

UNIVERSIDADE DO MINHO



MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

Definição de middleware “cloud based” para serviço a aplicações de monitorização de espaços físicos

ALUNO: JOSÉ FRANCISCO FERREIRA ALVES DE SOUSA A67724

ORIENTADO POR: ANTÓNIO NESTOR RIBEIRO

7 de Outubro de 2017

Conteúdo

1	INTRODUÇÃO	1
1.1	Objetivos	3
2	CONTEXTUALIZAÇÃO	4
3	ESTADO DE ARTE	6
3.1	Tecnologias	6
3.1.1	Apple Homekit	6
3.1.2	Android Things	8
3.1.3	Eclipse SmartHome	12
3.1.4	Arquitetura	15
3.2	Trabalho Relacionado	17
4	O PROBLEMA E OS SEUS DESAFIOS	22
4.1	Desafios	23
4.2	Funcionalidades	24
5	METODOLOGIA	25
5.1	Iterative and Incremental Development	25
5.2	Model Driven Development	26
5.3	Static Code Analysis	26
5.4	Continuous Integration	26
6	ARQUITETURA	29
6.1	Sistema	29
6.1.1	Hub	29
6.1.2	Api	30
6.1.3	Simuladores de Dispositivos	31
6.1.4	<i>Wrapper</i> da API	31
6.1.5	Aplicação Cliente	31
6.1.6	Visão Geral	32
6.2	Análise de Domínio	33
6.3	Design	36
6.3.1	Hub	36
6.3.2	Api	47

	Conteúdo
7	IMPLEMENTAÇÃO 58
7.1	Tecnologias 58
7.2	Detalhes 64
7.2.1	Hub 64
7.2.2	Api 67
8	DEMONSTRAÇÃO 77
8.1	Preparação 77
8.2	Execução 78
9	CONCLUSÃO 97
10	BIBLIOGRAFIA 99

Lista de Figuras

Figura 1	Arquitetura base dos elementos do Homekit	7
Figura 2	Exemplo de interfaces gráficas do Apple Home	8
Figura 3	Arquitetura base do Android Things	9
Figura 4	Arquitetura detalhada do Weave	10
Figura 5	Exemplo da interface gráfica do SmartHome	14
Figura 6	Exemplo dos controlos da divisão "garden"	14
Figura 7	Integração Travis & GitHub	27
Figura 8	Integração Travis & GitHub	27
Figura 9	Visão Geral dos Sistemas	32
Figura 10	Modelo de Domínio	33
Figura 11	Diagrama de classes: <i>Things</i>	38
Figura 12	Diagrama de classes: <i>Serviços</i>	39
Figura 13	Diagrama de classes: <i>Factories</i>	41
Figura 14	Diagrama de classes: <i>Controllers</i>	42
Figura 15	Diagrama de classes: <i>Serviços de descoberta de dispositivos</i>	43
Figura 16	Diagrama de classes: <i>Factories atualizadas para suportar serviços de descoberta</i>	44
Figura 17	Componentes preparados para a demonstração	78
Figura 18	Ecrãs de demonstração: <i>Login</i>	79
Figura 19	Ecrãs de demonstração: <i>Registo</i>	80
Figura 20	Ecrãs de demonstração: <i>Criar casa</i>	81
Figura 21	Ecrãs de demonstração: <i>Descoberta do hub</i>	82
Figura 22	Ecrãs de demonstração: <i>Lista de Casas</i>	83
Figura 23	Ecrãs de demonstração: <i>Lista de Dispositivos</i>	84
Figura 24	Ecrãs de demonstração: <i>Criar Dispositivo</i>	85
Figura 25	Ecrãs de demonstração: <i>Descoberta de Dispositivo</i>	86
Figura 26	Ecrãs de demonstração: <i>Descoberta de Dispositivo</i>	87
Figura 27	Ecrãs de demonstração: <i>Lista de Dispositivos</i>	88
Figura 28	Ecrãs de demonstração: <i>Ecrã de Cenário</i>	89
Figura 29	Ecrãs de demonstração: <i>Criação de um Scenario Thing</i>	90

Figura 30	Ecrãs de demonstração: Ecrã de Cenário válido	91
Figura 31	Ecrãs de demonstração: <i>Log</i> de aplicação de cenário	91
Figura 32	Ecrãs de demonstração: Criar <i>Timed Task</i>	92
Figura 33	Ecrãs de demonstração: Lista <i>Timed Tasks</i>	93
Figura 34	Ecrãs de demonstração: Logs <i>Timed Task</i>	93
Figura 35	Ecrãs de demonstração: Criar <i>Triggered Task</i>	94
Figura 36	Ecrãs de demonstração: Lista <i>Triggered Tasks</i>	95
Figura 37	Ecrãs de demonstração: Logs <i>Triggered Task</i>	95

Lista de Tabelas

Tabela 1	Tipos de items do SmartHome	15
Tabela 2	Ações REST	48

1. Introdução

Tem surgido recentemente *frameworks* existentes nos principais sistemas operativos de dispositivos móveis, iOS e Android, que pretendem reunir e disponibilizar informações sensoriais recolhidas em diversos ambientes, mas com predominância para o “espaço casa”. Além das *frameworks* existentes nos sistemas operativos referidos existem uma série de outras APIs proprietárias dos diversos fabricantes.

O objetivo desta dissertação consiste no estudo das diferentes APIs e na proposta arquitetural de um *mashup* “cloud based” que possibilite a gestão integrada das diversas APIs assim como a oferta de uma camada de serviço aplicacional para novas aplicações. Este *middleware* deverá ter mecanismo de registo de novas APIs, criar mecanismos de registo e, ainda, recolher informações de diversas APIs proprietárias.

Inicialmente, irá ser feita uma análise do estado da arte, incluindo um estudo sobre tecnologias existentes no campo da *Internet of Things* assim como trabalhos académicos relacionados. Ambos serão utilizados como base teórica para o desenvolvimento do projeto. As tecnologias a analisar em detalhe serão o Android Things e o Weave [Goo16a], a plataforma para desenvolvimento de soluções IoT e o protocolo de comunicação desenvolvidos pela Google, o Apple Homekit [App16], o serviço base para as aplicações orientadas a casas inteligentes da Apple, e, por fim, o Eclipse SmartHome, a *framework* open-source para desenvolvimento de soluções IoT orientadas às casas inteligentes.

O *middleware* a desenvolver deverá fazer a ligação entre os dispositivos do espaço casa e as aplicações cliente, oferecendo as funcionalidades dos dispositivos através de uma API que abstrai as componentes específicas dos equipamentos. A API deverá ter os serviços necessários para a descoberta e registo dos equipamentos dos utilizadores. Todos estes serviços devem estar disponíveis num serviço cloud-based, para disponibilizar o acesso remoto, mas seguro, aos dispositivos do utilizador, e para também oferecer integração com os vários serviços e aplicações já existentes na cloud.

O serviço fará uso de edge devices, pontos de entrada para a rede do utilizador, que irão agregar os dados dos dispositivos da rede local assim como efetuar a gestão de dispositivos. O serviço cloud em vez de comunicar com os dispositivos diretamente, comunica através

do edge device. Desta maneira, tarefas como a descoberta de dispositivos e agregação de dados serão feitas localmente, poupando assim os recursos computacionais do serviço cloud. Isto portanto assemelha-se a um panorama de edge computing [Cha+14], onde os dados são processados perto da fonte da origem, neste caso, as casas dos utilizadores.

Esta camada de serviço deverá oferecer mecanismos para o registo de novas API proprietárias, de modo a que novos dispositivos que possam surgir no mercado possam ser integrados no serviço sem grandes esforços. Vários protocolos de comunicação deverão ser suportados de modo a que se estenda o número de dispositivos compatíveis. Os protocolos mais comuns nesta área são o HTTP (*Hyper Text Transfer Protocol*), MQTT (*Message Queue Telemetry Transport*) e CoAP (*Constrained Application Protocol*).

Os utilizadores deverão ter a possibilidade de configurar as interações entre os seus dispositivos através de um sistema de "triggers", onde os dispositivos serão acionados com base no valor de outros dispositivos ou até serviços web, como por exemplo, ligar as luzes de uma sala a partir das 19 horas. Esta funcionalidade é bastante importante, uma vez que deixando esta componente de parte, perde-se grande parte da automação que se quer alcançar neste ambiente.

Pretende-se demonstrar a exequibilidade da abordagem criando uma aplicação móvel (iOS ou Android) que obtenha a informação pretendida através da utilização do *middleware* desenvolvido.

1.1 Objetivos

- Efetuar estudo sobre a oferta de dispositivos e soluções IoT, como as já acima referidas (Homekit, Brillo, entre outras), identificados as suas mais valias e problemas.
- Desenvolver uma API que opere na cloud, que unifique as diversas APIs e frameworks orientadas a IoT, para que possam ser usada por outras aplicações. Esta API deve basear-se das mais valias das frameworks estudadas, tentando resolver os seus problemas.
- Permitir a extensibilidade do serviço, de maneira a que possam ser adicionadas mais APIs proprietárias, mantendo o serviço atualizado sem grandes complicações.
- Desenvolver uma aplicação mobile que faça uso desta API e que permita demonstrar as capacidades da mesma, no que toca à abstração dos serviços proprietários e à automatização de comportamentos e gestão do espaço casa

2. Contextualização

Nesta última década vem se observando um crescimento bastante acentuado do mercado de dispositivos orientados ao espaço casa. Dispositivos que permitem automatizar diversas áreas de uma casa, como os estores, as portas, as luzes, o aquecimento e até sistemas de vídeo-vigilância. O número de dispositivos e as funcionalidades dos mesmos são bastante diversos, e todos têm a capacidade de serem controlados através de sistemas de informação, como por exemplo, *smartphones*.

Tais dispositivos compõem aquilo a que muitos chamam de “casas inteligentes”, que é uma das áreas da “Internet of Things”, ou IoT abreviadamente, que basicamente representa a rede de dispositivos físicos como sensores, eletrodomésticos, atuadores entre outros. Juntamente às “casas inteligentes” juntam-se outras áreas no top 5 da IoT, as plataformas cloud de IoT, automação industrial, indústria energética e “connected cities”.

De acordo com um questionário feito pelo IEEE [IEE16] a vários profissionais das tecnologias de informação, cerca de 46% dos correspondentes estavam a trabalhar na área das IoT e outros 29% tinham planos para desenvolver soluções IoT nos próximos 18 meses. Isto vem confirmar a crescente tendência em optar estas novas tecnologias, que pretendem melhorar a interação entre os vários dispositivos “inteligentes” que compõem o mundo.

Estes novos dispositivos inteligentes são normalmente acompanhados por aplicações clientes feitas pelos próprios fabricantes, que permitem, no mínimo, ativar e desativar o dispositivo e recolher a informação sensorial do mesmo, caso tenha esta funcionalidade. Nem todos os dispositivos recolhem informação do meio, por exemplo, uma fechadura eletrónica ou um dispositivo de iluminação.

No entanto, com a crescente utilização de dispositivos nas residências dos utilizadores, a utilização de várias aplicações proprietárias para gerir cada um destes dispositivos torna-se inconveniente [Ngu+16]. Daí a conceção de frameworks para gestão de dispositivos orientados à *home-automation*. Um dos exemplos mais conhecidos é o Apple Home, que permite controlar os dispositivos instalados numa residência através de qualquer dispositivo iOS. O utilizador adiciona os dispositivos ao Homekit, a base de dados por detrás do Apple Home, e passa a ter um local centralizado onde pode gerir os aparelhos da sua residência. No en-

tanto, esta framework não está disponível noutros sistemas operativos, e apenas suporta uma distinta gama de dispositivos, não sendo totalmente aberta.

Esta variedade de dispositivos e tecnologias nesta área provoca também alguns entraves ao desenvolvimento de novas soluções, uma vez que, não existem soluções “standard” para o seu desenvolvimento. Existem implementações que utilizam vários tipos de tecnologia, tornando difícil a interoperabilidade entre estes novos produtos. De notar que uma das filosofias da IoT é a interconectividade entre estes dispositivos e sem interoperabilidade entre dispositivos, torna-se impossível de alcançar tal coisa.

Algumas tecnologias têm vindo a surgir recentemente, como o Raspberry PI, que é uma excelente placa de prototipagem, que possui um circuito com 24 pinos de GPIO, que permite controlar qualquer tipo de aparelhos através desta placa. No campo de bibliotecas de software, o Eclipse SmartHome parece ser uma tecnologia interessante, semelhante ao Homekit, mas orientada para programadores e para o desenvolvimento de novos produtos, e o Google Brillo e o Weave parecem tecnologias interessantes orientadas à IoT baseadas no ecossistema da Google.

Este exemplo da Google demonstra um cenário “cloud-based”, o que é uma grande vantagem, uma vez que parte do esforço computacional é retirado do utilizador e dos seus equipamentos, e também centraliza a sua informação num serviço que pode ser acedido de qualquer lugar, desde que possua ligação à Internet. A quantidade de oferta de serviços de alojamento cloud torna o desenvolvimento nesta área bastante mais conveniente. Além disso, provedores de serviço cloud como a Amazon oferecem serviços especializados para IoT, assim como integração total com o resto dos seus serviços, mais uma vez demonstrando como o futuro da IoT reside na cloud.

De referir, que apesar disto ser um projeto a visar o espaço casa, as conclusões e resultados podem ser aplicados também ao espaço empresarial e industrial, onde os casos de uso são semelhantes, mas com um maior volume de dados e por vezes interações específicas, como as se verificam com os equipamentos industriais. Um caso de uso onde a *home automation* encaixa bastante bem é no espaço empresarial, por exemplo, nos escritórios, onde a regulação de equipamentos como ar condicionados, alarmes, sistemas de videovigilância, portas e fechaduras são aspetos de grande importância.

3. Estado de Arte

Existem várias frameworks, bibliotecas de software e estudos feitos na área da “Internet of Things” e da cloud relevantes para este projeto. Para obter as tecnologias mais relevantes nesta área foi analisado um estudo [IEE16] feito pelo IEEE sobre tecnologias IoT utilizadas por engenheiros a nível mundial. A partir destes resultados, as tecnologias mais relevantes foram escolhidas e um resumo geral sobre as mesmas foi elaborado.

3.1 Tecnologias

3.1.1 Apple Homekit

O Apple Homekit não é uma aplicação por si só, é uma espécie de base de dados semelhante ao Passkit e ao Healthkit também desenvolvidos pela Apple. Todas estas “kits” são utilizados pelas respetivas aplicações, neste caso o Homekit tem a aplicação Home, o PassKit tem o Passbook e o HealthKit o Health.

O Homekit neste caso tem como função armazenar uma coleção de dispositivos relativos a casas inteligentes, oferecendo ainda funcionalidades mais avançadas como *triggers*, onde é possível configurar um grupo de ações a executar com base em certos *triggers*, como a hora, a localização do utilizador ou com base nos dados de algum dispositivo. Por exemplo, é possível configurar o sistema de modo a que ele ligue as luzes e abra a porta principal quando o utilizador estiver nas imediações da casa, ou até, ligar ou desligar o aquecimento a uma certa hora do dia.

O Homekit possui também integração com a Siri, o assistente pessoal presente nos sistemas operativos da Apple, o que permite controlar os dispositivos registados no Homekit através de comandos de voz, como por exemplo, “Siri fecha a porta da garagem”.

3.1.1.1 Arquitetura

Os dados no Homekit são estruturados da seguinte forma hierárquica:

- Homes - são a unidade base de dados, que representam uma casa do utilizador. Um utilizador pode possuir várias casas geograficamente afastadas ou até dividir a sua casa em duas, como por exemplo, casa principal e anexo.
- Rooms - representam as salas ou quartos de uma casa. As salas no Homekit apenas possuem um nome que as identifica, como "sala" ou "cozinha", de maneira a que se possam executar comandos deste tipo: "ligar as luzes da sala".
- Accessories - são os dispositivos ou acessórios instalados nas casas e registados em salas específicas. Caso os acessórios não sejam registados em nenhuma sala, o Homekit coloca-os num "default room", um local especial para dispositivos sem uma sala específica.
- Services - são os serviços oferecidos por um dispositivo, como por exemplo, um serviço para ligar a luz de uma lâmpada. Os dispositivos podem também ter serviços internos, como um atualizador de sistema.

De notar que um único dispositivo pode oferecer vários serviços controlados pelo utilizador. Uma porta pode ter um serviço para a fechar e outro serviço para a abrir.

- Zones - são agrupamentos opcionais de salas, como por exemplo, "1º andar" ou "garagem". As zonas, tal como as salas, possuem apenas um nome. As zonas permitem ao utilizador utilizar comandos como, "Siri, liga as luzes do 1º andar", que irá ligar todas as luzes das salas do 1º andar.

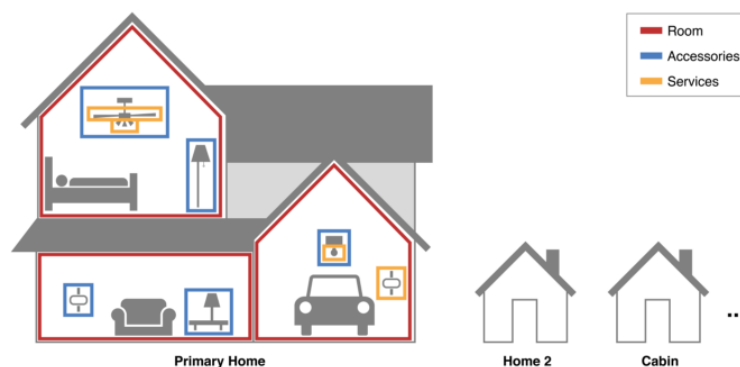


Figura 1: Arquitetura base dos elementos do Homekit

De seguida, um exemplo dos interfaces que demonstram os controlos do Apple Home, a aplicação que complementa o Homekit.

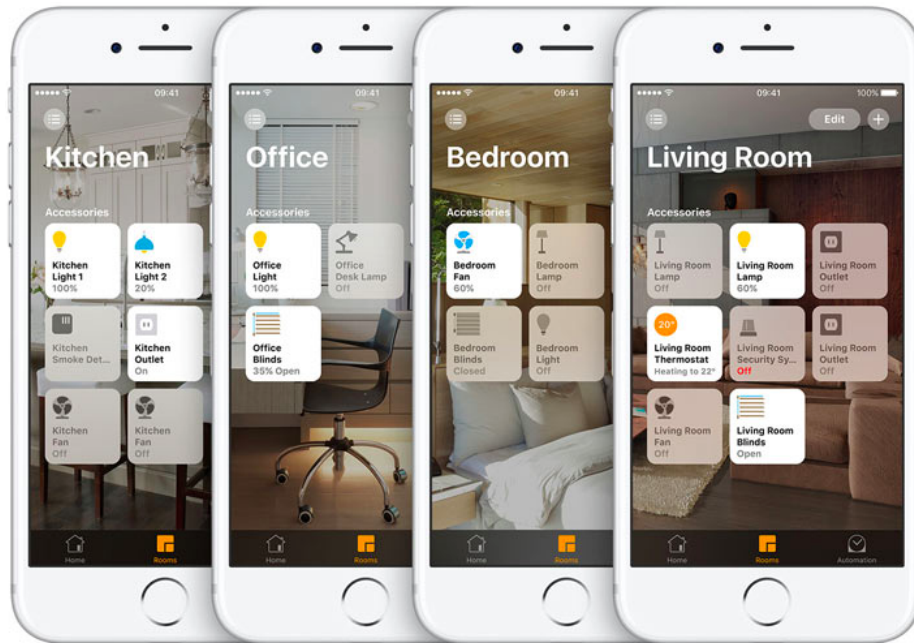


Figura 2: Exemplo de interfaces gráficas do Apple Home

3.1.1.2 API e interoperabilidade

A API oferecida pelo Homekit permite o controlo total da base de dados do utilizador mediante a autorização do mesmo. Além disto, apenas aplicações revistas pela Apple e distribuídas na AppStore é que podem aceder à API do Homekit. Isto não é entrave nenhum ao desenvolvimento no entanto, uma vez que o XCode, o ambiente de desenvolvimento da Apple, possui um módulo de simulação, onde é possível adicionar dispositivos de testes, para testar a funcionalidade de aplicações "Homekit enabled". Resumidamente, tudo o que é possível fazer através da aplicação Home, é possível através desta API programaticamente.

De notar, que a API possui apenas um SDK para Swift ou Objective-C, portanto, o desenvolvimento nesta tecnologia só pode ser feito no ecossistema da Apple, o que é um ponto a menos na interoperabilidade desta tecnologia.

3.1.2 Android Things

O Android Things, originalmente chamado de Google Brillo, é um projeto orientado à "Internet of Things" desenvolvido pela Google, que tem como objetivo fornecer um sistema operativo para sistemas embebidos assim como um protocolo de comunicação, chamado Weave, para comunicarem com a cloud e dispositivos móveis.

Este projeto é uma versão básica do sistema operativo Android, possuindo apenas os serviços "core" do mesmo, de maneira a que possua uma utilização de memória bastante reduzida, sendo ideal para sistemas embebidos, e tendo todas as ferramentas do SDK Android e interfaces programáticos de acesso aos serviços da Google.

Todos os dispositivos utilizando o Android Things podem e devem ser conectados aos serviços cloud da Google, para que estejam sempre com as últimas atualizações dos serviços do sistema operativo. Além disso, os sistemas da Google recolhem constantemente informações do funcionamento dos dispositivos, via o protocolo Weave, e assim é possível aos programadores monitorizarem várias métricas de todos os seus dispositivos, como por exemplo, crash reports.

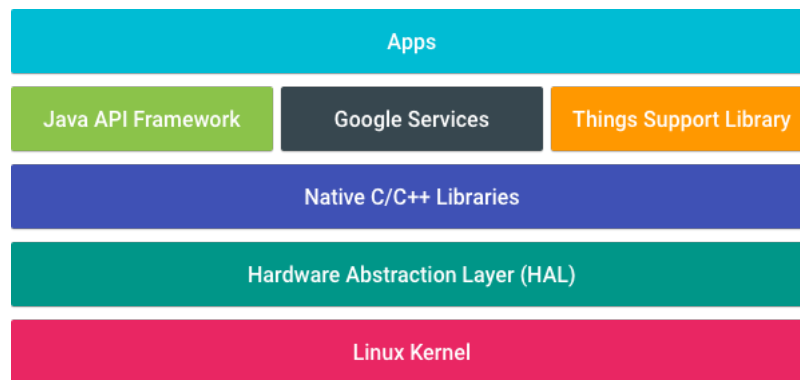


Figura 3: Arquitetura base do Android Things

3.1.2.1 Weave

O Weave é um protocolo de comunicação orientado à IoT, que permite que dispositivos desta área possam comunicar com smartphones e a cloud. O protocolo tem como objetivo abstrair as funções dos dispositivos IoT de modo a que possam ser reutilizado por várias aplicações e serviços web.

Este protocolo utiliza JSON como formato e a transmissão pode ser feita em HTTPS ou em XMPP. Além disso, este protocolo é protegido com SSL/TLS e a autenticação de dispositivos é feita utilizando OAuth 2.0 nos servidores de autenticação da Google. Utilizando este esquema de autenticação, a Google pode possibilitar a partilha destes dispositivos entre diferentes utilizadores, um pouco semelhante à partilha de documentos do Google Drive.

O Weave tem como objetivo fornecer um meio de comunicação entre dispositivos, smartphones e a cloud. A integração na cloud é um ponto muito forte, uma vez que com esta tecnologia aplicações poderão interagir com os dispositivos dos utilizadores, depois da autorização dos mesmos.

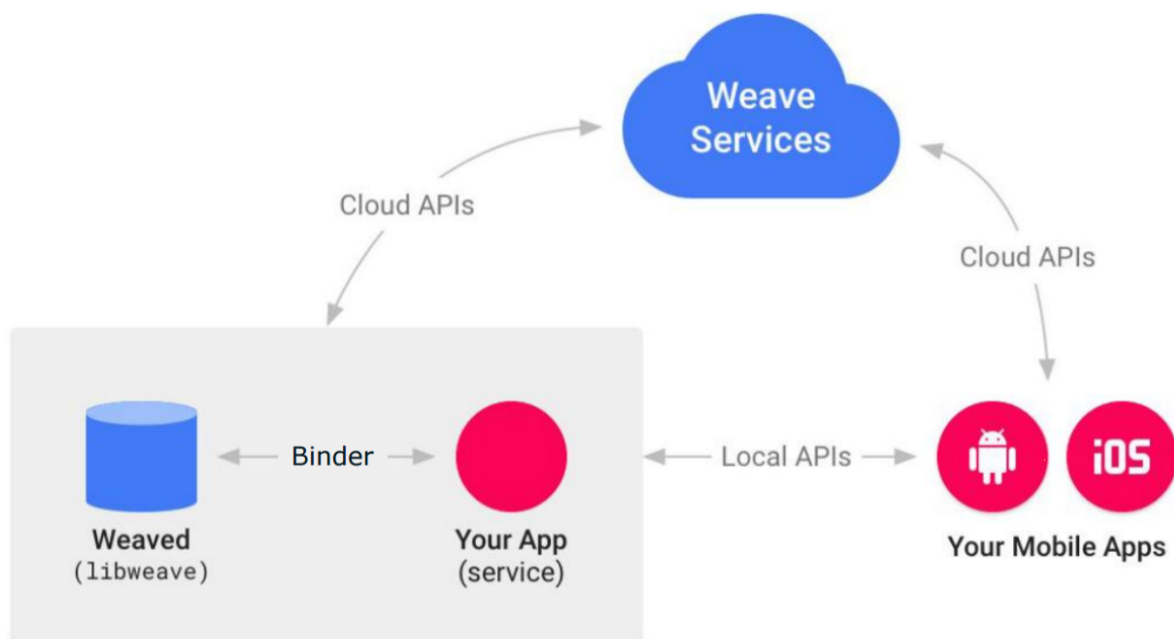


Figura 4: Arquitetura detalhada do Weave

De seguida, uma descrição das funcionalidades do Weave que se destacam:

- **Auto-configuração** - instalação de novos dispositivos podem ser feitas através de dispositivos Android (telemóveis) onde o utilizador efetua a autenticação do dispositivo e é redirecionado para a página da aplicação, caso o dispositivo a tenha.
- **Monitorização de dispositivos** - através do portal do programador dos serviços Google, é possível analisar estatísticas dos dispositivos que utilizam Weave, nomeadamente, número de dispositivos online, distribuição de versões, distribuição de modelos de dispositivo, comandos enviados por hora.
- **Over-the-air updates** - esta funcionalidade apenas funciona em dispositivos que utilizem o sistema operativo Brillo e consiste na atualização automática de dispositivos a partir do portal do programador da Google.
- **Crash reports** - os programadores podem consultar o número de crashes ocorridos nos seus dispositivos assim como o número de utilizadores afetados e um log detalhado do crash em si.
- **Dev Console** - é possível interagir com dispositivos protótipos, podendo testar comandos Weave sem ter que construir uma aplicação cliente. Muito útil para testar a fazer debug dos dispositivos antes de os colocar em produção.
- **Home Assistant** - utilizando os serviços cloud do Weave, é possível fazer integração com o Home Assistant da Google, uma inteligência artificial que pode ser comandada

com comandos voz. Desta maneira, o utilizador pode controlar os seus dispositivos através da sua própria voz, sem ter que recorrer à utilização física das aplicações.

3.1.2.2 API e Interoperabilidade

O Weave possui uma API que pode ser utilizada quer por dispositivos, quer por aplicações clientes nos dispositivos móveis ou até browsers web no desktop. Os utilizadores devem autorizar o acesso das aplicações clientes aos seus dispositivos, e depois disso, as aplicações clientes podem então interagir com os serviços do utilizador. O *libweave* possui wrappers em várias linguagens nomeadamente Java e Go, além da versão original em C/C++.

Quanto ao Android Things, este oferece os serviços core do sistema operativo Android via o SDK em Java do mesmo, tal como se fosse uma aplicação mobile, mas apenas com os serviços core, não possuindo serviços como os contactos, telefone, mensagens, entre outros. O SDK providencia em vez disso, interfaces para acesso a serviços da Google, nomeadamente:

- **Chromecast** - serviço de streaming de telemóveis, tablets e computadores para dispositivos com suporte a esta tecnologia, por exemplo, sistemas de som ou televisões;
- **Google Drive** - serviço de armazenamento de arquivos e edição de documentos;
- **Firebase** - plataforma cloud da Google para suporte ao desenvolvimento de aplicações, contendo várias funcionalidades como: *analytics*, *crash reporting*, base de dados tempo real, configuração remota de aplicações e armazenamento de ficheiros;
- **GoogleFit** - plataforma de monitorização de saúde e *fitness* que recolhe dados de várias aplicações e dispositivos do utilizador para elaborar um perfil do utilizador com a sua performance em várias atividades físicas;
- **Instance ID** - serviço de validação de aplicações para Android, iOS ou extensões do navegador Chrome. Esta API permite gerar tokens para autenticar o acesso a recursos *server side*. Este mecanismo garante que os recursos *server side* possam ser apenas acedidos pelas aplicações clientes registadas neste serviço;
- **Awareness and Location** - esta API utiliza os sensores presentes no dispositivo para obter informações sobre os utilizadores das mesmas assim como o ambiente em geral;
- **Nearby** - este serviço expõe métodos de *publish* e *subscribe* baseados na proximidade. Dispositivos próximos podem então recorrer a esta API para efetuarem comunicação em tempo real. Este serviço permite também efetuar *service discovery*, sendo possível reconhecer dispositivos na rede local.

- **Places** - através deste serviço é possível aceder a varias informações sobre locais no mundo, distribuídos por categorias, podendo por exemplo, localizar supermercados num raio de 5km à volta do utilizador;
- **Mobile Vision** - esta API é um serviço de reconhecimento facial, podendo identificar rostos humanos em fotos, vídeos e transmissões em direto. A API também é capaz de identificar elementos detalhados dos rostos humanos, detetando se uma pessoa esta de olhos fechados ou a sorrir por exemplo.

O sistema operativo pode ser instalado numa gama de placas de prototipagem seleccionadas, como o Intel Edison ou o Raspberry Pi 3, mas também pode ser compilado e instalado em qualquer tipo de dispositivo.

O projeto situa-se em *open-beta*, e pode ser acedido via o portal para programadores da Google [Goo16a].

3.1.3 Eclipse SmartHome

O EclipseSmart Home é uma framework para o desenvolvimento de produtos direcionados a casas inteligentes. É “open-source” e completamente livre ao público geral. A filosofia dos criadores desta framework assenta no facto que a única maneira de ter uma framework verdadeiramente extensível, aberta e preparada para a evolução constante deste mercado, é tornando a mesma “open-source”, para que qualquer indivíduo possa contribuir e para que não existam exclusividades proprietárias, como acontece noutras frameworks da área.

Esta framework é compatível com qualquer tipo de sistema que consiga correr a Java-VirtualMachine. É uma framework OSGi 4.2+, sendo compatível com por exemplo o RaspberryPI, e um número de outras placas de prototipagem. Isto é uma grande vantagem no desenvolvimento, já que é possível efetuar o desenvolvimento para esta framework em MacOS, Linux e Windows.

O SmartHome é orientado aos fabricantes de hardware, que podem preparar os seus dispositivos para serem compatíveis com soluções SmartHome, aos fabricantes de aplicações, que podem desenvolver as suas soluções para serem baseadas no SmartHome e tendo assim compatibilidade com uma vasta gama de dispositivos já existentes no mercado. Por fim, os programadores, que podem desenvolver extensões para o SmartHome que serão compatíveis com qualquer tipo de produto que utilize o SmartHome como base e até poderão disponibilizar estas extensões em plataformas comerciais.

De acordo com o inquérito [IEE16] a programadores realizado pela IEEE sobre tecnologias utilizadas nesta área, o Eclipse SmartHome é a tecnologia mais utilizada, provando a grande comunidade por detrás desta framework.

Esta framework é bastante completa, sendo compatível com grande parte das soluções IoT existentes no mercado, e é além disso, completamente extensível. Existem vários *plugins* desenvolvidos pela comunidade que oferecem compatibilidade para todo o tipo de dispositivos.

De seguida, um pequeno resumo sobre as *features* desta framework:

- **Configuração Textual** - é possível configurar os dispositivos presentes num sistema a base de ficheiros texto. O Eclipse SmartHome possui uma DSL própria, e oferece ao mesmo tempo uma aplicação para editar ficheiros que utilizam esta linguagem, o SmartHome Designer, que oferece mecanismos *auto-completion* e de validação.
- **Áudio e Voz** - o SmartHome oferece mecanismos de *Text-to-speech* e de *Speech-to-text* que permitem aos utilizadores, teoricamente, controlar o sistema utilizando comandos voz, utilizando o *Human Language Interpreter* desenvolvido pela Eclipse. De momento, os idiomas suportados são o Inglês e o Alemão.
- **Rest API** - a framework oferece uma Rest API que permite a serviços externos interagirem com o sistema via pedidos HTTP. A API permite listar e configurar dispositivos e ainda oferece suporte a *Server Sent Events*[Hic15], para que as aplicações clientes possam ser notificadas em tempo real de alterações dentro do sistema.
- **Regras** - essencial para qualquer sistema de *home-automation*, com esta framework é possível definir regras que automatizam o sistema, por exemplo, fechar os estores quando o vento está acima de uma certa velocidade ou fechar as portas da garagem a partir de uma certa hora.

A framework deixa definir estas regras textualmente, um exemplo:

```
ON item_id state changed
  IF item_id.state == desired_value
    THEN item_id2.state = desired_value2
```

- **Internacionalização** - todos os elementos desta framework podem ser internacionalizados, desde que sejam providenciados os corretos ficheiros, neste caso, ficheiros de propriedades i18n Java.
- **Interfaces Gráficas** - esta framework oferece bastantes elementos para o desenvolvimento de interfaces gráficas, como ícones, widgets, entre outros. Além disso, existem mecanismos para definir interfaces através de ficheiros texto. Estes ficheiros são *site-maps*, que define que dispositivos e que informações sobre os mesmos e mecanismos de controlo aparecem no interface gráfico.

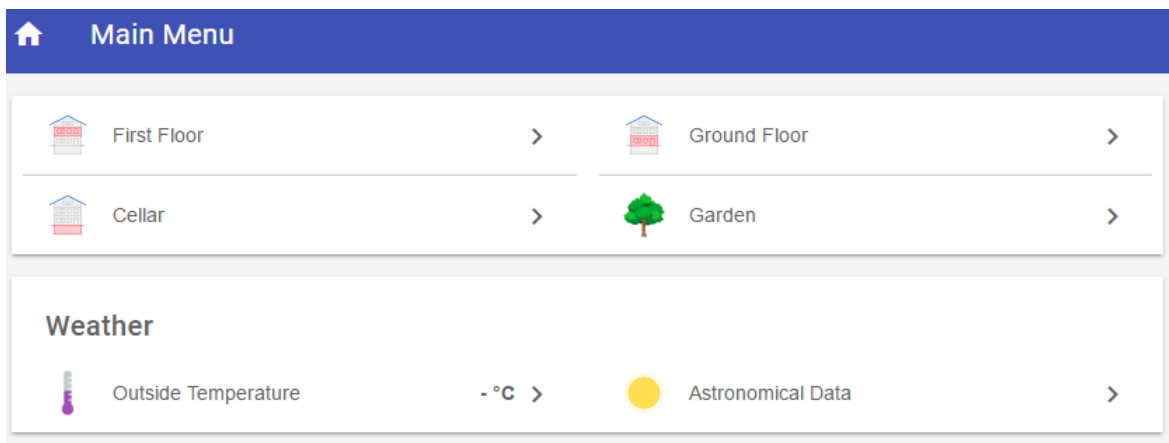


Figura 5: Exemplo da interface gráfica do SmartHome



Figura 6: Exemplo dos controlos da divisão "garden"

3.1.4 Arquitetura

O Eclipse SmartHome diferencia a sua arquitetura em dois paradigmas, o paradigma físico, que representa o estado físico do sistema, ou seja, dos dispositivos e serviços ligados aos SmartHome, e o paradigma funcional, que representa as funcionalidades do sistema disponíveis ao utilizador. Resumidamente, o paradigma físico é útil para a configuração do sistema e resolução de problemas, e o paradigma funcional é útil para a interação entre os utilizadores e o sistema.

A nível aplicacional, o paradigma físico é representado por **Things** e o paradigma funcional é representado por **Items**.

Things são entidades que podem ser adicionadas fisicamente ao um sistema e que podem fornecer várias funcionalidades ao mesmo tempo. **Things** podem ser tanto equipamentos reais que se podem adicionar um sistema como serviços web, como um serviço de meteorologia por exemplo. As **Things** oferecem a sua funcionalidade através de **Channels**.

Items representam as funcionalidades utilizadas pelas aplicações, neste caso, os interfaces gráficos e a lógica de automação. Estes **Items** são porventura ligados a **Thing Channels** através de links.

De seguida, os tipos de **Items** presentes nesta framework:

Itemname	Description	Command types
Color	Color information (RGB)	OnOff, IncreaseDecrease, Percent, HSB
Contact	Item storing status of e.g. door/window contacts	OpenClose
DateTime	Stores date and time	
Dimmer	Item carrying a percentage value for dimmers	OnOff, IncreaseDecrease, Percent
Group	Item to nest other items / collect them in groups	
Number	Stores values in number format	Decimal
Player	Allows to control players (e.g. audio players)	PlayerPause, NextPrevious, RewindFastforward
Rollershutter	Typically used for blinds	UpDown, StopMove, Percent
String	Stores texts	String
Switch	Typically used for lights (on/off)	OnOff

Tabela 1: Tipos de items do SmartHome

Os vários dispositivos e serviços web são então disponibilizados na framework recorrendo a **Bindings**, que disponibilizam estes serviços através dum conjunto de **Things**.

Além disso, o SmartHome alia este mecanismo com o seu serviço de **Inbox & Discovery**, que permite à framework procurar dispositivos compatíveis na rede e a adicionar-los a uma **Inbox**, para posterior configuração.

No entanto, a verdadeira configuração dos sistemas são feitos através de configuração textual, onde se podem especificar as **Things** dum sistema, juntamente com as suas configurações, como por exemplo, endereços IPs e afins. Além disto, também são especificados que tipos de **Items** irão ser retirados dos **Thing Channels**

De seguida, um pequeno exemplo de como a **Binding** do YahooWeather funciona.

A **binding** disponibiliza o serviço de meteorologia sobre o formato de um **thing**, que tem como parametros de configuração a **location**, local a analisar, e o outro parâmetro é o **refresh**, que define a taxa de atualização em segundos dos dados da meteorologia.

Esta **thing** possui então 3 **channels**:

Channel ID	Item Type	Description
temperature	Number	The current temperature in degrees Celsius
humidity	Number	The current humidity in %
pressure	Number	The current pressure in millibar (hPa)

Comparando esta situação a um paradigma orientado a objeto, os **channels** são semelhantes a métodos e a **thing** representa um objeto, com os **items** a atuar quase como tipos de dados.

De seguida, a configuração textual deste exemplo:

Things

Configuração do serviço de meteorologia.

```
yahooweather:weather:berlin [ location=638242 ]
```

Items

Definição do **Item Temperature**, que irá conter a temperatura retornada pelo serviço de meteorologia.

Number Temperature

"Outside Temperature"

```
{ channel="yahooweather:weather:berlin:temperature" }
```

Sitemap

Definição do interface gráfico associado a este exemplo. Neste caso iremos ter um menu chamado *Main Menu* e um elemento com o valor presente no **Item Temperature**

```
sitemap demo label="Main Menu"
{
    Frame {
        Text item=Temperature
    }
}
```

3.2 Trabalho Relacionado

A "Internet of Things" é já um campo com extensos estudos e investigações. Nesta secção irão ser analisados alguns artigos de relevância para este projeto.

No campo de middleware e camadas de serviço orientadas a IoT, Nastic et al. [NTD16] têm como objetivo o desenvolvimento de um *middleware* para o provisionamento de sistemas IoT cloud, para uma maior flexibilidade do consumo de recursos computacionais. Os investigadores avaliam a exequibilidade deste projeto utilizando aplicações do mundo real, neste caso, aplicações de gestão e controlo de edifícios.

As vantagens da computação em cloud são muitas, nomeadamente o provisionamento de recursos computacionais *on-demand*, onde os utilizadores podem dinamicamente alocar recursos para as suas necessidades. Este modelo computacional permite reduzir os custos de infraestrutura para a manutenção de sistemas de informação, recorrendo a estes serviços cloud em vez de infraestruturas locais físicas. Dadas estas vantagens era de esperar que grande parte das soluções IoT tirassem partido destas soluções, no entanto, grande parte das soluções IoT estão a recorrer aos serviços cloud para virtualização de *edge devices* como *IoT gateways*, acabando por adotar uma abordagem de "edge computing".

Apesar desta abordagem fazer o correto uso do fácil provisionamento de recursos computacionais através da virtualização de dispositivos, este método não é flexível, uma vez que cada dispositivo virtualizado é criado para executar uma tarefa específica, impedindo que os recursos na cloud sejam consumidos de forma genérica sendo necessário criar novos mecanismos de configuração e deployment, específicos para cada caso de uso. Este artigo tem como objetivo então tentar conceber um middleware de provisionamento de sistemas cloud

orientados à IoT, completamente abstrato e genérico que pode ser utilizado numa variedade de casos de uso.

Grande parte das abordagens de provisionamento de recursos cloud não providenciam soluções totais para as características dos sistemas que fazem uso das infraestruturas cloud, nomeadamente a variedade, a distribuição geográfica e dimensão destes. Muitas vezes durante a implementação de tais sistemas é preciso recorrer a vários softwares de provisionamento para alocar recursos na cloud, além disso, muitas vezes é necessário a configuração manual dos “edge devices”, e por vezes, presença física no local para configurar tais dispositivos.

Este trabalho de investigação acaba por propor uma arquitetura para um middleware de provisionamento de recursos cloud, que permite a configuração automática de edge devices. O middleware providencia mecanismos para o provisionamento e auto-configuração de “edge devices”, assim como a habilidade para os customizar e configurar de acordo com as necessidades de cada utilizador. Os “edge devices” efetuam assim a comunicação com os dispositivos IoT na sua rede e a “cloud” trata da gestão e configuração destes “edge devices”.

Ainda no panorama do “edge computing”, Nakamura et al. [Nak+16] referem que se está a tornar possível, com a tecnologia de hoje em dia, recolher qualquer tipo de informação do ambiente. Isto leva a que se queira recolher os dados gerados continuamente por estes dispositivos IoT em tempo real. O objetivo deste estudo consiste então no desenvolvimento de um middleware onde se processe estas streams de dados em tempo real e de forma distribuída.

O middleware fornece as seguintes funções:

- Distribuição de tarefas invocadas pelas aplicações clientes para sub-tarefas e a execução distribuída destas sub-tarefas pelos dispositivos IoT;
- Distribuição das streams de dados através de dispositivos IoT;
- Análise em tempo real das streams de dados;
- Integração de sensores e atuadores.

O middleware foi testado num Raspberry Pi para demonstrar a sua performance.

Muita das soluções existentes consistem no uso da cloud para recolher e processar todos os dados emitidos por estes dispositivos IoT. Desta forma os dados tem que ser processados antes de poderem ser devolvidos de volta às aplicações clientes, demorando mais tempo e utilizando mais poder computacional. Este modelo não é muito adequado para comunicação e feedback em tempo real, sendo portanto necessário uma alternativa, que utilize o poder

computacional dos próprios dispositivos para processar estes dados, sem ser preciso o uso de um sistema cloud complexo, assemelhando-se ao paradigma de Edge Computing.

Esta abordagem ainda faz mais sentido hoje em dia, uma vez que os dispositivos IoT estão cada vez mais evoluídos no que toca a recursos computacionais, e assim, utiliza-se estes recursos em vez de sobrecarregar os serviços cloud.

Também foram feitos extensos estudos sobre os problemas existentes no campo da IoT no que toca a funcionalidades e segurança. Ngu et al. referem na sua investigação [Ngu+16] a necessidade e a importância do middleware, a camada de ligação entre os dispositivos IoT e as aplicações clientes, referindo um exemplo do mundo real. Além disso, é feita uma comparação de vários middlewares, e de seguida uma análise detalhada dos desafios e das tecnologias necessárias para o desenvolvimento desta camada de serviço.

Esta investigação refere, como era de esperar, os desafios que advêm de desenvolver soluções IoT, nomeadamente a vasta heterogeneidade dos dispositivos IoT, ou por outras palavras, os dispositivos utilizam vários protocolos de comunicação diferentes e oferecem APIs distintas tornando o desenvolvimento nesta área mais complexo do que o necessário.

As arquiteturas existentes para middlewares IoT podem ser divididas nas três seguintes classes:

- ***Service Oriented Architectures*** - disponibiliza uma vasta gama de dispositivos e equipamentos IoT como serviços;
- ***Cloud-based*** - disponibiliza um serviço limitado em número e tipos de dispositivos, no entanto, coleciona e interpreta os dados dos dispositivos com facilidade pois os casos de uso estão bem definidos *apriori*;
- ***Actor-based*** - coloca ênfase na arquitetura "plug and play" de dispositivos IoT, distribuindo os dispositivos com atores reutilizáveis na rede.

Como é óbvio, um dos componentes mais importantes é a camada de "service discovery", que é responsável por recolher e configurar dispositivos presentes na rede onde o middleware está a operar. Esta camada de serviço deve também recuperar a conexão de dispositivos que eventualmente a possam perder. Isto tem bastante importância, uma vez que, é bastante incómodo ter que configurar manualmente todos os dispositivos IoT numa rede.

Por fim esta investigação reflete sobre a segurança e privacidade dos dados em middlewares IoT, referindo que é um dos mais importantes aspetos desta tecnologia. O artigo refere possíveis soluções e trabalhos da área que tentam resolver estes assuntos:

- ***Lightweight device authentication*** - para dispositivos low-power a autenticação pode ser uma operação moderadamente pesada em termos computacionais, o artigo refere

algumas pesquisas com alvo a resolver este problema. Isto aplica-se a qualquer tipo de solução para qualquer tipo de possível falha de segurança. Por vezes as soluções existentes são um pouco pesadas para estes dispositivos IoT com recursos escassos

- *Denial of service attacks* - estes ataques tem como objetivo esgotar os recursos computacionais, principalmente memória e CPU, da camada de middleware. O artigo refere alguns trabalhos na área com soluções para este objetivo
- *End-to-End security* - apesar de existirem protocolos de comunicação seguros na Internet, tais protocolos podem ser um pouco pesados e não indicados para a IoT. O trabalho refere algumas soluções de outros artigos.

Concluindo, os autores referem que a chave para desenvolver um sistema seguro é começar de raiz com a segurança e privacidade dos dados do sistema em mente, e não construindo a aplicação e introduzindo os mecanismos de segurança depois. Todo este artigo foi bastante útil, porque apresentou uma vasta gama de assuntos e problemas que os middlewares devem ter em conta, e portanto, será algo a rever no futuro.

Um problema da IoT é o elevado número de dispositivos concorrentes a reportarem dados e estatísticas aos serviços cloud, levando a que os protocolos de comunicação mais utilizados, como o HTTP por exemplo, se tornem desadequados a este ambiente de baixo consumo energético e de recursos computacionais limitados. P. Bellavista e A. Zanni [BZ16] tem como visão a utilização de protocolos orientados à IoT, neste caso o MQTT e o CoAP, para melhorar a escalabilidade e a eficiência das plataformas IoT cloud. Os investigadores também referem a emergência de soluções de edge computing, recorrendo a edge devices para “pré-processar” os dados emitidos pelos dispositivos IoT.

Ambos os protocolos foram desenvolvidos especialmente para aplicações na área da IoT, motivados pela necessidade de efetuar comunicação em larga escala utilizando o menor número de recursos possíveis, portanto são otimizados para as mensagens serem extremamente pequenas em tamanho, recorrendo a cabeçalhos otimizados para o efeito.

O MQTT, *Message Queue Telemetry Transport* é um protocolo de comunicação “publish-subscribe” baseado em TCP, onde cada mensagem possui um tópico, e vários clientes podem subscrever esse tópico para receber todas as mensagens publicadas utilizando esse tópico. Digamos que um tópico é semelhante a um URL. Os clientes enviam as mensagens para um “MQTT broker”, que irá distribuir depois as mensagens pelos “subscribers”. Uma funcionalidade bastante icónica do MQTT é o nível variável de “Quality of Service”, existindo 3 níveis diferentes, QoS 0, QoS 1 e QoS 2. Com cada nível de QoS a fiabilidade aumenta, mas a performance diminui.

O CoAP, *Constrained Application Protocol*, é um protocolo de comunicação baseado em UDP, muito semelhante ao HTTP, mas otimizado para ambientes de recursos computacio-

nais escassos e com a otimização da utilização da largura de banda de dados disponível. O protocolo foi desenvolvido com a arquitetura REST em mente, sendo portanto muito fácil de migrar sistemas desenvolvidos em HTTP para este protocolo. Este protocolo tem ainda suporte para a descoberta de serviços e recursos na rede, assim como mecanismos de "publish-subscribe", semelhantes ao MQTT. Leonard Richardson aborda a utilização de CoAP no desenvolvimento de *Restful Web APIs*, no seu livro [Ric13], demonstrando as capacidades deste protocolo.

Conhecer ambos estes protocolos é bastante importante, porque a utilização destes protocolos verifica uma adesão crescente, portanto o middleware a projetar durante esta tese deverá ser capaz de suportar todos estes protocolos emergentes.

No que toca à *home automation* em si, extensos estudos e trabalhos foram publicados, por exemplo, [DRD16] e [Azn+16] demonstram a utilização do Raspberry PI no desenvolvimento de sistemas de *home automation* acessíveis via navegador web e aplicação móvel. A segurança e os perigos ao meio ambiente também são aspetos importantes que não devem ser postos de parte, por exemplo, [Azn+16] tece várias considerações sobre os problemas que as baterias utilizadas nos dispositivos IoT causam ao meio ambiente. Frequências rádio são também analisadas e ultimamente determinadas como seguras, nomeadamente a frequência normal das redes WiFi.

Em grande parte do trabalho realizado nesta área, as funcionalidades são um ponto forte como é óbvio, mas a eficiência e poupança de energia são também fatores bastante importantes, sendo as várias formas de comunicação *wireless* escrutinadas para encontrar as mais eficientes e de menor custo, como se pode ver em [AR16].

4. O Problema e os seus Desafios

O problema consiste na diversidade e heterogeneidade das APIs e serviços IoT, levando a muitos entraves no desenvolvimento de aplicações nesta área, e, para além disso, uma vez que cada dispositivo possui a sua própria API proprietária, o desenvolvimento de novas aplicações torna-se moroso e mais complexo. As soluções existentes não são orientadas ao espaço casa especificamente, e as poucas que estão orientadas a este, são proprietárias e possuem restrições de sistemas operativos, não sendo totalmente abertas. As soluções open-source existentes normalmente requerem muita configuração e perícia tecnológica para a sua utilização e não são otimizadas para utilização em ambientes cloud, sendo de acesso exclusivo na rede local.

O objetivo deste projeto é portanto elaborar uma camada de serviço, middleware ou seja, que efetue a ligação entre os vários dispositivos do espaço casa e as aplicações clientes, abstraindo o espaço casa e os seus dispositivos para um acesso uniforme a estes recursos. Este componente deverá efetuar toda a gestão dos dispositivos das casas dos utilizadores, assim como outros serviços básicos de relevo para esta área, nomeadamente, auto configuração e descoberta de dispositivos na rede, monitorização do espaço casa e automação e programação dos dispositivos, de modo a que estes possam executar ações sem o input direto dos utilizadores. O sistema deverá suportar o registo de vários utilizadores e a adição das suas casas, assim como os dispositivos nelas presentes, ao sistema. O serviço deverá portanto ser acessível via Internet.

Outro objetivo, mais a nível de funcionalidades e benefícios dados ao utilizador, consiste na criação de cenários e tarefas automatizadas. Um cenário consiste num conjunto de estados pré definidos a ser aplicados aos dispositivos de um espaço casa, por exemplo, desligar todas as luzes de uma divisão e fechar uma porta. Assim, conjuntos de estados aplicados várias vezes pelo utilizador podem ser agrupados, poupando o mesmo de manualmente aplicar os estados um a um. As tarefas automatizadas consistem na alteração dos estados dos dispositivos sem necessitar a interação direta do utilizador. Estas tarefas podem ser aplicadas com base no tempo, por exemplo, fechar as portas às 21:00 todos os dias, ou através do estado de outro dispositivo, como por exemplo, ligar as luzes da garagem caso o sensor de movimento lá colocado registe movimento.

Uma proposta para o desenvolvimento é recorrer a *edge devices* para abstrair as implementações e interfaces dos dispositivos da rede local do utilizador, um princípio de *edge computing*. Estes *edge devices* efetuam a ligação entre os vários dispositivos do espaço casa e o serviço *cloud*, removendo a parte da descoberta de novos dispositivos e configuração dos mesmos, colocando-a nos sistemas locais. Para implementar este *edge device*, algo como o Raspberry PI poderá ser utilizado, ou qualquer tipo de placa de prototipagem disponível no mercado.

Idealmente, existirá um *edge device* por casa, que se deverá auto-configurar e começar a pesquisar novos dispositivos compatíveis, já existentes na rede. Os *edge devices*, também podendo ser apelidados de *home gateways*, deverão então gerir e abstrair todos os controlos dos dispositivos conectados a si, fornecendo uma API bem estruturada para controlo destes mesmos. O serviço *cloud* tem como função permitir ao utilizador aceder aos seus dispositivos via o *home gateway*, mesmo não estando na rede local. Um dos objetivos é possibilitar o controlo dos *edge devices* mesmo sem conectividade à Internet, desde que o utilizador esteja presente na mesma rede destes dispositivos. De notar, que para aceder aos *home gateways* e através disso, aos dispositivos do utilizador, é necessário efetuar autenticação, uma vez que cada *home gateway* estará associado a um utilizador àqueles que este der permissão.

4.1 Desafios

O desafio principal é o suporte às várias APIs dos dispositivos e a abstração das mesmas. Será necessário elaborar um mecanismo que torne dois dispositivos semelhantes, mas de fabricantes diferentes e com APIs distintas, efetivamente o mesmo para as aplicações clientes, ou seja, disponibilizar as APIs do mesmo tipo de dispositivo de uma forma unificada, independentemente da API do dispositivo. Por exemplo, dois tipos de termostatos, com funcionalidades semelhantes, mas de fabricantes diferentes e com APIs e protocolos diferentes, seja abstraído num recurso tipo termostato, que possui um conjunto de métodos bem definidos para serem invocados pelas aplicações clientes.

Vai ser preciso ter em conta a conectividade entre os dispositivos e o *middleware*, dado que maior parte dos consumidores possuem *routers* configurados e controlados pelas ISPs, operando em modo NAT e com uma *firewall* que bloqueia o acesso externo a recursos dentro da rede. Alguns dispositivos utilizam UPNP para efetuar *port forwarding*, para tornar estes dispositivos acessíveis de qualquer rede externa. No entanto, UPNP possui várias falhas a nível de segurança, e este processo é um pouco instável, não funcionando em certas condições (variável de acordo com o *router* e ISP). A solução em vista é *network tunneling*, que irá ser detalhada mais à frente nesta dissertação.

Outros desafios são a segurança de todo o *middleware*, que é um ponto importantíssimo uma vez que estamos a lidar com residências de seres humanos, e deixar os seus dispositivos inseguros é problema muito grave. A eficiência e a escalabilidade do serviço são fatores a ter em conta, é necessário assegurar que o serviço tenha a possibilidade de crescer horizontalmente com o aumento de número de utilizadores. Além de tudo isto, será também importante manter boas práticas de desenvolvimento e arquiteturais, uma vez que um serviço deste género pode crescer bastante no que toca ao tamanho e complexidade do código, de maneira a que a manutenção do código seja mais fácil.

Por fim, o último desafio será a simulação e a demonstração da interação com os dispositivos. Devido a problemas óbvios de financiamento e de recursos, não será possível adquirir muitos dispositivos existentes no mercado, portanto, uma das soluções será tentar encontrar emuladores ou implementações *stub* dos dispositivos reais, como por exemplo, este [Ste16] emulador do Philips HUE. Caso algum tipo de dispositivos não possuam emuladores, esses mesmo serão implementados por conta própria.

4.2 Funcionalidades

Definido o problema e encontrados os desafios principais, irão agora ser delineadas as funcionalidades principais que deverão ser encontradas no *middleware* e por ventura na aplicação de demonstração.

- Gestão de utilizadores básica, incluindo registos e autenticação
- Gestão e manipulação de dispositivos IoT
- Agrupamento de dispositivos em localizações (espaço casa)
- Abstração dos detalhes de implementação de cada dispositivo. O modo de operação deverá ser independente do tipo de dispositivo utilizado
- Configuração de novos dispositivos deverá ser assistida e automatizada, através dum *scanner* de dispositivos compatíveis
- Gestão e configuração de cenários
- Gestão e configuração de tarefas automatizadas, à base de expressões temporais ou com base no estado de outros dispositivos
- Garantir acesso aos dispositivos através da Internet, removendo a necessidade da presença na rede local onde os dispositivos estão instalados

5. Metodologia

Grande parte desta dissertação passa então pela concepção e implementação de um todo sistema de software, e devido a tal, serão utilizadas práticas de desenvolvimento comprovadas na indústria e que se adequem à natureza do projeto. Neste capítulo iremos então detalhar as abordagens que irão ser adotadas, assim como as suas vantagens relativas.

5.1 Iterative and Incremental Development

Esta prática consiste numa combinação de práticas incrementais e iterativas, onde essencialmente um produto de software é separado em componentes, sendo que cada um destes componentes é iterativamente implementado. Essencialmente, esta metodologia de desenvolvimento irá ter os seguintes passos, para a elaboração de cada componente:

- Requisitos - definição dos requisitos do sistema
- Design/Arquitetura - desenho da arquitetura do sistema
- Implementação - implementação dos elementos definidos na arquitetura
- Testes - testes unitários e de integração
- Avaliação - avaliação do sistema dum ponto de vista funcional e técnico

Durante a idealização da arquitetura, o sistema irá ser dividido em pequenos componentes ou sistemas que resultam no projeto final, sendo que cada um sistema irá ser implementado um de cada vez, seguindo esta metodologia. Na essência, esta metodologia é uma combinação da metodologia *Waterfall* e incremental, desenvolvendo pequenas partes do sistema de cada vez.

5.2 Model Driven Development

Ideal para sistemas de grande complexidade, durante o desenvolvimento dos componentes de software deste projeto irão ser utilizados diagramas para descrever o comportamento do sistema num alto nível. Utilizando modelos de domínio e diagramas de classe, entre outros, será possível descrever a um alto nível de abstração, a estrutura e o comportamento do sistema de software antes da sua implementação.

Esta metodologia tem grandes vantagens, sendo uma delas a facilitação do trabalho em equipa e a colaboração com outros, uma vez que é mais fácil explicar conceitos através de um diagrama em vez de um excerto de código. Além disso, dado que os diagramas não estão necessariamente ligados a uma dada tecnologia ou linguagem de programação, teremos uma grande vantagem na altura de migrar o sistema para outra tecnologia, caso seja necessário. Além disso, existem aplicações no mercado, capazes de transformar modelos em código, reduzindo o tempo de implementação necessário.

Neste preciso caso, será utilizado UML para desenhar os diagramas, uma linguagem de modelação bastante utilizada na indústria, possuindo várias ferramentas capazes de transformar diagramas em código.

5.3 Static Code Analysis

Sistemas de software de grande dimensão devem seguir os mesmos standards sintáticos e arquiteturais em toda o código. Para efetuar esta verificação, foram utilizadas ferramentas de análise de código, que permitem detetar *code smells* e más práticas arquiteturais para prevenir *bugs* e falhas de segurança. Também ajudam a manter o estilo do código uniforme, para melhorar a sua legibilidade e compreensibilidade.

5.4 Continuous Integration

Todos os componentes do sistema irão ser mantidos num sistema de controlo de versões, nomeadamente o *git*, utilizando o serviço de *hosting* do *GitHub*. Utilizando este serviço, é possível efetuar várias integrações com outras ferramentas de *testing* e *deployment* que nos permitem seguir esta metodologia de *continuous integration*, onde se testa e coloca-se uma

aplicação pronta para ambientes de produção continuamente, sem a interação manual dos programadores.

Utilizando estes recursos é possível testar a última versão do código assim que ele é *pushed* para o repositório remoto. Basicamente, assim que um novo *commit* é *pushed* para o *GitHub*, o *Travis* corre os testes dessa nova versão do software, reportando o resultado para o *GitHub*. Além dos testes, também são executadas nesta fase as ferramentas de análise de código, retornando uma *build* errônea caso elas detetem problemas.

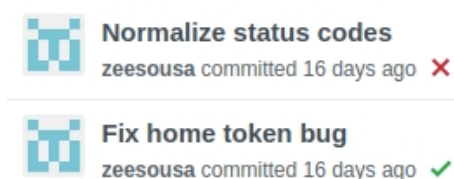


Figura 7: Integração Travis & GitHub

Neste exemplo, o *commit* "Fix home token bug" passou os testes definidos enquanto que o último *commit* "Normalize status codes" falhou os testes.

Quanto a metodologia dos testes não foi seguida nenhuma em específico, apenas foi feito um esforço para se ter o maior número de linhas de código e funcionalidades cobertas pelos mesmos.

Em todos os repositórios é possível obter um *coverage report*, que relata a cobertura dos testes sobre as aplicações. Para isto recorreu-se ao serviço *codecov*, que gera estes relatórios automaticamente a partir do *Travis*.



Figura 8: Integração Travis & GitHub

Acima é possível ver dois crachás gerados pelo *codecov* e pelo *Travis* respectivamente, que denotam a cobertura do código no que toca a testes, assim como o estado dos testes feitos à última *build*.

Os componentes do sistema que necessitarem de um sistema de *hosting* irão recorrer ao *Heroku*, uma *platform as a service*, que permite correr qualquer tipo de serviço web sem custos. A única limitação consiste no adormecimento dos serviços, ou seja, todos os serviços grátis "adormecem" ao fim de 30 minutos sem receberem qualquer tráfego.

O *Heroku* é semelhante ao *Travis* na medida em que sempre que deteta um novo *commit* no *GitHub* tenta efetuar o *deploy* dessa nova versão. Neste caso, como também temos o *Travis* no

nosso *workflow*, o *Heroku* aguarda que os testes passem antes de dar *deploy* ao código. Assim, asseguramos que temos sempre uma versão estável do sistema ativa.

É possível definir *pipelines* avançadas no *Heroku*, podendo ter sistemas de *staging* e sistemas de *production* ao mesmo tempo. Por exemplo, um sistema de *staging* pode ter passado nos testes mas ainda está na fase de avaliação, enquanto que um sistema de *production* já está pronto para utilização final. Além disso, também é possível ter uma aplicação por cada *feature branch* presente no *GitHub*, criando uma aplicação para cada *pull request* ativo. Neste caso, temos apenas uma aplicação de *staging* e uma para cada *pull request* ativo.

6. Arquitetura

Delineado os objetivos, vamos proceder para a elaboração da arquitetura do serviço, falando dos componentes num alto nível de abstração antes de mergulhar nos detalhes tecnológicos e de implementação. Aqui vamos detalhar com mais rigor as funcionalidades base do sistema assim como as entidades que o compõem.

6.1 Sistema

O *middleware*, devido ao seu tamanho e à sua complexidade, não será composto apenas por um único elemento de software, sendo composto por vários elementos mais concisos, cada um resolvendo um conjunto de problemas. Neste caso iremos dividi-lo em duas partes, *hub* e *api*, dois componentes de software que fornecem APIs RESTful, uma para uso interno e outra para uso externo respetivamente, ambas seguindo as recomendações de Leonard Richardson no seu livro sobre serviços web *RESTful* [Ric13]. Além destes dois elementos que compõem o *middleware*, também serão desenvolvidas as simulações dos dispositivos, pequenas aplicações que irão simular o funcionamento dos diversos tipos de dispositivos, e por fim, uma aplicação cliente que faça uso de todas as funcionalidades oferecidas por esta plataforma.

6.1.1 Hub

O *hub* consiste no conceito *edge device* que já foi abordado anteriormente, um dispositivo que fornece uma interface para interagir com os dispositivos do utilizador. Este componente deverá fornecer camadas de compatibilidade para diversos tipos de dispositivos, de modo a resolver o problema relativo à heterogeneidade dos dispositivos IoT.

Este componente será pré-instalado e configurado em dispositivos que deverão depois ser fornecidos aos utilizadores (num caso de utilização em massa). Neste caso de demons-

tração académica, será utilizado um Raspberry PI para simular este dispositivo. O serviço irá correr como um *daemon* nestes dispositivos, sendo iniciado durante o processo de *boot*.

Este componente deverá fornecer uma API HTTP RESTful, que permita interagir com os dispositivos no espaço casa, de acesso exclusivo à *api*. O *hub* deverá ter métodos *GET* e *PUT*, para manipular os dispositivos. Este métodos deverão ser acompanhados de parâmetros que identificam um dispositivo, como por exemplo, endereço IP, porta de acesso, e outros parâmetros exclusivos a um dispositivo. Estes parâmetros deverão ser obtidos em parte pelo sistema de descoberta de dispositivos presente neste componente, que procura no espaço casa dispositivos compatíveis, retornando os seus parâmetros de acesso.

O *hub* é portanto *stateless*, ou seja, não armazena nenhum tipo de informação ao longo do seu funcionamento, essa função está ao encargo da *api*, que vamos detalhar já de seguida. Além disso, este componente irá ser exposto através de um software de *tunneling*, que como já foi visto anteriormente, é uma melhor opção do que UPNP e *port-forwarding* para a exposição de recursos em redes locais.

6.1.2 Api

A *api* contém toda a lógica relacionada com utilizadores, casas, dispositivos e a manipulação de cenários e tarefas. Este componente é o típico servidor aplicacional com acesso a base de dados que contém toda a lógica de negócio do sistema. A *api*, tal como o *hub*, deverá fornecer uma API HTTP, seguindo também uma arquitetura *REST*. Esta API será de acesso público, contrastando com a API do *hub* que é de acesso exclusivo ao *middleware*.

Dado que o *hub* é um componente *stateless*, a *api* deverá armazenar os parâmetros de acesso aos dispositivos. Além disso, todas as funcionalidades do *hub* devem ser acessíveis a partir da API, efetuando reencaminhamento das aplicações clientes para o *hub*.

Tudo isto irá consistir num sistema de tamanho médio composto essencialmente de operações *CRUD*, tornando-se mais complexo na área das tarefas automatizadas, onde provavelmente se deverá investir em alguma *framework* de *background jobs* para monitorizar e controlar dispositivos assincronamente, sem criar carga adicional no servidor aplicacional.

Obviamente, os recursos irão pertencer a uma entidade "utilizador", sendo necessário algum tipo de autenticação para aceder aos mesmos, que em princípio será *token-based*.

6.1.3 Simuladores de Dispositivos

Estes simuladores têm como função emular o funcionamento de dispositivos do mundo real. Isto permite proceder a uma demonstração completa sem ter que adquirir vários tipos de dispositivos diferentes. Neste preciso caso os dispositivos deverão fornecer APIs HTTP Restful, um standard bastante utilizado na indústria. Para demonstrar a compatibilidade de vários protocolos, também serão desenvolvidos alguns exemplos em CoAP.

6.1.4 *Wrapper* da API

É boa prática que após o desenho e conceção de uma API à base de HTTP e JSON se desenvolvam *wrappers* para as linguagens de programação mais comuns. Um *wrapper* essencialmente converte as chamadas à API utilizando HTTP para métodos ou funções da linguagem respetiva.

Por exemplo, temos um recurso *foo* disponível numa API HTTP *Restful*, para efetuar a sua obtenção deveria ser feito algo como:

```
HTTP GET -> /foo/1
```

Um *wrapper* converteria estas comunicações HTTP para métodos de uma linguagem de programação. Um *wrapper* em Java provavelmente atuaria da seguinte maneira:

```
FooApiWrapper api = new FooApiWrapper(remoteHost);  
  
Foo foo = api.getFoo(1);
```

Internamente o *wrapper* faria a chamada à API remota e devolveria o resultado respetivo, um objeto em vez de uma resposta HTTP, fazendo toda a serialização automaticamente. Isto permite retirar uma grande porção de código que se iria repetir em todas as aplicações clientes. Neste caso, irá ser desenvolvido um *wrapper* para a *api* do *middleware* na linguagem que for utilizada na aplicação cliente.

6.1.5 Aplicação Cliente

Como prova de conceito irá ser desenvolvida uma aplicação *mobile*, em linguagem e sistema operativo mais à frente. A aplicação deverá fazer uma demonstração de todas as funcio-

nalidades e operações que o *middleware* implementa. Estas funcionalidades estão detalhadas na [secção 2](#) do capítulo 4. Todos esses pontos serão aplicados ao resultado final da aplicação.

De notar que esta aplicação apenas foi desenvolvida para efeitos de demonstração, portanto, grande parte desta dissertação será, como é óbvio, sobre o *middleware* e os seus componentes. Foi efetuado um esforço significativo para implementar boas práticas arquiteturais e de programação em todo o software desenvolvido, mas a aplicação cliente poderá ficar um pouco aquém dos restantes elementos.

6.1.6 Visão Geral

Um espaço casa deverá possuir um *hub* instalado, e a *api* comunica com os dispositivos do espaço através deste equipamento. Com isto separamos uma componente complexa da *api*, a camada de compatibilidade entre os diversos modelos de dispositivos, tornando todo o sistema mais manutenível.

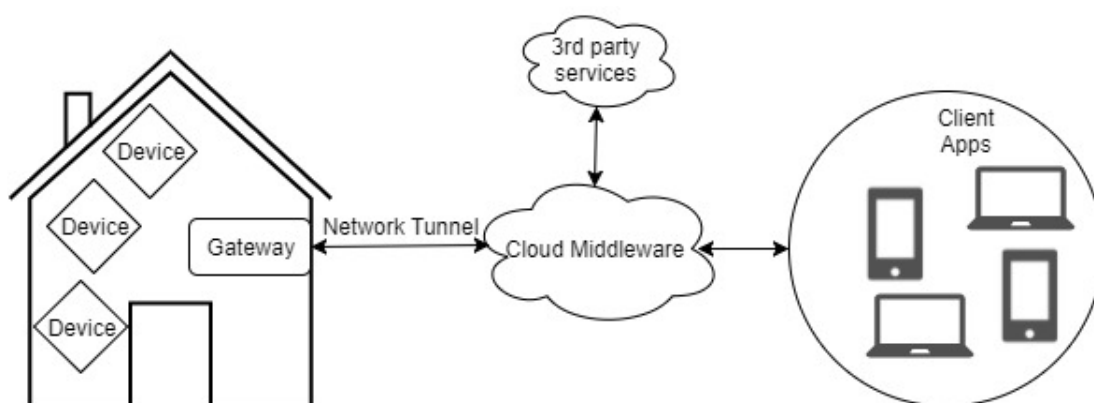


Figura 9: Visão Geral dos Sistemas

Aqui o acesso aos dispositivos será abstraído graças ao *software* que irá operar no *gateway/hub*. Apenas a *api* irá ter acesso ao *hub*, sendo que irá ser desenvolvido algum mecanismo de segurança de maneira a que se rejeite o acesso indevido ao *hub* dos utilizadores. De notar que apesar deste mecanismo de segurança proteger o *hub*, os dispositivos deverão ter as suas próprias medidas de segurança, garantidas pelos seus fabricantes.

As aplicações clientes e serviços de terceiros poderão aceder ao *middleware* recorrendo à API HTTP ou utilizando os *wrappers* desenvolvidos.

6.2 Análise de Domínio

Nesta fase identificam-se as classes conceptuais essenciais para resolver o problema em questão e implementar as funcionalidades desejadas. Utilizando esta abordagem, apenas se identificam conceitos do mundo real, ou seja, segundo o paradigma dos utilizadores, num alto nível de abstração. Este domínio aplica-se mais ao componente *api*, que, essencialmente, representa o *core* da funcionalidade do *middleware*.

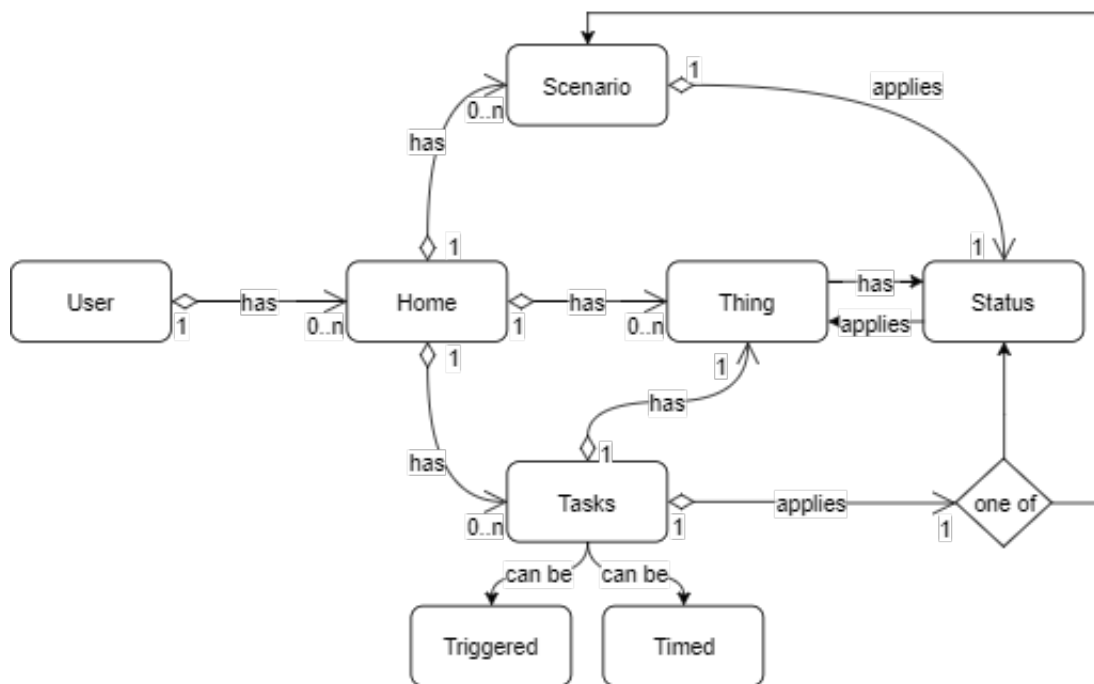


Figura 10: Modelo de Domínio

Resumidamente, o sistema possui vários utilizadores, sendo que cada um deles possui, portanto, uma ou mais casas e cada casa possui por sua vez um conjunto de dispositivos. Além disso, cada casa possui também os seus cenários e tarefas, podendo estas últimas serem ativadas com base em expressões temporais ou condicionantes baseadas noutros dispositivos do sistema. De seguida será apresentado um pequeno dicionário com o significado e função de cada entidade:

User

Representa o utilizador, que, como é normal, possui um email e uma password para autenticação. O utilizador é a entidade base do sistema, ou seja, qualquer outra entidade está diretamente ou indiretamente ligada a um utilizador.

Os dados do utilizador também serão utilizados como base para a autenticação e autorização. Ou seja, um utilizador não podem interagir com as entidades de outro utilizador.

Home

Representa a casa de um utilizador, contém dados para aceder à rede local e interage com os dispositivos lá instalados. Além disso, esta entidade também deverá possuir uma referência ou um identificador do *hub* instalado no espaço casa, algo que permite ao *middleware* saber onde e como comunicar com os dispositivos desse espaço.

Todas as restantes entidades possuem uma associação direta ou indireta a um espaço casa.

Thing

Representa os dispositivos, contendo os dados necessários para efetuar comunicação com o mesmo. O *hub* aceita uma lista de parâmetros para efetuar comunicação com um dispositivo compatível, mas como este é *stateless*, não tem capacidade para os armazenar. Essa função pertence à *api*, e é com recurso a esta entidade que se irá alcançar o objetivo pretendido.

Cada *thing* possui dois parâmetros obrigatórios: tipo e subtipo. O tipo representa a funcionalidade geral do dispositivo, por exemplo, uma luz, uma fechadura ou um termóstato. O subtipo é referente à implementação do dispositivo, ou por outras palavras, o modelo do dispositivo. Neste caso a divisão entre tipos:

- Things::Light - lâmpadas, dispositivos de iluminação
- Things::Lock - fechaduras, portas, portões
- Things::Thermostat - termostatos
- Things::Weather - estações meteorológicas, serviços de meteorologia
- Things::MotionSensor - sensores de movimento, alarmes

Quanto aos subtipos, irão existir os seguintes:

- Light - rest, coap, hue
- Lock - rest
- Thermostat - rest
- Weather - rest, owm
- MotionSensor - rest

Os subtipos *rest* correspondem às simulações de dispositivos desenvolvidas no âmbito deste projeto. O *coap* é também uma simulação, só que utiliza CoAP em vez de HTTP. O subtipo *hue* corresponde às lâmpadas do *Philips Hue* enquanto que o *owm* corresponde ao serviço remoto de meteorologia *OpenWeatherMap*.

Status

Representa o estado de um dispositivo. Os dispositivos retornam estas entidades mas estas também podem ser aplicadas ao dispositivo, efetivamente alterando o seu estado.

Isto permite-nos utilizar a arquitetura REST para alterar o estado dum dispositivo com base nos verbos HTTP, *PUT* e *GET*, onde o primeiro altera o estado e o segundo obtém o estado de um dispositivo.

Scenario

Representa um conjunto de estados que podem ser aplicados em conjunto. Tal como já foi referido anteriormente, podemos juntar estados que normalmente são aplicados ao mesmo tempo, como, desligar as luzes todas antes de o utilizador ir dormir, num só cenário que pode ser aplicado, simplificando muito a utilização da aplicação.

Task

Representam as tarefas, entidades que representam uma aplicação dum estado ou de um cenário com base numa condição (triggered ou timed). Isto permite definir as interações já referidas na introdução do problema, como, aplicar o cenário "desligar luzes" às 23:00, ou, ligar a luz da garagem quando o sensor de movimento lá colocado detetar movimento.

Como foi possível reparar nos exemplos referidos, temos dois tipos de tarefas, que serão agora apresentadas.

Triggered

Representam as tarefas condicionais, que são aplicadas com base no estado de um outro dispositivo presente neste sistema, como no exemplo acima referido, ligar a luz da garagem quando o sensor de movimento lá colocado detetar movimento.

Timed

Representam tarefas que são aplicadas com base numa dada expressão temporal. A tarefa "aplicar o cenário desligar luzes às 23:00" é deste tipo.

6.3 Design

De todos os componentes já referidos, apenas o *hub* e a *api* tiveram direito a um grande esforço no que toca ao desenho e elaboração da sua arquitetura utilizando diagramas, como dita o *model driven development*. Os outros componentes como os simuladores de dispositivos são bastante simples e concisos, sendo que não fazem parte do *scope* da dissertação, sendo apenas ferramentas para demonstração. O mesmo se aplica à aplicação cliente e a componentes relacionados à demonstração, que apenas são um *front-end* para demonstração do *middleware*.

6.3.1 Hub

Como já foi visto anteriormente, o *hub* tem como função agregar as APIs dos vários dispositivos de um espaço casa, numa API HTTP RESTful a ser consumida pelo *core* do *middleware*, a *api*. No geral, foi concebida uma arquitetura seguindo o padrão MVC. Neste caso, o *model* são os dispositivos, as *views* os estado dos mesmos, enquanto que os *controllers* aceitam os mais diversos parâmetros para estabelecer a comunicação com um dispositivo.

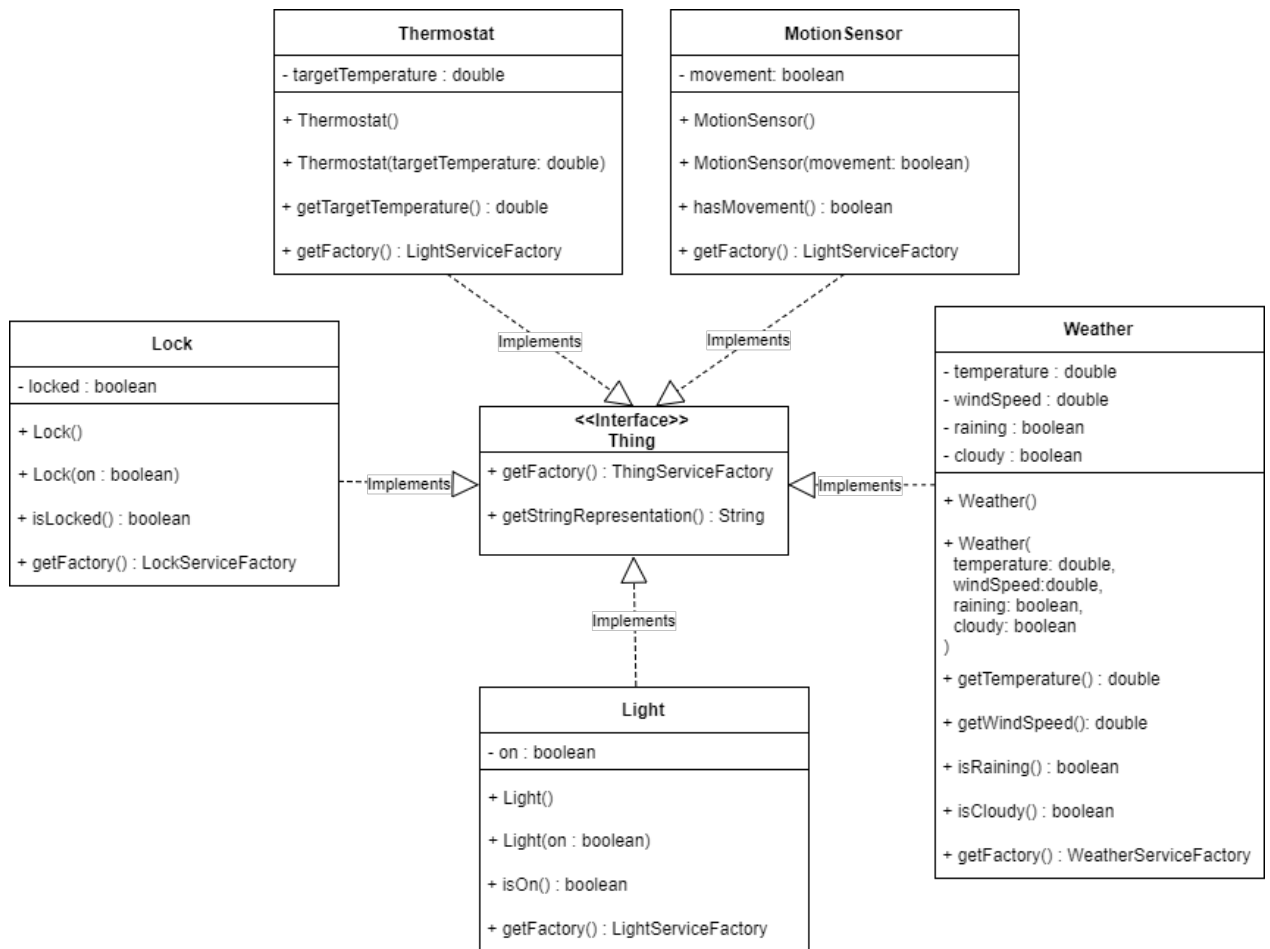
Things

Inicialmente foram definidas classes base para os diversos tipos de *things* suportadas. Estas classes representam o estado de um dispositivo, indicando, por exemplo, se uma porta está fechada ou se uma luz está ligada. Serão suportadas os 5 tipos de *things* já definidas no modelo de domínio: termóstato, sensor de movimento, fechadura, luz, estação de me-

teorologia. As *things* são classes imutáveis, ou seja, o seu conteúdo não pode ser alterado dinamicamente, servindo apenas para representar um estado.

O processo de escolha nos atributos que compõem o estado dos dispositivos consiste na análise do funcionamento comum entre os diversos dispositivos no mercado, ou seja, apenas foi decidido colocar atributos que estão presentes em grande maioria dos dispositivos. Por exemplo, certas lâmpadas *premium* oferecem regulação de luminosidade e cor, mas grande parte das lâmpadas no mercado não oferecem essa funcionalidade. Uma solução para isto poderia ser incluir os campos extra na mesma, não apresentado o seu valor caso não existissem, mas para o âmbito desta dissertação decidiu-se manter o estado dos dispositivos simples, para efeitos de demonstração.

As *things* não possuem muitas funcionalidades em comum, mas foram mesmo assim integradas num interface que possui dois métodos de utilidade. A partir de uma *thing* deveremos obter a sua representação textual, por exemplo, a classe *Lock* tem a representação textual *lock*. Também deveremos obter a *factory* que cria os serviços (definição dos mesmos mais à frente) que interagem com os dispositivos respetivos.

Figura 11: Diagrama de classes: *Things*

Serviços

Posto isto, teremos agora que idealizar a comunicação com os dispositivos. Neste caso faremos o uso de serviços, classes que possuem métodos para interagir com um dispositivo específico. Os serviços podem partilhar funcionalidade, como métodos para comunicação HTTP, e portanto esse tipo de funcionalidades estão encapsuladas em classes abstratas auxiliares. Neste caso, além do serviço base, temos também o serviço HTTP e CoAP, que possuem métodos auxiliares para comunicar com dispositivos que utilizem os protocolos respetivos.

Para cada subtipo de uma *thing* deve existir um serviço. Como já vimos anteriormente, um subtipo representa um modelo específico do dispositivo, e como cada subtipo exige um tipo de comunicação e interação diferente, é portanto necessário ter um serviço para cada um. No diagrama abaixo vemos os serviços para os subtipos da *thing* do tipo *light*, *coap* e *rest*.

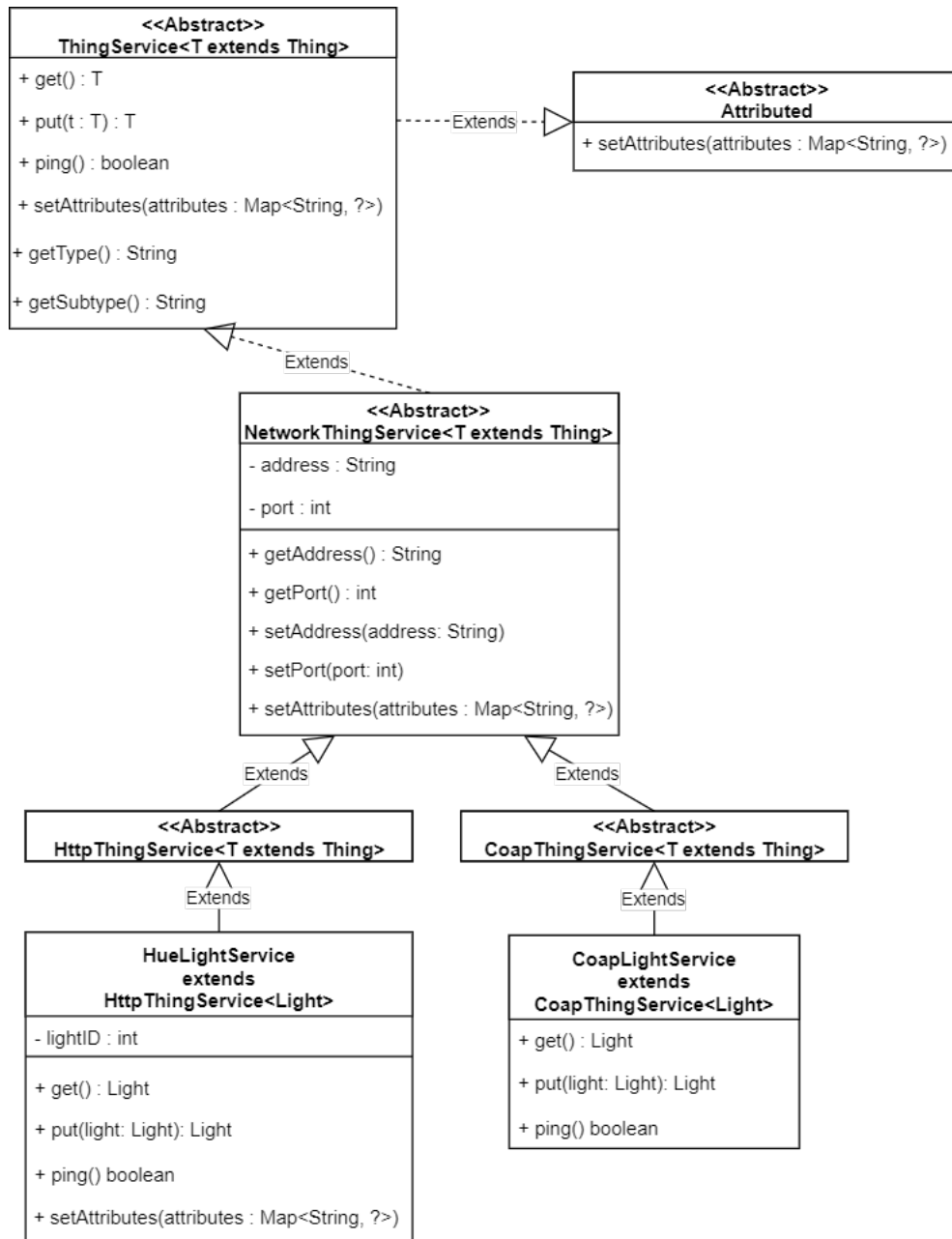


Figura 12: Diagrama de classes: Serviços

Os restantes serviços serão de seguida enumerados, indicando as classes abstratas em que se baseiam:

- *rest light* -> *RestLightService* -> *HttpThingService*<Light>
- *rest lock* -> *RestLockService* -> *HttpThingService*<Lock>
- *rest thermostat* -> *RestThermostatService* -> *HttpThingService*<Thermostat>
- *rest weather* -> *RestWeatherService* -> *HttpThingService*<Weather>
- *owm weather* -> *OWMWeatherService* -> *ThingService*<Weather>
- *rest motionsensor* -> *RestMotionSensor* -> *HttpThingService*<MotionSensor>

Os serviços possuem todos os parâmetros para efetuar comunicação com um dispositivo, nomeadamente o seu endereço, porta de acesso, entre outros. Estes parâmetros são fornecidos ao *hub* nas chamadas à sua API e por ventura são passados aos serviços. O método *setAttributes*, fornecido pela classe abstrata *Attributed*, aceita um *array* associativo, ou por outras palavras, um *map*, que contém todos os atributos necessários para o seu funcionamento. Caso faltem atributos ou estes tenham formatos inválidos, deverá ser lançado um erro, uma exceção, ou um qualquer mecanismo da linguagem para assinalar *inputs* inválidos. Apesar dos *setters* e dos *getters* presentes nos serviços, este método permite definir os atributos tendo apenas uma instância de *ThingService*, sem saber a classe concreta da instância.

As instâncias concretas dos serviços não são de acesso público, apenas o interface base e as extensões abstratas. Decidiu-se proceder desta maneira para reduzir o acoplamento entre os serviços e as outras partes da aplicação, nomeadamente os *controllers* que recebem parâmetros das chamadas à API e que comunicam com os dispositivos, utilizando os serviços para isso. Assim sendo, todos os serviços serão instanciados através de *factories*, que recebem um subtipo e devolvem uma instância dum serviço referente ao subtipo dado. Mais uma vez, é por esta razão que definiu-se o método *setAttributes*, para efetuar o *assign* dos atributos sem ter acesso à classe específica de uma instância de um serviço.

A classe auxiliar *Attributed*, em termos lógicos, basicamente assinala isto que já foi exposto, uma classe que permite mudar o seu estado interno, ou por outras palavras, os seus atributos, através de um *map* entre o nome do atributo e o seu valor. Este comportamento foi extraído do serviço porque é utilizado noutros pontos da aplicação, nomeadamente a descoberta de serviços, que utiliza este mecanismo pelas mesmas razões supracitadas.

Os dispositivos do tipo base *NetworkThingService* requerem todos um endereço e uma porta, que poderão ser passados à instância de um serviço através dos seus *setters*, no entanto, não podemos fazer tal coisa, porque apenas interagimos com instâncias do tipo *ThingService*, devido ao acesso privado das classes concretas. Para resolver isso utilizamos o método *se-*

tAttributes, como já se falou anteriormente. Classes que recorrem a parâmetros adicionais, como o *HueLightService*, que faz uso do novo parâmetro *lightID*, devem voltar a implementar o método *setAttributes* para acomodar o novo parâmetro.

Factories

Devido à abordagem à base de “um serviço para um subtipo”, iremos ter várias classes concretas para cada um dos subtipos de dispositivos, portanto, irá ser utilizado o padrão *abstract factory*, onde teremos várias fábricas de serviços, uma por cada tipo de dispositivo, para simplificar o processo de instanciação de serviços. Por exemplo, a fábrica do tipo “luz” é responsável por instanciar todos os serviços suportados por este tipo de dispositivo, aceitando um subtipo como parâmetro. As fábricas de objetos também devem ter um método de utilidade para ver se um dado subtipo é suportado. Além disso, o método *create* deve, obviamente, finalizar com um erro quando o subtipo não é suportado.

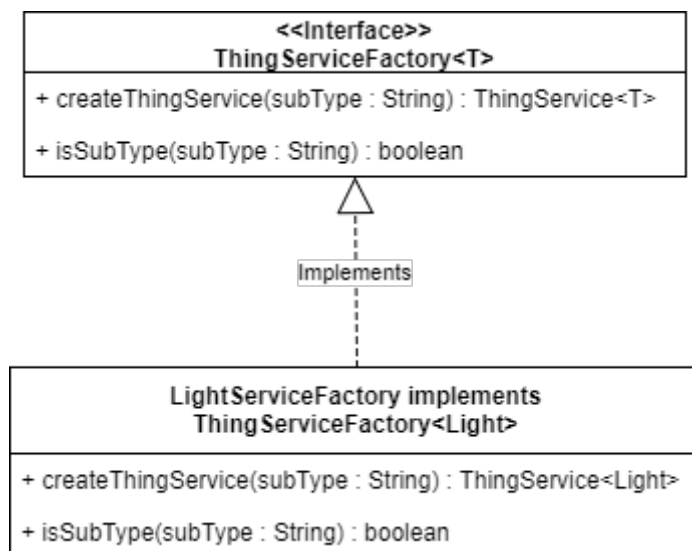


Figura 13: Diagrama de classes: *Factories*

Neste diagrama apenas está representada a *LightServiceFactory*, mas todos os outros tipos de dispositivos tem fábricas com assinaturas semelhantes. O método *create* retorna os serviços definidos acima, mas sem exportar os detalhes da classe concreta. Caso o método *create* seja chamado com o subtipo *hue* numa fábrica do tipo *LightServiceFactory* a fábrica retorna um *HueLightService* abstraído no interface *ThingService<Light>*.

O interface base *ThingServiceFactory* irá ter uma função vital nos *controllers*, permitindo efetuar *dependency injection* nestes, simplificando a operação dos mesmos. Assim, um *con-*

troller genérico consegue operar com qualquer tipo de dispositivo, baseado na *factory* que o compõe.

Controllers

Seguindo a arquitetura MVC, já temos os componentes que representam o *model*, neste caso as *things* e os serviços, restando os *controllers* e as *views*. Os *controllers* têm como função aceitar parâmetros provenientes da chamada à API, gerando alterações nos *models*, ou mais concretamente, nos dispositivos. Além de gerar alterações, os *controllers* também podem gerar as tais *views*, representações dos estados dos dispositivos.

Tal como foi explicado no *design* das *factories*, os *controllers* irão funcionar à base de *dependency injection*, sendo compostos por uma *ThingServiceFactory*, que irá ditar o tipo de dispositivos com que o *controller* lida. Assim, tendo apenas uma definição base de um *controller*, conseguimos trabalhar com todo o tipo de dispositivos, em vez de criar um *controller* para cada tipo individual. Todo o objetivo de lidar com os genéricos, que consistem nestas classes parametrizadas com um tipo genérico *T*, é atingir esta abstração, onde um *controller* consegue lidar com vários tipos e subtipos de dispositivos, abstraindo todo este processo de seleção de serviços e passagem de parâmetros.

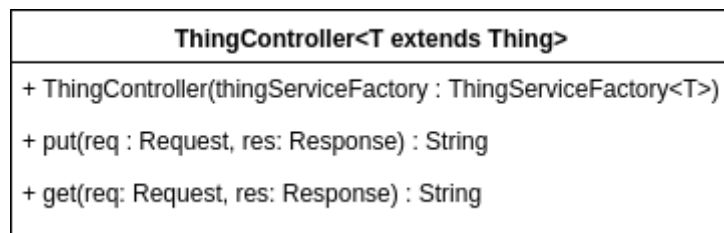


Figura 14: Diagrama de classes: *Controllers*

O *ThingController* é instanciado com uma *ThingServiceFactory*, e possui dois métodos, *get* e *put*, que correspondem aos verbos HTTP GET e PUT, seguindo assim uma metodologia *restful*, onde temos os recursos (*things*) e utilizamos verbos HTTP para interagir com os recursos. O método *put* tem como objetivo alterar o estado de um dispositivo dado um *request* e uma *response*, objetos que representam um pedido e uma resposta HTTP, normalmente de uma *framework* de desenvolvimento de servidores web. O método *get* serve apenas para retornar o estado atual de um dispositivo. Ambos estes métodos retornam uma *string*, que representa a resposta da API em JSON a um qualquer pedido.

Internamente, o *controller* recebe um pedido, retira o parâmetro subtipo dos parâmetros deste, cria um serviço através da *factory*, utiliza o método *setAttributes* atribuindo os restantes parâmetros, e depois chama o método correspondente no serviço, ou *get* ou *put*.

Descoberta de Dispositivos

Posto isto, temos os serviços que permitem estabelecer comunicação bidirecional com os dispositivos, faltando agora resolver o problema da descoberta de dispositivos na rede local. Uma parte muito importante de todo o paradigma da IoT e do espaço casa, uma vez que aumenta muito a facilidade de uso destes sistemas para o utilizador comum.

Inicialmente foram concebidas as classes para a descoberta de dispositivos, tendo como base o padrão *strategy*, que define estratégias diferentes para resolver um dado problema. Isto traduz-se para uma classe abstrata base com um método *perform*, que essencialmente leva a cabo a descoberta de dispositivos, utilizando uma fábrica de serviços e um subtipo como argumentos (passados através de um construtor).

As estratégias essencialmente dependem do subtipo de dispositivo utilizado. Dispositivos diferentes têm estratégias diferentes de descoberta, alguns podem utilizar UPNP, outros não, sendo necessário encontrar outras alternativas. Utilizando esta abordagem, é alcançável a elaboração de estratégias para cada um dos subtipos, mantendo esse processo transparente noutras camadas da aplicação, nomeadamente os *controllers*.

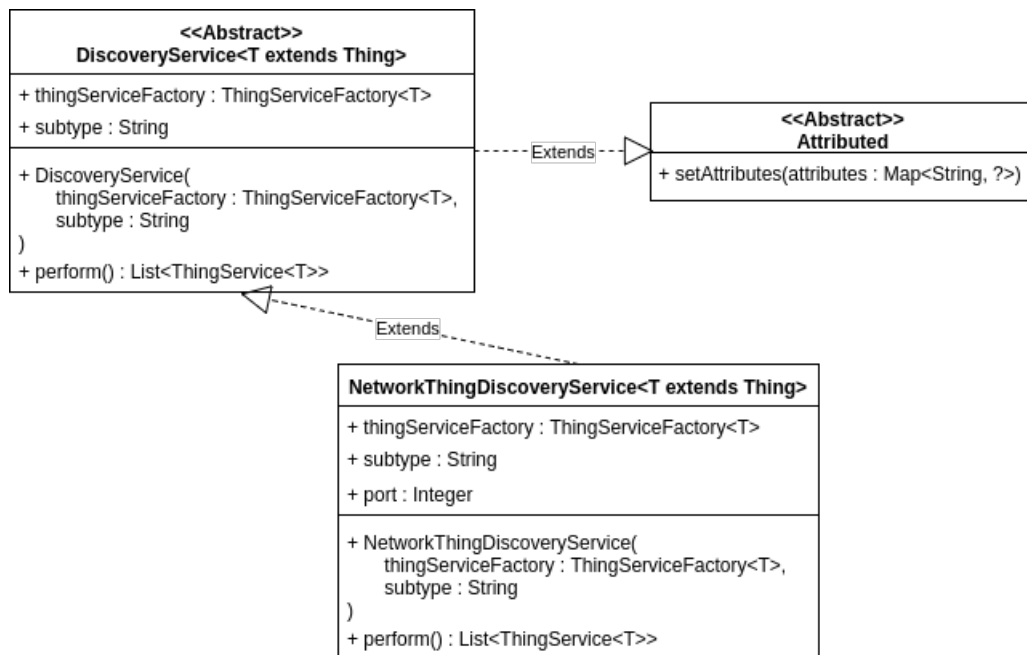


Figura 15: Diagrama de classes: *Serviços de descoberta de dispositivos*

Neste caso apenas possuímos um único serviço de descoberta, *NetworkThingDiscoveryService*, que descobre todo o tipo de dispositivos baseados no *NetworkThingService*, ou seja, dispositivos que essencialmente atuam utilizando o protocolo IP, ou protocolos que se baseiam neste anterior, como HTTP ou CoAP, estando presentes na rede local do *hub*. O

NetworkThingDiscoveryService, além do subtipo, que é necessário para todo o tipo de descobertas, também necessita de uma porta. Caso uma porta não seja passada como parâmetro será utilizada a porta "default" do protocolo utilizado (HTTP utiliza a porta 80 por exemplo).

Todos os subtipos são cobertos por esta estratégia de descoberta, exceto um, o subtipo *own*, que como se baseia num serviço remoto, não necessita de mecanismos de descoberta.

Concebida a arquitetura dos serviços, é preciso unir tudo e criar um *controller* para expor esta funcionalidade, mas antes disso, ainda falta resolver de alguns detalhes. Primeiro, está novamente presente o problema encontrado nos *thing services*, onde temos vários objetos distintos, tornando a sua instanciação um processo complexo. Para resolver isto, basta adicionar um outro método às *factories*, que devolve uma implementação de um *DiscoveryService* compatível com o *subtipo*.

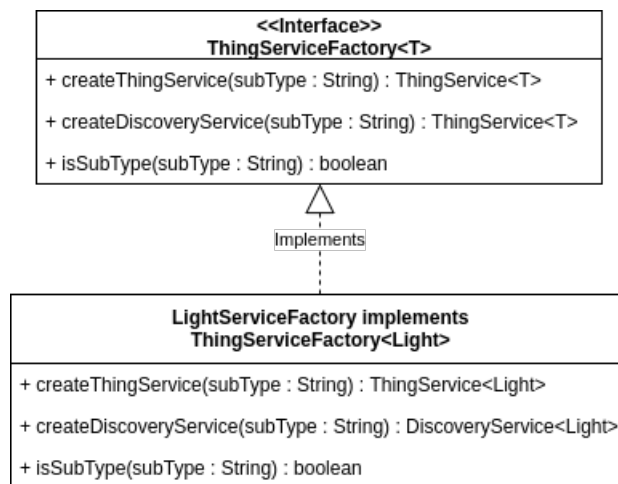


Figura 16: Diagrama de classes: *Factories atualizadas para suportar serviços de descoberta*

Assim, temos um mecanismo para obter a estratégia de descoberta de um determinado subtipo, simplificando todo esse processo da escolha da estratégia num novo método nas *factories*. Os *controllers* então só têm que trabalhar com instâncias de *DiscoveryService*, utilizando o método `setAttributes` da classe abstrata *Attributed* para passar os parâmetros recebidos pelo *controller*.

Por fim, efetua-se a conceção dos *controllers* que expõem os serviços de descoberta. Estes *controllers*, têm um funcionamento muito semelhante aos *controllers* dos serviços das *things*. Este tipo de *controllers* só possui suporte para métodos do tipo GET, retornando uma lista de dispositivos disponíveis na rede local, e os parâmetros necessários para o seu funcionamento.

O *controller* extrai o subtipo fornecido nos parâmetros, utilizando a *factory* para criar o *discovery service* correspondente ao subtipo. Depois disto, são passados o restos dos parâmetros, como a porta de acesso no caso dos *NetworkThingDiscoveryService*, utilizando o método

setAttributes. Por fim, após esta passagem de parâmetros é então executado o método *perform* do serviço de descoberta, retornando a lista de dispositivos encontrados na rede do *hub*.

API HTTP - Visão Geral

Concluindo o desenho arquitetural, agora prossegue-se à definição da API HTTP que irá ser consumida pelo *core* do *middleware*.

Uma API HTTP é normalmente constituída por recursos, identificados por um URL, e pelas ações disponíveis em cada um. Uma ação é identificada pelo verbo HTTP, como já foi demonstrado no *design* dos *controllers*. Exemplos de verbos HTTP podem ser o GET, que tem como função a simples obtenção de um recurso, e o PUT, que tem como função a alteração de um recurso existente.

Posto isto, temos primeiros os URLs de acesso aos dispositivos, cada um com suporte para operações GET e PUT, onde o GET retorna o estado do dispositivo e o PUT permite alterar o estado do dispositivo.

```
/devices/lights  
/devices/locks  
/devices/thermostats  
/devices/weather  
/devices/motionsensors
```

Os parâmetros de acesso, como o endereço IP do dispositivo, devem ser passados à API através de uma *query string*, adicionada ao URL. Estes parâmetros são recolhidos pelo *controller* e posteriormente passados para os serviços respetivos, quer os serviços das *things* ou de descoberta.

```
/devices/light?subtype=rest&address=foo.light.local&port=80
```

Este exemplo permite estabelecer contacto com um dispositivo do tipo *light* e do subtipo *rest*, utilizando o endereço *foo.light.local* e a porta 80 como parâmetros de rede. Internamente, este pedido e os seus parâmetros são passados ao *ThingController* e depois este trata de completar a operação em causa.

Com ambos os verbos HTTP, a resposta a uma chamada da API do *hub* é um objeto do tipo *thing* serializado para JSON. O corpo dos pedidos do tipo PUT também deve seguir a mesma estrutura e formato. De seguida podemos observar um objeto do tipo *light* no formato JSON.

```
{  
    "on" : false  
}
```

Todas as variáveis de instância de qualquer *thing* são convertidas na serialização, de modo a que as restantes representações em JSON dos outros tipos podem ser inferidas através da figura 11.

As rotas para interagir com os serviços de descoberta têm o seguinte formato:

```
/devices/lights/discover  
/devices/locks/discover  
/devices/thermostats/discover  
/devices/weather/discover  
/devices/motionsensors/discover
```

Para encontrar um dispositivo basta efetuar um pedido GET para uma destas rotas, fornecendo como parâmetros um subtipo mais outro subconjunto de parâmetros dependendo do subtipo em questão.

```
/devices/light/discover?subtype=rest&port=80
```

Este pedido procura por dispositivos do tipo *rest* que estejam a operar na porta 80. Um exemplo de resposta tem o seguinte formato:

```
[  
  {  
    "address": "foo.light.local",  
    "port": 80,  
    "subtype": "rest",  
    "type": "Things::Light"  
  }  
]
```

Basicamente trata-se de uma lista JSON com os dispositivos encontrados. Internamente são os serviços que são serializados para JSON, serializando os atributos relevantes ao funcionamento do serviço, neste caso *address*, *port*, *subtype* e *type*.

Segurança

Como em todos os projetos de IoT, a segurança é um tópico importantíssimo e o este projeto não é diferente nesse aspeto. Para atingir níveis aceitáveis de segurança efetuamos um método de autenticação à base de *tokens*, de modo a garantir acesso exclusivo de um *hub* à *api*.

O processo é muito simples, a aplicação irá ter um *controller* e uma rota associado a este, que permite a aquisição de um *token* que garanta o acesso exclusivo aos recursos do *hub*. Este *token* deverá ser obtido pela *api*, no momento em que o utilizador efetua a configuração do seu *hub* através da aplicação cliente, e daí em diante todos os pedidos feitos ao *hub* do utilizador devem ser acompanhados de este *token*, presente no cabeçalho do pedido HTTP, no campo *Authorization*.

De notar que até que um *token* seja adquirido, o *hub* encontra-se desprotegido, mas após a aquisição este fica completamente bloqueado a pedidos que não possuam este *token*. Também é necessário que apesar deste mecanismo, não é o nosso trabalho aumentar a proteção dos dispositivos a que o *hub* acede, apenas os protegemos do acesso externo feito a partir da API do *hub*, o trabalho de proteger os dispositivos de ataques diretos aos mesmos está ao cabo dos fabricantes.

6.3.2 Api

A *api* é a parte central do projeto, contendo toda a lógica e funcionalidades para responder aos problemas encontrados. Como já foi explicado anteriormente, a *api* irá adotar uma arquitetura *REST*, utilizando HTTP como protocolo de transporte. Nesta secção será delineada a arquitetura interna da *api*, e, posteriormente, os vários *endpoints* e métodos da aplicação.

Como arquitetura deverá ser adotado o modelo MVC, que efetua a correta separação entre as três camadas essenciais de uma aplicação *enterprise*, dados, lógica de negócio e apresentação, que mapeia para os conceitos, *model*, *controller*, *view*. Este padrão é bastante predominante em aplicações web, inclusive APIs *REST*, porque torna muito mais manutenível e legível, um sistema complexo com vários tipos de modelos e operações. Além desta abordagem será utilizada uma *service layer*, para efetuar operações complicadas de mais para aparecer num *controller*.

Neste cenário os *models* são apenas *placeholders* para os dados vindos da base de dados, sendo utilizado qualquer tipo de camada ORM para efetuar o mapeamento de dados. Os *controllers* têm como objetivo receber os parâmetros dos pedidos à API, efetuando operações

necessárias na base de dados, através da camada ORM, recorrendo a *service objects* quando a lógica se torna complexa demais.

Ao contrário do *hub*, que utilizava apenas um subconjunto muito simples da arquitetura REST, esta parte do projeto apresenta um nível de complexidade bastante superior, no que toca aos elementos RESTful, e como tal, serão detalhadas as ações REST utilizadas. Como é normal em *designs* REST, as aplicações são divididas em recursos, como por exemplo “utilizadores” ou “casas”, e cada recurso possui métodos CRUD para serem manipulados. Utilizando HTTP, os recursos são representados por URLs e são utilizados os métodos do protocolo para manipular os recursos. De seguida, uma especificação das ações REST mais comuns e a sua implementação utilizando o protocolo HTTP.

Ação	Método HTTP	Identificador de Recurso	Descrição
INDEX	GET	/recurso	Obtém todos os elementos presentes neste recurso
SHOW	GET	/recurso/:id	Obtém um elemento identificado pelo parâmetro variável “id”
CREATE	POST	/recurso	Adiciona um novo elemento ao recurso com os parâmetros fornecidos no <i>body</i> do pedido HTTP
UPDATE	PUT/PATCH	/recurso/:id	Atualiza um elemento identificado pelo parâmetro variável “id” com os parâmetros fornecidos no <i>body</i> do pedido HTTP
DESTROY	DELETE	/recurso/:id	Apaga um elemento identificado pelo parâmetro variável “id”

Tabela 2: Ações REST

Portanto, esta aplicação é composta por recursos, sendo cada um deles manipulado por um *controller*, fazendo alterações no respetivo *model*, sendo as suas alterações representadas pela sua *view*, que neste caso são simples classes de utilidade que serializam objetos para JSON. Os *controllers* serão compostos pelas ações da tabela acima, sendo cada ação um método no *controller*. Posto isto, cada método será mapeado para uma rota, que é uma combinação de um URL com um método HTTP, semelhante ao que vimos na tabela 2.

6.3.2.1 Recursos

Os recursos que compõem esta parte do projeto são baseados no modelo de domínio feito anteriormente, havendo quase um mapeamento perfeito do modelo para recursos da aplicação. De seguida, serão apresentados os diferentes recursos, as rotas de acesso aos mesmos, assim como o formato dos elementos dos diferentes recursos. O formato dos elementos segue os mesmos formatos da especificação JSON, porque de facto são objetos em JSON, tendo os seguintes tipos:

- string
- number
- object (outro objeto JSON)
- array
- boolean
- null

Users

Segue exatamente a mesma lógica da entidade *User* do modelo de domínio, representado os utilizadores da aplicação e servindo como base para os mecanismos de autenticação.

Este recurso é um pouco diferente do habitual, sendo um recurso singular, ou seja, não tem nenhum identificador nem mecanismos de listagem (ação *index*). O objetivo é que este recurso seja utilizado para o registo de novos utilizadores assim como a obtenção e atualização dos mesmos. O identificador deverá ser o próprio mecanismo de autenticação, através de um *token* presente nos cabeçalhos dos pedidos.

Ações

- POST /users -> CREATE
- GET /users/me -> SHOW
- PUT/PATCH /users/me -> UPDATE

Formato dos objetos de entrada

- name - string
- email - string
- password - string

- password_confirmation - string

Formato dos objetos de saída

- id - number
- name - string
- email - string

Homes

Este recurso representa as diversas casas dos utilizadores, contendo informação para interagir com os dispositivos dentro da casa do utilizador através dum *hub*. Essencialmente, uma casa é o objeto representativo do *hub*, sendo o interface de comunicação entre o utilizador e os seus dispositivos. De notar a existência de um *id* no objeto de saída, que é dado ao objeto depois da sua gravação na base de dados.

Ações

- GET /homes -> INDEX
- POST /homes -> CREATE
- GET /homes/:id -> SHOW
- PUT/PATCH /homes/:id -> UPDATE
- DELETE /homes/:id -> DESTROY

Formato dos objetos de entrada

- name - string
- location - string
- tunnel - object

Formato dos objetos de saída

- id - number
- name - string
- location - string
- tunnel - object
- ip_address - string

Os atributos *name* e *location* são apenas utilidades para o próprio utilizador categorizar a sua casa na aplicação, já o *tunnel* é um objeto que contém informação para conectar com o *hub* na casa do utilizador. Internamente, a *api* atribui um endereço IP à casa, o mesmo IP proveniente do pedido HTTP. Na próxima secção, onde irão ser detalhados os aspetos da implementação mais importantes, irá ser delineado a tecnologia utilizada no mecanismo de *tunneling* entre o *hub* e a *api*.

Things

Também conhecido como "coisas", é um termo utilizado para definir os dispositivos dos utilizadores presentes numa das suas casas. Cada *thing* possui as mesmas informações que são enviadas para os *ThingServices* no *hub*, ou seja, parâmetros como o *subtype*, *address* entre outros sendo todos armazenados neste recurso. Os dispositivos pertencem a uma casa e portanto a um utilizador (via esta ultima relação).

Ações

- GET /homes/:home_id/things -> INDEX
- POST /homes/:homes_id/things -> CREATE
- GET /things/:id -> SHOW
- PUT/PATCH /things/:id -> UPDATE
- DELETE /things/:id -> DESTROY

Nestas ações já se pode ver *nesting* de rotas, uma vez que é necessário saber a que casa pertencem os dispositivos. Este *nesting* é superficial, ou seja, se fornecido o *id* de um dispositivo já não é preciso fornecer o *id* de uma casa.

Formato dos objetos de entrada

- name - string
- type - string
- subtype - string
- connection_info - object

Formato dos objetos de saída

- id - number
- name - string
- type - string

- subtype - string
- connection_info - object

O atributo *name*, mais uma vez, só serve para caracterizar o dispositivo, não tendo nenhuma funcionalidade em específico. O tipo e subtipo são essenciais para comunicar com os dispositivos através do *hub*, sendo os restantes parâmetros que variam de acordo com o subtipo fornecidos através do atributo variável *connection_info*.

As *things* terão métodos para obter e alterar o seu estado, *getStatus* e *putStatus* utilizando o objeto *tunnel* de uma *home* para estabelecer conexão. Estes métodos serão utilizados noutro recurso para expor estes mecanismos na *api*.

Things Status

Este recurso permite obter os estados dos dispositivos, evocando os métodos do *get* e *put* dos *ThingServices* presentes no *hub*. Internamente é efetuada uma chamada à API do *hub* através do túnel presente na *home* da *thing* desejada.

Ações

- GET /things/:thing_id/status -> INDEX
- PUT/PATCH /things/:thing_id/status -> UPDATE

O formato dos objetos depende do tipo da *thing* fornecida, sendo que podem ser inferidos através da figura 11, que apresenta os tipos diferentes de *things* e os seus atributos.

Things Discovery

De uma maneira semelhante ao recurso anterior, este permite aceder ao serviço de descoberta de dispositivos presente num *hub*, invocando a API deste a partir do *tunnel*.

Ações

- GET /homes/:home_id/discovery -> INDEX

O formato das respostas é exatamente igual às respostas do *hub*, que podem ser vistas na última secção.

Scenarios

Os cenários, como já vimos anteriormente, são um conjunto de dispositivos e os respetivos estados que lhes devem ser aplicados. Essencialmente este recurso é o ponto de partida

para a funcionalidade dos cenários, que ainda é composta por mais outros dois recursos. Cada cenário pertence a uma casa.

Ações

- GET /homes/:home_id/scenarios -> INDEX
- POST /homes/:homes_id/scenarios -> CREATE
- GET /scenarios/:id -> SHOW
- PUT/PATCH /scenarios/:id -> UPDATE
- DELETE /scenarios/:id -> DESTROY

Formato dos objetos de entrada

- name - string

Formato dos objetos de saída

- id - number
- name - string
- scenario_things - array

O atributo *name*, tal como nos recursos anteriores, é apenas um campo identificativo útil para o utilizador nomear os seus cenários. De notar que no objeto de saída existe um atributo do tipo *array* chamado de *scenario_things*, que na verdade consiste noutro recurso que vamos ver já de seguida.

Scenario Things

Nos cenários é referido que os mesmos consistem num conjunto de dispositivos e os respetivos estados, no entanto estes não foram mencionados, isto porque foi necessário colocá-los noutro recurso devido à sua pluralidade. Um *scenario thing* consiste então na combinação de um estado com um dispositivo, sendo o estado um elemento variável de formato igual aos objetos do recurso *Thing Status*. Em linguagem natural, uma instância de *scenario thing* pode ser "uma luz apagada" ou "uma porta aberta". Tendo este recurso separado dos cenários podemos então combinar várias *scenario things* num só *scenario* podendo ao mesmo tempo "fechar uma porta" e "ligar uma luz". Um *scenario thing* pertence então a um *scenario*, podendo este último ter vários *scenario things*.

Ações

- GET /scenarios/:scenarios_id/things -> INDEX

- POST /scenarios/:scenarios_id/things -> CREATE
- GET /scenarios/:scenarios_id/things/:id -> SHOW
- PUT/PATCH /scenarios/:scenarios_id/things/:id -> UPDATE
- DELETE /scenarios/:scenarios_id/things/:id -> DESTROY

Formato dos objetos de entrada

- status - object
- thing_id - number

Formato dos objetos de saída

- id - number
- status - object
- thing - object

Scenario Applier

No que toca a cenários apenas falta um mecanismo para aplicar o mesmo. Este exemplo também demonstra bem a diferença entre REST e outras abordagens como SOAP. Em SOAP provavelmente teríamos um método remoto *applyScenario* que seria invocado pelo cliente, no entanto, em REST apenas temos recursos e métodos os métodos, que neste caso são os base do HTTP, e portanto criámos um recurso chamado de *Scenario Applier*, chamando o método POST no mesmo (porque se está a invocar alterações no servidor, portanto é boa prática usar POST). Este é um dos poucos recursos sem modelos ou persistência de dados associados.

Ações

- POST /scenarios/:scenarios_id/apply -> CREATE

Este recurso não tem nenhum tipo de objeto de entrada ou saída, enviando o *body* dos pedidos em branco, respondendo apenas com os *status codes* apropriados.

Timed Tasks

Durante a fase conceptual já foi definido o conceito de tarefas, ou *tasks*, basicamente consiste na aplicação de um estado a um dispositivo, ou de vários estados a vários dispositivos (um cenário no fundo), de acordo com uma dada condição. Neste caso a condição é uma expressão temporal, que indica quando uma dada tarefa deve ser aplicada.

Para definir a expressão temporal irão ser utilizadas expressões *cron*, que permitem grande flexibilidade na temporização de tarefas. Desta maneira conseguimos definir temporizadores bastante flexíveis e precisos como pode ser notado nos seguintes exemplos:

- 0 9 * * * - todos os dias às 9 horas
- 0 10 * * 1 - todas as segundas às 10h
- 0 20 * * 1,5 - todas as segundas e sextas às 20h
- */5 * * * * - de 5 em 5 minutos

As expressões *cron* indicam portanto intervalos de tempo podendo escolher os dias da semana e do mês onde as mesmas correm assim como os minutos e horas a que as mesmas devem correr. Recorrendo ao comando *man* ¹ de sistemas UNIX podemos saber mais sobre o *crontab*, o *scheduler* de tarefas destes sistemas, que usam estas tais expressões para definir os intervalos a que as tarefas devem ser executadas.

Ações

- GET /homes/:home_id/tasks/timed -> INDEX
- POST /homes/:homes_id/tasks/timed -> CREATE
- GET /tasks/timed/:id -> SHOW
- PUT/PATCH /tasks/timed/:id -> UPDATE
- DELETE /tasks/timed/:id -> DESTROY

Formato dos objetos de entrada

- id - number
- cron - string
- thing_id - number
- status_to_apply - object
- scenario_id - number

Formato dos objetos de saída

- id - number
- cron - string
- next_run - string
- thing - object

¹ <https://linux.die.net/man/1/crontab>

- `status_to_apply` - object
- `scenario` - object

Os objetos pertencentes a este recurso podem ativar um cenário ou uma única *thing*, e não ambos ao mesmo tempo, portanto não se pode fornecer um `scenario_id` ao mesmo tempo que se fornece um `thing_id`. Assim, temos dois tipos de objetos de entrada, uns com o ID do cenário a aplicar, e outros com um ID de um dispositivo mais o respetivo estado a aplicar, `status_to_apply`. O atributo `cron` é uma expressão temporal como já vimos em cima que define quando a tarefa vai ser aplicada.

Nos objetos de saída temos ainda mais um atributo bastante útil, chamado de `next_run`, que contém a data da próxima execução desta tarefa.

Triggered Tasks

Já possuímos tarefas encadeadas à base de tempo, faltando o outro tipo que são ativadas com base no valor de um certo dispositivo. Nestes cenários de IoT aplicados no espaço casa, é muito comum os utilizadores quererem que certos comportamentos sejam programados de acordo com certas mudanças no ambiente, por exemplo, um cenário comum é ligar luzes utilizando sensores de movimento, o que é possível fazer no nosso caso porque possuímos o tipo base para cada um destes dispositivos.

Posto isto, para alcançar esta funcionalidade foi criado este recurso, que permite aplicar uma dada tarefa com base no valor do estado de um outro dispositivo. Para isso, o utilizador tem que escolher um dispositivo a monitorizar e o estado que desencadeie a execução da tarefa. Internamente a aplicação vai fazendo uma comparação lógica com o estado do dispositivo fornecido com o estado atual do mesmo, e quando essa comparação tiver um valor verdadeiro a tarefa é aplicada.

Tal como as tarefas temporizadas, este novo tipo também segue as mesmas regras quanto à escolha de um cenário ou um dispositivo como alvo de ativação.

Ações

- GET `/homes/:home_id/tasks/triggered` -> INDEX
- POST `/homes/:homes_id/tasks/triggered` -> CREATE
- GET `/tasks/triggered/:id` -> SHOW
- PUT/PATCH `/tasks/triggered/:id` -> UPDATE
- DELETE `/tasks/triggered/:id` -> DESTROY

Formato dos objetos de entrada

- `id` - number
- `thing_to_compare_id` - number
- `status_to_compare` - object
- `comparator` - string
- `thing_id` - number
- `status_to_apply` - object
- `scenario_id` - number

Formato dos objetos de saída

- `id` - number
- `thing_to_compare` - object
- `status_to_compare` - object
- `comparator` - string
- `next_run` - string
- `thing` - object
- `status_to_apply` - object
- `scenario` - object

Os objetos deste recurso possuem todos um *thing_to_compare_id*, que indica o dispositivo que queremos monitorizar, um *status_to_compare_id*, que indica o estado que queremos utilizar na comparação com o dispositivo monitorizado, e por fim, um *comparator*, um operador lógico que deve ser utilizado na comparação entre o dispositivo monitorizado e o seu estado atual com o *status_to_compare* fornecido neste objeto.

Além destes atributos, este tipo de tarefas também contém os alvos de ativação como as tarefas temporizadas, ou seja, ou um cenário, ou um dispositivo com um estado a aplicar.

7. Implementação

Neste capítulo irá ser explicado todo o processo da implementação, desde as escolhas das tecnologias e às metodologias de desenvolvimento aos detalhes e pormenores de implementação de todos os componentes.

Após a definição da arquitetura, foram criados no sistema de controlo de versões remoto *Github* para a implementação da plataforma:

- Simuladores de dispositivos - <https://github.com/um-homewatch/api>
- *Hub* - <https://github.com/um-homewatch/hub>
- *Api* - <https://github.com/um-homewatch/stub-devices>
- *Wrapper* da *api* - <https://github.com/um-homewatch/js-wrapper>
- Aplicação cliente - <https://github.com/um-homewatch/app>

A ordem dos repositórios também ditam a ordem de implementação dos mesmos, seguindo a metodologia incremental e iterativa, onde os componentes foram implementados incrementalmente. O *hub*, a *api* e o *wrapper* exibem todo o método de *continuous integration*, recorrendo ao *GitHub* e ao *Travis*, enquanto que a *api* é *deployed* no *Heroku*.

7.1 Tecnologias

De seguida serão apresentadas as tecnologias utilizadas para implementar a solução. Cada componente de software, o *hub*, a *api*, as simulações dos dispositivos e a aplicação cliente utilizam linguagens e *frameworks* distintas. Aqui, iremos falar das motivações para a sua escolha.

Hub

Para o desenvolvimento deste componente recorreu-se à utilização da linguagem de programação Java, recorrendo à *framework* Spark¹.

Java foi uma escolha óbvia devido à experiência na linguagem e ao seu poder no que toca a genéricos, que permite atingir grandes níveis de reutilização de código, algo que será essencial neste caso, dado que vários dispositivos partilham o mesmo modo de comunicação por exemplo.

A *framework* Spark foi escolhida devido à necessidade de expor um serviço HTTP sem ser necessário associar a um modelo arquitetural, por exemplo, algumas *frameworks* são feitas para utilizar MVC. Neste caso, esta *framework*, que até pode ser considerado uma biblioteca devido à sua simplicidade, tem apenas como função expor um servidor HTTP.

Além do Spark, também foram utilizadas mais algumas tecnologias de relevo, nomeadamente clientes HTTP e CoAP, Jersey² e Californium³ respetivamente, assim como uma biblioteca de reflexões⁴ para desenvolver mecanismos de meta-programação, e por fim a conhecida biblioteca de serialização de objetos Jackson⁵.

Api

Este componente, sendo o mais complexo de todos, necessita de uma *framework* mais evoluída, portanto decidiu-se utilizar *Ruby on Rails* e portanto a linguagem de programação *Ruby*.

Esta *framework* foi uma escolha óbvia, mais uma vez devido à experiência em trabalhos académicos passados, e também devido à simplicidade de utilização. Utilizando *Ruby on Rails* podemos dedicar mais tempo a funcionalidades em vez de detalhes de implementação. Outra motivação é a existência de um bom suporte de *background tasks*, que como já se falou anteriormente, era um desafio a enfrentar, para se poder implementar as tarefas automatizadas.

Utilizando esta *framework*, tarefas como a camada de *ORM*, filtragem de parâmetros dos pedidos HTTP, serialização de objetos para *JSON* são todas tratadas pelo *Rails*, podendo dar assim mais atenção a outros aspetos mais importantes da aplicação.

1 <http://sparkjava.com>

2 <https://jersey.github.io/>

3 <https://www.eclipse.org/californium/>

4 <https://github.com/ronmamo/reflections>

5 <https://github.com/FasterXML/jackson>

No ecossistema do Ruby a utilização de dependências externas é extremamente facilitada com o mecanismo das *rubygems*, um módulo que efetua a gestão das dependências. Utilizando este mecanismo, é possível incluir um ficheiro denominado por *Gemfile*, que lista todas as dependências externas da aplicação. Assim incluímos várias dependências essenciais, como o próprio *Rails* por exemplo, o *Delayed Job* que permite uma declaração de tarefas em segundo plano de modo mais simples, assim como a sua extensão *Delayed Cron Job*, que permite definir tarefas com base numa expressão *cron*, muito útil para definir as nossas *Timed Tasks*.

Simulação de Dispositivos

Para as simulações de dispositivos recorreu-se à linguagem Javascript com o *runtime serverside* Node.JS, utilizando a framework *express*⁶. A combinação de *javascript* com o *express* permite muito rapidamente desenvolver servidores web, uma vez que estes simuladores irão fornecer APIs HTTP.

Também se utilizou a biblioteca *coap-router*⁷, que possui um funcionamento muito semelhante ao *express*, para criar os dispositivos que utilizam CoAP em vez de HTTP, como protocolo de comunicação.

De seguida, parte do código que compõe a simulação de um dispositivo do tipo *light*

```
app.get("/status", function (req, res) {
  console.log(`RECEIVED GET REQUEST: ${JSON.stringify(req.body)}`);
  res.send({ power: power });
});

app.put("/status", (req, res) => {
  console.log(`RECEIVED PUT REQUEST: ${JSON.stringify(req.body)}`);
  power = req.body.power;
  res.send({ power: power });
});
```

Este pequeno script expõe um serviço numa porta determinada na invocação do mesmo (ou 80 por defeito), que possui duas rotas para o URL *"/status"* uma utilizando o método GET e outra o PUT, permitindo "desligar" e "ligar" esta luz imaginária manipulando o estado do equipamento através de um objeto JSON que o descreve, como por exemplo:

⁶ <https://expressjs.com/>

⁷ <https://github.com/MagicCube/coap-router>

```
{  
  "power": true  
}
```

Foram feitos outros scripts muito semelhantes a este, mudando apenas o formato do objeto, para fechaduras, estações meteorológicas, termostatos e sensores de movimento. Quando a dispositivos que utilizam CoAP, apenas se simulou dispositivos do tipo lâmpada.

Wrapper da api

Este componente, como já foi explicado na arquitetura, providencia um adaptador para a *api* do *middleware* na linguagem da aplicação cliente, que neste caso, vai ser *Javascript*.

Este componente é bastante simples, tendo apenas que fornecer uma API utilizando a linguagem, onde internamente faz uso da API em HTTP desenvolvida para este *middleware*.

```
class Users {  
  constructor(axios) {  
    this.axios = axios;  
  }  
  
  /**  
   * Registers a user  
   * @param {Object} user  
   * @param {string} user.name  
   * @param {string} user.email  
   * @param {string} user.password  
   * @return {Promise}  
   */  
  register(user) {  
    return this.axios.post("/users", { user });  
  };  
}
```

No *wrapper*, utilizamos a biblioteca *axios*⁸, um cliente HTTP muito utilizado em aplicações desenvolvidas em *Javascript*, para interagir com os *endpoints* da *api* do *middleware*, que opera utilizando o protocolo HTTP.

O *wrapper* oferece uma classe base que contém vários módulos, um para cada *endpoint* da API. Cada módulo é também uma classe, que é instanciada utilizando o cliente HTTP *axios*.

⁸ <https://github.com/axios/axios>

```
class HomewatchApi {
  constructor(url, cache) {
    this.axios = axios.create({
      baseURL: url,
    });

    if (cache === true) {
      this.axios.get = getFromCache(this.axios.get);
      this.axios.interceptors.response.use(cacheClear);
    }
  }

  set auth(auth) {
    Object.assign(this.axios.defaults, {
      headers: { authorization: `Bearer ${auth}` }
    });
  }

  /**
   * @return {Users}
   */
  get users() {
    return new Users(this.axios);
  }
}
```

A classe base da API pode ser instanciada com dois argumentos, *url* que é uma string que indica o *endpoint* da API, que apenas é variável neste contexto de desenvolvimento. Num ambiente de produção seria o endereço fixo da API, e um valor booleano *cache* que ativa o *caching* dos pedidos GET feitos à API. Também temos um atributo *auth*, que deve conter um *token* de autenticação devolvido pelo método *login* da API.

De seguida poderemos então ver em ação o funcionamento deste *wrapper*:

```
async function exemplo(){
  const jose = {
    name: "jose",
    email: "jose@mail.com",
    password: "123456",
  };
}
```

```
// login
let user = await homewatch.users.login(jose);
homewatch.auth = user.data.jwt;

// current user
let currentUser = await homewatch.users.currentUser();

// list user's homes
let homes = await homewatch.homes.listHomes();
}
```

Todos os métodos do *wrapper* devolvem *Promises*, um mecanismo que facilita a programação assíncrona muito predominante nesta linguagem, devido a esta ser maioritariamente utilizada em contextos de aplicações clientes interativas. Uma *Promise* é um objeto que representa uma operação assíncrona que irá ser completada no futuro, podendo também falhar. Neste exemplo, utilizamos o mecanismo *async/await* introduzido pelo ES6, uma revisão da especificação do *Ecmascript* onde o *Javascript* é uma implementação deste, que facilita imenso a programação assíncrona neste ambiente.

Aplicação Cliente

Por fim, foi desenvolvida uma aplicação cliente para efetuar uma demonstração da API utilizando a *framework* *Ionic 2*⁹, que recorre a outra *framework* de *frontend* bastante popular chamada de *Angular 2*¹⁰.

O *Ionic* é essencialmente um ajudante à conversão de uma aplicação web para uma aplicação *mobile*, recorrendo ao *Angular* para providenciar um bom ambiente de desenvolvimento de aplicações gráficas ao mesmo tempo que oferece vários mecanismos para interagir com as APIs nativas dos aparelhos *mobile*. Essencialmente o *Ionic* faz uso dos *browsers* embebidos presentes nos 3 sistemas operativos suportados por esta *framework*, *Android*, *iOS* e *Windows Phone*, para apresentar uma aplicação web desenvolvida em *Angular* empacotada numa aplicação "nativa".

Como esta *framework* utiliza o *Angular*, temos que recorrer a *Typescript*, uma alternativa ao *Javascript* que basicamente adiciona *type safety* à linguagem. De notar que, apesar de não ser *Javascript*, que é a linguagem onde se desenvolveu o *wrapper*, podemos utilizar-lo na mesma, uma vez que em *Typescript* podemos ter dependências em módulos *Javascript* na mesma.

⁹ <http://ionicframework.com/>

¹⁰ <https://angular.io/>

Esta abordagem oferece algumas vantagens e também desvantagens onde neste caso, as primeiras superam as últimas. É possível desde já desenvolver aplicações para 3 sistemas operativos recorrendo à mesma *codebase*, o que é incrível por si só, permitindo ainda desenvolver as aplicações com linguagens feitas para aplicações gráficas, que é o caso do HTML, JS (Typescript neste caso) e CSS, mais fáceis de utilizar do que as alternativas, Java, Swift e C#. No entanto, também temos algumas desvantagens, sendo a performance um deles, porque é muito difícil combater com código nativo, uma vez que estamos essencialmente a correr a nossa aplicação num *stripped down browser*, e o acesso às APIs nativas também é um problema, porque apesar de o *Ionic* fornecer *wrappers* a estas, nem todas são acessíveis através dum ambiente como este.

Esta *framework* facilitou imenso o processo de desenvolvimento desta aplicação, que não era o foco da dissertação, mas permitiu mesmo assim apresentar algo onde se pudesse testar toda a aplicação num contexto mais semelhante ao mundo real, onde um utilizador pudesse realmente usufruir das abordagens aqui feitas. Neste capítulo não vamos abordar mais nem o *wrapper* nem esta aplicação, deixando isso para o capítulo 8, onde vamos ver a aplicação em ação de modo a que as funcionalidades do *middleware* fiquem mais claras.

7.2 Detalhes

Nesta secção irão ser detalhados alguns aspetos de uma importância acrescida, como o mecanismo de *tunneling* que efetua a ligação entre vários *hubs* e a *api*, assim como elementos de meta-programação que permitem o crescimento do sistema em termos de dispositivos suportados, reduzindo o tempo de implementação e a manutenibilidade destes componentes. Também alguns aspetos de segurança, muito importante em contextos de IoT, vão ser explicados em mais detalhe.

7.2.1 Hub

Utilizando a framework web *Spark*, conseguimos expor os nossos *ThingServices* via uma API HTTP, como é possível ver no seguinte exemplo:

```
ThingServiceFactory<Light> thingServiceFactory = new LightServiceFactory();
ThingController<Light> controller = new ThingController(
    thingServiceFactory,
    Light.class);
```

```
Spark.get("/devices/lights", controller::get);  
Spark.put("/devices/lights", controller::put);
```

Este exemplo faz uso da tal injeção de dependências explicada durante a fase da arquitetura, conseguindo assim ter apenas um controlador que faz a gestão de todos o tipos de dispositivos. Depois, são criadas duas rotas, uma utilizando o método GET e PUT, que se associam aos respetivos métodos no controlador. Aquela notação faz parte da API do Java 8, que permite passar métodos por referência.

Internamente, os métodos *get* e *put* recebem dois objetos da *framework* utilizada, do tipo *Request* e *Response*, que têm todos os elementos do pedido HTTP proveniente do cliente assim como o objeto que representa a resposta a ser enviada. Com estes objetos é possível manipular cabeçalhos, *query parameters*, entre outros.

No entanto, apesar dos mecanismos arquiteturais utilizados para simplificar o processo de adição de tipos e subtipos de dispositivos, ainda há um problema, sempre que é criado um novo tipo de dispositivo é preciso criar estas rotas manualmente, estando a aumentar a complexidade do código com *boilerplate code*, código repetido sem grande significado. Portanto recorremos a elementos de meta-programação para resolver esta duplicação de código.

Meta-programação consiste então nas muitas maneiras que um programa consegue manipular o seu próprio funcionamento durante o tempo de execução. Em muitas linguagens orientadas a objetos, como o Java por exemplo, existe um mecanismo muito usado de meta-programação, chamado de reflexões, que permite a uma aplicação obter informações sobre as classes e elementos que a compõem, por exemplo, obter uma lista com o nome dos métodos que uma classe define, ou obter todas as classes que implementam um dado interface ou que estendem uma dada classe.

Para resolver este pequeno problema decidiu-se recorrer à biblioteca de *reflections* do Java, que permite inferir durante o *runtime*, a lista de classes que implementa o interface *Thing*. Isto é bastante útil porque podemos utilizar os métodos definidos no interface, *getFactory* e *getStringRepresentation*, para automaticamente criar todas as rotas necessárias para o funcionamento do *hub*.

Primeiro foi criada uma classe auxiliar com um método *static* para obter todas as *Things* do projeto.

```
class ClassDiscoverer {  
    public static Set<Class<? extends Thing>> getThings() {  
        Reflections reflections = new Reflections("homewatch.things");  
        return reflections.getSubTypesOf(Thing.class);  
    }  
}
```

```

    }
}

```

Este método obtêm todas as classes que implementam o interface *Thing* e que estejam presentes no pacote no pacote *homewatch.things*, esta pesquisa por pacote permite reduzir o tempo de execução da mesma.

Depois utilizamos este método para criar as rotas correspondentes aos 5 tipos de dispositivos que se implementaram.

```

private static void deviceControllers() {
    ClassDiscoverer.getThings().forEach(klass -> {
        try {
            Thing t = klass.newInstance();
            ThingServiceFactory thingServiceFactory = t.getFactory();
            ThingController controller = new ThingController(thingServiceFactory, klass);

            Spark.get("/devices/" + t.getStringRepresentation(), controller::get);
            Spark.put("/devices/" + t.getStringRepresentation(), controller::put);
        } catch (InstantiationException | IllegalAccessException e) {
            LoggerUtils.logException(e);
        }
    });
}

```

Este método é chamado nas classes de *setup* do projeto, que tratam do processo de inicialização e configuração do pequeno servidor, e essencialmente percorre a lista de classes retornada pelo método *getThings*, utilizando os dois métodos utilitários, *getFactory* e *getStringRepresentation* para criar as rotas de acesso aos serviços dinamicamente. Desta maneira, desde que os interfaces dos dispositivos retornem a *factory* respectiva e afixando a descrição do dispositivo à rota de acesso.

Exemplificando o funcionamento deste mecanismo, se tivermos um dispositivo *Light* onde o método *getFactory* retorna um *LightServiceFactory* e onde o método *getStringRepresentation* retorna *"lights"*, irá ser criado um *controller* utilizando um *LightServiceFactory*, e depois serão criadas duas rotas, *"/devices/lights"* com os métodos GET e PUT.

Assim, conseguimos automaticamente instanciar todos os *controllers* sem ter que repetir pedaços de código idênticos, que iriam acabar por se tornar muito numerosos com o crescimento de tipos de dispositivos.

7.2.2 Api

No que toca à API temos 3 detalhes importantes que vale a pena referir, os mecanismos de *tunneling* para comunicar com o *hub*, os elementos de metaprogramação utilizados nas *triggered tasks*, e por fim, os mecanismos de *background jobs* utilizados para lidar com as tarefas e outros aspetos.

7.2.2.1 Tunneling

Como já foi referido várias vezes durante esta dissertação, a comunicação entre o *hub* e a *api* iria ser feita através de um qualquer mecanismo de *tunneling*, de modo a evitar tecnologias como encaminhamento de portas ou *UPNP*, que têm processos de configuração complicados e algumas vulnerabilidades a nível de segurança.

Portanto, como tecnologia de *tunneling* decidiu-se utilizar o *ngrok* ¹¹, um software que permite expor serviços web locais, mesmo por de trás de *firewalls* e NATs, disponibilizando este serviço num URL de acesso público. O *ngrok* apesar de ser um software orientado a programadores, que desejam expor as suas aplicações web locais sem recorrer a *port forwarding* para efeitos de demonstração, demonstrou ser uma das melhores soluções, pelo nesta fase do desenvolvimento.

O *ngrok* irá funcionar como um processo de background, que é executado sempre que o *hub* é inicializado, expondo a porta 4567 num URL gerado automaticamente.

```
ngrok http 4567
```

```
https://370ef754.ngrok.io
```

De notar que é gerado um URL cujo acesso pode ser feito utilizando o protocolo HTTPS, que assegura a encriptação dos dados que circulam neste túnel. Este URL deve ser guardado no recurso *Home* presente na *api* do *middleware*, no campo *tunnel*, assim, o *middleware* já consegue aceder à API exposta pelo *hub*.

Neste preciso caso, o *hub* possui antes um ficheiro de configuração com o túnel pré-definido

```
web_addr: 0.0.0.0:4040
region: eu
```

¹¹ <https://ngrok.com/>

```
tunnels:
  homewatch-hub:
    proto: http
    addr: 4567
```

Este ficheiro define o endereço base da API do *ngrok*, utilizada num mecanismo que vamos explorar já de seguida, a região do túnel, que pode ser na Europa ou nos Estados Unidos, e por fim, o túnel *homewatch-hub*, expondo a porta 4567 que utiliza o protocolo HTTP.

A própria API do *ngrok* é utilizada pelo *hub* para fornecer o URL do túnel às aplicações clientes, para estas poderem criar o recurso *Home* na *api* do *middleware* fornecendo o túnel do seu *hub*.

```
http://homewatch-hub:4567/tunnel

{
  "url" : "https://a10d2075.eu.ngrok.io"
}
```

Internamente o *hub* efetua um pedido à API do *ngrok* e extrai o URL do túnel, devolvendo-o no típico formato JSON. As aplicações podem sempre efetuar pedidos para aquele URL, uma vez que os *routers* os utilizadores possuem em casa têm na sua grande maioria um próprio servidor DNS, que resolve IPs para os *hostnames* das máquinas presentes na rede que adquiriram o seu IP via DHCP, e como o *hostname* de todos os *hubs* é *homewatch-hub*, este processo torna-se perfeitamente fazível. Este mecanismo é bastante útil para efetuar o *setup* inicial do *hub* com a API do *middleware*, podendo obter o URL do *hub* sem grandes problemas. No entanto, é preciso referir que obviamente só podemos recorrer a este mecanismo quando estamos na rede local do *hub*, sendo impossível recorrer a esta operação numa rede externa.

Apesar da facilidade de uso e configuração dos túneis do *ngrok*, temos problemas claros utilizando esta abordagem. De seguida iremos enumerar os principais problemas na utilização desta tecnologia.

Geração Aleatória de URLs

O primeiro contratempo encontrado logo antes de se montar o sistema à volta desta tecnologia foi a geração automática do URL, que obriga os utilizadores a reconfigurarem os dados da sua casa, na aplicação que acede ao *middleware*, sempre que o *hub* reinicia (regenerando assim o URL). A solução parcial para este problema foi o desenvolvimento do mecanismo de

deteção à base dos servidores de DNS locais (dos *routers* do utilizador) e da API do *ngrok*, que se traduz no clique de um botão na aplicação para detetar e atualizar o túnel de uma casa. Apesar de ser uma boa solução, não é perfeita, havendo o inconveniente de as tarefas automatizadas não funcionarem enquanto que o URL do túnel não for atualizado, uma vez que qualquer tentativa de acesso ao *hub* vai acabar com um erro.

Limite de Conexões HTTP

O outro problema consiste no limite de conexões HTTP imposto pelo *ngrok* (20 por minuto), que põe em causa o funcionamento do *polling* efetuado aos dispositivos através deste túnel no âmbito das tarefas automatizadas, uma vez que, cada *triggered task* está sempre a monitorizar um dado dispositivo. A solução que resolveu este problema com sucesso foi a utilização de conexões persistentes HTTP¹², que permite a reutilização de conexões HTTP, não causando a exaustão prematura das mesmas. Esta limitação também é evidente apenas no plano base (que é grátis), que foi utilizado para desenvolvimento.

Segurança

Por fim temos o problema de segurança, uma vez que apesar do tráfego dentro do túnel utilizar HTTPS (estando encriptado portanto), qualquer pessoa pode aceder ao túnel, caso tenha o URL de acesso, e enviar comandos para o *hub*. De momento resolvemos esse problema manualmente, com o tal sistema de *tokens* referido na arquitetura.

Num cenário ideal podia-se utilizar o *ngrok link*¹³, que oferece todos os benefícios deste software assim como outras funcionalidades que tornam a utilização desta tecnologia em massa muito mais acessível. Este plano resolve todos os problemas encontrados oferecendo ainda mecanismos avançados para a gestão em massa de túneis, que é exatamente o que acontece no nosso caso.

- Gestão de túneis *ngrok* em grande escala
- Reserva de endereços, basicamente garantindo a cada túnel um endereço único que nunca muda
- Encriptação avançada com *tokens* e credenciais diferentes para cada túnel
- Utilização de HTTP/2 que oferece ganhos de performance consideráveis face ao HTTP
- IP *whitelisting*, que permite garantir o acesso exclusivo apenas a certos IPs (neste caso apenas o IP da rede local do utilizador e o IP do servidor da *api* eram *whitelisted*)

¹² RFC HTTP Persistent Connections: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html>

¹³ *ngrok link*: <https://ngrok.com/product/ngrok-link>

Todas estas vantagens eram oferecidas a partir de uma API poderosa exclusiva aos assinantes do *ngrok link*. Esta tecnologia não foi utilizada uma vez que era orientada a negócios e a infraestruturas, não conseguindo ter acesso à mesma, no entanto, a escolha recaiu na versão base do *ngrok* que permite demonstrar o funcionamento desta arquitetura e num cenário perfeito seria utilizado o *ngrok link*.

No entanto, podemos concluir que, apesar de todas as desvantagens deste mecanismo, foi o único que se mostrou fiável o suficiente, tendo um custo de implementação bastante baixo. A utilização deste mecanismo simplificou bastante o processo de desenvolvimento e testes, permitindo alcançar a funcionalidade essencial do controlo remoto de dispositivos fora da rede local dos mesmos, mantendo um nível de segurança aceitável.

Exemplo de utilização

Utilizando este túnel, é possível então definir e obter os estados dos dispositivos na casa do utilizador, para isso basta chamar a API do *hub* através deste URL gerado pelo *ngrok*.

```
def get_status
  Curl.get(uri) do |http|
    http.headers["Content-Type"] = "application/json"
    http.headers["Authorization"] = home.token
  end
end

def send_status(status)
  Curl.put(uri, status.to_json) do |http|
    http.headers["Content-Type"] = "application/json"
    http.headers["Authorization"] = home.token
  end
end

private

def connection_params
  connection_info.merge(subtype: subtype)
end

def uri
```

```
home.tunnel + self.class.route + "?" + connection_params.to_query
end
```

Aqui definimos dois métodos auxiliares que ajudam a determinar a rota do dispositivo, onde fazemos uso do URL do túnel, combinado com a descrição da rota *self.class.route*, seguido depois dos parâmetros de conexão, basicamente os parâmetros presentes no *connection_info* mais o subtipo, que são enviados na *query string* do pedido. Depois fazemos uso do cliente HTTP *curl* para então comunicar através do túnel, no caso do *get_status* basta apenas fazer um pedido GET para a rota gerada para obter o estado do dispositivo, já o método *put_status* basta fazer um pedido HTTP do tipo PUT com o novo estado desejado para o dispositivo.

De resto, este túnel só voltar a ser utilizado na descoberta de dispositivos, onde basicamente o funcionamento é o mesmo.

7.2.2.2 Meta-Programação

Este componente, tendo sido desenvolvido em *Ruby on Rails*, teve um desenvolvimento rápido e sem grandes problemas de repetição de código, devido às políticas utilizadas na *framework*. Mesmo assim, houve uma pequena parte da aplicação onde se recorreu a meta-programação para efetivamente reduzir a quantidade de código desenvolvendo, melhorando a legibilidade e a manutenibilidade do mesmo.

O mecanismo está presente nas *triggered tasks*, e é responsável por efetuar a comparação entre o estado fornecido na tarefa e o estado atual do dispositivo a monitorizar. Recorreu-se então a um mecanismo muito interessante presente no *ruby*, que permite chamar um método num objeto dinamicamente, através de um método presente em todos os objetos do ruby, o *send*.

```
class HelloWorld
  def hello(n)
    n.times { puts "Hello World" }
  end
end

hello_world = HelloWorld.new

hello_world.send("hello", 5)
```

Utilizando este método podemos chamar outros métodos, inclusive métodos privados, passando como argumento o nome do método e os restantes argumentos caso existam. Neste

exemplo, criamos uma classe com um método *hello*, que imprime "Hello World" *N* vezes, e depois executamos esse método recorrendo ao *send*, passando o nome do método que queremos executar como segundo argumento, e os restantes argumentos de seguida, deste caso, o número de vezes que queremos imprimir aquela *string*. De notar que podemos passar o nome do método que desejamos executar como *string* ou símbolo, ou seja "hello" ou *:hello*.

No nosso caso recorreremos a este mecanismo para executar uma operação de comparação variável, comparando um dado valor com outro com base num operador lógico à escolha. Isto porque o caso de uso das *triggered tasks* era os utilizadores poder criar regras com base no estado de outros dispositivos com a maior liberdade possível, podendo criar tarefas como os seguintes exemplos:

- Ligar a luz da garagem se o sensor de movimento colocado na mesma detetar movimento
- Desligar o sistema de ar condicionados interior se a temperatura registada pela estação meteorológica do jardim for inferior a 18°C

Então, recorrendo a meta-programação, podemos pegar no *comparator* presente em cada *triggered task*, e utiliza-lo para comprar os estados do dispositivo monitorizado e o estado associado à tarefa.

```
def compare_remote_status(remote_status, status, comparator)
  status.each do |key, value|
    return false unless remote_status[key].send(comparator, value)
  end

  true
rescue NoMethodError
  false
end
```

Este método compara ambos os estados chave a chave, sendo que o *remote_status* é proveniente do dispositivo monitorizado e *status* é o estado alvo fornecido à tarefa em questão, utilizando um comparador que pode ser um qualquer operador lógico, como por exemplo, *==* ou *>=*, e quando a comparação tem um valor falso, terminamos de imediato o método, retornando um valor falso provando que ambos os estados não são iguais. Caso as comparações entre todos os elementos de ambos os estados tenham um valor verdadeiro, o valor retornado por este método também será verdadeiro, e, nesse caso, a tarefa será ativada.

Obviamente que este mecanismo é algo arriscado, uma vez que é propenso à injeção de instruções, porque o *send* interpreta todo o código enviado como parâmetro, mas neste caso

filtramos o valor do atributo *comparator*, aceitando apenas os comparadores lógicos, eliminando este mesmo risco.

7.2.2.3 Async Jobs

O último detalhe de relevo são as tarefas de *background*, *jobs* assíncronos que são executados em segundo plano, para não sobrecarregar os servidores aplicativos que tratam da lógica computacional e do tratamento de pedidos HTTP.

Por defeito, o *Rails* já inclui uma biblioteca de tratamento de tarefas assíncronas, chamada de *ActiveJob*, que permite delegar tarefas computacionais com um tempo de execução moderado ou longo para uma *queue* onde serão processados assincronamente. Tarefas como o envio de emails, *newsletters*, análise de dados, envio de notificações são normalmente executados utilizando esta biblioteca.

Por defeito o *ActiveJob* suporta dois modos de funcionamento, *inline* ou *async*, onde o primeiro executa as tarefas imediatamente (útil durante o desenvolvimento ou testes da aplicação), e o segundo executa as tarefas no mesmo processo só que numa *thread pool* limitada, alcançando uma execução assíncrona. Ambos estes modos são úteis para desenvolvimento, mas muito limitados para executar durante a fase de produção, sendo necessário mecanismos superiores para processar estas tarefas, sendo o *DelayedJob* um deles, um mecanismo que guarda os trabalhos por executar numa base de dados, que pode ser a mesma base de dados utilizada pelo *Rails*, tendo depois um ou vários processos consumidores, que consultam esta base de dados para irem executando tarefas assim que tiverem disponibilidade, funcionando como uma *queue* essencialmente. Isto tem muitas vantagens a nível de performance, porque podemos separar completamente o processamento de tarefas assíncronas para outros processos e até outros sistemas, desde que tenham acesso à base de dados que contém as tarefas.

O *DelayedJob* ainda tem uma outra funcionalidade interessante, oferecida por um plugin chamado de *DelayedCronJob*, que permite criar tarefas recorrentes recorrendo a expressões *cron*, basicamente oferecendo a funcionalidade das nossas *Timed Tasks*, que também utilizam expressões *cron*.

De seguida podemos ver um exemplo de um *ActiveJob*:

```
class LongRunningJob < ApplicationJob
  queue_as :default

  def perform(*args)
    # tarefa complexa de longo tempo de execução
  end
end
```

```
end  
end
```

Essencialmente, cada tarefa é uma classe, possuindo um método *perform* que executa a tarefa em si com os dados argumentos. Estas tarefas podem ser executadas da seguinte maneira:

```
# executar assincronamente  
LongRunningJob.perform_later(args)  
  
# executar daqui a uma semana  
LongRunningJob.set(wait: 1.week).perform_later(guest)
```

No entanto, o *DelayedJob* oferece um mecanismo mais simplista, que é o que é usado na *api*. De seguida um pequeno exemplo:

```
thing.delay.put_status(on: false)
```

Como podemos ver, isto torna a criação de *async jobs* muito mais simplista. Com o plugin *DelayedCronJob* podemos especificar a tal expressão *cron*:

```
thing.delay(cron: "0 0 12 * *").put_status(on: false)
```

Este mecanismo é utilizado então em 3 aspetos diferentes, gestão de tarefas *triggered* e *timed*, assim como os mecanismos de segurança do *hub*, e, por fim, a aplicação de cenários, onde os dispositivos associados a um cenário são ativados assincronamente, para não bloquear os servidores aplicacionais.

7.2.2.4 Service Objects

Como já foi referido na arquitetura, a *api* iria recorrer a uma arquitetura MVC, bastante utilizada em paradigmas web, que também é a arquitetura base adaptada pela framework *Ruby on Rails*. Nesta arquitetura dividimos o *core* da aplicação em 3 tipos de componentes, *models*, que representam a informação base do sistema, *views*, que são a representação visual ou textual dos dados do sistema, e por fim, os *controllers* que aceitam parâmetros externos fazendo alterações nos *models*.

Utilizando o *Rails* esta arquitetura torna-se muito simples de utilizar, tendo um *model* para cada recurso da aplicação, como um *User*, uma *Home* ou uma *Thing*, tendo cada um destes métodos para obtenção e manipulação de dados, recorrendo á biblioteca criada pela equipa do *Rails*, o *ActiveRecord*.

De seguida podemos ver um exemplo de um *controller* muito simples:


```
class TodosController < ApplicationController
  # POST /todos
  def create
    todo = Todo.create!(todo_params)

    render json: todo
  end

  private

  def todo_params
    # whitelist params
    params.permit(:title, :created_by)
  end
end
```

Como podemos ver este código segue muitas das filosofias valorizadas pelo *Rails*, sendo muito simples e fácil de compreender. Outra convenção é a nomeação dos métodos do *controller*, devendo seguir a tal arquitetura REST, que já visitamos anteriormente no capítulo da arquitetura.

No entanto, este *controller* é muito básico, apenas tendo funcionalidades CRUD. Por vezes, certos métodos dos *controllers* efetuam operações mais avançadas, por exemplo, no nosso caso sempre que um utilizador cria uma casa, a aplicação deve automaticamente efetuar a autenticação com o *hub*. Se introduzisse-mos essa lógica no método *create* do *controller*, este ficaria muito grande e difícil de manter, portanto recorreremos a um *service object*.

```
def create
  create_home_service = CreateHome.new(user: current_user, params: home_params)
  home = create_home_service.perform

  if create_home_service.status
    render json: home, status: :created
  else
    render json: home.errors, status: :unprocessable_entity
  end
end
```

Como podemos ver, mantemos uma legibilidade aceitável no método, extraindo a lógica mais complexa para um serviço. Os serviços são objetos comuns, que possuem um método

perform, responsável por executar a operação correspondente a este serviço. Caso a operação seja completada com sucesso, o serviço irá definir o seu atributo *status* como verdadeiro.

Este serviço em particular, grava o registo da casa do utilizador na base de dados, efetua a autenticação no *hub* gravando o *token* no próprio registo da casa, e depois cria um *DelayedJob* que regenera este *token* todos os dias às 00:00.

8. Demonstração

De seguida, iremos ver uma demonstração das capacidades do *middleware*, fazendo uso da aplicação cliente e dos simuladores de dispositivos, desenvolvidos para este efeito. Neste capítulo vamos detalhar o ambiente utilizado para a demonstração, assim como todas as funcionalidades chave do *middleware*, desde a configuração inicial de um *hub* até à definição de tarefas automatizadas.

8.1 Preparação

Para efetuar esta demonstração instalou-se o *hub* num *Raspberry PI* a correr o sistema operativo *Raspbian*, basicamente uma versão para processadores ARM do popular sistema operativo baseado em *Linux*, *Debian*. Além disto teremos 3 simuladores de dispositivos a correr neste sistema, uma fechadura, um sensor de movimento e um termostato.

Num cenário real, um utilizador teria uma *box* já pré-configurada que continha o *software* respetivo ao *hub*, e alguns dispositivos já instalados na sua residência. Neste caso, todos os dispositivos correm a partir do mesmo *Raspberry* assim como o *hub*, apenas para efeitos de demonstração, o funcionamento seria o mesmo caso eles estivessem a ser simulados noutros sistemas.

A aplicação será testada no modo de desenvolvimento, ou seja, a aplicação corre num *browser* num *desktop* de desenvolvimento em vez de um dispositivo *mobile*, isto porque é mais prático para efeitos de testes e documentação desta demonstração. Apesar disto, a aplicação foi testada no sistema operativo *Android*, e retinha toda a funcionalidade demonstrado no modo de desenvolvimento.

Além disto, também fizemos uso do emulador do *Philips Hue*¹, a correr no sistema de desenvolvimento onde se vai testar a aplicação.

De seguida poderemos ver a distribuição destes componentes num simples diagrama.

¹ <http://steveyo.github.io/Hue-Emulator/>

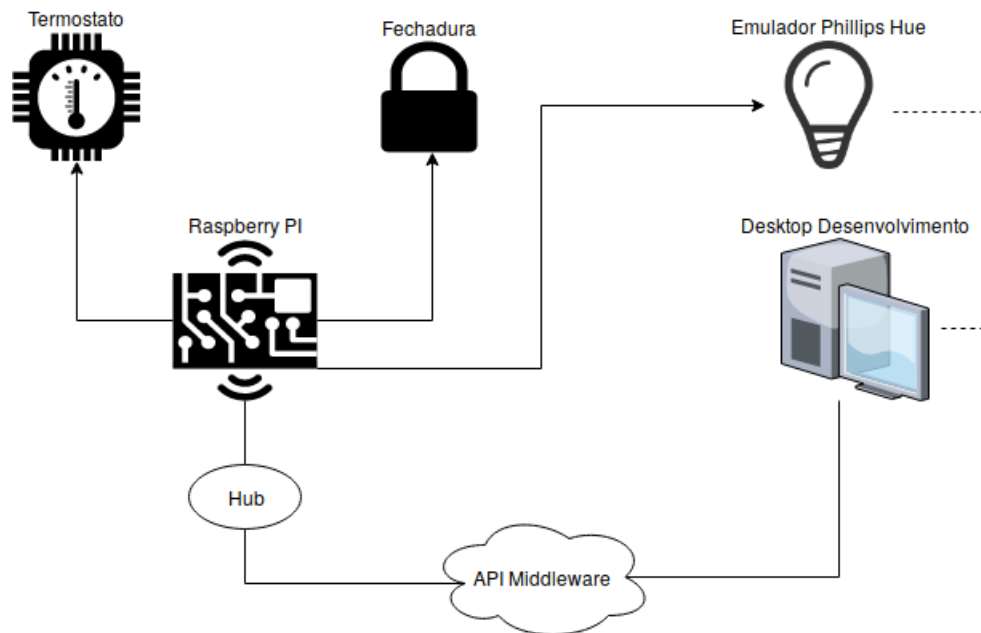


Figura 17: Componentes preparados para a demonstração

De notar que apesar da própria aplicação cliente estar a correr na mesma rede local dos dispositivos e do *hub*, toda a comunicação é feita através da API do *middleware*, recorrendo ao software de tunelização que já descrevemos anteriormente.

8.2 Execução

Agora vamos então efetuar a demonstração prática do *middleware*, recorrendo á aplicação cliente. Primeiro iremos verificar o registo e o *login* de um utilizador, seguido depois da configuração do seu espaço casa e do seu *hub*, passando depois pela configuração de alguns dispositivos, finalizando com a criação de um cenário e de algumas tarefas automatizadas.

Inicialmente temos, como é costume, o ecrã inicial de *login*. Muito simples de utilizar, tendo uma opção para criar um utilizador.

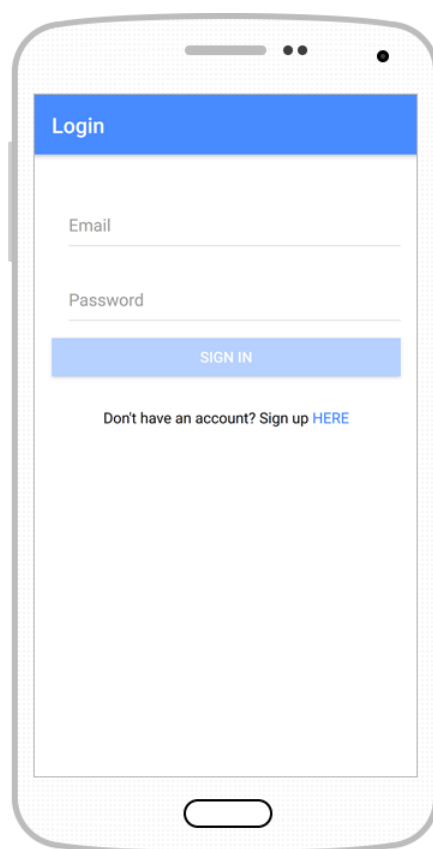
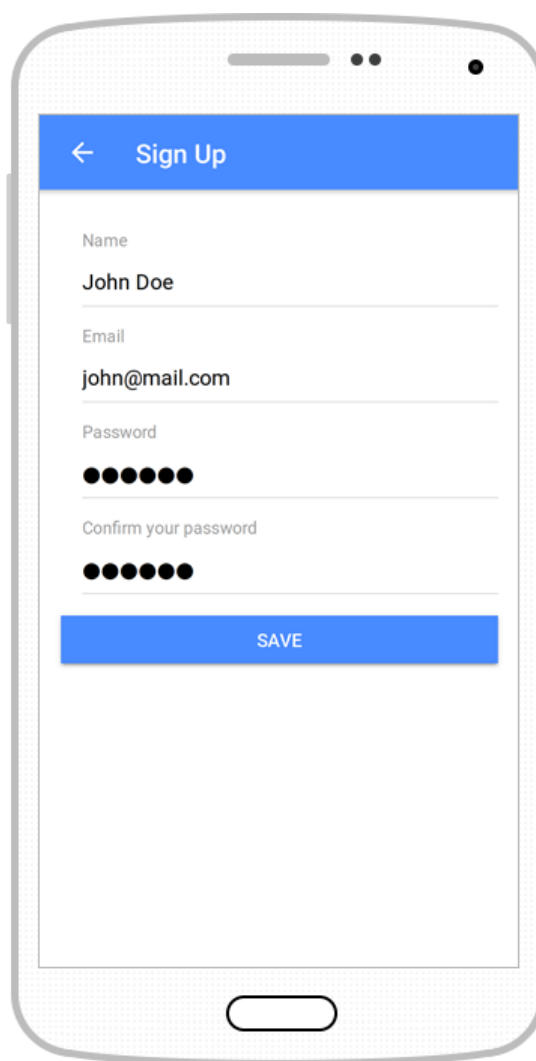


Figura 18: Ecrãs de demonstração: *Login*

Como não possuímos nenhum utilizador, deveremos efetuar o registo.



The image shows a smartphone screen with a 'Sign Up' form. The form has a blue header bar with a back arrow and the text 'Sign Up'. Below the header, there are four input fields: 'Name' with the value 'John Doe', 'Email' with the value 'john@mail.com', 'Password' with masked characters, and 'Confirm your password' with masked characters. A blue 'SAVE' button is at the bottom of the form.

← Sign Up

Name
John Doe

Email
john@mail.com

Password
●●●●●●

Confirm your password
●●●●●●

SAVE

Figura 19: Ecrãs de demonstração: Registo

Após o registo somos apresentados com o ecrã base da aplicação, neste caso com um aviso para o utilizador criar a sua primeira casa.

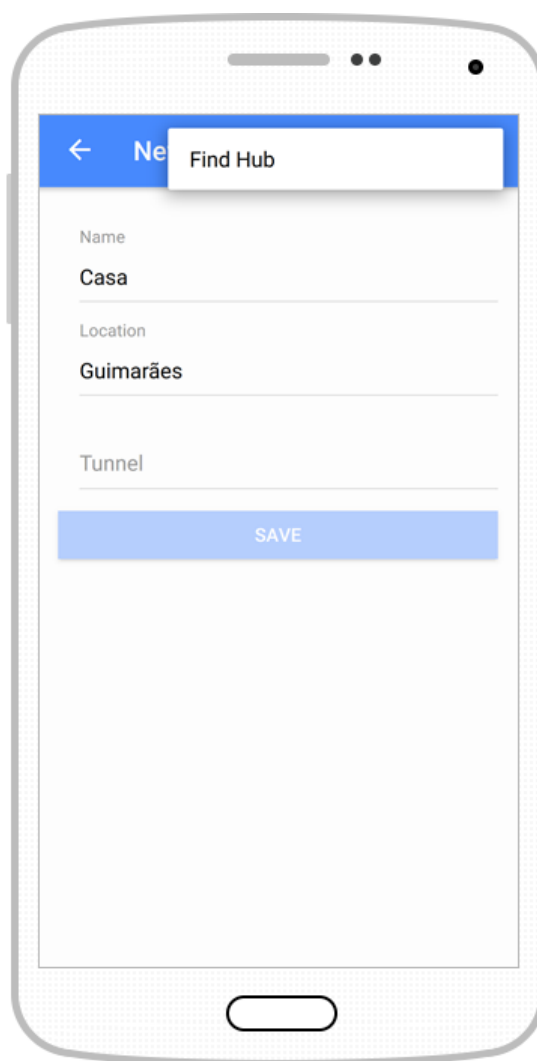


Figura 20: Ecrãs de demonstração: Criar casa

Para criar uma casa necessitamos do URL do *hub*, que pode ser descoberto com a ferramenta *"Find Hub"*. Este mecanismo já foi explicitado anteriormente, na detalhação do mecanismo de *tunneling*. Aqui simplesmente extraímos o *tunnel* disponibilizado pelo *hub*.

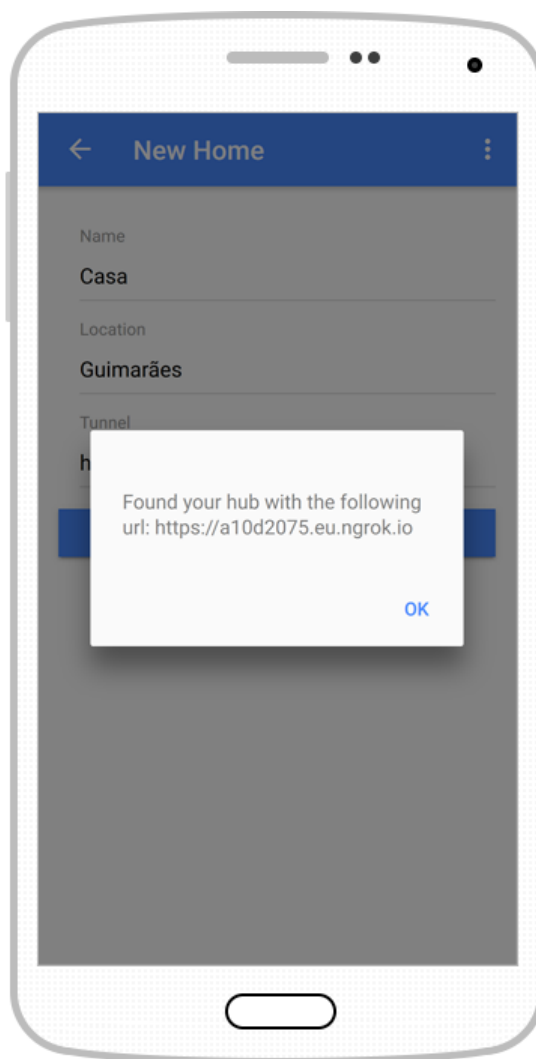


Figura 21: Ecrãs de demonstração: Descoberta do *hub*

Depois o nosso ecrã inicial mostra as casas do utilizador, uma vez que o *middleware* suporta várias casas, por exemplo, caso o utilizador tivesse uma casa de férias poderia colocá-la aqui também.

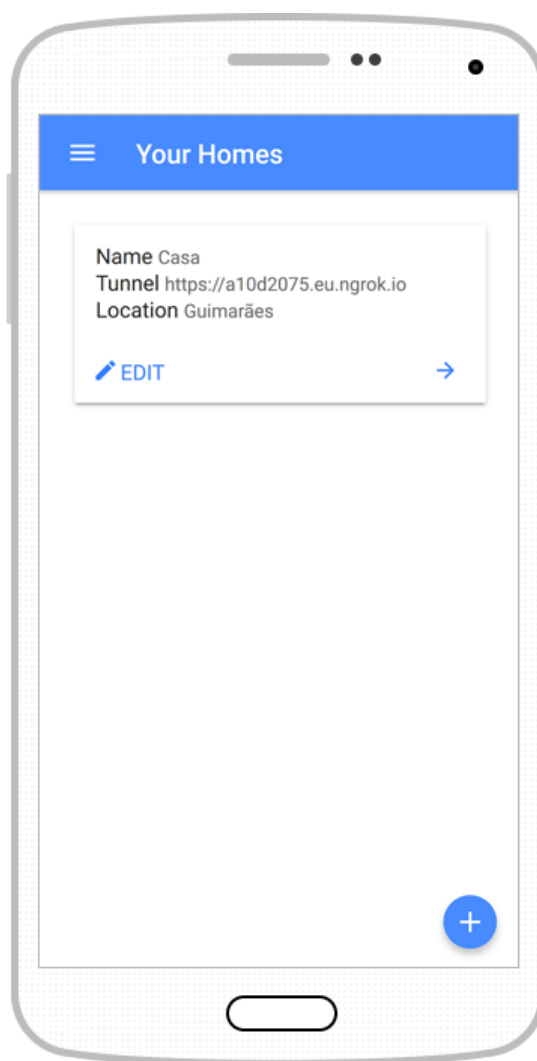


Figura 22: Ecrãs de demonstração: Lista de Casas

Clicando na casa que acabou de ser criada, ficamos perante o ecrã base da casa, possuindo 3 separadores, *scenarios*, *things* e *tasks*. Por omissão, somos redirecionados para o separador *things*, onde podemos gerir os dispositivos que um utilizador tem.

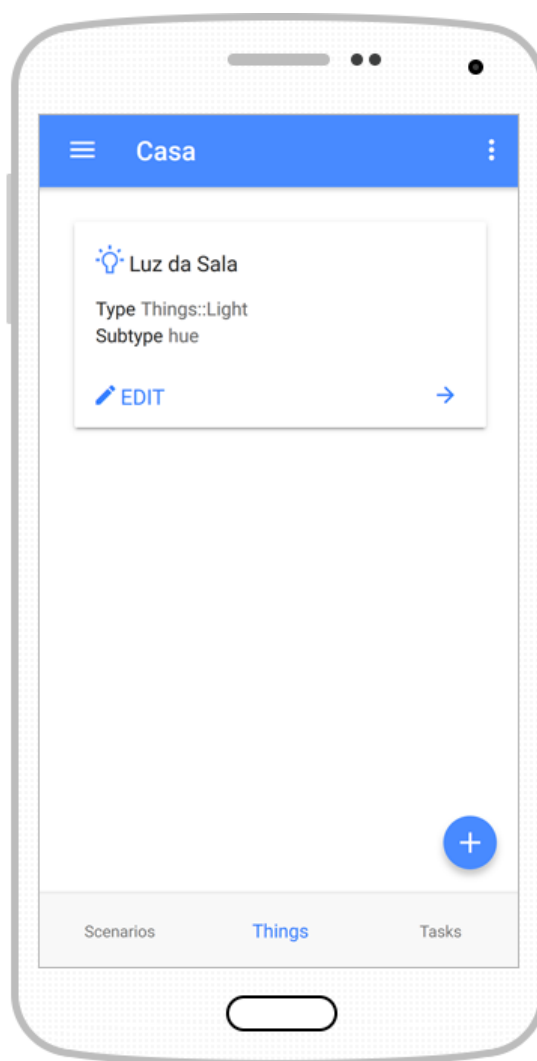


Figura 23: Ecrãs de demonstração: Lista de Dispositivos

De momento não possuímos qualquer tipo de dispositivo configurado, portanto vamos então iniciar esse processo. O objetivo irá passar por configurar uma lâmpada do emulador do *Philips Hue*, recorrendo ao serviço de descoberta.

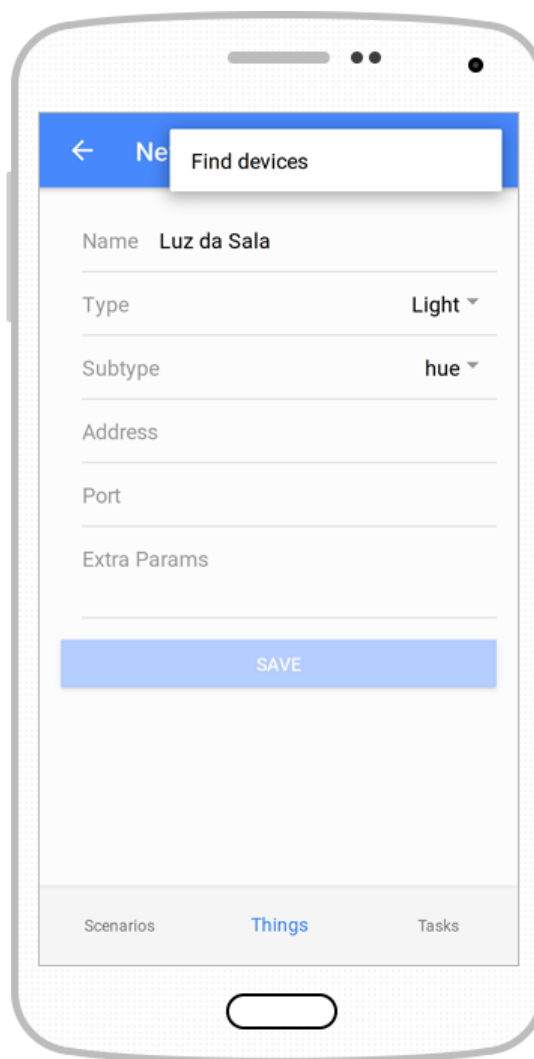


Figura 24: Ecrãs de demonstração: Criar Dispositivo

Aqui preenchemos os parâmetros de conexão ao dispositivo, deixando o endereço por preencher, uma vez que iremos recorrer ao serviço de descoberta do *hub* para o localizar. O serviço de descoberta irá preencher os campos necessários, sendo depois necessário preencher os *extra params* com o ID da lâmpada. Aqui conseguimos observar a modularidade do *middleware*, onde cada subtipo possui o seu próprio subconjunto de parâmetros de configuração, sendo possível associar com qualquer tipo de dispositivo desde que haja um serviço para o mesmo. Neste caso o *extra params* é apenas um *input* de JSON, não sendo ótimo em termos de usabilidade, servindo apenas para efeitos de demonstração. Num cenário ideal, cada subtipo teria um formulário diferente com os parâmetros necessários ao seu funcionamento.

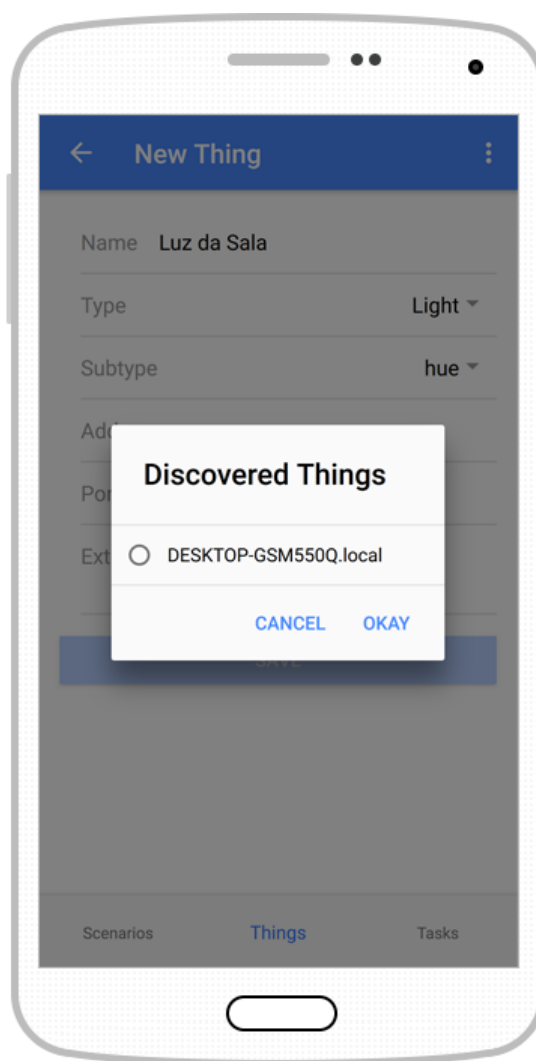


Figura 25: Ecrãs de demonstração: Descoberta de Dispositivo

Depois de executar o serviço de descoberta ficamos com o endereço do dispositivo, podendo assim gravar a configuração e ganhar acesso aos controlos do mesmo. Nos *extra params* inserimos o *light_id* da nossa lâmpada, normalmente é possível obter este ID através da aplicação oficial do *Phillips Hue*, no nosso caso o ID é obtido dentro do emulador.

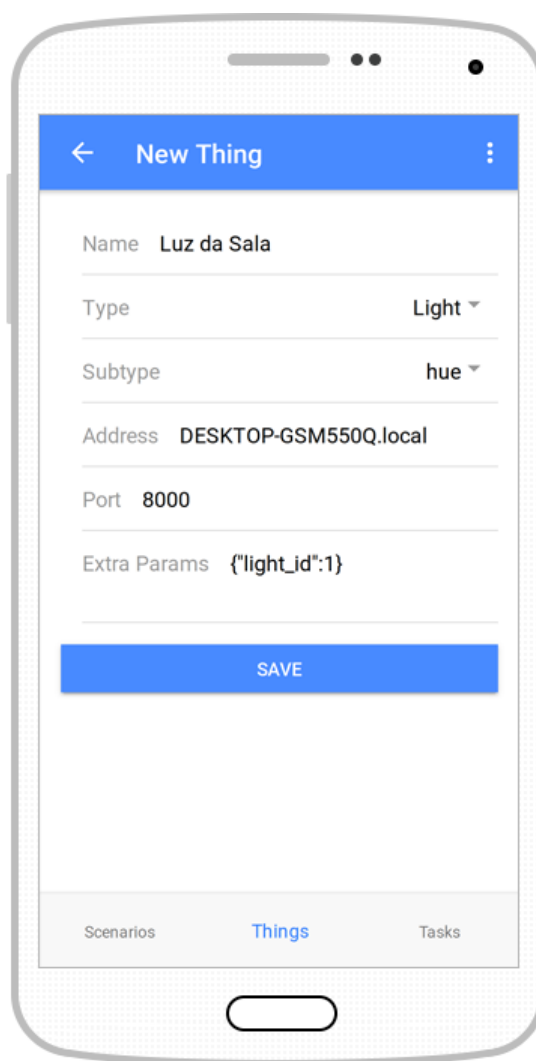


Figura 26: Ecrãs de demonstração: Descoberta de Dispositivo

De seguida podemos ver então o ecrã da lista de dispositivos já atualizado, assim como o ecrã base das informação e controlos de um dispositivo, onde podemos interagir com o mesmo.

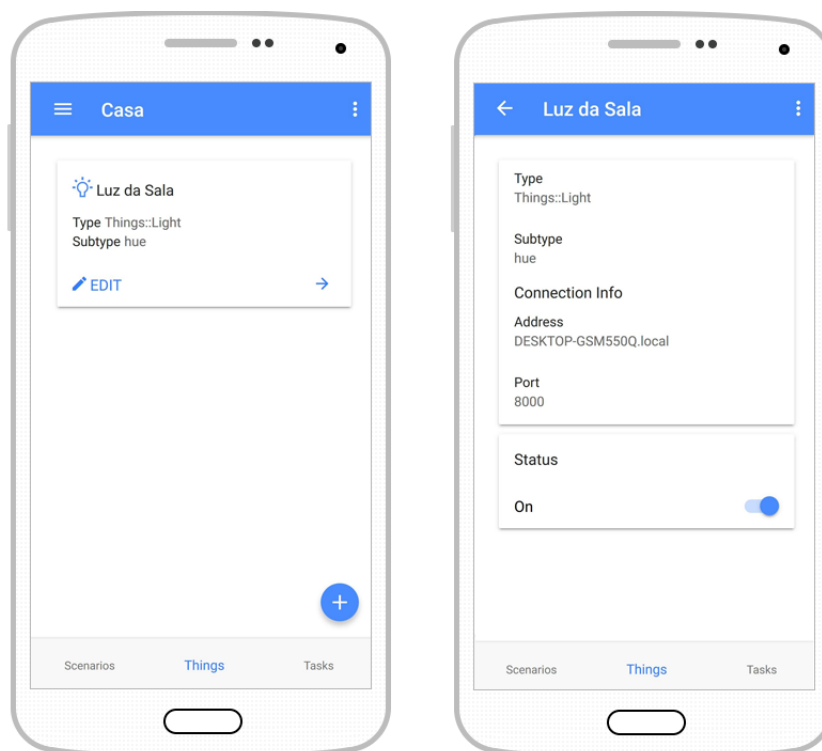


Figura 27: Ecrãs de demonstração: Lista de Dispositivos

Neste ecrã da direita podemos desligar ou ligar a luz, tendo um *feedback* real, ou seja, se por alguma razão a luz não acender a aplicação é notificada (ativação síncrona). Se a luz for desligada por outra aplicação também nos apercebemos disso, uma vez que este ecrã está ligado ao *middleware* através de *websockets*, recebendo atualizações em tempo real do estado do dispositivo.

Para esta demonstração configuramos mais 3 dispositivos, uma fechadura/porta, um termostato e um sensor de movimento, seguindo o mesmo método que foi mostrado anteriormente. Agora vamos mostrar as funcionalidades mais avançadas, nomeadamente os cenários e as tarefas.

No menu cenários podemos criar um cenário com um nome, um processo muito simples, e depois disto podemos então adicionar dispositivos e o seu estado desejado ao cenário.

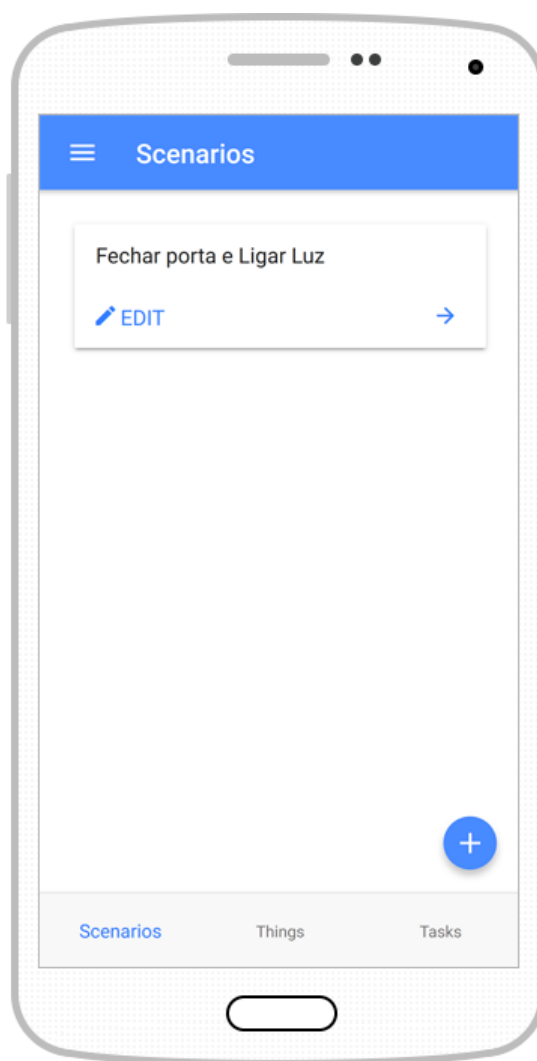


Figura 28: Ecrãs de demonstração: Ecrã de Cenário

De momento é impossível aplicar o cenário, dado que o mesmo não tem nenhum dispositivo associado, portanto para fazer jus ao nome do mesmo, iremos adicionar um dispositivo "porta" com a ação "fechar" e um dispositivo "luz" com a ação "ligar".

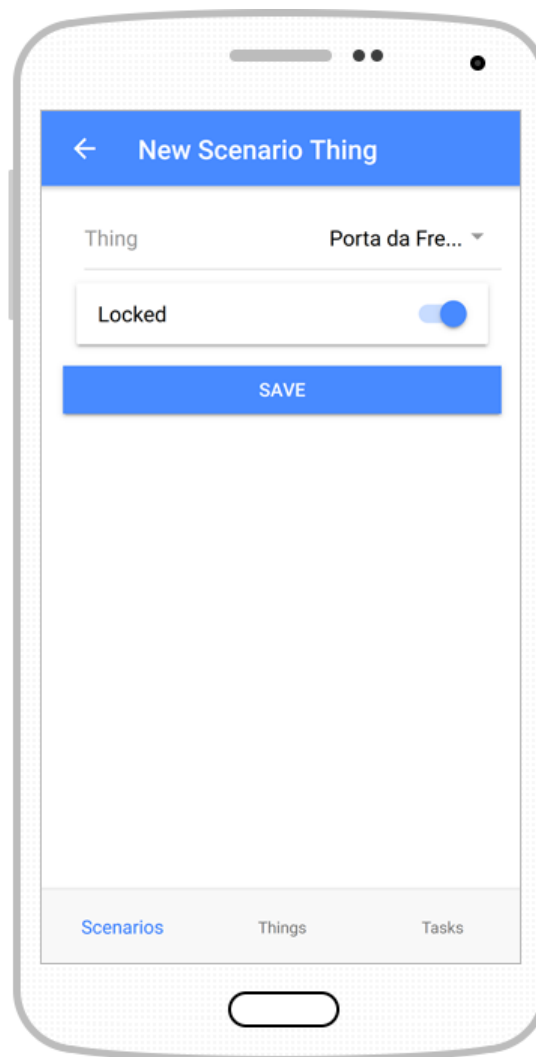


Figura 29: Ecrãs de demonstração: Criação de um *Scenario Thing*

Acima podemos ver o ecrã para adicionar *scenario things*, neste caso a adicionar a ação “fechar porta” ao cenário em questão.

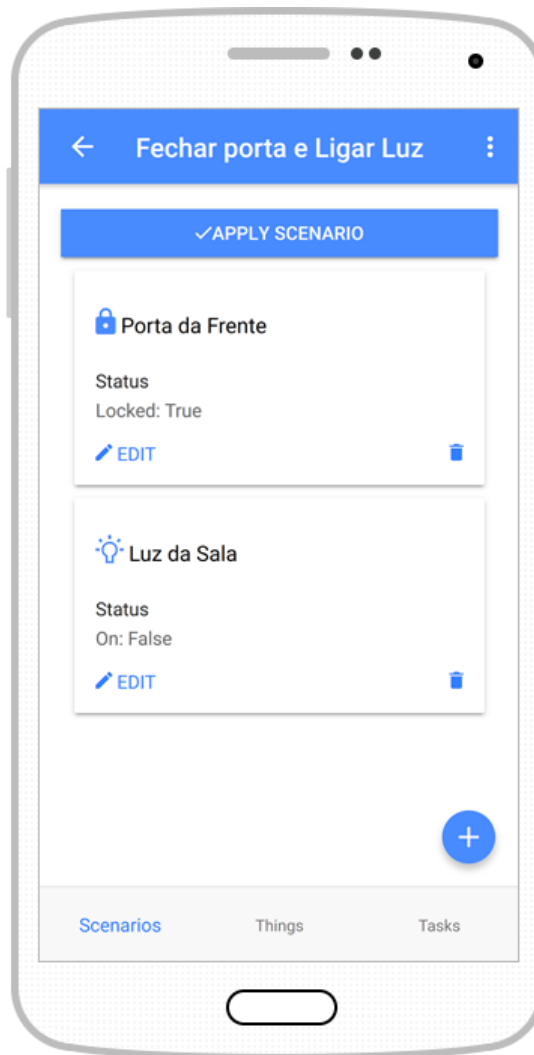


Figura 30: Ecrãs de demonstração: Ecrã de Cenário válido

Como podemos ver temos então as duas ações que desejávamos, podendo aplicar-las aos dispositivos em questão. Ao clicar naquele botão vamos interagir com o tal recurso *Scenario Applier*, como já vimos na arquitetura, que irá delegar ao *DelayedJob* a ativação dos dispositivos em série. A aplicação não recebe a confirmação da ativação dos mesmos como no caso do ecrã singular de um dispositivo. De seguida poderemos ver os *logs* do simulador, de notar que o primeiro corresponde ao nosso simulador enquanto que o segundo corresponde ao emulador do *Philips Hue*, sendo que o primeiro possui o *timestamp* em UTC e o segundo na hora do sistema (*timezone* de Lisboa).

```
[2017-10-06T15:45:13.603Z] RECEIVED PUT REQUEST: {"locked":true}
sex, 6 out 2017 16:45:10 [{"success":{"lights/1/state/on":false}}]
```

Figura 31: Ecrãs de demonstração: Log de aplicação de cenário

Por fim iremos abordar o separador das *tasks*, que inclui as funcionalidades das *timed* e *triggered tasks*. Como já vimos anteriormente, uma tarefa pode aplicar um cenário ou um único dispositivo com base num dado contexto, uma expressão *cron* ou o estado de um outro dispositivo.

Primeiramente iremos ver como se configura uma *timed task*. De notar a opção entre a escolha de um cenário ou um dispositivo como alvo, neste caso iremos escolher um dispositivo. A expressão *cron* irá ativar esta tarefa precisamente as 16:00, todos os dias, no *timezone Europe/Lisbon* (este último aspeto das expressões *cron* é exclusivo ao *DelayedCronJob*).

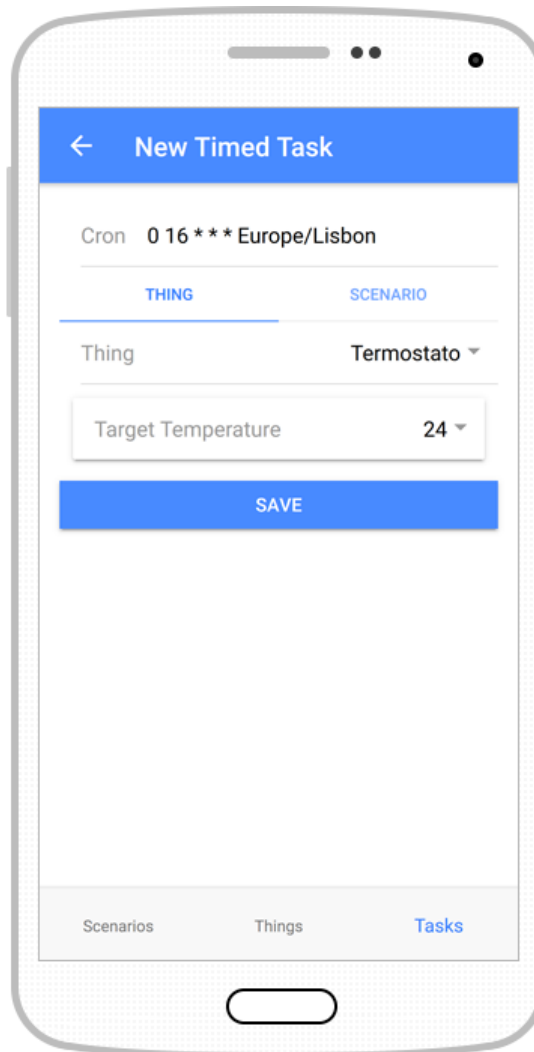
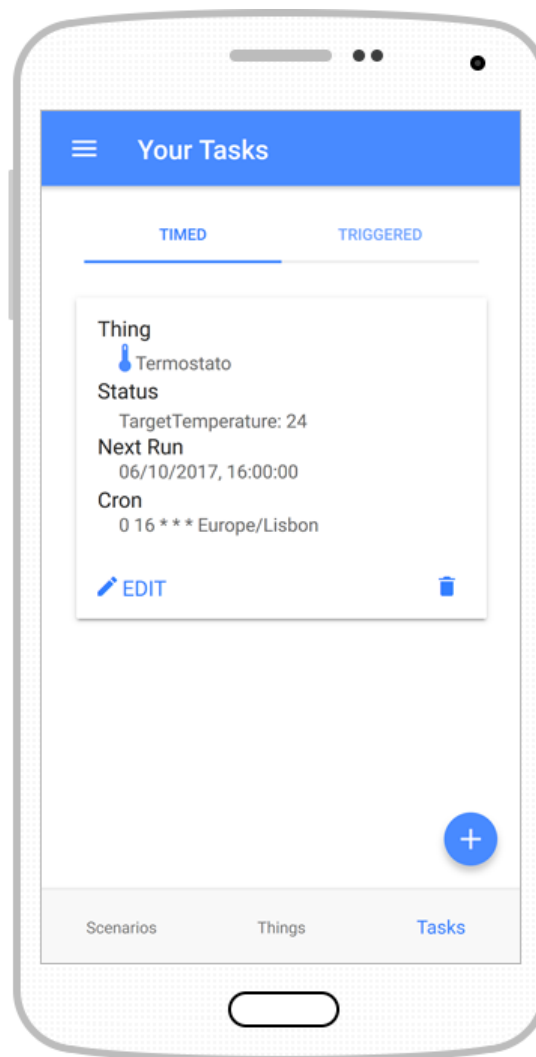


Figura 32: Ecrãs de demonstração: Criar *Timed Task*

Depois de criada a tarefa podemos ver o campo *Next Run* a mostrar a próxima execução da tarefa. Isto irá definir a temperatura do termostato a 24°C às 16:00.

Figura 33: Ecrãs de demonstração: Lista *Timed Tasks*

Para comprovar o funcionamento deste mecanismo, podemos observar os *logs* do simulador (tempo UTC), que recebe um pedido para alterar o estado segundos após as 16:00.

```
[2017-10-06T15:00:05.140Z] RECEIVED PUT REQUEST: {"target_temperature":24}
```

Figura 34: Ecrãs de demonstração: Logs *Timed Task*

Para concluir a demonstração, um exemplo de *triggered tasks*. Aqui vamos criar uma tarefa que ao detetar movimento num sensor irá ativar o cenário previamente definido, "fechar porta e ligar luz".

The screenshot shows a mobile application interface for creating a new triggered task. The title bar at the top is blue with a back arrow and the text "New Triggered Task". Below the title bar, there are two tabs: "THING" and "SCENARIO", with "SCENARIO" being the active tab. Under the "SCENARIO" tab, there is a "Scenario" label and a dropdown menu showing "Fechar port...". Below this, there is a section titled "Comparison Params". Inside this section, there are three fields: "Thing" with a dropdown showing "Sensor Exte...", "Comparator" with a dropdown showing "==", and "Status to Compare" with a text input containing the JSON object `{"movement": true}`. Below the "Status to Compare" field is a blue "SAVE" button. At the bottom of the screen, there is a navigation bar with three items: "Scenarios", "Things", and "Tasks", with "Tasks" being the active item.

Figura 35: Ecrãs de demonstração: Criar *Triggered Task*

Tal como o campo *extra params*, o campo *status to compare* também é um campo textual em formato JSON, útil para demonstração mas mau em termos de usabilidade. Num cenário ideal o utilizador escolheria os valores que queria comparar e utilizava inputs adequados para o efeito. Mesmo assim, a combinação entre o comparador e o estado traduz-se para a ação "movimento detetado no sensor", alcançando o nosso objetivo.

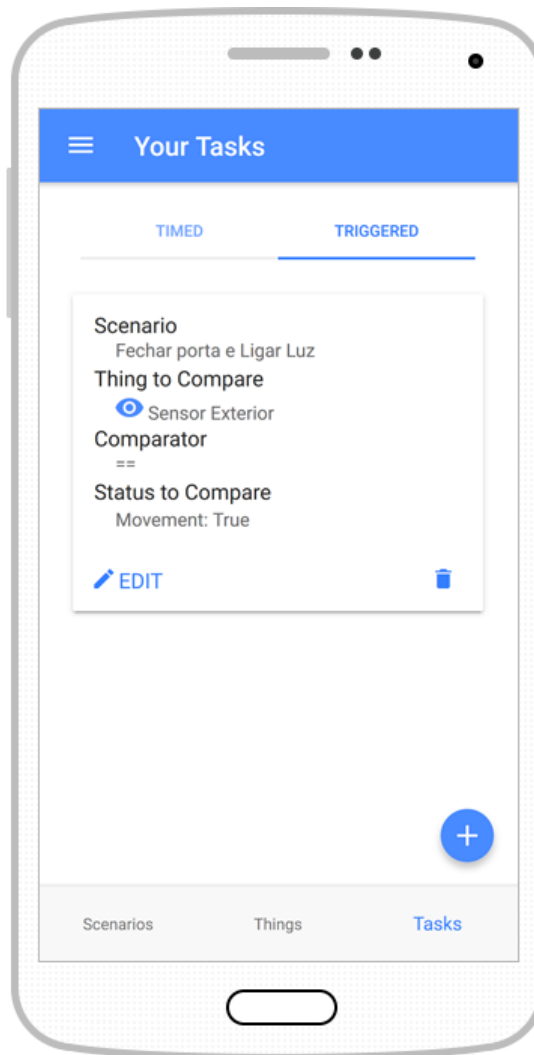


Figura 36: Ecrãs de demonstração: Lista *Triggered Tasks*

O sensor de movimento tem que ser ativado manualmente, através do simulador, e depois disto iremos visualizar os *logs* de ambos os dispositivos associados ao cenário para verificar a ativação dos mesmos, de modo a comprovar o funcionamento deste mecanismo.

```
[2017-10-06T15:54:00.434Z] RECEIVED PUT REQUEST: {"locked":true}
Sex, 6 out 2017 16:53:57 [{"success":{"lights/1/state/on":false}}]
```

Figura 37: Ecrãs de demonstração: Logs *Triggered Task*

Como podemos ver, ambos os dispositivos são ativados com uma diferença de segundos entre os mesmos, comprovando o funcionamento das *triggered tasks*

Para concluir, toda esta demonstração mostrou as funcionalidades do *middleware*, com algumas provas concretas mostradas através dos *logs*. Mesmo assim, além deste teste "ma-

nual", todos os módulos da aplicação possuem testes unitários, sendo que, estes testes "manuais" apenas têm como função analisar a interação entre os diversos módulos do sistema, o *hub*, os dispositivos e a *api*.

9. Conclusão

Chegando ao fim desta dissertação, conseguimos oferecer uma camada de *middleware* adequada aos requisitos explicitados, capaz de ligar diferentes tipos de dispositivos a aplicações clientes, possuindo mecanismos de automatização avançados, mantendo ao mesmo tempo uma arquitetura escalável e com boa manutenibilidade, assegurando a sua extensibilidade para suportar um número maior de dispositivos.

Conseguimos separar dois conceitos muito importantes, a camada de compatibilidade entre os vários dispositivos de múltiplos fabricantes, todos com APIs e protocolos variados, e, a camada de gestão conceptual, que é responsável pelo armazenamento dos dados dos dispositivos e pela automatização e gestão dos mesmos. Isto torna o sistema mais simples de manter, reduzindo muito a complexidade do mesmo sem passar por grandes esforços.

Trabalho Futuro

No entanto, este projeto possui várias áreas candidatas a melhorias, passando desde já pelo software de *tunneling*, o *ngrok*, que tem o tal problema dos URLs aleatórios, os sistemas de segurança e de autenticação, quer entre o *hub* e a *api*, quer entre as várias aplicações clientes e a *api*.

A partilha de casas entre utilizadores também seria algo a ver no futuro, recorrendo a um sistema de partilha avançado, com permissões e *roles*. A própria autenticação da API do *middleware* poderia ser melhorada, utilizando *OAuth* em vez do sistema de *tokens* simplista que se usa de momento. Utilizando *OAuth* teríamos mais controlo sobre os aspetos de autenticação, melhorando ainda a integração com outros sistemas, dado que este protocolo é bastante utilizado na web.

Além disto, a própria arquitetura do *hub* poderia ser alvo de alterações mais tarde, de momento cada fabricante, ou utilizadores com algum *background* técnico, teriam que eles próprios implementar as classes para os serviços dos dispositivos, essencialmente programando o *hub* para o efeito. Uma alternativa seria o desenvolvimento de uma DSL, que permitisse o

desenvolvimento rápido de uma camada de compatibilidade entre o *middleware* e um novo tipo de dispositivo.

Por fim, a própria performance do sistema poderia ser um fator de risco, uma vez que a API do *middleware* foi desenvolvida em *Ruby*, uma linguagem que é conhecida pelos seus ocasionais problemas de performance. Mais tarde poderia-se reescrever esse componente em *Elixir*, uma linguagem baseada em *Erlang*, ideal para ambientes distribuídos e de alta disponibilidade. Em *Elixir* teríamos acesso à framework *Phoenix*, muito semelhante ao *Rails*, podendo efetuar esta conversão de linguagem com menos esforço. Outra alternativa seria apenas reescrever partes vitais do sistema, nomeadamente o processamento de tarefas e o módulo de *websockets*, que são os componentes mais propensos a problemas de performance. Apesar de tudo isto, toda a arquitetura é capaz de escalar horizontalmente.

Lições e Experiência

Desta dissertação foram adquiridas várias competências quer ao nível de desenho de sistemas de grande escala, a nível de complexidade, de interoperabilidade serviços, devido à interação *hub-api*. Também se entrou em contacto com inúmeras tecnologias, *Ruby*, *Java* e *Typescript* são algumas delas, assim como várias técnicas de desenvolvimento, como *model driven development* e meta-programação.

Foi melhorado o conhecimento sobre redes, infraestruturas e disponibilidade de serviço. Também se aprofundou o conhecimento sobre vários protocolos como o HTTP e CoAP, assim como tópicos importantes relativos à IoT, nomeadamente a segurança, talvez um dos mais importantes.

10. Bibliografia

- [App16] Apple. *Apple's Homekit*. 2016. URL: <http://www.apple.com/ios/home>.
- [AR16] M. Asadullah e A. Raza. «An overview of home automation systems». Em: *2016 2nd International Conference on Robotics and Artificial Intelligence (ICRAI)*. Nov. de 2016, pp. 27–31. DOI: [10.1109/ICRAI.2016.7791223](https://doi.org/10.1109/ICRAI.2016.7791223).
- [Azn+16] M. N. Azni et al. «Home automation system with android application». Em: *2016 3rd International Conference on Electronic Design (ICED)*. Ago. de 2016, pp. 299–303. DOI: [10.1109/ICED.2016.7804656](https://doi.org/10.1109/ICED.2016.7804656).
- [Bry14] Erik Brynjolfsson. *The second machine age : work, progress, and prosperity in a time of brilliant technologies*. New York: W.W. Norton & Company, 2014. ISBN: 978-0-393-35064-7.
- [BZ16] P. Bellavista e A. Zanni. «Towards better scalability for IoT-cloud interactions via combined exploitation of MQTT and CoAP». Em: *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*. Set. de 2016, pp. 1–6. DOI: [10.1109/RTSI.2016.7740614](https://doi.org/10.1109/RTSI.2016.7740614).
- [Cha+14] H. Chang et al. «Bringing the cloud to the edge». Em: *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Abr. de 2014, pp. 346–351. DOI: [10.1109/INFCOMW.2014.6849256](https://doi.org/10.1109/INFCOMW.2014.6849256).
- [DRD16] S. Dey, A. Roy e S. Das. «Home automation using Internet of Thing». Em: *2016 IEEE 7th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. Out. de 2016, pp. 1–6. DOI: [10.1109/UEMCON.2016.7777826](https://doi.org/10.1109/UEMCON.2016.7777826).
- [Ecl16] Eclipse. *Eclipse Smarthome, A flexible framework for the smarthome*. 2016. URL: <http://www.eclipse.org/smarthome>.
- [Gam95] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. ISBN: 978-0201633610.
- [Goo16a] Android Google. *Android Things, Android OS for embedded systems*. 2016. URL: <https://developer.android.com/things/index.html>.
- [Goo16b] Brillo Google. *Google Brillo, old Google project for IoT*. 2016. URL: <https://developers.google.com/brillo>.

- [Hic15] Ian Hickson. *Server-Sent Events*. W3C Recommendation. W3C, fev. de 2015. URL: <http://www.w3.org/TR/2015/REC-eventsourcing-20150203>.
- [IEE16] IEEE. *IEEE IoT Developer Survey*. 2016. URL: <https://goo.gl/WCmjQM>.
- [Mye02] Judith Myerson. *The complete book of middleware*. Boca Raton, Fla: Auerbach, 2002. ISBN: 978-0849312724.
- [Nak+16] Y. Nakamura et al. «Design and Implementation of Middleware for IoT Devices toward Real-Time Flow Processing». Em: *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. Jun. de 2016, pp. 162–167. DOI: [10.1109/ICDCSW.2016.37](https://doi.org/10.1109/ICDCSW.2016.37).
- [Net16] Netatmo. *Netatmo weather station*. 2016. URL: <https://www.netatmo.com/product/weather/weatherstation>.
- [Ngu+16] A. H. H. Ngu et al. «IoT Middleware: A Survey on Issues and Enabling technologies». Em: *IEEE Internet of Things Journal* PP.99 (2016), pp. 1–1. ISSN: 2327-4662. DOI: [10.1109/JIOT.2016.2615180](https://doi.org/10.1109/JIOT.2016.2615180).
- [NTD16] S. Nastic, H. L. Truong e S. Dustdar. «A Middleware Infrastructure for Utility-Based Provisioning of IoT Cloud Systems». Em: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. Out. de 2016, pp. 28–40. DOI: [10.1109/SEC.2016.35](https://doi.org/10.1109/SEC.2016.35).
- [Phi16] Philips. *Philips Hue smart light bulbs*. 2016. URL: <http://www2.meethue.com>.
- [Ric13] Leonard Richardson. *RESTful Web APIs*. Sebastopol, Calif: O'Reilly, 2013. ISBN: 978-1449358068.
- [Sta06] Thomas Stahl. *Model-driven software development : technology, engineering, management*. Chichester, England Hoboken, NJ: J. Wiley & Sons, 2006. ISBN: 978-0-470-02570-3.
- [Ste16] SteveyO. *Philips Hue emulator by SteveyO*. 2016. URL: <http://steveyo.github.io/Hue-Emulator>.