferramenta que facilite a customização dos templates e gere código para aplicações na arquitetura *Serverless*, foi possível observar que melhorias podem ser realizadas dentro do gerador de código que já foi desenvolvido. Uma delas é automatizar a implantação da aplicação *Serverless* para que haja simplificação de tarefas como o *upload* de pacotes de implantação e a atualização da aplicação.

REFERÊNCIAS

FOWLER, Martin. **Using Metadata. Software, IEEE,** 2002. pp. 13–17. Disponível em: <https://www.martinfowler.com/ieeeSoftware/metadata.pdf> Acesso em: 21 de jan. de 2019.

GIFRIN, Christie.; FERREIRA, Otavio. **Event-Driven Computing with Amazon SNS and AWS Compute, Storage, Database, and Networking Services,** 2017. Disponível em: <https://aws.amazon.com/pt/blogs/compute/event-driven-computing-with-amazon-sns-compute-storage-database-and-networking-services/>. Acesso em: 30 de jan. de 2019.

GERHARDT, Tatiana Engel.; SILVEIRA Denise Tolfo. **Métodos de pesquisa**, 2009. UFRGS. – Porto Alegre: Editora da UFRGS, 2009. Disponível em: <http://www.ufrgs.br/cursopgdr/downloadsSerie/derad005.pdf> Acesso em: 25 de jan. de 2019.

NEWMAN, Sam. **Building Microservices. 1. Ed**. O'Reilly Media, Inc, Gravenstein Highway North, Sebastopol, USA, 2015. Cap. 2, p. 19.

ROBERTS, Mike. **Serverless Architectures,** 2018. Disponível em: <https://martinfowler.com/articles/serverless.html>. Acesso em: 12 de fev. de 2019.

SPINELLI, Lucas Pompeo Pontes. **DESENVOLVIMENTO DE UMA FERRAMENTA PARA GERAÇÃO AUTOMÁTICA DE CÓDIGO ABERTO EM JAVA SERVER FACES**. 2015. 46p. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação), Fundação Educacional do Município de Assis – Assis, 2015. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1211330440.pdf>. Acesso em: 15 de fev. de 2019.

VILAR, Rodrigo; OLIVEIRA, Delano; ALMEIDA, Hyggo. **Rendering patterns for enterprise applications.** In: Proceedings of the 20th European Conference on Pattern Languages of Programs. ACM, 2014. p. 22.