



**Desenvolvimento da gestão de manutenção da empresa ACATEL, Acabamentos Têxteis S.A.**

José Carlos Rodrigues Arantes,

n.º 8636

Realizado sob orientação do Professor Michael Matias

Ano letivo 2019/2020

Licenciatura de Engenharia de Sistemas Informáticos.

Escola Superior de Tecnologia do Instituto Politécnico do Cávado e Ave.

**Identificação do aluno**

José Carlos Rodrigues Arantes

Número 8636, regime diurno

Ocupação: Estudante e trabalhador.

Licenciatura em Engenharia de Sistemas Informáticos.

**Orientação académica**

Professor Michael Matias

**Informação sobre a empresa de acolhimento**

ACATEL - Acabamentos Têxteis S.A.

Rua do Barreiro, 310 - Gilmonde

4754-909 Barcelos

Orientação: Engenheiro José Carlos Catalo

“*Os analfabetos do próximo século não são aqueles que não sabem ler ou escrever, mas aqueles que se recusam a aprender, reaprender e voltar a aprender*.” Alvin Tofler (1928-2016)

### RESUMO

Toda a nossa vida está em constante mudança e também evolução. Sem excepção disso está o mundo das engenharias e tecnologias, com novas ferramentas em constante mudança.

Reflete-se também esta evolução em empresas de diversos sectores. Parte desta evolução traz consigo a informatização de sistemas de sistemas obsoletos e que não cumprem normas ou regras de aceitação por parte das associações acreditadoras, sendo estas normas necessárias para manter os clientes, que neste caso é essencial à empresa ACATEL, a organização de acolhimento.

A área da programação WEB, é apontada como prática fundamental para a preparação de profissionais para o mercado de trabalho.

Nesse contexto, este projeto incide na criação de uma página WEB, disponível no dominio de rede da empresa, em tempo real, utilizada para facilitar a consulta, mas também a alertar a necessidade de reparações dos equipamentos (máquinas de laboração ou de ajuda à laboração).

Os objetivos estruturantes do presente relatório residem numa fase introdutória, onde é exposta a situação atual e obsoleta, que condiciona a aprovação de acreditações de futuras vistorias na empresa. De seguida, o objetivo detalha o processo de desenvolvilmento ao ambiente apresentado e a sua análise de requisitos.

(Nota: ver palavras chave-tema, e palavras chave-tecnologia)

### ABSTRACT

### AGRADECIMENTOS

A realização deste projeto final do curso de licenciatura, que tem como objetivo a obtenção do grau de Engenheiro de Sistemas Informáticos, contou como seria de esperar de grande apoio e suporte, sem os quais tornaria esta tarefa ainda mais difícil.

Em primeiro lugar, ao Professor Michael Matias por ter aceite o convite para me orientar nesta jornada, pelo seu conhecimento transmitido, por todas as horas que estivemos em video-conferência (em tempos de pandémicos), de modo a conseguir resolver problemas que iam surgindo. De um modo especial, à sua incasável disponibilidade de modo a que nunca me deixasse ficar mal. Ao Professor Michael, o meu grande apreço.

Ao Engenheiro José Carlos Catalo, pessoa que me acolheu na empresa durante a realização deste projeto, e e ajuda ao mostrar o esquema de funcionamento.

Ao amigo e colega de trabalho, José Carreira, trabalhador na empresa de acolhimento, pela força transmitida.

A todos os que foram meus professores e que me encaminharam nesta jornada, em especial ao professor Luís Ferreira por toda a ajuda enquanto seu aluno.

A todos os meus amigos e família.

E por fim, especificamente aos meus pais, pelo grande apoio que me deram, por todo o sacríficio que também eles passaram para que eu possa ter concluído todo este percurso desde o início da licenciatura, até à fase final.

A todos, saúde e muito obrigado.

**Indíce**

[RESUMO 5](#_Toc60090013)

[ABSTRACT 6](#_Toc60090014)

[AGRADECIMENTOS 7](#_Toc60090015)

[ÍNDICE DE FIGURAS 11](#_Toc60090016)

[ÍNDICE DE TABELAS 12](#_Toc60090017)

[Glossário 13](#_Toc60090018)

[Siglas e Acrónimos 15](#_Toc60090019)

[1. Introdução 17](#_Toc60090020)

[1.1. Contexto e objetivos 17](#_Toc60090021)

[1.2. Estado da arte 18](#_Toc60090022)

[1.3. Estrutura do documento 19](#_Toc60090023)

[2. Análise 20](#_Toc60090024)

[2.1 Análise ao problema 21](#_Toc60090025)

[2.1.1 Requisitos funcionais 22](#_Toc60090026)

[2.1.2 Requisitos não funcionais 24](#_Toc60090027)

[2.1.3 Diagramas de casos de uso 26](#_Toc60090028)

[2.2 Plataformas de desenvolvimento 30](#_Toc60090029)

[3. Base de dados 31](#_Toc60090030)

[4. Servidor (API) 34](#_Toc60090031)

[4.1 REST API (Representational State Transfer) 35](#_Toc60090032)

[4.2 Entity Framework 37](#_Toc60090033)

[4.3 CORS 37](#_Toc60090034)

[4.4 DTO (Data Transfer Object) 38](#_Toc60090035)

[4.5 ASP.NET Web API 39](#_Toc60090036)

[4.6 Controladores 40](#_Toc60090037)

[5. Angular (frontend) 43](#_Toc60090038)

[5.1 Introdução à ferramenta Angular 43](#_Toc60090039)

[5.2 Arquitetura 44](#_Toc60090040)

[5.2.1 Injeção de dependência (Dependency injection) 44](#_Toc60090041)

[5.2.3 Diretivas e componentes 46](#_Toc60090042)

[5.2.4 Serviços (*services*) 47](#_Toc60090043)

[5.3 TypeScript 48](#_Toc60090044)

[6. Implementação 49](#_Toc60090045)

[6.1 Introdução 49](#_Toc60090046)

[6.2 As aplicações 49](#_Toc60090047)

[6.2.1 Estrutura da solução *backend* 49](#_Toc60090048)

[6.2.2 Estrutura da solução *frontend* 52](#_Toc60090049)

[6.3.1 Boas práticas no frontend 54](#_Toc60090050)

[6.3.2 Boas práticas no backend 55](#_Toc60090051)

[6.4 Serviço para pedidos REST API autenticados 57](#_Toc60090052)

[6.4.1 Autenticação no servidor 58](#_Toc60090053)

[6.5 Registo de um utilizador 59](#_Toc60090054)

[6.5.1 Registo de um utilizador (backend) 59](#_Toc60090055)

[6.5.2 Registo de um utilizador (services, models – frontend) 59](#_Toc60090056)

### ÍNDICE DE FIGURAS

[Figura 1 - Diagrama de blocos geral 16](file:///C:\Users\José%20Carlos\Documents\GitHub\ProjetoGestaoManuntencao\Relatorio.docx#_Toc59386905)

[Figura 2 - Processo têxtil 18](#_Toc59386906)

[Figura 3 - Diagrama caso de uso 23](file:///C:\Users\José%20Carlos\Documents\GitHub\ProjetoGestaoManuntencao\Relatorio.docx#_Toc59386907)

### ÍNDICE DE TABELAS

[Tabela 1- Requisitos funcionais 20](#_Toc59385011)

[Tabela 2 - Requisitos não funcionais 22](#_Toc59385012)

### Glossário

**Software** -na gíria comum, traduz-se como sendo um programa que traz a experiência de interagir com equipamentos. Em termos técnicos, é usado para designar a parte lógica que fornece diretivas para hardwares presentes num equipamento.

**Hash** - ou função Hash, são usadas para fins de segurança para criptografar por exemplo palavras-passe.

**Framework** - é uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica. Em termos mais simples, permite reutilizar código sem necessidade de ter de reescrever.

**Front-end** - parte da aplicação que interage diretamente com o utilizador e o servidor.

**Back-end** - parte da aplicação que interage entre a base de dados e o front-end.

**Token** - é um sistema de transferência de dados que pode ser enviado via URL, POST ou num cabeçalho HTTP (header) de maneira "segura", essa informação é assinada digitalmente.

**Header** - traduzido em português como cabeçalho, é uma componente usada nos pedidos e respostas HTTP, onde definem os parâmetros destas mesmas transações.

**Authorize** - filtro de autorização no back-end. Caso implementado num método, só permite execução desse mesmo método caso o utilizador esteja autenticado.

**Role** – propriedade utilizada no back-end, de forma a definir que tipo de utilizador da aplicação terá acesso a determinado método/componente.

**Web Browser** - é um programa que habilita utilizadores a interagirem com documentos HTML hospedados num servidor de rede.

### Siglas e Acrónimos

**API** Application Programming Interface (Interface de Programação de Aplicação – que estabelece um conjunto rotinas e padrões de um software).

**HTTP** HyperText Transfer Protocol (Protocolo de Transferência de Hipertexto).

**SQL** Structured Query Language (Linguagem de Consulta Estruturada).

**.NET** Plataforma única para desenvolvimento e execução de sistemas e aplicações.

**LINQ** Language Integrated Query (Consulta integrada à linguagem) - componente da framework .NET que adiciona funcionalidades de consulta em algumas linguagens de programação .NET.

**RF** Requisito funcional - representa o que o software faz, em termos de tarefas e serviços.

**RNF** Requisito não funcional - relacionado ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenção e tecnologias envolvidas.

**EXE** é uma extensão de ficheiros que podem ser executados por computadores que têm por base sistema operativo Windows.

**VPN** Virtual Private Network (Rede privada virtual), é uma rede de comunicações privada construída sobre uma rede de comunicações pública.

**ERP** Enterprise Resource Planning (planeamento de recurso corporativo), é um sistema de gestão empresarial.

**DTO** Data Transfer Object, um padrão usado em linguagens orientadas a objetos para o transporte de dados entre diferentes componentes de um sistema.

**SPA** Simple Page Application (Aplicação de página única)

**JSON** JavaScript Object Notation - formato de troca de dados entre sistemas.

**XML** Extensible Markup Language - linguagem de marcação que define um conjunto de regras para codificação.

**CORS** Cross-Origin Resource Sharing

**ORM** Object-relational mapping

# **Introdução**

Este documento descreve o processo de implementação de uma aplicação desenvolvida no âmbito do estágio do Curso de Licenciatura em Engenharia de Sistemas Informáticos.

Este estágio e projeto, tem como objetivo a aplicação prática dos vários conhecimentos que foram adquiridos ao longo dos 3 anos de licenciatura num contexto de trabalho real.

O projeto foi desenvolvido com vista a resolver um problema de registo e consulta de parâmetros de avarias, e suporte de equipamentos de produção da organização de acolhimento, que até aqui era feito em suporte de papel.

O projeto aqui apresentado interliga várias àreas de programação, desde base de dados, API e WEB, sendo posto logo de parte o uso de uma aplicação do tipo .exe, por várias razões: um sistema WEB pode ser acessado em qualquer equipamento (tablet ou computador) conectado ao à rede interna da empresa. A segunda razão, prende-se pelo facto da programação WEB ser bastante importante no mercado de trabalho.

## **Contexto e objetivos**

A [Acatel - Acabamentos Têxteis S.A.](https://acatel.pt/pt/) (empresa de acolhimento), presente no mercado desde 1985, presta serviços de tinturaria, mercerização, tingimento à peça, estamparia digital e tradicional e acabamentos de malhas e tecidos.

Como trabalhador nesta empresa e ao mesmo tempo estudante no IPCA, sabendo a admnistração que me encontrava naquele momento a estudar, foi feita a proposta da realização deste projeto.

A existência de variadas soluções para este tipo de projeto que existem no mercado real, não significam que estas se adequem à empresa em questão, pois estamos perante um projeto criado de raíz, de modo a satisfazer os pormenores pedidos pela organização.

Assim, por ter sido criado de raíz, exigiu um esforço adicional para que o código desenvolvido seja validado e testado, de modo a não permitir falhas durante a sua produção em tempo real.

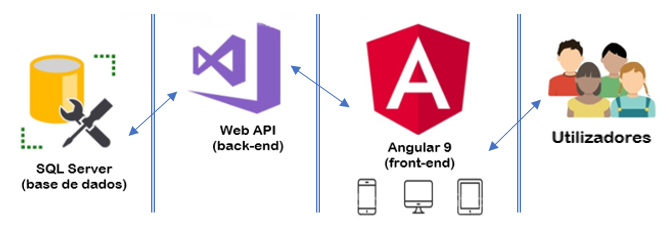
A seguinte figura representa o diagrama de blocos que apresenta uma visão geral sobre todo o sistema desenvolvido.

Figura 1 - Diagrama de blocos geral

Os utilizadores interagem com uma aplicação WEB, podendo ser utilizada num equipamento que suporte um browser compatível com as funcionalidades de Angular, com recurso a HTTP, sendo que esta utiliza a API (.NET Framework), que por sua vez estabelece conexão com a base de dados (SQL Server).

## **Estado da arte**

Esclarecidos os objetivos para este projeto, foi hora de analisar o que seria necessário para criar este trabalho. Como dito no capítulo anterior, o objetivo seria criar um sistema adequado às especificações da admnistração, num ambiente fechado (servidor interno da empresa de acolhimento, no domínio interno, afastado da rede exterior), necessário estar disponível em todos os computadores ou equipamentos móveis da empresa (telemóveis ou tablets que suportem um browser atualizado).

A empresa em questão está dividida em várias secções. Somando todo o número equipamentos fixos (computadores), são várias dezenas. A isto, soma-se o número de equipamentos móveis (tablets, telemóveis).

As aplicações WEB (dependendo sempre da àrea a que se aplicam) estão em constante evolução, contudo, aplicações desktop e WEB continuarão a coexistir por muito tempo, cada uma com suas vantagens e desvantagens. Neste caso, era desavantajoso instalar uma aplicação .*exe* em todos os computadores da empresa, estes são também limitados pelo *hardware* no qual são executados, devem ser desenvolvidos num sistema operacional específico.

Por outro lado, uma aplicação WEB apenas necessita de um browser que suporte as frameworks usadas no desenvolvimento de código, podendo até mesmo, ser acessíveis fora do dominio interno da empresa, através de VPN.

***Ver a evolução entre .exe app vs web app e também os ERP disponíveis.***

## **Estrutura do documento**

Este documento está organizado em (***MUDAR ISTO DE ACORDO COM O QUE TEM AGORA***)capítulos principais além do presente capitulo de introdução.

No capítulo inicial, é feita a análise ao problema, as ferramentas utilizadas, onde inclui uma análise dos requisitos, funcionalidades necessárias para a aplicação e apresentados os diagramas de caso de uso. Em seguida são expostas as tecnologias e metodologias utilizadas no processo de implementação do sistema.

No capítulo de desenvolvimento estão expostos todos os sistemas desenvolvidos. Este capitulo está dividido em subcapítulos que falam sobre cada um dos sistemas desenvolvidos durante o projeto, nomeadamente a base de dados, a API (back-end) e a interface com o utilizador (front-end), assim como os resultados conseguidos.

Por fim são apresentados possíveis melhorias na aplicação, e conclusões finais do projeto.

# **Análise**

Para o desenvolvimento da aplicação é necessário identificar as necessidades da empresa, ou seja, conhecer o modus *operandi* da organização de acolhimento, para perceber a necessidade de que dados recolher e como os registar.

A Acatel Acabamentos Têxteis S.A., é uma empresa têxtil que opera na área dos acabamentos e beneficiação têxtil. A empresa presta serviços a outras empresas na área de beneficiamento têxtil, mais concretamente em tinturaria, tingimento à peça, estamparia digital, estamparia convencional e acabamentos de malhas e tecidos.

O desenvolvimento da sua atividade industrial está dependente de vários sistemas informáticos, contudo, nenhum neste momento que seja específico para a manutenção dos equipamentos da empresa. A seguinte figura mostra a fase do processo têxtil onde a empresa de acolhimento se insere (destaque a vermelho).

Figura 2 - Processo têxtil

## **2.1 Análise ao problema**

A função da empresa de acolhimento contém diferentes secções. Diferentes secções, contêm diferentes grupos de máquinas, e para cada grupo de máquina, há máquinas com funcionalidades diferentes e distintas.

A realização deste trabalho, tem por função organizar a manutenção de uma empresa que contém mais de 150 (cinto e cinquenta) máquinas, de diferentes tipo, tamanhos e funcionalidades.

O esquecimento, ou o não cumprimento da manutenção destes equipamentos pode comprometer a reputação desta empresa, visto que as encomendas têm prazos de entrega, que necessitam de ser cumpridos. O não cumprimento destes prazos devido à falta de organização da manutenção, ou da não realização de tarefas de manutenção preventiva por parte de um funcionário, de modo a prevenir avarias, pode comprometer a relação entre um cliente e a empresa de acolhimento. Por isso, torna-se necessário ter um registo.

As tarefas de um funcionário pertencente à equipa de manutenção (serralheiro ou eletricista), dividem-se em duas categorias: preventivas ou curativas.

As manutenções preventivas, são aquelas que são programadas pelo Eng.º da equipa de manutenção, necessárias para o bom funcionamento destes equipamentos, de forma a minimizar o desgaste dos mesmos.   
Por outro lado, as manutenções curativas não são programadas, e acontecem quando a manutenção preventiva pode ter falhado por algum motivo alheio, e é necessária a sua reparação, podendo esta ser notificada, avaliada e realizada.

Tornar este processo mais dinâmico através desta aplicação, melhora a classificação das normas que a empresa necessitaria de cumprir (ISO 13372:2012)[[1]](#footnote-2)*,* requisitadas por auditorias externas.

## **Requisitos funcionais**

O levantamento de requisitos foi efetuado com atenção às funcionalidades e detalhes que seriam necessários/fundamentais a serem implementados na aplicação. Os mesmos foram separados por requisitos funcionais e requisitos não-funcionais.

Tabela 1- Requisitos funcionais

|  |  |  |  |
| --- | --- | --- | --- |
| **Requisitos Funcionais** | | | |
| **ID** | **Tag** | **Descrição** | **Prioridade** |
| RF01.1 |  | O sistema deverá permitir registar novos utilizadores:   * Login; * Password; * Nome completo; * Número de trabalhador; * Secção de trabalho; * Tipo de perfil (administrador, técnico, funcionário comum); | **SHOULD** |
| RF01.2 | Gestão de utilizadores | O sistema poderá permitir inativar utilizadores. | **MUST** |
| RF01.3 |  | O sistema poderá permitir associar o utilizador a um perfil de acesso. | **SHOULD** |
| RF01.4 |  | O sistema deverá conter pelo menos estes três perfis:   * Admistrador; * Técnico de reparação; * Funcionário (chefe de secção de cada turno, para fazer relatar as anomalias); | **SHOULD** |
| RF02.1 | Gestão de Equipamentos | O sistema deverá permitir adicionar equipamentos, com os seguintes campos obrigatórios:   * Código interno e descrição; * Grupo de máquina pertencente; * Marca e número de série; * Tipo; * Data de compra e desativação; * Estado do equipamento (ativo/desativo); | **SHOULD** |
| RF02.2 |  | O sistema deverá permitir inativar equipamentos (equipamentos que deixam de operar, vendidos ou obsoletos). | **MUST** |
| RF03.1 |  | O sistema deverá permitir a criação de pedidos de assistência do tipo curativa ou preventiva. | **SHOULD** |
| RF03.3 |  | Os pedidos e intervenções deverão ter os seguintes estados:   * Sem intervenção; * Programado/(a); * Fechado; | **SHOULD** |
| RF03.3 | Gestão de pedidos de assistência | O sistema deverá para o tipo curativo conter os seguintes campos:   * Hora do pedido (introduzida automaticamente pelo servidor); * Hora da intervenção (início e fim, introduzida pelo utilizador); * Tempo de intervenção por técnico (num pedido podem ocorrer várias várias intervenções); * Utilizador que faz o pedido (introduzido automaticamente pelo servidor, baseado no login); * Equipamento; * Descrição da avaria; * Técnico que fez a intervenção; * Descrição da intervenção realizada; | **SHOULD** |
| RF03.4 |  | O sistema deverá para o tipo de assistência preventiva conter os seguintes campos:   * Hora do pedido (introduzida automaticamente pelo servidor); * Hora da intervenção (início e fim, introduzida pelo utilizador); * Horas de intervenção por técnico ( num pedido podem ocorrer várias vári * as intervenções ); * Utilizador que faz o pedido (introduzido automaticamente pelo servidor, baseado no login); * Equipamento; * Técnico que fez a intervenção; * Equipamento; * Descrição da intervenção realizada; * Limite de intervenção (caso seja preventiva, e programada pelo Eng.º de manutenção); | **SHOULD** |
| RF04.1 | Gestão de Planos | O sistema deverá permitir associar um técnico a um grupo de equipamentos para manutenções preventivas. | **SHOULD** |
| RF04.2 |  | O sistema deverá alertar o técnico de manutenções programadas a executar. | **MUST** |
| RF04.3 | Gestão de planos  (continuação) | O Eng.º de manutenção, emite um pedido de manutenção preventiva, com uma data limite de manutenção para grupo de máquinas (plano de prevenção). | **SHOULD** |
| RF04.4 |  | O sistema deverá permitir criar um plano de manutenção anual. | **MUST** |
| RF05.1 |  | O sistema deverá permitir a consulta de manutenção de um determinado grupo de máquinas, alojado, em formato *pdf*. | **SHOULD** |
| RF05.2 | Consulta de registos | O sistema deverá permitir consultar intervenções anteriormente realizadas. | **SHOULD** |
| RF05.3 |  | O sistema deverá permitir a consulta do relatório de manutenção anual. | **COULD** |

## **2.1.2 Requisitos não funcionais**

|  |  |  |  |
| --- | --- | --- | --- |
| **Requisitos Não Funcionais** | | | |
| **ID** | **Requisito** | **Descrição** | **Prioridade** |
| RNF01 | Desempenho | O sistema deverá apresentar bom desempenho: respostas rápidas após clique e a carregar conteúdos. | **MUST** |
| RNF02 | Usabilidade | O sistema deverá apresentar uma interface simples e intuitivo, facilitando a sua utilização. | **MUST** |
| RNF04 | Interoperabilidade | O sistema deverá comunicar com SQL Server. | **MUST** |
| RNF05 | Confiabilidade | O sistema deverá estar disponível durante todo o tempo de produção. | **SHOULD** |
| RNF06 | Idiomas | O sistema deverá suportar o idioma português. | **SHOULD** |
| RNF07 | Segurança | O sistema deverá oferecer segurança. Nem todas as pessoas com acesso a um computador, podem ter acesso à aplicação. | **SHOULD** |

Tabela 2 - Requisitos não funcionais

AINDA PODEM SER ACRESCENTADOS ALGUNS REQUISITOS NÃO FUNCIONAIS.

## **2.1.3 Diagramas de casos de uso**

**Diagrama n.º 1**

A seguinte figura representa o diagrama de casos de uso de forma simplificada, referente a um pedido curativo ou preventivo simples.

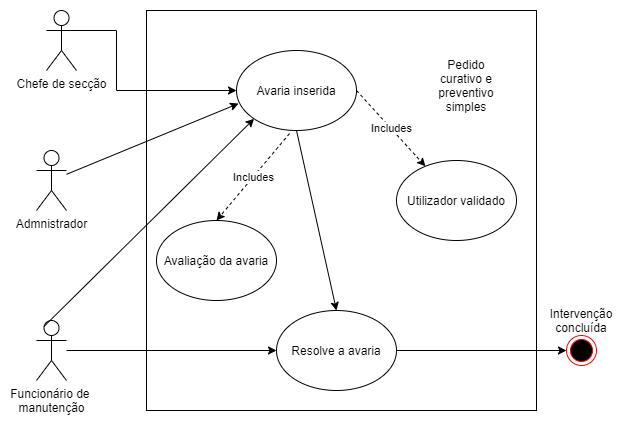


Figura 3 – Diagrama de caso de uso n.º 1

O diagrama demonstra as ações de uma parte da aplicação, nomeadamente operações de manutenção simples curativas ou preventivas.

Qualquer tipo de utilizador desta aplicação (desde que esteja registado), poderá inserir no sistema um alerta de avaria. Por outro lado, para resolver a avaria, apenas funcionários de manutenção têm acesso à componente de inserção de intervenções, de modo a deixar a avaria como resolvida.

**Tabelas descritivas do diagrama de casos de uso n.º 1**

Cada caso de uso tem uma descrição de funcionalidade que será construída no sistema proposto. De seguida são apresentadas as tabelas descritivas de cada caso de uso apresentado na figura anterior.

**AVARIA INSERIDA**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | Uma entidade insere um pedido (registo) de avaria. |
| **PÓS-CONDIÇÃO** | N/A |

**UTILIZADOR VALIDADO**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | O sistema verifica se esta entidade tem permissão para inserir um pedido (registo). |
| **PÓS-CONDIÇÃO** | Caso não tenha permissão, o pedido não é registado. |

**AVALIAÇÃO DA AVARIA**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | Uma entidade da manutenção verifica o tipo de problema ocorrido. |
| **PÓS-CONDIÇÃO** | N/A |

**RESOLVE A AVARIA**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | A intervenção foi feita e registada. |
| **PÓS-CONDIÇÃO** | O pedido é dado como resolvido no sistema. |

**Diagrama n.º 2**

A seguinte figura representa o diagrama de casos de uso de forma simplificada, referente a um pedido de manutenção preventiva programada.

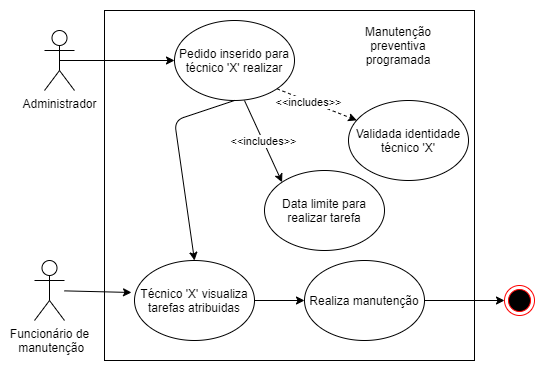


Figura 4 - Diagrama de caso de uso n.º 2

O diagrama demonstra as ações de uma outra parte da aplicação, nomeadamente operações de manutenção que o admnistrador solicita a determinado funcionário de manutenção. Este tipo de manutenção, é destinado para manuntenção que engloba grupos de máquina com similaridades entre os equipamentos pertencentes ao mesmo.

Esta componente, é mais dedicada ao funcionário de manutenção e admnistrador como partes fulcrais da manutenção da empresa, sendo que um funcionário de manutenção pode e deve ver as tarefas que o admnistrador lhe atribuiu, para que seja realizada até uma data limite, também esta definida pelo admnistrador.

**Tabelas descritivas do diagrama de casos de uso n.º 2**

Cada caso de uso tem uma descrição de funcionalidade que será construída no sistema proposto. De seguida são apresentadas as tabelas descritivas de cada caso de uso apresentado na figura anterior.

**PEDIDO INSERIDO PARA TÉCNICO ‘X’ REALIZAR**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | O admnistrador associa a necessidade de manutenção preventiva programada de um grupo de máquinas, a um funcionário de manutenção |
| **PÓS-CONDIÇÃO** | N/A |

**VALIDADA IDENTIDADE TÉCNICO ‘X’**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | O sistema verifica a identidade do técnico. |
| **PÓS-CONDIÇÃO** | N/A |

**DATA LIMITE PARA REALIZAR TAREFA**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | Cada pedido atribuído ao técnico ‘X’, tem uma data limite até que a manutenção seja realizada. |
| **PÓS-CONDIÇÃO** | Se ultrapassada a data limite, o técnico entra em incumprimento. |

**TÉCNICO ‘X’ VISUALIZA TAREFAS ATRIBUÍDAS**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | O técnico ‘X’ tem uma àrea dedicada a si, onde pode consultar tarefas que lhe tenham sido atribuídas pelo admnistrador da manutenção. |
| **PÓS-CONDIÇÃO** | N/A |

**REALIZA MANUTENÇÃO**

|  |  |
| --- | --- |
| **PRÉ-CONDIÇÃO** | N/A |
| **DESCRIÇÃO** | O técnico ‘X’ realiza tarefas que lhe foram atribuidas. |
| **PÓS-CONDIÇÃO** | Os pedidos são dados como ‘fechados’. Técnico ‘X’ deixa ser receber alerta de pedidos em espera. |

## **2.2 Plataformas de desenvolvimento**

* O Microsoft Visual Studio é um pacote de programas do Microsoft para desenvolvimento de software dedicado especialmente ao .NET Framework e às linguagens de programação Visual Basic, C, C++, C#, e J#.



Figura 5 - Logótipo Visual Studio

* O Microsoft SQL Server, é um sistema gerenciador de bancos de dados relacional, desenvolvido em parte pela Microsoft, compatível também com Visual Studio.



Figura 6 - Logótipo SQL Server

* O Visual Studio Code, é um editor de código criado pela Microsoft, para Windows, Linux e macOS, sendo ideal para trabalhar em linguagens tais como TypeScript (TS), JavaScript (JS), HTML e CSS.



Figura 7 - Logótipo Visual Studio Code

* Angular é uma framework para aplicações WEB e front-end, baseado em TypeScript, reescrita de AngularJS, criada pela Google.



Figura 8 - Logótipo Angular

# **Base de dados**

A base de dados disponível no servidor interno com licença, trata-se de SQL Server, tratando-se de um modelo de dados relacional. É também um bom candidato para usar em contínuo com a ferramenta Visual Studio, sem necessitar de drivers adicionais para estabelecer conexão entre a base de dados e a API a consumir.

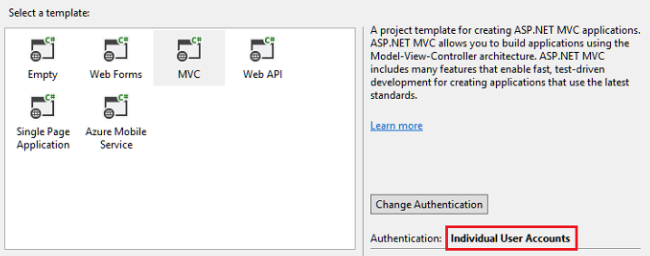
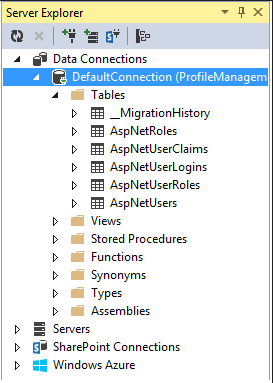
Por definição, um projeto .NET Framework escolhido com *individual user accounts authentication,* como o nome indica tem incorporado um sistema de autenticação de utilizadores. Assim, foi escolhido um projeto MVC (model, view, control). Nem tudo neste projeto criado por definição será usado, em substituição da *view,* o usado é a framework Angular (com recurso ao Visual Studio Code).

Figura 9 - Criação da API (Visual Studio)

Como referido anteriormente, por definição, este projeto acarta consigo o sistema de autenticação de contas de utilizador individuais, podendo guardar consigo dados de um utilizador, tais como email, password (encriptada sobre *hash)*.

Mais tarde neste relatório, é exposta a manipulação da classe *Register*, uma classe dedicada à criação de um novo utilizador na aplicação, para conter novos dados para além do *username* e da *password,* tais como o nome do utilizador, o tipo de permissão, entre outros.

Na figura abaixo, é mostrada a base de dados relacional para autenticação de utilizadores, contudo, não sendo necessárias todas as tabelas, esta foi adaptada de forma a coincidir com as necessidades da aplicação.

Nesta base de dados criada por definição, as tabelas que têm aqui interesse são as seguintes:

* + - * ***AspNetUsers***: tabela que contém as informações pessoais do utilizador.
      * ***AspNetRoles****:* tabela que contém o *ID* do tipo de permissão e a descrição dessa permissão.
      * ***AspNetUserRoles****:* tabela que serve de interface entre as duas tabelas descritas anteriomente, tendo a informação da *role* e o *ID* do utilizador.

Figura 10 - Base de dados do projeto

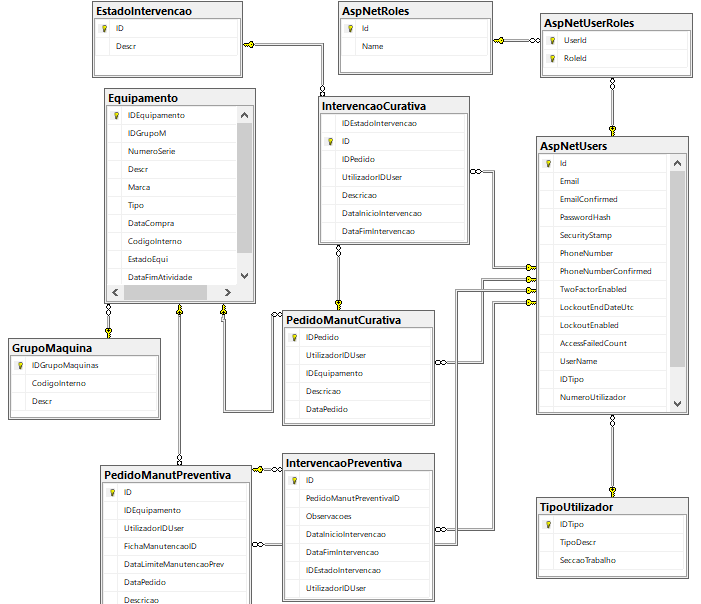
A figura seguinte, evidencia o transporte da base de dados anteriormente citada, para a base de dados do modelo de negócio.

Figura 11 - Base de dados da aplicação

A base de dados criada de forma a servir a aplicação, para além do que já foi citado no parágrafo anterior, contém essencialmente:

* Equipamentos: são todas as máquinas presentes na empresa que têm o propósito de servir o bom funcionamento da mesma.
* Grupo de máquina: destinada a conter a informação relativa aos diferentes grupos de equipamentos da empresa, podendo haver diferentes grupos de máquina na mesma secção da empresa.
* Pedidos de manutenção curativa: destinada a conter a informação dos pedidos curativos referentes aos equipamentos.
* Pedidos de manutenção preventiva: destinada a conter a informação dos pedidos preventivos referentes aos equipamentos.
* Intervenções curativas: destinada a conter a informação relativa a um pedido curativo. Um pedido curativo pode ter mais que uma intervenção.
* Intervenções preventivas: destinada a conter informação relativa a um pedido preventivo. Um pedido preventivo pode ter mais que uma intervenção.
* Estado de intervenção: destinada a conter informação relativa ao estado em que se encontra um pedido (curativo ou preventivo). Neste caso, o estado assenta na intervenção, contudo, no *front-end,* é feita uma transformação que, baseada na última intervenção de um pedido, é mostrado o estado em que se encontra o pedido (se contém alguma intervenção, se este já foi analisado mas ainda não reparado, ou se o problema já se encontra solucionado).
* Tipo de utilizador: destinado a conter informação relativa a um determinado utilizador, tal como a secção da empresa em que trabalha, e a sua função na empresa.

# **Servidor (API)**

No início, houve uma decisão de construir uma aplicação de página única (SPA, *simple page application*) na framework Angular para criar uma aplicação mais rápida e fácil de usar. O SPA move grande parte da apresentação e da lógica de negócios para o *browser*, mas o servidor é contudo muito importante. Existem áreas em que o servidor deve estar envolvido para atingir o nível de segurança ou consistência de dados desejados

Cada aplicação trabalha com os dados que cria, modifica ou transforma e geralmente é necessário persistir esses dados para uso posterior. Além disso, nem todas as operações devem ser executadas num *browser* e, portanto, é necessário um servidor que possa lidar com as operações que a aplicação de front-end não pode realizar. Estas são as seguintes áreas:

* Persistência de dados
* Autenticação e autorização
* Operações

Toda a aplicação é baseada num modelo de arquitetura cliente-servidor (*client-server*). Este modelo, descreve como um servidor fornece recursos e serviços a um ou mais clientes. Assim sendo, um servidor fornece recursos para dispositivos clientes, como computadores *desktop* ou *smartphones*.

A maioria dos servidores tem uma relação um-para-muitos com os clientes, o que significa que um único servidor pode fornecer recursos para vários clientes ao mesmo tempo (Christensson, Client-Server Model Definition 2016).

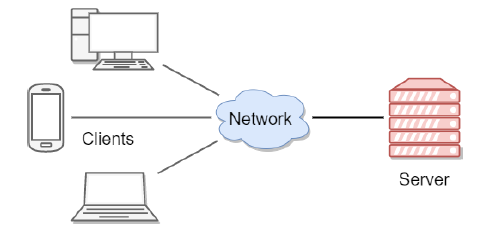
**A seguinte figura, ilustra um modelo *client-server.*

Figura 12 - Modelo cliente-servidor

O SPA é executado em dispositivos clientes para fornecer uma interface de utilizador, e a framework ASP.NET é executada no servidor para servir recursos e serviços. A comunicação entre o cliente e o servidor é realizada recorrendo ao protocolo HTTP (Hypertext Transfer Protocol), e a arquitetura REST (Representational State Transfer) é implementada para seguir as melhores práticas. A estrutura ASP.NET e a arquitetura REST serão descritas nos capítulos a seguir.

## **4.1 REST API (Representational State Transfer)**

*Representational State Transfer* (REST) é um estilo de arquitetura baseado num conjunto de princípios que descrevem como os recursos de rede são definidos e endereçados. Esses princípios foram descritos no ano 2000 por Roy Fielding como parte de sua tese de doutorado. REST é uma alternativa ao *Simple Object Access Protocol* (SOAP).

Quando falamos sobre APIs REST, palavras como URL são usadas com frequência. O *Uniform Resource Locator* (URL) é um identificador de localização deum controlador. Todos os controladores podem ser localizados por URL e deve ficar claro no URL qual controlador é endereçado.

As aplicações baseadas na arquitetura REST são frequentemente chamadas de RESTful. O principal objetivo REST API é ter acesso e manipular dados de maneira uniforme. Operações como GET, PUT, UPDATE ou DELETE, são frequentemente realizadas, e o nome do método HTTP define o tipo da operação desejada. A seguinte tabela mostra a relação entre o nome do método HTTP e a operação a ser realizada.

Tabela 3 - Métodos HTTP e as suas funções

|  |  |
| --- | --- |
| Método HTTP | Operação a ser executada |
| GET | Solicita a representação de um método/objeto específico. |
| POST | Utilizado para submissão de novos dados. |
| PUT | Substitui determinados dados, por novos dados inseridos. |
| DELETE | Apaga a informação relativa a determinado objecto. |

Em RESTful, o consumidor do serviço usa o método HTTP para definir a operação que deve ser executada. O provedor de serviços trata e processa a solicitação e envia a resposta ao consumidor para informá-lo do resultado da solicitação. Onde os códigos do estado da resposta são usados ​​para esse propósito. Os mais comuns são os seguintes:

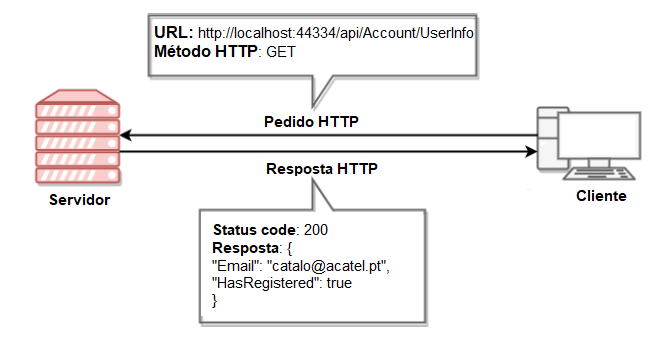
* **200 Ok** - A resposta HTTP foi concluída com sucesso.
* **201 Created** - A requisição foi bem sucedida e um novo recurso foi criado como resultado. Esta é uma tipica resposta enviada após um pedido POST.
* **400 Bad Request** - indica que o pedido do cliente não foi processado, porque o servidor não conseguiu compreender o que o cliente pede.
* **401 Unauthorized** - o cliente tem de efetuar login para obter a resposta solicitada.
* **403 Forbidden** - O cliente não tem direito de acesso ao conteúdo,assim o servidor rejeita dar a resposta. Diferente do código 401, aqui a identidade do cliente é conhecida.
* **404 Not found** - indica que o recurso pedido não está disponível.
* **500 Internal server error** - indica que a solicitação é válida, mas ocorreu um erro inesperado ao processar o pedido.

Figura 13 - Comunicação RESTful entre servidor e cliente (aplicação)

## **4.2 Entity Framework**

ADO.NET é um conjunto de classes que expõe serviços de acesso a dados para programadores da framework .NET. Este, fornece um conjunto de componentes para a criação de aplicativos de compartilhamento de dados distribuídos. É parte integrante do .NET Framework (ADO.NET 2017).

Por outro lado, Entity Framework é um ORM (Object-relational mapping), um conjunto de tecnologias ADO.NET que oferece suporte ao desenvolvimento de aplicações de software orientadas a objetos. Permite consultar entidades e relacionamentos no modelo mas também converter operações em comandos específicos da fonte de dados.

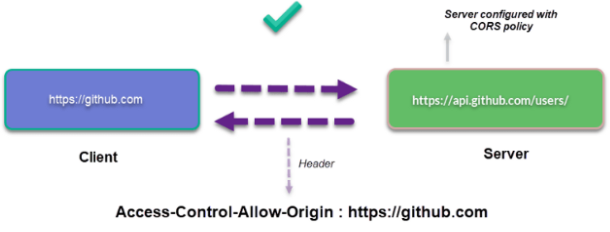
É uma ferramenta que facilita a gestão da base de dados automaticamente, sem necessidade de recorrer a outras, tais como por exemplo, *stored procedures*.

## **4.3 CORS**

*Cross-origin Resource Sharing* (CORS), é um mecanismo utilizado pelos browsers para partilhar recursos entre diferentes origens. Este faz parte dos cabeçalhos HTTP para informar um browser se determinado recurso pode ou não ser utilizado.

Figura 14 - Exemplo simplificado usando CORS

**4.8** / **5** ( **18** votes )

Ao enviar um pedido para uma origem diferente, o *browser* utiliza um *header* específico enviado pelo servidor chamado *Access-Control-Allow-Origin*. A partir desse e mais alguns *headers,* é determinado se esse mesmo recurso pode ser carregado ou não.

## **4.4 DTO (Data Transfer Object)**

Data Transfer Object, é um padrão de projetos, utilizados para transferir ou manipular dados entre subsistemas de um software.

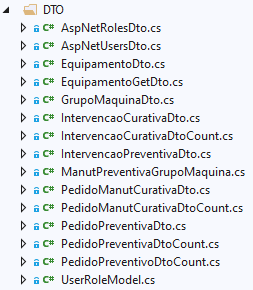
****A ideia consiste em agrupar um conjunto de atributos numa classe simples de forma a otimizar ou manipular a comunicação na forma pretendida.

Figura 15 - Pasta DTOs do backend

A imagem abaixo mostra parte de um método, que recebe informação do frontend. Neste caso trata-se de um pedido de inserção de dados, um pedido de manutenção curativa. A ajuda que um DTO nos pode oferecer neste caso, é a manipulação dos dados a nível do servidor, podendo fazer transformações ou manipulações dessa informação a ser inserida, antes que a mesma chegue ao método que insere a informação na base de dados. Trata-se neste caso da passagem de uma classe DTO (PedidoManutCurativaDto) para a classe propriamente criada pelo ADO.NET Entity Model (PedidoManutCurativa).

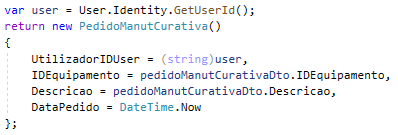


Figura 16 - Método da classe PedidoManutCurativaController

## **4.5 ASP.NET Web API**

A framework ASP.NET Web API é uma estrutura que facilita a construção de serviços HTTP, onde alcança uma ampla gama de clientes, incluindo navegadores (*browsers*) *e* dispositivos móveis. É uma plataforma ideal para construir aplicações RESTful.

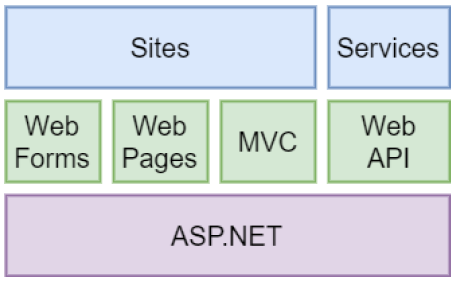


Figura 17 - Arquitetura da framework ASP.NET

API significa interface de programação de aplicação e é um conjunto de comandos, funções, protocolos e objetos que os programadores usam para criar softwares ou interagir com um sistema externo. Fornece a programadores, comandos padrão para executar operações comuns, para que não seja necessário escrever o código desde o início (Christensson, API Definition 2016). WEB API, como o nome sugere, é uma API na *web* e pode ser acessada por protocolo HTTP.

A ASP.NET Web API é uma estrutura extensível para a construção de serviços baseados em HTTP, que podem ser disponibilizados em diferentes aplicações em diferentes plataformas, tais como *web*, *windows*, mobile, entre outros. Funciona de certa forma como ASP.NET MVC, exceto que envia dados como uma resposta em vez de uma exibição HTML (a não utilização da camada *view* referente à arquitetura MVC). É como um serviço *web* ou serviço *WCF*, mas com a exceção que só oferece suporte ao protocolo HTTP.



Figura 18 - Arquitetura da aplicação Web API

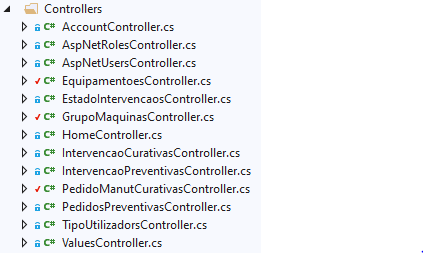
A framework ASP.NET Web API consiste em vários blocos, como definições de rota (routing), classes de controladores, modelo de objetos e camada de acesso a dados. Na figura 14 pode-se ver como as partes da aplicação funcionam entre si. Esta arquitetura de alto nível não é fixa e pode variar de aplicação para aplicação, dependendo do efeito pretendido.

## **4.6 Controladores**

Quando a framework Web API recebe um pedido, esta encaminha o pedido para o controlador e ação (método) específicos. Um controlador é um objeto que lida com os pedidos HTTP e envia uma resposta de volta ao cliente. O controlador pode ser criado na pasta Controllers ou em qualquer outra pasta na pasta raiz do projeto. O nome de uma classe de controlador deve terminar em "Controller" e derivado da classe System.Web.Http.ApiController. Todos os métodos públicos do controlador são chamados de métodos de ação.

Para informar o cliente sobre o resultado de um pedido, uma resposta adequada deve ser enviada. Esta framework oferece diferentes resultados de ação integrados, que ajudam a seguir o estilo de arquitetura REST.

A seguinte imagem ilustra o conjunto de controladores presentes no projeto.

Por definição, o nosso projeto cria alguns dos controladores, sendo estes essenciais, nomeadamente o *AccountController*. Trata-se de um controlador responsável por informações relativas a utilizadores, tais como por exemplo, o registo de um novo utilizador.

Os restantes controladores foram criados de forma a responder aos pedidos HTTP, mediante a necessidade de serem utilizados.

Figura 19 - Controllers da API

A seguinte figura, retrata o exemplo de uma ação que pretende inserir informação no modelo, ou seja, na base de dados. No momento em que esta é invocada, há toda uma verificação e validação daquilo que se pretende inserir, de forma a evitar erros que possam colocar em causa a confiabilidade da aplicação, nomeadamente o vulgo *crash,* ou simples duplicação de dados previamente introduzidos. Parte desta validação pode ainda ser reforçada no *frontend*.

Figura 20 - Exemplo de um método de um controller (PedidoManutCurativasController.cs)

O modelo pode ser definido como aquilo que contém ou representa os dados com os quais a aplicação vai tratar.

A Web API do ASP.NET pode automaticamente serializar o modelo para *JSON*, *XML* (ou outro formato) e de seguida, gravar os dados serializados no corpo da mensagem de resposta HTTP. O cliente define o formato que deseja, no cabeçalho do pedido HTTP.

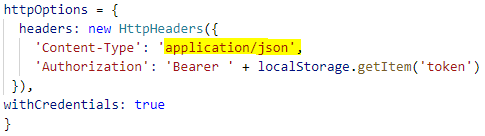


Figura 21 - Especificação do formato, num cabeçalho no frontend da aplicação

# **Angular (frontend)**

## **5.1 Introdução à ferramenta Angular**

A ferramenta utilizada para interface entre utilizador e o servidor, ou seja, a camada de apresentação é baseada em *TypeScript*, é *opensource*, e é mantida pela Google e por uma comunidade de programadores no *GitHub.*

Como dito no parágrafo anterior, é baseada na linguagem TypeScript e fornece uma experiência mais rica, pois ajuda por exemplo a auto completar código.

Todo o processo de desenvolvimente é focado numa programação orientada a objetos, e usa os beneficios do ES2016 (abreviação de ECMAScript) tal como classes ou decoradores, ou *decorators*, são funções que modificam classes *JavaScript*. A framework Angular tem decoradores que anexam metadados às classes, para saber o que essas classes significam e como devem funcionar.

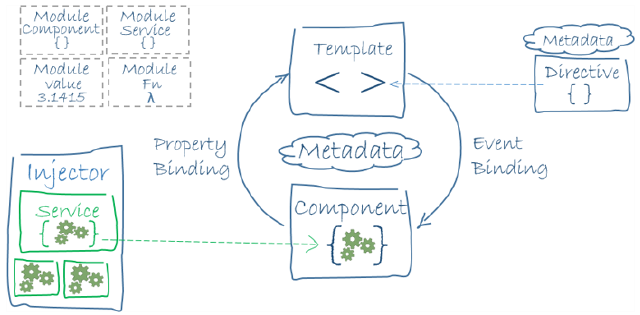
Esta framework consiste em várias bibliotecas, algumas fundamentais, outras são opcionais, mas principalmente consiste em templates HTML com uma marcação específica, como que “angularizada”. Consiste também em classes de componentes (components) para gerir os nossos templates (modelos), adicionando a lógica da aplicação a serviços (services), onde por fim são encaixotadas as componentes e os serviços, em módulos (modules).

Figura 22 - Arquitetura da framework Angular

A figura acima representada, expressa os principais blocos da ferramenta Angular. Este método de separação dos templates HTML, classes de componentes que gerem esses templates, serviços que contêm lógica do negócio e outras componentes, leva a uma projeto arquitetónico lógico com maior reutilização e testabilidade com os blocos separados.

Este princípio de separação de blocos em partes distintas, é conhecido da ciência da computação como separação de conceitos (SoC, *separation of concerns).*

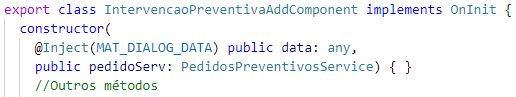
## **5.2 Arquitetura**

## **5.2.1 Injeção de dependência (Dependency injection)**

Injeção de dependência (DI) é uma forma de fornecer uma nova instância de uma classe, com as dependências formadas de que necessita. A maioria dessas dependências são serviços (services). A *framework* pode dizer quais serviços uma componente necessita, dependendo do tipo de parâmetros do construtor.

A seguinte figura mostra a injeção de serviço na componente *IntervencaoPreventivaAddComponent*.

Figura 23 - Injeção de serviços numa componente

A injeção de dependências na framework Angular consiste em duas partes chave:

* *Injector* - um serviço de gestão de dependências que cuida do registo de dependências e descobre como criá-las.
* *Dependencies* - serviços administrados pelo injector, são criados e fornecidos a outros serviços ou componentes.

Um provider é uma instrução para o sistema de [injeção de dependência](https://angular.io/guide/dependency-injection) sobre como obter um valor para uma dependência. Na maioria das vezes, essas dependências são serviços que criamos e fornecemos (*Angular guide about providers*, 2020).

Um facto sobre injeções de dependência, é que todos os serviços são *singletons* (existência de apenas uma instância de uma classe), o que significa que há apenas uma instância do serviço para determinado injetor.

Contudo há diferenças entre injetar *providers* num módulo ou numa componente. Quando este é registado num módulo, o serviço é fornecido em *singleton* para toda a aplicação, ao contrário, numa componente está disponível para essa mesma e para as componentes filho.

**5.2.2 Módulos (*Modules*)**

As aplicações Angular são modulares, e Angular tem o seu próprio sistema de modularidade chamado *NgModules.* Um *NgModule*, quer seja uma raiz (root) ou não, é uma classe com um *decorator* @*NgModule* e é uma característica fundamental desta framework.

Uma classe *NgModule* descreve como é que as partes da aplicação se encaixam. Cada aplicação tem pelo menos um *NgModule*, o módulo raiz que é inicializado para a aplicação. O nome convencional é *AppModule*.

A seguinte figura mostra o uso do *decorator NgModule* e a forma como o módulo é configurado.

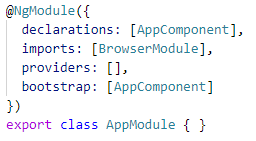


Figura 24 - Configuração simples do módulo raíz (por definição)

As propriedades mais importantes na configuração do *NgModule* são:

* Declarations – são as “*view classes*” que pertencem a este módulo.
* Imports – outros módulos (um ou mais) cujas classes exportadas são necessárias para componentes declaradas neste módulo.
* Providers – são criadores de serviços com os quais este módulo contribui para a coleção global de serviços (estes tornam-se acessíveis em qualquer parte da aplicação).
* Bootstrap – é a visualização da aplicação principal, também chamada de componente raíz.

## **5.2.3 Diretivas e componentes**

A forma como a ferramenta Angular foi criada para simplificar o aspecto de visualização, recorre a características como diretivas, elementos personalizados, interpolação, entre outros.

Os templates têm sintaxe HTML, enriquecida com elementos dinâmicos e, quando a ferramenta (Angular) os renderiza, transforma o DOM (Document Object Model) de acordo com as instruções dadas pelas diretivas.

A diretiva é uma classe com um decorador @Directive. São semelhantes a atributos HTML. Existem dois tipos de diretivas: as estruturais e de atributo. As diretivas estruturais alteram o *layout* adicionando, removendo e substituindo elementos no modelo de documento por objetos (DOM), enquanto as diretivas de atributo alteram a aparência ou o comportamento de um elemento existente.

Um tipo específico de diretiva é uma componente (*component*). A componente é uma diretiva com o decorador (*decorator*) @Component, onde o template da componente é definido.

As componentes são como blocos principais, usados para criar a interface com o utilizador. Dentro da componente, há uma lógica definida que determina o que fazer para oferecer suporte à interface com o utilizador.

No momento de desenvolvimento de um template de uma determinada componente, há necessidade de enviar dados para a parte HTML e definir métodos para lidar com as ações do utilizador. Para simplificar, a framework Angular traz uma característica chamada data binding (ligação de dados).

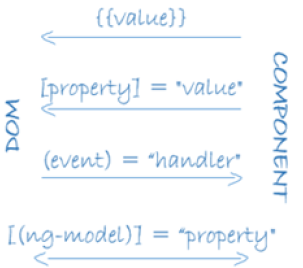
Esta ligação é um processo de sincronização entre um template e a sua respectiva componente.

Figura 25 - Sintaxe da ligação de dados e direção do fluxo de dados

## **5.2.4 Serviços (*services*)**

As componentes, são responsáveis por preparar os dados para a camada de apresentação e por definir métodos que serão utilizados consoantes as ações do utilizador. Por outro lado, a utilização de serviços tem o propósito de organizar o projeto de software na parte *fontend*, isolando a lógica do negócio e separando-a dos controladores.

Na prática não há diferença para o utilizador porque, provavelmente, utilizar serviços não afetará diretamente o comportamento da interface.

Um serviço é uma classe que pode ser utilizada por outros componentes do projeto para por exemplo, solicitar e manipular dados. A destacar é o que diferencia um serviço de outro tipo de componente e como um componente utiliza um serviço.

É caracterizada com o decorador (*decorator*) @*Injectable*() e tem um propósito específico. Este *decorater* diz à framework *Angular* que o serviço em si pode ter dependências definidas no construtor de um serviço e não é necessário se um serviço não tiver nenhuma dependência, contudo é recomendado usar.

Exemplos de serviços, podem ser os seguintes:

* Serviço de registo;
* Serviço de dados;
* Serviço de configuração;



Figura 26 - Implementação da classe serviço PedidosPreventivosService

## **5.3 TypeScript**

A linguagem TypeScritp é um conjunto de funcionalidade adicionadas ao JavaScript. Significa que a linguagem TypeScript gira em torno dos avanços ECMAScript [[2]](#footnote-3) 6, onde adiciona tipagem opcional e programação orientada a objetos. Isto significa que praticamente qualquer código JavaScript é também um código TypeScript.

Benefícios desta linguagem podem ser os seguintes:

* Paradigma orientado a objetos – característica como classes, interfaces, herança, módulos, (entre outros).
* Oferece suporte a bibliotecas JavaScript - TypeScript oferece suporte a cada elemento JavaScript.
* JavaScript é também TypeScript - código escrito em JavaScript com extensão .*js* pode ser convertido em TypeScript fazendo a alteração da extensão .*js* para .*ts*.
* TypeScript é portátil - pode ser executado em qualquer *browser*, dispositivo ou sistema operativo.
* O TypeScript destaca os erros no tempo de compilação durante o momento de desenvolvimento.

Figura 27 - Relação entre TypeScript e ECMAScript

# **Implementação**

## **6.1 Introdução**

A aplicação desenvolvida consiste em duas subpartes, devidamente distintas e separadas: a aplicação frontend, e o servidor backend. Ambas as partes usadas são independentes das tecnologias utilizadas e do ambiente de execução.

A ligação usada para a comunicação entre estas duas aplicações é a arquitetura *REST API*.

A aplicação do servidor utiliza *C#* como linguagem de programação utilizada, recorrendo à framework ASP.NET para criar os serviços.

Para o armazenamento de dados, foi escolhido o servidor *MS SQL* e a framework Entity Framework é usada como *ORM* para fazer a gestão da base de dados.

A aplicação frontend é escrita em TypeScript e usa a framework Angular para criar uma aplicação *user friendly*. Para tornar o desenvolvimento consistente no design da aplicação, o Angular Material foi usado.

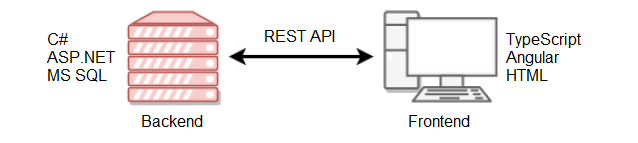
O *MS Visual Studio Code* é o compilador usado para desenvolver a aplicação frontend. É um editor de código com grande suporte para a linguagem TypeScript.

Figura 28 - Visão geral da aplicação.

## **6.2 As aplicações**

## **6.2.1 Estrutura da solução *backend***

A ferramenta usada para o *backend*, o Visual Studio, permite escrever código com precisão e eficiência, sem perder o contexto do arquivo atual. É atualmente uma poderosa ferramenta de desenvolvimento.

O protocolo HTTP não é utilizado apenas para servir páginas da *web*. HTTP é também uma poderosa plataforma para construir *APIs* que expõem serviços e dados. Este mesmo, é simples, flexível e está ao mesmo tempo em toda a parte.

Por outro lado, ASP.NET Web API é uma estrutura para construir APIs recorrendo à .NET Framework.

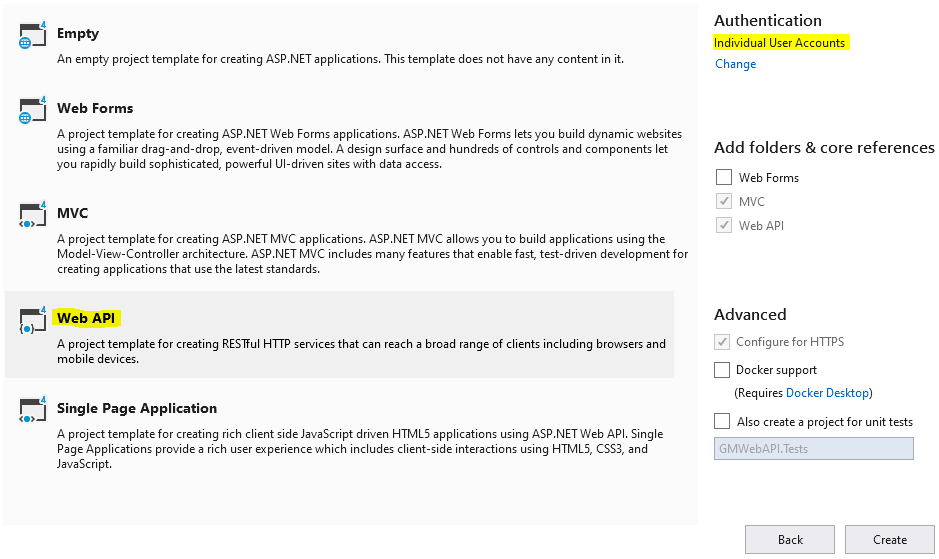
 Para este caso, o projeto é criado ao mesmo tempo que a solução, sendo atribuído o nome *GMWebAPI.*

Figura 29 - Criação do projeto (backend)

A figura acima, ilustra a criação da solução. De referir dois aspetos de extrema importância:

* O tipo de projeto – Web API, contém muitas semelhanças com o tipo de projeto *MVC*, contudo como é utilizada a framework Angular no frontend, não necessitamos da camada *View* referente ao tipo *MVC*.
* Autenticação - *Individual User Accounts*. Referente à necessidade de autenticação de utilizadores aquando da utilização da API, para aceder aos dados e métodos/recursos.

De seguida, o processo prosseguiu com a instação do *Entity Framework*, instalado através do *NuGet package manager*. Só assim é que posteriormente é importado o nosso modelo da base de dados (ADO.NET Entity Data Model).

A seguinte figura ilustra o esquema da solução, de destacar as seguintes partes mais importantes para a aplicação em questão:

* Pasta *AppStart* – referente a questões relacionadas com rotas (*routing*), assim como um validador das passwords referentes aos utilizadores e por fim, protocolo *OAuth;*
* Pasta *Controllers* – contém todas as classes controladores;
* Pasta *DTO* – contém todas as classes DTOs;
* Pasta *Models* – contém o nosso modelo ADO.NET Entity Data Model e do objeto referente ao registo de um utilizador (propriedades);
* Pasta *Providers* – contém métodos relativos à criação de um token durante o tempo de execução da aplicação do lado frontend, e também caracteristicas do utilizador, tais como o tipo de permissão que este tem para aceder a certas àreas da aplicação.
* Ficheiro *Web.config* – serve entre outros, para o IIS (*Internet Information Services)* ler este ficheiro e configurar quando este é hospedado. Contém também *connection string,* usado para estabelecer a ligação entre o servidor e a base de dados;

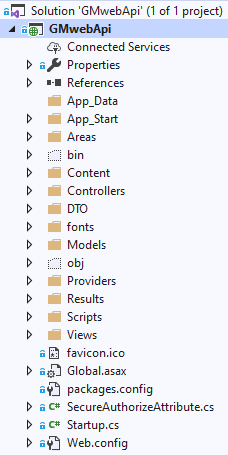


Figura 30 - Aspeto geral da solução

## **6.2.2 Estrutura da solução *frontend***

O Visual Studio Code, é um poderoso editor de código usado para criar a aplicação Angular. Contudo, a framework Angular precisa para compilar, construir e servir, módulos para instalar as depedências necessárias, nomeadamente Node Package Manager (NPM[[3]](#footnote-4)).

Em primeiro lugar, o pacote Angular CLI é instalado via NPM:

* $ *npm install -g @angular/cli*

De seguida, a criação do projeto utilizando o seguinte comando:

* $ *ng new GM\_CLIENT*

O comando anterior gera a aplicação Angular, estruturada em pastas como mostrado nas figuras abaixo. A pasta *src* contém a aplicação Angular e todos os arquivos, como componentes, estilos, modelos HTML e ficheiros de configuração. Os ficheiros fora da pasta *src* destinam-se a oferecer suporte à construção da aplicação.

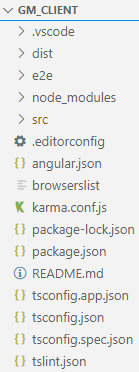
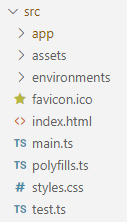


Figura 31 - Detalhe da pasta SRC

Figura 32 - Aspeto geral

Alguns dos ficheiros mais relavantes que não têm propriamente um sentido explicito no seu nome, são descritos na seguinte tabela.

Tabela 4 - Ficheiros do projeto e breve explicação

|  |  |
| --- | --- |
| **main.ts** | O principal ponto de entrada que inicializa o módulo raiz da aplicação para ser executado no *browser*. |
| **tsconfig.ts** | Ficheiro de configuração para o compilador TypeScript. |
| **index.html** | Ficheiro que é carregado primeiro quando alguém visita o *site*. A CLI adiciona todos *CLI* adiciona todos *assets* quando a aplicação é compilada. |
| **package.json** | Configuração do Node Package Manager, listando por exemplo depedências quando a aplicação é usada por terceiros. |
| **angular.json** | Ficheiro de configuração para Angular CLI. Aqui podem ser definidos vários padrões e também configurar quais ficheiros são incluídos quando o projeto é construído. |

Todos as componentes, serviços, módulos e tudo que criado por mim está localizados na pasta / src / app. É importante escolher a estratégia de estrutura certa para esta pasta, de forma a torná-la organizada e fácil de modificar a sua estrutura interna (refactoring).

A pasta *src* contém um ficheiro de seu nome AppModule. Este deve permanecer o mais simples possível, isto porque contém a componente mãe numa hierarquia de componentes. Aqui é também feita a importação dos módulos necessários para toda a aplicação. A seguinte figura ilustra a AppModule da aplicação (formato reduzido, nomeadadamente as *declarations* e *imports*).

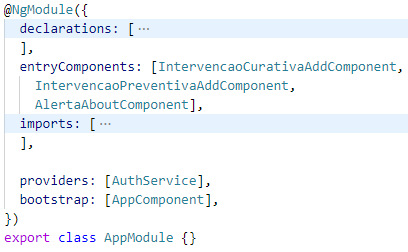


Figura 33 - Definição da AppModule

**6.3 Convenções e boas práticas utilizadas**

## **6.3.1 Boas práticas no frontend**

Nos serviços (services) de uma aplicação Angular, um exemplo de convenção comum é, utilizar a funcionalidade (por exemplo nome de um objeto), seguido pelo tipo (serviço, modelo, ou outro tipo), e por fim a extensão de arquivo.

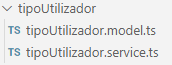


Figura 34 - Exemplo da pasta de um serviço

Na figura acima, o serviço relacionado com o tipo de utilizadores, tem dois ficheiros distintos, e fáceis de distinguir graças à convenção adoptada, sendo cada ficheiro identificado com o tipo de ficheiro (se é um modelo ou um serviço).

 É importante distinguir as pastas serviços das pastas componentes, o que faz sentido para não haver confusão entre pastas. Assim, a prática usada, ao invés de *Camel Case,* é a utilização de hífens entre diferentes palavras, sendo esta prática ilustrada na seguinte imagem, em destaque amarelado:

Figura 35 - Criação de uma componente por comando

Assim, o resultado do comando utilizado e que é ilustrada na imagem anterior, é uma pasta com os vários ficheiros necessários para o negócio de lógica e template (HTML + CSS). A seguinte imagem ilustra a explicação daquilo que foi referido neste parágrafo.

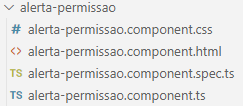


Figura 36 - Pasta referente ao conteúdo da componente

Ao contrário do que é usual em linguagens orientados a objetos, em aplicações Angular é muito usual na definição de um método ser usada a prática *lowerUpperCase.*

## **6.3.2 Boas práticas no backend**

No servidor (backend), a prática utilizada segue as normas comuns referentes a linguagens orientadas ao objetos, tais como *UpperCamelCase* para classes e métodos, e *lowerCamelCase* para atributos.

Contudo, as classes controladores (*controllers)* não diferem na prática *CamelCase* porém, de seguida à funcionalidade da classe controlador, o restante nome da mesma deve terminar em ‘controller’, seguido da sua extensão (neste caso *.cs*).

A figura abaixo ilustra a explicação do parágrafo anterior.

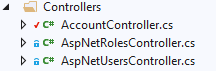


Figura 37 - Excerto da pasta Controllers

A mesma prática aplica-se às classes DTO (*Data Transfer Object*), seguida à funcionalidade da classe, o restante nome da mesma deverá terminar em ‘Dto’, seguido da sua extensão (neste caso *.cs*).

A figura abaixo ilustra a explicação do parágrafo anterior.

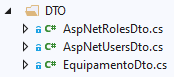


Figura 38 - Excerto da pasta DTO

Para converter objetos pertencentes a classes DTO, há que respeitar a prática adequada na criação de um método que converte esse mesmo objeto DTO para um ‘simples’ objeto.

Para isso, a boa prática utilizada consiste no nome da classe/objeto DTO que entra como parâmetro no método, seguida da palavra ‘TO’ (em português significa ‘para’), e por fim o nome da classe em queremos atribuir o DTO.

A figura abaixo ilustra a explicação anterior.

Figura 39 - Definição de um método DTO, classe IntervencaoPreventivasController

## **6.4 Serviço para pedidos REST API autenticados**

A maior parte da aplicação frontend é interligada a um servidor via HTTP. Todos os *endpoints* da API, exceto para autenticação, requerem um pedido HTTP com autenticação (email e palavra passe). Para esses pedidos, a autenticação utilizando *bearer token* foi a escolhida. Um *header* (cabeçalho) de autorização é anexado a cada pedido HTTP.

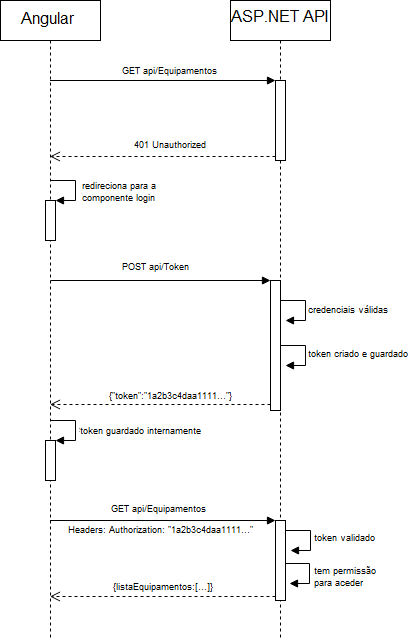
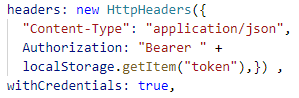
**Na figura abaixo (esquerda) é ilustrado o esquema da transição de pedidos HTTP, mostrando também o caso de autenticação, onde neste *endpoint, não é necessário permissão para aceder.*

Figura 40 - Headers frontend

Figura 41 - Esquema de um pedido HTTP

## **6.4.1 Autenticação no servidor**

O sistema de autenticação (por definição) da framework ASP.NET Web API, necessitava de sofrer alterações para alcançar resultados pretendidos, nomeadamente a adição de mais propriedades relativas ao objeto que envia de retorno para o *frontend.* Tal propriedade consiste no nível de permissão que um utilizador tem, consoante a que o admnistrador da aplicação lhe atribuiu.

Sempre que um utilizador faz autenticação, no header do pedido HTTP, é também enviado o tipo de permissão. Esta informação extra permite que no frontend haja um duplo reforço de proteção (em capítulos posteriores, será ilustrada a proteção de conteúdos no servidor e no frontend). A seguinte figura mostra parte do código necessário para alterar as propriedades enviadas no *header,* do pedido HTTP.

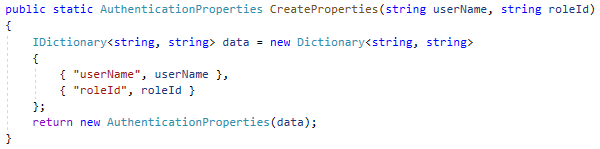


Figura 42 - Classe Startup.Auth.cs, método CreateProperties

Um teste feito com a ferramenta Postman[[4]](#footnote-5), mostra o sucesso alcançado com o código produzido, ilustrado nas seguintes imagens.

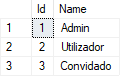


Figura 43 - Tabela AspNetRoles (MS SQL Server)

De referir, que a propriedade *roleId* ilustrada na imagem ao lado, corresponde à propriedade *Id* ilustrada na imagem acima.

Figura 44 - Pedido HTTP (Postman) http://myLocalhost/Token

## **6.5 Registo de um utilizador**

## **6.5.1 Registo de um utilizador (backend)**

O registo de um novo utilizador também necessitou de alterações, quer seja no objeto utilizador (tabela AspNetUsers da base de dados), quer seja no modelo e objeto importado através do ADO.NET Entity Data Model, visto que inicialmente (por definição do controlador) para o registo de um utilizador as propriedades necessárias eram apenas um email, password e a sua confirmação. Assim, de acordo com o desejado, novas propriedades foram acrescentadas na classe *RegisterBindingModel*, de forma a produzir o resultado pretendido.

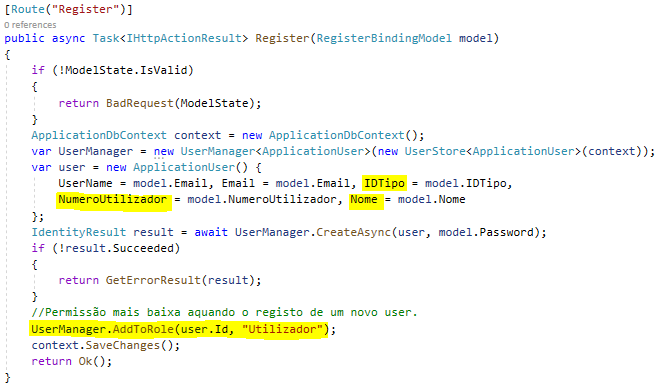
 É também importante referir que já no momento de registo de um novo utilizador, é feita a atribuição de uma permissão para poder ter acesso a componentes, métodos e dados da aplicação (ilustrado a amarelo nas últimas linhas de código da figura acima). Mais tarde, apenas administrador pode decidir qual o tipo de permissão determinado utilizador tem. Se for o caso de atribuição de permissão mais baixa, nada necessita ser feito.

Figura 45 - Método Register, controller AccountController.cs

## **6.5.2 Registo de um utilizador (services, models – frontend)**

Tal como explicado em capítulos anteriores, o frontend é composto por diferentes partes, tais como serviços, componentes, ficheiros de configuração, entre outros. Também em TypeScript (POO[[5]](#footnote-6)), há classes onde definimos um objeto e as suas propriedades, de modo a que, para além de serialização e deserialização, o servidor saiba que determinada propriedade de um objeto se trata e vice-versa.

A seguinte figura, ilustra a classe *Register*, objeto referente ao registo de um novo utilizador, e o local onde essa pasta se encontra. A pasta *shared* refere-se a pastas que contêm os serviços referentes à nossa aplicação, tais como métodos utilizados para pedidos HTTP.

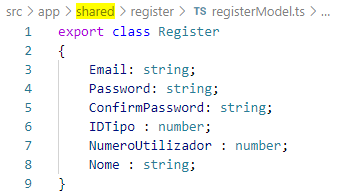


Figura 46 - Classe Register, pasta Register

Para haver transição de dados entre o servidor e o frontend, é necessário criar um pedido HTTP. Ilustrado na imagem abaixo, o método chamado *CreateRegisto,* recebe parâmetros (data) referentes às propriedades do objeto da imagem ilustrada acima. Para este método, fazemos uso de *Observable* do tipo do objeto *Register.*   
*Observables* são uma boa ferramenta pertencente à biblioteca *rxjs,* internamente utilizada pela framework Angular. São uma ajuda a trabalhar com transferência de dados assíncrona, sendo semelhante a *Promises,* contudo, este pode ser uma fonte de transferência de dados contínua, ou seja, o *Observable* poderá emitir dados várias vezes em distintos momentos da sua existência.

Todas as propriedades do objeto *Register* sofrem alteração, sendo estes transformandos de JavaScript para *JSON*, recorrendo ao método *JSON.stringify().*

Figura 47 - Cabeçalho do método CreateRegisto, Register.service.ts (frontend)

1. ISSO 13372:2012 - normas de manutenção. [↑](#footnote-ref-2)
2. ECMAScript é uma linguagem de programação, usada em tecnologias *web* para a criação de scripts executados no cliente (frontend) e no servidor (backend).  [↑](#footnote-ref-3)
3. Node Package Manager – gestor de pacotes para a linguagem JavaScript. [↑](#footnote-ref-4)
4. Postman – aplicação utilizada essencialmente para facilitar teste de código desenvolvido numa API. [↑](#footnote-ref-5)
5. POO – acrônimo de *Programação Orientada a Objetos* [↑](#footnote-ref-6)