dojot Documentation

Release v0.7.0

Dojot Team

07 out. 2021

Conteúdo:

Esta é a documentação de alto nível para a plataforma dojot IoT desenvolvida pelo CPqD. Esta plataforma visa proporcionar aos desenvolvedores de aplicativos e dispositivos uma interação mais concisa e integrada, ao mesmo tempo em que se beneficia de uma infraestrutura altamente personalizável e eficiente.

CAPÍTULO 1

Arquitetura

Este documento descreve a arquitetura atual que guia a implementação da *dojot*, detalhando os componentes que compõem a solução, assim como as suas funcionalidades e como cada um deles contribui para a plataforma como um todo.

Aqui é feita uma breve explicação dos componentes, sendo esta descrição em alto nível e sem o objetivo de explicar os detalhes de implementação de cada um deles. Para isso, procure a documentação própria do componente.

Uma visão geral de toda a arquitetura é mostrada na figura acima e nas seções a seguir são fornecidos mais detalhes sobre cada componente.

Índice

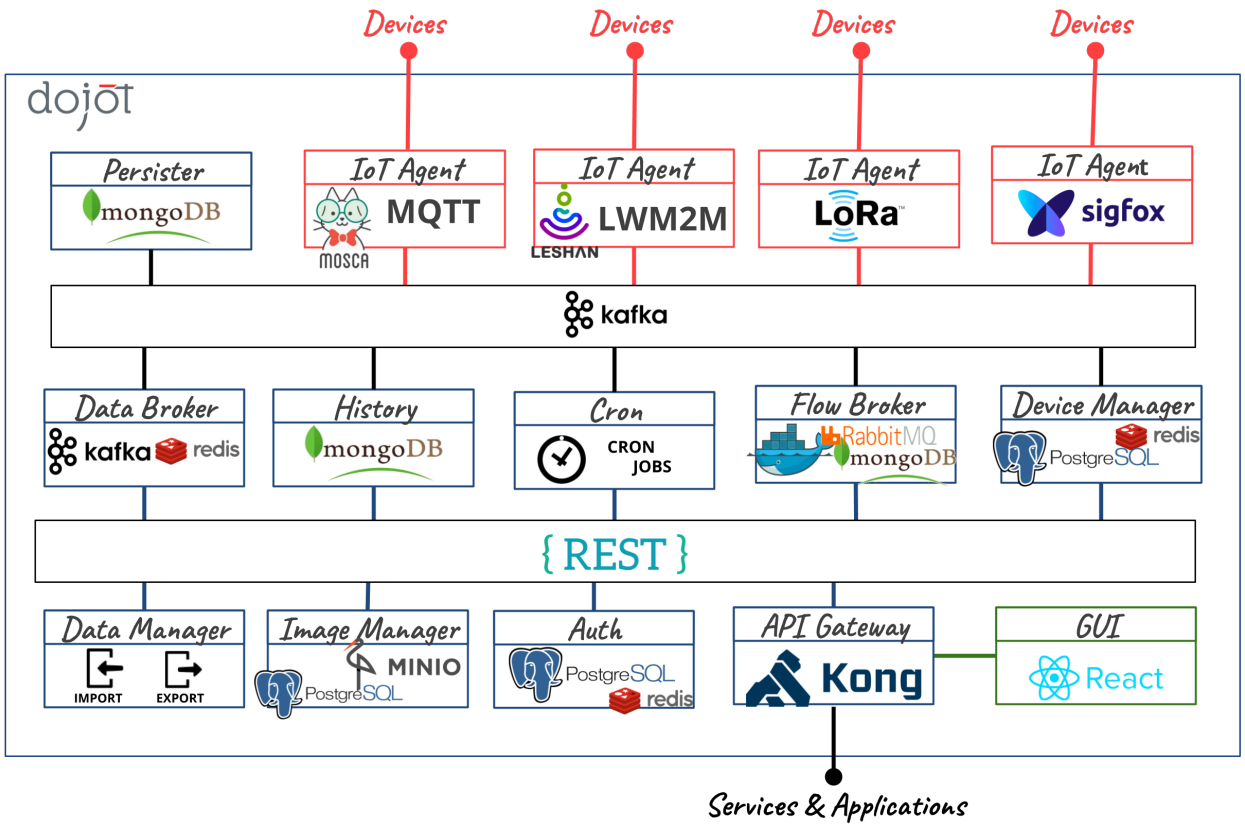


Fig. 1.1: A arquitetura de microsserviço da plataforma dojot.

1.1 Componentes

A *dojot* foi projetada para tornar possível uma prototipagem rápida, fornecendo uma plataforma fácil de usar, escalável e robusta. Sua arquitetura interna faz uso de muitos componentes conhecidos de código aberto, e outros projetados e implementados pela equipe dojot.

Usando a dojot: um usuário configura dispositivos de IoT por meio da GUI ou diretamente usando as APIs REST fornecidas pelo API Gateway. Os fluxos de processamento de dados também podem ser configurados - essas entidades podem executar uma variedade de ações, como gerar notificações quando um atributo de dispositivo específico atingir um determinado limite ou salvar todos os dados gerados por um dispositivo em um banco de dados externo. À medida que os dispositivos começam a enviar suas leituras para dojot, um usuário pode:

- receber essas leituras em tempo real pelos canais *socket.io* ou websocket;
- consolidar todos os dados em dispositivos virtuais;
- reunir todos os dados do banco de dados histórico e assim por diante.

Esses recursos podem ser usados por meio de APIs REST - esses são os blocos de construção básicos que qualquer aplicativo baseado na dojot deve usar. A GUI (interface gráfica), nossa aplicação exemplo, fornece uma maneira fácil de executar operações de gerenciamento para todas as entidades relacionadas à plataforma (usuários, dispositivos, modelos e fluxos) e também pode ser usada para verificar se tudo está funcionando bem.

O contexto do usuário é isolado e não há compartilhamento de dados, as credenciais de acesso são validadas pelo serviço de autorização para cada operação (solicitação da API). Portanto, um usuário pertencente a um contexto específico (*tenant*) não pode acessar nenhum dado (incluindo dispositivos, modelos, fluxos ou quaisquer outros dados relacionados a esses recursos) de outros.

Depois que os dispositivos são configurados, o IoT Agent é capaz de mapear os dados recebidos dos dispositivos, encapsulados no MQTT, por exemplo, e enviá-los ao *broker* de mensagens para distribuição interna. Dessa forma, os dados chegam ao serviço de persistência, por exemplo, para que possam persistir os dados em um banco de dados.

Para maiores informações sobre o que acontece na dojot, você pode conferir os *repositórios GitHub do projeto <https://github.com/dojot>*. Lá você encontrará todos os componentes utilizados pela plataforma.

Cada um dos componentes que compõem a arquitetura é brevemente descrito nas sessões subsequentes.

1.1.1 Kafka + DataBroker

O Apache Kafka é uma plataforma distribuída de mensageria que pode ser utilizada por aplicações que precisam transmitir dados ou consumir/produzir em canais de dados. Em comparação com outras soluções de mensagens de código aberto, o Kafka parece ser mais apropriado para cumprir os requisitos de arquitetura da *dojot* (isolamento de responsabilidade, simplicidade e assim por diante).

No Kafka, utiliza-se uma estrutura de tópicos especializada para garantir isolamento de dados de usuários e aplicações, viabilizando uma arquitetura *multi-tenant*.

O serviço DataBroker utiliza um banco de dados na memória para obter eficiência. Ele adiciona contexto ao Apache Kafka, possibilitando que serviços internos ou externos possam consumir dados em tempo-real com base no contexto. O DataBroker também pode ser um serviço distribuído para evitar que seja um ponto único de falha ou mesmo um gargalo para a arquitetura.

1.1.2 DeviceManager

O DeviceManager é uma entidade central responsável por manter as estruturas de dados de dispositivos e modelos (*templates*). Também é responsável por publicar quaisquer atualizações para todos os componentes interessados através do Kafka.

O serviço não mantém estados e tem seus dados persistidos em banco de dados, onde suporta isolamento de dados por tenants e aplicações, viabilizando uma arquitetura de *middleware* com *multi-tenancy*.

1.1.3 Agente IoT

Um agente IoT é um serviço de adaptação entre dispositivos físicos e componentes principais da *dojot*. Pode ser entendido como um *driver de dispositivo* para um conjunto de dispositivos. A plataforma *dojot* pode ter vários agentes IoT, cada um deles especializado em um protocolo específico, como, por exemplo, MQTT/JSON, CoAP/LWM2M, Lora/ATC, Sigfox/WDN e HTTP/JSON.

A comunicação por meio de canais seguros com dispositivos também é responsabilidade dos agentes IoT.

1.1.4 Serviço de Autorização de Usuários

Serviço que implementa o gerenciamento de perfil de usuários e controle de acesso. Basicamente qualquer chamada de aplicação através do API Gateway é validada por este serviço.

Para ser capaz de atender a um grande volume de chamadas de autorização, faz uso de cache, não mantém estados e pode ser escalado horizontalmente. Seus dados são mantidos em banco de dados clusterizável.

1.1.5 Flowbroker (Construtor de fluxo)

Esse serviço provê mecanismos para construir fluxos de processamento de dados para execução de um conjunto de ações. Os fluxos podem ser estendidos usando um bloco de processamento externo (que pode ser incluído utilizando APIs REST).

1.1.6 Data Manager

Este serviço gerencia a configuração de dados da dojot, possibilitando importar e exportar configurações.

1.1.7 Cron

Cron é um microsserviço da dojot que permite agendar eventos a serem emitidos - ou requisições a serem feitas - para outros microsserviços dentro da plataforma dojot.

1.1.8 Kafka2Ftp

O serviço kafka2ftp permite o encaminhamento de mensagens do Apache Kafka para servidores FTP. Ele se inscreve no tópico `tenant.dojot.ftp` do Kafka, onde as mensagens devem seguir um esquema específico. As mensagens podem ser redirecionadas para esses tópicos usando um nó específico do flowbroker.

1.1.9 Persister/History

O componente Persister funciona como um condutor de dados e eventos que devem ser persistidos em um banco de dados. Os dados são convertidos em uma estrutura de armazenamento que é enviada para o banco de dados correspondente.

Para armazenamento interno, utiliza-se uma base de dados não-relacional MongoDB que pode ser configurada em modo Sharded Cluster dependendo do caso de uso.

Os dados persistentes podem ser consultados por meio de uma API Rest fornecida pelo microsserviço *history*.

1.1.10 InfluxDB Storer e Retriever

Os serviços InfluxDB Storer e InfluxDB Retriever trabalham juntos, o InfluxDB Storer é responsável por consumir dados Kafka de dispositivos dojot e gravá-los no InfluxDB, enquanto o InfluxDB Retriever tem a função de obter os dados que foram escritos pelo InfluxDB Storer no InfluxDB via API REST .

1.1.11 Kong API Gateway

O Kong API Gateway é utilizado como um ponto de fronteira (*entry point*) entre as aplicações e serviços externos e os serviços internos da dojot. Isso resulta em inúmeras vantagens como, por exemplo, ponto único de acesso e facilidade na aplicação de regras sobre as chamadas de APIs como limitação de tráfego e controle de acesso.

1.1.12 GUI

A Interface Gráfica de Usuário (GUI) na *dojot* é uma aplicação WEB que provê interfaces responsivas para gerenciamento da plataforma, incluindo funcionalidades como:

- **Gerenciamento de perfil de usuários**: permite definir perfis e quais APIs podem ou não ser acessadas pelo respectivo perfil
- **Gerenciamento de usuários**: permite operações de criação, visualização, edição e remoção
- **Gerenciamento de modelos de dispositivos**: operações de criação, visualização, edição e remoção
- **Gerenciamento de dispositivos**: operações de criação, visualização (dispositivo e dados em tempo real), edição e remoção
- **Gerenciamento de fluxos de processamento**: permite operações de criação, visualização, edição e remoção de fluxos de processamento de dados
- **Notificações**: visualiza as notificações do sistema (em tempo real e histórico unificados)

1.1.13 Image manager

Este componente é responsável pelo armazenamento e recuperação de imagens de *firmware* de dispositivos. Ele é utilizado pelo mecanismo de atualização de *firmware*.

1.1.14 X.509 Identity Management

Este componente é responsável por atribuir identidades a dispositivos, tais identidades são representadas na forma de certificados x.509. Ele se comporta semelhante a uma Autoridade Certificadora (CA), onde é possível submeter um CSR e receber um certificado de volta. Tendo o certificado sido instalado no dispositivo, é possível se comunicar de forma segura com a plataforma dojot, pois os dados coletados pelo dispositivo são trafegados por um canal seguro (criptografado) e também é possível garantir a integridade dos dados.

1.1.15 Kafka WS

Este componente é responsável por obter dados do Apache Kafka via uma conexão Websocket pura. Foi desenhado para permitir que usuários da dojot possam recuperar dados brutos e/ou processados em tempo real de dispositivos da dojot.

1.2 Infraestrutura

Alguns outros componentes são utilizados na dojot, são eles:

- postgres: esse banco de dados é utilizado para persistir informações de vários componentes, como do gerenciador de dispositivos.
- redis: é um banco de dados em memória usado como cache em vários componentes, como o serviço de orquestração, gerenciador de subscrição, agentes IoT e outros. É bem leve e fácil de usar.
- rabbitMQ: *broker* de mensagens usado no orquestrador de serviço para implementar fluxos de ação relacionados que devem ser aplicados a mensagens recebidas de componentes.
- mongo: solução de banco de dados amplamente utilizada, fácil de usar e não adiciona *overhead* considerável (nos locais onde foi empregado na dojot).
- zookeeper: mantém sob controle serviços replicados em cluster.

1.3 Comunicação

Todos os componentes se comunicam de duas maneiras:

- Por meio de requisições HTTP: se um componente necessita recuperar dados de outro, como um agente IoT que precisa a lista de dispositivos configurados do gerenciador de dispositivos, ele pode enviar uma requisição HTTP para o componente apropriado.
- Por meio de mensagens Kafka: se um componente precisa enviar novas informações sobre um recurso controlado por ele (como novos dispositivos criados no gerenciador de dispositivos), o componente pode publicar esses dados através do Kafka. Utilizando esse mecanismo, qualquer outro componente que esteja interessado em tal informação precisa apenas ouvir um tópico específico para recebê-la. Note que este mecanismo não faz quaisquer associações fixas entre componentes. Por exemplo, o gerenciador de dispositivos não sabe quais componentes precisam de suas informações e um agente IoT não necessita saber qual componente está enviando dados através de um tópico específico.

CAPÍTULO 2

Arquitetura do Agente IoT

Este documento descreve a arquitetura do agente IoT usada pela dojot. Ele define um conjunto de recursos e opções básicas que devem ser seguidos para serem alinhados com a arquitetura dojot.

2.1 Quem deve ler isso?

Desenvolvedores que desejam criar novos agentes IoT para serem usados com a dojot.

2.2 Introdução

Usar dojot envolve lidar com as seguintes entidades:

- **dispositivos físicos**: dispositivos que enviam mensagens para agentes IoT. Eles podem ter sensores e podem ser configuráveis, mas isso não é obrigatório. Além disso, eles devem ter algum tipo de conectividade com outros serviços, para que possam enviar suas leituras para esses serviços.
- **usuários**: quem envia solicitações a dojot para gerenciar recursos, recuperar dados históricos do dispositivo, criar subscrição, gerenciar fluxos e assim por diante.
- **tenants**: separação lógica entre recursos que podem estar associados a vários usuários.
- **recursos**: elementos associados a uma entidade específica. Eles são:
  - *dispositivos*: representação de um elemento que possui atributos. Esse elemento pode ser um dispositivo físico ou virtual - um que não recebe atualizações de atributos diretamente por um dispositivo.
  - *modelos de dispositivo (templates)*: *blueprints* de dispositivos que contêm uma lista de atributos associados a essa classe de dispositivos. Todos os dispositivos são criados com base em um modelo, do qual herdarão atributos.
  - *tópicos*: Canais de comunicação Kafka usados para enviar e receber mensagens entre serviços dojot.
  - *flows*: Sequência de blocos de processamento criados por um usuário ou aplicativo e utilizados para analisar e pré-processar dados.

- **subjects**: grupo de tópicos que compartilham um fluxo de mensagens comum. Por exemplo, pode haver muitos tópicos usados para transmitir dados do dispositivo. Todos eles pertencem ao mesmo *subject device-data*.

Quando um novo agente IoT é criado, todas essas entidades devem ser levadas em consideração de maneira coordenada. Este documento lista todos os requisitos básicos para um novo agente IoT e eles são categorizados nos seguintes grupos:

1. **Segurança do dispositivo**: Os agentes IoT devem poder verificar se uma conexão de dispositivo é válida ou não. Uma conexão de dispositivo válida é definida como uma originada por um dispositivo físico confiável (ou qualquer elemento representativo, como *gateways*) que tem permissão para se conectar ao agente IoT. Um dispositivo é considerado confiável por: (1) criar um dispositivo associado a ele (que pode incluir informações de segurança, como chaves criptográficas) ou (2) indicar diretamente ao agente IoT que um dispositivo ou um elemento representativo tem permissão para conectar-se a ele (para que os elementos que servem como conexões de relé possam ser usados de maneira adequada e segura).
2. **Separação de contexto de informações**: cada recurso (dispositivo, modelos, tópicos e fluxos) está associado a um *tenant* específico e as entidades que não pertencem a esse *tenant* não devem ter permissão para acessar seus recursos. Isso é válido em todo a dojot e não é uma exceção para agentes IoT. Portanto, um agente IoT deve tratar separadamente todos os dispositivos que pertencem a diferentes *tenants* - incluindo o fato de que ninguém em um *tenant* deve saber a existência de outros *tenants*. Por exemplo, um agente MQTT IoT não deve permitir que mensagens enviadas para seu broker de dispositivos associados ao *tenant* A sejam publicadas em dispositivos inscritos no mesmo tópico pertencente ao *tenant* B.
3. **Informações e gerenciamento de agentes IoT**: qualquer agente IoT deve publicar seus recursos e modelos de informação. Por exemplo, ele deve informar outros serviços sobre qual é o modelo de dispositivo aceito para receber e enviar mensagens corretamente para um dispositivo físico específico. Também deve oferecer uma interface de gerenciamento para que um usuário possa alterar e recuperar seu comportamento, como opções de *log*, estatísticas, cotas etc.
4. **Operação do agente IoT**:Os agentes IoT devem poder receber e enviar mensagens (se permitido pelo protocolo) aos dispositivos e, portanto, enviar atualizações para outros serviços dojot com base nas mensagens recebidas do dispositivo. Todas as mensagens recebidas de um dispositivo específico e enviadas para outros serviços dojot devem ser enviadas na mesma ordem em que foram recebidas. Os agentes IoT também devem poder ativar ou desativar o processamento de mensagens de um dispositivo específico e detectar o estado do dispositivo.

   Um recurso extra que um agente IoT pode implementar são as atualizações de *firmware*. Dependendo do protocolo, pode ser possível fazer isso de maneira fácil, segura e confiável.

Cada um desses grupos será detalhado nas seções a seguir.

2.3 Segurança do dispositivo

Um gerenciamento IoT deve levar em consideração os seguintes aspectos da comunicação do dispositivo:

1. Identidade do dispositivo: deve aceitar apenas conexões de dispositivos físicos autorizados. A verificação se uma nova conexão foi originada por um dispositivo autorizado (que inclui verificar se um dispositivo específico está autorizado ou não) deve depender de chaves públicas e/ ou certificados assinados.
2. Segurança do canal de comunicação: todas as mensagens trocadas com um dispositivo físico devem ser criptografadas usando padrões criptográficos conhecidos, como TLS. Quaisquer protocolos de segurança próprios devem ser evitados.
3. Revogação de certificado: o agente IoT deve poder descartar qualquer mensagem do dispositivo autorizado anteriormente se seus dados de segurança tiverem sido comprometidos de alguma forma. Por exemplo, se a chave privada associada a um dispositivo específico vazar, todas as suas mensagens deverão ser ignoradas, pois não há garantia de que elas vieram desse dispositivo.

Cada um desses aspectos será detalhado nas seções a seguir.

2.3.1 Identidade do dispositivo

A verificação de identidade do dispositivo é o ponto de partida para lidar com a segurança da comunicação. Essa validação indicará ao agente IoT se o dispositivo que abriu a conexão é quem diz que é. Além disso, o agente IoT deve, uma vez que essa validação for bem-sucedida, verificar se este dispositivo pode se conectar a ele, verificando seu ID. Esta seção mostrará como fazer isso.

Para protocolos orientados a conexão, o agente IoT deve aceitar apenas conexões para dispositivos que possuem um certificado assinado por uma autoridade confiável pela dojot. Dado que esse certificado é válido, a identidade do dispositivo pode ser verificada de duas formas:

- ID do dispositivo codificado no certificado: embora seja um mecanismo menos confiável, ele permite maior flexibilidade usando muitos dispositivos em uma implantação controlada. Isso se baseia na configuração do nome comum (campo de certificado CN) como ID do dispositivo dojot. Portanto, o agente IoT deve verificar se este dispositivo existe ou não e permitir ou negar a conexão imediatamente, dependendo dessa verificação. Os pontos fracos desses mecanismos é que o certificado do dispositivo deve ser assinado pela CA interna da dojot (uma vez que existe um procedimento para assinar apenas um certificado por dispositivo) e, se esse certificado for válido, seu ID também deverá ser válido. Se qualquer outra autoridade de certificação for usada, esse mecanismo não terá uso válido.
- O agente IoT possui todos os certificados válidos: se um administrador deseja usar uma CA externa para assinar todos os certificados de dispositivo, não há controle real de qual ID de dispositivo foi usado para gerar um certificado específico. Portanto, o agente IoT deve ter todos os certificados válidos mapeados adequadamente em uma lista de dispositivos - isso garantirá que apenas um certificado seja permitido para um dispositivo específico e vice-versa.

Usando o primeiro mecanismo, o dispositivo (ou um operador que configura um dispositivo pela primeira vez) deve chamar a CA dojot para gerar um certificado assinado para si. Não há nenhuma ação adicional para o agente IoT usar a CA da dojot.

O segundo mecanismo, no entanto, exige que um agente IoT ofereça métodos para gerenciar certificados. O desenvolvedor deve levar em conta também que esse agente IoT deve poder escalar - esses certificados devem estar acessíveis a todas as instâncias do agente IoT, se permitido pela implantação.

2.3.2 Segurança da comunicação

Com um certificado válido, um dispositivo pode criar um canal de comunicação com a dojot. Para canais orientados a conexão, esse certificado deve ser usado juntamente com chaves criptográficas para fornecer um canal criptografado. Para outros tipos de canal (como canais para troca de mensagens por meio de um *gateway*, como LoRa ou sigfox), basta garantir que a conexão entre a dojot e o servidor de back-end seja segura. A identidade do *back-end* deve ser declarada previamente. Uma vez que se sabe que é confiável, todas as suas mensagens podem ser processadas sem grandes preocupações.

2.3.3 Revogação de certificado

Um agente IoT deve poder ser informado sobre certificados revogados. Ele deve expor uma API ou mensagens de configuração para permitir isso. Não deve permitir nenhuma comunicação com um dispositivo específico que usa um certificado revogado.

2.4 Separação de contexto de informações

Um *tenant* pode ser pensado simplesmente como um grupo de usuários que compartilham alguns recursos. Mas seu significado pode ir além disso - pode implicar que esses recursos não compartilhem nenhuma infraestrutura comum (considerando qualquer coisa que transmita, processe ou armazene dados) com recursos pertencentes a outros *tenants*. Pode-se ter instâncias de software separadas para processar dados de *tenants* diferentes, para que o processamento de dados de um *tenant* não afete o processamento de dados do outro, atingindo um nível mais alto de separação de contexto.

Embora isso seja desejável, alguns cenários de implantação podem forçar o uso de parte da mesma infraestrutura para diferentes *tenants* (por exemplo, quando a implantação tem um número reduzido de unidades de processamento ou conexões de rede). Portanto, para ter uma separação mínima de contexto entre os *tenants*, um agente IoT deve usar tudo o que puder para separá-los, como diferentes threads, filas, soquetes etc., e não deve confiar apenas nos recursos dos cenários de implantação (como agentes IoT diferentes para diferentes *tenants*). Por exemplo, para protocolos baseados em tópicos, como o MQTT, pode-se forçar tópicos diferentes para diferentes *tenants*. Se um dispositivo publicar dados em um tópico específico de propriedade de outro *tenant*, essa mensagem será ignorada ou bloqueada (o envio de um erro de volta ao dispositivo pode ser um comportamento opcional). Portanto, nenhum dispositivo de um *tenant* pode enviar mensagens para qualquer dispositivo de outro *tenant*.

O mecanismo pelo qual a separação de contexto é implementada depende muito de qual protocolo é usado. Uma análise completa deve ser realizada para implementar adequadamente esse recurso.

2.5 Informações e gerenciamento de agentes IoT

Um agente IoT deve expor todas as informações necessárias para usá-lo corretamente. Deve expor:

- **Modelo de dispositivo**: um agente IoT deve publicar qual é o modelo de dados aceito para um dispositivo válido. Isso deve ser feito publicando um novo modelo de dispositivo em outros serviços dojot. Deve haver um mecanismo para que instâncias diferentes do mesmo agente IoT publiquem o mesmo modelo de dispositivo (incluindo quaisquer IDs de modelo). Se o modelo do dispositivo for atualizado em uma versão mais recente de um agente IoT, o ID do modelo do dispositivo deverá mudar.
- **APIs de gerenciamento**: um agente IoT deve ser gerenciável e deve expor suas APIs para isso. O conjunto mínimo de APIs de gerenciamento que um agente IoT deve oferecer são:
  - *Log*: deve haver uma maneira de alterar o nível de log de um agente IoT;
  - *Estatísticas*: um agente IoT pode expor uma API para permitir que um usuário ou aplicativo recupere informações estatísticas sobre sua execução. Um administrador pode querer ativar ou desativar a geração de uma variável estatística específica, como o tempo de processamento.

Um agente IoT também deve ser capaz de coletar informações estatísticas relacionadas à sua execução. Além disso, deve permitir que um administrador defina cotas para essas quantidades medidas. Essas quantidades podem incluir, mas não estão limitadas a:

- estatísticas de transmissão
  - número de mensagens recebidas do dispositivo (total, por dispositivo, por *tenant*)
  - número de mensagens publicadas do dispositivo para dojot (total, por dispositivo, por *tenant*)
  - número de mensagens enviadas aos dispositivos (total, por dispositivo, por *tenant*)
  - [opcional] tempo decorrido entre o recebimento de uma mensagem de um dispositivo físico e a publicação (total - média, por dispositivo - média, por *tenant* - média)
- Verificação de integridade do serviço do agente IoT - estatísticas do sistema (memória, disco etc.) usadas pelo serviço

Muitos outros valores podem ser reunidos. A lista acima é a lista mínima que um agente IoT deve expor a outros serviços. Especialmente para verificação de integridade, há um documento detalhando como expô-lo.

2.6 Operação do agente IoT

O principal objetivo de um agente IoT é publicar dados de um dispositivo específico em outros serviços dojot. Sua operação é dupla: receba e processe mensagens relacionadas ao gerenciamento de dispositivos de outros serviços, bem como receba mensagens dos próprios dispositivos (ou de seus elementos representativos) e publique esses dados em outros serviços.

As seções a seguir descrevem como um agente IoT pode enviar e receber mensagens de/para outros serviços da dojot e quais são os pontos que ele deve levar em consideração ao receber mensagens de dispositivos físicos.

2.6.1 Mensagens

Tenants

No início, todos os agentes IoT (na verdade, todos os serviços que precisam receber ou enviar mensagens relacionadas aos dispositivos) devem conhecer a lista de *tenants* configurados. Esta é a informação mais básica que o agente IoT precisa saber para funcionar corretamente. A solicitação que deve ser enviada ao serviço Auth é esta (todas as solicitações enviadas dos serviços dojot para seus próprios serviços devem usar o usuário e *tenant* 'dojot-management'):

Host: Auth	
Endpoint: /admin/tenants	Método: GET
Requisição	
Cabeçalhos	Authorization: Bearer ${JWT}
Resposta	
Cabeçalhos	Content-Type: application/json
Formato do corpo	`tenants =>` `  tenant => string`

Um exemplo de resposta para esta requisição:

```
{
  "tenants": [
    "admin",
    "users",
    "system"
  ]
}
```

Após o *bootstrap*, é necessário se inscrever para receber eventos de *tenant* usando o tópico `dojot-management.dojot.tenancy` do Kafka.

O tópico `dojot-management.dojot.tenancy` do Kafka será usado para receber eventos de ciclo de vida dos *tenants*. Sempre que um novo *tenant* é criado ou excluído, a seguinte mensagem será publicada:

Tópico: `dojot-management.dojot.tenancy`	
Formato do corpo (JSON)	`type="CREATE"/"DELETE"` `tenant=>string`

Uma mensagem de exemplo recebida por este tópico é:

```
{
  "type": "CREATE",
  "tenant": "new_tenant"
}
```

Este prefixo do tópico pode ser configurado, veja mais na documentação do componente *Auth* *Componentes e APIs*.

Veja mais sobre *Inicialização dos tenants* na comunicação interna.

Subjects

Os seguintes *subjects* devem ser usados pelo agente IoT:

- *Subject*: *dojot.device-manager.device*
- *Subject*: *device-data*

Com a lista de *tenants*, o agente IoT pode solicitar tópicos para receber eventos de ciclo de vida do dispositivo e para publicar novos dados de atributo do dispositivo. Isso é feito enviando os *subjects* como na seguinte solicitação ao DataBroker:

Host: DataBroker	
Endpoint: /topic/{subject}	Método: GET
Requisição	
Cabeçalhos	Authorization: Bearer ${JWT}
Resposta	
Cabeçalhos	Content-Type: application/json
Formato do corpo	`topic => string`

Um exemplo de resposta para esta requisição:

```
{
  "topic": "admin.device-data"
}
```

Cada um desses aspectos será detalhado nas seções a seguir

Subject: *dojot.device-manager.device*

O tópico relacionado a este *subject* será usado para receber eventos do ciclo de vida do dispositivo para um *tenant* específico. Seu formato é:

Subject: *dojot.device-manager.device*	

Formato do corpo (JSON)	event => "create" / "update"
	meta => service
	  service => string
	data =>
	  id => string
	  label => string
	  templates => *number
	  attrs => [*template_attrs]
	  created => iso_date

Formato do corpo (JSON)	event => "remove"
	meta =>
	  service => string
	data =>
	  id => string

Formato do corpo (JSON)	event => "configure"
	meta =>
	  service => string
	  timestamp => int (Unix Timestamp - ms)
	data =>
	  id => string
	  attrs => *device_attrs

O atributo *device_attrs* é um JSON de chave/valor ainda mais simples, como:

```
{
  "temperature" : 10,
  "height" : 280
}
```

Uma mensagem de exemplo recebida por este tópico é:

```
{
  "event": "create",
  "meta": {
    "service": "admin"
  },
  "data": {
    "id": "efac",
    "label": "Device 1",
    "templates": [1, 2, 3],
    "attrs": {
      "1": [
        {
          "template_id": "1",
```

(continua na próxima página)

(continuação da página anterior)

```
            "created": "2018-01-05T15:41:54.840116+00:00",
            "label": "this-is-a-sample-attribute",
            "value_type": "float",
            "type": "dynamic",
            "id": 1
          }
        ]
    },
    "created": "2018-02-06T10:43:40.890330+00:00"
  }
}
```

Subject: *device-data*

O tópico relacionado a este *subject* será usado para publicar dados recuperados de um dispositivo físico em outros serviços da dojot. Seu formato é:

Subject: *device-data*	

Formato do corpo (JSON)	`metadata => deviceid tenant timestamp`
	`  deviceid => string`
	`  tenant => string`
	`  timestamp => int (Unix Timestamp - ms or s)`
	`attrs => *device_attrs`

O *timestamp* está associado a quando os valores de atributo foram coletados pelo dispositivo (isso pode ser feito pelo próprio dispositivo - ao enviar diretamente o atributo *timestamp* na mensagem - ou pelo agente IoT, se nenhum *timestamp* foi definido pelo dispositivo). O *timestamp* pode ser tanto UNIX quanto ISO, em segundos ou milissegundos.

Uma mensagem de exemplo recebida por este tópico é:

```
{
  "metadata": {
    "deviceid": "c6ea4b",
    "tenant": "admin",
    "timestamp": 1528226137452,
  },
  "attrs": {
    "humidity": 60
  }
}
```

Veja mais sobre *Enviando mensagens via Kafka* na comunicação interna.

2.6.2 Atualização de *Firmware*

Um agente IoT pode implementar mecanismos para atualizar o *firmware* nos dispositivos.

2.7 Comportamento

A ordem na qual um dispositivo físico envia seus atributos não deve ser alterada quando o agente IoT publica esses dados em outros serviços dojot.

Se o protocolo impuser qualquer ID único externo para cada dispositivo, o agente IoT deverá criar uma tabela de correlação para converter adequadamente esse ID único externo em ID de dispositivo dojot e vice-versa.

2.8 Bibliotecas para auxiliar no desenvolvimento de novos agentes IoT

Temos bibliotecas que abstraem alguns pontos descritos nos tópicos anteriores para facilitar o desenvolvimento de um Agente Iot.

Existem duas bibliotecas:

- node.js **recomendada** (https://www.npmjs.com/package/@dojot/iotagent-nodejs)
- java (https://jitpack.io/#dojot/iotagent-java)

CAPÍTULO 3

Conceitos

Este documento fornece informações sobre os conceitos e abstrações da dojot.

Índice

Nota:

- **Público**
  - Usuários que desejam dar uma olhada em como a dojot funciona;
  - Desenvolvedores de aplicativos.
- Nível: básico

3.1 noções básicas da dojot

Antes de usar a dojot, você deve estar familiarizado com algumas operações e conceitos básicos. Eles são muito simples de entender e usar, mas sem eles, todas as operações podem se tornar obscuras e sem sentido. Vamos nos concentrar em explicar o que são dispositivos, modelos e fluxos na dojot.

Se você quiser obter mais informações sobre como a dojot funciona internamente, consulte a *Arquitetura* para se familiarizar com todos os componentes internos.

3.1.1 Autenticação de usuário

Todas as solicitações HTTP suportadas pela dojot são enviadas para o *API gateway*. Para controlar qual usuário deve acessar quais *endpoints* e recursos, a dojot utiliza JSON Web Token (uma ferramenta útil é o jwt.io) que codifica coisas como (não se limitando a isso):

- Identidade do usuário
- Dados de validação
- Data de validade do *token*

O componente responsável pela autenticação do usuário é auth. Você pode encontrar um tutorial de como autenticar um usuário e obter um token de acesso em *Obtendo um token de acesso*.

3.1.2 Autenticação de dispositivo

A autenticação de dispositivos se baseia na utilização de chaves criptográficas assimétricas e é feita através de certificados x.509 gerenciados pela dojot. Para isso, é preciso instalar no dispositivo tal certificado. A dojot possui um serviço de emissão de certificados onde é possível obter o certificado para ser instalado no dispositivo.

Além do certificado e das chaves assimétricas, o dispositivo precisa *confiar* na *Autoridade Certificadora* da dojot, ou seja, também é preciso instalar o certificado raiz da plataforma dojot.

O certificado é requisitado pelo administrador do tenant ao qual os dispositivos estão cadastrados. Uma vez que o administrador siga os passos necessários para a obtenção do certificado, ele terá que instalar no dispositivo o certificado emitido e o certificado raiz da dojot. É importante ressaltar que o certificado possui em sua composição uma chave criptográfica assimétrica, esta chave é chamada de chave pública (que qualquer um tem acesso), enquanto que no dispositivo deve ser instalada a chave privada (chave que só o dispositivo tem acesso).

Uma vez que o dispositivo contenha a chave privada, o certificado (contendo a chave pública) e também o certificado raiz da dojot, é possível estabelecer um canal de comunicação seguro com a plataforma dojot, na qual o dispositivo é identificado através do seu certificado.

Após o canal seguro ser estabelecido, o dispositivo é capaz de publicar dados e também receber dados desde que ele esteja autorizado a isso.

Verifique *Usando MQTT com segurança (TLS)* para mais informações sobre seu uso.

3.1.3 Dispositivos e modelos (templates)

Na dojot, um dispositivo é uma representação digital de um dispositivo ou gateway real com um ou mais sensores ou de um virtual com sensores/atributos inferidos de outros dispositivos. Em toda a documentação, esse tipo de dispositivo será chamado simplesmente de "dispositivo". Se o dispositivo real precisar ser referenciado, nós o chamaremos de "dispositivo físico".

Considere, por exemplo, um dispositivo físico com sensores de temperatura e umidade; ele pode ser representado na dojot como um dispositivo com dois atributos (um para cada sensor). Chamamos esse tipo de dispositivo como dispositivo normal ou por seu protocolo de comunicação, por exemplo, dispositivo MQTT, dispositivo CoAP, etc.

Também podemos criar dispositivos que não correspondem diretamente aos seus homólogos físicos, por exemplo, podemos criar um com maior nível de informação de temperatura (está ficando mais quente ou mais frio) cujos valores são inferidos a partir de sensores de temperatura de outros dispositivos. Esse tipo de dispositivo é chamado de dispositivo virtual.

Todos os dispositivos são criados com base em um ou mais modelos (*templates*), que podem ser pensados como um modelo de dispositivo. Como "modelo", poderíamos pensar em números de peça ou modelos de produtos - um protótipo a partir do qual os dispositivos são criados. Os modelos na dojot têm um rótulo (qualquer sequência alfanumérica), uma lista de atributos que conterão todas as informações emitidas pelo dispositivo.

De fato, os modelos podem representar não apenas "modelos de dispositivos", mas também podem abstrair uma "classe de dispositivos". Por exemplo, poderíamos ter um modelo para representar todos os termômetros que serão usados na dojot. Este modelo teria apenas um atributo chamado, digamos, "temperatura". Ao criar o dispositivo, o usuário selecionaria seu "modelo físico", digamos TexasInstr882 e o modelo "termômetro". O usuário também teria que adicionar instruções de tradução (implementadas em termos de fluxos de dados, construídas no construtor de fluxo) para mapear a leitura de temperatura que será enviada do dispositivo para um atributo de "temperatura".

Para criar um dispositivo, um usuário seleciona quais modelos irão compor esse novo dispositivo. Todos os seus atributos são mesclados e associados a ele - eles estão fortemente vinculados ao modelo original para que qualquer atualização de modelo reflita todos os dispositivos associados.

O componente responsável pelo gerenciamento de dispositivos (reais e virtuais) e modelos (*templates*) é o DeviceManager.

A documentação do *DeviceManager* no ReadMe do GitHub explica com mais detalhes todas as operações disponíveis. Você pode encontrar o link em *Componentes e APIs*.

3.1.4 Fluxos (*Flows*)

Um fluxo é uma sequência de blocos que processa um evento ou mensagem de dispositivo específica. Contém:

- ponto de entrada: um bloco representando qual é o gatilho para iniciar um fluxo específico;
- blocos de processamento: um conjunto de blocos que executam operações usando o evento. Esses blocos podem ou não usar o conteúdo desse evento para processá-lo ainda mais. As operações podem ser: testar conteúdo para valores ou intervalos específicos, análise de posicionamento geográfico, alterar atributos de mensagens, executar operações em elementos externos e assim por diante.
- ponto de saída: um bloco que representa para onde os dados resultantes devem ser encaminhados. Esse bloco pode ser um banco de dados, um dispositivo virtual, um elemento externo e assim por diante.

O componente responsável por lidar com esses fluxos é flowbroker.

Verifique *Usando o construtor de fluxos (Flowbroker)* para mais informações sobre seu uso.

CAPÍTULO 4

Componentes e APIs

4.1 Componentes

Tabela 4.1: Componentes

Componentes	Repositório / Site principal	Documentação dos componentes	API de componentes Documentação
MongoDB	MongoDB site	MongoDB doc.	
Postgres	PostgreSQL site	PostgreSQL doc.	
Kong API gateway (Community Edition)	Kong site	Kong doc.	
Redis	Redis site	Redis doc.	
Zookeeper	Zookeeper site	Zookeeper doc.	
Kafka	Kafka site	Kafka doc.	
VerneMQ	VerneMQ site	VerneMQ doc.	
Leshan	Leshan site	Leshan doc.	
InfluxDB	InfluxDB site	InfluxDB doc.	
Auth	GitHub - auth		API - auth
Dojot Kong	GitHub - Dojot Kong		
History	GitHub - history		API - history
Device Manager	GitHub - DeviceManager		API - DeviceManager
Image Manager	GitHub - image-manager		API - image-manager
GUI	GitHub - GUI		
GUI - V2	GitHub - V2		
Flowbroker	GitHub - flowbroker		API - flowbroker
Databroker	GitHub - data-broker		API - data-broker
IotAgent VerneMQ (MQTT) - **Padrão**	GitHub - iotagent-vernemq		
	GitHub - iotagent-mosca		

4.2 APIs expostas (API Gateway)

O *API gateway* usado na dojot reencaminha alguns dos *endpoints* dos componentes. A tabela a seguir mostra o mapeamento entre **endpoints expostos** e **endpoints de componentes**. Veja mais sobre como usar as APIs em *Utilizando a API da dojot*.

Tabela 4.2: Endpoints expostos

Componentes	Endpoints expostos pelo API *gateway*	Componentes *Endpoint*	Componentes *Endpoint* Documentação	Precisa de Autenticação
GUI	/	/		Não
Dashboard	/v2	/		Não
Device Manager	/device	/device	API - DeviceManager	Sim
Device Manager	/template	/template	API - DeviceManager	Sim
Flowbroker	**/flows**	/	API - flowbroker	Sim
Auth	**/auth**	/	API - auth	Não
Auth	**/auth**/revoke	/revoke	API - auth	Não
Auth	**/auth**/user	/user	API - auth	Sim
Auth	**/auth**/pap	/pap	API - auth	Sim
History	**/history**	/	API - history	Sim
Data Manager	/import	/import	API - Data Manager	Sim
Data Manager	/export	/export	API - Data Manager	Sim
Cron	/cron	/cron	API - Cron	Sim
Image Manager	**/fw-image**	/	API - image-manager	Sim
Data Broker	/device/ {deviceID} /latest	/device/ {deviceID} /latest	API - data-broker	Sim
Data Broker	/subscription	/subscription	API - data-broker	Sim
Data Broker	/stream	/stream	API - data-broker	Sim
Data Broker	/socket.io	/socket.io	API - data-broker	Não
X.509 Identity Management	**/x509/v1**	**/api/v1**	API - x509-identity-mgmt	Sim
Kafka WS	**/kafka-ws/v1/ticket**	/v1/ticket	API - kafka-ws	Sim
Kafka WS	**/kafka-ws/v1**	/v1	API - kafka-ws	Não
InfluxDB Retriever	/tss/v1/	/tss/v1/	API - InfluxDB-Retriever	Sim
InfluxDB Retriever - Documentation	/tss/v1/api-docs	/tss/v1/api-docs	API - InfluxDB-Retriever	Não

NOTA: Alguns *endpoints* de componentes não são expostos, mas são utilizados internamente.

Além disso, o *API gateway* redireciona os *endpoints* com suas respectivas portas do componente, para que se tornem uniformes: todos eles são acessíveis pela mesma porta (o padrão é a porta TCP 8000), consulte a tabela a seguir.

Tabela 4.3: *Endpoints* originais para o API gateway

Componentes	*Endpoints* Originais	*Gateway Endpoint*
GUI	host:80/	host:8000/
Dashboard	host:80/	host:8000/v2
Device Manager	host:5000/device	host:8000/device
Device Manager	host:5000/template	host:8000/template
Flowbroker	host:80/	host:8000/flows
Auth	host:5000/	host:8000/auth
Auth	host:5000/revoke	host:8000/auth/revoke
Auth	host:5000/user	host:8000/auth/user
Auth	host:5000/pap	host:8000/auth/pap
History	host:8000/	host:8000/history
Data Manager	host:3000/import	host:8000/import
Data Manager	host:3000/export	host:8000/export
Cron	host:5000/cron	host:8000/cron
Image Manager	host:5000/	host:8000/fw-image
Data Broker	host:80/device/{{deviceID}}/latest	host:8000/device/{deviceID}/latest
Data Broker	host:80/subscription	host:8000/subscription
Data Broker	host:80/stream	host:8000/stream
Data Broker	host:80/socket.io	host:8000/socket.io
X.509 Identity Management	host:3000/api/v1	host:8000/x509/v1
Kafka WS	host:8080/v1/ticket	host:8000/kafka-ws/v1/ticket
Kafka WS	host:8080/v1/topics	host:8000/kafka-ws/v1/topics
InfluxDB Retriever	host:3000/tss/v1/	host:8000/tss/v1/
InfluxDB Retriever - Documentation	host:3000/tss/v1/api-docs	host:8000/tss/v1/api-docs

4.3 Bibliotecas

Dojot também possui várias bibliotecas usadas em seus próprios componentes. Essas bibliotecas estão abaixo listadas:

Tabela 4.4: Component Libraries by Language

Componentes	Linguagem	Bibliotecas
Module	Python	https://github.com/dojot/dojot-module-python https://pypi.org/project/dojot.module/
Module	Java	https://github.com/dojot/dojot-module-java https://jitpack.io/#dojot/dojot-module-java
Module	Node JS	https://github.com/dojot/dojot-module-nodejs https://www.npmjs.com/package/@dojot/dojot-module
IoT Agent	Java	https://github.com/dojot/iotagent-java https://jitpack.io/#dojot/iotagent-java
IoT Agent	Node JS	https://github.com/dojot/iotagent-nodejs https://www.npmjs.com/package/@dojot/iotagent-nodejs
Module Logger	Node JS	https://github.com/dojot/dojot-module-logger-nodejs https://www.npmjs.com/package/@dojot/dojot-module-logger
Helthcheck	Node JS	https://github.com/dojot/healthcheck-nodejs https://www.npmjs.com/package/@dojot/healthcheck
Microservice SDK	Node JS	https://github.com/dojot/dojot-microservice-sdk-js https://www.npmjs.com/package/@dojot/microservice-sdk

4.4 Mensagens do Kafka

Essas são as mensagens enviadas pelos componentes e seus tópicos. Se você estiver desenvolvendo um novo componente interno (como um novo agente IoT), consulte a API - data-broker para verificar como receber mensagens.

Tabela 4.5: *Endpoints* Originais

Componentes	Mensagem	Subject
DeviceManager	Dispositivo CRUD (Criar, Ler, Atualizar e Excluir) (Mensagens - DeviceManager)	`dojot.device-manager.device`
iotagent-mosca	Atualização de dados do dispositivo (Mensagens - iotagent-mosca)	`device-data`
auth	Criação/remoção de *Tenants* (Mensagens - auth)	`dojot.tenancy`

CAPÍTULO 5

Comunicação interna

Esta página descreve como cada serviço na dojot se comunica.

5.1 Componentes

Os principais componentes atuais na dojot são mostrados em Fig. 5.1.

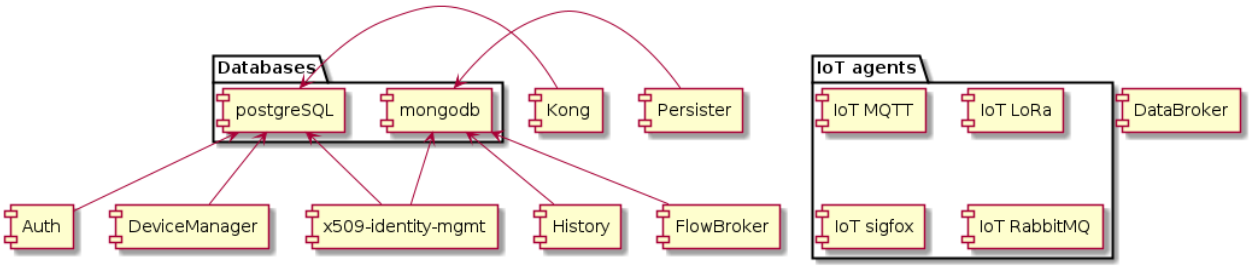


Fig. 5.1: Componentes da Dojot

Eles são:

- Auth: mecanismo de autenticação
- DeviceManager: armazenamento de dispositivo e modelo.
- Persister: componente que armazena todos os dados gerados por dispositivos.
- History: componente que expõe todos os dados gerados por dispositivos.
- DataBroker: lida com *subjects* e tópicos do Kafka, além de conexões socket.io.
- Flowbroker: lida com fluxos (CRUD e execução de fluxo)
- IoT agents: agentes para diferentes protocolos.

Cada serviço será descrito brevemente nesta página. Mais informações podem ser encontradas na documentação de cada componente.

5.2 Mensagens e autenticação

Existem dois meios pelos quais os componentes dojot podem se comunicar: via solicitações HTTP REST e via Kafka. Eles são destinados a diferentes propósitos.

As solicitações HTTP podem ser enviadas no momento da inicialização quando um componente deseja, por exemplo, informações sobre recursos específicos, como lista de dispositivos ou *tenants*. Para isso, eles devem saber qual componente possui qual recurso para recuperá-los corretamente. Isso significa - e isso é muito importante porque conduz as escolhas arquiteturais na dojot - que apenas um único serviço é responsável por recuperar modelos de dados para um recurso específico (observe que um serviço pode ter várias instâncias). Por exemplo, o DeviceManager é responsável por armazenar e recuperar o modelo de informações para dispositivos e modelos, FlowBroker para descrições de fluxo, History para dados históricos e assim por diante.

O Kafka, por outro lado, permite uma comunicação pouco acoplada entre instâncias de serviços. Isso significa que um produtor (quem envia uma mensagem) não sabe quais componentes receberão sua mensagem. Além disso, qualquer consumidor não sabe quem gerou a mensagem de que está sendo consumida. Isso permite que os dados sejam transmitidos com base em "interesses": um consumidor está interessado em receber mensagens com um determinado "assunto" (*subject*) (mais sobre isso mais tarde) e os produtores enviarão mensagens para todos os componentes que estiverem interessados nele. Observe que esse mecanismo permite que vários serviços emitam mensagens com o mesmo "assunto" (*subject*), bem como vários serviços que consomem mensagens com o mesmo "assunto" (*subject*), sem soluções alternativas complicadas.

5.2.1 Enviando solicitações HTTP

Para enviar solicitações via HTTP, um serviço deve criar um *token* de acesso, descrito aqui. Não há outras considerações além de seguir a descrição da API associada a cada serviço. Isso pode ser visto na figura Fig. 5.2. Observe que todas as interações descritas aqui são abstrações das reais. Além disso, deve-se notar que essas interações são válidas apenas para componentes internos. Qualquer serviço externo deve usar o Kong como ponto de entrada.

Nesta figura, um cliente recupera um *token* de acesso para o usuário *admin* cuja senha é p4ssw0rd. Depois disso, um usuário pode enviar uma solicitação para as APIs HTTP usando-o. Isso é mostrado na Fig. Fig. 5.3. Nota: o mecanismo de autorização real é detalhado no documento *Auth + API gateway (Kong)*.

Nesta figura, um cliente cria um novo dispositivo usando o *token* recuperado em Fig. 5.2. Essa solicitação é analisada pelo Kong, que chamará o Auth para verificar se o usuário definido no *token* tem permissão para `POST` para o *endpoint* `/device`. Somente após a aprovação dessa solicitação, o Kong a encaminhará para o DeviceManager.

5.2.2 Enviando mensagens via Kafka

Kafka usa uma abordagem bem diferente. Cada mensagem deve ser associada a um *subject* e um *tenant*. Isso é mostrado na Fig. 5.4;

Neste exemplo, o DeviceManager precisa publicar uma mensagem sobre um novo dispositivo. Para isso, ele envia uma solicitação ao DataBroker, indicando qual *tenant* (dentro do token JWT) e qual *subject* (`dojot.device-manager.devices`) deseja usar para enviar a mensagem.

Para entender melhor como tudo funciona, você pode verificar a documentação do *Data Broker* para o componente e API, os links estão em *Componentes e APIs*.

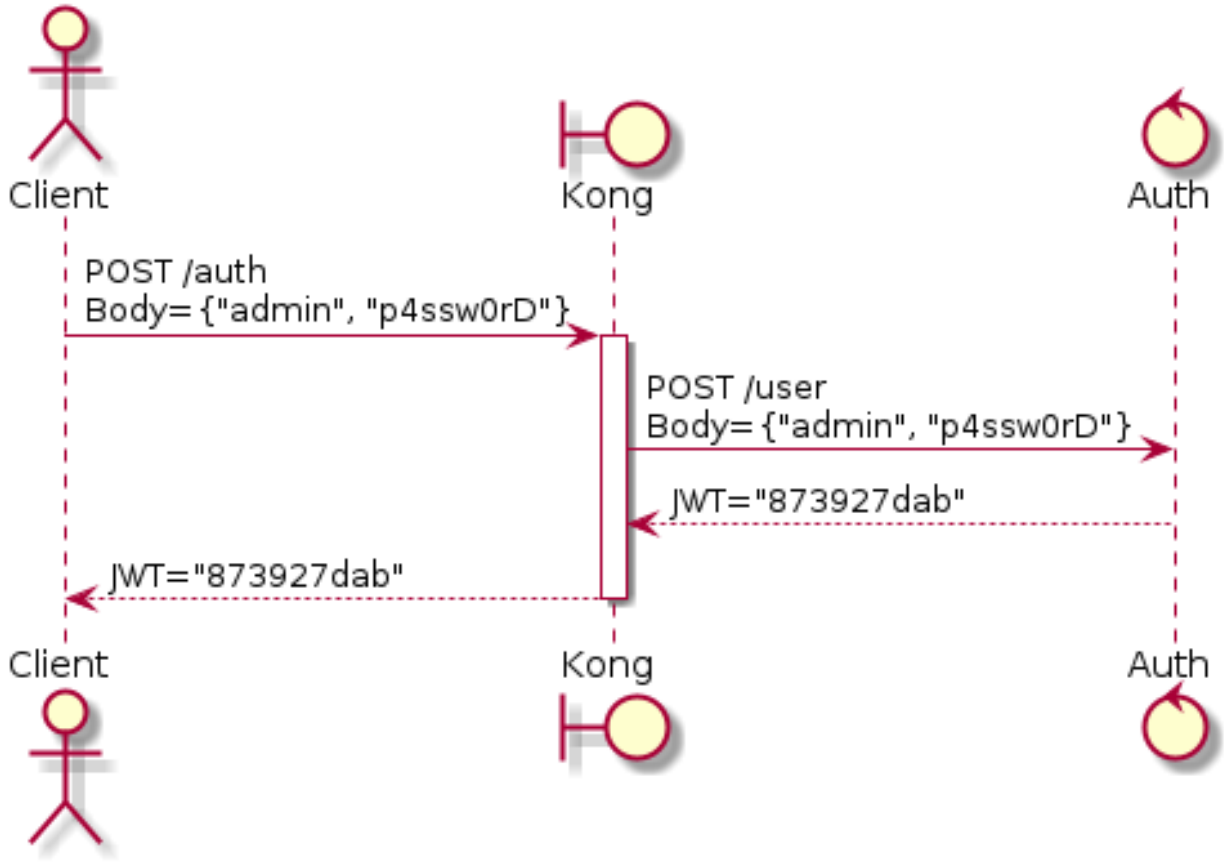


Fig. 5.2: Autenticação inicial

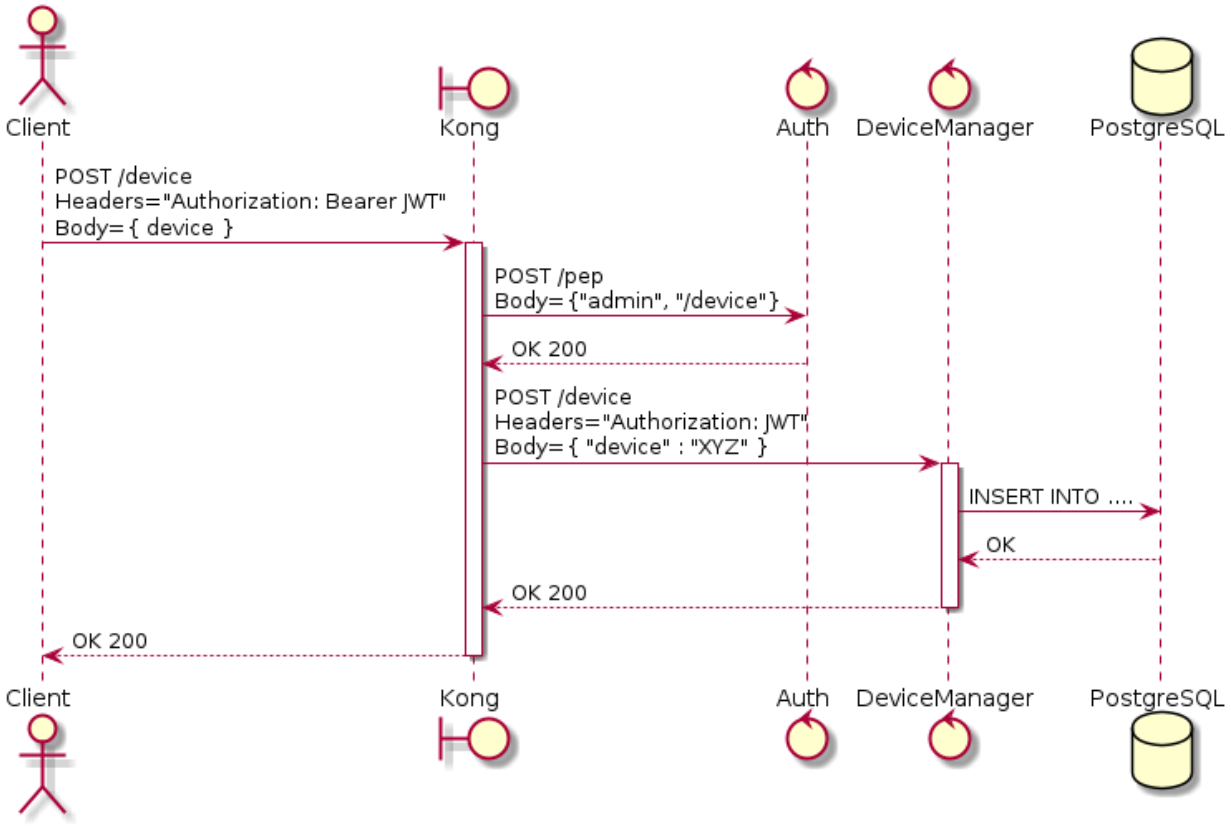


Fig. 5.3: Enviando mensagens para a API HTTP

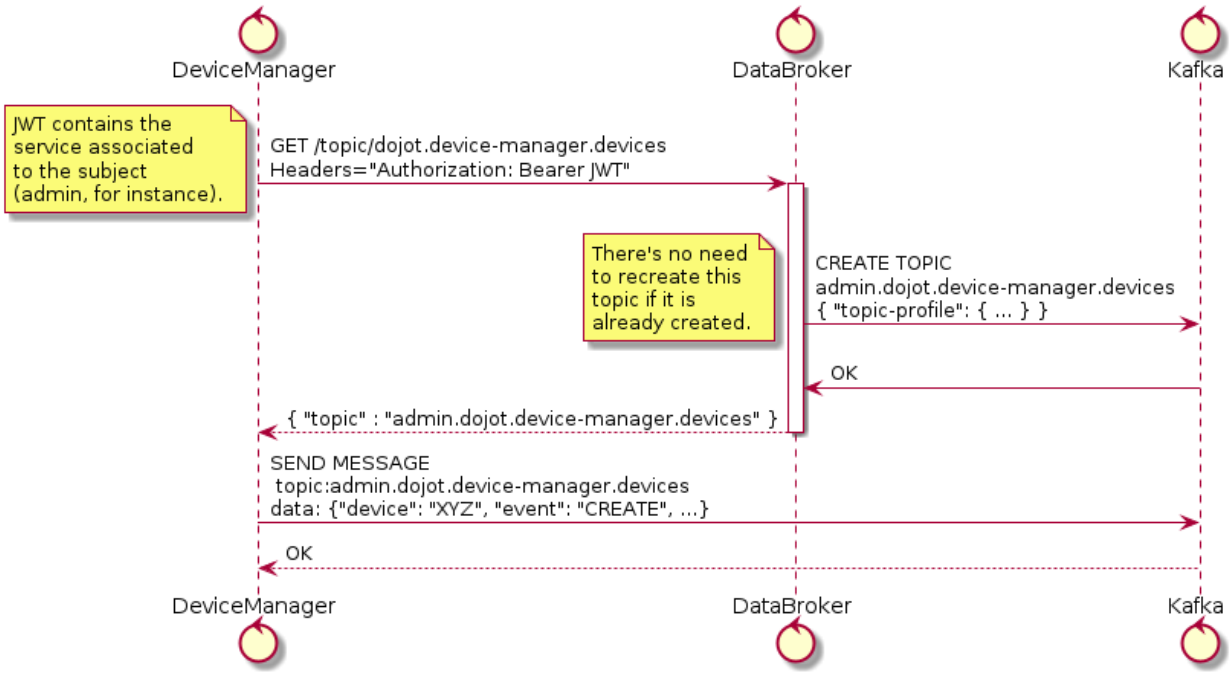


Fig. 5.4: Recuperando tópicos do Kafka

5.2.3 Inicialização dos *tenants*

Todos os componentes estão interessados em um conjunto de *subjects*, que serão usados para enviar ou receber mensagens do Kafka. Como a dojot agrupa tópicos do Kafka e tenants em *subjects* (um *subjects* será composto por um ou mais tópicos Kafka, cada um transmitindo mensagens para um *tenant* específico), o componente deve iniciar cada *tenant* antes de enviar ou receber mensagens. Isso é feito em duas fases: tempo de inicialização do componente e tempo de execução do componente.

Na primeira fase, um componente solicita ao Auth para recuperar todos os *tenants* configurados no momento. Está interessado, digamos, em consumir mensagens dos *subject device-data* e *dojot.device-manager.devices*. Portanto, ele solicitará ao DataBroker um tópico para cada *tenant* para cada *subject*. Com essa lista de tópicos, ele pode criar Produtores e Consumidores para enviar e receber mensagens através desses tópicos. Isso é mostrado em Fig. 5.5.

A segunda fase inicia após a inicialização e seu objetivo é processar todas as mensagens recebidas pelo Kafka se subscrevendo no tópico `dojot-management.dojot.tenancy`. Isso incluirá qualquer *tenant* criado após todos os serviços estarem em funcionamento. Fig. 5.6 mostra como lidar com essas mensagens.

Todos os serviços que estão de alguma forma interessados em usar *subjects* devem executar este procedimento para receber corretamente todas as mensagens.

5.3 Auth + API gateway (Kong)

Auth é um serviço profundamente conectado ao Kong. É responsável pelo gerenciamento, autenticação e autorização do usuário. Como tal, é invocado pelo Kong sempre que uma solicitação é recebida por um de seus*endpoints* registrados. Esta seção detalha como isso é realizado e como eles funcionam juntos.

5.3.1 Configuração do Kong

Existem dois procedimentos de configuração ao iniciar o Kong na dojot:

1. Migrando Dados Existentes
2. Registrando endpoints e plugins de API.

A primeira tarefa é realizada simplesmente invocando o Kong com uma *flag* especial.

O segundo é executando um *script* de configuração. Seu único objetivo é registrar *endpoints* no Kong, tal como:

```
#create a service
curl  -sS -X PUT \
--url ${kong}/services/data-broker \
--data "name=data-broker" \
--data "url=http://data-broker:80"

#create a route to service
curl  -sS -X PUT \
--url ${kong}/services/data-broker/routes/data-broker_route \
--data 'paths=["/device/(.*)/latest", "/subscription"]' \
--data "strip_path=false"
```

Este comando registrará o *endpoint /dispositivo/*/latest* e */subscription* e todas as solicitações serão encaminhadas para *http//data-broker:80*. Você pode verificar a documentação sobre como adicionar *endpoints* na documentação do Kong em *Componentes e APIs*.

Para alguns dos *endpoints registrados*, o *script* adicionará dois *plugins* aos *endpoints* selecionados:

1. Geração JWT. A documentação para este *plugin* está disponível na Kong JWT plugin page.

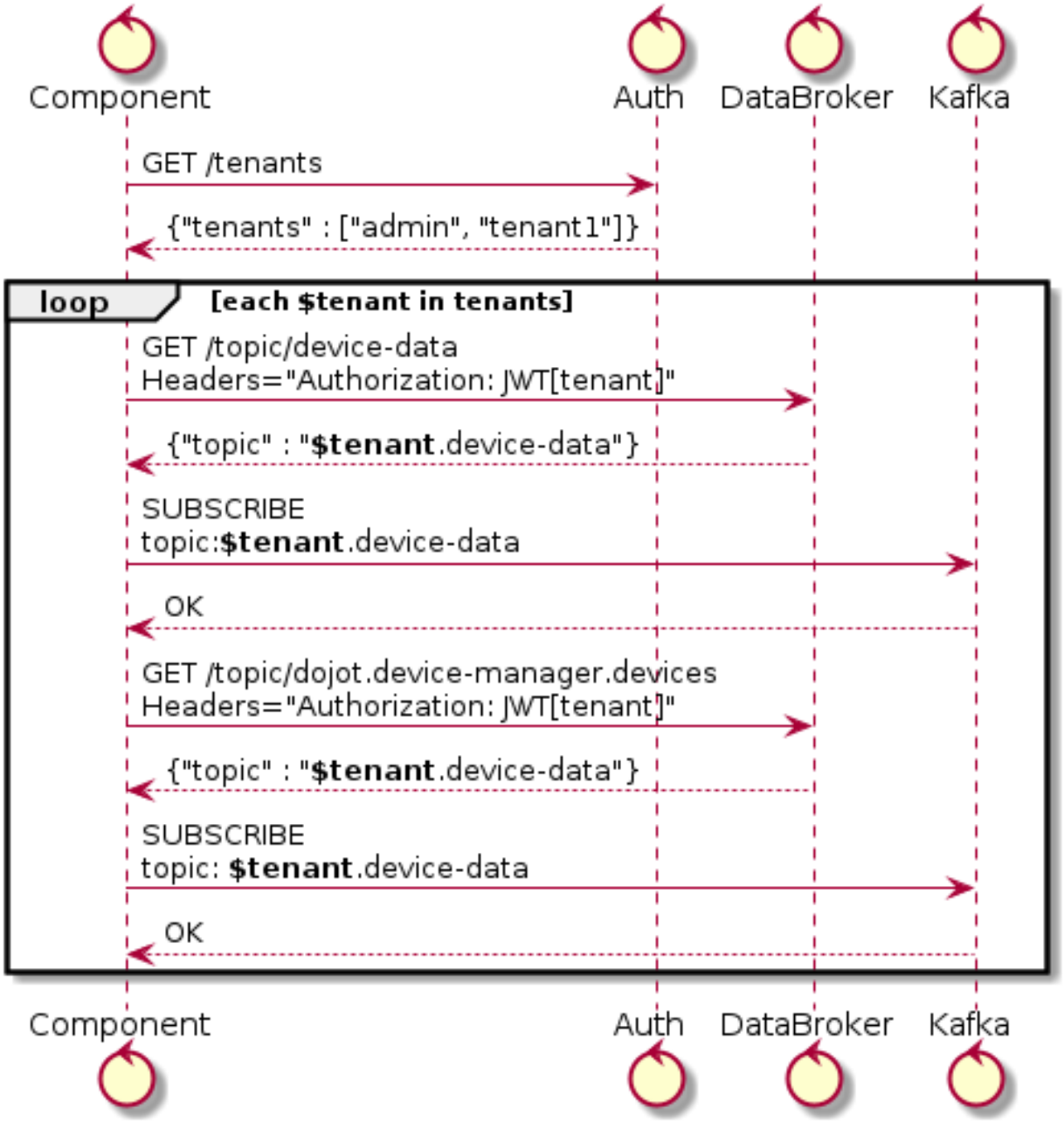


Fig. 5.5: Inicialização do *tenants* no início

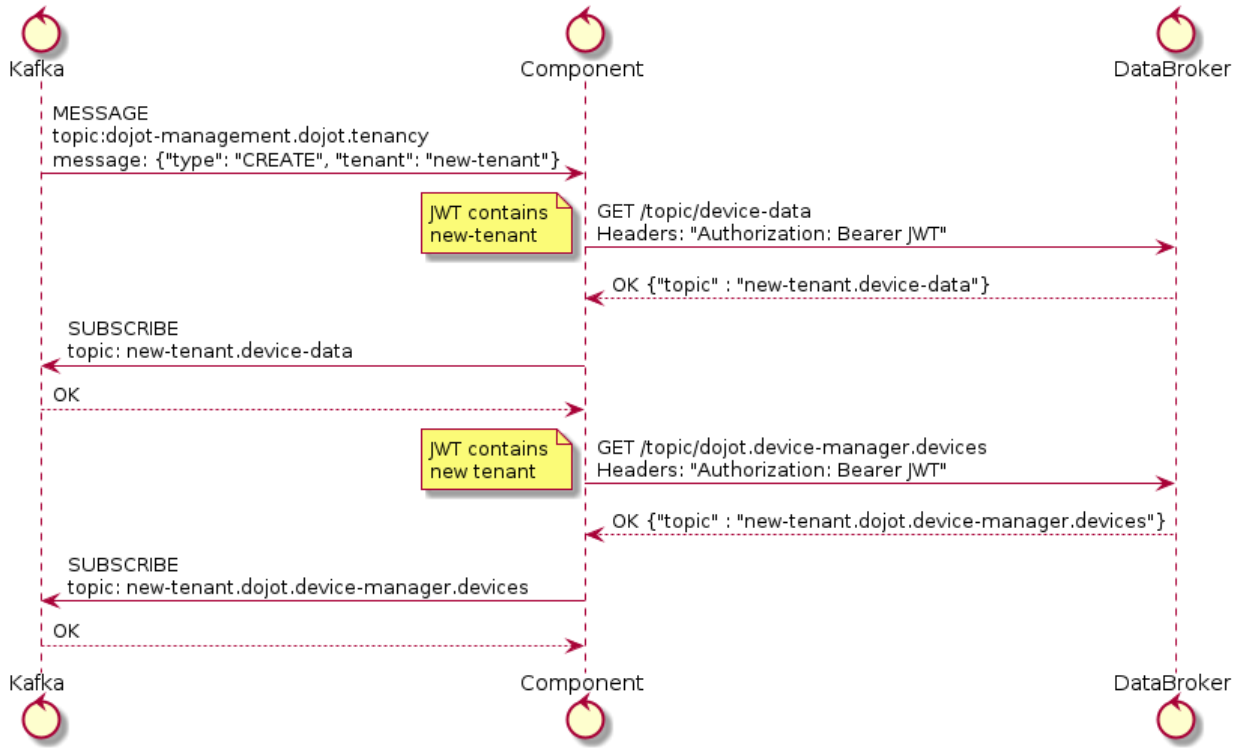


Fig. 5.6: Inicialização do *tenant*

2. Configura um *plugin* que encaminhará todas as solicitações para o Auth para autenticar solicitações. Este *plugin* está disponível dentro do Kong repository.

A solicitação a seguir instala esses dois *plugins* na API do data-broker:

```
#pepkong - auth
curl  -sS  -X POST \
--url ${kong}/services/data-broker/plugins/ \
--data "name=pepkong" \
--data "config.pdpUrl=http://auth:5000/pdp"

#JWT generation
curl  -sS  -X POST \
--url ${kong}/services/data-broker/plugins/ \
--data "name=jwt"
```

Mensagens emitidas

O Auth emitirá apenas uma mensagem via Kafka para a criação do *tenant*:

```
{
  "type" : "CREATE",
  "tenant" : "XYZ"
}
```

E uma para exclusão do *tenant*:

```
{
  "type" : "DELETE",
  "tenant" : "XYZ"
}
```

Por padrão, essas mensagens são criadas no tópico `dojot-management.dojot.tenancy` do Kafka.

Este prefixo do tópico pode ser configurado, verifique a documentação do componente`Auth` em *Componentes e APIs*.

5.4 Device Manager

O DeviceManager armazena e recupera modelos de informações para dispositivos e modelos e algumas informações estáticas sobre eles também. Sempre que um dispositivo é criado, removido ou apenas editado, ele publica uma mensagem no Kafka. Depende apenas do DataBroker e Kafka pelos motivos já explicados neste documento.

A documentação do *DeviceManager* no GitHub ReadMe explica com mais detalhes todas as mensagens publicadas. Você pode encontrar o link em: *Componentes e APIs*.

5.5 Agente IoT

Os agentes de IoT recebem mensagens de dispositivos e os convertem em uma mensagem padrão a ser publicada no Kafka. Para fazer isso, eles podem querer saber quais dispositivos são criados para filtrar corretamente as mensagens que não são permitidas na dojot (usando, por exemplo, informações de segurança para bloquear mensagens de dispositivos não autorizados). Ele usará o *subject* `device-data` e a inicialização de *tenants*, conforme descrito em *Inicialização dos tenants*.

Após solicitar os tópicos para todos os *tenants* no *subject device-data*, o agente IoT começará a receber dados dos dispositivos. Como há várias maneiras pelas quais os dispositivos podem fazer isso, esta etapa não será detalhada nesta seção (isso depende muito de como cada agente de IoT funciona). No entanto, ele deve enviar uma mensagem para Kafka para informar outros componentes de todos os novos dados que o dispositivo acabou de enviar. Isso é mostrado na Fig. 5.7, neste caso, estamos usando o *tenant admin*.

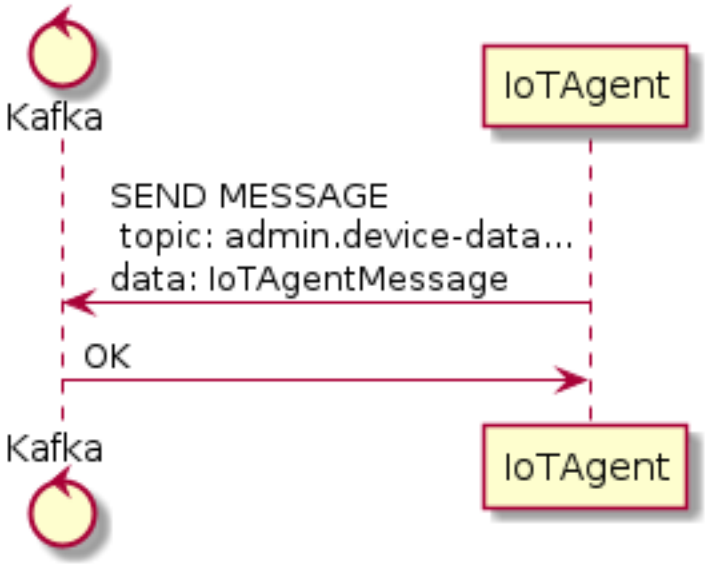


Fig. 5.7: Mensagem do agente de IoT para Kafka

Os dados enviados pelo agente de IoT têm a estrutura mostrada na Fig. 5.8.

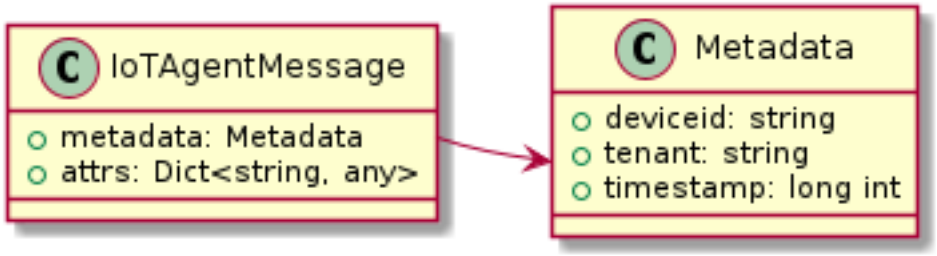


Fig. 5.8: Estrutura de mensagens do agente IoT

Essa mensagem seria:

```
{
    "metadata": {
        "deviceid": "c6ea4b",
        "tenant": "admin",
        "timestamp": 1528226137452
    },
    "attrs": {
        "humidity": 60,
        "temperature" : 23
    }
}
```

5.6 Persister

Persister é um serviço muito simples, cujo único objetivo é receber mensagens dos dispositivos (usando o *subject* `device-data`) e armazená-las no MongoDB. Para isso, é realizado o procedimento de inicialização (detalhado em *Bootstrapping tenants*) e, sempre que uma nova mensagem é recebida, ele cria um novo documento Mongo e o armazena na coleção do dispositivo.

Este serviço é simples, pois é por design.

5.7 History

O History também é um serviço muito simples: sempre que um usuário ou aplicativo envia uma solicitação, ele consulta o MongoDB e cria uma mensagem adequada para enviar de volta ao usuário/aplicativo. Isso é mostrado na Fig. 5.10.

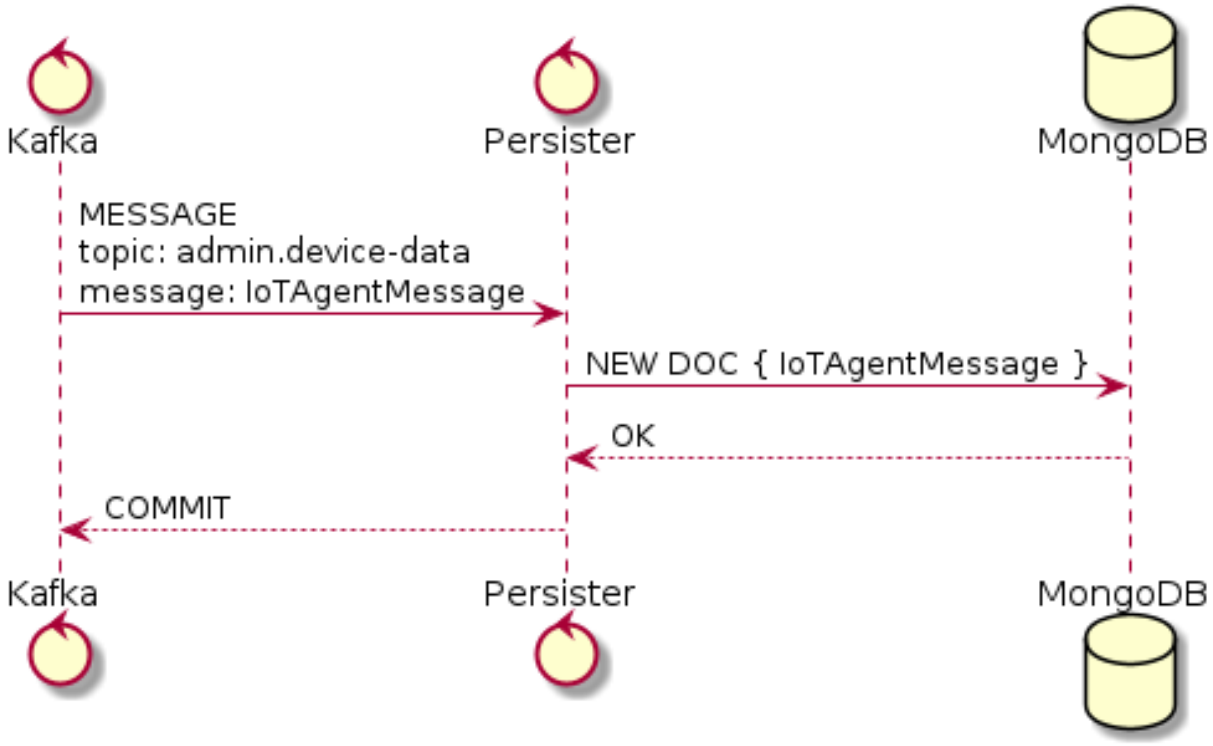


Fig. 5.9: Persister

5.8 Data Broker

O DataBroker possui algumas funcionalidades a mais do que apenas gerar tópicos para pares `{tenant, subject}`. Ele também servirá conexões socket.io para emitir mensagens em tempo real. Para fazer isso, ele recupera todos os tópicos para o *subject device-data*, assim como em qualquer outro componente interessado nos dados recebidos dos dispositivos. Assim que receber uma mensagem, ela será encaminhada para uma 'sala' (usando o vocabulário do socket.io) associada ao dispositivo e ao *tenant* associado. Portanto, todo cliente conectado a ele (como interfaces gráficas de usuário) receberão uma nova mensagem contendo todos os dados recebidos. Para obter mais informações sobre como abrir uma conexão socket.io com o DataBroker, consulte a documentação da *API* do DataBroker em *Componentes e APIs*.

Nota: As conexões socket.io em tempo real via Data Broker serão descontinuadas em versões futuras. Use o *Kafka WS* ao invés dele.

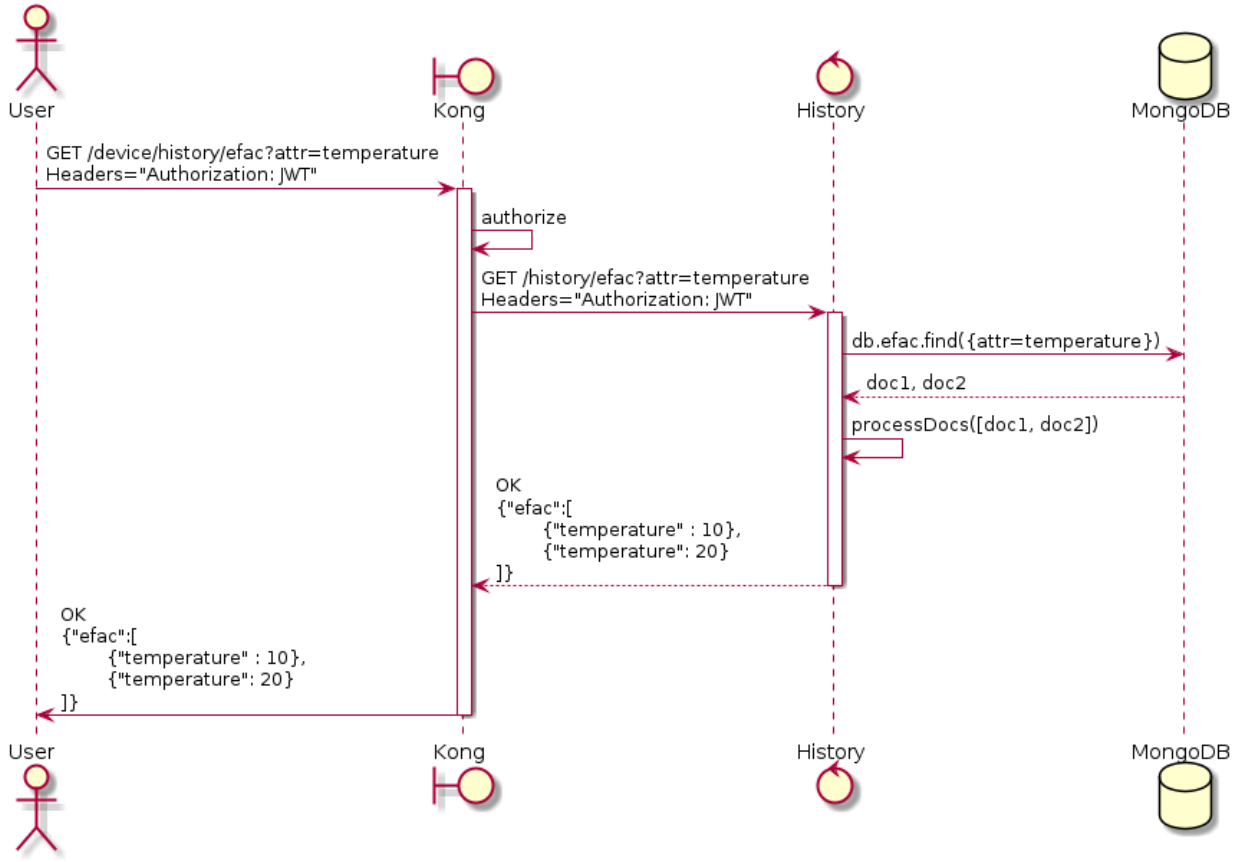


Fig. 5.10: History

5.9 Autoridade Certificadora

A plataforma dojot possui internamente uma autoridade certificadora (CA) capaz de emitir certificados x.509 para que os dispositivos possam se comunicar com a plataforma através de um canal seguro (usando o protocolo TLS). Ao requisitar um certificado para a plataforma, é necessário informar um CSR, o qual passará por uma série de validações até chegar na Autoridade Certificadora interna, que por sua vez, se todas as verificações passarem com sucesso, assinará um certificado e vinculará este certificado ao registro do dispositivo. O componente *x509-identity-mgmt* é responsável por oferecer os serviços relacionados a certificados para dispositivos.

5.10 Kafka WS

O serviço *Kafka WS* permite que os usuários recuperem dados condicionais e/ou parciais em tempo real de um determinado tópico da dojot em um Cluster Kafka. Ele funciona com conexões puras de websocket, para que se possam criar clientes websocket em qualquer linguagem desejada, desde que eles suportem RFC 6455.

5.10.1 Conectando com o serviço

A conexão é realizada em dois passos: primeiro é obtido um *ticket* de uso único via requisição HTTP, e depois o cliente se conecta ao serviço via websocket passando-o como parâmetro.

Primeiro passo: obter um ticket de uso único

Um ticket permite o usúario se subscrever em um tópico da dojot. Para obter o ticket é necessário ter um token JWT gerado pelo serviço de Autenticação/Autorização da plataforma. A requisição HTTP para obter um ticket deve ser realizada usando o verbo GET para o endpoint *<base-url>/kafka-ws/v1/ticket*. A requisição deve ter o cabeçalho *Authorization* com o token JWT obtido anteriormente como valor. Exemplo:

GET <base-url>/kafka-ws/v1/ticket
Authorization: Bearer [Encoded JWT]

O componente responde com a seguinte sintaxe:

HTTP/1.1 200 OK
Content-type: application/json

```
{
  "ticket": "[an opaque ticket of 64 hexadecimal characters]"
}
```

Nota: No contexto de um deployment da dojot, o token é providenciado pelo serviço de Autenticação e é validado pelo Gateway API antes de redirecionar a conexão para o *Kafka WS*. Portanto, nenhuma validação é feita pelo Kafka WS.

Segundo passo: Estabelecer a conexão websocket

A conexão é feita via websocket pura, usando a URI *<base-url>/kafka-ws/v1/topics/:topic*. Você deve passar o ticket gerado anteriormente como parâmetro para esta URI. Também é possivel passar opções e filtros condicionais como parâmetros para esta URI.

5.10.2 Comportamento ao solicitar um ticket e uma conexão websocket

Abaixo podemos entender o comportamento do serviço Kafka WS quando um usuário (por meio de um user agent) solicita um ticket para estabelecer uma comunicação via websocket com Kafka WS.

Observe que quando o usuário solicita um novo ticket, o Kafka WS extrai algumas informações do *token de acesso do usuário* (JWT) e gera um *payload assinado*, para ser usado posteriormente na decisão de autorizar (ou não) a conexão via websocket. A partir do payload, é gerado um *ticket* e os dois são armazenados no Redis, onde o ticket é a chave para obter o payload. Um TTL é definido pelo Kafka WS, então o usuário deve usar o ticket dentro do tempo estabelecido, caso contrário, o Redis apaga automaticamente o ticket e o payload.

Após obter o ticket, o usuário faz uma solicitação HTTP ao Kafka WS requisitando um *upgrade* de protocolo para se comunicar via *websocket*. Como a especificação dessa solicitação HTTP limita o uso de cabeçalhos adicionais, é necessário enviar o ticket pela URL, para que possa ser validado pelo Kafka WS antes de autorizar o upgrade.

Dado que o ticket esteja válido, ou seja, corresponde a uma entrada no Redis, o Kafka WS recupera o payload relacionado ao ticket, verifica sua integridade e exclui essa entrada no Redis para que o ticket não possa ser usado novamente.

Com o payload é possível tomar a decisão de autorizar ou não o *upgrade* para websocket. Se a autorização for concedida, o Kafka WS abre um canal de subscrição com base em um tópico específico no Kafka. A partir daí, o *upgrade* para websocket é estabelecido e o usuário começa a receber os dados à medida que vão sendo publicados no Kafka.

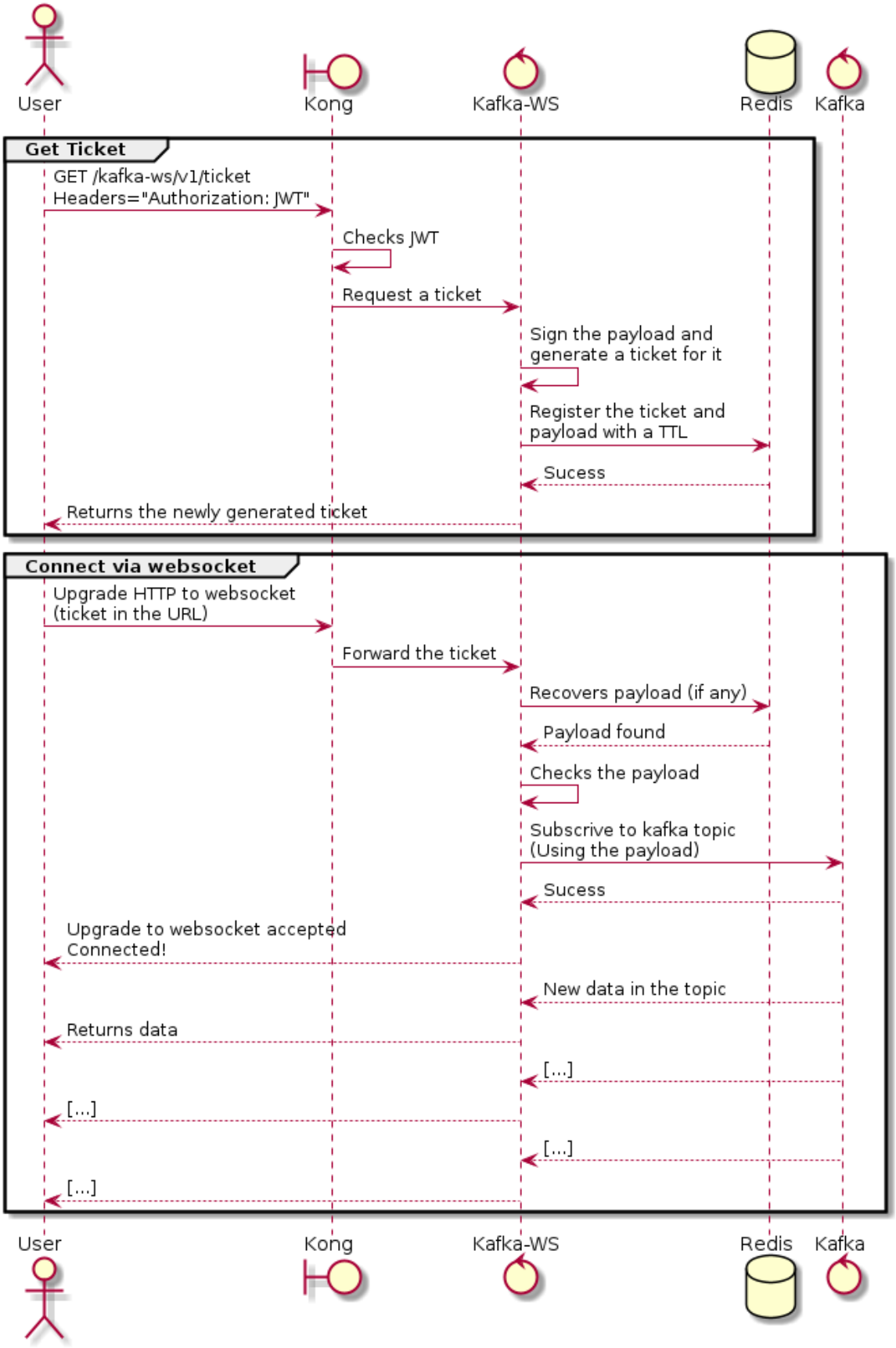


Fig. 5.11: Obtenção de ticket e conexão via websocket

CAPÍTULO 6

Guia de instalação

Esta página contém informação de como instalar a dojot utilizando o Docker Compose e o Kubernetes.

Table of Contents

6.1 Requisitos de hardware

Ao escolher as configurações de hardware para a implantação da Dojot , deve-se considerar o número de dispositivos e o intervalo de mensagens com os quais eles serão configurados. Por exemplo, os requisitos de hardware estimados para 500 dispositivos com um intervalo de mensagens a cada 15s são:

Tabela 6.1: Requisitos de hardware para 500 dispositivos

Implantação da dojot		CPU	RAM	Espaço livre em disco
Docker Compose		4 Núcleos	6GB	20GB
Kubernetes	Master	2 Núcleos	2GB	30GB
Kubernetes	Worker	4 Núcleos	6GB	40GB
Kubernetes	Balancer	1 Núcleo	1GB	10GB

Além disso, são necessários:

- **Acesso à rede**
- **As seguintes portas devem estar abertas para a instalação com Docker Compose:**
  - **TCP**: 8000 *(acesso a interface web)*; 1883 *(MQTT, se você irá usar MQTT)*; 5896 *(LWM2M, se você irá usar o serviço de arquivos LWM2M por HTTP ao invés do protocolo coap, UDP)*.
  - **TLS**: 8883 *(MQTTS, se você irá usar MQTT com TLS, no modo seguro)*.
  - **UDP**: 5683 e 5693 *(LWM2M, se você irá usar LWM2M)*; 5684 e 5694 *(LWM2M, se você irá usar LWM2M com DTLS)*.
- **As seguintes portas devem estar abertas no balanceador de carga para a instalação com Kubernetes:**
  - **TCP**: 80 *(acesso a interface web)*; 1883 *(MQTT, se você irá usar MQTT)*; 5896 *(LWM2M, se você irá usar o serviço de arquivos LWM2M por HTTP ao invés do protocolo coap, UDP)*.
  - **TLS**: 8883 *(MQTTS, se você irá usar MQTT com TLS, no modo seguro)*.
  - **UDP**: 5683 e 5693 *(LWM2M, se você irá usar LWM2M)*; 5684 e 5694 *(LWM2M, se você irá usar LWM2M com DTLS)*.

Nota: Os núcles acima são de aproximadamente 3.5 GHz (x86-64)

6.2 Docker Compose

Este documento contém instruções de como criar um ambiente para instalação trivial da dojot em um único host utilizando o Compose como o processo de orquestração da plataforma.

Muito simples, esta opção de instalação é a que melhor se adapta para desenvolvimento e verificação da plataforma dojot, mas não é aconselhável para ambientes de produção.

Este guia foi verificado utilizando-se o sistema operacional Ubuntu 18.04 LTS.

As seções seguintes descrevem todas as dependências do Docker Compose.

6.2.1 Docker Engine (motor do Docker)

Este guia foi verificado utilizando o Docker Engine na versão 19.03

Informações atualizadas e procedimentos de instalação para o Docker Engine podem ser encontrados na documentação do projeto:

https://docs.docker.com/engine/install/ubuntu/

Nota: Um passo adicional no processo de instalação e configuração do Docker em um determinado equipamento é definir quem será elegível para criar/iniciar instâncias do Docker.

Caso os passos pós-instalação não tiverem sido executados (mais especificamente o "Manage Docker como usuário não-root"), todos os comandos do Docker e do Compose devem ser executados pelo super usuário (root), ou invocando o sudo.

https://docs.docker.com/engine/installation/linux/linux-postinstall/

6.2.2 Docker Compose

Este guia foi verificado utilizando o Compose na versão 1.27

Informações atualizadas sobre procedimentos de instalação para o Compose podem ser encontradas na documentação do projeto:

https://docs.docker.com/compose/install/

6.2.3 Instalação

Para construir o ambiente, simplesmente clone o repositório e execute os comandos abaixo.

O repositório com os scripts de instalação e configuração do Docker Compose pode ser encontrados em:

https://github.com/dojot/docker-compose

ou com o comando git clone:

```
git clone https://github.com/dojot/docker-compose.git
# Let's move into the repo - all commands in this page should be executed
# inside it.
cd docker-compose
```

Uma vez que o repositório esteja propriamente clonado, selecione a versão a ser utilizada por meio da tag apropriada (note que o tag_name deve ser substituído):

```
# Must be run from within the deployment repo

git checkout tag_name -b branch_name
```

Por exemplo:

```
git checkout v0.7.0 -b v0.7.0
```

Nota: Para obter um guia sobre como usar **HTTPS**, acesse este link: https://github.com/dojot/docker-compose/tree/v0.7.0#how-to-secure-dojot-with-nginx-and-lets-encrypt

---

Atenção: Antes de executar o comando abaixo, é necessário definir seu domínio ou IP no arquivo `.env` na variável `DOJOT_DOMAIN_NAME`.

Feito isso, o ambiente pode ser iniciado assim:

```
# Must be run from the root of the deployment repo.
# May need sudo to work: sudo docker-compose up -d
docker-compose up -d
```

Nota: Para ficar completamente pronto, **saudável**, todos os serviços neste *docker-compose* levam em média pelo menos 12 minutos.

Para verificar o estado de um container individual, comandos do Docker podem ser utilizados, como por exemplo:

```
# Shows the list of currently running containers, along with individual info
docker ps

# Shows the list of all configured containers, along with individual info
docker ps -a
```

Nota: Todos os comandos para Docker e Docker Compose podem requerer credenciais de super usuário (root) ou sudo.

Para permitir usuários "não-root" gerenciar o Docker, confira a documentação do Docker:

https://docs.docker.com/engine/installation/linux/linux-postinstall/

6.2.4 Volumes

Quando fazemos o deploy da dojot com o comando 'docker-compose up -d', por padrão, os volumes são habilitados e criados.

Os volumes dos microsserviços que a dojot utiliza podem ser incompatíveis entre as versões da dojot. Isso significa que não é possível usar os volumes da dojot v0.4.x na dojot v0.5.x ou acima e vice versa.

Para usar diferentes versões da dojot no mesmo ambiente, você primeiramente deve derrubar os volumes da outra versão.

Nota: Se você derrubar os volumes da dojot, você perderá totos os dados coletados na plataforma até o momento.

Para derrubar os volumes basta passar o parâmetro '-v' no comando 'docker-compose down' como exibido abaixo:

```
docker-compose down -v
```

Dessa forma, a dojot e os volumes serão derrubados e você será capaz de fazer deploy de uma versão diferente da dojot.

6.2.5 Utilização

A interface web está disponível em `http://localhost:8000`. O usuário é `admin` e a senha é `admin`. Você também pode interagir com a plataforma utilizando o *Componentes e APIs*.

Atenção: Sempre altere a senha do usuário "admin" para uma senha adequada e mantenha-a segura.

Leia o *Utilizando a API da dojot* e *Usando a interface WEB* para maiores informações sobre como interagir com a plataforma dojot.

6.3 Kubernetes

Para uma instalação simples com kubernetes por favor, verifique o pdf abaixo.

`Clique aqui para acessar o guia de instalação da dojot com kubernetes`

Se você deseja instalar uma Dojot mais robusta que suporte até 100k dispositivos, verifique o pdf abaixo.

Nota: No ambiente 100k, a dojot não processa e nem armazena as mensagens enviadas pelos dispositivos. Esse ambiente só irá funcionar para testes de carga e apenas alguns componentes da dojot estarão disponíveis.

`Clique aqui para acessar o guia de instalação da dojot 100k com kubernetes`

Nota: Infelizmente ainda não possuímos suporte para língua inglesa nesse tutorial.

CAPÍTULO 7

Dúvidas Mais Frequentes

Aqui estão algumas respostas às dúvidas mais frequentes sobre a plataforma dojot.

Não encontrou aqui uma resposta para a sua dúvida? Por favor, abra uma *issue* no repositório da dojot no Github.

Sumário

7.1 Gerais

7.1.1 O que é a dojot? Por que eu deveria utilizá-la? Por que abrir o código?

É uma plataforma brasileira para IoT que surgiu com uma proposta de código aberto, para facilitar o desenvolvimento de soluções e o ecossistema IoT com conteúdo local voltado às necessidades brasileiras.

O dojot atua como uma plataforma facilitadora com:

- APIs abertas tornando fácil o acesso das aplicações aos recursos da plataforma.
- Capacidade de armazenamento de grandes volumes de dados em diferentes formatos.
- Conectores para diferentes tipos de dispositivos.
- Construção de fluxos de dados e regras de forma visual, permitindo a rápida prototipação e validação de cenários de aplicações IoT.
- Processamento de eventos em tempo real aplicando regras definidas pelo desenvolvedor.

7.1.2 Onde eu posso baixar?

Todos os componentes estão disponíveis no repositório da dojot no GitHub: https://github.com/dojot.

7.1.3 Qual é o principal repositório?

Existem 3 repositórios principais:

- https://github.com/dojot/dojot: é aqui que concentramos o acompanhamento de tudo relacionado a este projeto como decisões, melhorias e aqui também existem códigos de alguns componentes.
- https://github.com/dojot/docker-compose: para executar dojot em um ambiente docker-compose usando a ferramenta *docker-compose*. Este é o repositório que recomendamos para começar com dojot.
- https://github.com/dojot/ansible-dojot: repositório com os arquivos e configurações para executar a dojot em ambiente *Kubernetes*.

Veja como usar os repositórios *docker-compose* e *ansible-dojot* em *Guia de instalação*.

7.1.4 Então, encontrei um probleminha chato. Como posso informá-los sobre isso?

Não encontrou aqui uma resposta para a sua dúvida? Por favor, abra uma *issue* no repositório da dojot no Github.

Se você puder analisar e resolver o problema, por favor faça isso e crie um *pull-request* com uma breve descrição do que foi feito.

7.2 Uso

7.2.1 Por onde eu começo? É baseado em CLI ou possui uma interface gráfica?

A dojot pode ser acessada via Interface Web ou APIs REST. Considerando que você já tenha instalado o `docker` e o `docker-compose` e tenha clonado o repositório `docker-compose` da dojot, para iniciar todos os serviços, basta executar o comando abaixo:

```
$ docker-compose up -d
```

E é isto.

A interface Web está disponível em `http://localhost:8000`. O usuário é `admin` e a senha é `admin`.

APIs REST são explicadas na seção *Aplicações*.

7.2.2 Pronto, já iniciei e fiz o *login*. E agora?

Legal! Agora você pode criar seus primeiros modelos (templates) e dispositivos, descrito em *Dispositivos*, criar alguns fluxos e registrar-se para eventos de dispositivos, ambos descritos em *Fluxos de Dados*.

7.2.3 Como posso atualizar o meu ambiente com a última versão da dojot?

Basta seguir alguns passos:

1 Atualize o repositório do *docker compose* com a última versão.

```
$ cd <path-to-your-clone-of-docker-compose>
$ git checkout master && git pull
```

Se você precisar de uma outra versão, você pode fazer *checkout* de uma *tag*:

```
$ git tag
  0.1.0-dojot
  0.1.0-dojot-RC1
  0.1.0-dojot-RC2
  0.2.0
  v0.3.0-beta.1
  v0.3.1
  v0.4.0
  v0.4.1
  v0.4.1_rc2
  v0.4.2
  v0.4.2-rc.1
  v0.4.3
  v0.4.3-rc.1
  v0.4.3-rc.2
  v0.5.0-alpha.1
  v0.5.0-alpha.2
  v0.5.0-alpha.3
  v0.5.0-alpha.4
  v0.5.0-rc.1
  v0.5.0
  v0.5.1
  v0.5.2
  v0.6.0
  v0.7.0


$ git checkout v0.7.0
```

7.3 Dispositivos

7.3.1 O que são dispositivos para a dojot?

Na dojot, um dispositivo é uma representação digital para um dispositivo real ou *gateway* com um ou mais sensores ou uma representação para um dispositivo virtual com sensores/atributos inferidos de outros dispositivos.

Considere, por exemplo, um dispositivo real com um termômetro e um higrômetro; ele pode ser representado na dojot como um dispositivo com dois atributos (um para cada sensor). Chamamos este tipo de dispositivo de *dispositivo normal* ou usando o seu protocolo de comunicação, como *dispositivo MQTT* ou *dispositivo CoAP*.

Nós também podemos criar dispositivos que não correspondem diretamente a dispositivos reais, por exemplo, podemos criar um dispositivo com informação em alto nível de temperatura (*está ficando mais quente* ou *está ficando mais frio*) cujos valores são inferidos a partir de sensores de temperatura de outros dispositivos. Este tipo de dispositivo é denominado de *dispositivo virtual*.

7.3.2 Qual é a relação entre este dispositivo e um dispositivo real?

É simples como parece: o *dispositivo* para a dojot é um espelho (gêmeo/cópia digital) do dispositivo real. Você pode escolher quais atributos são disponibilizados para as aplicações e outros componentes, adicionando cada um deles através da interface de criação de dispositivo.

7.3.3 O que são *dispositivos virtuais*? Como se diferenciam dos demais?

Dispositivos são criados para serem como espelhos (gêmeo/cópia digital) dos dispositivos e sensores reais. Um *dispositivo virtual* é uma abstração que modela coisas que não são factíveis no mundo real. Por exemplo, digamos que um usuário tenha alguns sensores para detectar fumaça em um laboratório, sendo que cada um tem diferentes atributos.

Não seria bom se existisse um dispositivo chamado *Laboratório* que possui um atributo *emChamas*? Assim a aplicação dependeria apenas deste atributo para tomar alguma ação.

Uma outra diferença é a maneira como os dados dos dispositivos virtuais são populados. Os dispositivos são preenchidos com informações enviadas a plataforma dojot por dispositivos ou *gateways* e os virtuais são preenchidos por fluxos ou por aplicações.

7.3.4 E o que são modelos (*templates*)?

Modelos (*templates*) são "modelos para dispositivos" que servem como base para a criação de um novo dispositivo. Um dispositivo é construído usando um conjunto de templates - seus atributos serão herdados de cada template (não deve haver nenhum atributo com mesmo nome, no entanto). Se um modelo é alterado, todos os dispositivos associados àquele modelo serão também alterados automaticamente.

7.3.5 Como posso enviar dados via MQTT para a dojot de forma que apareçam no *dashboard*?

Primeiramente, crie uma representação digital para o seu dispositivo real. Depois, configure o seu dispositivo real para enviar dados para a dojot de maneira que os dados possam ser associados ao seu representante digital.

Tomemos como exemplo uma estação meteorológica que monitora temperatura e umidade e publica essas medidas via MQTT periodicamente. Inicialmente, cria-se um dispositivo do tipo MQTT com dois atributos (temperatura e umidade) e, em seguida, configura-se a estação meteorológica para publicar os dados para a dojot.

> **Atenção:** A partir da versão **v0.5.2**, você pode escolher entre dois *brokers* MQTT: Mosca ou VerneMQ. Por padrão, o VerneMQ é utilizado, mas você pode utilizar o Mosca também. Verifique o *Guia de instalação* para mais informações.

Para enviar dados para a dojot via MQTT (usando o Mosca ou VerneMQ), existem alguns pontos a se ter em mente:

- Ao usar o Mosca, o tópico deve seguir o formato `/<tenant>/<device-id>/attrs` (e.g.: `/admin/efac/attrs`). Dependendo de como o IoT agent MQTT foi inicializado (mais restritivo), o client ID também deve ser configurado para `<tenant>:<device-id>`, como `admin:efac`.
- Ao utilizar o VerneMQ, o tópico deve seguir o formato `<tenant>:<device-id>/attrs` (e.g.: `admin:efac/attrs`). Além disso, você deve passar o *username* do cliente no formato `<tenant>:<device-id>`, como `admin:efac`, sendo que este deve ser igual à mesma parte passada no tópico. Você também pode passar o ID do cliente (opcional).
- O *payload* MQTT precisa ser um JSON com cada chave sendo um atributo de um dispositivo definido na dojot, como:

```
{ "temperature" : 10.5, "pressure" : 770 }
```

Você pode usar certificados com MQTT, veja *Usando MQTT com segurança (TLS)* para mais informações.

7.3.6 No *dashboard* alguns atributos são exibidos como tabelas e outros como gráficos. Como são escolhidos/configurados?

O tipo do atributo determina o modo de exibição dos dados no *dashboard* como segue:

- `Geo`: mapa georreferenciado.
- `Boolean`, `Text` e `JSON`: tabela.
- `Integer` e `Float`: gráfico de linha.

7.3.7 Estou interessado em integrar à dojot o meu dispositivo que é super legal. Como eu faço isso?

Se o seu dispositivo envia mensagens via MQTT (com *payload* do tipo JSON), CoAP ou HTTP, existe uma grande chance de ser possível integrá-lo com mínima ou nenhuma modificação.

7.3.8 Existem restrições para as mensagens enviadas pelo meu dispositivo para a dojot? Formato, tamanho, frequência?

Nenhuma com exceção do formato, que é descrito na questão *Como posso enviar dados via MQTT para a dojot de forma que apareçam no dashboard?*.

7.3.9 Como posso enviar comandos para o meu dispositivo através da dojot?

Por enquanto, você pode enviar requisições HTTP para a dojot contendo algumas instruções sobre qual dispositivo deve ser configurado e os dados para atuação. Mais detalhes podem ser encontrados em *Enviando mensagens*.

7.3.10 Não encontrei o protocolo suportado pelo meu dispositivo na lista de tipos, existe algo que eu possa fazer?

Existem algumas possibilidades. A primeira é desenvolver um *proxy* para traduzir o seu protocolo para um dos suportados pela dojot. A segunda é desenvolver um agente IoT, conector, similar aos existentes para MQTT, CoAP e HTTP. Dê uma olhada em https://github.com/dojot/iotagent-nodejs.

7.3.11 Como eu posso obter dados históricos para um dispositivo em particular?

Você pode fazer isso enviando uma requisição para */history/*, como:

```
curl -X GET \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsIn...' \
  "http://localhost:8000/history/device/3bb9/history?lastN=3&attr=temperature"
```

o qual retornará as últimas 3 entradas do atributo *temperature* do dispositivo *3bb9*:

```
[
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:47:07.050000Z",
    "value": 29.76,
    "attr": "temperature"
  },
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:42.455000Z",
    "value": 23.76,
    "attr": "temperature"
  },
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:21.535000Z",
    "value": 25.76,
    "attr": "temperature"
  }
]
```


Para mais detalhes sobre a recuperação de dados do histórico, verifique o tutorial em *Conferindo dados históricos*.

Há mais operadores que podem ser usados para filtrar entradas. Veja a API do History em *Componentes e APIs* para ver todos os possíveis operadores e filtros.

7.4 Fluxos de Dados

7.4.1 O que é um fluxo?

É um processamento de mensagens de um dispositivo. Com um fluxo, você pode analisar dinamicamente cada nova mensagem para fazer validações, inferir informações e tomar ações ou gerar notificações.

7.4.2 A interface dos fluxos... ela se parece com o node-RED. Eles tem alguma relação?

A interface dos fluxos é baseada no node-RED, mas a aplicação das regras e execução das ações é feita por um mecanismo próprio da dojot. Se o node-RED for familiar para você, não será difícil usar o *flowbroker*.

7.4.3 Por que eu deveria usar um fluxo?

Ele permite uma das coisas mais interessantes do IoT de uma forma simples e intuitiva que é analisar dados para extração de informações e execução de ações.

7.4.4 O que ele pode fazer, exatamente?

Você pode fazer coisas como:

- Criar visões para um dispositivo (renomear atributos, agregá-los, alterá-los, etc.)
- Inferir informações baseadas em regras de detecção de borda e georreferenciamento.
- Notificar de várias maneiras, como HTTP.

O componente responsável pelo fluxo de dados está em desenvolvimento constante e novas funcionalidades são adicionadas a cada versão.

Há mecanismos para adicionar novos blocos de processamento a fluxos. Veja *Como eu posso adicionar um novo tipo de nó no menu?* para mais informações sobre isto.

7.4.5 Pois bem, como eu posso usá-lo?

Há um tutorial para como utilizar a fluxo, veja mais em *Usando o construtor de fluxos (Flowbroker)*.

7.4.6 Posso aplicar o mesmo fluxo para múltiplos dispositivos?

Você pode usar um modelo como entrada para indicar que ele deve ser executado para todos os dispositivos associados àquele modelo (*template*). É válido indicar que o fluxo é sempre executado para cada mensagem.

7.4.7 Posso correlacionar dados de diferentes dispositivos no mesmo fluxo?

Uma vez que os fluxos são aplicados individualmente para cada mensagem, você deve criar um dispositivo virtual para agregar todos os atributos e então usar este dispositivo como entrada de um novo fluxo.

Você também pode criar um nó (ou usar um já existente) que lida com contextos.

7.4.8 E se eu quiser enviar um HTTP *POST*?

Um aviso importante: assegure-se de que a dojot consegue acessar seu servidor.

7.4.9 Eu quero renomear os atributos de um dispositivo. O que eu devo fazer?

Primeiramente, você deve criar um dispositivo virtual com os novos atributos, para então construir um fluxo de dados para renomeá-los. Isto pode ser feito conectando um nó 'change' após um dispositivo de entrada para mapear os atributos de entrada a seus correspondentes na saída.

7.4.10 Quero agregar os atributos de múltiplos dispositivos. O que eu devo fazer?

Inicialmente, você deve criar um dispositivo virtual para agregar todos os atributos. Com este dispositivo criado, você deve criar fluxos para mapeamento dos atributos de cada dispositivo real de entrada neste dispositivo virtual. Isto pode ser feito em nós 'change' conectados a cada um dos dispositivos de entrada a fim de criar uma variável contendo todos os atributos de saída. Todos os nós change devem ser, por fim, conectados ao nó de saída representando o dispositivo virtual.

7.4.11 Como eu posso adicionar um novo tipo de nó no menu?

Há um tutorial sobre como adicionar novos nós e dois exemplos de nós também, verifique flowbroker library para mais detalhes.

7.5 Aplicações

7.5.1 Quais APIs estão disponíveis para aplicações?

Você pode ver todas as APIs disponíveis na página *Componentes e APIs*..

7.5.2 Como posso usá-los?

Há um tutorial simples e rápido em *Utilizando a API da dojot*.

7.5.3 Estou interessado(a) em integrar minha aplicação com a dojot. O que devo fazer?

Isto deve ser bastante direto. Há três formas de integrar sua aplicação à dojot:

- **Obtenção de dados históricos**: você pode querer ler todos os dados históricos relacionados a um dispositivo de forma periódica. Isto pode se feito usando a API do *history*
- **Recuperando dados em tempo real**: você pode querer ler dados em tempo real relacionados ao dispositivo. Isso pode ser feito usando o *Kafka WS*, uma implementação baseada em *websocket*. Para entender melhor como usar *Kafka WS* confira *Kafka WS*.
- **Usando o flowbroker para pré-processar dados**: eles podem ajudar a processar e transformar os dados para que possam ser enviados corretamente para o seu aplicativo por meio de solicitação HTTP ou armazenados em um dispositivo virtual (que pode ser usado para gerar notificações conforme descrito anteriormente).

Todos esses *endpoints* devem ter um *token* de acesso. Para isso, confira *Obtendo um token de acesso*

Verifique a documentação da API do *History* e do *Kafka WS* em *Componentes e APIs*. E para um tutorial de como usar o fluxo veja *Usando o construtor de fluxos (Flowbroker)*.

CAPÍTULO 8

Copyright e licença

A plataforma dojot IoT é baseada em softwares de código aberto conhecidos como Apache Kafka, RabbitMQ, PostgreSQL, MongoDB, Redis, Kong Gateway, and VerneMQ, que têm suas próprias licenças.

Os serviços desenvolvidos pela equipe dojot para integrar esses componentes de software de código aberto e implementar as lógicas de negócios são protegidos por direitos autorais pelo CPQD e até o lançamento v0.5.1 foram licenciados sob o GPLv3. A partir do lançamento v0.5.2 em diante, esses serviços estão sendo licenciados sob a Apache License, Version 2.0.

Você pode usar e/ou modificar o software gratuitamente (sem custos de licença).

CAPÍTULO 9

Histórico de lançamento

9.1 Full Contact - 2021.07

9.1.1 Serviços

Novos Serviços

Certificate ACL

- O certificate-acl é responsável por manter na memória uma associação entre a *fingerprint* dos certificados e seus proprietários para que os serviços dojot que precisam dessas informações possam consultá-la em vez do serviço x509-identity-mgmt, que mantém essas informações apenas no disco.

Cert-sidecar

- O Cert-Sidecar, certificate sidecar, é um utilitário de serviço para gerenciar certificados do x509-identity-mgmt para uso com conexões TLS.

Melhorias e correções

IotAgent MQTT VerneMQ

- Integração com Cert-sidecar
- Integração com Certificate ACL

V2k-Bridge

- Integração com Cert-sidecar

K2v-Bridge

- Integração com Cert-sidecar

Kakfa-ws

- Melhorias na verificação de *health check*

X.509 Identity Management

- Suporte para certificados externos

Outros

- Correção de pequenos bugs

9.1.2 Implantações

Docker-compose

- Correção de pequenos bugs

Ansible-dojot

- Melhorias na documentação
- Recurso de volumes
- Recurso de *Labels*
- Rotação de arquivos de log do Docker
- Atualiza a versão do Kubernetes para 1.19.8
- Correção de pequenos bugs

CAPÍTULO 10

Usando a interface WEB

Este tutorial descreve as operações básicas na dojot, como criar dispositivos, conferir seus atributos e criar fluxos, importar/exportar, atualização de firmware e gerar certificados e relatório de histórico do dispositivo

Nota:

- Para quem é esse tutorial: usuários iniciantes
- Nível: básico
- Tempo de leitura: 30 minutos

10.1 Gerenciamento de dispositivo

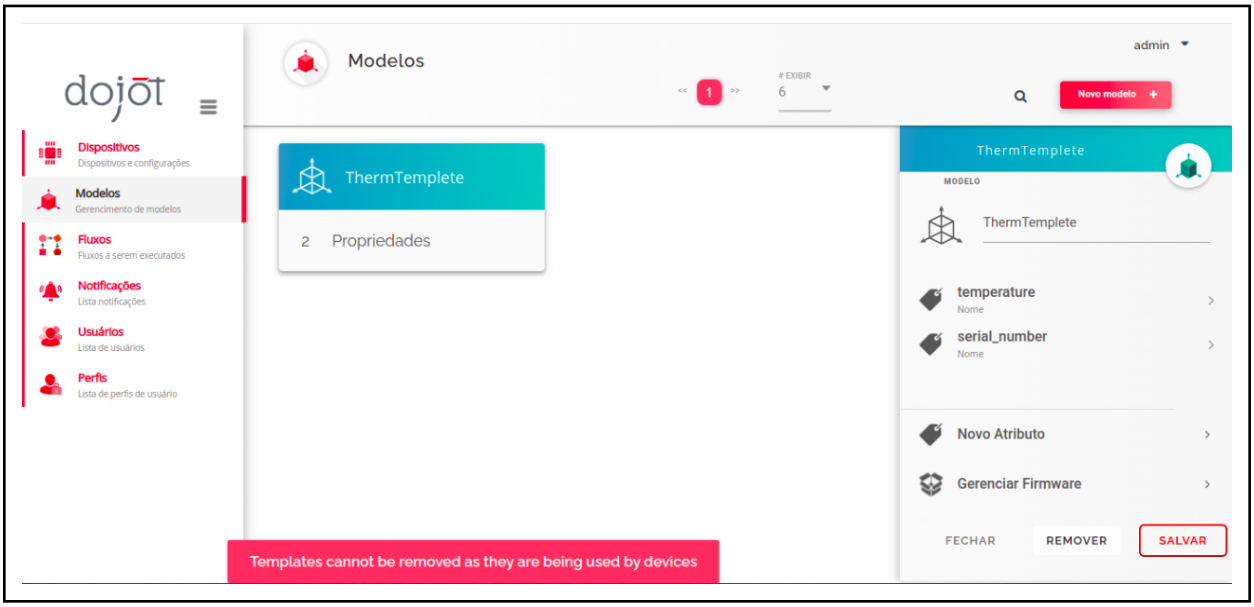
Esta seção mostra como gerenciar dispositivos. Para tal, serão utilizados dois dispositivos sensores de temperatura e um dispositivo virtual, esse último com a função de observar as temperaturas medidas nos dois primeiros e gerar alarmes em determinadas condições.

Como descrito em *Conceitos*, todos os dispositivos são baseados em um ou mais modelos (templates). Para a criação de um modelo, você deve acessar a opção Modelos (Templates) na lateral esquerda da tela e então criar um Novo Modelo (New Template), como mostrado abaixo.

Para criar novos dispositivos, deve-se voltar para a opção Dispositivos (Devices) e criar um Novo Dispositivo (New Device), selecionando os modelos nos quais o dispositivo será baseado, como mostrado abaixo.

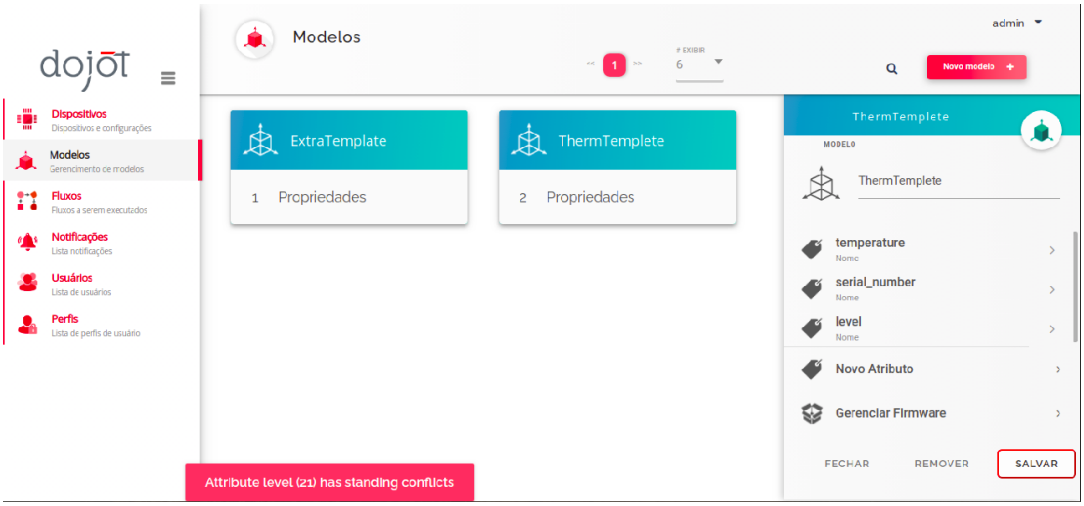
Note que, quando um modelo é selecionado no painel direito da tela de criação de dispositivo, todos os atributos são herdados para aquele dispositivo. É possível adicionar mais de um modelo, tendo em mente que modelos que compõem o dispositivo não podem compartilhar atributos com o mesmo nome.

Atenção: Como os dispositivos estão fortemente associados aos modelos, caso queira remover um modelo, primeiramente dave-se remover todos os seus dispositivos associados. Caso isso ocorra, a seguinte mensagem de erro aparecerá:



Atenção:

Você pode adicionar e remover atributos dos modelos, fazendo com que as alterações sejam imediatamente refletidas nos dispositivos associados. No caso de novos atributos serem adicionados, no entanto, deve-se observar que os atributos dos modelos que compõem um determinado dispositivo não podem possuir o mesmo nome. Se isso acontecer, a seguinte mensagem aparecerá:



Essa imagem da tela foi capturada quando um novo modelo foi criado (`ExtraTemplate`) com um atributo chamado `level`. Depois um novo dispositivo baseado em ambos os modelos foi criado e um novo atributo também chamado `level` foi adicionado ao modelo `ThermTemplate`.

Quando isso ocorre, nenhuma modificação é aplicada ao modelo (nenhum atributo com nome "level" relativo ao "ThermTemplate" é criado). Contudo, o atributo é mantido nas configurações do modelo (template) para que o usuário perceba o que está acontecendo. Se o usuário atualizar a tela, as informações serão revertidas para o estado que estava antes da modificação.

Agora os dispositivos físicos podem enviar mensagens para dojot. Há alguns pontos de atenção: o tópico MQTT deve

estar no formato `<tenant>:<device-id>/attrs`.

Por questão de simplicidade, será emulado um dispositivo utilizando-se a ferramenta `mosquito_pub`. O parâmetro `username` será configurado utilizando a opção `-u` do mosquito_pub. Veja mais sobre em *Utilizando a API da dojot* no tópico Enviandos Mensagens.

Atenção: Nos vídeos, nós estamos utilizando o Mosca como broker MQTT, então as mensagens estão sendo enviadas para tópicos no formato antigo. Se você está utilizando o VerneMQ, mude-os de acordo com o formato supracitado. Verifique a página de *Dúvidas Mais Frequentes* para maiores informações.

Nota: Os exemplos estão utilizando MQTT sem segurança. A abordagem recomendada é utilizar TLS. Verifique a página *Usando MQTT com segurança (TLS)*.

Estando criados os sensores de temperatura, falta agora a criação do dispositivo virtual. Ele será a representação de um alarme de sistema disparado quando algo ruim for detectado pelos sensores. Por exemplo, se os sensores de temperatura estivessem instalados em uma cozinha, a medição de uma temperatura acima de 40ºC poderia indicar que o local estaria em chamas. Essa representação do alarme poderia ter dois atributos: nível de severidade e mensagem textual, para que o usuário pudesse ser informado do acontecimento.

Assim como "dispositivos regulares", dispositivos virtuais também são baseados em modelos. Portanto, um modelo será criado, como mostrado abaixo.

10.2 Configuração de fluxo

Uma vez criado o dispositivo virtual, pode-se adicionar um fluxo para implementar a lógica por detrás da geração de alarmes. A ideia é: se a temperatura medida for menor ou igual a 40ºC, o sistema de alarmes será atualizado com uma mensagem de severidade 4 (média) e uma mensagem indicando que a cozinha está OK. Caso a temperatura medida seja maior que os 40ºC, uma mensagem de severidade 1 (muito alta) será enviada com a mensagem que a cozinha está em chamas. Isto é feito como mostrado abaixo.

É importante notar que os nós do tipo "change" têm uma referência a uma entidade "output". Isso pode ser visto como uma simples estrutura de dados, onde existem os atributos `message` e `severity` que casam com aqueles do dispositivo virtual. Este "objeto" é referenciado no nó de saída (output) como uma fonte de dados para o dispositivo que será atualizado (nessa caso, o dispositivo virtual criado). Em outras palavras, pode-se dizer que há uma informação que é transferida dos nós do tipo "change" para o "dispositivo virtual" com os nomes "msg.output.message" e "msg.output.severity", onde "message" e "severity" são atributos do dispositivo virtual.

Vamos, agora, enviar mais algumas mensagens e ver o que acontece para aquele dispositivo virtual.

10.3 Importar e Exportar

Essa seção mostra como usar as funcionalidades Importar e Exportar. Tais opções permitem que seus dados de configuração sejam salvos em um arquivo, no caso de uma exportação, e carregados na dojot, no caso de Importação. Esse arquivo possui o formato *JSON* e contém os dados de modelos, dispositivos, fluxos, nós remotos e tarefas de agendamento que foram cadastrados do seu *tenant*. Para executar o procedimento de exportação dos dados de configurações, expanda o menu no canto superior direito da página, clique em "Import/Export (Import./Exportação)" e então, em "Export (Exportar)", conforme ilustrado no vídeo a seguir:

O arquivo exportado pode ser armazenado como backup e posteriormente importado novamente na Dojot.

Para executar o procedimento de importação de uma configuração, expanda o menu no canto superior direito da página, clique em "Import/Export (Import./Exportação)" e então, em "Import (Importar)". Na janela que aparece é possível arrastar e soltar seu arquivo ou navegar na pasta de destino e selecioná-lo, sendo só permitido adicionar um arquivo de extensão *JSON*, no formato esperado, conforme ilustrado no vídeo a seguir:

Atenção: Ao executar o procedimento de importação todas as configurações atuais do tenant tais como: dispositivos, modelos, fluxos, nós remotos e tarefas de agendamento, serão excluídas permanentemente para que as novas sejam criadas. Os dados de histórico não fazem parte do importar e exportar!

10.4 Atualização de Firmware

Durante a vida útil de um dispositivo, pode ser necessário atualizar seu software de controle (firmware) a fim de corrigir eventuais problemas encontrados durante seu uso ou até mesmo adicionar novas funcionalidades. Atualmente a dojot suporta o procedimento de atualização de firmware via o protocolo de comunicação LwM2M. Para saber detalhes a respeito do procedimento para integração com seu dispositivo por favor verifique a especificação do protocolo LwM2M. Caso seu dispositivo se comunica via este protocolo e possui o procedimento de atualização de firmware implementado, você pode seguir os passos a seguir para atualizar a versão do seu dispositivo.

O processo de atualização de firmware é composto por três etapas:

- gerenciamento das imagens
- transferência da imagem para o dispositivo;
- aplicação da imagem no dispositivo

O detalhamento da execução destas encontra-se a seguir.

Para que a habilitação de gerenciamento de firmware seja disponibilizada, é preciso criar um modelo e, depois de salvo, habilitar o gerenciador de firmware. Após isso, é possível enviar as imagens de firmware para o repositório da dojot que são associadas a este modelo. Atenção: o nome do arquivo da imagem deve possuir a extensão ".hex".

Note que assim que o Gerenciador de Firmware é habilitado, são atribuídos ao modelo cinco atributos que são usados para suportar a atualização de imagens. Os nomes dos atributos podem ser editados conforme a necessidade da aplicação. Os atributos são:

- Estado do dispositivo (Device State):
  - Estado atual da atualização de firmware
- Resultado da versão de aplicação (Result of apply version)
  - Resultado da última aplicação de atualização de uma imagem de firmware;
- Define qual versão transferir (Sets which version to transfer):
  - Indica ao agente IoT, responsável pelo dispositivo, qual deve ser o nome e a versão da imagem de firmware a ser transferida e atualizada no dispositivo
- Acionar atualização da versão (Trigger version update)
  - Atuador utilizado para iniciar o procedimento de atualização de firmware
- Versão atual da imagem (Current version of the image):
  - Versão atual da imagem de firmware, caso disponibilizado pelo mesmo

Após criar o modelo com a opção de gerenciamento de firmware habilitada, é preciso associá-lo a um dispositivo. Assim é possível transferir uma imagem e aplicá-la no dispositivo, conforme o vídeo abaixo:

Observe que em cada etapa, o status e o resultado do processamento da imagem são mostrados.

10.5 Gerando certificados para dispositivos

Nesta seção será demonstrado como gerar certificados x509 para um dispositivo, para que a dojot consiga se comunicar com os dispositivos de forma segura via TLS. Para conseguir realizar o envio de uma publicação é necessário o download dos três arquivos, sendo eles a chave privada "admin 4302d4.key", o certificado do dispositivo "admin 4302d4.crt" e o certificado da CA "ca.crt".

> **Atenção:** A geração de certificados via interface gráfica (GUI) só funciona em deployments em que é possível acessar a GUI via *HTTPS* ou *localhost*.

Para conseguir executar o comando do vídeo de exemplo é necessário estar no mesmo diretório que os 3 arquivos estão. Comando utilizado no exemplo:

```
mosquitto_pub -h localhost -p 8883 -t admin: 4302d4/attrs -m '{"humidity": 7}' --cert
↪"admin 4302d4.crt" --key "admin 4302d4.key" --cafile ca.crt
```

10.6 Gerando relatório de histórico de dispositivos

Nesta seção será demonstrado como gerar um relatório de histórico de um dispositivo. O relatório consegue mostrar os dados de um ou mais atributos do respectivo dispositivo. Para isso, é necessário selecionar os atributos desejados, definir o período e clicar em "gerar".

10.7 Realizando acesso ao Dashboard

O dashboard faz parte da GUI-V2, para acessá-lo é necessário utilizar outra URL, na versão atual é só adicionar ao final da URL "/v2" em relação a interface utilizada nos itens anteriores, por exemplo, no caso do localhost seria http://localhost:8000/v2, veja mais em *Componentes e APIs*. As credenciais de login e senha são as mesmas utilizadas no restante da dojot. Após se logar, uma nova tela abrirá e aparecerá um botão `ADICIONAR` no canto superior direito, no qual dará as opções de vários tipos de visualização. Neste momento na primeira tela "Geral" será necessário adicionar um nome e opcionalmente uma descrição para a visualização.nome para visualização e opcionalmente uma descrição. Na próxima tela aparecerá uma lista de dispositivos, caso não encontre o dispositivo desejado, você poderá pesquisar pelo nome. Após selecionar o dispositivo, serão listados os atributos que estão vinculados a ele, podendo ser escolhida uma cor para apresentação de cada atributo, é possível ainda adicionar uma legenda para cada atributo. Em "recuperar registros por:" será possível configurar alguns filtros, você pode selecionar o tipo de filtro de dados de histórico como os "últimos registros", por "ordem"(minuto, horas dias e meses), e também poderá escolher um intervalo de tempo, além disso você ainda poderá visualizar os registros em "Tempo real", conforme novos dados forem sendo recebidos eles irão ser exibidos na visualização escolhida. Depois de tudo configurado, será mostrado um resumo com o nome e os atributos escolhidos. Ao acessar as visualizações, será possível alterar o tamanho, corrigir (o que desabilitará a opção de alterar o tamanho) e ainda excluir as visualizações.

CAPÍTULO 11

Utilizando a API da dojot

Esta seção descreve o passo a passo completo de como criar, alterar, enviar mensagens e conferir dados históricos relativos a um dispositivo. Este tutorial assume que está sendo utilizada a instalação docker-compose e que todos os componentes necessários estão sendo executados corretamente na dojot.

Nota:

- Audiência: desenvolvedores
- Nível: básico
- Tempo de leitura: 15 minutos

11.1 Pré-requisitos

Vamos utilizar:

- curl para acessar as *APIs* da plataforma dojot;
- jq para processar o retorno *JSON* das *APIs* da plataforma dojot.
- mosquitto publicar e se subscrever no iotagent-mosca (or iotagent-mqtt) via MQTT.

Em distribuições Linux baseadas em Debian, você pode executar:

```
sudo apt-get install curl
sudo apt-get install jq
sudo apt-get install mosquitto-clients
```

11.2 Obtendo um *token* de acesso

Todas as requisições devem conter um *token* de acesso que seja válido. É possível gerar um novo *token* enviando a seguinte requisição:

```
JWT=$(curl -s -X POST http://localhost:8000/auth \
-H 'Content-Type:application/json' \
-d '{"username": "admin", "passwd" : "admin"}' | jq -r ".jwt")
```

Checar:

```
echo $JWT
```

Se o intuito for gerar um *token* para outro usuário, é necessário somente mudar o *username* e *passwd* no corpo da requisição. O token ("eyJ0eXAiOiJKV1QiL...") deve ser usado em toda a requisição HTTP enviada para a dojot, colocando-o no cabeçalho da mensagem. A requisição seria algo desse tipo:

```
curl -X GET http://localhost:8000/device \
  -H "Authorization: Bearer eyJ0eXAiOiJKV1QiL..."
```

É importante ressaltar que o token deve estar inteiro no cabeçalho da requisição, não apenas parte dele. No exemplo, somente os primeiros caracteres foram mostrados por questão de simplificação. Todas as demais requisições serão compostas da variável de ambiente chamada `bash ${JWT}` que contém o token obtido da dojot (mais especificamente do componente de autorização da dojot).

11.3 Criação de dispositivo

A fim de configurar um dispositivo físico na dojot, é necessário criar sua representação na plataforma. O exemplo mostrado aqui é apenas uma parte pequena do que é oferecido pelo componente DeviceManager. Para mais informações sobre esse componente, confira a documentação do DeviceManager.

Primeiramente vamos criar um modelo (*template*) para o dispositivo, pois todos os dispositivos são baseados em modelos, não esqueça.

```
curl -X POST http://localhost:8000/template \
-H "Authorization: Bearer ${JWT}" \
-H 'Content-Type:application/json' \
-d ' {
  "label": "Thermometer Template",
  "attrs": [
    {
      "label": "temperature",
      "type": "dynamic",
      "value_type": "float"
    },
    {
      "label": "fan",
      "type": "actuator",
      "value_type": "float"
    }
  ]
}'
```

Esta requisição deve retornar a seguinte mensagem:

```
1  {
2    "result": "ok",
3    "template": {
4      "created": "2018-01-25T12:30:42.164695+00:00",
5      "data_attrs": [
6        {
7          "template_id": "1",
8          "created": "2018-01-25T12:30:42.167126+00:00",
9          "label": "temperature",
10         "value_type": "float",
11         "type": "dynamic",
12         "id": 1
13       }
14     ],
15     "label": "Thermometer Template",
16     "config_attrs": [],
17     "attrs": [
18       {
19         "template_id": "1",
20         "created": "2018-01-25T12:30:42.167126+00:00",
21         "label": "temperature",
22         "value_type": "float",
23         "type": "dynamic",
24         "id": 1
25       },
26       {
27         "template_id": "1",
28         "created": "2018-01-25T12:30:42.167126+00:00",
29         "label": "fan",
30         "type": "actuator",
31         "value_type": "float",
32         "id": 2
33       }
34     ],
35     "id": 1
36   }
37 }
```

Note que o *template* (modelo) ID é 1 (linha 35), caso você já tenha criado algum outro *template* este ID será diferente.

Para criar um dispositivo baseado nesse modelo (ID 1), envie a seguinte requisição para a dojot:

```
curl -X POST http://localhost:8000/device \
-H "Authorization: Bearer ${JWT}" \
-H 'Content-Type:application/json' \
-d ' {
  "templates": [
    "1"
  ],
  "label": "device"
}'
```

A lista de IDs de modelos na linha 6 contém um único ID do modelo configurado até o momento. Para conferir os dispositivos configurados, basta enviar uma requisição do tipo GET para /device:

```
curl -X GET http://localhost:8000/device -H "Authorization: Bearer ${JWT}"
```

Que deve retornar:

```
{
  "pagination": {
    "has_next": false,
    "next_page": null,
    "total": 1,
    "page": 1
  },
  "devices": [
    {
      "templates": [
        1
      ],
      "created": "2018-01-25T12:36:29.353958+00:00",
      "attrs": {
        "1": [
          {
            "template_id": "1",
            "created": "2018-01-25T12:30:42.167126+00:00",
            "label": "temperature",
            "value_type": "float",
            "type": "dynamic",
            "id": 1
          },
          {
            "template_id": "1",
            "created": "2018-01-25T12:30:42.167126+00:00",
            "label": "fan",
            "value_type": "actuator",
            "type": "float",
            "id": 2
          }
        ]
      },
      "id": "0998", # <-- this is the device-id
      "label": "device_0"
    }
  ]
}
```

O *device id* utilizado nos próximos passos deve ser alterado conforme o retornado na criação do device. Na execução acima o *id* retornado foi *0998* na linha 34. Portanto, todos os lugares nos próximos passos com referencias a *0998* devem ser alterados.

11.4 Enviando mensagens

Até o momento um token de acesso foi obtido, um modelo (*template*) e um dispositivo (baseado no modelo) foram criados. Em um sistema real, o dispositivo físico publica mensagens para a dojot com todos os seus atributos contendo valores correntes. Nesse tutorial serão publicadas mensagens MQTT montadas "na mão" para a plataforma, emulando um dispositivo físico. Para tal, será utilizado o *mosquito_pub* e *mosquitto_sub* do projeto mosquitto.

O formato padrão de mensagem usado pela dojot é um simples "chave-valor" JSON (é possível traduzir qualquer formato para esse esquema utilizando fluxos), como abaixo:

```
{
  "temperature" : 10.6
```

(continua na próxima página)

(continuação da página anterior)

```
}
```

Atenção: Algumas distribuições Linux, como as baseadas em Debian, têm dois pacotes para mosquitto - um contendo ferramentas para cliente (ou seja, mosquitto_pub e mosquitto_sub para publicar mensagens e se subscrever em tópicos) e outro contendo um *broker* MQTT também. Neste tutorial, **apenas as ferramentas do pacote `mosquitto-clients` em Distribuições Linux baseadas no Debian serão usadas**. Verifique se um outro *broker* MQTT **não está em execução** antes de iniciar a dojot (executando comandos como `ps aux | grep mosquitto`) para evitar conflitos de porta.

Por simplicidade, nós não estamos utilizando TLS nos exemplos abaixo. Verifique o artigo *Usando MQTT com segurança (TLS)* para maiores esclarecimentos sobre sua utilização.

Nota: Para executar *mosquitto_pub* e` mosquitto_sub` sem usar TLS como nos exemplos abaixo, você precisa definir algumas configurações (ou para como desativar o modo sem TLS). Para obter mais detalhes sobre este tópico, consulte a página *Modo inseguro do MQTT (sem TLS)*.

A partir da versão **v0.5.2**, você pode escolher entre dois brokers MQTT: Mosca ou VerneMQ. Por padrão, o VerneMQ é utilizado, mas você pode utilizar o Mosca também. Verifique o *Guia de instalação* para mais informações.

11.4.1 Usando o VerneMQ

Como descrito no *Dúvidas Mais Frequentes*, existem algumas considerações a respeito dos tópicos MQTT:

- Você deve adicionar o `username` que originou a mensagem usando o atributo de mesmo nome do MQTT. Ele deve seguir o seguinte padrão: `<tenant>:<device-id>`, como `admin:efac`. Ele deve ser o mesmo que o colocado no tópico.
- O tópico para publicação deve assumir o padrão `<tenant>:<device-id>/attrs` (por exemplo: `admin:efac/attrs`).
- O tópico de subscrição deve assumir o padrão `<tenant>:<device-id>/config` (por exemplo: `admin:efac/config`).
- Os dados da mensagem MQTT (payload) devem ser um JSON com cada chave sendo um atributo do dispositivo cadastrado na dojot, como:

```
{ "temperature" : 10.5, "pressure" : 770 }
```

A requisição deve resultar na seguinte mensagem:

```
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
↪": 10.6}' -q 1
```

Se não houver saída (output), a mensagem foi enviada ao *broker* MQTT.

Repare que estamos enviando a publicação com o parâmetro `-q 1`. Isto significa que a mensagem usará *QoS* 1, i.e., é garantido pelo menos um envio da mensagem.

Além disso, você pode enviar uma mensagem de configuração da dojot para o dispositivo. O atributo de destino deve ser do tipo "actuator" ou "atuador".

Para simular o recebimento da mensagem em um dispositivo, podemos usar o `mosquitto_sub`:

```
mosquitto_sub -h localhost -p 1883 -u admin:0998 -t admin:0998/config -q 1
```

Disparando o envio da mensagem da dojot para o dispositivo.

```
curl -X PUT \
    http://localhost:8000/device/0998/actuate \
    -H "Authorization: Bearer ${JWT}" \
    -H 'Content-Type:application/json' \
    -d '{"attrs": {"fan" : 100}}'
```

11.4.2 Usando o Mosca (legado)

Como descrito no *Dúvidas Mais Frequentes*, existem algumas considerações a respeito dos tópicos MQTT:

- Pode-se configurar o ID do dispositivo origem da mensagem utilizando o parâmetro MQTT `client-id`. Deve seguir o seguinte padrão: `<service>:<deviceid>`, como em `admin:efac`.
- Se algo o impossibilita de realizar isto, então o dispositivo deve configurar seu ID no tópico utilizado para publicar as mensagens. O tópico deve assumir o padrão `/<service-id>/<device-id>/attrs` (e.g.: `/admin/efac/attrs`).
- O tópico deve assumir o padrão `/<service-id>/<device-id>/config` (por exemplo: `/admin/efac/config`).
- Os dados da mensagem MQTT (payload) devem ser um JSON com cada chave sendo um atributo do dispositivo cadastrado na dojot, como:

```
{ "temperature" : 10.5, "pressure" : 770 }
```

Atenção: VerneMQ é o novo broker MQTT padrão. O suporte ao Mosca será eventualmente retirado, então use o VerneMQ se possível!

Vamos publicar a mensagem a seguir:

```
mosquitto_pub -h localhost -t /admin/0998/attrs -p 1883 -m '{"temperature": 10.6}'
```

Se não houver saída (output), a mensagem foi enviada ao *broker* MQTT.

Além disso, você pode enviar uma mensagem de configuração da dojot para o dispositivo. O atributo de destino deve ser do tipo "actuator" ou "atuador".

Para simular o recebimento da mensagem em um dispositivo, podemos usar o `mosquitto_sub`:

```
mosquitto_sub -h localhost -p 1883 -t /admin/0998/config
```

Disparando o envio da mensagem da dojot para o dispositivo.

```
curl -X PUT \
    http://localhost:8000/device/0998/actuate \
    -H "Authorization: Bearer ${JWT}" \
    -H 'Content-Type:application/json' \
    -d '{"attrs": {"fan" : 100}}'
```

Nota: Consideraremos a utilização de VerneMQ pelo restante do tutorial.

11.5 Conferindo dados históricos

A fim de se conferir todos os valores que foram publicados pelo dispositivo para um atributo particular, pode-se utilizar a API do History, veja mais em *Componentes e APIs*. Vamos, então, enviar agora alguns outros valores à dojot para que possamos conseguir resultados um pouco mais interessantes:

```
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
↪": 36.5}' -q 1
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
↪": 15.6}' -q 1
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
↪": 10.6}' -q 1
```

Para recuperar todos os valores enviados dos atributos `temperature` desse dispositivo:

```
curl -X GET \
  -H "Authorization: Bearer ${JWT}" \
  "http://localhost:8000/history/device/0998/history?lastN=3&attr=temperature"
```

O *endpoint* do histórico é construído por meio desses valores:

- `.../device/0998/...`: o ID do dispositivo é `0998` - isso é obtido do atributo `id` do próprio dispositivo
- `.../history?lastN=3&attr=temperature`: o atributo requerido é temperature e deve ser recuperado os 3 últimos valores.

  A requisição deve resultar na seguinte mensagem:

```
[
  {
    "device_id": "0998",
    "ts": "2018-03-22T13:47:07.050000Z",
    "value": 10.6,
    "attr": "temperature"
  },
  {
    "device_id": "0998",
    "ts": "2018-03-22T13:46:42.455000Z",
    "value": 15.6,
    "attr": "temperature"
  },
  {
    "device_id": "0998",
    "ts": "2018-03-22T13:46:21.535000Z",
    "value": 36.5,
    "attr": "temperature"
  }
]
```

A mensagem acima contém todos os valores previamente enviados pelo dispositivo.

CAPÍTULO 12

Usando o construtor de fluxos (Flowbroker)

Este tutorial mostrará como usar corretamente o construtor de fluxo para processar mensagens e eventos gerados pelos dispositivos.

Nota:

- Para quem é: usuários iniciantes
- Nível: básico
- Tempo de leitura: 20 min

Índice

12.1 Nós da dojot

- *Notificação (notification)*
- *Definir valor (Change)*
- *Interruptor (Switch)*
- *Modelo (Template)*
- *Cron*
- *Cron em lote (Cron batch)*
- *Geo referência (Geofence)*
- *Obter contexto (Get Context)*
- *Mesclar dados (Merge data)*
- *Soma acumulada (Cumulative sum)*
- *Publicar no tópico FTP (Publish in FTP topic)*
- *Nós descontinuados*
  - *Device in*
  - *Device template in*
  - *Device out*
  - *Actuate*

12.1.1 Evento dispositivo - entrada (Device Event in)

Este nó especifica as mensagens recebidas de ou enviadas para um dispositivo específico. A mensagem criada por este nó é um pouco diferente daquela criada pelo nó DeviceIn.

Para configurar o dispositivo no nó, uma janela como a Fig. 12.1 será exibida.

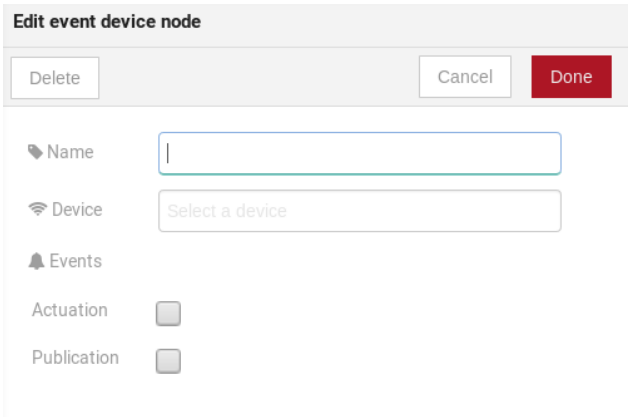


Fig. 12.1: : Dispositivo na janela de configuração

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó

- **Dispositivo (Device)** *(obrigatório)*: o dispositivo *dojot* que acionará o fluxo
- **Eventos (Events)** *(obrigatório)*: selecione quais eventos acionarão esse fluxo. A opção *Atuação (Actuation)* seleciona mensagens de atuação (aquelas enviadas ao dispositivo) e *Publicação (Publication)* seleciona todas as mensagens publicadas pelo dispositivo.

Exemplo de mensagem gerada por este nó:

Para eventos de publicação (`Publication`):

```
{
    "data": {
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "publish"
}
```

Para eventos de atuação (`Actuation`):

```
{
    "data": {
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "configure"
}
```

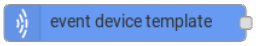
Nota: Nos eventos *Publication* e *Actuation* os atributos estaticos também estarão sempre como uma *key* no json dentro da key `data.attrs` com seus respectivos **rótulo** e **valor**

Essa estrutura pode ser referenciada em nós como *Modelo do dispositivo* e outros como:

```
Sample message {{payload.data.attrs.temperature}}
```

Nota: Se o dispositivo que aciona um fluxo for removido, o fluxo não funcionará mais.

12.1.2 Modelo de dispositivo - Eventos - entrada (event device template)



Este nó especifica que as mensagens dos dispositivos compostos por um modelo específico acionarão esse fluxo. Por exemplo, se o modelo de dispositivo definido neste nó for o modelo A, todos os dispositivos compostos com o modelo A acionarão o fluxo. Por exemplo: dispositivo1 é composto pelos modelos [A, B], dispositivo2 pelo modelo A e dispositivo3 pelo modelo B. Então, nesse cenário, apenas as mensagens do dispositivo1 e do dispositivo2 iniciarão o fluxo, porque o modelo A é um dos modelos que compõem esses dispositivos.

Para entender melhor o JSON dentro da chave *data* para os eventos` Creation`, *Update*,` Removal`, consulte a documentação do *device-manager*.

Nota: Nos eventos *Publication* e *Actuation* os atributos estaticos também estarão sempre como uma *key* no json dentro da key `data.attrs` com seus respectivos **rótulo** e **valor**

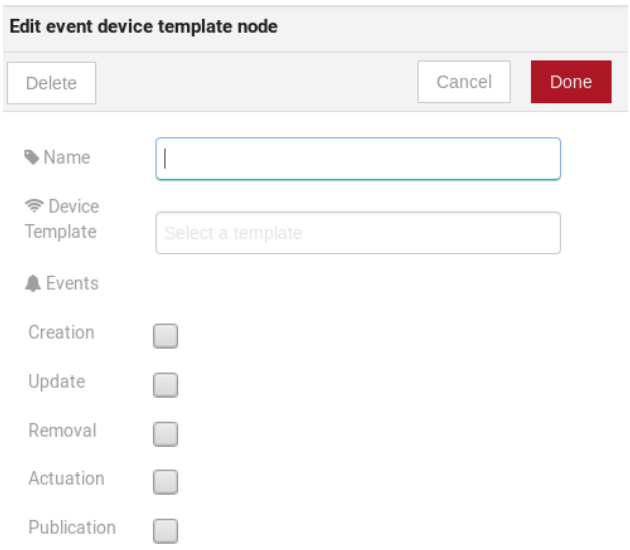


Fig. 12.2: : Modelo de dispositivo janela de configuração

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó.
- **Dispositivo (Device)** *(obrigatório)*: o dispositivo *dojot* que acionará o fluxo.
- **Eventos (Events)** *(obrigatório)*: Selecione qual evento acionará esse fluxo. A *criação (Creation)*, a *atualização (Update)* e a *remoção (Removal)* estão relacionadas às operações de gerenciamento de dispositivos. *Atuação (Actuation)* acionará esse fluxo no caso de enviar mensagens de atuação para o dispositivo e *Publicação (Publication)* acionará esse fluxo sempre que um dispositivo publicar uma mensagem para executar.

Exemplo de mensagem gerada por este nó:

Para eventos de publicação (`Publication`):

```
{
    "data": {
```

(continua na próxima página)

(continuação da página anterior)

```
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "publish"
}
```

Para eventos de atuação (`Actuation`):

```
{
    "data": {
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "configure"
}
```

Para um evento de criação (`Creation`) - criando um dispositivo usando esse modelo(*template*):

```
{
    "data": {
        "label":"template_name"
        "id":"9face8"
        "templates":["1"]
        "created":"2021-06-08T14:46:27.008321+00:00"
        "attrs": {
            "temperature": {
                    "id":1,
                    "value_type":"float",
                    "static_value":"",
                    "type":"dynamic",
                    "template_id":"1",
                    "created":"2021-06-08T14:33:24.330779+00:00",
                    "is_static_overridden":false
                }
            },
    },
    "metadata": {
        "tenant":"admin"
    },
    "event": "create"
}
```

Para um evento de atualização (`Update`) - Gerado alterando um modelo(*template*) ou adicionando um

modelo (*template*) a um dispositivo existente:

```
{
    "data": {
        "label":"template_name"
        "id":"9face8"
        "templates":["1"]
        "created":"2021-06-08T14:46:27.008321+00:00"
        "updated":"2021-06-08T14:51:16.003916+00:00"
        "attrs": {
            "temperature": {
                    "id":1,
                    "value_type":"float",
                    "static_value":"",
                    "type":"dynamic",
                    "template_id":"1",
                    "created":"2021-06-08T14:33:24.330779+00:00",
                    "is_static_overridden":false
                }
            },
    },
    "metadata": {
        "tenant":"admin"
    },
    "event": "update"
}
```

Para um evento de remoção (`Removal`) - Ao remover um modelo (*template*) de um dispositivo:

```
{
    "data": {
        "label":"template_name"
        "id":"9face8"
        "templates":["1"]
        "created":"2021-06-08T14:46:27.008321+00:00"
        "updated":"2021-06-08T14:51:16.003916+00:00"
        "attrs": {
            "temperature": {
                    "id":1,
            },
    },
    "metadata": {
        "tenant":"admin"
    },
    "event": "remove"
}
```

Essa estrutura pode ser referenciada em nós como *Modelo do dispositivo* e outros como:

```
Sample message {{payload.data.attrs.temperature}}
```

12.1.3 Saída para múltiplos dispositivos (Multi device out)

A saída do dispositivo determinará qual dispositivo (ou dispositivos) terá seus atributos atualizados na dojot de acordo com o resultado do fluxo. Lembre-se de que este nó não envia mensagens para o seu dispositivo; ele atualiza apenas os atributos na plataforma. Normalmente, o dispositivo escolhido é um dispositivo virtual, que existe apenas na dojot.

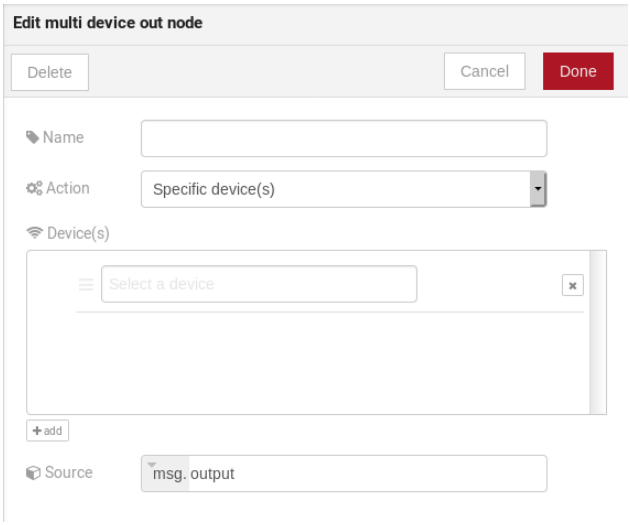


Fig. 12.3: : Janela de configuração do dispositivo

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó.
- **Ação (Action)** ***(obrigatório)*****: Qual dispositivo receberá a atualização. As opções são:**
  - O dispositivo que acionou o fluxo: isso atualizará o mesmo dispositivo que enviou a mensagem que acionou esse fluxo.
  - *Dispositivo(s) específicos (Specific device(s))*: quais dispositivo(s) que receberão a atualização.
  - *Dispositivo(s) definido(s) durante o fluxo (Device(s) defined during the flow)*: quais dispositivo(s) que receberão a atualização. Isso é referenciado por uma lista de valores, assim como os valores de saída (msg.list_of_devices).
- **Dispositivo (Device)** *(obrigatório)*: selecione "O dispositivo que acionou o fluxo" fará com que o dispositivo que foi o ponto de entrada seja o ponto final do fluxo. "Dispositivo específico" qualquer dispositivo escolhido será a saída do fluxo e "um dispositivo definido durante o fluxo" tornará um dispositivo que o fluxo selecionou durante a execução o ponto final.
- **Origem (Source)** *(obrigatório)*: estrutura de dados que será mapeada como mensagem para saída do dispositivo

12.1.4 Multi atuador (Multi actuate)

O nó de atuação é, basicamente, a mesma coisa que o nó de dispositivo (saída). Mas pode enviar mensagens para um dispositivo real, como dizer a uma lâmpada para desligar a luz e etc.

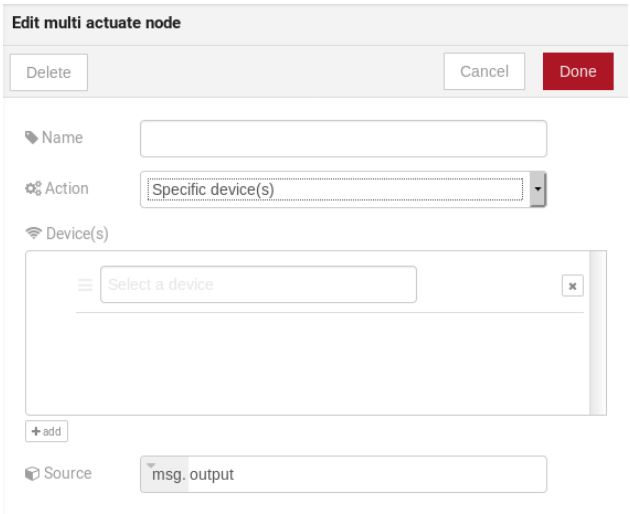
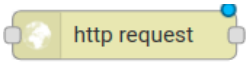


Fig. 12.4: : Configuração de atuação

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó.
- **Ação (Action)** ***(obrigatório)*****: para qual dispositivo uma mensagem será enviada. As opções são:**
  - O dispositivo que acionou o fluxo: isso enviará uma mensagem para o mesmo dispositivo que enviou a mensagem que acionou esse fluxo.
  - *Dispositivo(s) específicos (Specific device(s))*: para quais dispositivo(s) a mensagem será enviada.
  - *Dispositivos definidos durante o fluxo (Device(s) defined during the flow)*: para quais dispositivo(s) a mensagem será enviada. Isso é referenciado por uma lista de valores, assim como os valores de saída (msg.list_of_devices).
- **Dispositivo (Device)** *(obrigatório)*: selecione "O dispositivo que acionou o fluxo" fará com que o dispositivo que foi o ponto de entrada seja o ponto final do fluxo. "Dispositivo específico" qualquer dispositivo escolhido será a saída do fluxo e "um dispositivo definido durante o fluxo" tornará um dispositivo que o fluxo selecionou durante a execução o ponto final.
- **Origem (Source)** *(obrigatório)*: estrutura de dados que será mapeada como mensagem para saída do dispositivo

12.1.5 Requisição HTTP



Este nó envia uma solicitação http para um determinado endereço e, em seguida, pode encaminhar a resposta para o próximo nó no fluxo.

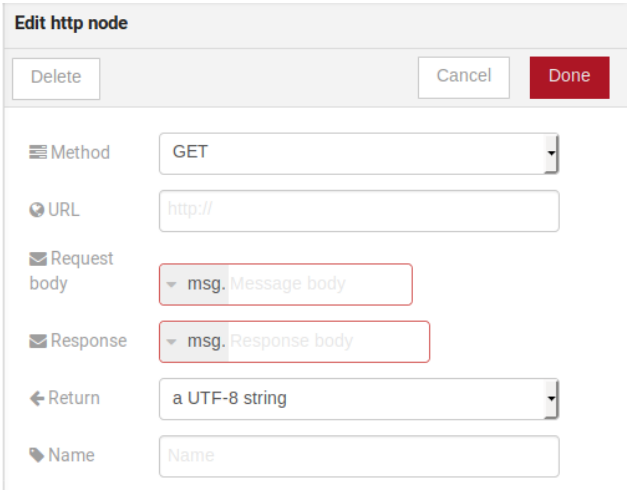


Fig. 12.5: : Configuração do nó Requisição HTTP

Campos:

- **Método (Method)** *(obrigatório)*: o método http (GET, POST, etc . . .).
- **URL** *(obrigatório)*: a URL que receberá a solicitação http
- **Corpo da solicitação (Request body)** *(obrigatório)*: variável que contém o corpo da solicitação. Este valor pode ser atribuído à variável usando o nó *modelo*, por exemplo.
- **Resposta (Response)** *(obrigatório)*: variável que receberá a resposta http.
- **Retorno (Return)** *(obrigatório)*: Tipo do retorno.
- **Nome (Name)** *(opcional)*: Nome do nó.

12.1.6 Requisição FTP

Este nó envia um arquivo para um servidor FTP. Ao fazer upload de um arquivo, seu nome pode ser definido configurando o campo "Nome do arquivo" da mesma maneira que outras variáveis de saída (ele deve se referir a uma variável definida no fluxo). A codificação do arquivo definirá o *enconding* do arquivo, que pode ser, por exemplo, "base64" ou "utf-8".

Campos:

- **Método (Method)** *(obrigatório)*: a ação do FTP a ser executada (PUT).
- **URL** *(obrigatório)*: o servidor FTP
- **Autenticação (Authentication)** *(obrigatório)*: Nome de usuário e senha para acessar este servidor.

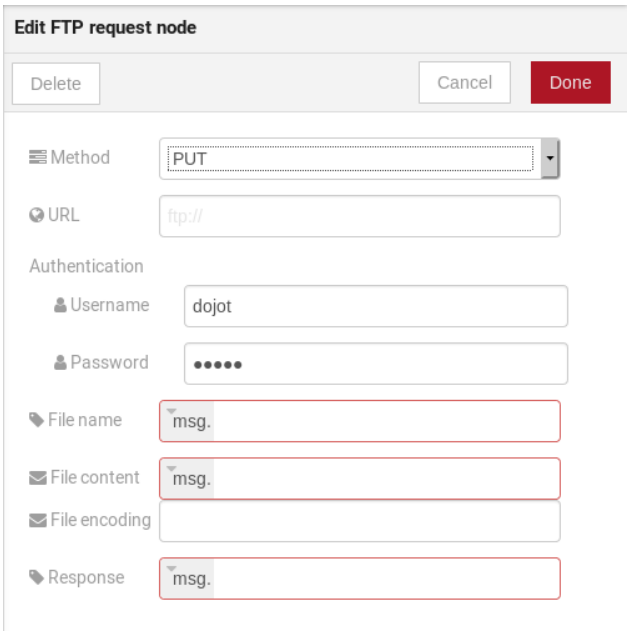


Fig. 12.6: : Modelo de dispositivo janela de configuração

- **Nome do arquivo (File name)** *(obrigatório)*: variável que contém o nome do arquivo a ser carregado.
- **Conteúdo do arquivo (File content)** *(obrigatório)*: Essa variável deve conter o conteúdo do arquivo.
- **Codificação de arquivo (File encoding)** *(obrigatório)*: como o arquivo é codificado
- **Resposta (Response)** *(obrigatório)*: variável que receberá a resposta FTP
- **Nome (Name)** *(opcional)*: Nome do nó.

12.1.7 Notificação (notification)

Este nó envia uma notificação do usuário para outros serviços na dojot. Isso pode ser útil para gerar notificações de aplicativos que podem ser consumidas por uma aplicação front-end. O usuário pode definir uma mensagem estática a ser enviada ou, como outros nós de saída, configurar um conjunto de variáveis em um nó anterior que será resolvido no tempo de execução. Além disso, os metadados podem ser adicionados à mensagem: pode ser um objeto de chave-valor simples que contém dados arbitrários.

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó
- **Mensagem (Message)** *(obrigatório)*: Estática se a notificação contiver um texto estático. Ou dinâmico que permitirá que uma variável seja configurada como saída para este nó. Essa variável será substituída no tempo de execução.
- **Valor (Value)** *(obrigatório)*: conteúdo da mensagem (texto estático ou referência variável).
- **Metadados (Metadata)** *(obrigatório)*: referência de variável que contém um dicionário simples (pares de chave-valores) que contém os metadados a serem adicionados à mensagem

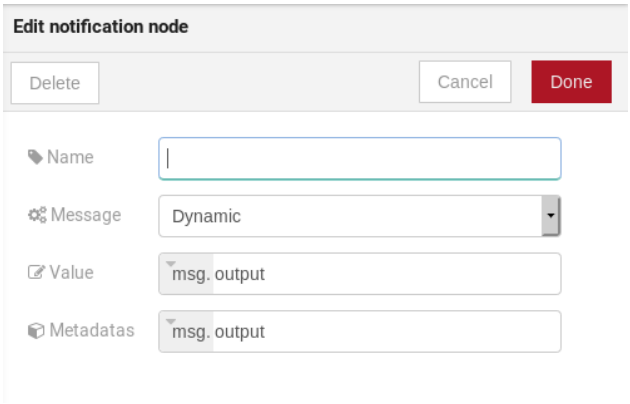
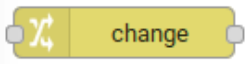


Fig. 12.7: : Modelo de dispositivo janela de configuração

12.1.8 Definir valor (Change)



O nó de Definir valor é usado para copiar ou atribuir valores a uma saída, por exemplo, copie valores de atributos de uma mensagem para um dicionário que será atribuído ao dispositivo virtual.

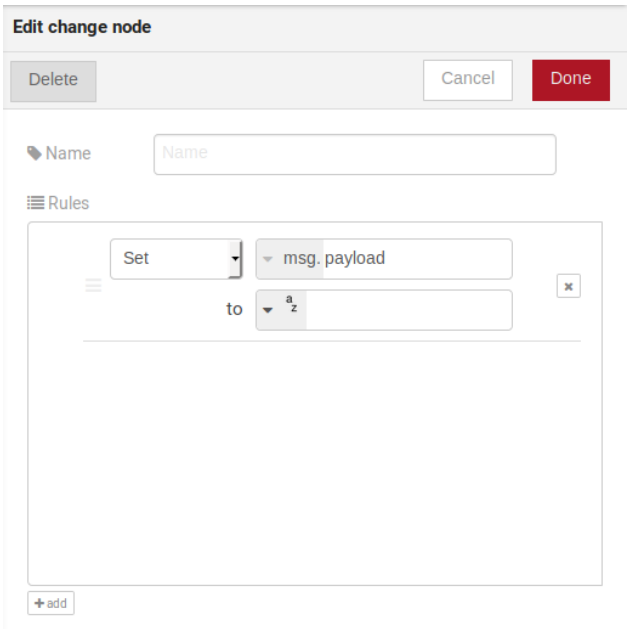


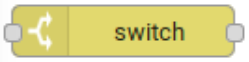
Fig. 12.8: : Definir valor (Change) configuração

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó
- **msg** *(obrigatório)*: definição da estrutura de dados que será enviada para o próximo nó e receberá o valor definido no campo para
- **para (to)** *(obrigatório)*: atribuição ou cópia de valores

Nota: Mais de uma regra pode ser atribuída clicando em *+adicionar* abaixo da caixa de regras.

12.1.9 Interruptor (Switch)



O nó Interruptor permite que as mensagens sejam roteadas para diferentes ramificações de um fluxo avaliando um conjunto de regras em relação a cada mensagem.

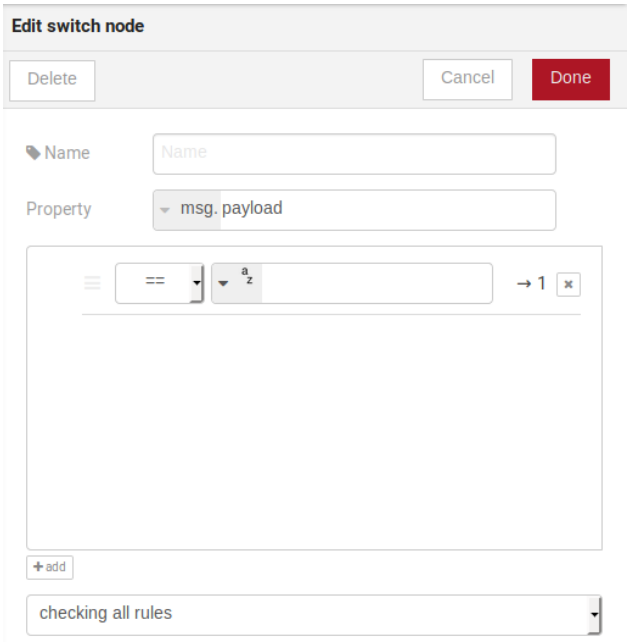


Fig. 12.9: : interruptor configurações

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó
- **Propriedade (Property)** *(obrigatório)*: variável que será avaliada
- **Caixa de regras** *(obrigatório)*: regras que determinarão a ramificação de saída do nó. Além disso, ele pode ser configurado para interromper a verificação de regras quando encontrar uma que corresponda ou verificar todas as regras e rotear a mensagem para a saída correspondente.

Nota:

- Mais de uma regra pode ser atribuída clicando em *+adicionar* abaixo da caixa de regras.
- As regras são mapeadas individualmente para os conectores de saída. Que a primeira regra está relacionada à primeira saída, a segunda regra à segunda saída e etc...

12.1.10 Modelo (Template)

Nota: Apesar do nome, este nó não tem nada a ver com modelos de dispositivos da dojot

Este nó atribuirá um valor a uma variável de destino. Este valor pode ser uma constante, o valor de um atributo que veio do dispositivo de entrada e etc.

Ele usa a linguagem mustache. Verifique a Fig. 12.10 como exemplo: o campo **a** da *payload* será substituído pelo valor de **payload.b**

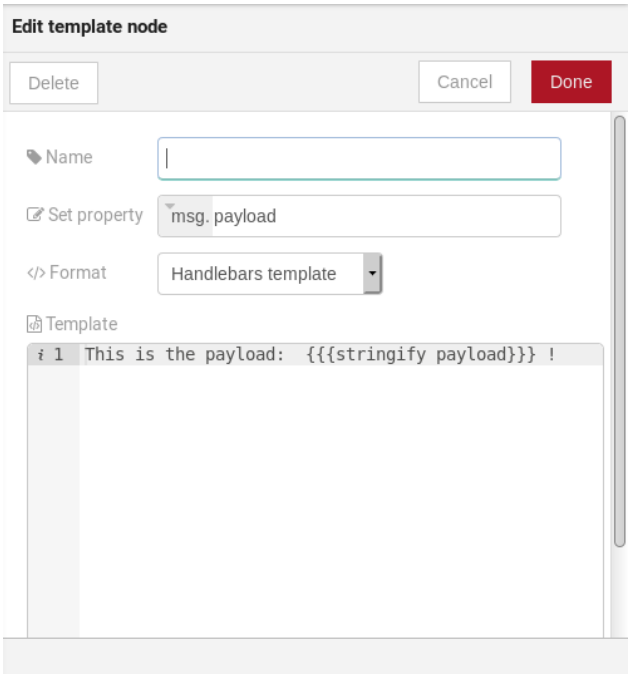


Fig. 12.10: : Configurações do Modelo

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó
- **Propriedade (Set Property)** *(obrigatório)*: variável que receberá o valor
- **Formato (Format)** *(obrigatório)*: o modelo de formato que será gravado
- **Modelo (Template)** *(obrigatório) *: Valor que será atribuído à variável de destino definida em **Propriedade**
- **Saída (Output as)** *(obrigatório)*: o formato da saída

12.1.11 Cron

Nó de processamento para criar/remover tarefas cron. Cron permite agendar tarefas para: enviar eventos para o data broker ou executar uma requisição http.

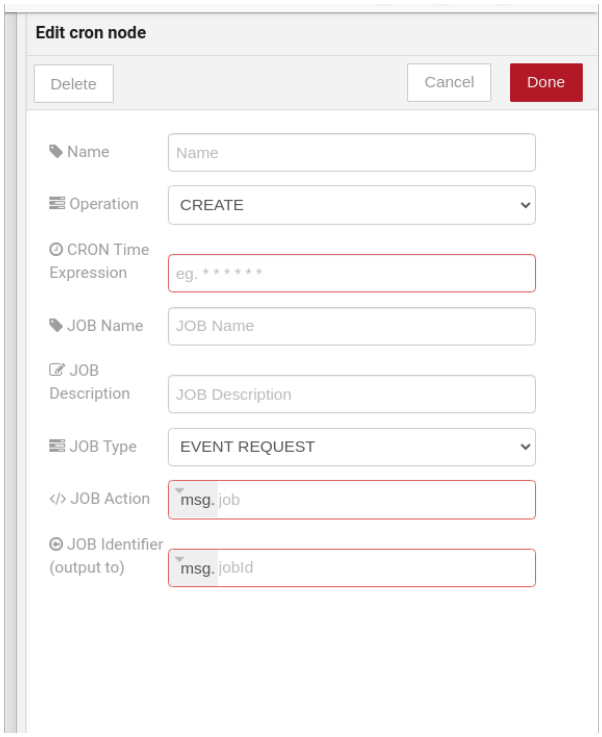


Fig. 12.11: : Cron configuração

Campos:

- **Operação** *(obrigatório)*: define o tipo de processamento ao criar ou remover trabalhos cron (CREATE, REMOVE).
- **Expressão de Tempo do CRON** *(obrigatório)*: Expressão de Tempo do CRON, por exemplo * * * * * *. Obrigatório ao usar a operação do tipo CREATE.
- **Nome do Trabalho** *(opcional)*: Nome do JOB.
- **Descrição do Trabalho** *(opcional)*: Descrição do Trabalho.
- **Tipo do Trabalho** *(obrigatório)*: As opções são EVENT REQUEST ou HTTP REQUEST.
- **Ação do Trabalho** *(obrigatório)*: Variável que contém o JSON para Ação do Trabalho. Este valor pode ser atribuído à variável usando o nó do modelo, por exemplo.
- **Identificador do Trabalho (saída para)** *(obrigatório)*: Variável que receberá o Identificador do Trabalho.
- **Nome (Name)** *(opcional)*: Nome do nó

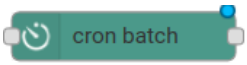
Exemplo de *Ação do Trabalho* quando *Tipo do Trabalho* é **HTTP REQUEST**:

```
{
    "method": "PUT",
    "headers": {
                  "Authorization": "Bearer ${JWT}",
                  "Content-Type": "application/json"
                },
    "url": "http://device-manager:5000/device/${DEVICE_ID}/actuate",
    "body": {
                "attrs": {"message": "keepalive"}
            }
}
```

Exemplo de *Ação do Trabalho* quando *Tipo do Trabalho* é **EVENT REQUEST**:

```
{
    "subject": "dojot.device-manager.device",
    "message": {
                  "event": "configure",
                  "data": { "attrs": { "message": "keepalive"},
                            "id": "6a1213"
                          },
                  "meta": { "service": "admin"}
                }
}
```

12.1.12 Cron em lote (Cron batch)



Funciona como o *cron node*, mas aqui você pode usar agendamentos em lote.

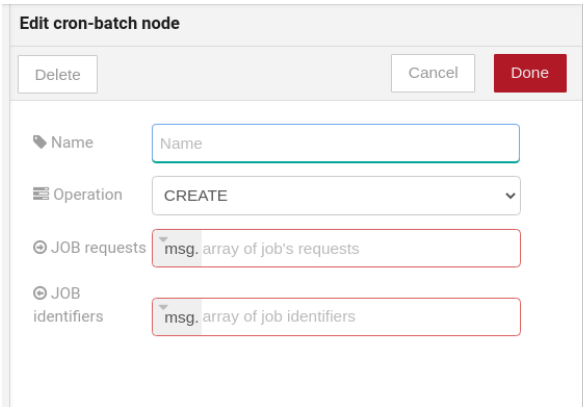


Fig. 12.12: : Cron batch configurações

Campos:

- **Operação** *(obrigatório)*: define o tipo de processamento ao criar ou remover trabalhos cron (CREATE, REMOVE).
- **Requisições de trabalho** *(obrigatório)*: Variável que contém o *array* de JSONs para ações de trabalho.
- **Identificadores de trabalho** *(obrigatório)*: Variável que receberá o *array* de identificadores de trabalho.

- **Nome (Name)** *(opcional)*: Nome do nó

12.1.13 Geo referência (Geofence)

Selecione uma área de interesse para determinar quais dispositivos irão ativar o fluxo

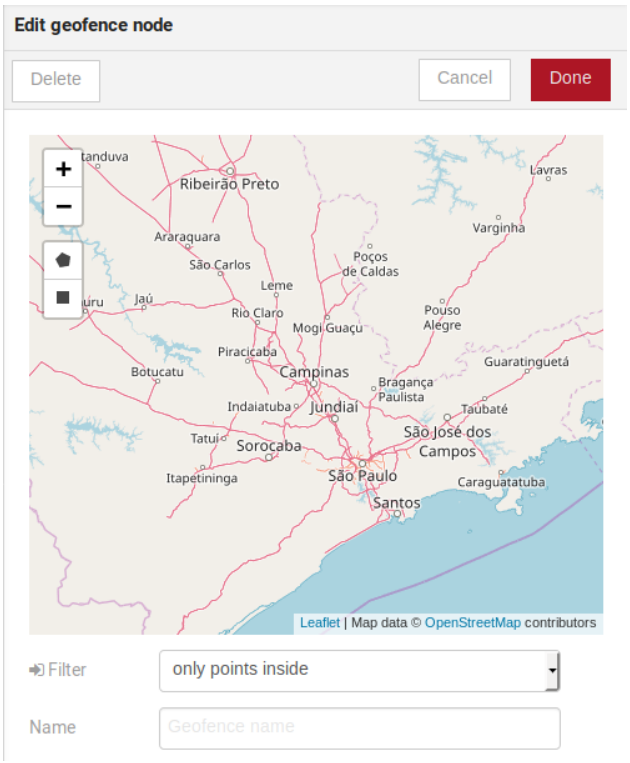
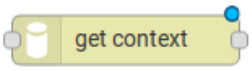


Fig. 12.13: : Geo referência (Geofence) configuração

Campos:

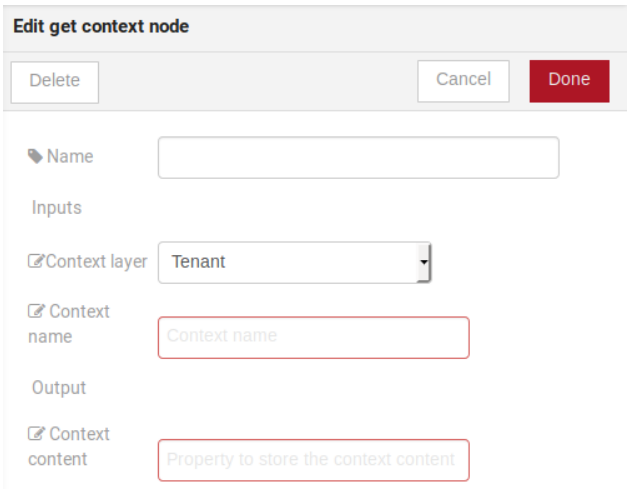
- **Área (Area)** *(obrigatório)*: Área que será selecionada. Pode ser escolhido como um quadrado ou um pentano.
- **Filtro (Filter)** *(obrigatório)*: Qual lado da área será selecionado: dentro ou fora da área marcada no campo acima.
- **Nome (Name)** *(opcional)*: Nome do nó

12.1.14 Obter contexto (Get Context)



Este nó é usado para obter uma variável que está no contexto e atribuir seu valor a uma variável que será usada no fluxo.

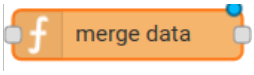
Nota: O contexto é um mecanismo que permite que um determinado conjunto de dados persista além da vida do evento, possibilitando o armazenamento de um estado para os elementos da solução.



Campos:

- **Nome (Name)** *(opcional)**: Nome do nó
- **Camada de contexto (Context layer)** *(obrigatório)**: a camada do contexto em que a variável está
- **Nome do contexto (Context name)** *(obrigatório)**: a variável que está no contexto
- **Conteúdo do contexto (Context content)** *(obrigatório)**: a variável no fluxo que receberá o valor do contexto

12.1.15 Mesclar dados (Merge data)



Este nó permite que os objetos sejam mesclados no contexto do **flow**.

Campos:

- **Dado alvo (JSON)** *(required)*: Variável que contém os dados a serem mesclados.
- **Dado mesclado (JSON) (JSON)** *(required)*: Variável que receberá os novos dados mesclados com os dados existentes.
- **Nome (Name)** *(opcional)*: Nome do nó

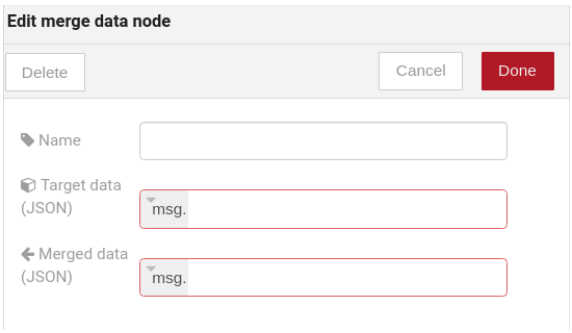
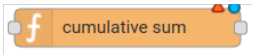


Fig. 12.14: : Configuração de Mesclar dados

12.1.16 Soma acumulada (Cumulative sum)



O nó de soma cumulativa acumula os dados para um atributo em uma janela temporal e mantém isso no contexto do **flow**.

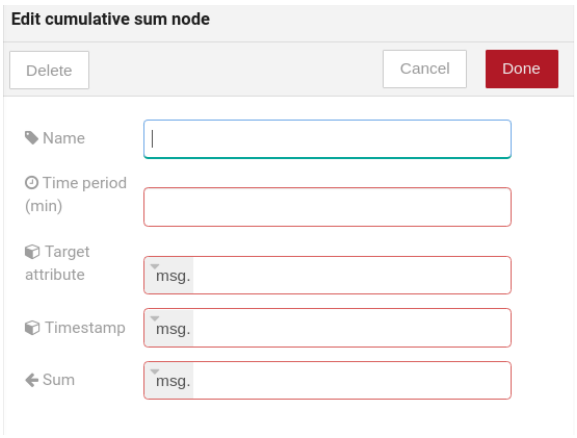
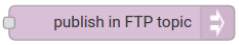


Fig. 12.15: : Configuração de Soma acumulada

Campos:

- **Período de tempo (min)** *(obrigatório)*: Tempo em minutos para manter a soma.
- **Atributo alvo** *(obrigatório)*: Variável que contém o valor a ser soma.
- **Tempo do evento** *(Obrigatório)*: Variável que contém o timestamp advindo do device ou da dojot. Na maioria das vezes pode ser setado com *payload.metadata.timestamp*.
- **Somatório** *(Obrigatório)*: Variável que receberá a soma.
- **Nome (Name)** *(opcional)*: Nome do nó

12.1.17 Publicar no tópico FTP (Publish in FTP topic)



Nó para encaminhar mensagens para o tópico de FTP do Apache Kafka.

Publica no tópico `tenant.dojot.ftp` (*tenant* é definido por qual tenant o fluxo pertence) onde as mensagens são produzidas com informações sobre o nome do arquivo, codificação e conteúdo do arquivo.

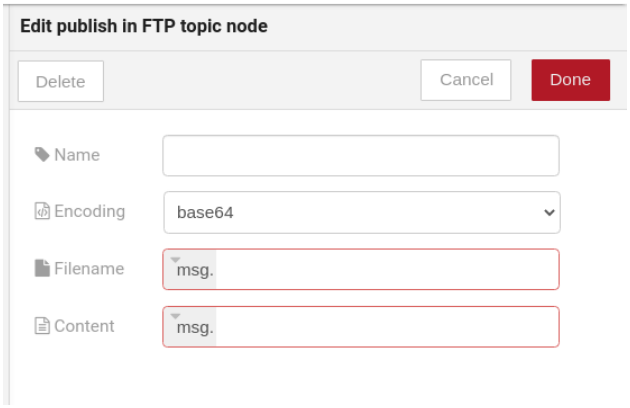


Fig. 12.16: : Configurações do *Publicar no tópico FTP*

Campos:

- **Codificação** *(obrigatório)*: Codificação do conteúdo do arquivo a ser enviado. Os valores válidos são: ascii, base64, hex, utf16le, utf8 e binary.
- **Nome do arquivo (Filename)** *(obrigatório)*: variável que contém o nome do arquivo a ser enviado.
- **Conteúdo do arquivo (Content)** *(obrigatório)*: Essa variável deve conter o conteúdo do arquivo a ser enviado.
- **Nome (Name)** *(opcional)*: Nome do nó

Exemplo de mensagem enviada por este nó:

```
{
    "metadata": {
        "msgId": "33846252-659f-42cc-8831-e2ccb923a702",
        "ts": 1571858674,
        "service": "flowbroker",
        "contentType": "application/vnd.dojot.ftp+json"
    },
    "data": {
        "filename": "filename.jpg",
        "encoding": "base64",
        "content": "..."
    }
}
```

Onde as chaves acima são:

- msgId: Valor do tipo uuidv4 usado para identificar unicamente a mensagem no contexto da dojot.
- ts: Timestamp no formato Unix Timestamp (ms) do momento em que a mensagem foi produzida.
- service: Nome do serviço que gerou a mensagem.

- contentType: Tipo de codificação usado pelo arquivo.
- filename: Nome do arquivo a ser enviado ao servidor FTP.
- encoding: codificação do conteúdo do arquivo. Os valores válidos são: ascii,base64, hex, utf16le, utf8 and *binary*.
- content: conteúdo do arquivo.

Este nó pode ser usado com o componente kafka2ftp. Veja mais em *Componentes e APIs*.

12.1.18 Nós descontinuados

Esses nós estão programados para serem removidos em versões futuras. Eles funcionarão sem problemas com os fluxos atuais.

Device in

Este nó determina um dispositivo específico para ser o ponto de entrada de um fluxo. Para configurar o dispositivo no nó, uma janela como a Fig. 12.17 será exibida.

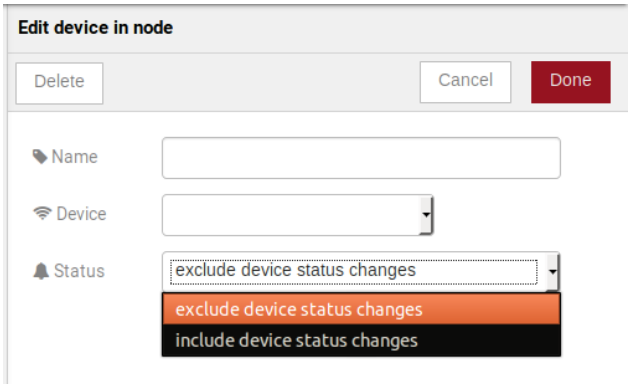


Fig. 12.17: : Dispositivo na janela de configuração

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó
- **Dispositivo (Device)** *(obrigatório)*: o dispositivo *dojot* que acionará o fluxo
- **Estado (Status0** *(obrigatório)*: excluir alterações de status do dispositivo não usará alterações de status do dispositivo (online, offline) para acionar o fluxo. Por outro lado, incluir alterações de status dos dispositivos usará esses status para acionar o fluxo.

Nota: Se o dispositivo que aciona um fluxo for removido, o fluxo se tornará inválido.

Device template in

Esse nó fará com que um fluxo seja acionado por dispositivos compostos por um determinado modelo. Se o modelo de dispositivo configurado no **modelo de dispositivo** no nó for o modelo A, todos os dispositivos compostos com o modelo A acionarão o fluxo. Por exemplo: *dispositivo1* é composto pelos modelos [A, B], *dispositivo2* pelo modelo A e *dispositivo3* pelo modelo B. Então, nesse cenário, apenas as mensagens do *dispositivo1* e do *dispositivo2* iniciarão o fluxo, porque o modelo A é um dos modelos que compõem esses dispositivos.

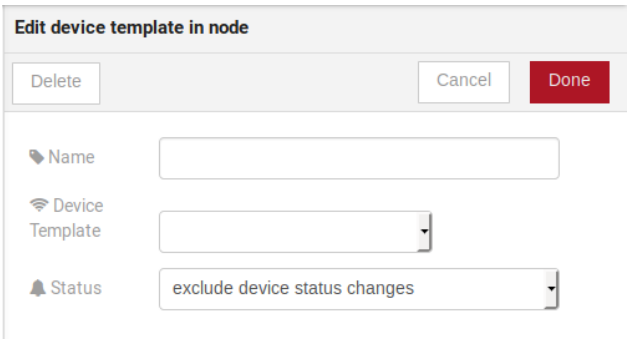
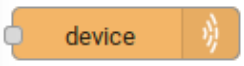


Fig. 12.18: : Modelo de dispositivo janela de configuração

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó.
- **Dispositivo (Device)** *(obrigatório)*: o dispositivo *dojot* que acionará o fluxo.
- **Estado (Status)** *(obrigatório)*: escolha se as alterações de status dos dispositivos acionarão ou não o fluxo.

Device out



A saída do dispositivo determinará qual dispositivo terá seus atributos atualizados na dojot de acordo com o resultado do fluxo. Lembre-se de que este nó não envia mensagens para o seu dispositivo; ele atualiza apenas os atributos na plataforma. Normalmente, o dispositivo escolhido é um dispositivo virtual, que existe apenas na dojot.

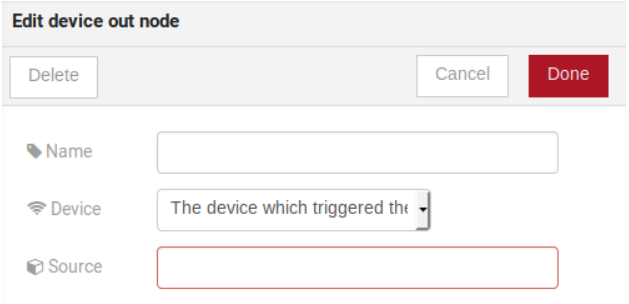


Fig. 12.19: : Janela de configuração do dispositivo

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó.
- **Dispositivo (Device)** *(obrigatório)*: selecione "O dispositivo que acionou o fluxo" fará com que o dispositivo que foi o ponto de entrada seja o ponto final do fluxo. "Dispositivo específico" qualquer dispositivo escolhido será a saída do fluxo e "um dispositivo definido durante o fluxo" tornará um dispositivo que o fluxo selecionou durante a execução o ponto final.
- **Origem (Source)** *(obrigatório)*: estrutura de dados que será mapeada como mensagem para saída do dispositivo

Actuate

O nó de atuação é, basicamente, a mesma coisa que o nó de dispositivo (saída). Mas pode enviar mensagens para um dispositivo real, como dizer a uma lâmpada para desligar a luz e etc.

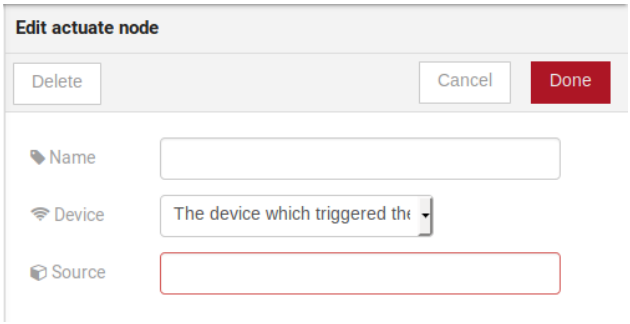


Fig. 12.20: : Configuração de atuação

Campos:

- **Nome (Name)** *(opcional)*: Nome do nó.
- **Dispositivo (Device)** *(obrigatório)*: um dispositivo real na dojot
- **Origem (Source)** *(obrigatório)*: estrutura de dados que será mapeada como mensagem para saída do dispositivo

12.2 Aprenda por exemplos

- *Usando nó http*
- *Usando o nó de geo referência (geofence)*
- *Usando os nós de soma acumulada, interruptor e notificação*

12.2.1 Usando nó http

Imagine este cenário: um dispositivo envia um *usuário* e uma *senha* e, a partir desses atributos, o fluxo solicitará a um servidor um token de autenticação que será enviado a um dispositivo virtual que possua um atributo de *token*.

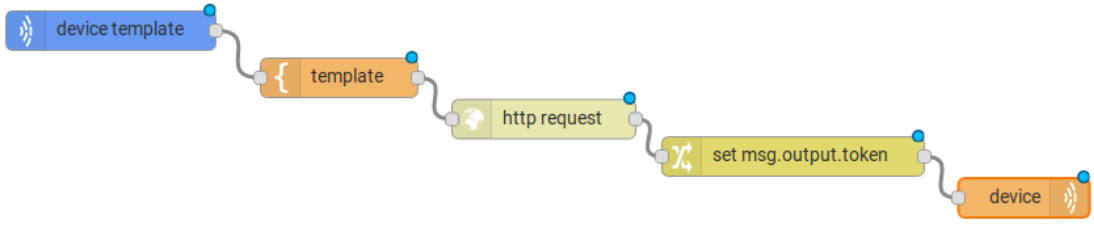


Fig. 12.21: : Fluxo usado para explicar o nó http

Para enviar essa solicitação ao servidor, o método http deve ser um POST e os parâmetros devem estar dentro da requisição. Portanto, no nó do modelo, um objeto JSON será designado a uma variável. O corpo (parâmetros *usuário* e *senha*) da requisição será designado à chave de *payload* do objeto JSON. E, se necessário, esse objeto também pode ter um *headers* como chave.

Em seguida, no nó http, o campo Requisição receberá o valor do objeto criado no nó do modelo. E, a resposta será atribuída a qualquer variável, neste caso, é *msg.res*.

Nota: Se o buffer UTF-8 String for escolhido no campo de retorno, o corpo da resposta será uma string. Se o objeto JSON for escolhido, o corpo será um objeto.

Como visto, a resposta do servidor é *req.res* e o corpo da resposta pode ser acessado em **msg.res.payload**. Portanto, as chaves do objeto que veio na resposta podem ser acessadas por: **msg.res.payload.key**. Na figura Fig. 12.24, o token que veio na resposta é atribuído ao token de atributo do dispositivo virtual.

Em seguida, o resultado do fluxo é que o *token* de atributo do dispositivo virtual seja atualizado com o token que veio na resposta da solicitação http:

12.2.2 Usando o nó de geo referência (geofence)

Um bom exemplo para aprender como o nó geo referência (geofence) funciona é com o fluxo abaixo:

O nó de geo referência é definido como parece na Fig. 12.28. A única coisa que diferencia os nós de geo referência *in area* e *out of the area* é o campo *Filtro* que, no primeiro, é configurado para apontar apenas para dentro e para fora no segundo, respectivamente.

Em seguida, se o dispositivo definido como dispositivo de entrada enviar uma mensagem com um atributo geográfico, o nó geo referência avaliará o ponto geográfico de acordo com sua regra e se corresponder à regra, o nó encaminhará as informações para o próximo nó e, se não , a execução da ramificação, que possui a geo referência que a regra não corresponde, para.

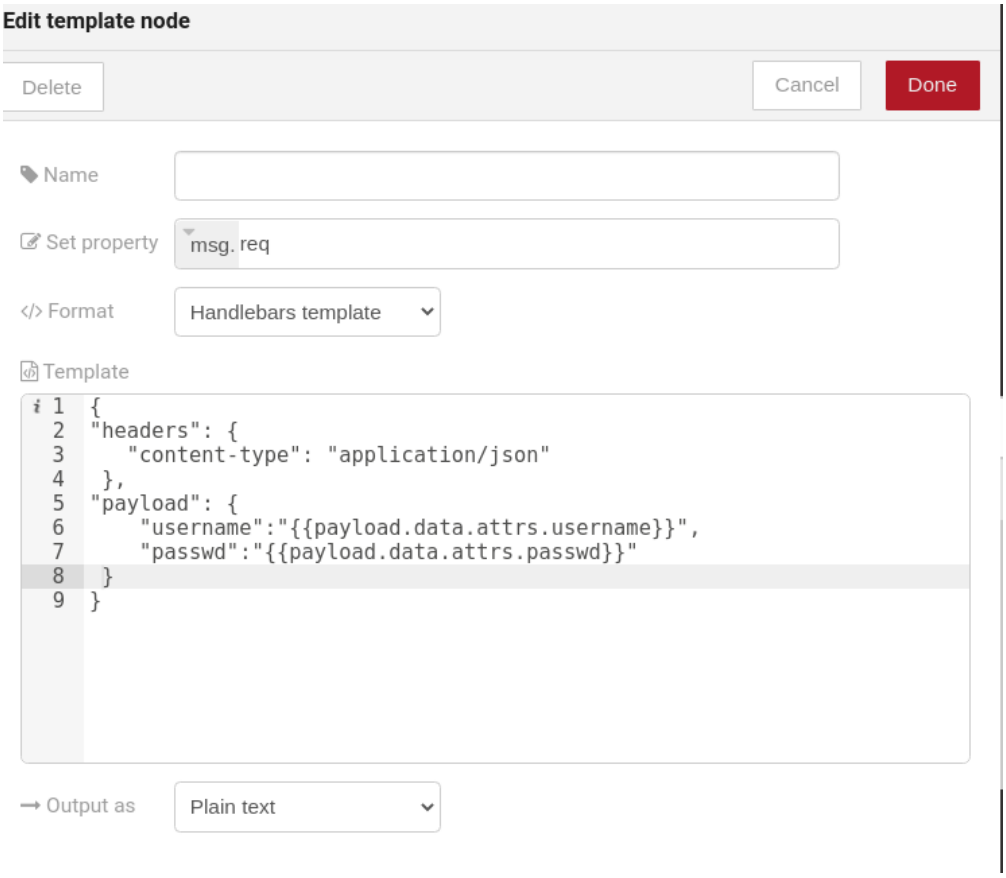


Fig. 12.22: : Configuração do nó do modelo

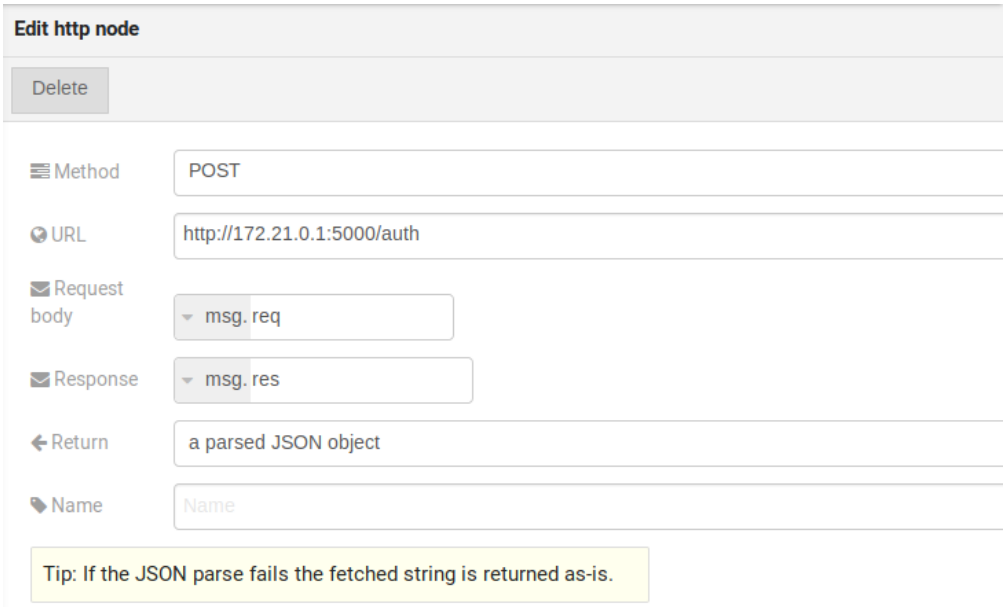


Fig. 12.23: : Configuração do nó do modelo

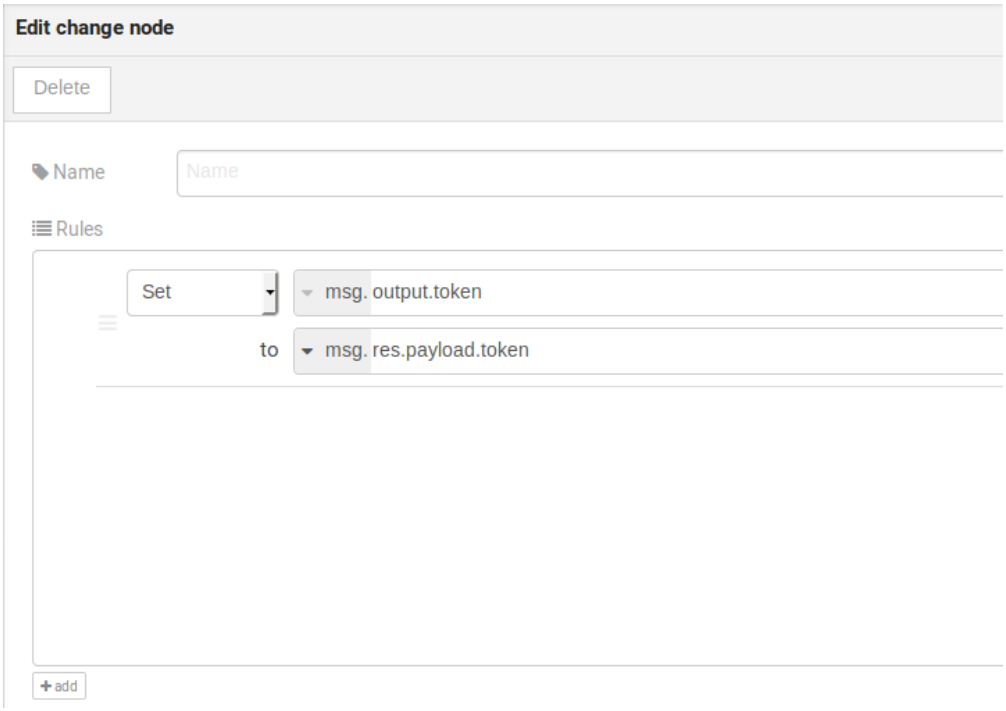


Fig. 12.24: : Configuração do nó do modelo

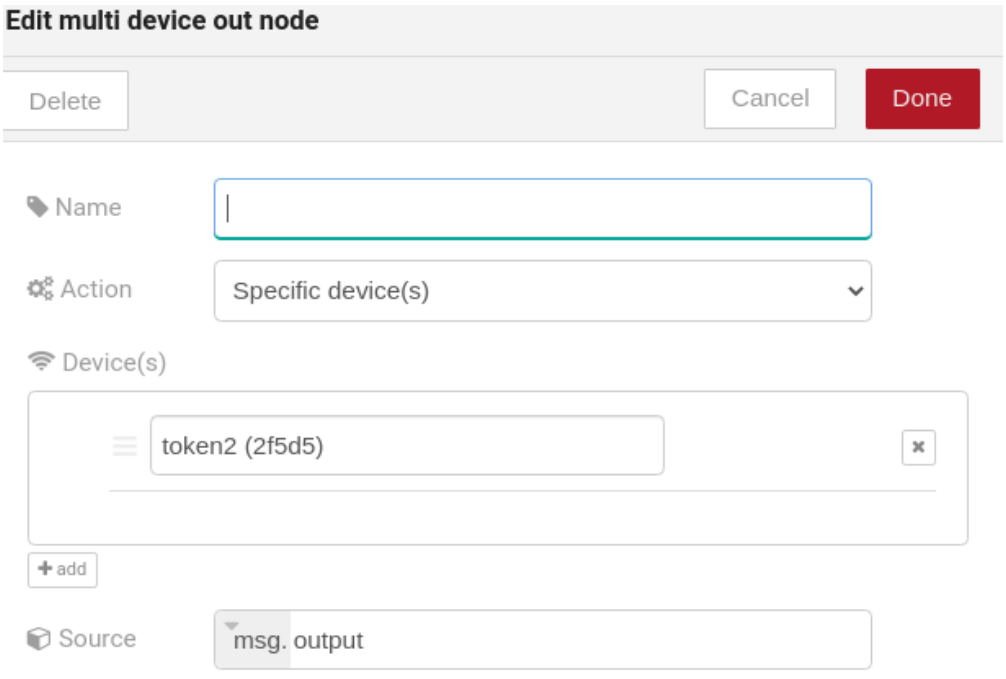


Fig. 12.25: : Configuração do dispositivo de saída

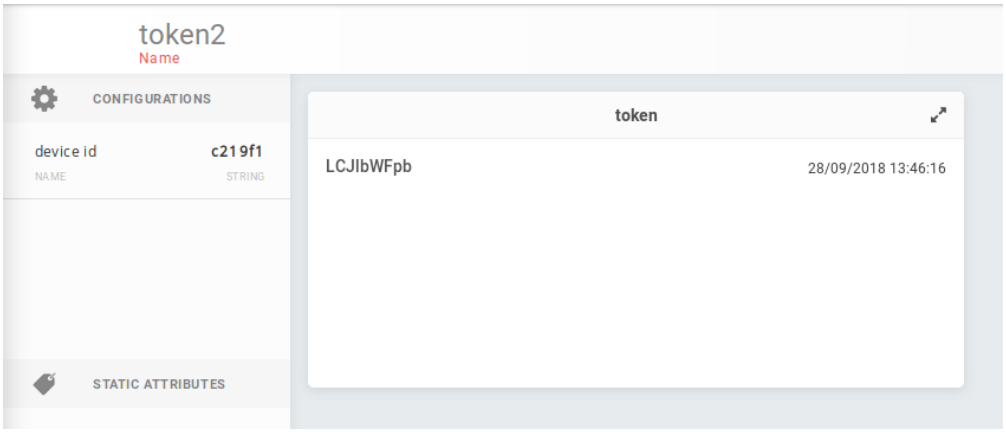


Fig. 12.26: : Dispositivo atualizado

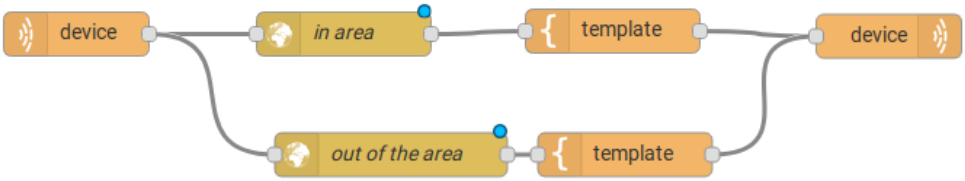


Fig. 12.27: : Fluxo usando geo referência

Nota: Para o nó de geo referência funcionar, a mensagem recebida deve ter um atributo geográfico; caso contrário, as ramificações do fluxo serão interrompidas nos nós de geo referência.

De volta ao exemplo, se o carro enviar uma mensagem de que ele está na área marcada, como `{"position": "-22.820156, -47.2682535"}`, a mensagem recebida no dispositivo será "Carro está dentro da área marcada" e se enviar `{"position": "0,0"}` o dispositivo receberá "O carro está fora da área marcada"

12.2.3 Usando os nós de soma acumulada, interruptor e notificação

Imagine este cenário: um dispositivo envia o nível de chuva, queremos gerar uma notificação se o acumulado, soma, das chuvas nas últimas horas é maior que 100.

No nó *cumulative sum*, nós iremos acumular o valor de *rain* (*Target attribute*) na janela de tempo de 60 minutos (*Time period*) e iremos setar essa soma em um novo atributo chamado *payload.data.attrs.rain60Min* (*Sum*). A configuração *Timestamp* refere-se ao *timestamp* advindo do dispostivo ou da dojot. Na maioria das vezes pode ser setado com *payload.metadata.timestamp*. Veja mais em Fig. 12.32.

Nós queremos que a notificação seja disparada apenas se o valor de chuva acumulado seja maior que 100, para isso usaremos o nó *switch*. Como na imagem Fig. 12.33.

Agora, caso nosso valor seja maior que 100 precisamos gerar a notificação, para tal usaremos um nó auxiliar antes, o nó *template*. No nó *template* iremos criar a mensagem que irá aparecer na notificação e definir seus metadados, Fig. 12.34 .

Finalmente, vamos configurar o nó de notificação, como na imagem Fig. 12.35.

Portanto, se a estação meteorológica (dispositivo definido no nó do *dispositivo de evento* com publicação marcada)

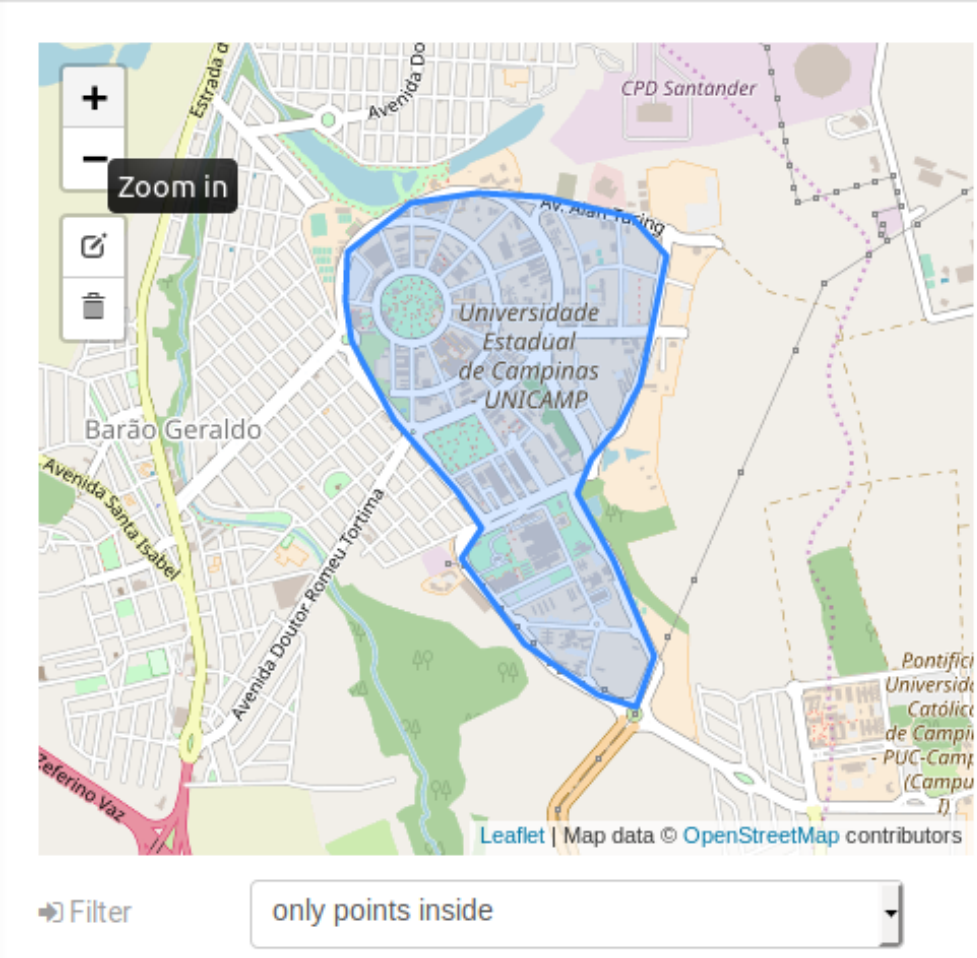


Fig. 12.28: : Configuração do nó de geo referência

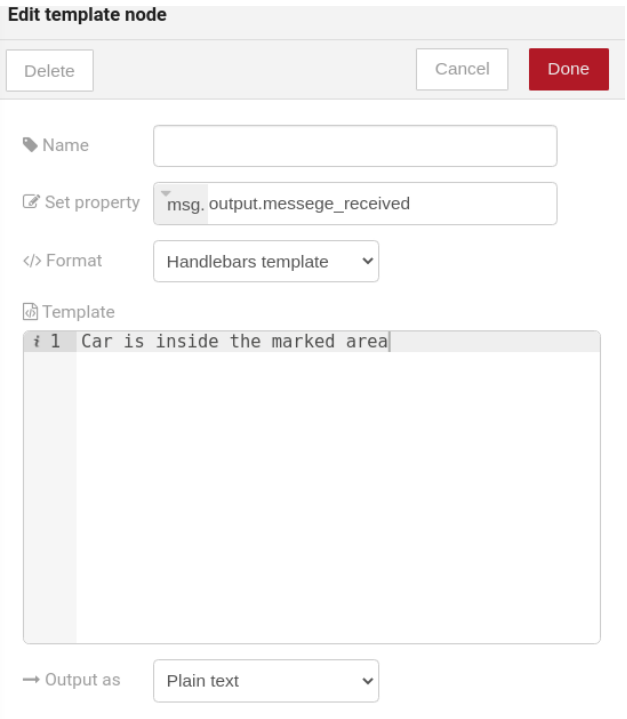


Fig. 12.29: : Configuração do nó do modelo se o carro estiver na área marcada

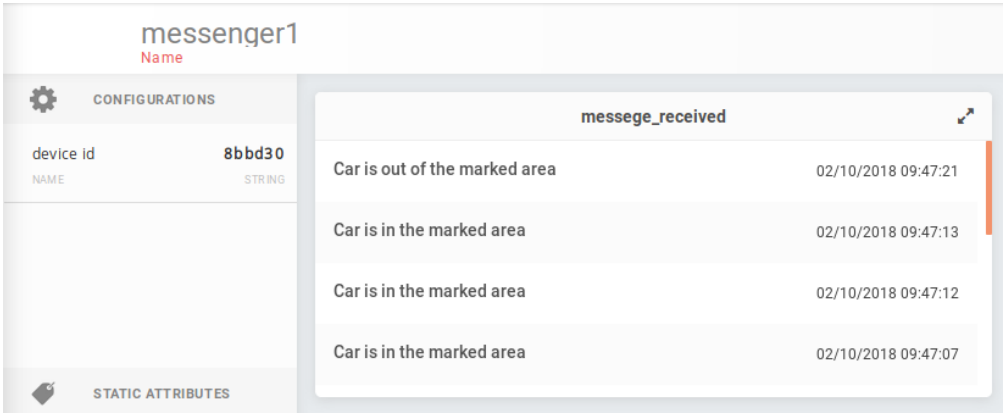


Fig. 12.30: : Saída no dispositivo

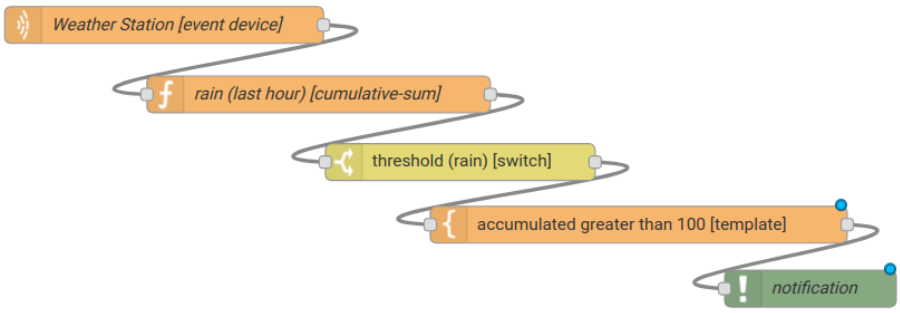


Fig. 12.31: : Fluxo usando os nós de soma acumulada, interruptor e notificação

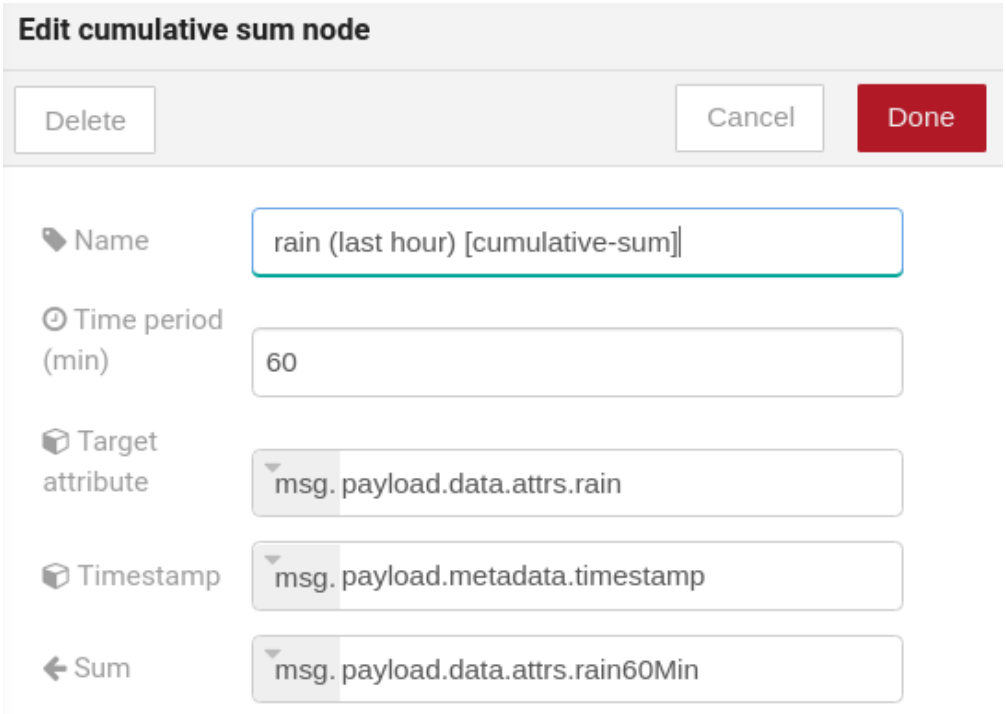


Fig. 12.32: : Configuração de Soma acumulada

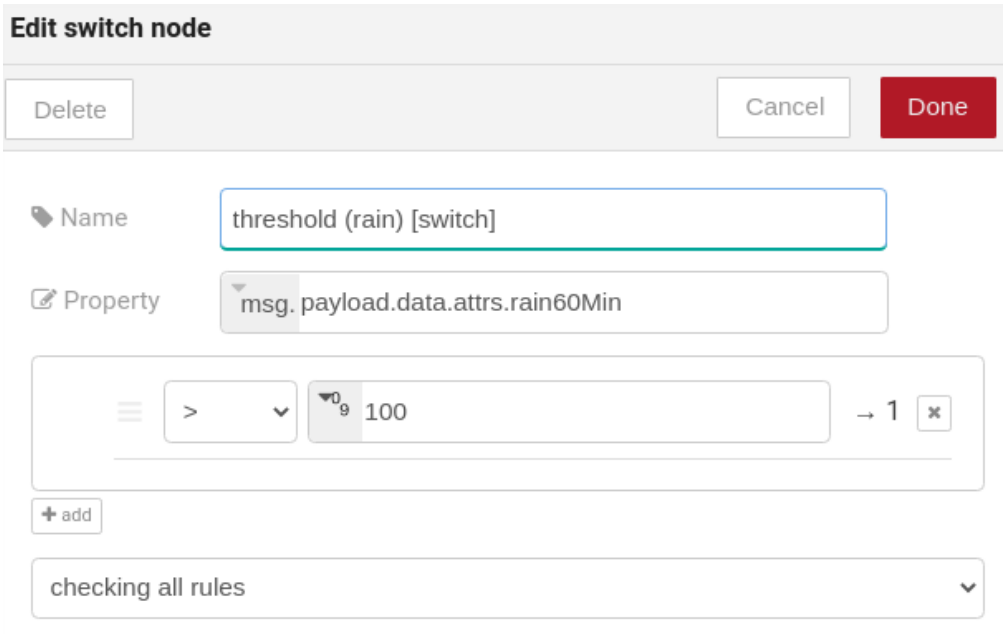


Fig. 12.33: : Configurações do nó interruptor (Switch)

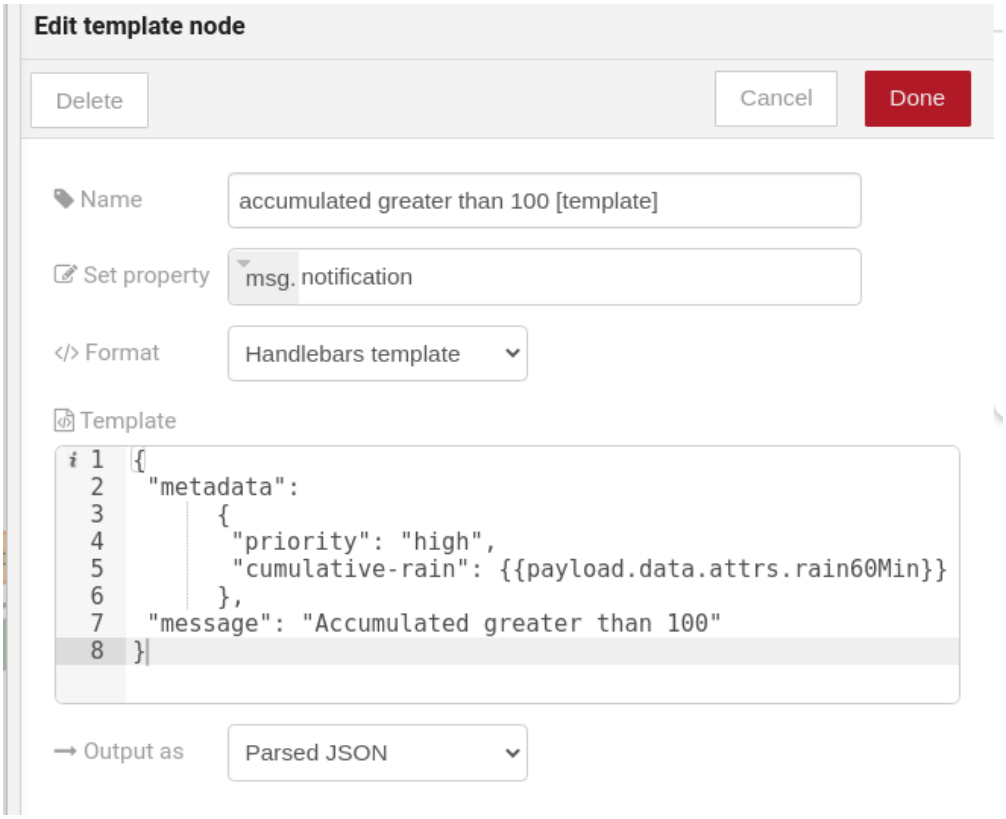


Fig. 12.34: : Configuração do nó do modelo

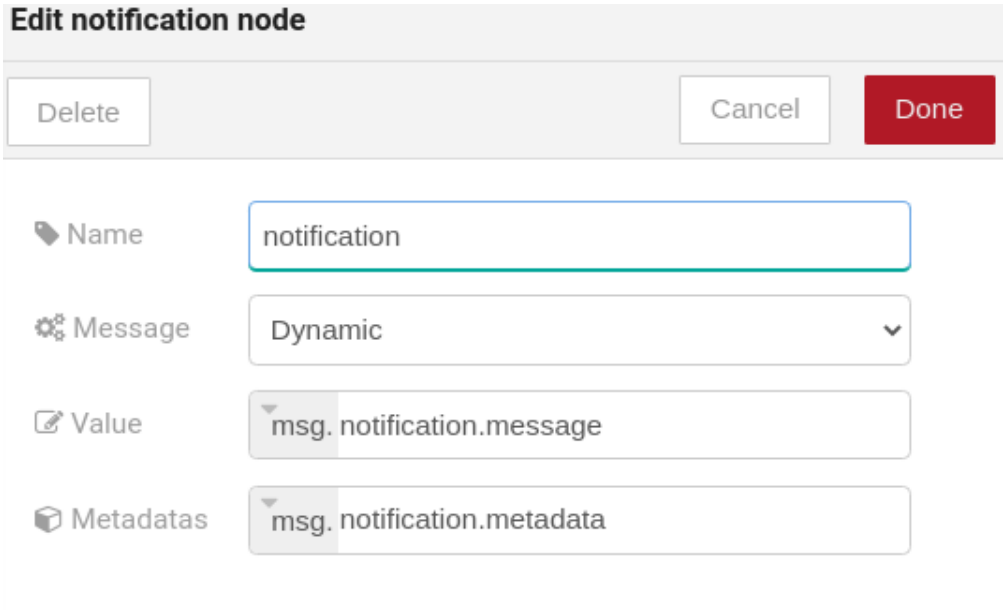


Fig. 12.35: : Configuração do nó de notificação

envia várias mensagens como *{"rain": 5}* durante a última hora e em uma dessas vezes a soma for superior a 100, uma notificação será gerada. Nota: Várias notificações podem ser geradas, desde que o valor acumulado seja superior a 100 na última hora. Veja a imagem Fig. 12.36.

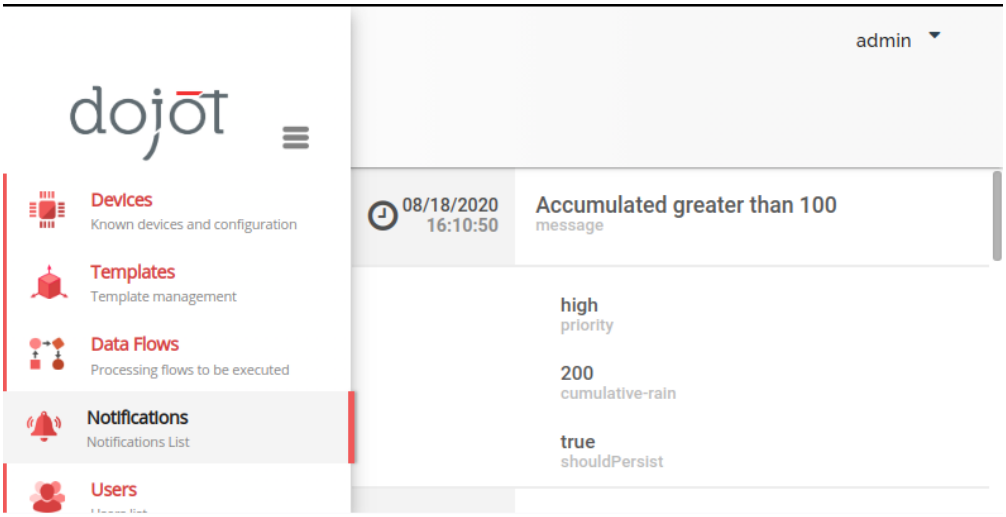


Fig. 12.36: : Notification

CAPÍTULO 13

Usando MQTT com segurança (TLS)

Nota:

- Público: usuários
- Nível: intermediário
- Tempo de leitura: 15 m

Este documento descreve como configurar a dojot e publicar/subscrever usando MQTT com TLS (MQTTS) ao usar o microsserviço *Broker MQTT* IotAgent-VerneMQ ou *Broker MQTT* legado IotAgent-Mosca.

Índice

13.1 Componentes

13.1.1 X.509 Identity Management

O objetivo deste componente é fornecer identificação x.509 para entidades dojot finais, ou seja, dispositivos IoT que se comunicam com a plataforma de IoT dojot. Veja mais sobre em X.509 Identity Management.

O que é um certificado?

Um certificado contém a chave pública de uma entidade (um usuário, dispositivo, website), juntamente com informações sobre essa entidade, sobre a CA que assina o certificado, o uso permitido do certificado e uma soma de verificação. Quando uma entidade deseja que um certificado seja assinado, a entidade deve criar um arquivo CSR e enviá-lo para a CA desejada. O arquivo CSR é uma 'intenção de certificação'. O arquivo contém as informações necessárias da entidade e algumas informações sobre o uso do certificado, nomes de host e IPs onde o certificado residirá, nomes alternativos para a entidade etc.

13.1.2 *MQTT Brokers*

IotAgent VerneMQ (Padrão)

O *IotAgent VerneMQ* estende o VerneMQ com alguns recursos e serviços para o caso da dojot, veja mais em IotAgent-VerneMQ. Este é o *Broker MQTT* **padrão** em *deployments* dojot.

IotAgent Mosca (Legado)

O *IotAgent Mosca* usa o Mosca que é um *broker* mqtt em node.js , veja mais sobre este serviço em IotAgent-Mosca. Este **não é** o *Broker MQTT* **padrão** em *deployments* dojot.

Sobre o uso do *IotAgent VerneMQ* ou *IotAgent Mosca*

Você não deve usar os dois *brokers* juntos para evitar conflitos de porta. Use VerneMQ (padrão) ou Mosca (legado). Por padrão, nossas *deployments* estão usando VerneMQ.

**É necessário configurar o domínio no qual o *broker* estará acessível: **

Para a implantação do Ansible:

- É necessário configurar a variável `dojot_domain_name` antes de iniciar com o domínio ou o ip que será usado para se comunicar com o MQTT via TLS.

Para a implantação Docker-compose:

- É necessário configurar a variável ` DOJOT_DOMAIN_NAME`` no arquivo *.env*. antes de começar iniciar com o domínio ou o ip que será usado para se comunicar com o MQTT via TLS.

Além disso, você pode escolher entre *IotAgent VerneMQ* ou` IotAgent Mosca` ao configurar o *deployment via Ansible* (ou seja, Kubernetes). No docker-compose, você precisa descomentar e comentar os serviços no arquivo `docker-compose.yml`, há uma explicação comentada sobre isso neste arquivo.

Verifique o *Guia de instalação* para mais informações.

Todos os certificados serão criados automaticamente, não sendo necessário configurar manualmente os certificados nos *Brokers*.

13.2 Como conectar um dispositivo com o *IotAgent VerneMQ* ou *IotAgent Mosca* com TLS mútuo

Atenção: Em primeiro lugar, você precisará de um ambiente da dojot em execução. Depois, será necessário criar o dispositivo na dojot (or usar um que já existe) e obter seu ID.

13.2.1 Recuperando certificado para um dispositivo

Para um dispositivo se conectar usando MQTT sobre TLS (MQTTS), ele deve possuir:

- Um par de chaves (arquivo .key);
- Um certificado assinado por uma Autoridade de Certificação (CA) confiável pela dojot (arquivo .crt);
- O certificado desta CA (arquivo .crt);

O objetivo ao recuperar o certificado para um dispositivo é obter estes três arquivos: o certificado do dispositivo, o par de chaves do dispositivo e o certificado da *CA*.

Existem duas ferramentas para facilitar a obtenção de certificados na plataforma dojot:

- O *script* CertReq .
- Utilitário de geração de certificado embutido da GUI (mais detalhes em *Gerando certificados para dispositivos*).

Além disso, você pode usar o OpenSSL para criar certificados e assinar usando a API - x509-identity-mgmt, veja mais em X.509 Identity Management.

Uma breve explicação de como usar CertReq

Como pré-requisitos, ele usa git, OpenSSL, curl e jq.

Em distribuições Linux baseadas em Debian, você pode instalar esses pré-requisitos executando:

```
sudo apt install git curl jq openssl
```

Baixe CertReq em sua máquina diretamente do repositório dojot e mude para a versão correspondente do seu ambiente dojot:

```
git clone https://github.com/dojot/dojot.git
cd dojot
git checkout v0.7.0
```

Entre no diretório `certreq`:

```
cd tools/certreq
```

Finalmente, você pode executar o script para gerar certificados e chaves da seguinte maneira:

```
./bin/certreq.sh \
   -h http://localhost \
   -p 8000 \
   -i 'a1998e' \
   -u 'admin' \
   -s 'admin'
```

Dado um *nome de usuário* `admin` e uma *senha* `admin`, este comando irá solicitar um certificado com *ID do dispositivo (identificador)* `a1998e` para a plataforma dojot no *host* `localhost` na *porta* `8000`.

Nota: Para mais detalhes sobre os parâmetros do CertReq, verifique o documento CertReq - Parameters. Outros recursos úteis para este assunto são o tutorial How to connect a device with the IoTAgent-VerneMQ with mutual TLS e a documentação do CertReq.

E no final esta ferramenta irá criar os diretórios `./ca` e `./cert_{DISPOSTIVO_ID}` para armazenar os certificados e chaves públicas/privadas.

13.2.2 Simulando um dispositivo com mosquitto

Vamos usar *mosquitto* para simular um dispositivo; ele publicará e subscreverá na dojot via MQTT.

Antes de continuar, instale o *mosquitto_pub* e *mosquitto_sub* do pacote *mosquitto-clients* em distribuições Linux baseadas no Debian:

Atenção: Algumas distribuições Linux, distribuições Linux baseadas em Debian em particular, tem dois pacotes para mosquitto um contendo ferramentas para cliente (ou seja, *mosquitto_pub* e *mosquitto_sub* para publicar mensagens e se subscrever a tópicos) e outro contendo o *Broker* MQTT também. Neste tutorial, apenas as ferramentas do pacote *mosquitto-clients* serão utilizadas. Verifique se um outro *Broker* MQTT não está em execução antes de iniciar a dojot (executando comandos como `ps aux | grep mosquitto`) para evitar conflitos de porta.

Em distribuições Linux baseadas em Debian, você pode instalar `mosquitto-clients` executando:

```
sudo apt-get install mosquitto-clients
```

IotAgent VerneMQ (Padrão)

Para publicar e subscrever usando os certificados apropriados, você deve ter *IotAgent VerneMQ Broker*, *V2K Bridge*, *K2V Bridge* e o *X.509 Identity Management* em execução, veja mais em IotAgent-VerneMQ.

Simulando uma publicação de dispositivo com mosquitto

```
mosquitto_pub  -h localhost -p 8883 -t <tenant>:<deviceId>/attrs  -m '{"attr_example
↪": 10}' --cert <device .crt file> --key <device .key file> --cafile <ca .crt file>
```

Um exemplo de publicação com os certificados e chaves gerados no tópico anterior com a ferramenta CertReq.

```
mosquitto_pub \
-h localhost \
-p 8883 \
-t admin:a1998e/attrs \
-m '{"attr_example": 10 }' \
--cert './cert_a1998e/cert.pem' \
--key './cert_a1998e/private.key' \
--cafile './ca/ca.pem'
```

Simulando um dispositivo subscrevendo com mosquitto

```
mosquitto_sub  -h localhost -p 8883 -t <tenant>:<deviceId>/config  --cert <device .
↪crt file> --key <device .key file> --cafile <ca .crt file>
```

Para obter mais detalhes sobre como simular um dispositivo, consulte Simulating a device with mosquitto e mais sobre simular um dispositivo com segurança em Simulating a device with mosquitto with security.

IotAgent Mosca (Legado)

Para publicar e subscrever usando os certificados apropriados, você deve estar com o *IotAgent Mosca Broker* e o *X.509 Identity Management* em execução, veja mais em IotAgent-Mosca. Além disso, você precisa usar um **tópico diferente** do VerneMQ e passar o identificador para publicar e subscrever, da seguinte maneira:

Como publicar:

```
mosquitto_pub  -h localhost -p 8883 -t /<tenant>/<deviceId>/attrs -i <tenant>:
↪<deviceId> -m '{"attr_example": 10}' --cert <device .crt file> --key <device .key␣
↪file> --cafile <ca .crt file>
```

Como se subscrever:

```
mosquitto_sub  -h localhost -p 8883 -t /<tenant>/<deviceId>/config -i <tenant>:
↪<deviceId> --cert <device .crt file> --key <device .key file> --cafile <ca .crt␣
↪file>
```

Nota: Nestes exemplos, a mensagem publicada possui o atributo *attrs_example*. Você precisa alterar para os atributos do seu dispositivo.

13.2.3 Modo inseguro do MQTT (sem TLS)

Atenção: MQTT sem segurança não é recomendado, use-o apenas para teste.

Em *deployment via Ansible* (kubernetes), você pode desabilitar o modo não seguro alterando a variável `dojot_insecure_mqtt` para` *false*`, isso é válido em ambos os *brokers*. Verifique *Guia de instalação* para mais informações.

IotAgent VerneMQ (Padrão)

Você pode desabilitar o *modo inseguro* se você tornar a porta 1883 indisponível para acesso externo.

Para obter mais detalhes sobre como simular um dispositivo sem segurança, verifique o tutorial Simulating a device with mosquitto without security .

IotAgent Mosca (Legado)

Você pode desabilitar o *modo inseguro* no Mosca mudando a variável de ambiente *ALLOW_UNSECURED_MODE* para `'false'` ou removendo o acesso externo à porta *1883*. Veja mais em IotAgent-Mosca.

13.3 Como ler um certificado

Um arquivo de certificado pode estar em dois formatos: PEM (texto base64) ou DER (binário). OpenSSL oferece ferramentas para ler esses formatos:

Como ler um certificado:

```
openssl x509 -noout -text -in certFile.crt
```

Obtendo a *fingerprint* do certificado:

```
openssl x509 -in certFile.crt -noout -fingerprint -sha256
```

CAPÍTULO 14

Aplicando teste de carga na plataforma Dojot

Neste tutorial será demonstrado como rodar um teste de carga usando a implementação do Locust feita para a dojot.

Atenção: A implementação do Locust foi feita para realizar testes com o VerneMQ. Testes com o Mosca não têm garantia de funcionamento.

Índice

- *Dicas gerais para o teste*

14.1 Preparando o ambiente

Primeiramente, você precisa de um ambiente dojot rodando. Verifique o *Guia de instalação* para mais informações.

To access the Locust implementation, download the dojot repository on your machine and switch to the same version as your current environment:

```
git clone https://github.com/dojot/dojot.git
cd dojot
git checkout v0.7.0
```

Entre na pasta do Locust:

```
cd connector/mqtt/locust
```

14.2 Rodando um teste simples

Nesta seção, será demonstrado como configurar e rodar o Locust, além de como gerar certificados com o script `generate_certs` que serão utilizados para executar o teste de carga. Este é um teste simples, onde serão criados 100 clientes que irão enviar mensagens à dojot. Antes de rodar os testes

14.2.1 Configuração

Nota: Verifique o README incluído no diretório do Locust para aprender mais sobre a arquitetura e configurações. Neste tutorial serão cobertas somente as configurações mais importantes.

Antes de rodar os testes, algumas configurações nos arquivos de docker compose do Locust precisam ser alteradas.

Abra o arquivo `Docker/docker-compose-master.yml` e mude as seguintes variáveis de ambiente:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same␣
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"
```

Abra o arquivo `Docker/docker-compose-slave.yml` e mude as seguintes variáveis de ambiente:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same␣
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"
```

Nota: Estamos assumindo que você está rodando o *master* e o *slave* na mesma máquina, i.e. no *modo não distribuído*. Mais à frente será mostrado como distribuir os *slaves* em múltiplas máquinas.

Abra o arquivo `Docker/scripts/generate_certs/docker-compose.yml` e mude as seguintes variáveis de ambiente:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
```

14.2.2 Gerando os certificados

Como dito anteriormente, a comunicação entre Locust e dojot é segura, portanto é necessário o uso de certificados.

Existem duas maneiras de se simular os dispositivos: você pode criar dispositivos falsos (que não aparecerão na GUI da dojot) ou dispositivos reais. Nesta parte do tutorial serão criados os dispositivos reais para que você possa verificar a chegada das mensagens na GUI.

Antes de rodar o script, é necessário inicializar o Locust master. Dentro do repositório do Locust, rode:

```
docker-compose -f Docker/docker-compose-master.yml up
```

Após sua inicialização, rode o contêiner do script `generate_certs` e entre nele:

```
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml up -d
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml exec generate-
↪certs bash
```

Crie os dispositivos na dojot:

```
generate_certs dojot create --devices 100
```

Agora é possível verificar que os dispositivos foram criados na dojot.

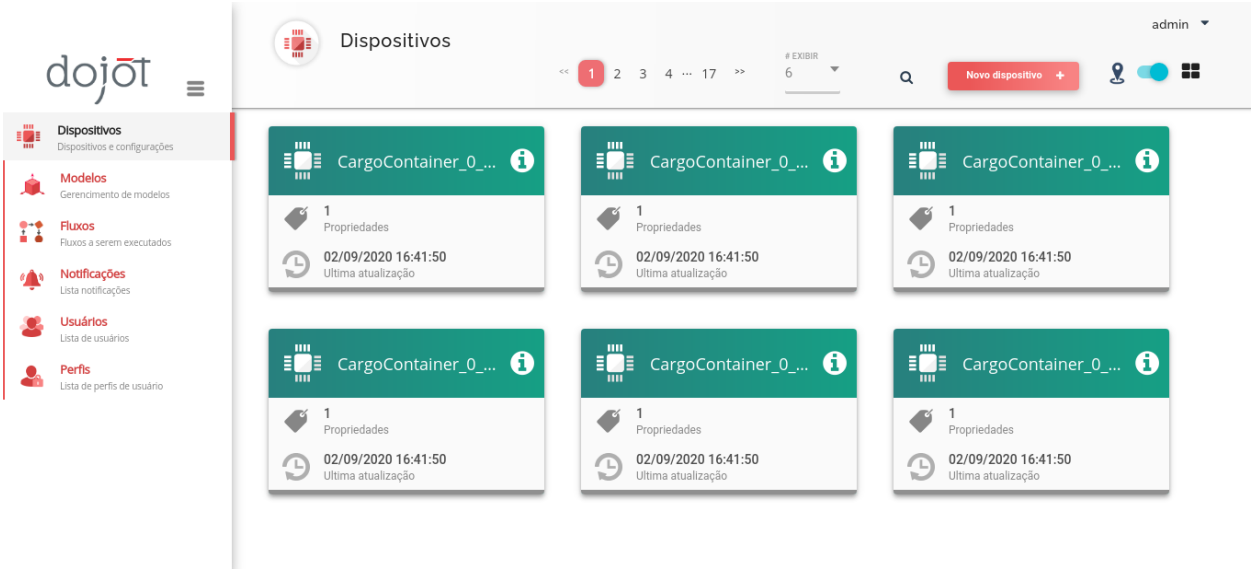


Fig. 14.1: : Alguns dos dispositivos que o `generate_certs` criou na dojot.

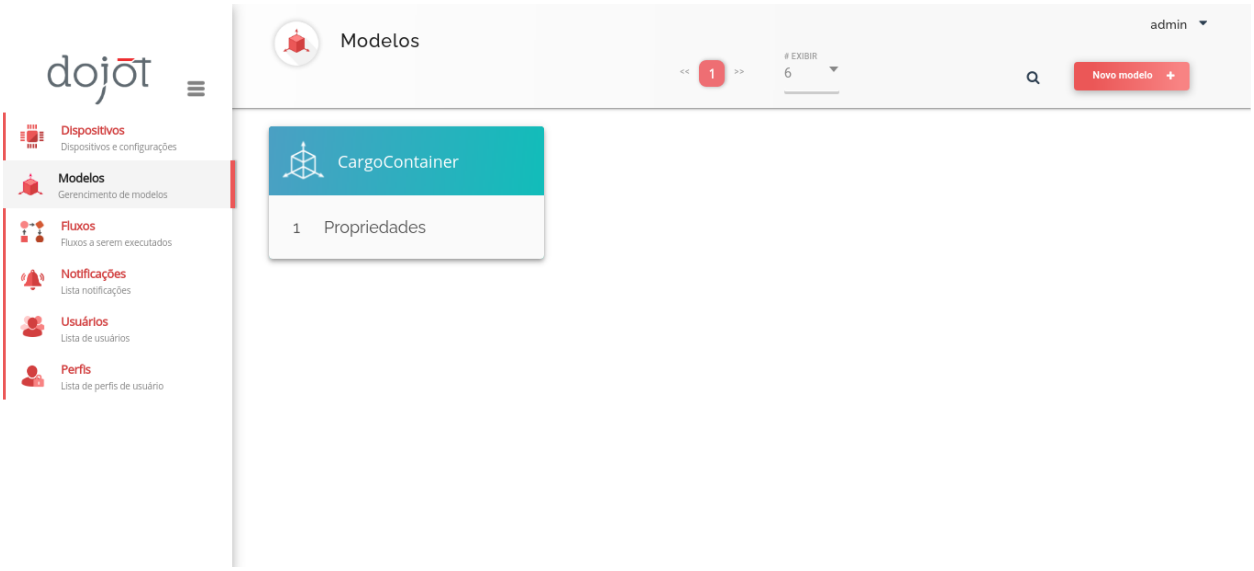


Fig. 14.2: : O *template* usado pelo `generate_certs` para criar os dispositivos.

Gere os certificados para eles:

```
generate_certs cert --dojot
```

Os certificados são exportados para a pasta `cert`. Agora o teste pode ser inicializado!

14.2.3 Inicializando os slaves

O Locust *master* não faz nada por si só. Quem realmente faz todo o trabalho de enviar as requisições é o contêiner do Locust *slave*. Para inicializá-lo, rode:

```
docker-compose -f Docker/docker-compose-slave.yml up
```

Você deve ver no Locust *master* uma mensagem dizendo que há um *slave* conectado a ele.

14.2.4 Rodando o teste

Está tudo pronto para inicializar o teste. Para rodá-lo, é preciso acessar a interface do Locust em seu navegador no endereço `localhost:8089` (supondo que você está rodando o Locust *master* no `localhost`).

Digite `100` em `Number of users to simulate`, `10` em `Hatch rate` e clique em `Start swarming`. Isso dirá ao Locust para rodar 100 clientes, criando 10 deles por segundo.

A configuração padrão dita que cada cliente mande uma mensagem a cada 30 segundos, então você deverá aguardar um pouco para ver as mensagens chegando na dojot.

Você pode ir até a dojot e ver as mensagens chegando por lá.

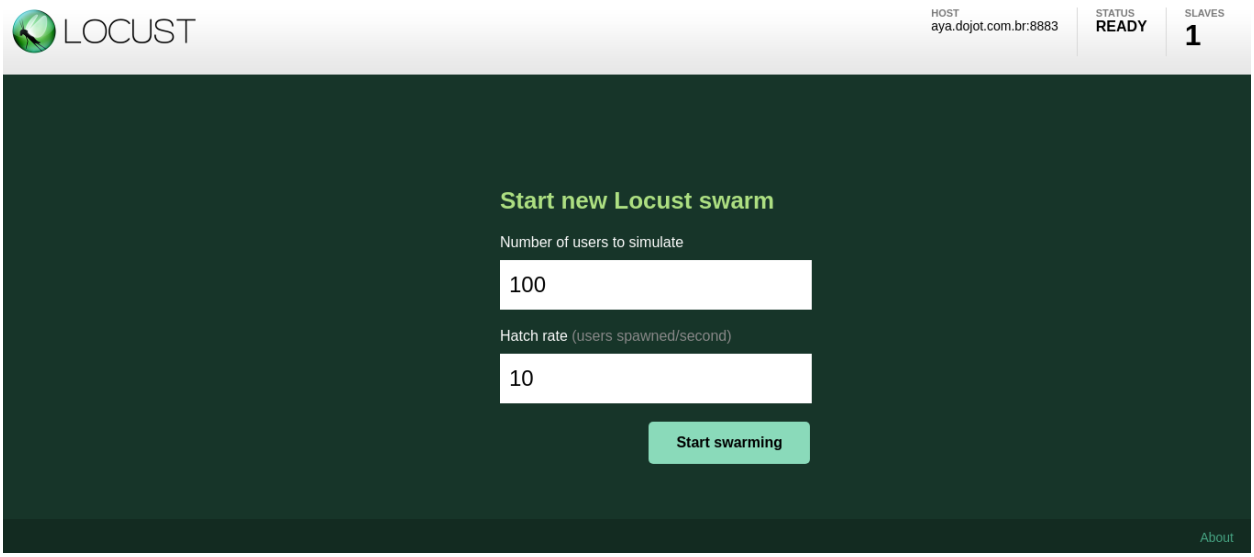


Fig. 14.3: : Configurando o Locust para rodar os clientes.

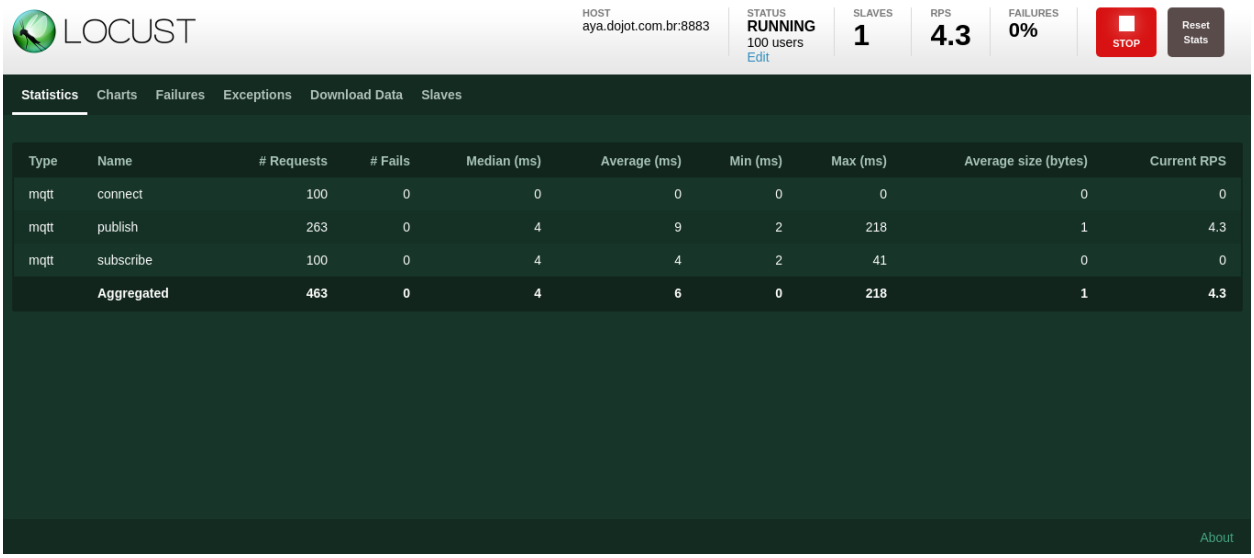


Fig. 14.4: : As estatísticas do Locust após rodar os testes por alguns minutos.

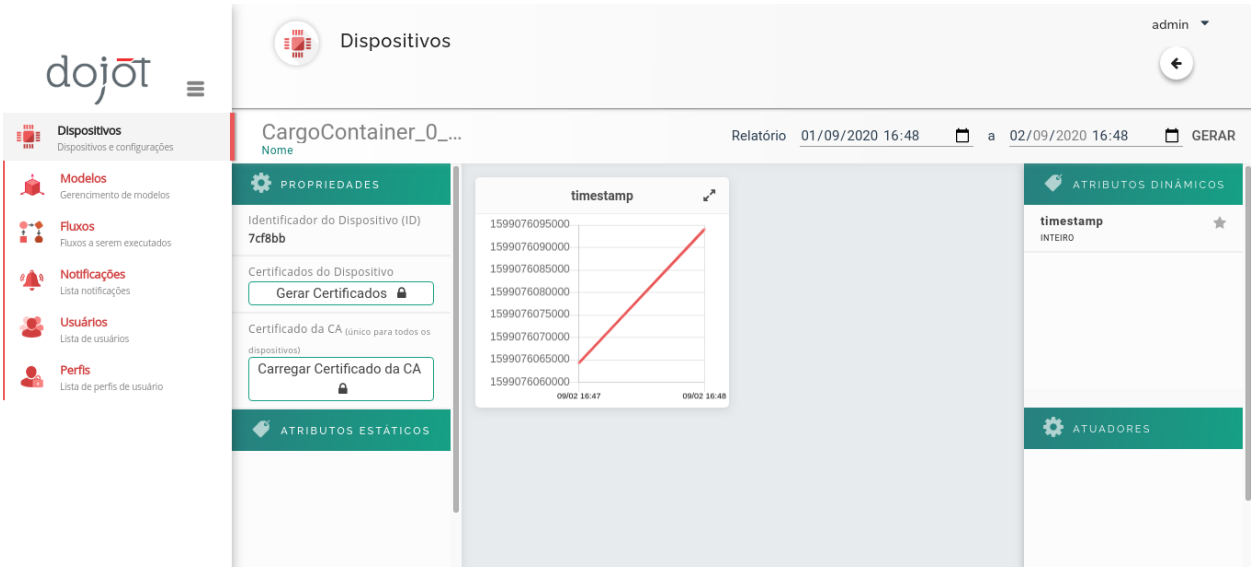


Fig. 14.5: : Um dispositivo de exemplo recebendo mensagens.

14.3 Rodando um teste distribuído

Para testes pequenos, o procedimento supracitado deve ser o suficiente, mas se você deseja realmente forçar a dojot, barreiras serão encontradas ao usar somente um *slave* e/ou uma máquina. Para solucionar este problema, o Locust provê o **modo distribuído**, permitindo a inicialização de múltiplos *slaves* em múltiplas máquinas, limitando o desempenho do Locust ao montante de poder de processamento (e orçamento!) que você tem.

Serão utilizadas duas máquinas virtuais rodando 4 *slaves* (2 em cada VM) e 1 *master* (em uma das VMs) para criar os 1.000 dispositivos falsos. Vamos nos referir à máquina com o *master* como **primária** e à outra como **secundária**.

Atenção: Não há uma correlação entre os números escolhidos: o número de clientes que cada *slave* suporta é dependente de diversas variáveis, como o número de núcleos de CPU disponíveis.

Dica: Recomendamos que você rode 1 *slave* por núcleo da CPU de sua máquina, i.e. se você tem uma VM com 4 CPUs, rode 4 *slaves*.

14.3.1 Configuração para o modo distribuído

Antes de rodar os testes, há mudanças a serem feitas nos docker compose do Locust. Note que você deve clonar o repositório da dojot em cada máquina em que o Locust rodará.

Dica: Como as configurações podem ser as mesmas para os *slaves* e para o script *generate_certs*, você pode os compartilhar entre máquinas usando o `sshfs`. Assumindo que você esteja na pasta do Locust na máquina secundária, rode o comando:

```
sshfs -o allow_other -o nonempty <user>@<ip>:/path/to/dojot/connector/mqtt/locust/
↪Docker Docker
```

Verifique a documentação do sshfs para mais detalhes.

Abra o arquivo `Docker/docker-compose-master.yml` na máquina primária e mude as seguintes variáveis de ambiente:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same␣
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"
```

Abra o arquivo `Docker/docker-compose-slave.yml` e mude as seguintes variáveis de ambiente:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same␣
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"

# If it's in the same machine as the master, you can leave as it is
LOCUST_MASTER_HOST: "locust-master"

# If it's in the same machine as the master, you can leave as it is
REDIS_HOST: "redis"
# Change to 6380 if the master is in another machine
REDIS_PORT: "6379"
```

Abra o arquivo `Docker/scripts/generate_certs/docker-compose.yml` e mude as seguintes variáveis de ambiente:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"

# If it's in the same machine as the master, you can leave as it is
REDIS_HOST: "redis"
# Change to 6380 if the master is in another machine
REDIS_PORT: "6379"
```

Como pode-se ver, as configurações mudaram pouco, sendo que as principais mudanças são somente na localização do Redis e do *master*.

14.3.2 Gerando os certificados

Nesta parte do tutorial, serão criados dispositivos falsos que, ao contrário do teste simples, não aparecerão na GUI da dojot.

Antes de rodar o script, precismos inicializar o Locust *master*. Dentro da pasta do Locust na máquina primária, rode:

```
docker-compose -f Docker/docker-compose-master.yml up
```

Após a inicialização, rode o docker compose do `generate_certs` na máquina **primária** e entre nela:

```
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml up -d
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml exec generate-
↪certs bash
```

Crie os certificados:

```
generate_certs cert --devices 1000
```

Nota: Os dispositivos falsos são simulados como certificados.

Agora entre em sua máquina **secundária**, inicialize o `generate_certs` e, dentro dele, rode:

```
generate_certs redis --export
```

Como os certificados são armazenados no Redis, você pode simplesmente exportá-los com o comando mostrado em qualquer máquina, prevenindo o trabalho tedioso de copiar em cada VM a pasta `cert` com os certificados.

14.3.3 Inicialização do slave

Rode em suas máquinas **primária** e **secundária**:

```
docker-compose -f Docker/docker-compose-slave.yml up --scale locust-slave=2
```

Este comando criará dois contêineres do Locust *slave* em cada máquina. Você deverá notar no Locust *master* mensagens confirmando a conexão de cada *slave*.

14.3.4 Rodando o teste distribuído

Está tudo pronto para a realização do teste. Para iniciá-lo, você deve acessar a interface do Locust no seu navegador através do domínio do Locust *master*, e.g. `localhost:8089`.

Digite `1000` em `Number of users to simulate`, `10` em `Hatch rate` e clique em `Start swarming`. Isso dirá ao Locust para rodar 1.000 clientes, criando 10 deles por segundo.

14.4 Usando o dashboard do Locust no Grafana

The Locust web interface is easy and simple to use, but there are some downsides. The major one is the persistence: The history data will be deleted as soon as you close or refresh the page.

Para resolver este problema, foi adicionada a imagem Locust Exporter no docker compose do Locust *master*, permitindo a exportação das métricas em um formato compatível com o Prometheus. Desta forma, é possível armazenar estas informações no Prometheus e centralizar todos os dashboards no Grafana. Infelizmente ainda é necessário acessar a interface web do Locust para começar o teste.

Atenção: Como a stack Grafana/Prometheus só está disponível na instalação por meio de Kubernetes, esta seção não é aplicável para a instalação por meio de docker compose. Nós o encorajamos a verificar o *Guia de instalação* para mais informações sobre os métodos de instalação da dojot.

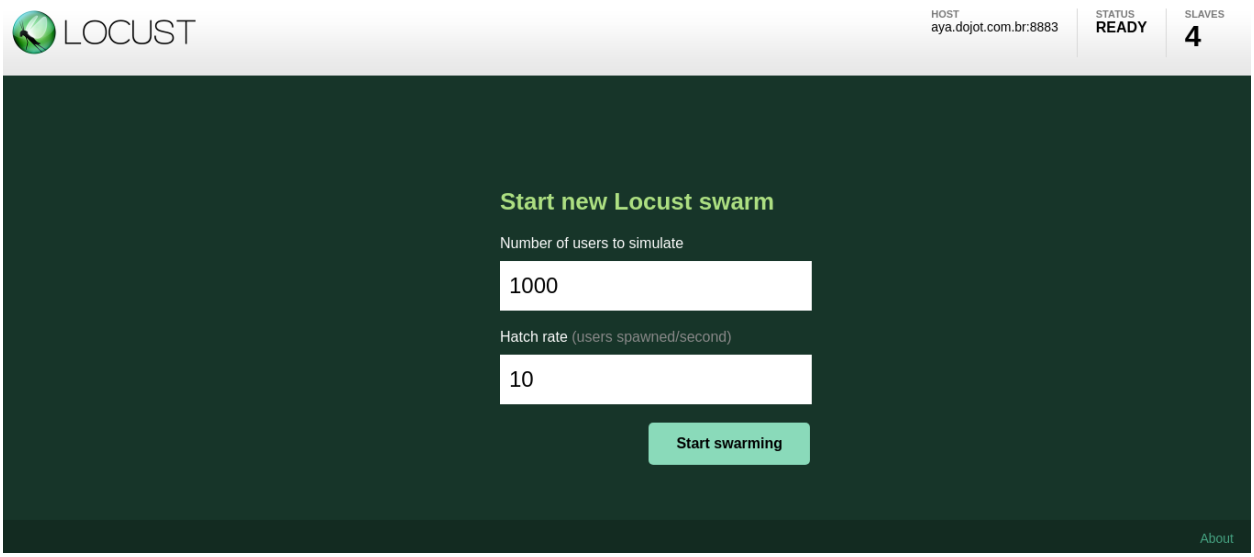


Fig. 14.6: : Configurando o Locust para rodar os clientes.

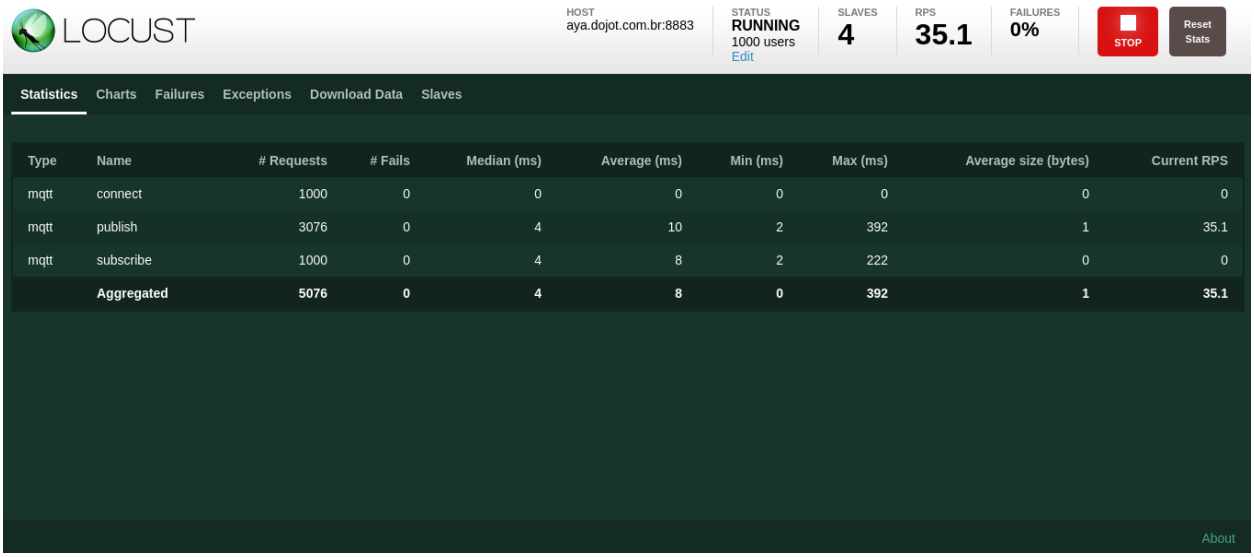


Fig. 14.7: : As estatísticas do Locust após rodar os testes por alguns minutos.

14.4.1 Configuração do Ansible

Você deve decidir onde o seu Locust *master* estará de antemão antes de rodar o *playbook* do Ansible. As configurações do Ansible que você precisa modificar para conectar o Locust Exporter com o Prometheus são:

```
dojot_enable_locust_exporter: true
dojot_locust_exporter:
  ip: 1.2.3.4
```

Mude o IP para o do Locust *master* e rode o *playbook*. Agora você pode inicializar um teste (distribuído ou não) como mostrado nas seções anteriores e deverá ver os dados do Locust sendo enviados ao dashboard do Locust no Grafana.

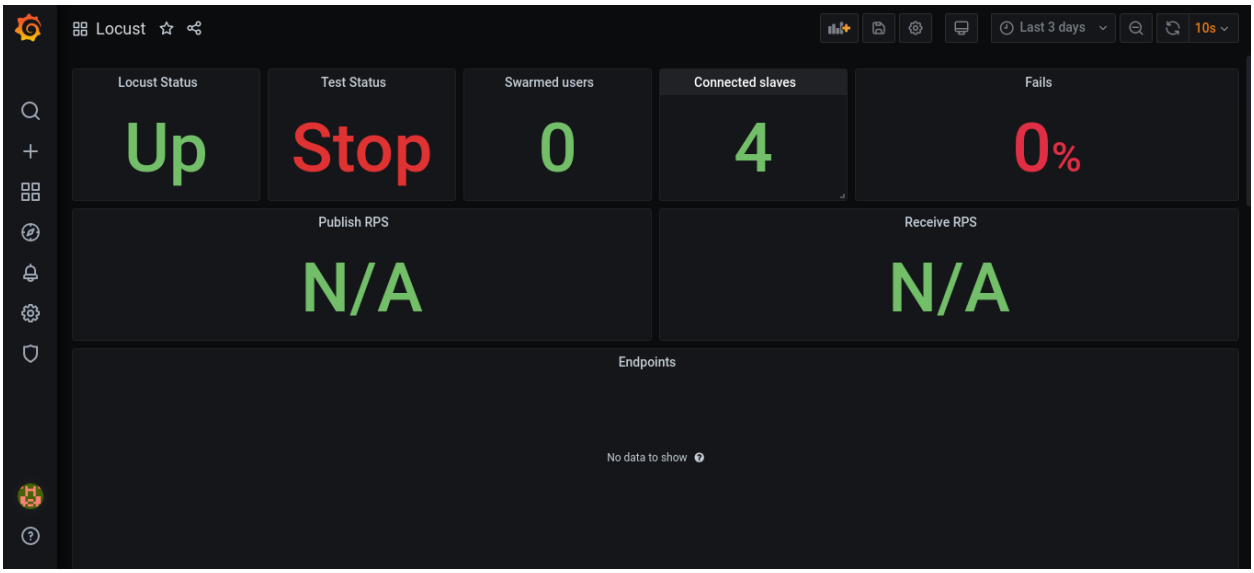


Fig. 14.8: Estatísticas do Locust no Grafana - antes de começar o teste.

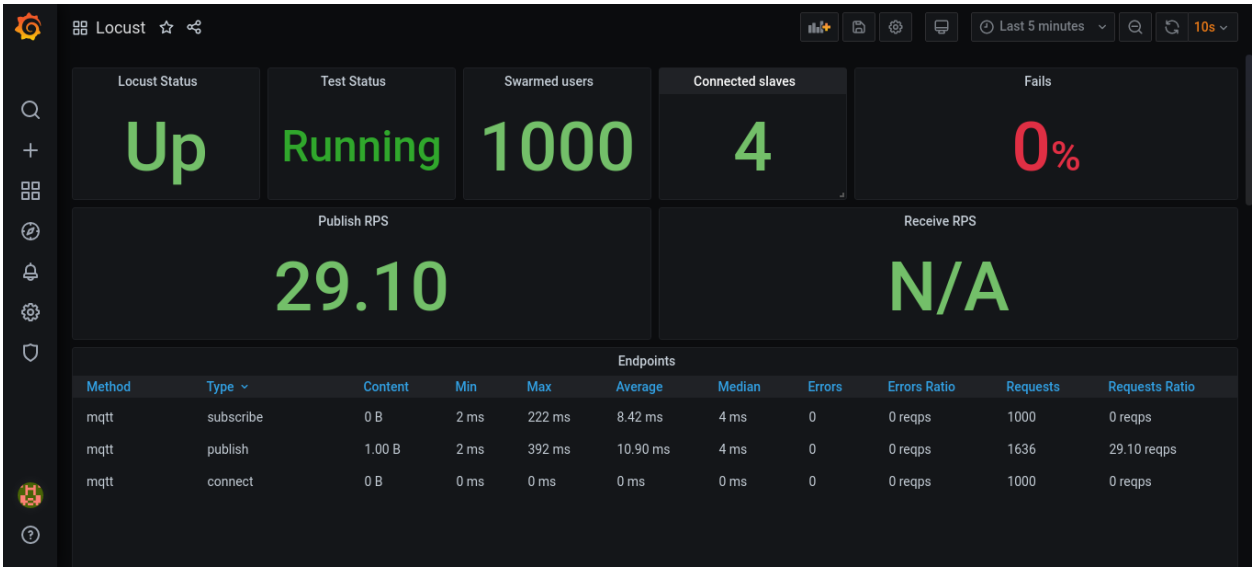


Fig. 14.9: Estatísticas do Locust no Grafana - depois da inicialização do teste.

14.5 Requisitos para um teste com 100.000 clientes

Agora que você sabe como rodar testes distribuídos utilizando o Locust, você consegue executar o teste com 100.000 clientes. Para tal, você precisará de bastante poder computacional e múltiplas máquinas, tanto para a dojot como para o Locust, O objectivo é chegar a 100.000 conexões MQTTS simultâneas com taxa de ~3.333 RPS (tanto para publicação quanto para recepção), i.e. uma mensagem a cada 30 segundos para cada cliente conectado.

Como este é somente um caso especial de teste distribuído, cobriremos somente as especificações e algumas dicas para o teste, já que o procedimento para configuração é o mesmo que nós já fizemos.

Nota: Como você já deve saber, este teste só é possível na instalação via Kubernetes da dojot.

14.5.1 Especificações de hardware

Para a plataforma dojot:

Nome da máquina	Serviços presentes	CPU	RAM
dojot-verne-1	VerneMQ, K2V e V2K	8	8GB
dojot-verne-2	VerneMQ, K2V e V2K	8	8GB
dojot-verne-3	VerneMQ, K2V e V2K	8	8GB
dojot-x509	x509 identity manager	4	4GB
dojot-kafka	Kafka e Zookeeper	6	6GB
dojot-dojot	O restante dos serviços	4	4GB
haproxy	Load balancer	4	4GB

Para o Locust, usaremos 5 réplicas da mesma máquina, com **14GB** de RAM e **9** CPUs.

14.5.2 Dicas gerais para o teste

- O Ansible tem a tag `100k` para preparar o ambiente para um teste com 100.000 clientes. Isto roda uma versão minificada da dojot, excluindo alguns serviços; tal estratégia foi adotada pois nem todos os serviços da dojot suportam tamanha carga.
- Ao compartilhar um volume persistente entre *pods* da EJBCA, é possível escalá-la para aumentar a velocidade de criação de certificados.
- A criação de certificados pode ser distribuída entre todas as máquinas de Locust. Em nosso exemplo - com 5 máquinas para o Locust - nós conseguimos gerar 20.000 certificados em cada máquina. Isso aumenta substancialmente a velocidade de criação de certificados quando aliado ao aumento de instâncias da EJBCA rodando.
- Após gerar os certificados, confirme que todas as máquinas estão com todos os certificados. Você pode os exportar rodando o comando `generate_certs redis --export` dentro do contêiner do `generate_certs`.
- Para verificar o número de certificados, rode:

```
ls cert | wc -l
```

  O valor retornado deve ser `200.003`. Este número inclui a chave e o certificado de cada dispositivo, o certificado da CA e as pastas `renew` e `revoke`.

- É altamente recomendado rodar um *slave* por núcleo, totalizando 45 *slaves* neste exemplo.

- Como a interface web do Locust não persiste nenhum dado, use o dashboard do Locust no Grafana para monitorar o seu teste. Verifique a seção anterior para mais informações sobre como configurar o Locust Exporter.
- Você também pode rodar o teste com `revoke` e `renew`. Verifique o README do repositório para mais detalhes de configuração.