Geração Automática de APIs REST a Partir de um Modelo Aberto de Descrição de Serviços

Bruno Cesar Batista
Universidade Estadual do Norte do Paraná
Bandeirantes, Brasil
bruncbatista@gmail.com

Wellington Aparecido Della Mura
Universidade Estadual do Norte do Paraná
Bandeirantes, Brasil
wellington@uenp.edu.br

***Abstract*—Due to the need for interoperability between an ecosystem of technologies, the use of data access interfaces (APIs) has become highly esteemed among developers. The vast majority of these interfaces use common features in development (authentication via token, notifications, sending e-mails, among others) that take up developers' time (rework). This document implements a tool, capable of generating code based on the principles of clean architecture. From an interactive documentation, which makes reference to the necessary attributes and resources for the problem in question, an interpreter will be responsible for generating code with the entire base of the system implemented (declaration of common attributes and functionalities), the code aims to optimize the development time of interfaces assisting the development teams, in addition to providing a code based on clean architecture that allows enormous robustness and ease of maintenance. The generated code will be implemented in accordance with the criteria imposed by the REST standard.**

***Resumo*—Em virtude da necessidade de interoperabilidade entre um ecossistema de tecnologias, o uso de Interface de Programação de Aplicações (API) se tornou muito estimado entre os desenvolvedores. Estas interfaces, na grande maioria, utilizam de funcionalidades comuns no desenvolvimento que ocupam tempo dos desenvolvedores com retrabalho. Este trabalho implementa uma ferramenta, capaz de gerar código baseado nos princípios da arquitetura limpa. A partir de uma documentação interativa, que faz referência aos atributos e recursos necessários para o problema em questão, um interpretador pode gerar código com uma grande parte da base do sistema implementada (declaração de atributos e funcionalidades comuns). O código tem como objetivo otimizar o tempo de desenvolvimento auxiliando as equipes desenvolvimento, além de fornecer um código baseado em arquitetura limpa que permite uma enorme robustez e facilidade de manutenção. O código gerado é implementado em conformidade com os critérios imposto pelo padrão REST.**

***Palavras-chave*—Arquitetura limpa; REST API; Geração de código.**

I. Introdução

Em virtude do grande ecossistema de dispositivos tecnológicos e a necessidade de interoperabilidade entre esses dispositivos, o uso das APIs (*Application Programming Interface*) torna-se um diferencial de mercado para as organizações [1]. Uma API permite que soluções ou dados internos de uma aplicação sejam expostos através de um serviço para desenvolvedor interno ou externo. Em contexto geral, as API visam o aumento da produtividade, por meio da maximização da reutilização de código e lógicas; e agregar valor de negócio a aplicações de terceiros, pois adicionam ao seu código uma lógica ou um conjunto de funcionalidades externos [2] [3].

Existem diferentes padrões ou modelos arquiteturais para a implementação de uma API. O padrão ou modelo arquitetural REST se caracteriza como o mais popular e o mais utilizado no desenvolvimento. Uma API é denominada API REST ou RESTful somente quando ela está em conformidade com todos as restrições impostas no modelo REST [4]. O modelo REST se caracteriza pelas restrições que visam garantir maior desempenho, escalabilidade, simplicidade, portabilidade, confiabilidade e visibilidade para as aplicações [3]. Apesar das restrições do modelo REST, não é dispensado o uso de uma arquitetura de software, visto que o uso em conjunto permite o desenvolvimento de uma aplicação robusta e de fácil manutenção.

Durante o projeto de um software, o processo de projeto de arquitetura é o primeiro estágio, visto que deve ser realizado em fases em que o impacto de mudança é relativamente pequeno, permitindo a implementação da arquitetura desejada [5]. Uma arquitetura quando bem planejada torna o software desacoplado, independentemente de qualquer tecnologia externa, permitindo que decisões, como escolher qual sistema de banco de dados será utilizado ou qual servidor web será utilizado na implantação, sejam postergadas [6]. Dessa forma, MARTIN [6] propõe uma arquitetura de software capaz de independer de qualquer agente externo.

Uma API, quando embasada em uma boa modelagem e documentação, pode ser uma vantagem competitiva de mercado para as organizações. As API entregam uma interface de acesso a lógicas e dados internos de uma determinada aplicação ou serviço que permite os desenvolvedores internos aumentarem a

produtividade e os valores de negócio [2] [3] [4]. Com a grande popularização das API devido a necessidade de interoperabilidade entre o enorme ecossistema de tecnologias atuais [1], soluções para agilizar o desenvolvimento são necessárias.

O tempo gasto com o desenvolvimento de funcionalidades comuns é a grande motivação para o desenvolvimento da solução que será abordada neste trabalho. O desenvolvimento de funcionalidades bases no início de um projeto consome muito tempo dos desenvolvedores. Funcionalidades como a configuração de autenticação, notificações, envio de e-mails, criação de entidades base, manipulação básica de dados, entre outras funcionalidades que fazem parte do principal fluxo do projeto.

Este trabalho tem o objetivo de descrever o desenvolvimento de uma ferramenta capaz de gerar código de uma API REST fundamentada nos princípios de desenvolvimento de uma arquitetura de software limpa criada por Robert C. Martin [6], com objetivo aumentar a produtividade no desenvolvimento de aplicações, inibindo o processo de desenvolvimento da base das aplicações. No geral, o trabalho implementa uma ferramenta capaz de agilizar o processo de desenvolvimento de aplicações, gerando parte do código de uma API fundamentada nos princípios da arquitetura limpa, permitindo desenvolvedores adotarem quais tecnologias desejarem nas funcionalidades geradas pela ferramenta. Por favor note que a ferramenta proposta só é capaz de gerar código se tiver origem de uma boa modelagem da API, fundamentada nos requisitos levantados durante a análise.

II. Referencial Teórico

Nesta seção são apresentados conceitos relacionados a API que atendem as restrições impostas no modelo arquitetural REST, como também métodos de modelagem e ferramentas utilizadas neste processo; os conceitos de uma arquitetura de software limpa também são descritos, assim como a contextualização da engenharia de software e o seu papel no projeto de sistema de software.

A. API REST

Uma API é uma interface para um software que permite com que outros sistemas acessem sua lógica e dados internos, deste modo, garantindo a interoperabilidade entre sistemas de diferentes organizações. Novos sistemas ou negócios surgem fundamentados no funcionamento de uma determinada API, uma vez que, toda a lógica por trás dos resultados está oculta e os serviços de terceiros não devem se preocupar com a solução do problema, apenas com a realização das chamadas e o retorno de uma resposta [2] [3].

API são necessárias porque auxiliam na solução de um problema comum no desenvolvimento de software: o reuso de código. A implementação de uma API que expõe determinada solução de uma organização soluciona em partes o reuso de código, visto que, para consumir determinada funcionalidade, basta realizar uma chamada a um recurso de uma API e seu código teria aquela lógica sem ao menos implementar uma solução local no projeto [2]. Uma API pode ser consumida de maneira interna e externa. Uma API desenvolvida para o consumo interno visa o aumento na produtividade das equipes e a reutilização dos códigos. Quando desenvolvida para o consumo externo, visa agregar valor de negócio de outras organizações, além de permitir o aprimoramento das suas funcionalidades internas por desenvolvedores de terceiros [3].

Em alguns casos existem estratégias de negócio no desenvolvimento de uma API, como explica JACOBSON et al [1] o mercado mudou, as API passaram a ser relevantes para os negócios que necessita de uma interoperabilidade entre diferentes dispositivos, ou até mesmo quando se necessita de uma experiencia de “navegação contínua”, como citado, o caso da Netflix [7] que permite o usuário a iniciar o *streaming* de vídeo em um dispositivo e continuar de onde parou em um dispositivo completamente diferente. Algumas necessidades de negócio podem surgir que motivam a criação e uso de API, como: a necessidade de um segundo aplicativo móvel, clientes ou parceiros que solicitam uma API, a necessidade de fornecer um conteúdo ou serviço de forma flexível; entre outras.

Segundo JIN et al [2] definições são necessárias para a criação de uma API capaz de ser passar por possíveis manutenções e expansões se necessário, então padrões de desenvolvimento são adotados para a implementação da interface que tem como objetivos expor dados de um serviço, entre eles: REST, RPC e GraphQL.

Um dos padrões mais comuns de implementação de APIs, é o padrão REST [2]. Esse padrão ou modelo arquitetura surgiu durantes os primeiros anos da década de 2000 e tem como objetivo aprimorar o desempenho, escalabilidade, simplicidade, portabilidade, confiabilidade e visibilidade das interfaces que adotam suas restrições [3].

O padrão REST utiliza dos métodos definidos no protocolo HTTP para associar uma determinada ação a um recurso, sendo os métodos GET, POST, PUT e DELETE associados respectivamente as seguintes ações, RECUPERAR, CRIAR, ATUALIZAR E DELETAR. Os códigos definidos no protocolo HTTP também são utilizados para adicionar significados aos retornos das chamadas a uma determinada resposta de um recurso, as faixas 1XX, 2XX, 3XX, 4XX e 5XX do protocolo HTTP são adicionadas as respostas para informarem de maneira resumida qual a situação daquela solicitação. As respostas no modelo REST geralmente são bem estruturadas e comumente baseadas na estrutura JSON, além de possibilitarem o desenvolvedor a implementar a resposta em diferentes

representações, como XML, texto, entre outras formas [2] [3].

Algumas restrições são impostas pelo padrão, segundo PATNI [3] o modelo impõe como restrição a implementação dos seguintes requisitos: Cliente-Servidor, Uniform Resource Interface, Layered System, Caching, Stateless e HATEOS.

Uma API que está em conformidade com o padrão/modelo REST é considerada uma API REST ou uma API RESTful como caracterizado em outras literaturas. Portanto, uma API que implementa os conceitos abordados anteriormente pode obter uma vantagem de mercado em relação a seus concorrentes, tendo em vista que APIs REST bem projetadas atraem mais usuários para seus serviços. Uma boa execução de um projeto de uma API REST está diretamente associada a uma boa modelagem [4].

B. Modelagem de Uma API REST

A modelagem de uma API e seu fator de qualidade está extremamente vinculado quando falamos no sucesso de uma API [4] [8]. Utiliza-se de uma cadeia de práticas recomendadas para se obter o máximo de uma API, tal modelagem está implícita no protocolo HTTP, utiliza de métodos e semânticas impostas pelo protocolo [2] [4].

Uma API mal projetada pode assumir terríveis consequências, além de estarem totalmente fadadas ao fracasso geralmente são difíceis de serem entendidas e usadas. API mal projetadas podem criar vulnerabilidades de segurança, permitindo pessoas mal intencionadas abusarem para benefício próprio ou de terceiros. Uma modelagem defeituosa pode gerar custos para correção [8].

Para JIN et al [2] a modelagem se caracteriza em dois casos: API modelada especificamente para Casos da Vida Real; e API modelada para uma Boa Experiência do Desenvolvedor. Na modelagem para Casos da Vida Real, os Casos de Usos são definidos para gerar uma API a partir desses casos que são validados e verificados com os possíveis desenvolvedores que irão consumir tal API. Já a Modelagem para uma Boa Experiência do Desenvolvedor são fundamentais para não ocorrer desistência no uso de sua API. O foco na experiência enfatiza características como ser rápido e fácil de começar, ter consistência, seja um facilitador na solução de problemas e sua API deve estar em condições de possíveis evoluções.

A modelagem para uma boa experiência do desenvolvedor, abordada por JIN et al [2] está estritamente relacionada aos conceitos empregados na obra de LAURET [8]. Uma boa experiência de uso pode conquistar a atenção dos desenvolvedores que são os principais consumidores de uma API, tal experiência é possível através de uma boa modelagem.

Uma API REST tem como principal característica em sua modelagem a consistência, isso se refere a sua nomenclatura e estruturação de recursos, parâmetros de entrada e resposta de saída [2]. Há regras que abrangem todos os aspectos de nomenclatura e estruturação dentro de uma modelagem, e serão abordadas nos próximos parágrafos [4].

Para os recursos, sua modelagem é composta por diferentes tipos, para estes tipos existem regras, que tornam a modelagem mais consistentes. Os diferentes tipos são caracterizados por MASSÉ [4]:

- Coleção (collection): diretório de recursos do servidor.
- Armazenamento (store): diretório de um recurso gerenciado pelo cliente.
- Documento (document): um objeto ou registro.
- Controlador (controller): são funções executáveis.

Estes recursos quando empregados em uma URI (*Uniform Resource Identifier*), ou seja, em uma cadeia usa para identificar ou denominar um recurso, devem seguir o modelo apresentado no diagrama da Figura 1.

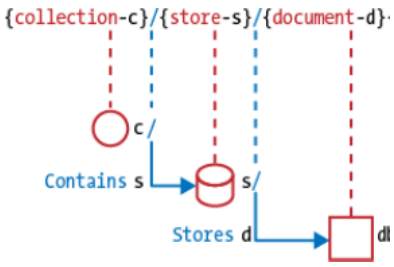

Fig. 1. Modelo de Recurso [4]

As regras de nomenclatura de recursos são estabelecidas visando padronização e consistência nos segmentos criados, parte das regras propostas por MASSÉ [4] são descritas abaixo:

- Um substantivo no singular deve ser usado para nomear os documentos;
- Um substantivo no plural deve ser usado para nome de coleção;
- Um substantivo no plural deve ser usado para nome de armazenamento;
- Um verbo ou frase verbal deve ser usado para nome de controladores.

Tarefas de manipulação dos modelos de recursos não devem ser feitas através das URIs. Uma API REST está fundamentada no protocolo HTTP e utiliza dos métodos propostos, que cada um possui uma semântica, para executar funções através de chamadas [4]: GET (recuperar); PUT (atualizar); DELETE (remover); e POST (criar).

Para JIN et al [2] uma API REST é responsável por facilitar a solução de problemas, ou seja, durante o processo de modelagem de uma API devemos pensar em soluções para os possíveis erros durante o seu uso. Os erros devem ser significativos, devem obstruir alguns problemas (como por

exemplo erros na base de dados), códigos legíveis e mensagens em alto nível são características importantes durante o processo retorno de erros. Os códigos HTTP servem para caracterizar e estruturar os possíveis erros de uma API [4].

Um ponto importante a ressaltar é que ferramentas interativas podem auxiliar no processo de modelagem e documentação de uma API REST. Documentações são necessárias para suprir as necessidades dos desenvolvedores durante o consumo das APIs. Ferramentas interativas podem captar novos desenvolvedores para o seu produto, visto a possibilidade de realizar testes sem grandes esforços [2].

Diversas ferramentas propõem um ambiente que é possível modelar uma API REST através de uma plataforma interativa, ou seja, é possível mapear e nomear os recursos que serão utilizados, atributos de cada modelo da sua interface, possíveis códigos de erro de cada recurso e seu significado, entre outras possibilidades que serão abordadas no capítulo de desenvolvimento.

C. Engenharia de Software

Nos últimos tempos a utilização de softwares cresceu significativamente, visto que estamos vivendo em um mundo totalmente conectado pela internet e os softwares se caracterizam por ser a porta de entrada para esta comunicação. Por este motivo, a engenharia de software é essencial, porque tem que como principal objetivo de garantir a qualidade de um produto de software através de um processo sistemático [9] [5].

Todo tipo de software, independente da sua forma, tipo e aplicação, passam pelos processos de engenharia. O processo de engenharia de software é caracterizado pelo emprego de princípios sólidos da engenharia com o objetivo de obter um produto de software econômico, confiável e eficiente [5].

No âmbito da engenharia de software a busca por métodos ou técnicas universais não são válidas, em virtude de que tipos de software exigem abordagens diferentes. Existem três aspectos gerais que afetam vários tipos diferentes de software e que possivelmente podemos adotar soluções similares, entre eles estão Heterogeneidade; Mudança de Negócio/Social; e Segurança/Confiança [9].

Alguns fundamentos propostos por SOMMERVILLE [9] se aplicam a todos os tipos de sistemas, sendo esses fundamentos descritos a seguir. Softwares devem ser desenvolvidos em um processo gerenciado e compreendido, baseado em planejamento e conhecimento sobre o que será produzido; Confiança e desempenho são importantes para todos os tipos de sistema, sistemas devem ser seguros e protegidos contra ataques externos; Especificação e os requisitos do software devem ser entendidos, a equipe deve saber o que os clientes e/ou usuários esperam do sistema; e devemos fazer o melhor uso possível dos recursos existentes.

Existem atividades comuns para os diferentes tipos de sistemas, há quatro atividades que todo processo de software tem, sendo elas: engenharia de requisitos, modelagem de sistemas, teste de software e projeto de arquitetura compões as atividades relatadas [9] [5].

A engenharia de requisitos tem como principal objetivo entender os desejos dos clientes/usuários. Portanto, fornece mecanismo apropriado para tal objetivo, através da análise de necessidade, análise de viabilidade, negociação de soluções razoáveis, especificação da solução, validação da especificação e gerenciamento das necessidades. Os requisitos descrevem detalhadamente as necessidades de um cliente, serviços as serem ofertados e restrições de funcionamento. São necessários em diferentes níveis de detalhamentos, visto que, são utilizados por diferentes leitores para diversas funcionalidades. A utilização de processos é necessária pois auxilia as etapas descritas na engenharia de requisitos [9] [5]. Para PRESSMAN [5], a engenharia de requisitos abrange sete tarefas distintas que acontecem em paralelo, e podem ser adaptadas de acordo com as necessidades, sendo elas: concepção, levantamento, negociação, especificação, validação e gestão.

A modelagem de sistemas tem como principal objetivo criar representações do projeto como um todo, além de auxiliar diretamente no levantamento dos requisitos, modela representações daquilo que o cliente requer e todo seu fluxo de interações. No geral as representações utilizam de notações gráficas e normalmente baseada nas notações da UML, que propõe diversos diagramas que agregam em uma modelagem, habitualmente utilizamos os diagramas de atividades, caso de uso, sequência, classe e estado [9] [5]. Para BOOCH et al [10] os modelos abrangem planos detalhados e fornecem a base do projeto de um sistema, não são restritos a apenas grandes sistemas, porém quanto mais complexo o sistema, maior a importância de uma boa modelagem. SOMMERVILLE [9] fragmenta a modelagem em quatro tipos de modelos: modelos de contexto, modelos de interação, modelos estruturais e modelos comportamentais. Para cada tipo de modelo há diagramas da UML correspondentes, que auxiliam na representação.

O teste de software tem como objetivo principal verificar se um programa faz o que é proposto, se atende a seus requisitos funcionais e não funcionais. Os testes se resumem em um conjunto de atividades, que podem ser planejados com antecedência, executadas sistematicamente utilizando dados fictícios para a verificação da presença de erros. Vale ressaltar que os testes verificam a presença dos erros, e não sua ausência. Para os softwares convencionais a verificação é concluída se executada uma série de passos de testes, sendo eles: testes de unidade, teste de integração e teste de validação [9] [5].

O projeto de arquitetura se caracteriza por ser um conjunto de estruturas necessárias para a organização e estruturação de

um projeto de software. Tal estrutura depende das necessidades levantadas nos requisitos não funcionais para sua definição, visto que depende diretamente do requisito não funcional mais importante para sua escolha [9] [11]. Vale ressaltar que a arquitetura não garante a funcionalidade ou a qualidade de um projeto, apenas visa soluções para robustez, capacidade de distribuição e manutenibilidade [9] [5] [11]. Trata-se de um processo criativo que é dependente do tipo de sistema a ser desenvolvido, a formação e experiência do arquiteto e dos requisitos específicos para o sistema [9].

Uma arquitetura quando bem planejada torna seu software desacoplado, independente de qualquer tecnologia externa, permitindo que decisões, como escolher qual sistema de banco de dados será utilizado ou qual servidor web utilizar na implantação, sejam postergadas [6].

Dentre todos os tipos de arquitetura existentes: Arquitetura em camadas, Arquitetura de repositório, Arquitetura cliente-servidor e Arquitetura de duto e filtro [9]. Em seu livro, MARTIN [6] propõe uma arquitetura de sistema que tem como principal objetivo implementar as seguintes características: Independência de *framework*, Testabilidade, Independência da UI, Independência do banco de dados e Independência de qualquer agência externa.

Todos os conceitos descritos a seguir são propostos por MARTIN [6] em seu livro "Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software". Este capítulo em como objetivo referenciar a obra do autor que será a fundamentação do trabalho exposto neste trabalho. O principal objetivo da arquitetura limpa é separar as preocupações, ou seja, o software é dividido em camadas permitindo a separação de cada característica de um software acontecendo a inexistência de dependência externa no interno do software. As seguintes características estão presentes na arquitetura:

- Independência de *frameworks*: permite que o framework seja utilizado como ferramentas, pois não depende da existência de nenhuma biblioteca de software.
- Testabilidade: qualquer elemento externo não deve interferir nos testes da regra de negócio.
- Independência a UI: as regras de negócio não devem ser alteradas quando ocorrer uma mudança de UI.
- Independência do banco de dados: as regras de negócio não devem estar atreladas a um banco de dados específico, o que permite a alteração da tecnologia utilizada sem dificuldades.
- Independência de qualquer agência externa: a regra de negócio não conhece nada sobre o mundo externo.

As camadas na arquitetura limpa são representadas na Figura 2. A regra primordial para o funcionamento dessa arquitetura é ter como base a regra de dependência, que define

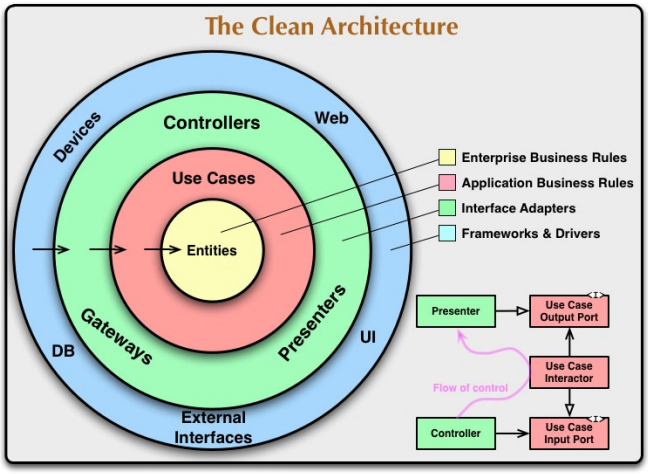

Fig. 2. Camadas Arquitetura Limpa [6]

a dependência como algo que só deve apontar para o interno, como mostra na Figura 2. Os elementos (funções, classes ou variáveis) de um círculo interno não podem saber nada sobre os elementos de um círculo externo. Do mesmo jeito, os formatos de dados declarados em um círculo externo não devem ser usados em um círculo interno.

A seguir são apresentadas as definições e responsabilidades de cada camada exposta na Figura 2:

- A camada de entidades (*entities*) reúne as regras de negócio da empresa inteira, as entidades podem ser um objeto com métodos ou um conjunto de estrutura de dados e funções. Tal camada permite o uso por diferentes aplicações da empresa, porém nenhuma mudança em qualquer aplicação deve influenciar a camada da entidade.
- A camada de caso de uso (*use cases*) reúne as regras de negócio específicas da aplicação, implementa todos os casos de uso do sistema, sendo esses os responsáveis por orquestrar o fluxo de dados. Apenas mudanças na operação da aplicação devem afetar os casos de uso, já que se trata de uma camada isolada, além de que nenhuma mudança realizada nesta camada deve afetar as entidades.
- A camada de adaptadores de interface (*interface adapters*) reúne um conjunto de adaptadores que convertem dados no formato utilizado pelos casos de uso e entidades para dados que serão utilizados por algum agente externo (base de dados, entre outros). Pertencem a essa camada os apresentadores, visualizações e controladores. Nenhum código desta camada deve saber algo sobre o agente externo. Vale ressaltar que esta camada também é responsável por converter os dados externos para dados que serão

utilizados pelos casos de uso e entidades.
- A camada de *frameworks* e *drives* reúne um conjunto de *frameworks* e ferramentas, no geral somente códigos de associação são criados nesta camada para estabelecer uma comunicação com o círculo interno.

A comunicação com uma camada externa é feita por meio de interfaces genéricas, que permitem uma interface interna trabalhar sem ao menos conhecer qual tecnologia está sendo utilizada, garantindo o baixo acoplamento nesta arquitetura. A arquitetura não está limitada apenas essas camadas, como todas as outras, pode ser adaptada de acordo com a necessidade do projeto de software que está sendo criado. Mas vale lembrar que a regra de dependência deve ser levada em consideração, pois é o princípio da arquitetura limpa proposta por MARTIN [6].

D. Trabalhos Relacionados

Após uma pesquisa na bibliografia trabalhos foram encontrados que implementam ferramentas com a mesma finalidade: gerar código automaticamente a partir de alguma representação, seja ela gráfica ou não. Abaixo alguns destaques dentre os trabalhos encontrados na bibliografia:

- **GERAÇÃO AUTOMÁTICA DE CÓDIGO A PARTIR DE PADRÕES DE PROJETO:** O artigo apresenta uma ferramenta proposta por BUDINSKY et al [12], tal ferramenta é capaz de gerar código a partir de informações fornecidas pelos usuários em uma plataforma web. As informações eram fornecidas para a ferramenta gerar código específico de um Padrão de Projeto informado pelo usuário. Um interpretador, desenvolvido pelos autores, foi implementado para realizar a "conversão" das informações descritas pelo usuário para o código solicitado. A ferramenta se assemelha com a metodologia que será utilizada nesse documento para a implementação. A diferença está apenas na complexidade do código gerando, enquanto o autor do artigo gera código de apenas um arquivo por requisição, iremos gerar uma grande massa de arquivos a partir de funcionalidades pré-moldadas.
- **GERAÇÃO DE CÓDIGO POR TRANSFORMAÇÃO DE MODELO:** O artigo apresenta um estudo de caso de uma ferramenta proposta por HEMEL et al [13], a ferramenta utiliza de uma linguagem específica de domínio criada pelos autores, a linguagem WebDSL para modelar aplicações web com um modelo de dados rico. Pouco se assemelha ao proposto neste documento, já que utiliza de um interpretador de alto nível, não criado pelo autor. A utilização de uma linguagem específica de domínio está próxima ao esperado como entrada na ideia relatada neste documento.
- **GERAÇÃO AUTOMÁTICA DE CÓDIGO DE UM MODELO UML PARA IEC 61131-3 E FERRAMENTAS DE CONFIGURAÇÃO DO SISTEMA:** O artigo apresenta uma abordagem capaz de gerar código IEC 611331-3 a partir de um modelo UML, tal abordagem foi proposta por VOGEL-HEUSER et al [14] em um artigo. O código gerado é importado automaticamente para o ambiente de programação PLC. A ferramenta pouco se assemelha a abordagem proposta neste documento, já que gera diretamente código de baixo nível, código de máquina e pouco se preocupa com a manutenibilidade deste código com futuras alterações.

III. Análise de requisitos da ferramenta

Algumas definições antecedem a criação da ferramenta desenvolvida neste trabalho. Para o desenvolvimento se faz necessário as seguintes definições: linguagem de programação adotada no desenvolvimento; ferramenta de documentação e modelagem e também o framework base do código gerado. Nesta seção serão abordados os passos e relatos para a definição de cada situação descrita.

A. Definição da Linguagem de Programação

Essencialmente uma linguagem de programação deve ser utilizada no desenvolvimento da ferramenta e de seu retorno (Código Gerado). No desenvolvimento do projeto, adotou-se o PHP como principal linguagem de programação. O PHP é uma linguagem de script de uso geral popular, interpretada no lado do servidor. Está entre as linguagens mais utilizadas na web atualmente, além de oferecer uma documentação rica e facilitada [15] [16]. A escolha da linguagem foi dada a partir de afinidade, no geral, não há limitações no desenvolvimento que independem a escolha de outra linguagem. A estrutura da ferramenta/interpretador será de fácil desenvolvimento e passível de implementação em diferentes linguagens, assim como a estrutura do código gerado, onde a Arquitetura Limpa pode ser facilmente construída em diferentes linguagens de programação.

B. Definição da Ferramenta de Documentação

Um estudo foi realizado para definirmos a ferramenta de documentação principal para o projeto desenvolvido neste trabalho. As mais utilizadas ferramentas de documentação foram estudadas e escolhemos a que mais se encaixa dentro das necessidades do projeto. Alguns parâmetros de seleção foram criados considerando a necessidade principal do projeto: gerar código a partir da documentação de uma API. Ou seja, a saída da documentação deve permitir sua leitura e garantir que as características principais do código sejam levantadas. As ferramentas estudadas foram Swagger, API Designer e a Restlet Studio. Escolhidas por afinidade, já que na concepção do autor são as ferramentas mais utilizadas entre as demais.

O Swagger possibilita diretamente a Modelagem da API e a Documentação, onde específica a Open API: uma linguagem para descrição de contratos de APIs REST que permite uma modelagem consistente através de uma linguagem "padrão" para descrição em diversas ferramentas [17]. Com o objetivo de facilitar a documentação e modelagem o Swagger permite sua criação a partir de uma linguagem – o formato YAML – que comporta as definições.

Tal ferramenta é comumente utilizada pela comunidade, visto que existem diversas linguagens de programações que oferecem pacotes, criados pelas comunidades, que ficam responsáveis por gerar uma documentação a partir de um código escrito. A partir do seu retorno (JSON exportado) será possível levantar todas as definições necessárias para o interpretador desenvolvido.

C. Definição do Framework para o Código Gerado

O código resultante da transformação da documentação necessita de uma estrutura base que sirva de facilitador para as comunicações básicas, validações, roteamento, acesso ao banco de dados, entre outras funcionalidades. Como abordado anteriormente, o código resultante será implementado baseado na Arquitetura Limpa, um framework que comporte tal implementação deve ser escolhido. Para isso, um estudo foi realizado entre os seguintes frameworks: Laravel, Lumen e o SlimPHP.

A escolha do Laravel está baseada em popularidade, o framework está entre os maiores e mais utilizados para o PHP [18]. Visto que o Laravel é um framework robusto que provê diversas soluções, a seleção dos demais frameworks tem como objetivo levantar variantes menos robustas, com isso selecionamos o Lumen e o SlimPHP. Lumen é uma variante do Laravel, dedicada para a criação de API e tem como principal característica ser uma versão simplificada. O SlimPHP tem como principal característica sua simplicidade e seu tamanho reduzido, permitindo maior customização das suas soluções. Maiores detalhes sobre os frameworks serão expostos ao decorrer do trabalho.

A aplicação do estudo prático se resume na Criação da Documentação na ferramenta escolhida e Implementação do Código resultante, onde o autor age no campo do interpretador que será desenvolvido a fim de validar e verificar o framework utilizado, além de analisar as limitações que a ferramenta impõe.

Um projeto de API foi idealizado para concluir o objetivo de escolher o framework ideal para gerarmos código. O projeto que será implementado nos frameworks escolhidos será: uma API que permite o usuário se cadastrar e se autenticar, e também manipular Tarefas.

Uma modelagem da API foi criada no Swagger, o que permite a criação do código a partir do seu retorno. A modelagem se inicia na escrita de um documento estruturado, o formato de entrada é o YAML e alguns atributos específicos são necessários para gerar a modelagem final. Tal modelagem resulta em uma documentação que permite a interação do leitor com cada item definido, onde as rotas possuem até uma propriedade que permite a execução de uma chamada a partir da própria documentação. Além de permitir a visualização de diversas outras configurações, como: retorno de cada rota criada, parâmetros necessários para aquela rota, erros gerados, se necessita ou não de uma autenticação.

Após a criação da documentação o estudo se iniciou baseado no JSON exportado, onde testes de implementação foram realizados nos frameworks citados. O resultado do estudo será exposto nos próximos parágrafos, vale ressaltar que o estudo não tem por objetivo inviabilizar a implementação de APIs com os frameworks citados, mas sim definir o framework que mais se encaixa para as necessidades do projeto.

Os testes se iniciaram com a criação de uma API utilizando o Laravel. O framework por si só mostra que é muito capaz, porém por se tratar de um framework muito robusto e que entrega diversas funcionalidades em uma estrutura que não possibilita muita customização torna-se incapaz para as necessidades do projeto. Sua arquitetura está totalmente baseada no Model-View-Controller e dificilmente podemos mudar isso, por isso o mesmo foi descartado.

Logo após, testes com o Lumen foram realizados, o framework trata-se de uma versão reduzida do Laravel que tem como principal objetivo servir para a criação de APIs. Lumen se comportou muito bem nos testes, em alguns pontos sua estrutura permite a customização e adequação aos princípios da arquitetura limpa e também a criação de toda a estrutura base da API modelada. Porém seu uso foi descartado ao constatar que sua estrutura e organização fere alguns princípios de dependências expostos pela arquitetura limpa, onde o framework (Lumen) deveria atuar como um agente externo e nos comportamentos dos testes havia uma enorme interferência dos controladores do framework na estrutura criada.

Por fim, foram realizados testes com o SlimPHP, entre os frameworks estudados este é menor de todos, sua estrutura é extremamente customizável e não há muitas dependências inicias, o desenvolvedor pode inserir o que ele deseja a estrutura do projeto para complementar no desenvolvimento, o que é um ponto positivo para o projeto. Se comportou muito bem trabalhando com os princípios da arquitetura limpa e sua estrutura está totalmente adequada, o framework age como um agente externo, além de prover alguns facilitadores externos para o código criado, que independe de qualquer abordagem utilizada.

O SlimPHP foi escolhido para se tornar a base do código, visto suas possibilidades e tamanho, foi o framework que mais

se adequou ao objetivo principal. Com o SlimPHP, diversos facilitadores são entregues para o código, o que amplia ainda mais as capacidades do resultado.

IV. A FERRAMENTA PARA GERAR CÓDIGO

Para o pleno funcionamento da ferramenta, se faz necessário o desenvolvimento de um interpretador que tem como principal responsabilidade processar o JSON de saída da documentação criada no Swagger e transformá-lo em código. O código resultante segue os princípios imposto pela arquitetura limpa, além das facilidades do SlimPHP, framework que compõe a base do código entregue. Nesta seção será abordado quais os passos necessários para o desenvolvimento do interpretador e consequentemente da ferramenta capaz de gerar código baseado na arquitetura limpa. A implementação funcional da ferramenta descrita neste trabalho está disponível no GitHub por meio do endereço: https://github.com/brunocbatista/code-generator-core.

A. Funcionamento da Ferramenta

O código implementado para agir como interpretador está inserido no mesmo projeto que serve como base para o código entregue, então executa alterações em uma estrutura previamente criada, com pastas, modelos de arquivos, configurações, entre outros detalhes necessários para o funcionamento.

A fim de executar a regra para gerar código o interpretador busca por prefixos no JSON inserido para transforma-los em código. Um estudo no JSON foi realizado para a definição das equivalências, por exemplo: um determinado prefixo, no contexto em que está inserido, leva a dizer que aquele objeto aborda a declaração de um modelo. A sintaxe e a estrutura do JSON serão responsáveis por determinar o que interpretador será capaz de construir no construir no código resultante. Considerando todos os pontos expostos sobre a arquitetura limpa, o estudo resultou nas seguintes premissas:

- Toda rota modelada na ferramenta se torna uma Ação/Caso de Uso;
- Toda definição informada por parâmetro na modelagem, pode se tornar uma regra de validação;
- Algumas das definições de objetos podem se tornar Modelos/Entidades;
- Rotas que foram modeladas com algum tipo de segurança só podem ser acessadas se o usuário estiver autenticado;
- Definições como “consumes” e “produces” – tipos de atributos que podem ser declarados na modelagem no Swagger – permitem a definição dos tipos de entrada e saída dos dados.

No atual cenário, o interpretador é dependente das possibilidades através do JSON gerado pelo Swagger, ou seja, a estrutura entregue pela ferramenta de modelagem é o principal limitador das possibilidades. Neste contexto, definimos as capacidades para o interpretador implementado: Gerar modelos; Gerar rotas; Gerar ações para Criar, Listar, Atualizar e Deletar um recurso; Gerar erros; e Gerar métodos de autenticação (autenticação e registro) para os usuários. Vale ressaltar que todas as capacidades e criações serão executadas em uma arquitetura limpa, facilitando sua implementação, já que a base do código será implementada.

Os métodos e seus resultados são limitados, ou seja, em alguns casos não possibilitam algumas customizações ou criação de uma regra específica, como por exemplo: não foram encontrados meios que possibilitam a declaração de lógicas customizadas para Ações executadas, então toda ação é identificada e criada a partir documentos genéricos. Regras específicas são as maiores limitações do interpretador implementado e da ferramenta de documentação escolhida, no geral há possibilidade de criar a base do código a partir da documentação, possibilidade na qual não fere o objetivo deste trabalho.

Em um contexto geral, os testes foram úteis para identificar as capacidades e incapacidades da ferramenta implementada e também para validar a implementação. Durante os testes erros foram solucionados e melhorias na lógica foram realizadas a fim de desenvolver uma ferramenta capaz e confiável. A verificação do objetivo se deu através da comparação entre a API desenvolvida manualmente pelo autor e a API que foi gerada pela ferramenta, ambos os códigos estão equivalentes e atendem as funcionalidades da modelagem criada via Swagger.

B. Aplicação Prática da Ferramenta

Esta seção aborda uma aplicação prática da ferramenta desenvolvida neste trabalho, a fim de verificar e validar o código gerado a partir de uma modelagem. A ferramenta tem como requisito mínimo de uso as seguintes tecnologias:

- PHP: o PHP é a principal linguagem de programação do projeto desenvolvido. A versão 7.4 foi utilizada para o desenvolvimento e testes do interpretador e também do código resultante.
- Composer: o Composer é um gerenciador de dependências específico para o PHP. Fica responsável por gerenciar as dependências necessárias do interpretador e do código resultante.
- Postman: software no qual serve de interface para a comunicação entre o usuário e a ferramenta desenvolvida. Responsável por emitir todas as requisições para o interpretador desenvolvido e também para a API gerada.
- Swagger: a ferramenta desenvolvida tem como dependência a ferramenta de modelagem, já que utiliza do arquivo gerado para executar suas funcionalidades.

Inicialmente uma documentação na plataforma Swagger deve ser criada. A princípio sua a documentação é criada a partir

de uma linguagem estruturada conhecida por YAML, através de atributos específicos a plataforma permite a criação de uma documentação visual e interativa, que posteriormente tem como saída um JSON estruturado que é utilizado pelo interpretador para gerar o código base de uma API.

O JSON resultante da modelagem é o principal documento processado pelo interpretador criado. A entrada para o interpretador é feita através de uma requisição feita pelo Postman. O JSON gerado deve ser enviado junto ao corpo da requisição para uma rota específica implementada do projeto do interpretador.

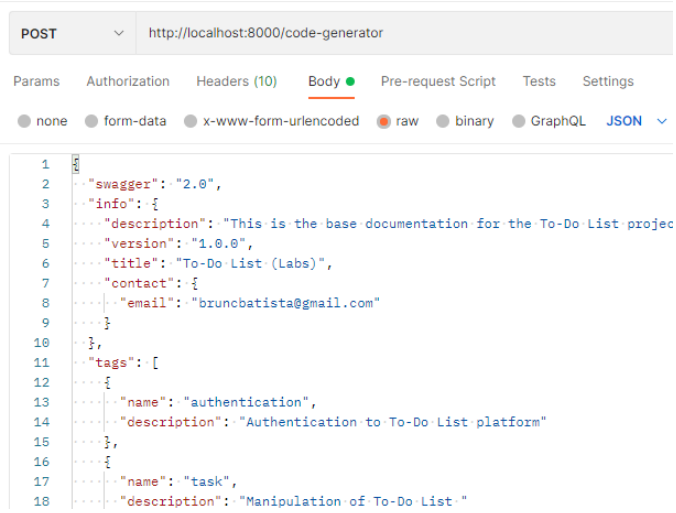

Fig. 3. Requisição para o interpretador

A fim de demonstrar o funcionamento do interpretador, utilizaremos o exemplo já abordado neste trabalho, em síntese a API resultante irá atender as necessidades de uma modelagem que deve possuir rotas de autenticação e cadastrado e também a manipulação de tarefas.

Como demonstra a Figura 3, o JSON resultante deve ser passado para o interpretador na integra, o interpretador será responsável por manipular e filtrar a estrutura da saída da ferramenta de modelagem. Os filtros e manipulações acontecem através da busca por atributos identificadores para a estrutura de entrada, como demonstrado em maiores detalhes na seção 4. Ao efetuar a requisição para a rota demonstrada na Figura 3, o usuário deve aguardar pelo retorno da validação e manipulação realizada pelo interpretador, que identifica os pontos e cria as estruturas identificadas.

Ao obter retorno do interpretador o usuário deve visualizar a estrutura base do projeto e analisar quais foram os arquivos e soluções geradas a partir da documentação de entrada. Caso ocorrer erro no processo de interpretação, não inviabiliza a criação de algumas soluções, o interpretador gera código até onde o erro foi ocasionado.

O interpretador manipula uma estrutura base que atendente as especificações da Arquitetura Limpa, tal estrutura foi criada previamente, a fim de otimizar o processamento do interpretador, que atribuí aos modelos criados especificações levantadas a partir do arquivo de entrada. Abaixo, algumas imagens retratam a estrutura base criada pelo interpretador a partir de uma modelagem realizada no Swagger.

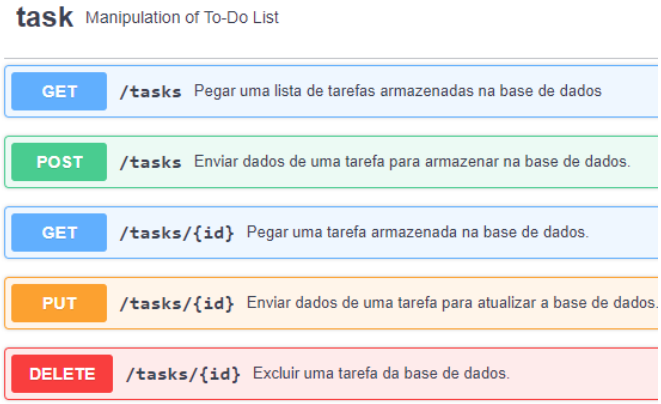

Fig. 4. Endpoints criados pelo interpretador

A Figura 4 tem como objetivo expor a equivalência entre a modelagem criada pelo usuário e o código gerado pelo interpretador. Cada Ação modelada através das Rotas no arquivo gerado pelo Swagger torna-se uma Ação para código gerado, tais ações são inseridas na estrutura base do projeto e ao final do processamento está disponível para o usuário manipular conforme sua necessidade.

Vale ressaltar que o código gerado para cada ação mapeada, trata-se de uma estrutura genérica, para fins de complementar a base do projeto, gerando os arquivos bases – e funcionais – para cada rota. Cumprindo as especificações impostas pela Arquitetura Limpa, deste modo facilitando ainda mais o desenvolvimento, já que a implementação da Arquitetura Limpa é custosa do ponto de vista do desenvolvimento, mas sua implementação torna o código de fácil manutenção, permitindo futuras modificações a estrutura criada.

O interpretador ainda não se limita apenas a criação das Ações, também é possível criar e manipular rotas, modelos, funcionalidades que servem como facilitadores, exceções, entre outras coisas que estão expostas na Figura 5, os arquivos gerados pelo interpretador estão destacados em vermelhos – alguns arquivos não são apresentados na imagens pois estão inseridos em subpastas que complicam a apresentação –, para melhorar a

visualização, os arquivos que estão sem destaques são padrões e pertencem a estrutura base desenvolvida previamente para facilitar a manipulação realizada pelo interpretador na estrutura do código resultante. Posteriormente o usuário pode customizar o código resultante de acordo com suas necessidades específicas sem grandes problemas.

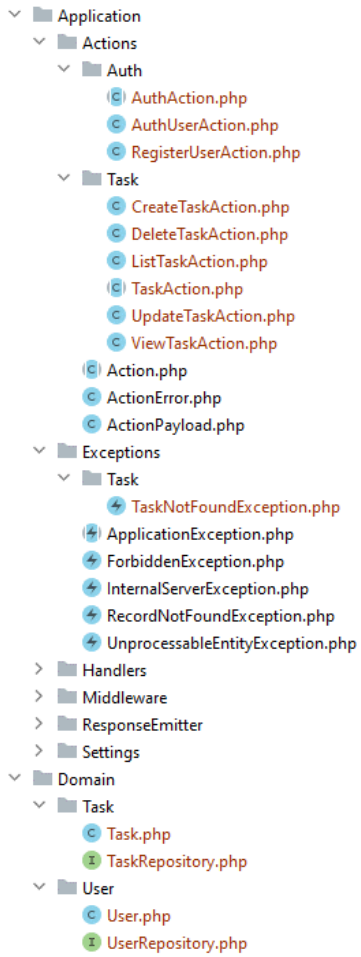

Fig. 5. Arquivos criados pelo interpretador (destacados em vermelho)

V. Conclusão

Este trabalho apresentou o desenvolvimento de uma ferramenta capaz de gerar código base para um API REST baseada nos princípios da Arquitetura Limpa. O código gerado tem como base o framework SlimPHP, o qual provê soluções e facilitadores para o funcionamento do código. A ferramenta tem como entrada o JSON de saída de uma modelagem criada na ferramenta Swagger, o interpretador desenvolvido fica responsável por processar a entrada e identificar pontos de equivalência para os arquivos de Modelos, Ações, entre outros que são gerados pela ferramenta. O interpretador manipula um código base já implementado, os arquivos são criados a partir de modelos, definidos previamente e adaptados de acordo com os atributos da entrada da documentação. A ferramenta possui limitações, no estado atual da ferramenta não há condição para processar funcionalidades ou regras customizadas, para as ações geradas pelo interpretador funcionamento genérico é entregue a partir do contexto em que a rota foi modelada – em específico baseado no método HTTP que a rota atende. Como trabalho futuro pretende-se o aprimoramento da ferramenta desenvolvida neste trabalho. Como aprimoramento entende-se a solução das limitações expostas e implementação de uma interface gráfica para a entrada dos dados. A solução das limitações está atrelada diretamente a criação de funcionalidades ou regras semânticas. Alguns meios serão analisados e estudados para permitir a personalização do comportamento via modelagem realizada no Swagger e tal ampliação visa contribuir para o aprimoramento do interpretador implementado, que possivelmente será capaz de gerar ações comportamentais personalizadas.

Referências

[1] D. e. a. JACOBSON, *APIs: A Strategy Guide*. O'Reilly Media Inc, 2012.
[2] B. e. a. JIN, *Designing Web APIs: BUILDING APIS THAT DEVELOPERS LOVE*. O'Reilly Media Inc, 2018.
[3] S. PATNI, *Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS*. Apress, 2017.
[4] M. MASSÉ, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media Inc, 2011.
[5] R. S. PRESSMAN, *Engenharia de Software: Uma Abordagem Profissional*. AMGH Editora Ltda, 2011.
[6] R. C. MARTIN, *Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software*. Alta Books, 2019.
[7] S. Bhartiya. (2022, sep) How netflix built spinnaker, a high-velocity continuous delivery platform. [Online]. Available: https://thenewstack.io/netflix-built-spinnaker-high-velocity-continuous-delivery-platform/
[8] A. LAURET, *The Design of Web APIs*. Manning Publications Co, 2019.
[9] I. SOMMERVILLE, *Engenharia de Software*. PEARSON, 2011.
[10] G. e. a. BOOCH, *UML: Guia do Usuário*. Elsevier Editora Ltda, 2012.
[11] L. e. a. BASS, *Software Architecture in Practice*. PEARSON, 2013.
[12] F. J. e. a. BUDINSKY, "Automatic code generation from design patterns," 1996.
[13] Z. HEMEL, *Code Generation by Model Transformation: A Case Study in Transformation Modularity*. ICMT, 2008.
[14] V.-H. et al, *Automatic Code Generation from a UML model to IEC 61131-3 and system configuration tools*. ICCA, 2005.
[15] S. TIOBE, "Tiobe, the software quality company," TIOBE, Tech. Rep., Setembro 2021. [Online]. Available: https://www.tiobe.com/tiobe-index/
[16] R. SEBESTA, *Conceitos de Linguagens de Programação*. Bookman, 2011.
[17] S. SCHWICHTENBERG, *From Open API to Semantic Specifications and Code Adapters*. IEEE 24th International Conference on Web Services, 2017.
[18] L. LAZZARIN, *Uma Visão Geral de Frameworks PHP Populares para Programação Web*. Ciências Exatas e da Terra e a Dimensão Adquirida através da Evolução Tecnológica, 2019.