UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DAVI BARBOSA SILVA SOUSA

GERAÇÃO DE CLIENTES PARA COMUNICAÇÃO ASSÍNCRONA COM BASE NA ESPECIFICAÇÃO ASYNCAPI

CAMPINA GRANDE - PB

2023

DAVI BARBOSA SILVA SOUSA

GERAÇÃO DE CLIENTES PARA COMUNICAÇÃO ASSÍNCRONA COM BASE NA ESPECIFICAÇÃO ASYNCAPI

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Adalberto Cajueiro de Farias

CAMPINA GRANDE - PB

2023

DAVI BARBOSA SILVA SOUSA

GERAÇÃO DE CLIENTES PARA COMUNICAÇÃO ASSÍNCRONA COM BASE NA ESPECIFICAÇÃO ASYNCAPI

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Adalberto Cajueiro de Farias
Orientador – UASC/CEEI/UFCG

Maxwell Guimarães De Oliveira
Examinador – UASC/CEEI/UFCG

Melina Mongiovi Cunha Lima Sabino
Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 17 de Novembro de 2023.

CAMPINA GRANDE - PB

RESUMO

O desenvolvimento de aplicações que interagem entre si através de microsserviços pode ser feito de forma síncrona e assíncrona. Este trabalho explora a arquitetura orientada a eventos (Event-Driven Architecture) que possui uma especificação para documentação de rotas chamada AsyncAPI, com ela é possível gerar APIs assíncronas partindo de templates de geração de código. No entanto, os geradores de código existentes são construídos com a finalidade de produzir código publicador-consumidor para determinados conjuntos de tecnologias e ainda há uma lacuna de ferramental para dispositivos embarcados e aplicações Web. O objetivo deste trabalho é construir uma ferramenta para as linguagens C++ e Typescript que auxilia os usuários a integrar APIs assíncronas em suas aplicações a partir de uma especificação AsyncAPI. Desta forma, a solução desenvolvida baseia-se em desenvolvimento dirigido por modelos através de uma ferramenta que gera código para publicação e consumo de mensagens gerenciadas por um Message Broker de forma mais dinâmica e parametrizável. Nesse sentido, o desenvolvimento de sistemas assíncrono tem foco principal na especificação (modelo) e na injeção de aspectos de negócio no código gerado pela tradução automática. Isso promove a redução do esforço e tempo no desenvolvimento de sistemas assíncronos.

CLIENT GENERATION FOR ASYNCHRONOUS COMMUNICATION BASED ON THE ASYNCAPI SPECIFICATION

ABSTRACT

The development of applications that interact with each other through microservices can be done synchronously or asynchronously. This work explores the Event-Driven Architecture, which has a specification for documenting routes called AsyncAPI, with which it is possible to generate asynchronous APIs from code generation templates. However, the existing code generators are built with the aim of producing publisher-consumer code for certain sets of technologies and there is still a gap in tooling for embedded devices and web applications. The aim of this work is to build a tool for the C++ and Typescript languages that helps users integrate asynchronous APIs into their applications based on an AsyncAPI specification. In this way, the solution developed is based on model-driven development through a tool that generates code for publishing and consuming messages managed by a Message Broker in a more dynamic and parameterizable way. In this sense, the development of asynchronous systems focuses mainly on the specification (model) and the injection of business aspects into the code generated by automatic translation. This reduces the effort and time involved in developing asynchronous systems.

GERAÇÃO DE CLIENTES PARA COMUNICAÇÃO ASSÍNCRONA COM BASE NA ESPECIFICAÇÃO ASYNCAPI

Davi Barbosa Silva Sousa
Federal University of Campina Grande
Campina Grande, Paraíba, Brasil
davi.sousa@ccc.ufcg.edu.br

Adalberto Cajueiro de Farias
Federal University of Campina Grande
Campina Grande, Paraíba, Brasil
adalberto@computacao.ufcg.edu.br

RESUMO

O desenvolvimento de aplicações que interagem entre si através de microsserviços pode ser feito de forma síncrona e assíncrona. Este trabalho explora a arquitetura orientada a eventos (*Event-Driven Architecture*) que possui uma especificação para documentação de rotas chamada *AsyncAPI*, com ela é possível gerar *APIs* assíncronas partindo de templates de geração de código. No entanto, os geradores de código existentes são construídos com a finalidade de produzir código publicador-consumidor para determinados conjuntos de tecnologias e ainda há uma lacuna de ferramental para dispositivos embarcados e aplicações *Web*. O objetivo deste trabalho é construir uma ferramenta para as linguagens *C++* e *Typescript* que auxilia os usuários a integrar *APIs* assíncronas em suas aplicações a partir de uma especificação *AsyncAPI*. Desta forma, a solução desenvolvida baseia-se em desenvolvimento dirigido por modelos através de uma ferramenta que gera código para publicação e consumo de mensagens gerenciadas por um *Message Broker* de forma mais dinâmica e parametrizável. Nesse sentido, o desenvolvimento de sistemas assíncrono tem foco principal na especificação (modelo) e na injeção de aspectos de negócio no código gerado pela tradução automática. Isso promove a redução do esforço e tempo no desenvolvimento de sistemas assíncronos.

Keywords

Geração de código, *AsyncAPI*, *Message Broker*, Text-Templates.

1. INTRODUÇÃO

O desenvolvimento de APIs assíncronas tem se tornado cada vez mais popular, com isso os aplicativos desenvolvidos nessa arquitetura tem se tornado mais complexos. Para facilitar a integração entre microsserviços que usam essas *APIs* foi criada uma especificação para documentar canais e mensagens utilizados para comunicação chamada de *AsyncAPI* [1]. Esta especificação é escrita em *JSON* ou *YAML* e possui informações de canais e objetos de um *Message Broker*.

A partir da especificação *AsyncAPI* outras ferramentas foram criadas para apoiar o desenvolvimento de aplicações assíncronas, dentre elas está o *AsyncAPI Generator* [2] que tem a capacidade de gerar código a partir de uma especificação *AsyncAPI* utilizando a ajuda de templates de código [7]. Porém, até a data de publicação deste trabalho não, este gerador de código baseado em templates não possui implementações para as linguagens *Typescript* e *C++*, como observado na listagem de repositórios públicos [3]. Há uma lacuna ferramental para estas linguagens, que são amplamente utilizadas por aplicações da *Web* e dispositivos embarcados com suporte a linguagem *C++*.

Além disto, a maioria dos templates listados contemplam a criação de código partindo de uma visão de projeto, não de módulos. Nesse sentido, a ferramenta desenvolvida neste trabalho gera um projeto com todos arquivos necessários para compilação e execução. Embora essa abordagem represente inicialmente alguma incompatibilidade para sistemas já criados, ela provê uma camada de comunicação sobre a qual qualquer integração pode ser implementada, reduzindo, assim, o acoplamento sem alterar a lógica do sistema.

Tendo em vista este problema, torna-se um desafio configurar a ferramenta de geração de código citada para produzir códigos facilmente acoplados a qualquer sistema *Web* ou sistema embarcado. Para sistemas grandes que precisam se integrar a diversos microsserviços assíncronos a geração automática de código a partir de uma especificação torna-se uma ferramenta auxiliar no desenvolvimento, aumentando a produtividade dos desenvolvedores e a qualidade do código através da redução de erros [8]. Além disso, a especificação pode sofrer alterações com a mudança de versões da aplicação. Desta forma surge um problema para manter o código sincronizado com a especificação.

Este trabalho propõe uma ferramenta de geração automática de clientes assíncronos usando a especificação *AsyncAPI*. Os clientes são gerados nas linguagens *C++* e *Typescript* para o *framework Angular*. Ambos os clientes utilizam o protocolo *MQTT* para comunicação assíncrona. A ideia desta ferramenta surgiu a partir da cooperação com a Universidade de Sevilha que busca a criação de servidores assíncronos para braços robóticos.

2. TRABALHOS RELACIONADOS

A escrita de geradores de código não é uma tarefa trivial, e de acordo com Jeroen Arnoldus [7] há diversas abordagens para escrita de geradores, entre elas estão os geradores de código homogêneos, onde a metalinguagem de geração e a linguagem alvo são a mesma, e os geradores heterogêneos,

em que a linguagem alvo e a metalinguagem são diferentes. Os geradores heterogêneos se classificam entre Árvores Sintáticas Abstratas, Declarações Impressas, Reescrita de Termos e Templates-Texto. Este trabalho irá focar no desenvolvimento de geradores usando a abordagem Template-Texto.

Os geradores de código baseados em *templates* podem ser utilizados com uma especificação de entrada e gerar código para qualquer linguagem, desde que o *template* seja escrito para isto. Como exemplos temos o *OpenAPI Generator* [10], que recebe uma especificação *OpenAPI* e produz o código desenvolvido no *template*, estes templates são desenvolvidos usando a linguagem de *template Mustache*. Além deste, temos o *AsyncAPI Generator* [2], que também se baseia em templates, mas a entrada é uma especificação *AsyncAPI* e a saída é gerada através de templates com os motores de template *Nunjucks* e *React*.

Geradores de código também podem ser criados e usar as ferramentas disponibilizadas pelas linguagens ou motores de renderização para produzir código. Este artigo [8] mostra o processo de criação de um novo gerador de código utilizando as linguagens de *template Handlebars* e a *Text Template Transformation Toolkit*, conhecida como T4. Para este trabalho, visto que já existe um gerador de código para *AsyncAPI* não será necessário criar um gerador de código, apenas os *templates* que serão utilizados em conjunto com o gerador.

3. METODOLOGIA

Esta seção explicita as atividades realizadas para a execução deste trabalho.

3.1 Estudo da especificação AsyncAPI

Para compreender o nível de descrição da especificação *AsyncAPI [4]* foi necessário realizar um estudo que investigou todos os atributos disponíveis pela especificação e como se relacionavam entre si. A partir deste estudo pôde-se observar que os atributos mais importantes para a geração de código são *Servers, Channels* e *Messages*. Os *Servers* definem qual protocolo de comunicação será utilizado e qual endereço do *message broker*. Os *Channels* definem quais operações são realizadas, *subscribe* ou *publish*. Já as *Messages* definem o corpo das operações definidas em *Channels,* especificando os *Schemas* que serão enviados ou recebidos por essas operações e suas relações de composição e herança. Estes atributos foram a base para o desenvolvimento dos *templates*.

3.2 Estudo comparativo entre geradores de código

Esta etapa do trabalho define a abordagem de geração de código. De acordo com Jeroen Arnoldus [7] há quatro abordagens para geração de código heterogêneas são elas: Árvore Sintática Abstrata, Declarações impressas, Reescrita de termos e Template-Texto. Para este trabalho as abordagens escolhidas para comparação são: Declarações impressas ou Template-Texto com *AsyncAPI Generator,* variando os motores de renderização (*Nunjucks* ou *React*).

Os seguintes aspectos foram considerados para comparação das abordagens:

- Legibilidade do código do template. Facilidade em ler os fragmentos de código presentes na abordagem.
- Possibilidade de depurar a geração de código.
- Possibilidade de adicionar testes unitários.
- Componentização de blocos de código do template. Facilidade em desacoplar blocos de código.
- Capacidade e facilidade de manipulação dos objetos da AsyncAPI, ex: Iterar sobre uma lista de canais, mapear e filtrar objetos, gerar arquivos condicionalmente, entre outros.
- Exemplos de templates disponíveis publicamente que geram APIs assíncronas.

Abordagem de geração de código	Print Statements (com AsyncAPI Parser)	Template-Texto (com AsyncAPI Generator e Nunjucks)	Template-Texto (com AsyncAPI Generator e React)
Legibilidade	Baixa	Média	Alta
Debug	Sim	Não	Sim
Testes	Sim	Não	Sim
Componentização	Sim	Sim (com *partials*)	Sim
Capacidade de manipulação de código	Alta	Baixa (limitada à própria *template engine*)	Alta
Exemplos disponíveis	Não	Sim, vários	Sim, poucos

Tabela 1. Comparativo entre as abordagens de geração de código

Através desta tabela comparativa podemos perceber que a abordagem de geração de código utilizando do *AsyncAPI Generator* com o motor de renderização *React* é a mais apropriada para este trabalho.

3.3 Exploração dos templates de código do AsyncAPI Generator

Nesta etapa um estudo exploratório foi conduzido para investigar como os templates oficiais listados pelo *AsyncAPI Generator* produziam código. Questões como estrutura de arquivos, configurações de pacote, motor de renderização, código gerado, protocolos utilizados e bibliotecas foram levados em consideração. A partir deste estudo exploratório notou-se que todos os templates seguem uma mesma estrutura de arquivos, composta por um arquivo de configuração *package.json*, um diretório *template* e opcionalmente diretórios de *hooks* e *filters*. Além disso, pode-se perceber que a maioria dos *templates* foram criados com o motor de renderização *Nunjucks*. Há poucos exemplos de templates utilizando o motor de renderização *React*. Outro resultado relevante deste estudo foi a descoberta da ferramenta *Modelina* [5], que permite a criação de classes a partir dos *Schemas* encontrados na especificação *AsyncAPI,* com esta ferramenta é possível acelerar o desenvolvimento dos templates, já que a

responsabilidade de criar os arquivos de modelos utilizados no código é da biblioteca.

3.4 Desenvolvimento da ferramenta de geração de código

Esta etapa foi utilizada para o desenvolvimento da infraestrutura necessária para a geração de código. Para gerar código na linguagem *C++* foi construído um *template* tomando como base os *templates* oficiais. Da mesma maneira foi criado um *template* para linguagem *Typescript*, especificamente para o *framework Angular*. O objetivo destes *templates* é servir de entrada para o gerador que, juntamente com uma especificação *AsyncAPI* e um conjunto de parâmetros customizáveis, irá gerar o código estabelecido pelo *template*, na Figura 1 podemos ver o fluxo do gerador do AsyncAPI.

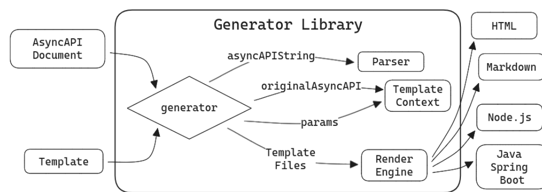

Figura 1. Processo de geração de código pela ferramenta AsyncAPI Generator

Para facilitar a utilização dos templates construídos foi necessário construir uma interface gráfica que permitisse ao usuário escrever ou abrir um arquivo com a especificação *AsyncAPI* e descarregar o código gerado através desta interface. Esta interface gráfica se conecta com um servidor que utiliza o gerador do *AsyncAPI* e os *templates* criados para gerar código.

3.5 Teste da ferramenta

Para testar a ferramenta foi utilizado uma especificação AsyncAPI de teste que possui todos os atributos necessários para criação completa de um cliente. Esta especificação foi construída com a cooperação da Universidade de Sevilha para construção de um servidor assíncrono para braços robóticos, os atributos presentes na especificação pertencem ao contexto de controle e comunicação entre os braços robóticos e o *message broker*. Os atributos presentes na especificação são informações sobre servidores, canais e mensagens. A partir destes foi possível criar um cliente compilável e executável para *C++* e *Angular*.

Para testar o código criado foi necessário utilizar o *message broker Mosquitto* [6]. A partir deste broker foi possível publicar mensagens em canais presentes na especificação e observar o tratamento da operação "*subscribe*", bem como publicar mensagens para o *broker*.

4. Solução

Nesta seção são apresentadas as etapas de desenvolvimento das ferramentas necessárias para geração de código. A Figura 2 apresenta como estão dispostos os microsserviços desenvolvidos.

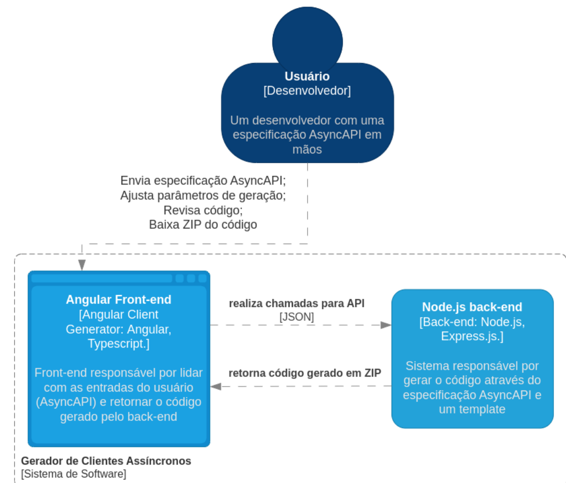

Figura 2. Diagrama de Sistema da aplicação

4.1 Templates de código

Nesta subseção são detalhados os *templates* produzidos neste trabalho.

Estes projetos são responsáveis por fornecer o modelo de saída para a ferramenta de geração de código. O template é um projeto construído em *Node.js* que não está acoplado ao *AsyncAPI Generator*. Neste projeto os componentes necessários para que o template seja reconhecido pelo gerador de código são:

- arquivo *package.json,* que define os protocolos suportados, os parâmetros utilizados e o motor de renderização do *template*. Além das bibliotecas necessárias para executar as funcionalidades do template.
- pasta *template*, que fornece os arquivos que serão renderizados pelo gerador de código.
- pasta *hooks*, que define ações que serão executadas antes ou após a geração do código. Esta função é útil para executar ações antes e depois da geração.

4.1.1 Template para C++

Repositório deste template: https://github.com/davibss-tcc/asyncapi-cpp-template.

O template de *C++* fornece um modelo que irá gerar um projeto em *C++* compilável e executável. O código gerado irá se comunicar com um broker *Mosquitto* através de publicação e subscrição.

No template, são gerados os arquivos fonte dentro do diretório *src*, estes arquivos que serão responsáveis por abrir e manter a conexão com o *broker*, através de arquivos de serviço. Estes serviços usam os *schemas* gerados como structs para trocar informações. Este diretório *src* será o módulo desacoplado que poderá ser adicionado em qualquer aplicação.

Para gerar o arquivo que irá abrir a conexão com o *broker* e os serviços associados foi utilizado o modelo de geração através do motor de renderização do *React*. Já para gerar os *structs* necessários para troca de informações foi utilizada a biblioteca *Modelina*. Esta biblioteca consegue gerar classes de objetos para diversas linguagens a partir da especificação *AsyncAPI*, inclusive para *C++*, e esta abordagem foi usada para gerar os *structs* para *C++*. Devido à limitação de geração de *structs* pela biblioteca foi

necessário adicionar um serializador de *JSON* com o apoio da biblioteca *nlohmann/json*. A biblioteca *Modelina* permite fácil customização das classes geradas através *presets* e parâmetros adicionais no gerador de *C++*. Os atributos gerados por padrão eram gerados com o *template* de classe *optional* e para simplificação este *template* foi removido.

Os serviços foram gerados a partir dos canais disponíveis na especificação *AsyncAPI*. Para cada canal foi gerada uma função que se sobrescreve nele. Após receber a mensagem a função desserializa a mensagem para o *struct* do *payload* correspondente identificado na operação de subscrição do canal.

Além dos arquivos centrais da aplicação gerada também foram gerados arquivos de suporte à compilação inicial, como o *CMakeList.txt*, *mosquitto_build.sh* e *mosquitto.conf*. Estes arquivos são responsáveis por baixar todas as dependências necessárias, compilar e executar o cliente. Também foi gerado um *README.md* que documenta todo o processo necessário para compilar e executar o cliente gerado.

Após a geração completa do código, o hook *generate:After* irá formatar todos os arquivos com extensão *.cpp* e *.hpp* através do formatador *clang-format*. Além disso, caso o parâmetro *zip* passado no gerador tiver o valor "true", todo o código gerado será compactado e colocado na saída da geração.

Para ajudar o desenvolvimento do template foi utilizada a *API* de documentação *JSDoc* em alguns pontos do código para definir tipos de parâmetros e retorno de funções, além de disponibilizar a documentação dos objetos utilizados na biblioteca *AsyncAPI Parser*, que são utilizados diretamente pelo gerador da *AsyncAPI*.

4.1.2 Template para Angular

Repositório deste template: https://github.com/davibss-tcc/asyncapi-angular-template.

O template de *Angular* fornece um modelo que irá gerar uma aplicação em Angular compilável e executável. O código gerado irá se comunicar com um *broker Mosquitto* através de publicação e subscrição. Este template assim como o anterior também usa o motor de renderização do React.

No template, são gerados arquivos fonte dentro do diretório *client*, estes arquivos são responsáveis por iniciar serviços injetáveis do *Angular* que estabelecem comunicação com o *broker*. Este diretório *client* será o módulo desacoplado que poderá ser adicionado em qualquer aplicação *Angular*. Dentre esses arquivos estão serviços que estendem de uma classe de serviço base que se inscreve e publica em tópicos. Cada implementação desta classe base irá se sobrescrever o tópico padrão e definir o objeto que será trafegado entre o cliente e o broker. Os tópicos são os nomes dos canais definidos na especificação *AsyncAPI*.

O arquivo *client_implementation.ts* é responsável por se inscrever em todos os tópicos e construir uma função *callback* que redireciona a mensagem recebida para a próxima função.

As classes utilizadas pela aplicação foram construídas com a biblioteca *Modelina*, estas classes se baseiam nos *schemas* definidos na especificação. A classe gerada contém todos os atributos definidos na especificação além de funções auxiliares como *getters* e *setters*. Para a geração de classes na linguagem *Typescript* há a possibilidade de incluir serialização automática, porém devido a limitações na inclusão de atributos extras nestes foi necessário adicionar manualmente estes serializadores. Assim como no template de *C++* foi necessário utilizar a abordagem de *presets* da *Modelina* para customizar a geração base das classes em *Typescript*. Além disso, foi necessário sobrescrever a geração de listas da biblioteca, para que o tipo da lista seja único, diferente da abordagem anterior em que o tipo de lista não era definido.

Antes da geração dos arquivos centrais do cliente, o *hook generate:before* foi executado para que um projeto *Angular* fosse criado, através de comandos feitos com a biblioteca *@angular/cli*. Após a geração foi adicionada a biblioteca *ngx-mqtt* como dependência da aplicação.

Após a geração dos arquivos fonte do cliente, o código foi formatado através da biblioteca *Prettier*. Além disso, caso o parâmetro *zip* passado no gerador tenha sido "true", o código gerado será compactado e colocado na saída da geração.

O código gerado em *Angular* fornecerá uma interface básica para publicação e subscrição de mensagens, onde o usuário poderá escolher o *schema* que será enviado ou recebido, como mostra na Figura 3.

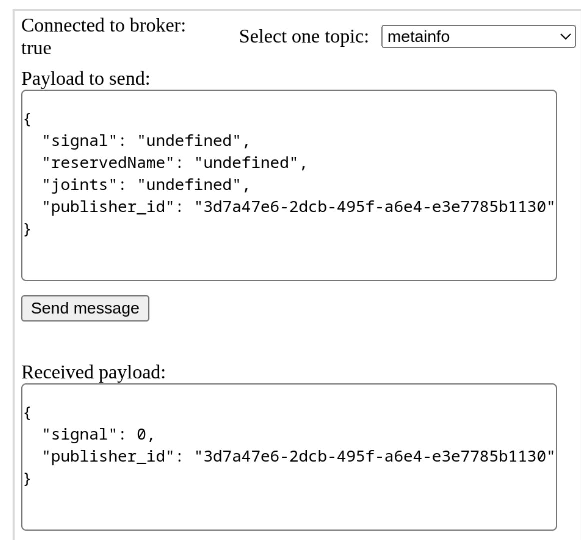

Figura 3. Aplicação Angular gerada pelo gerador através do template

4.2 Back-end

Repositório desta aplicação: https://github.com/davibss/asyncapi-client-generator/tree/main/client-generator-backend.

O backend é uma aplicação *REST* em *Node.js* utilizando o *framework Express.js*. O objetivo desta aplicação é prover um meio para interface gráfica utilizar o *AsyncAPI Generator* através de chamadas *HTTP*, visto que não é possível utilizar o gerador de código no *browser* do usuário devido às chamadas para módulos de gerenciamento de arquivos e diretórios como *path* e *fs*.

Para gerar código através de templates esta aplicação possui como dependências as bibliotecas do *AsyncAPI Generator* e os templates em *C++* e *Angular* produzidos anteriormente. As funcionalidades desta aplicação são gerar código, através do método *POST,* e obter o arquivo

compactado em *ZIP* do cliente gerado através do método *GET*.

A rota para gerar código necessita de uma especificação *AsyncAPI* como corpo da requisição e dois parâmetros adicionais, o *template* e o *params*. O parâmetro *template* define qual template será chamado para geração de código, os valores aceitos são *CPP* para o template em C++ e *ANGULAR* para o template em *Angular*. Já o parâmetros *params* define quais parâmetros devem ser passados durante a geração do código, o formato deste é a serialização de um objeto contendo chave e valor dos parâmetros disponíveis para uso dos templates. Após a geração do código é gerado um *hash* que identifica unicamente em qual pasta está o arquivo compactado daquele usuário, este *hash* é retornado para o usuário no corpo da resposta.

A rota para obter o *ZIP* necessita do *hash* gerado pela requisição de geração. Nesta rota o *backend* irá procurar por arquivos no formato *ZIP* dentro da pasta identificada com o *hash* e irá retornar esse arquivo para o usuário.

4.3 Front-end

Repositório desta aplicação: https://github.com/davibss/asyncapi-client-generator/tree/main/client-generator-angular.

O *frontend* é uma aplicação *Single Page Application* (SPA) construída em Angular. Esta aplicação se comunica com o *backend* para gerar o código do cliente assíncrono.

A aplicação tem três funcionalidades principais: Inserção da especificação *AsyncAPI*, configuração do gerador de código e revisão de código.

Na etapa de inserção da especificação *AsyncAPI* o usuário pode carregar o arquivo da especificação ou copiar o texto para o editor, como mostra na Figura 4.

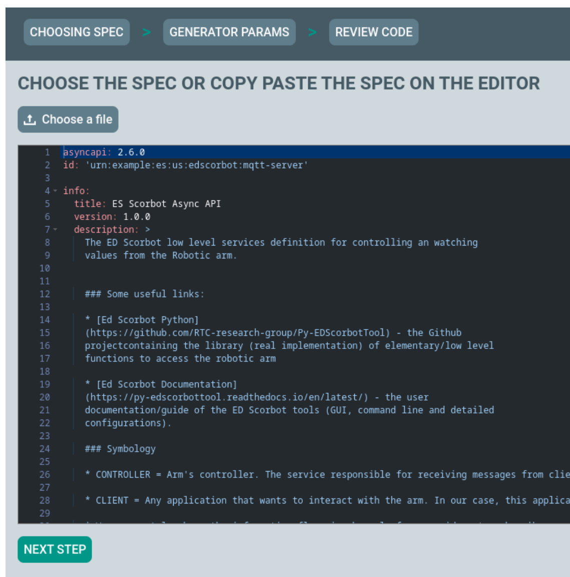

Figura 4. Página para inserção da especificação AsyncAPI

Na etapa de configuração do código gerado o usuário pode escolher em qual tecnologia o cliente será gerado. Além disso, o usuário poderá configurar parâmetros disponíveis pelos templates. Em ambos os *templates* o usuário pode escolher gerar somente os arquivos fonte ou gerar a aplicação inteira.

A Figura 5 mostra a página de configuração de geração de código para o template *C++*. A Figura 6 mostra a configuração para o template *Angular*.

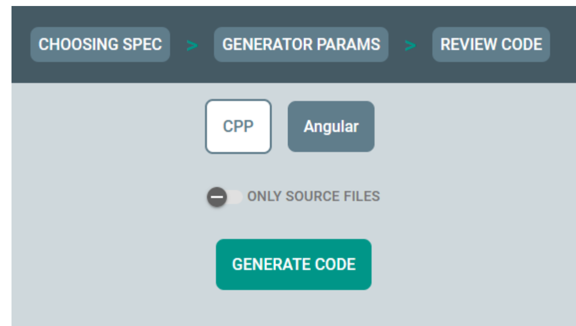

Figura 5. Página para configuração do gerador com o template C++

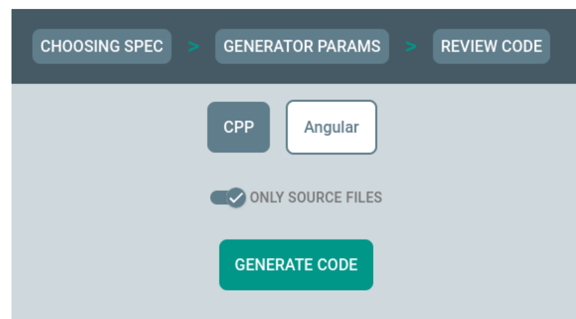

Figura 6. Página para configuração do gerador com o template Angular

Na última etapa o usuário poderá visualizar todos os arquivos gerados pelo *backend* e baixá-los no formato *ZIP*. A Figura 7 mostra um exemplo desta etapa utilizando a especificação de exemplo.

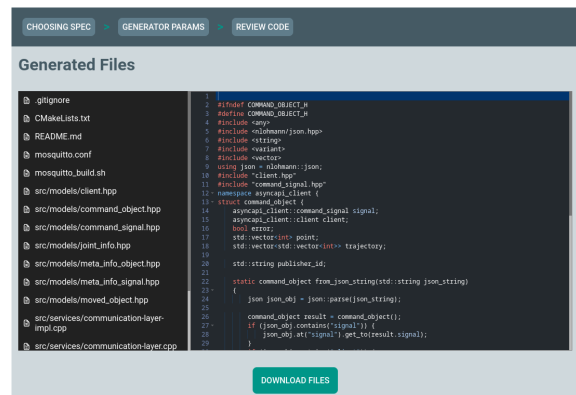

Figura 7. Página de revisão do código gerado

4.4 Implantação

Para implantação do *frontend* e do *backend* foram utilizados as plataformas da *Vercel* e da *Seenode* (previamente *ErlaServers*) respectivamente.

Deploy da aplicação *Web* (*frontend*): https://asyncapi-client-generator-nine.vercel.app/.

Deploy da aplicação *REST* (*backend*): https://web-6o6fqqtvz0bl.up-pl-waw1-1.apps.run-on-seenode.com/.

Para executar localmente as duas aplicações há um arquivo *docker-compose.yaml* disponível no repositório que irá

subir dois contêineres, um para o *frontend* e outro para o *backend*, as portas disponíveis são as portas 81 e 3334 respectivamente.

5. Conclusão E Trabalhos Futuros

O gerador de clientes automático partindo da especificação *AsyncAPI* é uma ferramenta útil para desenvolvedores de aplicações embarcadas ou aplicação Web em Angular que querem integrar aplicações assíncronas, gerando módulos ou sistemas completos que se comunicam com o *broker* especificado. Este código gerado pode ser facilmente atualizado com o auxílio dos *templates* criados e do gerador do *AsyncAPI*. Com isso, a especificação *AsyncAPI* se torna uma documentação do próprio cliente gerado, facilitando a produção de códigos complexos e permitindo que os desenvolvedores usem o tempo ganho em outras atividades.

Como trabalhos futuros espera-se que os templates criados sejam evoluídos para abarcar outros protocolos além do *MQTT* como *WebSockets*, *AMQP*, *STOMP*, *Kafka* e *NATS*. Desta forma outros *message brokers* poderiam ser utilizados além do *Mosquitto*, a exemplo do *Kafka*, do *RabbitMQ*, entre outros. Além disso, outras funcionalidades podem ser adicionadas como geração de testes unitários automáticos para validar os serviços criados e testes ponta a ponta para testar o *message broker*.

Outra atividade que pode ser adicionada são verificações a respeito da especificação que podem ser validadas antes da geração do código para certificar que não haja erros nos *templates*.

Futuramente, a especificação AsyncAPI pode ser mais explorada, por exemplo os *schemas* podem utilizar de composição e herança e os modelos podem utilizar desta abordagem para serem gerados. Outros atributos úteis são os de exemplo, que podem ser utilizados para gerar testes automaticamente e documentar o código gerado.

6. Referências

[1] A. Initiative, “Building the future of Event-Driven Architectures (EDA),” GitHub, 2023. https://www.asyncapi.com/ (accessed Oct. 15, 2023).

[2] A. Initiative, “Introduction,” GitHub, 2023. https://www.asyncapi.com/docs/tools/generator (accessed Oct. 15, 2023).

[3] A. Initiative, “List of official generator templates,” GitHub, 2023. https://github.com/search?q=topic%3Aasyncapi%2Btopic%3Agenerator%2Btopic%3Atemplate& (accessed Oct. 15, 2023).

[4] A. Initiative, “AsyncAPI Specification 2.6.0,” Asyncapi.com, 2023. https://www.asyncapi.com/docs/reference/specification/v2.6.0#asyncapi-specification (accessed Oct. 15, 2023).

[5] A. Initiative, “Modelina,” Asyncapi.com, 2023. https://www.asyncapi.com/tools/modelina (accessed Oct. 15, 2023).

[6] E. Foundation, “Eclipse Mosquitto,” Eclipse Mosquitto, Jan. 08, 2018. https://mosquitto.org/

[7] Jeroen Arnoldus, Mark, A. Serebrenik, and J.J. Brunekreef, Code Generation with Templates. Springer Science & Business Media, 2012.

[8] Burak Uyanik and Veysel Harun Şahin, “A template-based code generator for web applications,” Turkish Journal of Electrical Engineering and Computer Sciences, vol. 28, no. 3, pp. 1747–1762, May 2020, doi: https://doi.org/10.3906/elk-1910-44.

[9] N. Lohmann, “nlohmann/json,” GitHub, Oct. 30, 2020. https://github.com/nlohmann/json

[10] S. Software, “Hello from OpenAPI Generator | OpenAPI Generator,” openapi-generator.tech, 2018. https://openapi-generator.tech/