



Otimização do Desempenho de Websites Públicos – CDN – ISR

DevScope

2023 / 2024

1180969 Diogo de Sousa Ferreira

ISEP INSTITUTO SUPERIOR
DE ENGENHARIA DO PORTO

Otimização do Desempenho de Websites Públicos – CDN – ISR

DevScope

2023 / 2024

1180969 Diogo de Sousa Ferreira



Licenciatura em Engenharia Informática

Setembro de 2024

Orientador ISEP: **Constantino Martins**

Supervisor Externo: **David Mota**

«*Dedicatória*»

Agradecimentos

Nesta secção (opcional) colocam-se notas de agradecimento às pessoas que contribuíram para a realização da tarefa. No caso de terem usufruído de financiamento via bolsa ou qualquer outro mecanismo formal, deverá ser incluído um agradecimento.

Resumo

Este resumo apresenta o projeto desenvolvido no âmbito do estágio curricular realizado na DevScope.

O problema consiste na criação de uma aplicação para se consultar informações sobre salas e reuniões marcadas pelo utilizador. Para isso, foi necessário recorrer-se ao ISR e a técnicas de pesquisa para se obter a informação necessária, sendo estes os principais objetivos da aplicação.

Este relatório descreve a metodologia necessária para o desenvolvimento do projeto, sendo neste caso aplicado o *Github* que serviu para o acompanhamento do desenvolvimento do projeto. Relativamente a ações foram criadas uma base de dados MongoDB e as respetivas APIs a pedido da organização, onde foram armazenados os dados para serem disponibilizados ao utilizador em tabelas após o seu tratamento onde foi utilizado o ISR. As tabelas também utilizaram as técnicas de pesquisa e de paginação que permitiram uma pesquisa mais rápida. A criação, edição e eliminação de reuniões foram alguns dos casos de uso que o utilizador pode efetuar na aplicação. O resultado dessas operações é refletido na tabela das reuniões e enviado para as respetivas APIs e base de dados.

Este projeto permitiu aplicar os conhecimentos adquiridos ao longo da licenciatura e adquirir novos conhecimentos sobre os temas abordados no mesmo e contribuir para a criação de uma aplicação moderna e eficiente, embora possa estar sujeita a alterações e melhorias.

Palavras-chave (Tema): ISR, técnicas de pesquisa, tabelas, paginação, formulários

Palavras-chave (Tecnologias): MongoDB, ISR, API, CRUD, Next.js, React

Abstract

This summary presents the project developed as part of the curricular internship carried out at DevScope.

The problem consists of creating an application to consult information about rooms and meetings scheduled by the user. For this, it was necessary to use ISR and search techniques to obtain the necessary information, which are the main objectives of the application.

This report describes the methodology required for the development of the project, in which GitHub was used to monitor the project's progress. Regarding actions, a MongoDB database and the respective APIs were created at the request of the organization, where the data were stored to be made available to the user in tables after processing, where ISR was used. The tables also utilized search and pagination techniques, allowing for faster searches. The creation, editing, and deletion of meetings were some of the use cases that the user can perform in the application. The result of these operations is reflected in the meetings table and sent to the respective APIs and database.

This project allowed the application of knowledge acquired throughout the degree and the acquisition of new knowledge on the topics addressed in the project, contributing to the creation of a modern and efficient application, although it may be subject to changes and improvements.

Keywords (Topic): ISR, search techniques, tables, pagination, forms

Keywords (Technologies): MongoDB, ISR, API, CRUD, Next.js, React

Índice

<i>Resumo</i>	<i>ix</i>
<i>Abstract.....</i>	<i>x</i>
1 Introdução	2
1.1 Enquadramento/Contexto	2
1.2 Descrição do Problema.....	2
1.3 Objetivos	3
1.4 Abordagem.....	3
1.5 Resultados esperados	4
1.6 Planeamento do trabalho.....	5
1.7 Estrutura do relatório.....	6
2 Estado da arte	7
2.1 Trabalhos relacionados	7
2.2 Tecnologias existentes	11
2.3 Sumário	15
3 Análise e desenho da solução	16
3.1 Domínio do problema	16
3.2 Requisitos funcionais e não funcionais.....	17
3.3 Desenho da solução	24
4 Implementação da Solução	33
4.1 Descrição da implementação.....	33
4.2 Testes.....	88
5 Conclusões.....	95
5.1 Objetivos concretizados	95
5.2 Limitações e trabalho futuro	96
5.3 Apreciação final	97
Referências.....	99

Índice de Figuras

Figura 1 - Repositório do Github utilizado durante o desenvolvimento do projeto.....	4
Figura 2 - Diagrama de Gantt do projeto	5
Figura 3 - Modelo de Domínio.....	17
Figura 4 - Diagrama de Casos de uso.....	19
Figura 5 – SSD Efetuar Login.....	20
Figura 6 – SSD Listar Salas	21
Figura 7 – SSD Listar Reuniões.....	21
Figura 8 – SSD Criar Reunião	22
Figura 9 – SSD Editar Reunião	23
Figura 10 – SSD Eliminar Reunião.....	24
Figura 11 – Diagrama ER da base de dados.....	25
Figura 12 – Base de dados Mongo DB.....	25
Figura 13 - Diagrama de vista lógica de nível 1	26
Figura 14 - Diagrama de vista lógica de nível 2	27
Figura 15 - Diagrama de vista lógica de nível 2 alternativo.....	28
Figura 16 - Diagrama de vista lógica de nível 3	29
Figura 17 - Diagrama alternativo de vista lógica de nível 3.....	30
Figura 18 - Diagrama de vista física	31
Figura 19 - Diagrama de vista física alternativo	32
Figura 20 – Esquema da base de dados no MongoDB	33
Figura 21 – Exemplo de reunião na base de dados.....	34
Figura 22 – Exemplo de sala na base de dados	35
Figura 23 - Conexão à base de dados no Visual Studio Code.....	36
Figura 24 - Base de dados na aplicação.....	37
Figura 25 - String de conexão	37

Figura 26 - Classe mongodb.js	38
Figura 27 - Pasta model	38
Figura 28 - Appointment.js	39
Figura 29 - Room.js	40
Figura 30- Pasta API com os ficheiros.....	40
Figura 31 – Ficheiro route.js no caminho "/api/appointments"	41
Figura 32 - Lista parcial de reuniões na API das reuniões	42
Figura 33 - Ficheiro route.js no caminho "/api/appointments/[id]"	43
Figura 34 - API de uma reunião filtrada pelo id.....	44
Figura 35 - Ficheiro route.js no caminho "/api/rooms"	44
Figura 36 - Lista parcial de salas na API das salas.....	45
Figura 37 – Ficheiro route.js no caminho "/api/rooms/[id]"	46
Figura 38 - API de uma sala filtrada pelo id	46
Figura 39 - Fluxograma do login	47
Figura 40 - Variáveis no ficheiro .env para o login	47
Figura 41 – Botão de login.....	48
Figura 42 - Ficheiro _app.js.....	48
Figura 43 - Classe route.js.....	49
Figura 44 - Login pela Microsoft.....	49
Figura 45 - Criação do botão de logout	50
Figura 46 - Fluxograma da listagem de informação sobre as salas	50
Figura 47 - Pasta do dashboard	51
Figura 48 - Conteúdo da classe layout.tsx	51
Figura 49 - Classe sidenav.tsx	52
Figura 50 - Parte superior da barra lateral de navegação	53
Figura 51 - Conteúdo da classe page.tsx	54
Figura 52 - Tabela de salas	55

Figura 53 - Reunião da sala que se encontra ocupada.....	55
Figura 54 - Filtragem de salas na barra de pesquisa	55
Figura 55 - Pasta 'lib'.....	56
Figura 56 - Parte inicial do bloco try da função fetchFilteredRooms	56
Figura 57 - Constante revalidationTime	56
Figura 58 - Bloco catch da função fetchFilteredRooms.....	57
Figura 59 - Criação de objetos para serem tratados	57
Figura 60 – Definição da disponibilidade das salas e tratamento final dos dados.....	58
Figura 61 - Tipo RoomsTableType	59
Figura 62 - Fluxograma da listagem de reuniões.....	60
Figura 63 - Lista de reuniões.....	61
Figura 64 - Conteúdo da classe page.tsx responsável pela listagem de reuniões.....	62
Figura 65 - Parte inicial da função fetchAppointmentsPages	63
Figura 66 – Bloco catch.....	64
Figura 67 - Iteração da lista de reuniões na função fetchAppointmentPages	64
Figura 68 - Tipo AppointmentsTableType	65
Figura 69 - Número de reuniões máximo por cada página	65
Figura 70 – Aquisição das reuniões a serem listadas na tabela	66
Figura 71 - Parte inicial da função fetchFilteredAppointments()	66
Figura 72 - Bloco catch	67
Figura 73 - Bloco catch	67
Figura 74 - Iteração da lista de reuniões para eliminar as reuniões passadas	67
Figura 75 - Mensagem sobre a eliminação de reuniões passadas no terminal	68
Figura 76 - Tabela de reuniões atualizada.....	68
Figura 77 - Iteração da lista de reuniões na função fetchFilteredAppointments()	69
Figura 78 - Definição das variáveis a serem mostradas na tabela	70
Figura 79 – Criação da tabela de reuniões	71

Figura 80 - Fluxograma da criação da reunião	72
Figura 81 - Classe page.tsx.....	73
Figura 82 - Função fetchRooms().....	74
Figura 83 - Parte inicial do create-form.tsx	75
Figura 84 - Formulário de criação.....	75
Figura 85 - Tipo da variável date no create-form.tsx	75
Figura 86 - Tipo da variável start no create-form.tsx	76
Figura 87 - Tipo da variável end no create-form.tsx	76
Figura 88 - Tratamento da submissão do formulário.....	76
Figura 89 - Critérios de validação do formulário.....	78
Figura 90 - Validação e transformação dos dados do formulário de criação.....	79
Figura 91 - Inserção da reunião na API.....	80
Figura 92 - Fluxograma da edição de uma reunião	81
Figura 93 - Classe page.tsx.....	82
Figura 94 - Função fetchAppointmentById	83
Figura 95 - Formulário de edição.....	83
Figura 96 - Exemplo de variável com dados definidos da reunião no formulário de edição ..	84
Figura 97 - Tratamento dos dados da reunião a ser editada e da sua submissão	84
Figura 98 - Inserção da reunião editada na API.....	85
Figura 99 - Fluxograma da eliminação de uma reunião	86
Figura 100 - Função DeleteAppointments.....	87
Figura 101 - Pop-up de eliminação.....	87
Figura 102 - Body Criar Reunião	88
Figura 103 - Teste de Criar Reunião	89
Figura 104 - Código de sucesso da criação da reunião.....	89
Figura 105 - Reunião criada.....	89
Figura 106 - Teste de Listar Reuniões.....	89

Figura 107 - Código de sucesso da listagem de reuniões	90
Figura 108 - Teste Obter Reunião por ID	90
Figura 109 - Reunião Obtida por ID	90
Figura 110 - Código de sucesso da listagem de uma reunião específica.....	91
Figura 111 - Body Editar Reunião	91
Figura 112 - Teste Editar Reunião	91
Figura 113 - Código de sucesso da edição de uma reunião	91
Figura 114 - Reunião Editada.....	92
Figura 115 - Teste Eliminar Reunião.....	92
Figura 116 - Código de sucesso da eliminação de uma reunião	92
Figura 117 - Processo de eliminação de uma sala.....	93
Figura 118 - Teste Listar Salas	93
Figura 119 - Código de sucesso da listagem das salas.....	93
Figura 120 - Teste para Obter Sala por ID	94
Figura 121 - Sala Obtido por ID	94
Figura 122 - Código de sucesso da listagem de uma sala específica.....	94

Índice de Tabelas

Tabela 1 - Planeamento do trabalho	5
Tabela 2 - Comparação entre SSR, SSG e ISR [6]	14

Notação e Glossário

ISR	<i>Incremental Static Regeneration</i>
SSR	<i>Server Side Rendering</i>
SSG	<i>Static Site Generation</i>
CDN	<i>Content Delivery Network</i>
SEO	<i>Search Engine Optimization</i>
CSR	<i>Client Side Rendering</i>
SPA	<i>Single Page Applications</i>
CMS	<i>Content Management Systems</i>
DPR	<i>Distributed Persistent Rendering</i>

1 Introdução

O presente relatório tem como apresentar e descrever o projeto realizado no âmbito da unidade curricular de Projeto/Estágio (PESTI) do 2º semestre do 3º ano da Licenciatura em Engenharia Informática do ISEP.

O projeto desenvolvido neste estágio curricular foi realizado na empresa DevScope – Soluções de Sistemas e Tecnologias de Formação, S.A..

A DevScope [1] é uma empresa de desenvolvimento de *software* que foi fundada em 2003 no Porto, em Portugal. A empresa possui um escritório no Porto e entre 51 e 200 funcionários.

A DevScope [1] é parceira da *Microsoft* para soluções nas áreas de *Data & AI*, *Digital & App Innovation* e *Modern Workplace* e possui 2 especializações avançadas em *Analytics* e *Low Code Application Development*. A empresa também recebeu prémios em 2021 de Parceiro do Ano da *Analytics* pela *Microsoft Portugal*, em 2022 com Parceiro do Ano em *Power Platform* e em 2023 com o Parceiro do Ano em *Low Code*.

1.1 Enquadramento/Contexto

Este projeto foi proposto pela organização depois de uma entrevista que foi realizada onde tentaram perceber as potenciais áreas de interesse.

A área em questão coincide com algumas áreas em que a organização desempenha os seus serviços e está a evoluir e modernizar-se nos últimos anos devido à cada vez maior utilização de aplicações da *Microsoft* e outras empresas de tecnologias e ao facto de as organizações das mais diversas áreas necessitarem de aplicações informáticas para a manutenção e expansão dos seus negócios.

1.2 Descrição do Problema

O problema consiste na otimização do desempenho de *websites* públicos, tendo em conta que os utilizadores procuram cada vez mais *websites* mais rápidos e com um maior desempenho. Para isso, este projeto foca-se em temas como o Centro de Distribuição de Comandos (CDN), geração de websites estáticos, programação assíncrona, Regeneração

Estática Incremental (ISR) e pesquisas eficientes como potenciais soluções para o problema e consequentemente melhorar as experiências dos utilizadores relativamente à utilização de websites.

1.3 Objetivos

O principal objetivo deste estágio foi a criação de uma aplicação que permitisse a marcação de reuniões pelos membros da organização e a consulta das informações dessas mesmas reuniões que estão listadas em tabelas.

A aplicação permitirá aos utilizadores marcarem, editarem e eliminarem as reuniões associadas ao mesmo e consultarem as informações sobre as salas onde as reuniões foram marcadas.

Para isso, foram criadas uma base de dados MongoDB onde serão armazenadas as informações sobre as salas e as reuniões e as respetivas APIs para permitirem a consulta e a inserção de novos dados sobre as reuniões através de operações CRUD.

Antes de os dados serem listados nas tabelas, serão tratados, sendo aí que será aplicado o ISR que permitirá apenas que os conteúdos e as páginas que foram alterados sejam recarregados, tornando a aplicação mais rápida e diminuindo a utilização de recursos.

Nas tabelas foram aplicadas técnicas de pesquisa com base em vários critérios o que permite uma filtragem mais rápida e eficiente dos dados necessários. Na tabela das reuniões também foi aplicada a paginação onde será possível listar apenas algumas instâncias de reuniões em várias páginas.

1.4 Abordagem

A metodologia utilizada durante o desenvolvimento deste projeto foi o Github em que foi criado um repositório, representado na figura 1, para onde foram submetidas as alterações à medida que foram efetuadas. Este repositório foi compartilhado com o orientador do ISEP e o supervisor da organização para que os mesmos pudessem acompanhar o desenvolvimento do projeto.















 DiogoFerreira2000 Classe inutilizada apagada	884b782 · last week	 20 Commits
 app	Classe inutilizada apagada	last week
 docs	Pasta doc reformulada	last week
 .eslintrc.json	Initial commit from Create Next App	10 months ago
 .gitignore	Initial commit from Create Next App	10 months ago
 .nvmrc	Initial commit from Create Next App	10 months ago
 .rnd	Criação da conexão à base de dados e à api	5 months ago
 next.config.js	Initial commit from Create Next App	10 months ago
 package-lock.json	Login	last week
 package.json	Login	last week
 postcss.config.js	Initial commit from Create Next App	10 months ago
 tailwind.config.ts	Initial commit from Create Next App	10 months ago
 tsconfig.json	Criação da conexão à base de dados e à api	5 months ago

Figura 1 - Repositório do Github utilizado durante o desenvolvimento do projeto

1.5 Resultados esperados

Relativamente aos resultados esperados, pode-se destacar os seguintes pontos que poderão impactar o projeto e as interações com futuros utilizadores de uma forma positiva:

- Redução do tempo de carregamento e da carga do servidor: o servidor só irá ser consultado em caso de atualização do conteúdo do site ou quando a página em questão for acedida o que irá diminuir o tempo de carregamento caso a página em questão não sofra alterações;
- Aumento da eficiência: apenas algumas partes do *website* serão atualizadas sem necessidade de o *website* ser todo regenerado no caso do ISR;
- Pesquisa mais ágil: a implementação destas tecnologias permitirá diminuir o tempo de carregamento do *website* e consequentemente favorecê-lo no caso de pesquisa no *browser*, aumentando a probabilidade de o mesmo aparecer na primeira página e de ser visitado por mais utilizadores;
- Técnicas de pesquisa: as técnicas de pesquisa permitirão aos utilizadores encontrar o que precisam nas tabelas mais rapidamente e eficazmente;

1.6 Planeamento do trabalho

Na figura 2 encontra-se representado o Diagrama de Gantt com as datas em que as diferentes etapas do projeto e do relatório foram realizadas. À medida que o projeto foi desenvolvido, foram adicionados ao relatório conteúdos relacionados com o mesmo e outros conteúdos derivados de pesquisas bibliográficas que complementam e informam sobre as tecnologias e abordagens utilizadas no desenvolvimento do projeto.

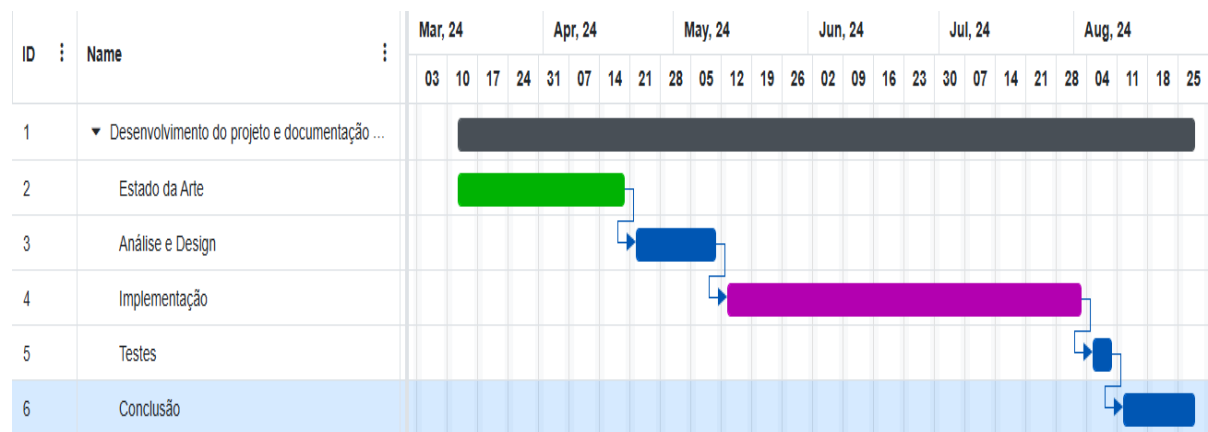


Figura 2 - Diagrama de Gantt do projeto

A tabela 1 descreve com maior detalhe as datas apresentadas na figura 1 e complementa o diagrama de Gantt.

Tabela 1 - Planeamento do trabalho

Tarefa	Início	Fim
Estado da Arte	12/03/2024	19/04/2024
Análise e Design	22/04/2024	10/05/2024
Implementação	13/05/2024	02/08/2024
Testes	05/08/2024	09/08/2024
Conclusão	12/08/2024	28/08/2024

1.7 Estrutura do relatório

Para além da introdução, o relatório contém mais 4 capítulos.

No segundo capítulo é apresentado o estado da arte, onde são expostos os trabalhos relacionados em que são analisados os níveis de progresso, impactos, vantagens e desvantagens das tecnologias utilizadas neste projeto juntamente com outras tecnologias relacionadas.

No terceiro capítulo são abordados o domínio do projeto e o desenho da solução juntamente com a análise dos requisitos funcionais e não funcionais do projeto.

No quarto capítulo é explicada a implementação da solução descrita no capítulo anterior e são demonstradas as funcionalidades implementadas, assim como os testes e a avaliação do projeto desenvolvido.

No último capítulo é apresentada a conclusão onde são incluídos os objetivos concretizados, as limitações, pontos a melhorar e uma opinião pessoal sobre o projeto desenvolvido e a experiência vivida durante o estágio curricular.

2 Estado da arte

O presente capítulo encontra-se dividido em duas secções que são os trabalhos relacionados e as tecnologias existentes.

Na primeira secção será analisado o impacto principalmente do ISR e do CDN mas também de outras tecnologias semelhantes no desempenho de websites e como isto afeta os seus utilizadores e respetivas interações. Na segunda secção, o ISR e as tecnologias relacionadas que existem serão analisadas e comparadas juntamente com uma justificação para a escolha da aplicação das tecnologias no projeto.

2.1 Trabalhos relacionados

Neste ponto vão ser apresentados trabalhos e projetos recentes relacionados com o tema do projeto com as respetivas citações e referências apresentadas no capítulo das Referências.

Com base no conteúdo desses projetos, também serão discutidos e expostos os impactos e as consequências em variados cenários na utilização das tecnologias que irão ser abordadas neste projeto e nas suas interações com os utilizadores.

Esta análise torna-se necessária para se entender em que nível de progresso se encontram as tecnologias relacionadas com o tema, as suas respetivas vantagens e desvantagens e consequentemente quais são as melhores abordagens a serem utilizadas no projeto. Por esta razão, é imperativo realizar esta análise antes de se abordar as outras temáticas neste relatório e de se começar o desenvolvimento do projeto.

2.1.1 Análise do impacto do *Next.js* no desempenho de um *website* e no SEO

Neste relatório [2] são analisados os impactos da utilização do *Next.js* e outras tecnologias, tais como o ISR, e técnicas de otimização no desempenho de *websites* e na interação do utilizador com *websites* que utilizem essas tecnologias e como é possível melhorar o SEO e o desempenho do *website* em parâmetros tais como a velocidade de carregamento e a visibilidade do *website*.

As abordagens utilizadas foram a implementação do SSR e do ISR juntamente com o *Next.js* que pode ser executado quer do lado do cliente e do servidor, o que reduziu o tempo de carregamento do *website*, melhorou o desempenho do *website* e a experiência do utilizador. Consequentemente, isto teve um impacto positivo no SEO que prioriza *websites* que fornecem uma experiência mais agradável ao utilizador e são mais rápidos. A melhoria do desempenho assegura uma maior visibilidade do *website* no *browser* e mais visitantes.

A redução do tamanho do código de *Java Script* juntamente com a implementação de SSR e ISR contribuiu para uma maior facilidade na manutenção do *website*, tornando as atualizações e adição de novas funcionalidades mais práticas. Isto permite que o *website* esteja sempre atualizado e adaptado aos avanços tecnológicos e seja bem-sucedido, indo ao encontro das necessidades e requisitos dos utilizadores tais como *websites* mais rápidos e práticos e dos *browsers* e motores de pesquisa que favorecem os *websites* mais relevantes e que possuem as melhores interações com os seus utilizadores.

O *Google Analytics* e o *PageSpeed* são ferramentas que ajudam os *developers* a verificarem o desempenho do *website* e a identificarem os parâmetros e as áreas que precisam de serem melhoradas. Os *developers* também analisam e comparam o desempenho do seu *website* com o desempenho dos *websites* concorrentes. Isto permite que o *website* continue atualizado e a ser o melhor do seu ramo.

As atualizações frequentes do *website* permitem melhorar o desempenho do *website* e a experiência dos utilizadores. A correção de erros, a criação de novo conteúdo e a atualização do *website* com as tecnologias mais recentes permitem manter o *website* relevante e atrair novos utilizadores. O crescente acesso a dispositivos eletrónicos e à Internet torna esta abordagem mais pertinente para um *website* se manter relevante e manter e atrair utilizadores novos.

O *Bootstrap* é uma *framework* que permite a criação de vários tipos de temas e layouts que podem ser facilmente adaptados a vários dispositivos. Isto assegura a funcionalidade do *website* em variados dispositivos, traduzindo-se numa melhor interatividade com o utilizador e no aumento da visibilidade do *website* aquando de pesquisas.

Finalmente, a aplicação de *Next.js* juntamente com técnicas de otimização apresenta os seguintes benefícios:

- A melhoria do desempenho e do SEO que se traduz em tempos de carregamento mais rápidos e uma maior visibilidade do *website* aquando da pesquisa;

- Uma melhor interatividade para o utilizador que se deve a um tempo menor de carregamento das páginas e uma experiência mais agradável para o utilizador;
- Aumento do tráfego e do lucro do *website*, tendo em conta que os utilizadores tendem a visitar e a interagir com páginas que apresentam uma experiência de interação mais fiável e que aparecem no topo das pesquisas nos browsers;
- Possibilidade de atrair audiências globais que falam diversas línguas.

As abordagens utilizadas são essenciais para manter um *website* competitivo, viável e atualizado num mundo cada vez mais evoluído tecnologicamente e com os utilizadores cada vez mais ligados à *Internet* e interessados em *websites* mais eficientes, interativos e práticos, com requisitos mais avançados e rígidos nesse aspeto.

2.1.2 Implicações do Edge Computing no Static Site Generation

Neste projeto [3] são analisadas as consequências da implementação, em termos de oportunidades e de desafios, dos mais recentes desenvolvimentos de *Edge Computing* no SSG, no ISR e outros métodos de renderização. A evolução das tecnologias de renderização de websites também será analisada.

A renderização de *websites* evoluiu ao longo do tempo e foi suportada largamente pelo crescimento do mercado e pela mudança do tipo de plataformas web que predominam em cada época, começando no site, passando pela aplicação e posteriormente pelas redes sociais.

Os primeiros *websites* que foram desenvolvidos nos anos 90 recorriam ao SSR, em que o site em questão consistia de páginas e documentos HTML que eram armazenados em um servidor e servidos ao cliente em um *browser* sem customização adicional, embora fosse adicionada a funcionalidade dinâmica para permitir alterações ao conteúdo e interatividade.

No final da década de 2000 e início da década de 2010, o SSR perdeu popularidade em detrimento do CSR. Isto aconteceu devido ao facto de o CSR permitir a mudança apenas das partes do website que foram alteradas sem ser necessário um pedido para recarregar a página toda tal como acontece com o SSR. O aparecimento do SPA que permite ajustar o conteúdo conforme as interações do utilizador também é algo notável.

O SSG complementa o SSR e CSR, utiliza um servidor estático e possui vários benefícios tais como tempos de carregamento mais rápidos, maior segurança, maior compatibilidade com diferentes sistemas e gestão mais eficiente de recursos. Esta tecnologia tem sido utilizada

em sites mais pequenos mas torna-se inconveniente em sites maiores pois o site inteiro terá de ser recarregado caso o seu conteúdo seja minimamente alterado, embora exista uma técnica chamada compilação incremental que permite que apenas as partes que foram alteradas sejam recompiladas.

As *frameworks* mais recentes como o caso do *Next.js* disponibilizam o SSR, CSR e SSG, permitindo um funcionamento híbrido. Esta abordagem permite a utilização da tecnologia mais vantajosa para o projeto que estiver a ser desenvolvido. Além disso, foi criado um método de renderização que é o ISR.

O ISR combina SSG e SSR sem ser necessário recarregar o site inteiro. As páginas são armazenadas na cache e os pedidos vão buscar que estiverem aí armazenados. Em 2021 foi introduzido o DPR que permite corrigir as falhas do ISR. Enquanto no ISR, depois do primeiro *refresh* à página em questão após as alterações terem sido armazenadas na cache, ainda poderá ser visto conteúdo antigo e desatualizado, no DPR isso já não acontece.

Os CDNs que surgiram por causa da maior procura permitem que se distribua o conteúdo por um maior número de servidores para que seja diminuída a sobrecarga de um servidor e responder a pedidos a partir de servidores próximos do cliente reduzindo assim a latência.

O *Edge Computing* acaba por ser a evolução natural do CDN pois acaba por lidar também com a computação com o cliente e não apenas com pedidos com acontece com o CDN. Esta abordagem leva a que sejam reconsideradas estruturas tradicionais para serem compatíveis com a infraestrutura global embora demonstre que melhora a renderização de páginas.

O *Edge Computing* é uma opção mais viável que o CDN para os programadores pois permite uma maior capacidade de programação e adaptabilidade em termos de pedidos aos clientes e respostas dos servidores, traduzindo-se em tempos de resposta mais baixos.

Este estudo permitiu-nos concluir em termos de benefícios que as latências das plataformas *edge* são baixas, especialmente nos casos do SSG e do ISR. No caso do SSG as latências são mais elevadas devido ao maior número de pedidos ao servidor.

Relativamente sobre a temática sobre quando deve ser utilizado o ISR em vez do SSG, ficou concluído que o ISR deve ser utilizado em *websites* complexos onde existem alterações frequentes e rápidas ao seu conteúdo tais como redes sociais embora se fosse aplicada a compilação incremental juntamente com o SSG poderia diminuir os custos

As questões que se levantam são as limitações em termos de aplicabilidade em outros ambientes e os custos.

2.2 Tecnologias existentes

O *Next.js* [4] é uma *framework* do *React* que é utilizada para criar aplicações *web full-stack*. Enquanto o *React* é utilizado para criar interfaces que serão utilizadas na aplicação *web*, o *Next.js* disponibiliza detalhes e funcionalidades adicionais, permitindo a configuração de ferramentas necessárias para o *React*, tal como a sua compilação. O *Next.js* foi utilizado nesta aplicação pois é uma das *frameworks* [5] onde é possível a implementação do ISR.

A comparação do ISR com outras tecnologias tais como SSR e SSG [6] deve-se ao facto de estas tecnologias de renderização poderem também ser utilizadas em projetos que recorram ao *Next.js* e de serem utilizadas simultaneamente. Para isso, é necessário saber como funcionam, as suas vantagens e desvantagens e quando devem ser utilizadas para proporcionar um melhor funcionamento e desempenho da aplicação juntamente com uma melhor experiência para o utilizador.

O *Next.js* possui 2 tipos de diferentes de *routers* [4] que são o *App Router*, que vai ser utilizado neste projeto, e o *Pages Router*. O *App Router* permite a utilização de funcionalidades mais recentes como o *Server Components* e o *Streaming*. O *Pages Router* é a versão original e mais antiga utilizado em aplicações que recorrem ao SSR.

O *Next.js* pré-renderiza [4] todas as páginas, o que significa que o HTML é gerado antecipadamente para cada página ao invés de ser o *Java Script* a fazer tudo isso do lado do cliente, proporcionando um melhor rendimento. O HTML possui o código necessário para que a página *web* funcione corretamente quando é aberta no *browser*.

Existem 2 formas de efetuar a pré-renderização [7] que são o SSG e o SSR. Ambos os modelos podem ser utilizados em apenas uma aplicação *Next.js*, onde se utiliza o SSG em determinadas páginas e o SSR nas outras páginas. O SSR possui um melhor desempenho relativamente ao SSG, pelo que é recomendado utilizar-se o primeiro modelo, embora em certas situações, o SSR seja a única abordagem possível.

2.2.1 SSR

No caso do SSR [7], o HTML necessário para a página é gerado em cada pedido. A página é renderizada no servidor e posteriormente é enviada para o cliente.

As vantagens [8] incluem o facto de que as páginas carregam e funcionam mais rapidamente, permitindo que as mesmas apareçam no topo das pesquisas quando se efetua uma pesquisa no *Google*, a auxílio no carregamento da página quando o utilizador possui uma conexão lenta e/ou instável à Internet e/ou um dispositivo desatualizado.

As desvantagens [8] traduzem-se no aumento da complexidade deste conceito e do tempo de execução à medida que o tamanho e a complexidade da aplicação também aumentam e na utilização do servidor para a geração de páginas.

2.2.2 SSG

Relativamente ao SSG [9], o HTML necessário para a página é gerado quando a mesma é compilada e executada, sendo que o HTML vai ser reutilizado em cada pedido que seja feito daí em diante. Por esta razão, as páginas são mais rápidas quando este tipo de pré-renderização é utilizado.

Se for possível efetuar *pre-rendering* [9] antes dos pedidos do utilizador, deve ser utilizado o SSG. Este método não é a melhor opção se o conteúdo da página ou do *site* em questão estiver sujeito a constantes alterações, podendo o conteúdo mudar a cada pedido. Neste caso, é recomendada a utilização do SSR.

O SSG [9] pode ser utilizado em vários tipos de páginas, tais como as páginas de *marketing*, *posts de blogs*, listagem dos produtos de um site de *e-commerce* e documentação, por estas páginas não estarem sujeitas a alterações constantes.

2.2.3 ISR

Neste projeto, foi solicitado pela organização que fosse utilizado o ISR para a pré-renderização dos dados.

O ISR [10] permite ao utilizador criar ou atualizar páginas depois de ter gerado todo o *website* e também permite a utilização do SSG em páginas individuais sem ser necessário recarregar todo o *website*, podendo ser aplicado em milhões de páginas de um *website*.

O ISR [10] permite que um utilizador veja a mesma versão de uma página durante um determinado período. Esta versão encontra-se armazenada na cache. Após esse período, a página é regenerada, armazenada na cache e mostrada ao utilizador caso tenham sido realizadas alterações.

As principais vantagens do ISR [5] são o menor tempo de compilação, a redução de carga do *back-end* por recorrer à cache onde armazena e vai buscar os dados e o melhor desempenho associado.

2.2.4 Comparação entre tecnologias

Neste subcapítulo vão ser analisados os cenários adequados para a utilização de cada uma das tecnologias nas aplicações *Next.js* para se poder otimizar o desempenho e a usabilidade e vai ser feita uma comparação entre as várias tecnologias recorrendo a vários critérios pertinentes ao desenvolvimento de uma aplicação e respetiva interação com o utilizador.

O SSR [11] é ideal para aplicações mais pequenas que necessitam de ser atualizadas constantemente para obter dados em tempo real, através da geração de HTML a cada pedido.

O SSG [11] é a melhor tecnologia para ser aplicada em *websites* cujo conteúdo muda com pouca frequência pois o HTML necessário é pré-renderizado apenas uma vez e reutilizado sempre que necessário.

O ISR [11] é a melhor tecnologia para projetos idênticos ao projeto desenvolvido, ou seja que são complexos com várias páginas sujeitas a alterações mais frequentes pois combina o melhor do SSG, permitindo a atualização do conteúdo em determinados intervalos de tempo definidos, com o facto de não ser necessária a reconstrução do *website* ou aplicação inteira, sendo apenas atualizados os conteúdos que foram alterados.

Para a comparação entre as diferentes tecnologias vão ser utilizadas vários critérios [6] tais como o tempo de compilação, a adequação para aplicações com conteúdo dinâmico, o SEO, o tempo de renderização da página no browser e o grau de novidade do conteúdo a ser exibido, ou seja, um indicador que irá verificar se o conteúdo mais recente é mostrado. Esta comparação será visível na tabela 2.

Tabela 2 - Comparação entre SSR, SSG e ISR [6]

	SSR	SSG	ISR
Tempo de compilação	Baixo	Elevado	Elevado
Adequação para aplicações com conteúdo dinâmico	Elevada	Baixa	Baixa
SEO	Elevado	Elevado	Elevado
Tempo de renderização da página no browser	Elevado	Baixo	Médio
Grau de novidade do conteúdo a ser exibido	Elevado	Baixo	Médio-alto

De acordo com os dados desta tabela podemos concluir que o SSG e o ISR são mais adequados para aplicações com muito conteúdo mais estático [6] devido aos tempos de renderização mais baixos e tempos de compilação mais elevados, embora o ISR seja mais complexo e penas possa ser utilizado em certas *frameworks* [5], tal como foi mencionado mais acima. O SSR pode ser utilizado em aplicações *web* mais dinâmicas [6] sujeitas a alterações mais frequentes mas mais pequenas e por isso o tempo de compilação nesta situação é mais baixo.

No ISR o tempo de renderização é mais elevado [6] devido às requisições periódicas ao servidor embora o grau de novidade seja relativamente elevado, mas mais baixo que no SSR [6], pois existe um pequeno período antes da atualização do conteúdo da *cache* em que o utilizador irá ver conteúdo antigo. Apesar disso, o grau de novidade é relativamente elevado [6] pois apenas o conteúdo alterado durante aquele período irá ser recarregado. Isto permite reforçar o que foi mencionado acima na descrição do ISR sobre o facto de que o ISR seja a tecnologia ideal para aplicações mais complexas [10] que possuam algumas páginas que possam ser alvo de algumas alterações pequenas.

Tal como mencionado acima na secção dos trabalhos relacionados [2], o SEO mais elevado permite que aplicações *web* que utilizem esta tecnologia sejam favorecidas quando o utilizador efetua pesquisas no *browser*, aparecendo assim nas primeiras pesquisas e consequentemente conseguirão obter mais tráfego e utilizadores.

2.3 Sumário

Este capítulo está dividido em 2 secções que são os trabalhos relacionados e as tecnologias existentes e destaca o facto de ser necessário investigar e explorar sobre o funcionamento e evolução das várias tecnologias para se saber qual é a mais adequada. Caso contrário não teria sido possível escolher as tecnologias e desenvolver o projeto.

A primeira secção serviu para elucidar o leitor sobre a evolução das tecnologias que foram utilizadas no desenvolvimento do projeto e de outras tecnologias relacionadas, a sua aplicação no quotidiano e as implicações, quer positivas ou negativas, que possam ter no desempenho das aplicações em que são aplicadas.

A segunda secção teve como objetivo descrever as tecnologias utilizadas no projeto e as razões que motivaram a sua utilização juntamente com tecnologias relacionadas que são utilizadas em cenários semelhantes. Esta descrição incidiu sobre as vantagens e desvantagens assim como o seu funcionamento. No final foi feita uma comparação recorrendo a vários parâmetros de desempenho e usabilidade para que o leitor possa entender melhor como essas tecnologias realmente funcionam e as razões da sua utilização no projeto.

3 Análise e desenho da solução

Neste capítulo serão apresentados e explorados os requisitos funcionais e não funcionais que foram definidos com o supervisor. Para além dos requisitos definidos, foi também permitida alguma autonomia no desenvolvimento da aplicação para adicionar novas funcionalidades e requisitos que permitissem melhorar a solução final.

3.1 Domínio do problema

A solução apresentada consiste na criação de um *website* responsável por gerir as marcações de reuniões por parte de um funcionário registado na organização para a qual o projeto foi desenvolvido.

Este projeto irá explorar a aplicação de CDNs com o objetivo de melhorar a distribuição de conteúdos e aumentar a velocidade e fiabilidade do *website* desenvolvido. O ISR também será explorado pois permite a atualização partes do *website* sem a necessidade de regenerar todo o conteúdo, aumentando assim a eficiência.

Inicialmente foi identificado o ator do sistema que é o **utilizador** do website que possui uma **conta** afiliada à **organização**. Após efetuar login com as suas credenciais da **organização** e ter acesso a todas as suas funcionalidades, o **utilizador** marca uma **reunião** a ser realizada em uma determinada **sala**.

Na figura 3, é apresentado o modelo de domínio que permite uma melhor compreensão do problema.

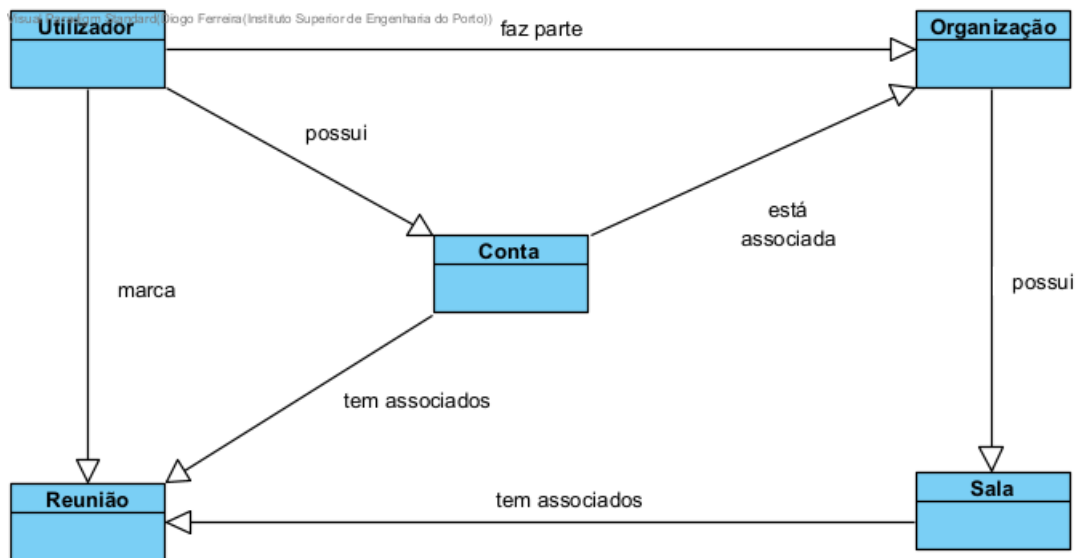


Figura 3 - Modelo de Domínio

3.2 Requisitos funcionais e não funcionais

Nesta secção são apresentados e explicados os requisitos funcionais e não funcionais da aplicação que foram sujeitos a alterações ao longo do projeto no sentido de tornar o projeto mais completo e realista.

3.2.1 Requisitos não funcionais

Os requisitos não funcionais não influenciam o funcionamento da aplicação, pelo que o resultado esperado irá ser sempre devolvido pelo sistema, e descrevem como é que os requisitos funcionais devem ser aplicados tendo em conta as limitações do sistema. Ambos os tipos de requisitos são importantes embora o sistema não irá funcionar corretamente se não atender aos requisitos funcionais.

Neste projeto, podem-se identificar os seguintes requisitos não funcionais:

- Utilização de uma base de dados *online* MongoDB para armazenamento dos dados;
- Utilização da *framework* *Next.js* na realização deste projeto;
- Aplicação da linguagem *JavaScript* na elaboração do *backend* onde se encontra o CRUD e que efetua a comunicação com a API e a base de dados;
- Utilização da linguagem *Typescript* na parte do *frontend*;

- Utilização do *Microsoft 365* para efetuar o *login* no website com a conta da organização;
- Utilização do ISR para a busca de dados a partir das APIs e consequente criação de tabelas.

3.2.2 Requisitos funcionais

Os requisitos funcionais definem como o sistema se deve comportar e consistem nas especificações e funcionalidades que o sistema deve ser capaz de executar tendo em conta as necessidades do utilizador. Estes requisitos são visíveis pelo utilizador da aplicação.

Durante o desenvolvimento do estado da arte e da análise da aplicação foram levantados vários requisitos que foram sujeitos a alterações ao longo do desenvolvimento do projeto. Para esse efeito, foram efetuadas várias reuniões com o supervisor deste projeto no sentido de o mesmo dar o seu *feedback* sobre o trabalho desenvolvido e sugerir as alterações necessárias. Estas reuniões foram necessárias para um desenvolvimento consistente e eficaz do projeto que cumprisse com os requisitos pedidos e para a atualização dos requisitos no sentido de o projeto ser o mais desenvolvido e completo possível.

Os requisitos funcionais que foram identificados ao longo do projeto e que aparecem representados na figura na página seguinte são os seguintes:

- Efetuar login;
- Listar informação sobre as salas;
- Listar reuniões;
- Criar reunião;
- Editar reunião;
- Eliminar reunião.

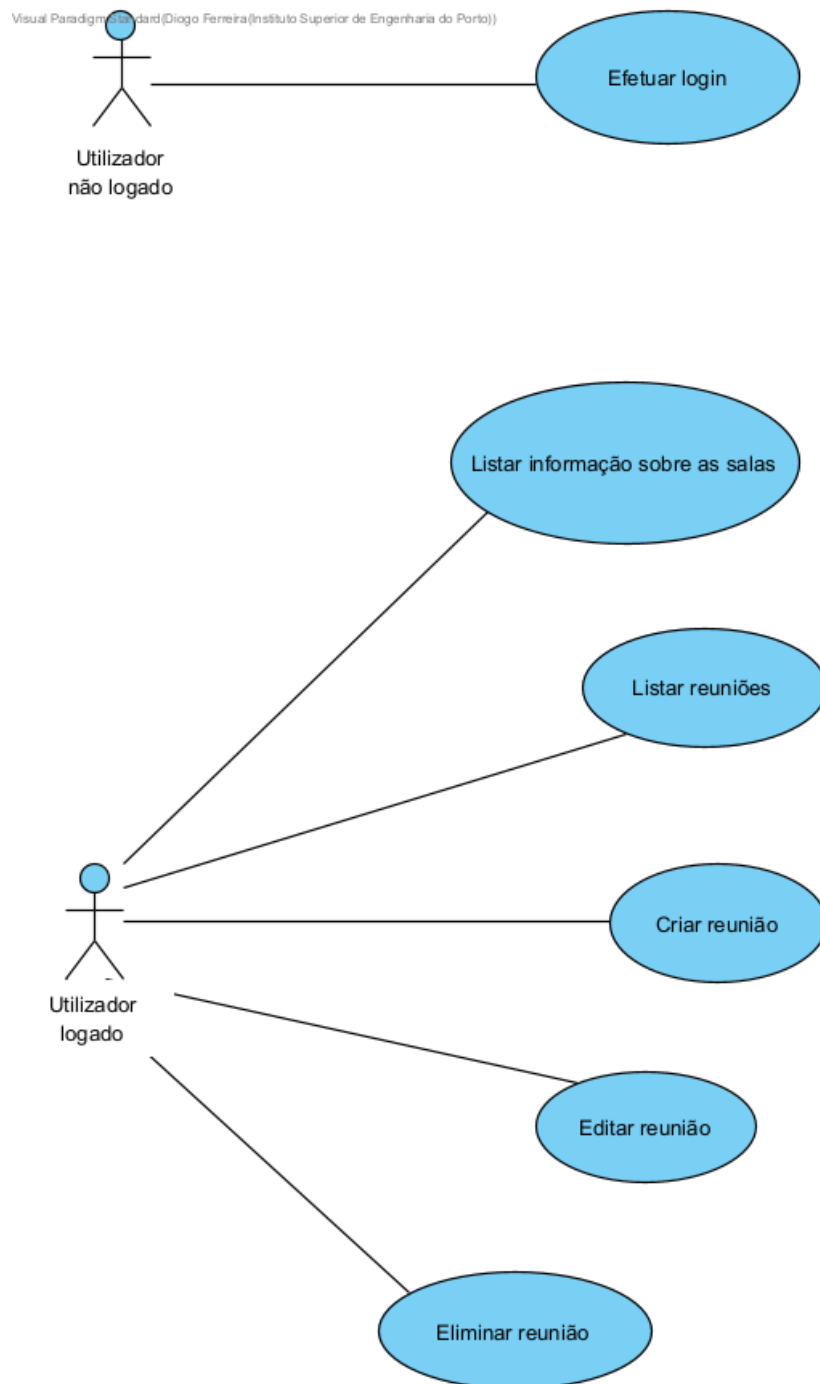


Figura 4 - Diagrama de Casos de uso

O utilizador tem de estar registado na organização e necessita de efetuar o *login* para aceder à aplicação e poder usufruir das suas funcionalidades que aparecem associadas ao utilizador que já efetuou o *login*, tal como aparece no diagrama de casos de uso na figura 4.

As operações CRUD aplicam-se apenas às reuniões, enquanto no caso das salas será possível listar as mesmas juntamente com informação associada ao seu estado atual de ocupação e ao número de reuniões marcadas que irão decorrer numa determinada sala.

3.2.2.1 Efetuar Login

O utilizador já registado na organização efetua o *login* para poder aceder à aplicação e o sistema solicita que este insira os dados necessários. Depois da inserção dos dados, o sistema valida-os e informa o utilizador sobre o resultado da operação.

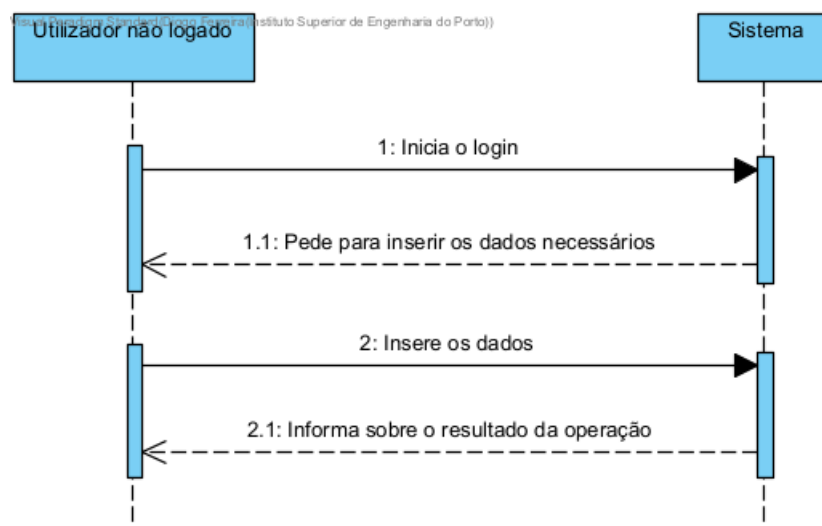
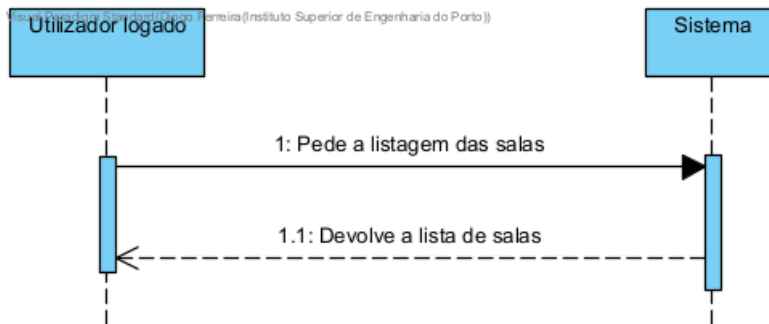


Figura 5 – SSD Efetuar Login

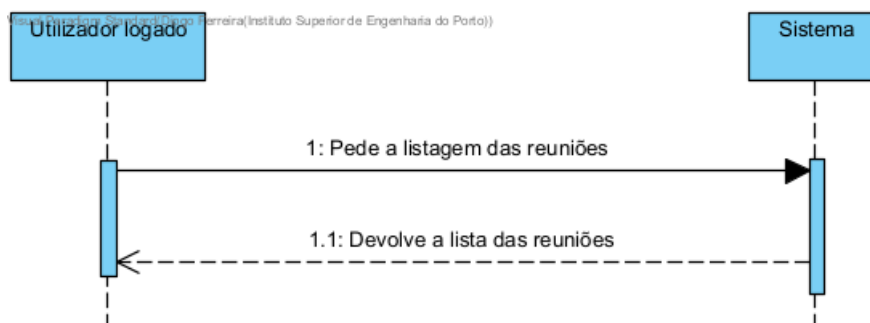
3.2.2.2 Listar Salas

O processo de listagem de salas é iniciado pelo utilizador que já efetuou o *login* e o sistema devolve as informações relacionadas com as salas depois de o utilizador ter efetuado o pedido.

*Figura 6 – SSD Listar Salas*

3.2.2.3 Listar Reuniões

O utilizador que efetuou previamente o *login* para poder efetuar esta e outras funcionalidades da aplicação, pede ao sistema para devolver a lista de reuniões que foram marcadas pelo mesmo. O sistema devolve-lhe a lista de reuniões.

*Figura 7 – SSD Listar Reuniões*

3.2.2.4 Criar Reunião

O utilizador que efetuou previamente o *login* para poder efetuar esta e outras funcionalidades da aplicação, dá início ao processo de criação de uma nova reunião. O sistema exige que o utilizador insira os dados necessários. Após o utilizador ter inserido os dados, o sistema valida os dados e informa-o sobre o sucesso da operação.

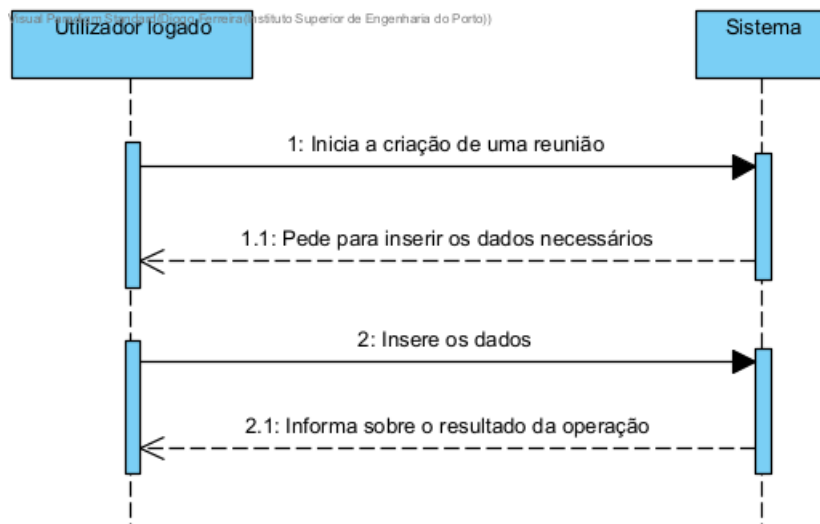


Figura 8 – SSD Criar Reunião

3.2.2.5 Editar Reunião

O utilizador que efetuou previamente o *login* para poder efetuar esta e outras funcionalidades da aplicação, dá início ao processo de edição de uma reunião. O sistema lista todas as reuniões que foram anteriormente marcadas pelo utilizador para que o mesmo possa escolher uma delas para ser editada. Após a escolha da reunião, o sistema exige os dados necessários para a edição da mesma. Após o utilizador ter editado os dados necessários, o sistema valida os dados e informa-o sobre o sucesso da operação.

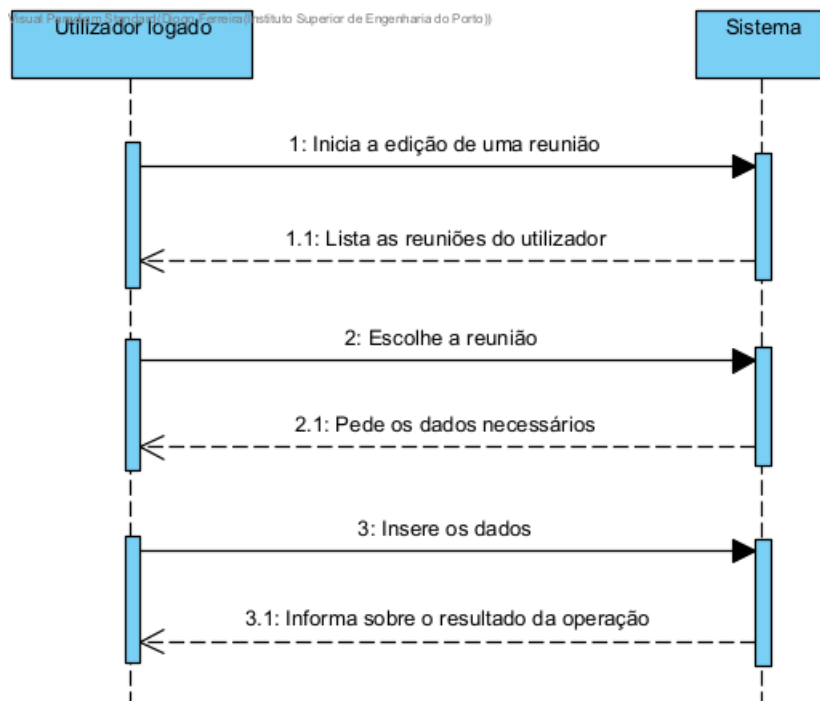


Figura 9 – SSD Editar Reunião

3.2.2.6 Eliminar Reunião

O utilizador que efetuou previamente o *login* para poder efetuar esta e outras funcionalidades, dá início ao processo de eliminação de uma reunião. O sistema lista todas as reuniões que foram anteriormente marcadas pelo utilizador para que o mesmo possa escolher a reunião que irá ser eliminada. Após a escolha da reunião, o sistema pergunta ao utilizador, através de um pop-up na aplicação, se ele deseja eliminar essa reunião. Caso a resposta seja afirmativa, o sistema apaga a reunião. Independentemente da resposta do utilizador, o mesmo é informado sobre o sucesso da operação.

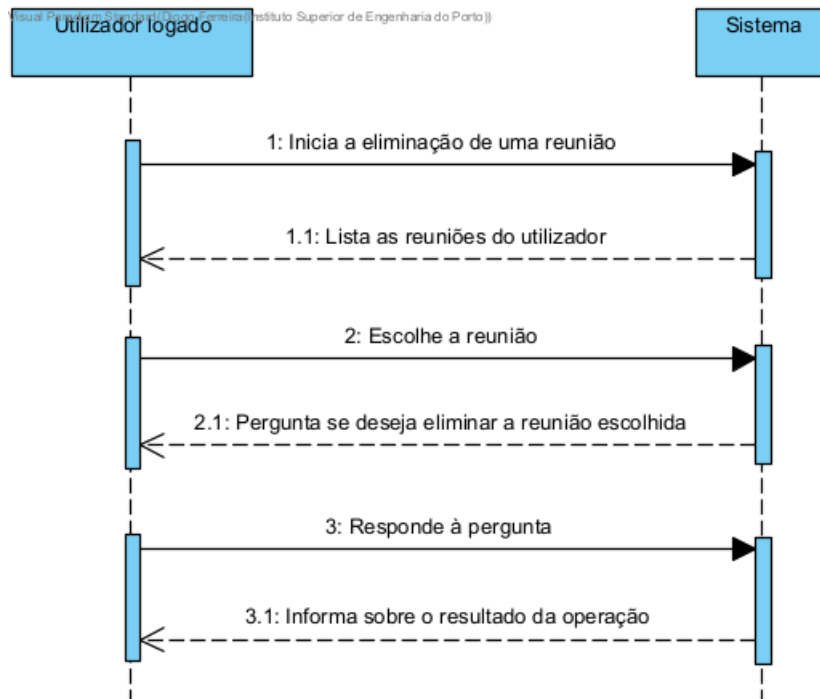


Figura 10 – SSD Eliminar Reunião

3.3 Desenho da solução

Na figura 11 está representado o diagrama ER da base de dados. A reunião é constituída pelo seu id que é automaticamente atribuído pela base de dados aquando da sua criação, sendo uma *primary key* e assumindo um valor único. A reunião também possui um assunto e organizador que são do tipo *string*, a data, o horário de começo e horário de fim que são variáveis do tipo *data*, o open é do tipo *string* e serve para verificar se a reunião vai ser pública ou privada. O *room_id* é uma *foreign key* que devolve o id da sala onde a reunião vai ser realizada.

A sala possui um id que é uma *primary key* foi automaticamente atribuído pela base de dados aquando da sua inserção na base de dados e que é único. A sala também possui um nome, designação e email que são variáveis do tipo *string*.

Existe uma relação de um para muitos entre a sala e a reunião pois uma sala pode albergar várias reuniões e uma reunião é marcada apenas numa determinada sala.

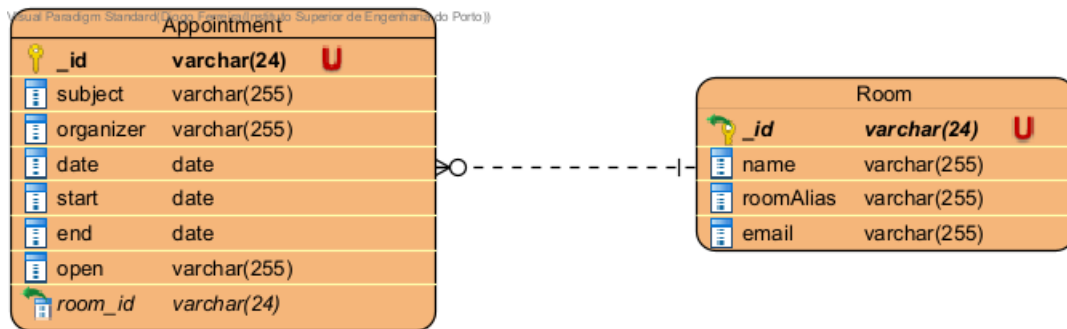


Figura 11 – Diagrama ER da base de dados

A figura 12 é uma representação da base de dados Mongo DB onde são armazenados os dados. A base de dados possui 2 coleções distintas que são a lista de reuniões e a lista de salas. A coleção das salas foi criada aquando da criação da base de dados pois as suas informações não vão ser alteradas e já existiam anteriormente em outra API. A coleção das reuniões irá armazenar as salas que forem adicionadas ou editadas na aplicação.

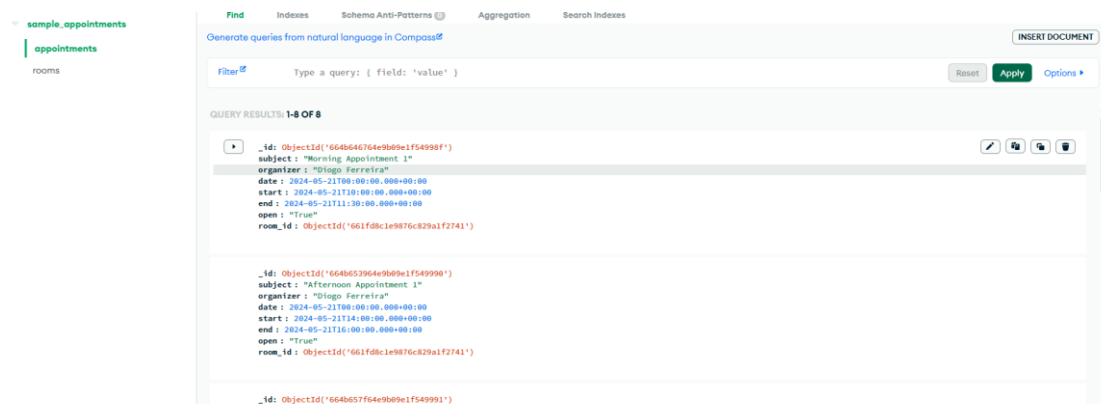


Figura 12 – Base de dados Mongo DB

Os diagramas de componentes e a vista física vão ser exibidos e explicados para que se possa entender a estrutura do projeto.

Na figura 13, encontra-se representado o diagrama de vista lógica de nível 1. O componente PESTI representa o projeto todo que requer a API do Azure AD que foi responsável pela criação do login através do *Microsoft 365* para que o utilizador que já possui conta na organização, neste caso o UI1, possa aceder à aplicação.

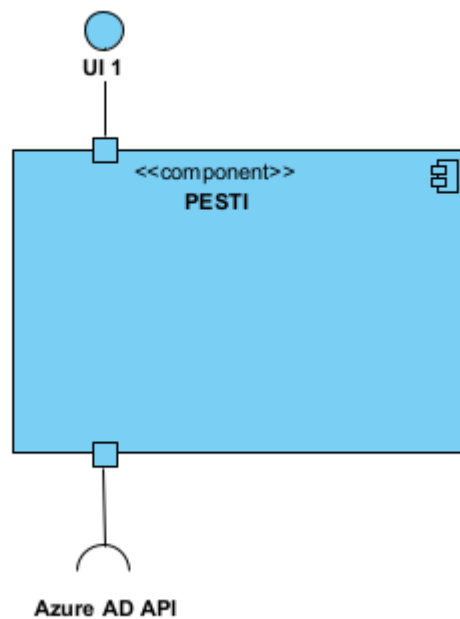


Figura 13 - Diagrama de vista lógica de nível 1

Na figura 14, encontra-se representado o diagrama de vista lógica de nível 2. O componente PESTI representa o projeto todo contém pastas que são a *dashboard*, *ui*, *lib* e a *api*. A pasta *dashboard* que representa as diferentes páginas do projeto requer a API do Azure AD que foi responsável pela criação do login através do *Microsoft 365* para que o utilizador que já possui conta na organização, neste caso o UI1, possa aceder à aplicação.

A pasta *api*, que representa o *backend* da aplicação, armazena as informações sobre as salas e as reuniões provenientes da base de dados em APIs separadas que vão requeridas pela pasta *lib* onde os dados vão ser tratados para serem exibidos nas páginas da aplicação. A pasta *ui* trata da disposição dos vários componentes da página e por vezes vão utilizar os dados definidos na pasta *lib*. Os dados da pasta *lib* por vezes são encaminhados diretamente para as páginas da aplicação.

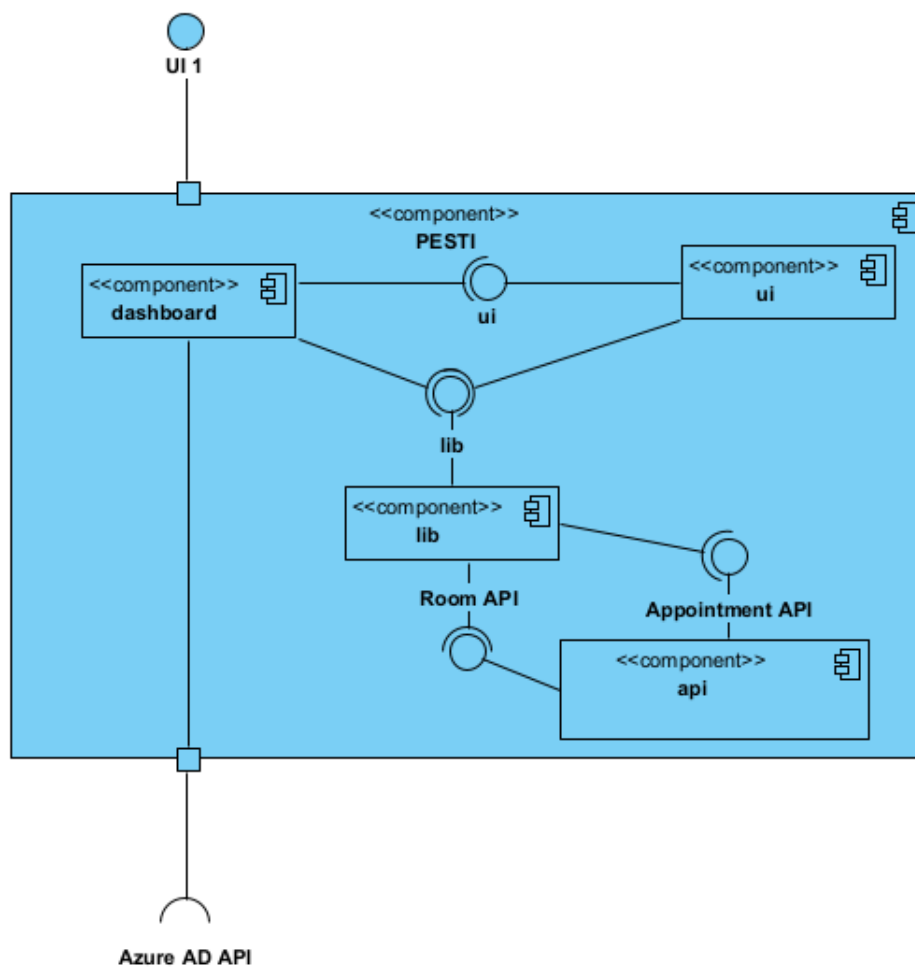


Figura 14 - Diagrama de vista lógica de nível 2

Na figura 15, temos um diagrama de vista lógica de nível 2 alternativo. Tendo em conta que o processo de *login* foi abordado vagamente no diagrama de vista lógica de nível 1 e que o utilizador precisa de efetuar *login* com as credenciais da organização para poder ter acesso às funcionalidades da aplicação, foi elaborado um diagrama de vista lógica de nível 2 que irá salientar a importância do *login*, analisando assim todo o processo do *login* mais detalhadamente e de uma forma mais ampla que no diagrama de vista lógica de nível 1.

A pasta “/api/auth/nextauth” que está localizada no *backend* contém um ficheiro onde estão as informações necessárias para o *login* e que é requerido da API do Azure AD onde foram criadas essas configurações. A pasta *dashboard* representa as páginas da aplicação e o utilizador necessita de inserir as suas credenciais para poder aceder às mesmas. Para esse efeito, esta pasta requer a informação a partir da API *login*.

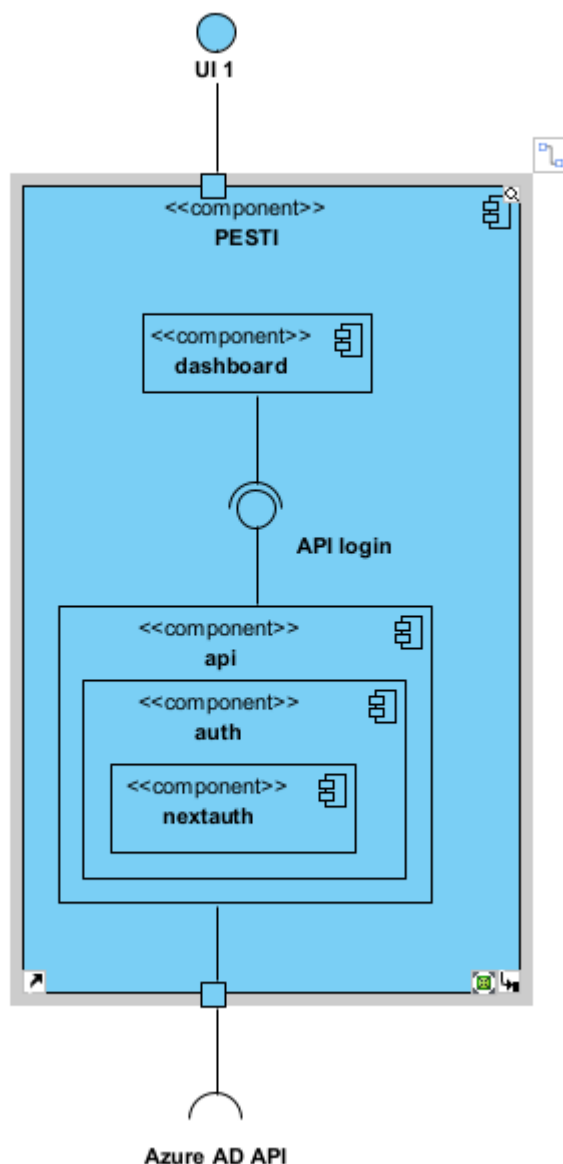


Figura 15 - Diagrama de vista lógica de nível 2 alternativo

Na figura 16, encontra-se representado o diagrama de vista lógica de nível 3. O componente PESTI representa o projeto todo contém pastas que são a *dashboard*, *ui*, *lib* e a *api*. A pasta *dashboard* que representa as diferentes páginas do projeto requer a API do Azure AD que foi responsável pela criação do login através do *Microsoft 365* para que o utilizador que já possui conta na organização, neste caso o UI1, possa aceder à aplicação. A pasta *dashboard* contém as pastas (*overview*) e *appointments* que representam as secções do *dashboard* onde estão representadas as salas e as reuniões.

A pasta *api* contém 2 pastas que são a *rooms* e a *appointments* que representam as respectivas APIs, onde estão armazenados separadamente os dados relativos às salas e às reuniões, que vão requeridas pela pasta *lib* onde os dados vão ser tratados para serem exibidos nas páginas da aplicação.

A pasta *ui* trata da disposição dos vários componentes da página e por vezes vão utilizar os dados definidos na pasta *lib*. Os dados da pasta *lib* por vezes são encaminhados diretamente para as páginas da aplicação. A pasta *ui* apresenta 2 pastas que são a *rooms* e a *appointments* que podem utilizar os dados vindo da pasta *lib* que representam a disposição dos itens nas páginas associadas às salas e às reuniões respetivamente.

Os dados da pasta *ui* podem ser utilizados individualmente em cada uma das pastas dentro da pasta *dashboard* ou em toda a pasta, aparecendo a disposição definida em todas as páginas.

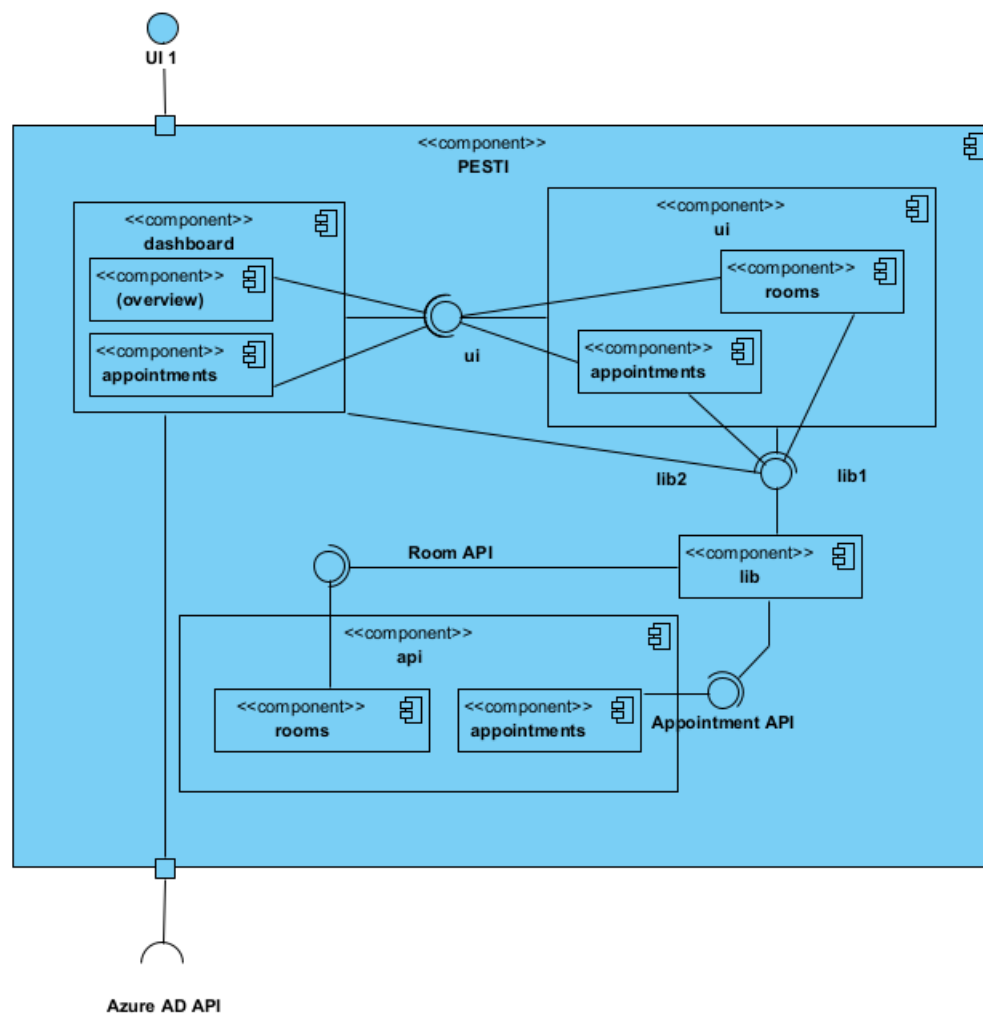


Figura 16 - Diagrama de vista lógica de nível 3

Na figura 17, temos um diagrama de vista lógica de nível 3 alternativo. Este diagrama foi adotado pois também é importante analisar em maior detalhe e de forma mais ampla todo o processo de obtenção dos dados a partir da base de dados, passando pelas respetivas APIs e terminando no *frontend*, onde vão ser mostrados na aplicação e sem o qual a aplicação não teria o seu propósito. O facto da obtenção de dados a partir da base de dados ter sido analisada de forma mais superficial em um dos diagramas de vista lógica de nível 2 também influenciou a decisão.

Neste diagrama começa por ser retratada a base de dados como uma pasta onde estão armazenados os dados e que fornece esses dados à pasta do mongodb que estabelece a conexão à base de dados para que depois a pasta api possa utilizar esses dados.

Na pasta model são definidos o formato das reuniões e das salas a serem tratadas nas respetivas APIs, onde serão requeridos os dados da pasta model relativos ao formato das instâncias de reuniões e salas para serem listados na API. Posteriormente, a pasta api comunica com a API que representa o frontend que se chama api para pode enviar e receber informações sobre as reuniões e salas. Esta abordagem também foi considerada pois o tratamento dos dados no *backend* e na base de dados é essencial para o funcionamento correto da aplicação no *frontend*.

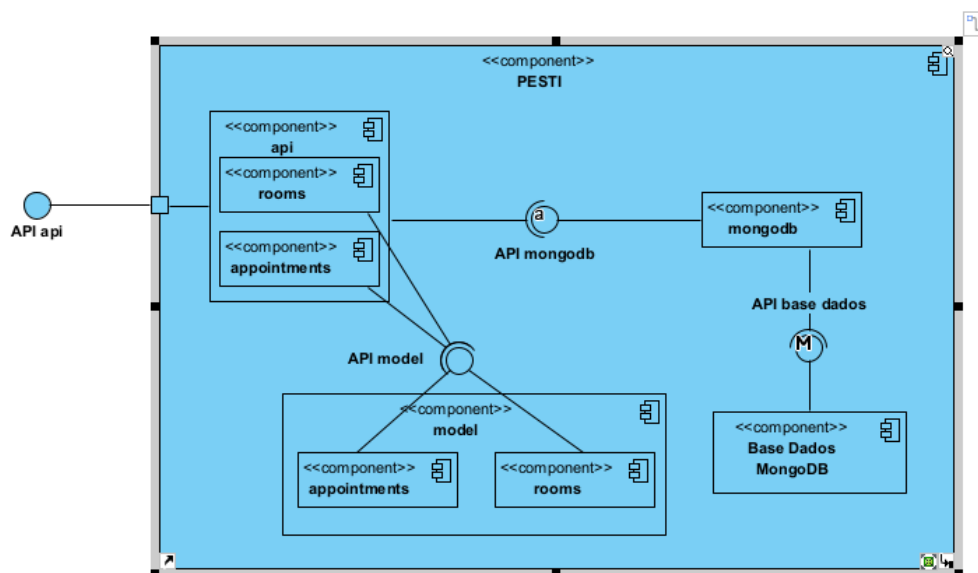


Figura 17 - Diagrama alternativo de vista lógica de nível 3

Na figura 18 temos o diagrama da vista física do projeto. A base de dados está albergada em uma base de dados MongoDB que se encontra online e cujo número de porta é 27017 que o padrão. As APIs comunicam com a base de dados através de requisições HTTP e encontram no caminho `"/app/api"` dentro do projeto com `"/rooms"` para as salas e `"/appointments"` para as reuniões, sendo esses caminhos utilizados no URL. O frontend do projeto que se encontra em várias pastas dentro da pasta raiz `"/app"` também comunica com os endpoints das APIs através de requisições HTTP para obter e tratar dos dados lá armazenados e listá-los nas tabelas de salas e reuniões ou para enviar requisições HTTP com novos dados para as APIs. O frontend encontra-se no URL `localhost:3000`, embora existam mais ramificações para cada caso de uso.

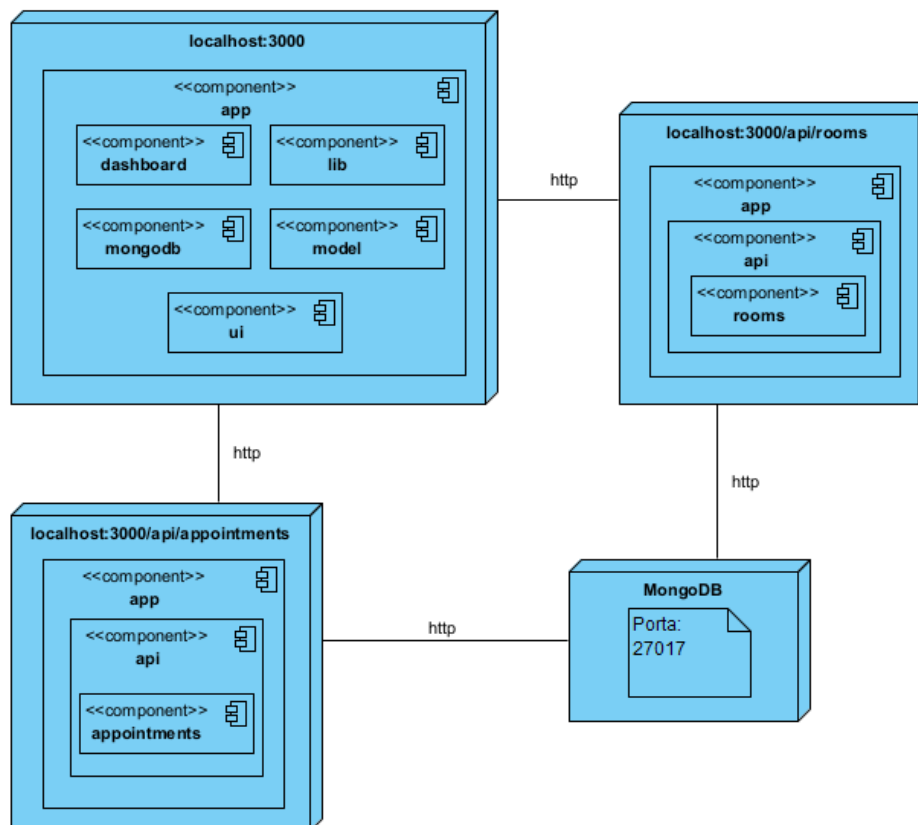


Figura 18 - Diagrama de vista física

Na figura 19, temos um diagrama de vista física alternativo que é idêntico ao diagrama da figura 18 embora com uma diferença em que estão incluídos componentes relacionados com o *login*. A parte relacionada com o login foi adicionada pois é necessário que o utilizador já registado na organização efetue login para poder ter acesso às funcionalidades da aplicação

e para que tudo o resto, ou seja, a aquisição de dados provenientes da base de dados e o seu subsequente tratamento para que sejam mostrados na aplicação e a alteração dos dados existentes e o seu envio para a base de dados aconteça.

As configurações do *login* foram criadas *online* em uma API do Azure AD cuja porta é a 443 e comunica via HTTPS, que permite a comunicação segura, com um ficheiro dentro da pasta *"/app/api/auth/nextauth"* onde se encontram as configurações que permitam o *login* por aquele utilizador específico que se encontra registado na organização. Esse componente está ligado à pasta *"/app/dashboard"* pois permite a navegação pelas páginas que utilizam os outros componentes para obter e enviar informações para as APIs e base de dados, após o *login* bem-sucedido.

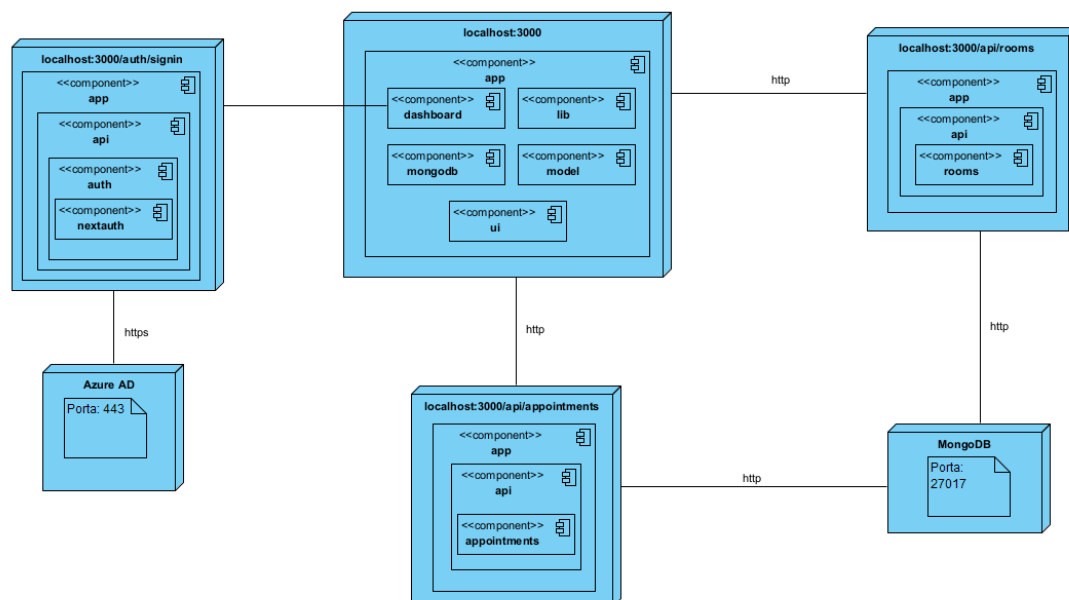


Figura 19 - Diagrama de vista física alternativo

4 Implementação da Solução

Neste capítulo será apresentada uma descrição da forma como foi aplicada a solução final, tendo como base o design apresentado no capítulo anterior. Também será abordada a forma como as tecnologias discutidas no capítulo do Estado da Arte foram implementadas no projeto.

Os testes como forma de garantir o correto funcionamento do website também serão abordados.

4.1 Descrição da implementação

O projeto descrito neste relatório foi realizado com recurso às linguagens *JavaScript* no *backend* e *Typescript* no *frontend*, ao *Next.js* que é uma framework do *React* e ao *MongoDB* que serviu para a criação da base de dados. Para se adquirirem conhecimentos sobre o *Next.js* foi efetuado um tutorial [12] que serviu de base para o projeto embora com as respetivas adaptações necessárias para os requisitos pedidos pela organização.

4.1.1 Base de dados

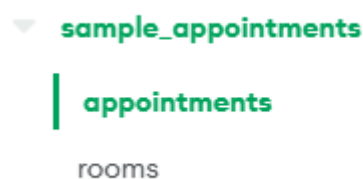


Figura 20 – Esquema da base de dados no MongoDB

Sample_appointments representa o nome da base de dados do Mongo DB. *Appointments* é uma coleção onde estão armazenados todos as reuniões enquanto *rooms* é a outra coleção onde está armazenada a lista de salas.

```
_id: ObjectId('664b646764e9b09e1f54998f')
subject : "Morning Appointment 1"
organizer : "Diogo Ferreira"
date : 2024-05-21T00:00:00.000+00:00
start : 2024-05-21T10:00:00.000+00:00
end : 2024-05-21T11:30:00.000+00:00
open : "True"
room_id : ObjectId('661fd8c1e9876c829a1f2741')
```

Figura 21 – Exemplo de reunião na base de dados

Relativamente à reunião temos as seguintes variáveis:

- `_id`: esta variável é criada automaticamente pela base de dados quando uma reunião é criada e é do tipo *ObjectId*;
- `subject`: é uma variável do tipo *String* e representa o assunto da reunião;
- `organizer`: é uma variável do tipo *String* e representa o nome do organizador da reunião;
- `date`: é uma variável do tipo *Date*, encontra-se no formato ISO e representa a data da reunião;
- `start`: é uma variável do tipo *Date*, encontra-se no formato ISO e representa a hora do início da reunião;
- `end`: é uma variável do tipo *Date*, encontra-se no formato ISO e representa a hora do fim da reunião;
- `open`: é uma variável do tipo *String*;
- `room_id`: é uma variável do tipo *ObjectId* que representa o id da sala na qual a reunião vai ser realizada.

O `start` e o `end` são variáveis do tipo *Date* para se poder comparar as 2 variáveis aquando da criação de uma reunião e verificar-se se a hora do `end` é posterior à hora do `start`.

```
_id: ObjectId('661fd8c1e9876c829a1f2741')  
name : "Arkanoid Room (1..3)"  
roomAlias : "arkanoid-room-(1..3)"  
email : "arkanoid@devscope.net"
```

Figura 22 – Exemplo de sala na base de dados

Relativamente à sala na base de dados temos as seguintes variáveis:

- `_id`: esta variável foi criada automaticamente aquando da inserção das salas na base de dados e é do tipo *ObjectId*;
- `name`: é uma variável do tipo *String* e representa o nome da sala;
- `roomAlias`: é uma variável do tipo *String* e representa o identificador da sala;
- `email`: é uma variável do tipo *String* e representa o email associado à sala.

As salas já existiam em uma outra API e portanto já estavam definidas, não vão ser alteradas. O seu `_id` vai ser utilizado para ser associado às reuniões.

4.1.2 Conexão à base de dados e API

Para que as operações CRUD se reflitam na base de dados, foi necessário criar uma conexão à base de dados no *Visual Studio Code*. Para isso, foi necessário conectar e inserir a *string* de conexão.

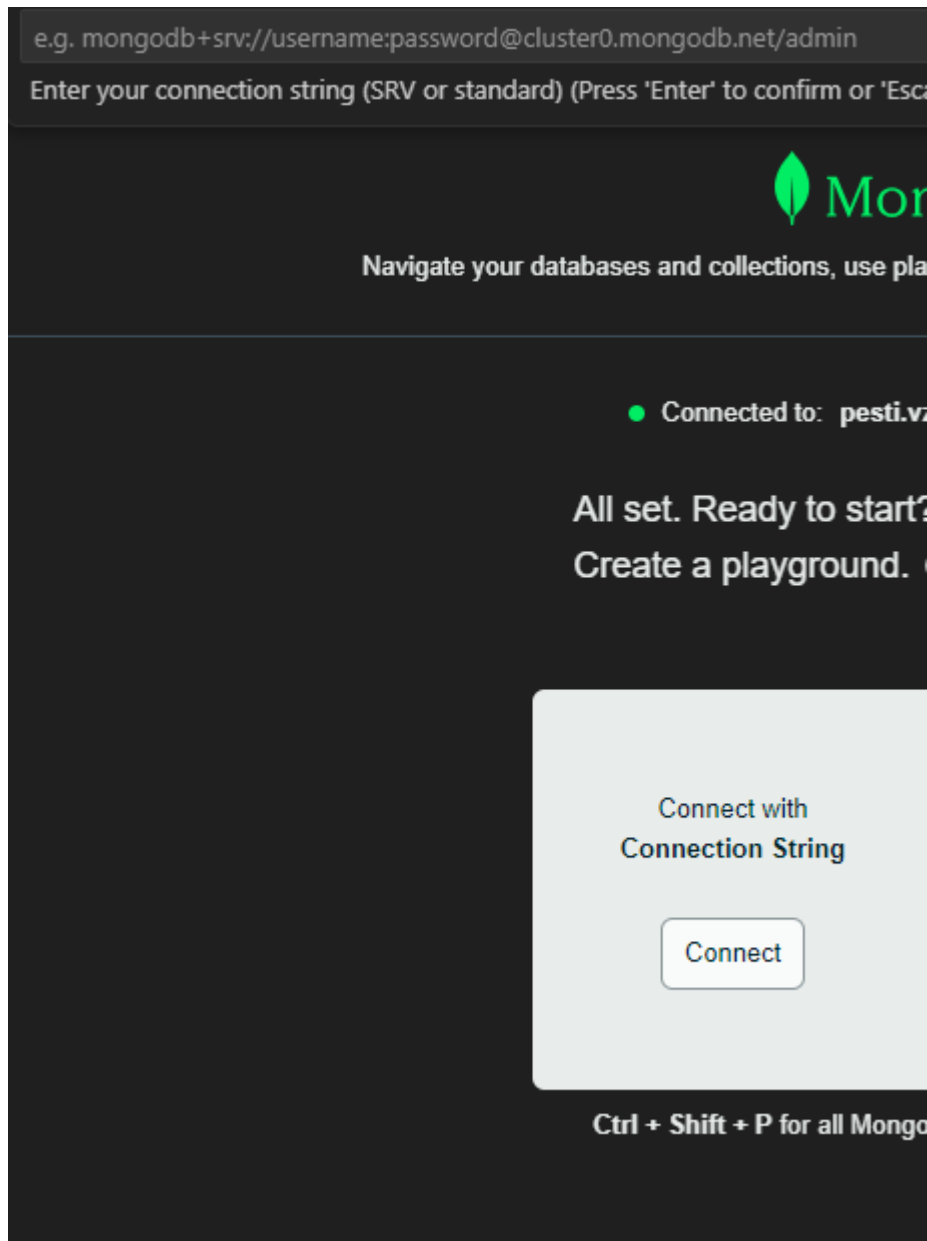


Figura 23 - Conexão à base de dados no Visual Studio Code

Quando a conexão é realizada com sucesso, aparece uma representação da base de dados na aplicação. Esta conexão deve estar ligada quando a aplicação é executada para que potenciais operações CRUD que sejam realizadas, apareçam na base de dados.

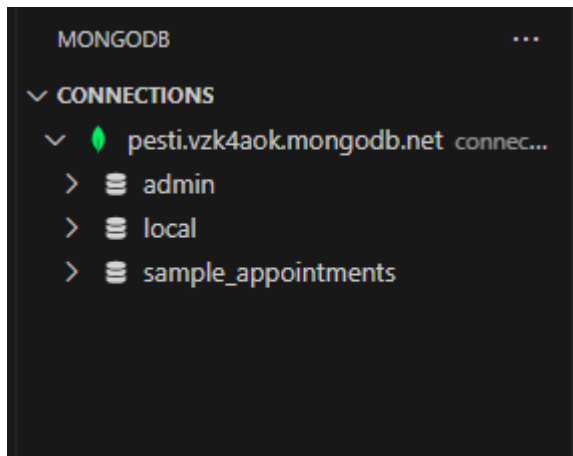


Figura 24 - Base de dados na aplicação

A *string* de conexão, que está armazenada numa pasta secreta chamada `.env`, irá ser utilizada para conectar a aplicação à base de dados e possui a seguinte estrutura:

- Nome de utilizador do MongoDB que está representado por `diogoferreira`;
- *Password* do MongoDB que vem a seguir em que o `@` da *password* aparece representado como `%40` por causa da codificação de URL;
- O nome do *cluster* que aparece no site onde se encontra a base de dados;
- A seguir à barra final aparece o nome da base de dados que neste caso é `sample_appointment`;

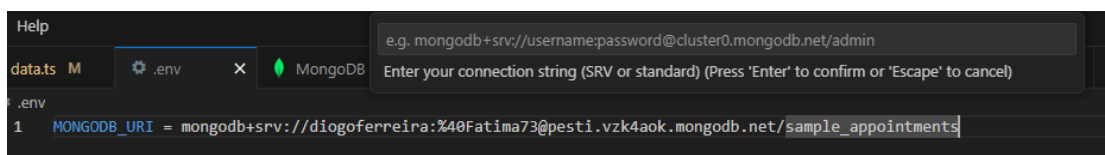


Figura 25 - String de conexão

A classe `mongodb.js` que se encontra em uma pasta própria com o mesmo nome possui a estrutura que está representada na figura 26.

```
import mongoose from "mongoose";

const connectMongoDB = async () => {
  try {
    mongoose.connect(process.env.MONGODB_URI);
    console.log("Connected to MongoDB.");
  } catch (error) {
    console.log(error);
  }
}

export default connectMongoDB;
```

Figura 26 - Classe *mongodb.js*

Esta classe recorre ao pacote “mongoose” que é uma biblioteca do Next.js para modelar bases de dados baseadas em MongoDB.

De seguida, foi criada uma função assíncrona com o nome *connectMongoDB* que é responsável por estabelecer conexão com a base de dados através da *string* de conexão definida como a constante *MONGODB_URI*. A função possui um bloco *try-catch* onde dentro do bloco *try* tenta estabelecer a conexão com a base de dados através da *string* de conexão. Se for bem-sucedido, imprime o *console.log* imediatamente abaixo no terminal durante a execução. Caso contrário, passa para o *catch* e o erro em questão é exibido no terminal.

A função é exportada para que possa ser utilizada em outras classes.

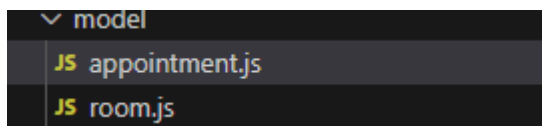


Figura 27 - Pasta *model*

Na pasta *model* encontram-se 2 classes com o nome das coleções na base de dados. Estas classes servem para definir separadamente modelos *Mongoose* para as respetivas coleções na base de dados.


```
import mongoose, { Schema } from "mongoose";

const appointmentSchema = new Schema(
  {
    subject: { type: String, required: true },
    organizer: { type: String, required: true },
    date: { type: Date, required: true },
    start: { type: Date, required: true },
    end: { type: Date, required: true },
    open: { type: String, required: true },
    room_id: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Room',
      required: true
    }
  },
  {
    versionKey: false,
    collection: "appointments"
  }
);

const Appointment = mongoose.models.Appointment || mongoose.model("Appointment", appointmentSchema);

export default Appointment;
```

Figura 28 - Appointment.js

O *mongoose* e o *Schema*, que é um construtor, são utilizados para modelar bases de dados e facilitar as interações com o MongoDB em aplicação que utilizam *Node.js*.

Abaixo, foi criado um esquema com o nome *AppointmentSchema* que irá definir a estrutura da coleção na base de dados. Os campos são definidos pelo tipo da variável e pela sua obrigatoriedade, representada pelo campo *'required'*. Na variável *room_id*, que é do tipo *ObjectId*, foi feita referência à coleção *Room* e neste caso representará o id da sala em que a reunião vai ser realizada. Isto é necessário para que o nome dessa sala apareça na tabela das reuniões como vai ser abordado.

A primeira configuração adicional serve para desativar a criação de uma chave de versão, representada por *'_v'* e que aparece quando é criada uma reunião fora da base de dados quando a *versionKey* não está a false como aqui. A outra especifica o nome da coleção do MongoDB onde os dados e as respetivas configurações serão armazenados.

O *Appointment* é definido como modelo *Mongoose*. A existência do modelo é verificada. Caso já exista, o existente é reutilizado. Caso contrário, será criado um modelo com base no esquema definido acima.

O *Appointment* é exportado para ser utilizado em outras partes da aplicação.

```
import mongoose, { Schema } from "mongoose";

const roomSchema = new Schema(
  {
    name: String,
    roomAlias: String,
    email: String,
  },
  {
    collection: "rooms"
  }
);

const Room = mongoose.models.Room || mongoose.model("Room", roomSchema);

export default Room;
```

Figura 29 - Room.js

Na classe *Room.js* são aplicados os mesmos princípios, embora de uma forma mais simples que é evidenciada pela ausência de referências a outras coleções e de configurações adicionais além da *collection* após a definição dos campos da sala. Isto deve-se ao facto de a lista das salas ser constante e não ser sujeita a operações CRUD tal como as reuniões.

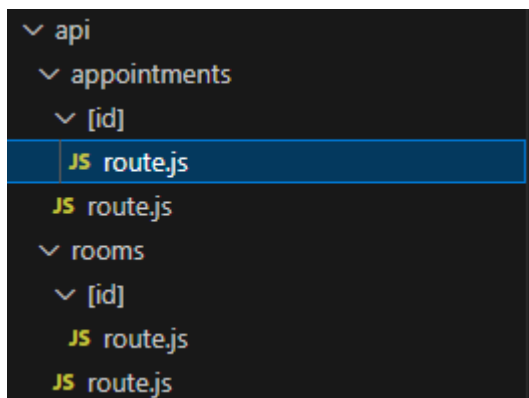


Figura 30- Pasta API com os ficheiros

A pasta *api* do projeto serve de backend e contém as operações CRUD para as reuniões, e no caso das salas, apenas para as funções do tipo GET. Esta pasta também serve de comunicação entre o *frontend*, que é onde serão tratados os requisitos funcionais, e a base de dados.

Ambas as pastas que se encontram dentro da pasta *api* contém 2 ficheiros chamados *route.js* que estão escritos em *JavaScript*. O ficheiro dentro da pasta *[id]* que se encontra

dentro das pastas `appointments` e `rooms` serve para operações relacionadas com o tratamento de dados de uma reunião ou sala com um determinado id. O outro ficheiro serve para operações relacionadas com a lista de reuniões ou salas em geral.

```
import connectMongoDB from "@app/mongodb/mongodb";
import Appointment from "@app/model/appointment";
import { NextResponse } from "next/server";

export async function POST(request) {
  const { subject, organizer, date, start, end, open, room_id } = await request.json();
  await connectMongoDB();
  await Appointment.create({ subject, organizer, date, start, end, open, room_id });
  return NextResponse.json({ message: "Appointment Created" }, { status: 201 });
}

export async function GET() {
  await connectMongoDB();
  const appointments = await Appointment.find();
  return NextResponse.json(appointments);
};

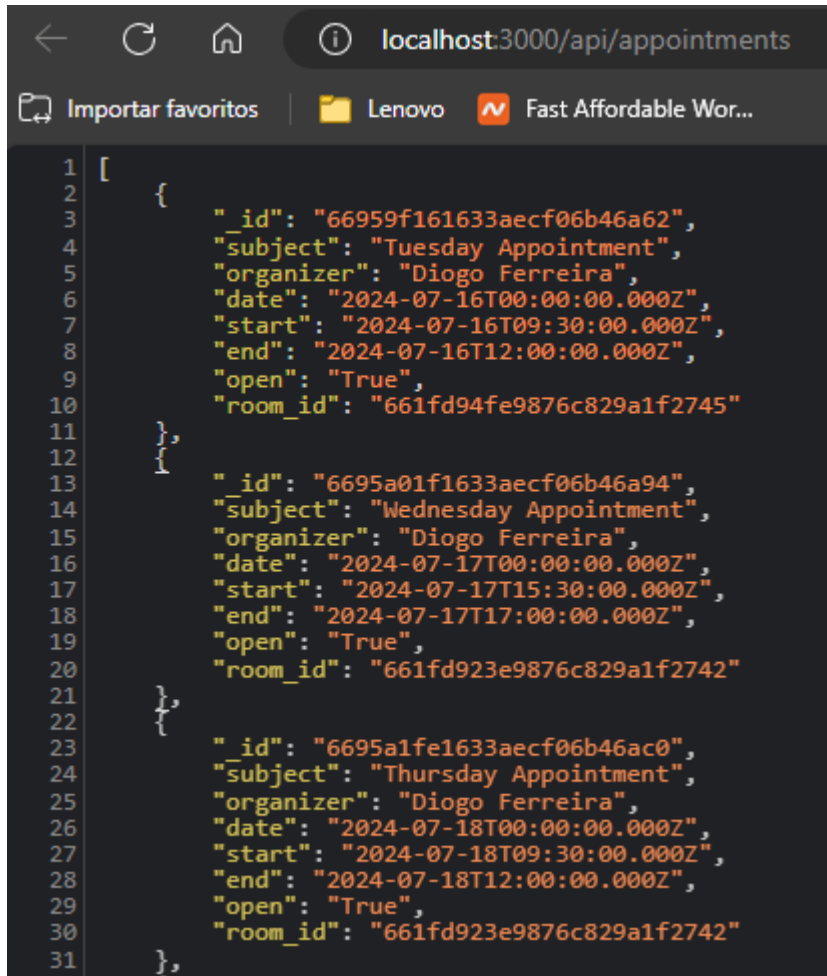
export async function DELETE(request) {
  const id = request.nextUrl.searchParams.get("_id");
  await connectMongoDB();
  await Appointment.findByIdAndDelete(id);
  return NextResponse.json({ message: "Appointment deleted" }, { status: 200 });
}
```

Figura 31 – Ficheiro route.js no caminho "/api/appointments"

O ficheiro representado na figura 31 e que se encontra no caminho geral contém as funções POST, GET e DELETE.

A função POST serve para criar uma reunião. Quando o cliente efetua uma requisição HTTP como POST, esses dados são geralmente recebidos no formato JSON. Na primeira linha, essa requisição é convertida em um objeto *JavaScript* e só passa para a fase seguinte tal como evidenciado pela presença do *await* no código. Posteriormente, a ligação à base de dados é estabelecida. Se a ligação for estabelecida, é iniciada a criação de uma reunião no MongoDB e caso seja bem-sucedido, é retornado uma resposta de código 201 ao cliente sobre o sucesso da operação.

A função GET lista todas as reuniões. A função começa por estabelecer a conexão à base de dados. De seguida, ele busca todos as instâncias que correspondem ao modelo Appointment e devolve a lista de reuniões em formato JSON no caminho correspondente às pastas existentes quando a aplicação é executada.

A screenshot of a web browser window displaying a JSON array of appointment data. The browser's address bar shows the URL 'localhost:3000/api/appointments'. The JSON data is displayed in a dark-themed editor with line numbers on the left. The data consists of three appointment objects, each with fields for '_id', 'subject', 'organizer', 'date', 'start', 'end', 'open', and 'room_id'.

```
1  [  
2    {  
3      "_id": "66959f161633aecf06b46a62",  
4      "subject": "Tuesday Appointment",  
5      "organizer": "Diogo Ferreira",  
6      "date": "2024-07-16T00:00:00.000Z",  
7      "start": "2024-07-16T09:30:00.000Z",  
8      "end": "2024-07-16T12:00:00.000Z",  
9      "open": "True",  
10     "room_id": "661fd94fe9876c829a1f2745"  
11   },  
12   {  
13     "_id": "6695a01f1633aecf06b46a94",  
14     "subject": "Wednesday Appointment",  
15     "organizer": "Diogo Ferreira",  
16     "date": "2024-07-17T00:00:00.000Z",  
17     "start": "2024-07-17T15:30:00.000Z",  
18     "end": "2024-07-17T17:00:00.000Z",  
19     "open": "True",  
20     "room_id": "661fd923e9876c829a1f2742"  
21   },  
22   {  
23     "_id": "6695a1fe1633aecf06b46ac0",  
24     "subject": "Thursday Appointment",  
25     "organizer": "Diogo Ferreira",  
26     "date": "2024-07-18T00:00:00.000Z",  
27     "start": "2024-07-18T09:30:00.000Z",  
28     "end": "2024-07-18T12:00:00.000Z",  
29     "open": "True",  
30     "room_id": "661fd923e9876c829a1f2742"  
31   },  
32 ]
```

Figura 32 - Lista parcial de reuniões na API das reuniões

A função DELETE apaga uma determinada reunião da base de dados. Inicialmente, o id da reunião é extraído do URL da requisição. De seguida é estabelecida a ligação à base de dados. Depois de essa ligação estar estabelecida, é efetuada uma filtragem pelo id dentro da coleção das reuniões no MongoDB e quando for encontrado uma reunião com esse id, a mesma é apagada. Caso essa operação seja bem-sucedida, o utilizador recebe uma resposta de código 200.

```

import connectMongoDB from "@app/mongodb/mongodb";
import Appointment from "@app/model/appointment";
import { NextResponse } from "next/server";

export async function PUT(request, { params }) {
  const { id } = params;
  const {
    subject: newSubject,
    organizer: newOrganizer,
    date: newDate,
    start: newStart,
    end: newEnd,
    open: newOpen,
    room_id: newRoomId
  } = await request.json();

  await connectMongoDB();

  const update = {
    subject: newSubject,
    organizer: newOrganizer,
    date: newDate,
    start: newStart,
    end: newEnd,
    open: newOpen,
    room_id: newRoomId
  };

  await Appointment.findByIdAndUpdate(id, update);
  return NextResponse.json({ message: "Appointment updated" }, { status: 200 });
}

export async function GET(request, { params }) {
  const { id } = params;
  await connectMongoDB();
  const appointment = await Appointment.findOne({ _id: id });
  return NextResponse.json({ appointment }, { status: 200 });
}

```

Figura 33 - Ficheiro route.js no caminho `"/api/appointments/[id]"`

O ficheiro representado na figura 33 serve para operações relacionadas com um id específico e contém as funções PUT e GET. Estas funções recebem o id como argumento e por isso foram criadas em um ficheiro separado em uma pasta distinta.

A função PUT serve para editar os campos de uma reunião existente e recebe o id como parâmetro, sendo extraído na primeira linha. A requisição JSON com os novos dados é recebida como argumento da função e é traduzida em *JavaScript* e os seus parâmetros são renomeados com prefixo *'new'* para distinguir do que já existe. Depois disso, é efetuada a conexão à base de dados e criado um objeto *update* onde se encontram os novos valores. A

coleção das reuniões é percorrida para que se possa encontrar pelo id a reunião a ser editada. Caso isso aconteça, a reunião em questão é editada com os dados atualizados e o utilizador é informado sobre o sucesso da operação com um código 200.

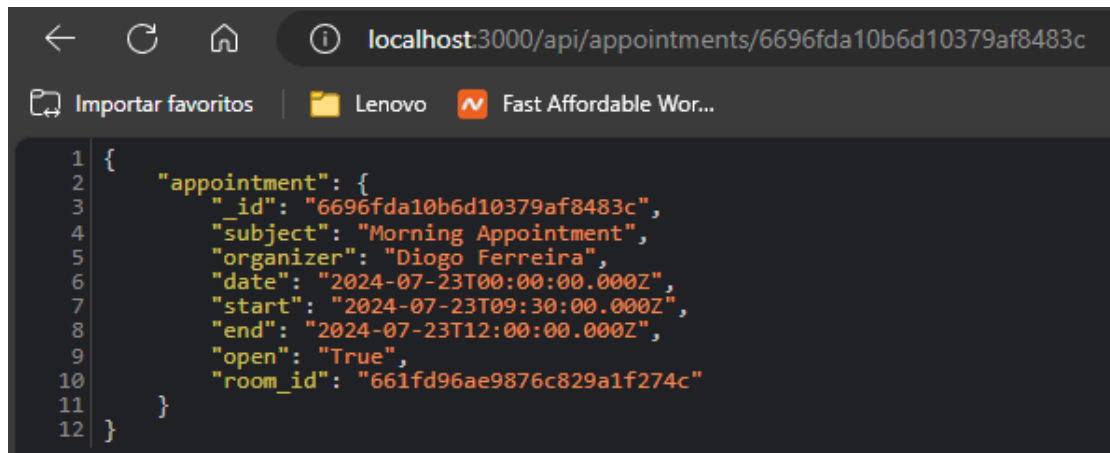


Figura 34 - API de uma reunião filtrada pelo id

A função GET serve para devolver uma reunião que possui um determinado id e recebe esse mesmo id como parâmetro, sendo o mesmo extraído na primeira linha. De seguida efetua a conexão à base de dados e percorre a coleção das reuniões à procura da reunião com aquele id. Se essa reunião for encontrada, a mesma é devolvida em formato JSON, tal como aparece na figura 31.

```

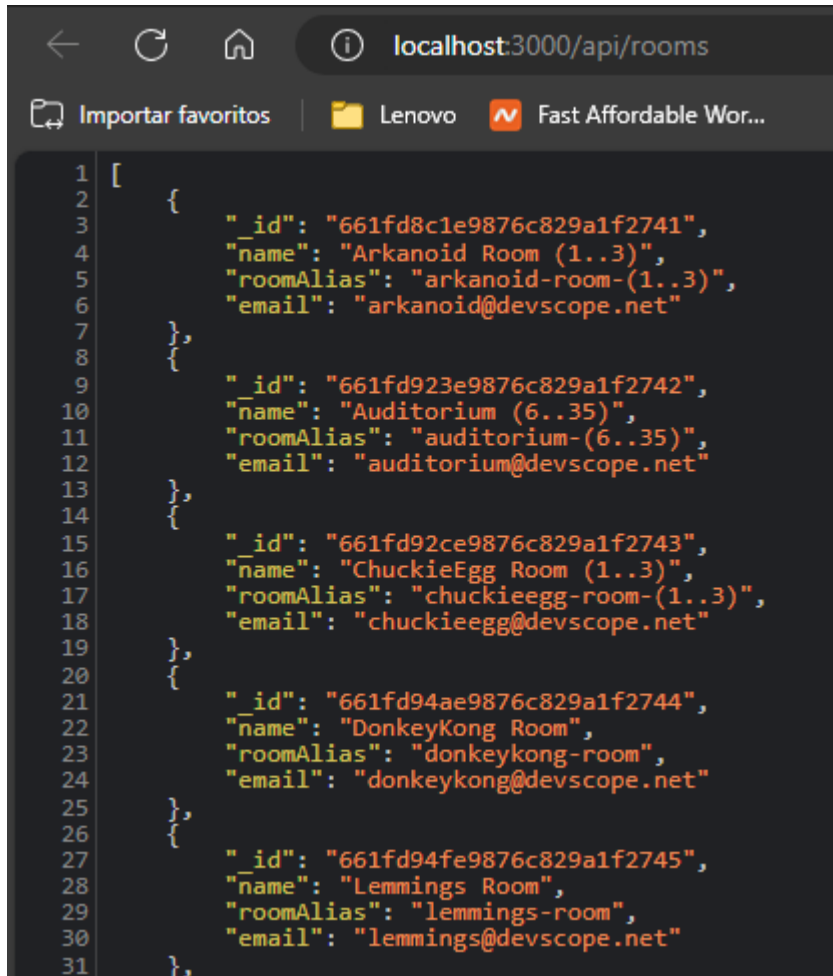
import connectMongoDB from "@app/mongodb/mongodb";
import Room from "@app/model/room";
import { NextResponse } from "next/server";

export async function GET() {
  await connectMongoDB();
  const rooms = await Room.find();
  return NextResponse.json(rooms);
};

```

Figura 35 - Ficheiro route.js no caminho "/api/rooms"

O ficheiro representado na figura 35 demonstra a função GET que serve para devolver a lista de todas as salas. No caso das salas, não termos mais operações de outro tipo além do GET em ambos os ficheiros pois as mesmas não irão ser editadas nem eliminadas, nem serão criadas salas e apenas serão consultadas.



```
1 [
2   {
3     "_id": "661fd8c1e9876c829a1f2741",
4     "name": "Arkanoid Room (1..3)",
5     "roomAlias": "arkanoid-room-(1..3)",
6     "email": "arkanoid@devscope.net"
7   },
8   {
9     "_id": "661fd923e9876c829a1f2742",
10    "name": "Auditorium (6..35)",
11    "roomAlias": "auditorium-(6..35)",
12    "email": "auditorium@devscope.net"
13  },
14  {
15    "_id": "661fd92ce9876c829a1f2743",
16    "name": "ChuckieEgg Room (1..3)",
17    "roomAlias": "chuckieegg-room-(1..3)",
18    "email": "chuckieegg@devscope.net"
19  },
20  {
21    "_id": "661fd94ae9876c829a1f2744",
22    "name": "DonkeyKong Room",
23    "roomAlias": "donkeykong-room",
24    "email": "donkeykong@devscope.net"
25  },
26  {
27    "_id": "661fd94fe9876c829a1f2745",
28    "name": "Lemmings Room",
29    "roomAlias": "lemmings-room",
30    "email": "lemmings@devscope.net"
31  },
32 ]
```

Figura 36 - Lista parcial de salas na API das salas

A função GET lista todas as salas. A função começa por estabelecer a conexão à base de dados. De seguida, ele busca todas as instâncias que correspondem ao modelo *Room* e devolve a lista de salas em formato JSON no caminho correspondente às pastas existentes quando a aplicação é executada, tal como aparece na figura 36.

```
import connectMongoDB from "@app/mongodb/mongodb";
import Room from "@app/model/room";
import { NextResponse } from "next/server";

export async function GET(request, { params }) {
  const { id } = params;
  await connectMongoDB();
  const appointment = await Room.findOne({ _id: id });
  return NextResponse.json({ appointment }, { status: 200 });
}
```

Figura 37 – Ficheiro route.js no caminho `"/api/rooms/[id]"`

O ficheiro representado na figura 37 demonstra a função GET que neste caso serve para devolver uma sala que possui um id específico.

A função GET serve para devolver uma reunião que possui um determinado id e recebe esse mesmo id como parâmetro, sendo o mesmo extraído na primeira linha. De seguida efetua a conexão à base de dados e percorre a coleção das salas à procura da sala com aquele id. Se essa sala for encontrada, a mesma é devolvida em formato JSON no *endpoint* da API correspondente, tal como aparece na figura 38.

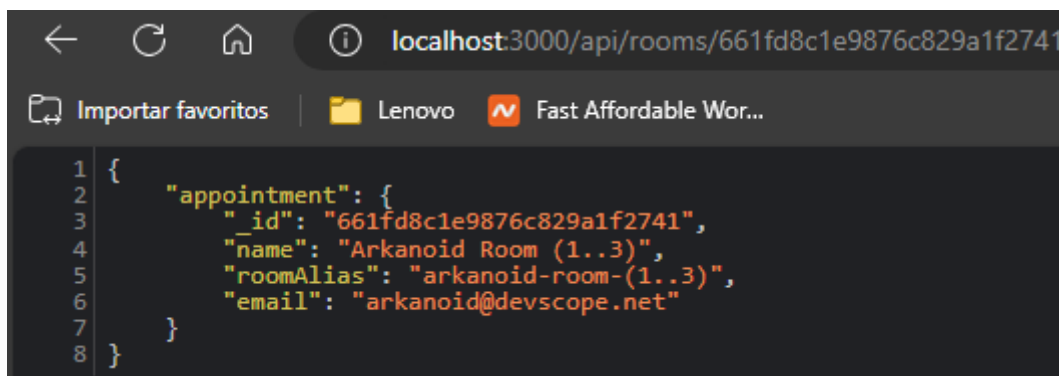


Figura 38 - API de uma sala filtrada pelo id

4.1.3 Login

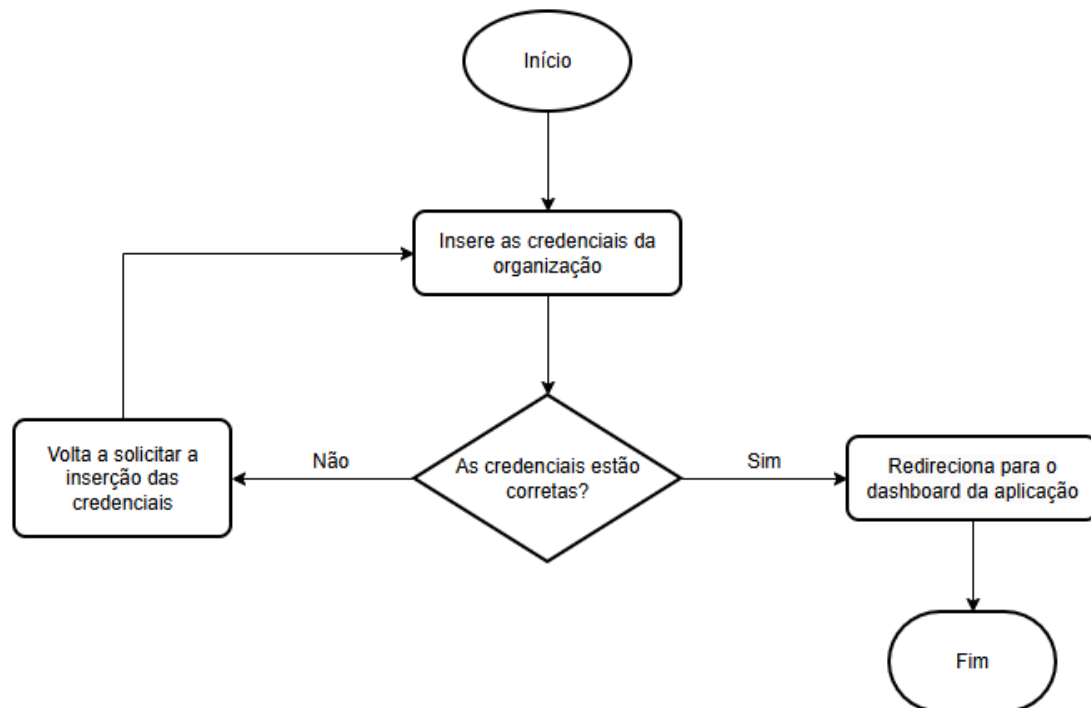


Figura 39 - Fluxograma do login

Na figura 39, temos o fluxograma que descreve a interação do utilizador na aplicação relativamente ao caso de uso do *login*. O utilizador registado na organização efetua o login com as credenciais da organização recorrendo ao *Microsoft 365*. De seguida, o sistema verifica se as credenciais estão corretas. Se estiverem, o utilizador passa a ter acesso à aplicação, sendo redirecionado para página inicial que é o *dashboard*. Caso contrário, é pedido ao utilizador que volte a inserir as suas credenciais de acesso.

Foram criadas variáveis de ambiente no Azure que permitissem efetuar o *login* e inseridas no ficheiro `.env`, tal como aparece na figura 40.

```

AZURE_AD_CLIENT_ID="2a5b2938-74cb-48e8-be82-fccddb51998"
AZURE_AD_CLIENT_SECRET="POp8Q~sKXWi0.vEQgnPejUyiWIy2YymT.CuNVc5S"
AZURE_AD_TENANT_ID="09e251dc-5e87-48bf-b4d2-71b01adb984a"
NEXTAUTH_SECRET="xNqmQlWquu5saM3/jfdw5BaHGt/0vKAB9nqyoCrXifA="
NEXTAUTH_URL="http://localhost:3000"
  
```

Figura 40 - Variáveis no ficheiro `.env` para o login

Foram criadas várias classes para se tratar do *login*.

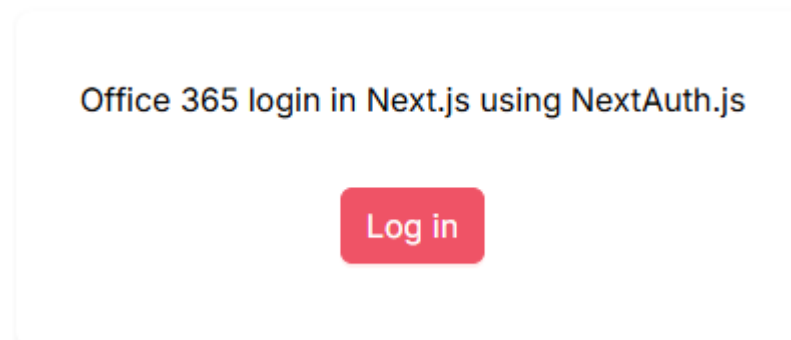


Figura 41 – Botão de login

A classe *page.tsx* encontra-se no caminho “/app” e possui um botão de login com uma pequena descrição por cima, tal como se encontra na figura 41, e o *login* será feito recorrendo ao azure-ad e após o *login* seremos redirecionados para o *dashboard*. O ‘*use client*’ significa que esta classe possui interação com o utilizador através de botões neste caso.

```
import '@/app/ui/global.css';
import { SessionProvider } from 'next-auth/react';

export default function App({
  Component,
  pageProps: { session, ...pageProps },
}) {
  return (
    <SessionProvider session={session}>
      <Component {...pageProps} />
    </SessionProvider>
  );
}
```

Figura 42 - Ficheiro *_app.js*

O ficheiro *_app.js* encontra-se na figura 42 e o *SessionProvider* serve para que qualquer componente dentro da aplicação utilize os dados do utilizador.

```
import NextAuth from 'next-auth'
import AzureADProvider from 'next-auth/providers/azure-ad'

const authOptions = {
  providers: [
    AzureADProvider({
      clientId: process.env.AZURE_AD_CLIENT_ID ?? '',
      clientSecret: process.env.AZURE_AD_CLIENT_SECRET ?? '',
      tenantId: process.env.AZURE_AD_TENANT_ID,
    }),
  ],
  pages: {
    signIn: '/auth/signin',
  },
}

const handler = NextAuth(authOptions)

export { handler as GET, handler as POST }
```

Figura 43 - Classe route.js

No route.ts, que esta na figura 43 e que se encontra no caminho `/api/auth/[...nextauth]`, são definidos os provedores de autenticação, neste caso o Azure, e os outros obtidos a partir das variáveis de ambiente. O caminho onde o *login* será efetuado juntamente com *handler* para as operações de *login* e *logout* também são definidos.



Iniciar sessão

E-mail, telefone ou Skype

[Não consegue aceder à sua conta?](#)

Seguinte

Figura 44 - Login pela Microsoft

Quando se carrega no botão de *login*, somos redirecionados para uma página de *login* da *Microsoft*, tal como aparece na figura 44.

```
<button onClick={() => signOut({ callbackUrl: '/' })}  
  className="flex h-[48px] grow items-center justify-c  
  <PowerIcon className="w-6" />  
  <div className="hidden md:block">Sign Out</div>  
</button>
```

Figura 45 - Criação do botão de *logout*

Na classe *sidenav.tsx* que se encontra no caminho *"/ui/dashboard"* no projeto, é criado o botão de *logout*, que quando é premido, envia-nos de volta para a página inicial onde se encontra o botão de *login*. O botão de *logout* aparece no canto inferior da barra lateral de navegação.

4.1.4 Listar informação sobre salas

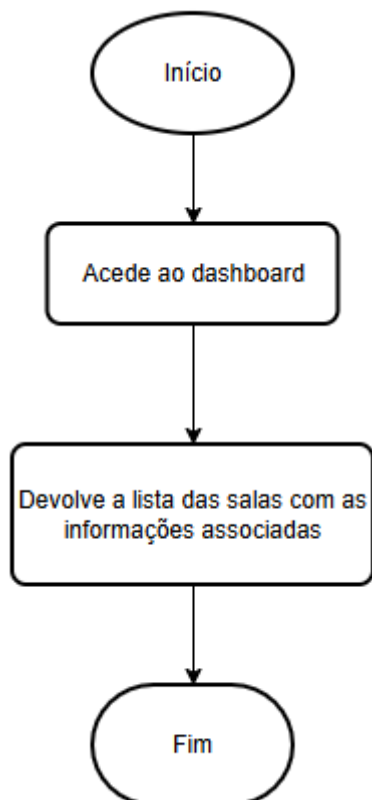


Figura 46 - Fluxograma da listagem de informação sobre as salas

Na figura 46, temos o fluxograma que descreve a interação do utilizador na aplicação relativamente ao caso de uso da listagem de informação sobre as salas. O utilizador já logado na aplicação acede à secção do *dashboard* onde se encontram as informações sobre as salas. Esta ação devolve uma tabela com as informações sobre as salas.

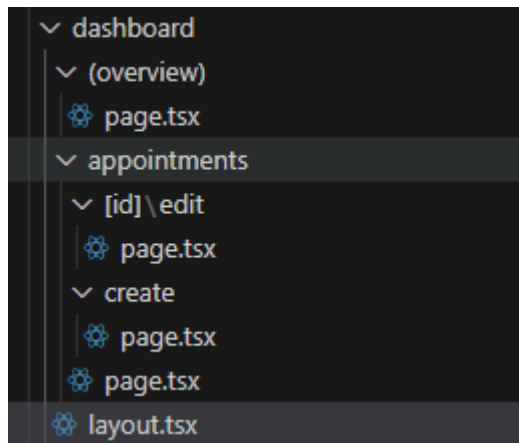


Figura 47 - Pasta do dashboard

Na figura 47, temos várias pastas dentro da pasta dashboard. Na pasta *'(overview)'* temos a página onde irá aparecer a tabela com informações sobre as salas. Esta pasta e outros ficheiros importados na classe *page.tsx* dentro da mesma serão analisados neste subcapítulo. A classe *layout.tsx* está apenas dentro da pasta *'dashboard'* e será reutilizada em todas as páginas pois contém a barra lateral de navegação para se poder aceder às tabelas das salas e reuniões. O ficheiro diretamente dentro da pasta *"/appointments"* representam as páginas onde será listada a lista de reuniões, e no caso dos ficheiros que estão dentro das pastas *"/create"* e *"/[id]/edit"*, representam os formulários de criação e edição de reuniões respetivamente.

```
import SideNav from '@app/ui/dashboard/sidenav';

export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <div className="flex h-screen flex-col md:flex-row md:overflow-hidden">
      <div className="w-full flex-none md:w-64">
        <SideNav />
      </div>
      <div className="flex-grow p-6 md:overflow-y-auto md:p-12">{children}</div>
    </div>
  );
}
```

Figura 48 - Conteúdo da classe layout.tsx

Na figura 48, é importada a barra lateral de navegação a partir de um caminho específico que será utilizada em todas as páginas que estejam dentro da pasta *'dashboard'*.

A função define um componente do *React* denominado de *Layout* que pode ser utilizado em várias páginas e recebe um componente *'children'* que representa o conteúdo dinâmico dentro desse layout e que pode ser qualquer elemento válido do *React*.

Na estrutura JSX do *layout*, a div principal trata da organização e disposição dos elementos. O segundo div trata da barra lateral de navegação, em que em ecrãs menores ocupará todo espaço e em ecrãs maiores terá um tamanho definido e evitará alterações no tamanho através do item *'flex-none'*. No último div, é tratado o conteúdo das tabelas que será exibido ao lado da barra lateral.

```
'use client';

import Link from 'next/link';
import NavLinks from '@app/ui/dashboard/nav-links';
import { PowerIcon } from '@heroicons/react/24/outline';
import { signOut } from 'next-auth/react';

export default function SideNav() {
  return (
    <div className="flex h-full flex-col px-3 py-4 md:px-2">
      <Link
        className="mb-2 flex h-20 items-end justify-start rounded-md bg-gray-50 p-4 md:h-40"
        href="/dashboard"
      >
      </Link>
      <div className="flex grow flex-row justify-between space-x-2 md:flex-col md:space-x-0 md:space-y-2">
        <NavLinks />
        <div className="hidden h-auto w-full grow rounded-md bg-gray-50 md:block"></div>
        <form>
          <button onClick={() => signOut({ callbackUrl: '/' })}
            className="flex h-[48px] grow items-center justify-center gap-2 rounded-md bg-gray-50 p-3 text"
          >
            <PowerIcon className="w-6" />
            <div className="hidden md:block">Sign Out</div>
          </button>
        </form>
      </div>
    </div>
  );
}
```

Figura 49 - Classe *sidenav.tsx*

A classe *sidenav.tsx*, representada na figura 49, que é importada na figura 45 na classe *layout.tsx*, encontra-se na pasta *'/ui/dashboard'*. A pasta *'/ui'* serve para definir a disposição dos componentes na página.

Neste caso, um bloco cinzento, que aparece na figura 50 acima dos botões que permitem aceder às 2 páginas principais, permite ao utilizador voltar à página inicial, neste caso, o *dashboard*.

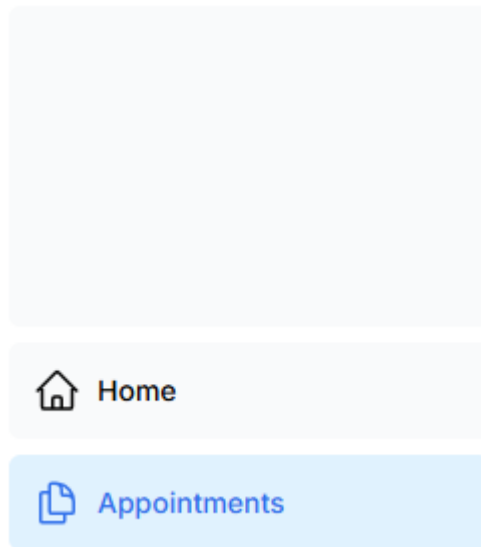


Figura 50 - Parte superior da barra lateral de navegação

Na classe *nav-links.tsx* que é importada na classe *sidenav.tsx* são definidos os *links* de acesso quando se clica nos botões que aparecem acima. O home corresponde ao dashboard que se encontra no caminho *'/dashboard'* e os appointments à lista de reuniões que se encontra no caminho *'/dashboard/appointments'*.

```

import { fetchFilteredRooms } from '@app/lib/data';
import RoomsTable from '@app/ui/dashboard/table';
import { Metadata } from 'next';

export const metadata: Metadata = {
  title: 'Rooms',
};

export default async function Page({
  searchParams,
}: {
  searchParams?: {
    query?: string;
  };
}) {
  const query = searchParams?.query || '';

  const rooms = await fetchFilteredRooms(query);

  return (
    <main>
      <RoomsTable rooms={rooms} />
    </main>
  );
}

```

Figura 51 - Conteúdo da classe *page.tsx*

Na *page.tsx* trata-se da página que exibe informação sobre as salas. No primeiro import vai-se buscar a função responsável por tratar sobre os dados sobre as salas. No segundo define-se a tabela sobre as salas e no último define-se um título através dos metadados.

Na primeira constante define-se o título da página para se informar o utilizador sobre o facto de estar em uma página que contém informação sobre as salas.

A função é assíncrona porque necessita de aguardar pela informação proveniente da função *fetchFilteredRooms* que trata de devolver informação sobre as salas e filtragem das mesmas segundo alguns critérios. O *searchParams* é passado por parâmetro e contém parâmetros opcionais de consulta passados na URL. A *query* é a string que será inserida na barra de pesquisa para filtrar as salas com base em determinados critérios. Se a mesma não existir ele atribui uma string vazia.

O retorno da função é um JSX que contém a *RoomsTable* que recebe *'rooms'* que são as salas como propriedade e passa os dados para uma tabela.

Rooms

Name	Alias	Email	Number of Appointments	Busy
Arkanoid Room (1.3)	arkanoid-room-(1.3)	arkanoid@devscope.net	1	Yes
Auditorium (6.35)	auditorium-(6.35)	auditorium@devscope.net	1	No
ChuckieEgg Room (1.3)	chuckieegg-room-(1.3)	chuckieegg@devscope.net	1	No
DonkeyKong Room	donkeykong-room	donkeykong@devscope.net	1	No
Lemmings Room	lemmings-room	lemmings@devscope.net	0	No
MetalGear Room	metalgear-room	metalgear@devscope.net	0	No
Minesweeper Room	minesweeper-room	minesweeper@devscope.net	1	No
Outrun Room	outrun-room	outrun@devscope.net	0	No
PacMan Room (6.12)	pacman-room-(6.12)	pacman@devscope.net	1	No

Figura 52 - Tabela de salas

Tuesday Appointment	Diogo Ferreira	Arkanoid Room (1.3)	2024-08-13	10:30	12:30	No	Update	Delete
---------------------	----------------	---------------------	------------	-------	-------	----	--------	--------

Figura 53 - Reunião da sala que se encontra ocupada

A tabela de salas devolve informações em várias colunas tais como o nome, alias, email, o número de reuniões agendadas na sala e o estado atual da sala. Neste caso a primeira sala encontra-se ocupada e as outras estão livres.

localhost:3000/dashboard?page=1&query=Arkanoid

Rooms

Name	Alias	Email	Number of Appointments	Busy
Arkanoid Room (1.3)	arkanoid-room-(1.3)	arkanoid@devscope.net	1	Yes

Figura 54 - Filtragem de salas na barra de pesquisa

A sala pode ser filtrada de acordo o seu nome, alias ou email. Para isso, temos de inserir uma *string* na barra de pesquisa que neste caso devolveu a sala ocupada. A *query* inserida na barra de pesquisa aparece no URL da página. Normalmente a barra de pesquisa aparece vazia por causa da *query* ser opcional.

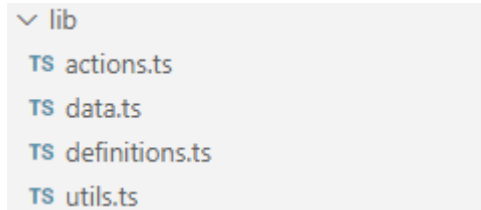


Figura 55 - Pasta 'lib'

A função *fetchFilteredRooms* que se encontra na classe *data.ts* na pasta 'lib' serve para buscar as informações sobre as reuniões e as salas às respectivas APIs locais e efetuar o tratamento desses dados para se preencher a tabela com as informações sobre as salas. Nessa classe também existem outras funções semelhantes que efetuam o tratamento de dados para a tabela das reuniões e outras funcionalidades relacionadas com as reuniões.

```
export async function fetchFilteredRooms(query: string) {
  noStore();

  try {
    const room_res = await fetch('http://localhost:3000/api/rooms');
    if (!room_res.ok) {
      throw new Error(`HTTP error! Status: ${room_res.status}`);
    }
    const rooms = await room_res.json();

    const appointment_res = await fetch('http://localhost:3000/api/appointments', { next: { revalidate: revalidationTime } });
    if (!appointment_res.ok) {
      throw new Error(`HTTP error! Status: ${appointment_res.status}`);
    }
    const appointments = await appointment_res.json();
  }
}
```

Figura 56 - Parte inicial do bloco try da função *fetchFilteredRooms*

Na figura 56, o método *noStore()* serve para impedir que as respostas HTTP não devem ser armazenadas em cache. A função possui um bloco *try-catch* para tratamento de erros.

A função faz uma requisição HTTP às APIs locais que possuem as listas de salas e reuniões utilizando o *fetch*. No caso das reuniões, possui um parâmetro *revalidate* para que as informações sobre as reuniões sejam atualizadas em um determinado período definido pela constante *revalidationTime* que irá aparecer em outras funções *fetch* desta classe que tratem das reuniões. Isto consiste na aplicação do ISR no código em que a informação sobre as reuniões são armazenadas em cache e apenas depois do *revalidationTime*, que está definido como uma constante no início da classe tal como na figura 57, ter acabado é que a página será atualizada com a informação nova e a informação que não foi alterada continuará na mesma.

```
const revalidationTime = 5;
```

Figura 57 - Constante *revalidationTime*

O `await` serve para apenas avançar quando os resultados do `fetch` estiverem prontos e se existirem erros nesse processo, um erro é enviado e entra no bloco `catch` que se encontra na figura abaixo.

```
} catch (error) {  
  console.error('Database Error:', error);  
  throw new Error('Failed to fetch the rooms table.');
```

Figura 58 - Bloco catch da função `fetchFilteredRooms`

O método `.json()` serve para extrair o corpo da resposta do `fetch` e convertê-lo de uma `string` JSON tal como se encontra nas APIs locais para um objeto `TypeScript` para que possa ser manipulado no código e só avançará quando tiver o resultado transformado.

```
const appointmentRoomCounts: Record<string, number> = {};  
appointments.forEach((appointment: Appointment) => {  
  const room_id = appointment.room_id;  
  appointmentRoomCounts[room_id] = (appointmentRoomCounts[room_id] || 0) + 1;  
});  
  
const roomAvailability: Record<string, boolean> = {};  
rooms.forEach((room: Room) => {  
  roomAvailability[room._id] = true;  
});
```

Figura 59 - Criação de objetos para serem tratados

Na figura 59, são criados dois objetos.

O primeiro objeto serve para calcular o número de reuniões marcadas em cada sala. Nessa situação, começa por percorrer a lista de reuniões. O `room_id`, que é o id da sala onde vai decorrer a reunião é passado para uma variável `room_id`. Posteriormente verifica se a sala já possui um `array` onde serão armazenados o número de reuniões. Se o contador não foi definido para uma determinada sala, começa do 0. Caso contrário, adiciona 1 ao valor atual.

O segundo objeto serve para determinar a disponibilidade da sala no momento atual. A lista de salas é percorrida para inicializar um array com todas as salas, que são as chaves, com o valor de true para que depois possa ser editado.

```
appointments.forEach((appointment: Appointment) => {
  const room_id = appointment.room_id;

  const appointmentDate = new Date(appointment.date);
  const appointmentStart = new Date(appointment.start);
  const appointmentEnd = new Date(appointment.end);

  const startTime = appointmentStart.toISOString().split('T')[1].substring(0, 5);
  const endTime = appointmentEnd.toISOString().split('T')[1].substring(0, 5);

  const [startHours, startMinutes] = startTime.split(':').map(Number);
  const [endHours, endMinutes] = endTime.split(':').map(Number);

  const start = new Date(appointment.date).setHours(startHours, startMinutes);
  const end = new Date(appointment.date).setHours(endHours, endMinutes);

  const currentDate = new Date();

  if(appointmentDate.toDateString() === currentDate.toDateString()) {
    if(start <= currentDate.getTime() && end > currentDate.getTime()) {
      roomAvailability[room_id] = false;
    }
  }
});

const filteredRooms = rooms
  .filter((room: Room) =>
    room.name.includes(query) || room.roomAlias.includes(query) || room.email.includes(query)
  )
  .map((room: Room): RoomsTableType => {
    return {
      name: room.name,
      roomAlias: room.roomAlias,
      email: room.email,
      total_appointments: appointmentRoomCounts[room._id] || 0,
      busy: roomAvailability[room._id],
    };
  });

return filteredRooms;
```

Figura 60 – Definição da disponibilidade das salas e tratamento final dos dados

A lista de reuniões é iterada e o id da sala onde vai decorrer a reunião é armazenado em um variável e as datas e horários são transformadas em variáveis do tipo *Date* pois na API local, onde as reuniões estão armazenadas em formato JSON, essas variáveis são do tipo *string*. Os horários são convertidos no formato de data ISO e são filtrados e divididos para seja apenas mostrado as horas e minutos e se possa obter as horas e minutos separadamente. Posteriormente, são criadas instâncias de *start* e *end*, onde se encontra a data da reunião e são definidos as horas e minutos de começo e de fim da reunião. Este processo foi realizado

por causa dos horários de Verão, em que sem estas instruções iriam ser realizados cálculos com base no horário de Verão e uma determinada sala iria aparecer como livre quando na realidade deveria estar ocupada pois os horários das reuniões estavam adiantados 1 hora por causa do horário de Verão.

Seria efetuada uma comparação entre a data e horários atuais e a data e os horários de começo e fim da reunião para se verificar se a sala está ocupada. Caso a data da reunião correspondesse à data atual e o horário atual estivesse entre o horário de começo e o horário de fim, a disponibilidade da sala associada a essa reunião passa a ser false.

De seguida é criada uma instância de *filteredRooms*, onde a lista de salas será filtrada através de uma *query* que seria inserida na barra de pesquisa acima da tabela com as informações sobre as salas. Os parâmetros que seriam utilizados na filtragem da *query* inserida seriam o nome, o alias e o email da sala. A tabela das salas seria criada com base no tipo *RoomsTableType*, que consiste das colunas que aparecem na tabela que são o nome, alias e email da sala, o número de reuniões marcadas na sala e a disponibilidade da sala a ser iterada naquele momento no map. Finalmente seriam retornadas as *filteredRooms* com base na *query* inserida na barra de pesquisa.

```
export type RoomsTableType = {  
  name: string;  
  roomAlias: string;  
  email: string;  
  total_appointments: number;  
  busy: boolean;  
};
```

Figura 61 - Tipo *RoomsTableType*

Os tipos que foram utilizados nesta função e em outras da classe *data.ts* encontram-se na classe *definitions.ts* e servem para a criação de tabelas de salas e reuniões e formulário de criação de reuniões. Também se encontram lá os tipos da sala e da reunião que contém todas as variáveis associadas aos mesmos, tal como se encontra na base de dados MongoDB, para se poder iterar as instâncias de salas e reuniões para o tratamento de dados.

4.1.5 Listar reuniões

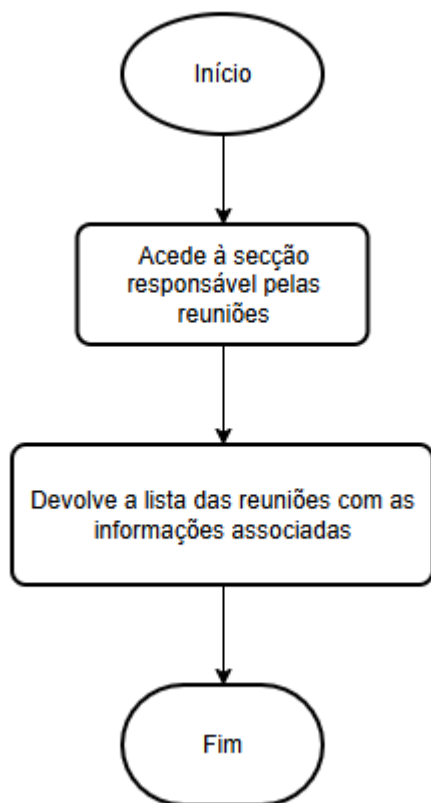


Figura 62 - Fluxograma da listagem de reuniões

Na figura 62, temos o fluxograma que descreve a interação do utilizador na aplicação relativamente ao caso de uso da listagem das reuniões. O utilizador já logado na aplicação acede à secção das reuniões onde se encontram as informações sobre as reuniões. Esta ação devolve uma tabela com as informações sobre as reuniões e botões responsáveis pela criação, edição e eliminação de reuniões.

Na figura 63, encontra-se a tabela com a lista de reuniões. Cada reunião contém as seguintes variáveis: assunto, organizador, sala, data, hora de começo e de fim e o estado da reunião que pode ser público ou privado. Cada reunião pode ser editada ou eliminada tal como se pode verificar nos botões no lado direito de cada linha, em que os botões *update* e *delete*, que servem respetivamente para editar e eliminar a reunião em questão, possuem cores distintas para se poder distinguir melhor as suas funções. A barra de pesquisa tal como no caso da lista de salas, serve para procurar reuniões com base em certos critérios através da inserção de uma *query*. Caso não seja inserida qualquer *query*, serão devolvidas todas as reuniões existentes organizadas em várias páginas.

Subject	Organizer	Room	Date	Start	End	Private	
Monday Appointment	Diogo Ferreira	Minesweeper Room	2024-08-19	15:00	17:30	No	Update Delete
			2024-08-20	10:30	12:30	No	Update Delete
Wednesday Appointment	Diogo Ferreira	ChuckieEgg Room (1..3)	2024-08-21	10:00	12:00	No	Update Delete
Morning Appointment	Diogo Ferreira	PacMan Room (6..12)	2024-08-22	09:30	11:30	No	Update Delete
Afternoon Appointment	Diogo Ferreira	Pang Room (2..4)	2024-08-22	14:30	16:30	No	Update Delete
Friday Appointment	Diogo Ferreira	DonkeyKong Room	2024-08-23	10:00	12:00	No	Update Delete

Figura 63 - Lista de reuniões

O caminho da página é *"/dashboard/appointments"* pois a classe *page.tsx* responsável por esta página encontra-se dentro dessa pasta no projeto. Os formulários de criação e edição de reuniões também possuem os seus próprios caminhos e ficheiros correspondentes.

```

import AppointmentsTable from '@app/ui/appointments/table';
import Search from '@app/ui/search';
import Pagination from '@app/ui/appointments/pagination';
import { CreateAppointment } from '@app/ui/appointments/buttons';
import { lusitana } from '@app/ui/fonts';
import { Suspense } from 'react';
import { fetchAppointmentsPages } from '@app/lib/data';
import { AppointmentsTableSkeleton } from '@app/ui/skeletons';
import { Metadata } from 'next';

export const metadata: Metadata = {
  title: 'Appointments',
};

export default async function Page({
  searchParams,
}): {
  searchParams?: {
    query?: string;
    page?: string;
  };
} {
  const query = searchParams?.query || '';
  const currentPage = Number(searchParams?.page) || 1;
  const totalPages = await fetchAppointmentsPages(query);

  return (
    <div className="w-full">
      <div className="flex w-full items-center justify-between">
        <h1 className={` ${lusitana.className} text-2xl`}>Appointments</h1>
      </div>
      <div className="mt-4 flex items-center justify-between gap-2 md:mt-8">
        <Search placeholder="Search appointments..." />
        <CreateAppointment />
      </div>
      <Suspense key={query + currentPage} fallback=<AppointmentsTableSkeleton />>
        <AppointmentsTable query={query} currentPage={currentPage} />
      </Suspense>
      <div className="mt-5 flex w-full justify-center">
        <Pagination totalPages={totalPages} />
      </div>
    </div>
  );
}

```

Figura 64 - Conteúdo da classe `page.tsx` responsável pela listagem de reuniões

Na figura 64, encontra-se a classe necessária para a criação da página onde se encontra a tabela das reuniões. Esta classe irá usar métodos provenientes de outras classes necessários para a obtenção das instâncias de reuniões que existem, botões para a criação, edição e eliminação de reuniões e para a paginação.

A classe possui um título chamado *'Appointments'* que aparece no topo da página para a identificar e para isso é importada a interface `Metadata`.

A função é assíncrona e passa por parâmetro a variável `searchParams` que são a query que será inserida na barra de pesquisa e o número da página que está a ser mostrada ao utilizador, sendo ambas as variáveis do tipo `string` e opcionais. Caso a query seja vazia, a barra de pesquisa aparecerá vazia e todas as reuniões serão listadas. De seguida será importada da classe `data.tsx` uma função chamada `fetchAppointmentsPages` que irá retornar o número total de páginas.

O JSX, que corresponde a parte do `return` e o que se encontra dentro de parenteses, corresponde aquilo que irá ser mostrado na página da lista de reuniões. No primeiro `div` dentro do `div` principal será estilizado o título que foi definido anteriormente como `Metadata`. No segundo `div`, são definidos a barra de pesquisa que terá o texto “Search appointments...” como padrão caso a barra esteja vazia e ao lado o botão que irá direcionar o utilizador para o formulário de criação, que é criado na classe `buttons.tsx` de onde será importado. O componente `Suspense` será utilizado na parte da tabela para se lidar com carregamento assíncrono de uma forma mais limpa, mostrando um esqueleto de tabela, neste caso o `AppointmentsTableSkeleton` que é definido no `fallback`. O `key` é útil para que quando a query ou a `currentPage` forem alterados, os dados a serem mostrados na tabela, que é importada da classe `table.tsx` onde é tratada a tabela de reuniões a ser mostrada, sejam corretamente alterados. No último `div`, será mostrada a configuração da paginação que será importada da classe `pagination.tsx` na pasta “/ui/appointments” e que incluirá o número total de páginas.

```
export async function fetchAppointmentsPages(query: string) {
  noStore();

  try {
    const appointment_res = await fetch('http://localhost:3000/api/appointments', { next: { revalidate: revalidationTime } });
    if (!appointment_res.ok) {
      throw new Error(`HTTP error! Status: ${appointment_res.status}`);
    }
    const appointments = await appointment_res.json();

    const room_res = await fetch('http://localhost:3000/api/rooms');
    if (!room_res.ok) {
      throw new Error(`HTTP error! Status: ${room_res.status}`);
    }
    const rooms = await room_res.json();

    const roomNameMap: Record<string, string> = {};
    rooms.forEach((room: Room) => {
      roomNameMap[room._id] = room.name;
    });
  }
}
```

Figura 65 - Parte inicial da função `fetchAppointmentsPages`

Na figura 65, é apresentada a parte inicial da função que se encontra classe `data.tsx` tal como outras funções `fetch` e que é utilizada para calcular o número de páginas que vão ser utilizadas caso todas as reuniões sejam listadas ou caso seja inserida uma query, que é passada como parâmetro do tipo `string` nesta função, na barra de pesquisa. O número de cada página

vai aparecer na paginação abaixo da tabela de reuniões para que o utilizador possa consultar cada página.

No início do bloco *try*, começa-se por obter os dados sobre as salas e reuniões que se encontram armazenados nas respetivas APIs. Caso exista algum erro nesse processo, é enviado um erro e passa-se para o bloco *catch* que se encontra na figura 63.

```

} catch (error) {
  console.error('Database Error: ', error);
  throw new Error('Failed to fetch the total number of appointments.');
```

Figura 66 – Bloco *catch*

Depois da obtenção dos dados, é declarada uma constante chamada *roomNameMap* como um objeto vazio, cujas chaves e valores associados serão do tipo *string*. Posteriormente a lista de salas é iterada e para cada sala, o id da sala passa ser a chave e o nome da sala passa a ser o valor associado à chave. Isto serve para que posteriormente seja listada na tabela de reuniões o nome da sala onde está marcada a reunião para que as reuniões sejam filtradas com base no nome da sala.

```

const filteredAppointments = appointments
  .filter((appointment: Appointment) =>
    | appointment.subject.includes(query) || roomNameMap[appointment.room_id].includes(query)
  )
  .map((appointment: Appointment): AppointmentsTableType => {
    const roomName = roomNameMap[appointment.room_id];

    const appointmentDate = new Date(appointment.date);
    const appointmentStart = new Date(appointment.start);
    const appointmentEnd = new Date(appointment.end);

    const date = appointmentDate.toISOString().split('T')[0];
    const startTime = appointmentStart.toISOString().split('T')[1].substring(0, 5);
    const endTime = appointmentEnd.toISOString().split('T')[1].substring(0, 5);

    return {
      _id: appointment._id,
      subject: appointment.subject,
      organizer: appointment.organizer,
      roomName: roomName,
      date: date,
      start: startTime,
      end: endTime,
      open: appointment.open
    }
  });

const totalPages = Math.ceil(filteredAppointments.length / ITEMS_PER_PAGE);
return totalPages;
```

Figura 67 - Iteração da lista de reuniões na função *fetchAppointmentPages*

Na figura 67, começa por ser aplicado um filtro à lista de reuniões. Esse filtro verifica se o assunto ou o nome de cada sala, que é obtido a partir do objeto `roomNameMap`, contém a query em questão e apenas as reuniões que contiverem essa query serão mostradas.

De seguida, as reuniões que foram filtradas são transformadas em objetos do tipo `AppointmentsTableType` para sejam listadas na tabela com as variáveis necessárias.

As variáveis necessárias são tratadas para que sejam posteriormente listadas. O valor associado à chave do id da sala no objeto `roomNameMap` é passado para uma constante chamada `roomName`, a data e os horários de começo e fim da reunião são transformados em variáveis do tipo `Date` e de seguida são listados em formato ISO do tipo `string`. Isto serve para que o que está antes do 'T' seja associado à data de reunião no formato 'YYYY-MM-DD' e que sejam selecionados os 5 primeiros valores depois do 'T' no caso dos horários de começo e de fim, sendo formatados para o formato 'HH:MM'.

```
export type AppointmentsTableType = {
  _id: string;
  subject: string;
  organizer: string;
  roomName: string;
  date: string;
  start: string;
  end: string;
  open: 'True' | 'False';
};
```

Figura 68 - Tipo AppointmentsTableType

O método `map` retorna um objeto com as novas variáveis que vão aparecer na tabela de reuniões, no formato do tipo `AppointmentsTableType` que se encontra na classe `definitions.ts` representada na figura 68.

```
const ITEMS_PER_PAGE = 6;
```

Figura 69 - Número de reuniões máximo por cada página

O número total de páginas a serem listadas é calculado com base no resultado da divisão do número de reuniões filtradas dividido pelo número de reuniões que irão aparecer em uma única página, que aparece como uma constante na figura 69. O método `Math.ceil` arredonda por excesso o resultado dessa divisão para garantir que todas as reuniões aparecem na tabela mesmo que a última página possua menos objetos que as outras.

```
import AppointmentStatus from '@app/ui/appointments/status';
import { UpdateAppointment, DeleteAppointment } from '@app/ui/appointments/buttons';
import { fetchFilteredAppointments } from '@app/lib/data';

export default async function AppointmentsTable({
  query,
  currentPage,
}): {
  query: string;
  currentPage: number;
}) {
  const appointments = await fetchFilteredAppointments(query, currentPage);
```

Figura 70 – Aquisição das reuniões a serem listadas na tabela

Na figura 70 encontra-se representada a parte inicial da classe table.tsx onde vai ser representada a tabela de reuniões com base em uma query e na página atual. Para esse efeito foi importada a função `fetchFilteredAppointments` que recebe como parâmetros a query, do tipo string e a `currentPage`, do tipo *number*. Esta função irá retornar as reuniões a serem mostradas na página específica da tabela.

```
export async function fetchFilteredAppointments(query: string, currentPage: number) {
  const offset = (currentPage - 1) * ITEMS_PER_PAGE;
  noStore();

  try {
    const room_res = await fetch('http://localhost:3000/api/rooms');
    if (!room_res.ok) {
      throw new Error(`HTTP error! Status: ${room_res.status}`);
    }
    const rooms = await room_res.json();

    const appointment_res = await fetch('http://localhost:3000/api/appointments', { next: { revalidate: revalidationTime } });
    if (!appointment_res.ok) {
      throw new Error(`HTTP error! Status: ${appointment_res.status}`);
    }
    const appointments = await appointment_res.json();

    const roomNameMap: Record<string, string> = {};
    rooms.forEach((room: Room) => {
      roomNameMap[room._id] = room.name;
    });
```

Figura 71 - Parte inicial da função `fetchFilteredAppointments()`

Na figura 71, é apresentada a parte inicial da função `fetchFilteredAppointments` e que é utilizada para listar as reuniões que irão aparecer em uma determinada página com base em uma query inserida na barra de pesquisa ou na query vazia que devolverá todas as reuniões existentes na página atual.

A variável `offset` serve para definir qual é o ponto a partir do qual as reuniões serão extraídas para a página atual, sendo calculado através da subtração do número da página atual por 1 e a multiplicação do resultado da operação anterior pela constante `'ITEMS_PER_PAGE'`.

No início do bloco *try*, começa-se por obter os dados sobre as salas e reuniões que se encontram armazenados nas respetivas APIs. Caso exista algum erro nesse processo, é enviado um erro e passa-se para o bloco *catch* que se encontra na figura 69.

```

} catch (error) {
  console.error('Database Error:', error);
  throw new Error('Failed to fetch the appointments table.');
```

Figura 72 - Bloco *catch*

Depois da obtenção dos dados, é declarada uma constante chamada *roomNameMap* como um objeto vazio, cujas chaves e valores associados serão do tipo *string*. Posteriormente a lista de salas é iterada e para cada sala, o id da sala passa a ser a chave e o nome da sala passa a ser o valor associado à chave. Isto serve para que posteriormente seja listada na tabela de reuniões o nome da sala onde está marcada a reunião para que as reuniões sejam filtradas com base no nome da sala.

```

appointments.forEach(async (appointment: Appointment) => {
  const id = appointment._id;
  const appointmentEnd = new Date(appointment.end);

  const endTime = appointmentEnd.toISOString().split('T')[1].substring(0, 5);
  const [endHours, endMinutes] = endTime.split(':').map(Number);
  const end = new Date(appointment.date).setHours(endHours, endMinutes);

  const currentDate = new Date();

  if(end < currentDate.getTime()) {
    try {
      const res = await fetch(`http://localhost:3000/api/appointments?_id=${id}`, {
        method: "DELETE",
      });
      if (!res.ok) {
        throw new Error(`Failed to delete appointment with ID: ${id}`);
      }
      console.log(`Appointment with ID: ${id} deleted.`);
    } catch (error) {
      console.error(`Error deleting appointment with ID: ${id}`, error);
    }
  }
});
```

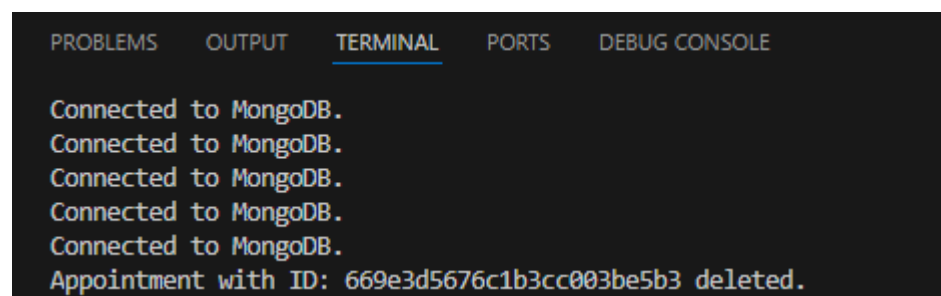
Figura 74 - Iteração da lista de reuniões para eliminar as reuniões passadas

O código da figura 74 começa por iterar a lista de reuniões e por obter de seguida o id da reunião a ser iterada e o horário do fim da reunião é transformado em um objeto do tipo *Date* para ser tratado nas linhas seguintes e também porque estava armazenado na API como um objeto do tipo *string*. Esse objeto do tipo *Date* é convertido em uma string ISO de onde é

extraída a parte da hora que se encontra a seguir ao 'T'. De seguida as horas e minutos são separados no formato "HH:MM" e convertidos para números inteiros.

É criado um objeto do tipo *Date* em que a parte da data corresponde à data da reunião e a parte do horário às horas e minutos que foram tratados acima. Isto foi feito para evitar problemas relacionados com mudanças para o horário de verão o que poderia afetar os cálculos e adiantar em uma hora os horários das reuniões caso este código não tivesse sido implementado assim.

Um objeto *Date* que representa o horário atual é criado para que seja comparado com o horário de fim da reunião a ser iterada. Se for, entra no ciclo dentro do *if*. Dentro do *if* existe um bloco *try-catch* onde é enviada uma requisição 'DELETE' para a API no endereço que incluirá o id da reunião em questão. Se a requisição for bem-sucedida, aparecerá uma mensagem no terminal do *Visual Studio Code*, com informações sobre o id da reunião que foi eliminada, como aparece na figura 75.



```

PROBLEMS  OUTPUT  TERMINAL  PORTS  DEBUG CONSOLE

Connected to MongoDB.
Connected to MongoDB.
Connected to MongoDB.
Connected to MongoDB.
Connected to MongoDB.
Appointment with ID: 669e3d5676c1b3cc003be5b3 deleted.
  
```

Figura 75 - Mensagem sobre a eliminação de reuniões passadas no terminal

Subject	Organizer	Room	Date	Start	End	Private	
Tuesday Appointment	Diogo Ferreira	Arkanoid Room (1.3)	2024-08-20	10:30	12:30	No	Update Delete
Wednesday Appointment	Diogo Ferreira	ChuckieEgg Room (1.3)	2024-08-21	10:00	12:00	No	Update Delete
Morning Appointment	Diogo Ferreira	PacMan Room (6.12)	2024-08-22	09:30	11:30	No	Update Delete
Afternoon Appointment	Diogo Ferreira	Pang Room (2.4)	2024-08-22	14:30	16:30	No	Update Delete
Friday Appointment	Diogo Ferreira	DonkeyKong Room	2024-08-23	10:00	12:00	No	Update Delete
Arkanoid Appointment	Diogo Ferreira	Arkanoid Room (1.3)	2024-08-26	10:30	12:30	No	Update Delete



Figura 76 - Tabela de reuniões atualizada

A tabela de reuniões atualizada que aparece na figura 76 irá ser atualizada depois da API receber informação atualizada da base de dados, embora demore alguns segundos devido ao ISR em que só é possível o utilizador receber novas informações depois de passado um período que está definido no código como *revalidationTime* e seja preciso dar *refresh* ao *browser* para que apareçam as informações atualizadas.

```
const filteredAppointments = appointments
  .filter((appointment: Appointment) =>
    | appointment.subject.includes(query) || roomNameMap[appointment.room_id].includes(query)
  )
  .map((appointment: Appointment): AppointmentsTableType => {
    const roomName = roomNameMap[appointment.room_id];

    const appointmentDate = new Date(appointment.date);
    const appointmentStart = new Date(appointment.start);
    const appointmentEnd = new Date(appointment.end);

    const date = appointmentDate.toISOString().split('T')[0];
    const startTime = appointmentStart.toISOString().split('T')[1].substring(0, 5);
    const endTime = appointmentEnd.toISOString().split('T')[1].substring(0, 5);

    return {
      _id: appointment._id,
      subject: appointment.subject,
      organizer: appointment.organizer,
      roomName: roomName,
      date: date,
      start: startTime,
      end: endTime,
      open: appointment.open
    }
  });

const paginatedAppointments = filteredAppointments.slice(offset, offset + ITEMS_PER_PAGE);
return paginatedAppointments;
```

Figura 77 - Iteração da lista de reuniões na função *fetchFilteredAppointments()*

Na figura 77, começa por ser aplicado um filtro à lista de reuniões. Esse filtro verifica se o assunto ou o nome de cada sala, que é obtido a partir do objeto *roomNameMap*, contém a *query* em questão e apenas as reuniões que contiverem essa *query* serão mostradas.

De seguida, as reuniões que foram filtradas são transformadas em objetos do tipo *AppointmentsTableType* para sejam listadas na tabela com as variáveis necessárias.

As variáveis necessárias são tratadas para que sejam posteriormente listadas. O valor associado à chave do id da sala no objeto *roomNameMap* é passado para uma constante chamada *roomName*, a data e os horários de começo e fim da reunião são transformados em variáveis do tipo *Date* e de seguida são listados em formato ISO do tipo *string*. Isto serve para que o que está antes do 'T' seja associado à data de reunião no formato 'YYYY-MM-DD' e que

sejam selecionados os 5 primeiros valores depois do 'T' no caso dos horários de começo e de fim, sendo formatados para o formato 'HH:MM'.

No final, é criada uma constante chamada *paginatedAppointments* que retornará as reuniões já tratadas a partir do índice *offset* e terminará no índice imediatamente anterior à soma do *offset* com o número de reuniões por página. De seguida, ele retornará essas reuniões que serão mostradas em uma determinada página.

```
return (
  <div className="mt-6 flow-root">
    <div className="inline-block min-w-full align-middle">
      <div className="rounded-lg bg-gray-50 p-2 md:pt-0">
        <div className="md:hidden">
          {appointments?.map((appointment: {
            _id: string;
            subject: string;
            organizer: string;
            roomName: string;
            date: string;
            start: string;
            end: string;
            open: string;
          })) => (
            <div
              key={appointment._id}
              className="mb-2 w-full rounded-md bg-white p-4"
            >
              <div className="flex items-center justify-between border-b pb-4">
                <div>
                  <p className="text-lg text-gray-500">{appointment.subject}</p>
                  <p className="text-lg text-gray-500">{appointment.organizer}</p>
                  <p className="text-lg text-gray-500">{appointment.roomName}</p>
                  <p className="text-lg font-gray-500">{appointment.date}</p>
                  <p className="text-lg text-gray-500">{appointment.start}</p>
                  <p className="text-lg text-gray-500">{appointment.end}</p>
                </div>
                <AppointmentStatus status={appointment.open} />
              </div>
              <div className="flex w-full items-center justify-between pt-4">
                <div className="flex justify-end gap-2">
                  <UpdateAppointment id={appointment._id} />
                  <DeleteAppointment id={appointment._id} />
                </div>
              </div>
            </div>
          )
        </div>
      </div>
    </div>
  )
)
```

Figura 78 - Definição das variáveis a serem mostradas na tabela

As reuniões vão ser mapeadas individualmente, verificando a reunião atual existe. De seguida são definidas as variáveis que serão retornadas e o formato das mesmas. O id de cada reunião é envolvido em um div, embora essa variável não seja mostrada na tabela. Os valores

das variáveis são definidos em vários parágrafos que irão corresponder às colunas e encontram dentro de um div. O estado da reunião encontra-se dentro de um div mais externo pois é uma variável que tem apenas 2 valores distintos que serão tratados na classe *AppointmentStatus*. Em outro div externo, são criados os botões responsáveis pela edição e eliminação das reuniões ao qual está associado o id da reunião.

```
<table className="hidden min-w-full text-gray-900 md:table">
  <thead className="rounded-lg text-left text-sm font-normal">
    <tr>
      <th scope="col" className="px-4 py-5 font-medium sm:pl-6">
        Subject
      </th>
      <th scope="col" className="px-3 py-5 font-medium">
        Organizer
      </th>
      <th scope="col" className="px-3 py-5 font-medium">
        Room
      </th>
      <th scope="col" className="px-3 py-5 font-medium">
        Date
      </th>
      <th scope="col" className="px-3 py-5 font-medium">
        Start
      </th>
      <th scope="col" className="px-3 py-5 font-medium">
        End
      </th>
      <th scope="col" className="px-3 py-5 font-medium">
        Private
      </th>
      <th scope="col" className="relative py-3 pl-6 pr-3">
        <span className="sr-only">Edit</span>
      </th>
    </tr>
  </thead>
```

Figura 79 – Criação da tabela de reuniões

Na figura 79 é criada a tabela de reuniões e que cada coluna é associada às variáveis que serão mostradas.

4.1.6 Criar reunião

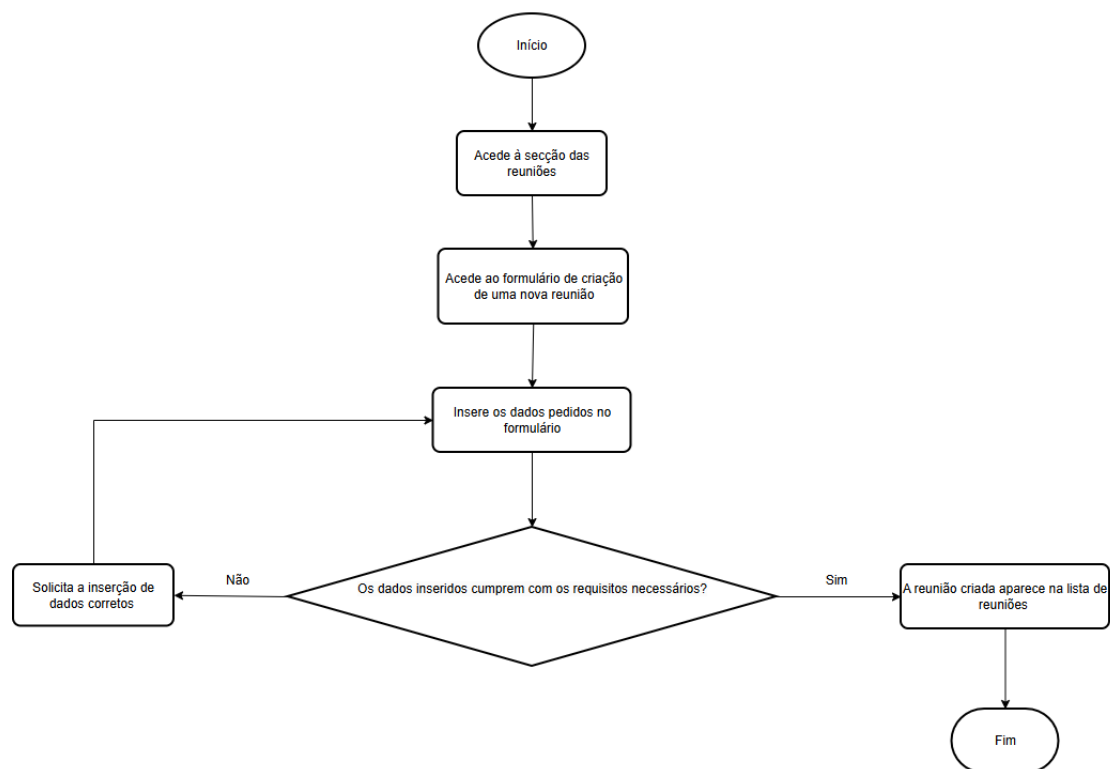


Figura 80 - Fluxograma da criação da reunião

Na figura 80, está representado o fluxograma responsável pela criação da reunião. O utilizador já logado começa por aceder à secção da aplicação responsável pelas reuniões caso ainda não esteja lá. De seguida, acede ao formulário de criação de uma nova reunião e insere os dados pedidos. O sistema verifica se os dados estão no formato correto e cumprem com os requisitos necessários. Caso isso seja verdade, a reunião é criada e aparece na tabela das reuniões atualizada com a nova reunião. Caso contrário, o sistema solicita ao utilizador que corrija os dados que estão errados.

A criação da reunião será efetuada em um formulário de criação que é criado e invocado no ficheiro *page.tsx* no caminho “/dashboard/appointments/create” que aparece na figura 81.

```
'use client';

import Form from '@app/ui/appointments/create-form';
import Breadcrumbs from '@app/ui/appointments/breadcrumbs';
import { fetchRooms } from '@app/lib/data';

export default async function Page() {
  const rooms = await fetchRooms();

  return (
    <main>
      <Breadcrumbs
        breadcrumbs={[
          { label: 'Appointments', href: '/dashboard/appointments' },
          {
            label: 'Create Appointments',
            href: '/dashboard/appointments/create',
            active: true,
          },
        ]}
      />
      <Form rooms={rooms} />
    </main>
  );
}
```

Figura 81 - Classe *page.tsx*

Esta classe possui o ‘*use client*’ pois será acionada quando o utilizador clica no botão *Create* ao lado da barra de pesquisa. Depois, a função *fetchRooms()* é chamada que permite ao utilizador escolher uma sala quando estiver a preencher o formulário. O *Form* que é importado da classe *create-form.tsx* na pasta “/ui/appointments” representa o formulário de criação de uma reunião.

A função *fetchRooms()* encontra-se representada na figura 82.

```

export async function fetchRooms() {
  noStore();

  try {
    const res = await fetch('http://localhost:3000/api/rooms');
    if (!res.ok) {
      throw new Error(`HTTP error! Status: ${res.status}`);
    }
    const rooms = await res.json();

    const allRooms: RoomField[] = rooms.map((room: Room): RoomField => {
      return {
        _id: room._id,
        name: room.name,
      };
    });

    return allRooms;
  } catch (error) {
    console.error('Database Error:', error);
    throw new Error('Failed to fetch all rooms.');
```

Figura 82 - Função `fetchRooms()`

A função começa por adquirir a lista de salas presente na API das salas e caso exista um erro, é enviada para o bloco *catch*. Depois da lista das salas ter sido tratada, a lista de salas é mapeada e são filtrados o `_id` e o nome das salas todas que serão armazenados em um novo tipo chamado *RoomField* que inclui essas 2 variáveis. A lista de salas armazenada nesse tipo é retornada.

Este tipo será utilizado no *create-form.tsx* que está na pasta *"/ui/dashboard"*. Essa classe passa por parâmetro a lista de salas que foi criada no código que aparece na figura 69 para que o utilizador possa escolher uma sala no formulário. A constante *getMinDate()* definida antes da função serve para que a data inserida pelo utilizador no formulário nunca seja menor que a data atual. A classe é do tipo *'use-client'* pois o cliente irá interagir com a classe quando preenche o formulário. Isto tudo aparece na figura 80.

```
'use client';

import { RoomField } from '@app/lib/definitions';
import Link from 'next/link';
import { Button } from '@app/ui/button';
import { createAppointment, State } from '@app/lib/actions';
import { useRouter } from 'next/navigation';
import { useEffect, useState } from 'react';

const getMinDate = () => {
  const today = new Date();
  const minDate = today.toISOString().split('T')[0];
  return minDate;
};

export default function CreateAppointmentForm({
  rooms,
}): {
  rooms: RoomField[];
}) {
```

Figura 83 - Parte inicial do create-form.tsx

Figura 84 - Formulário de criação

No formulário de criação na figura 84, insere-se o assunto, escolhe-se a sala, a data através do ícone no lado direito e os horários de começo e fim através do ícone do direito, e o estado escolhemos uma de duas opções. A variável date é do tipo *date* e as variáveis dos horários são do tipo *time*, estando definido no código, tal como aparece nas figuras 85, 86 e 87, permitindo assim o aparecimento dos ícones no lado direito das respectivas barras.

```
type="date"
```

Figura 85 - Tipo da variável date no create-form.tsx

```
type="time"
```

Figura 86 - Tipo da variável *start* no *create-form.tsx*

```
type="time"
```

Figura 87 - Tipo da variável *end* no *create-form.tsx*

Após a inserção dos dados no formulário, os dados serão validados.

```
const [state, setState] = useState<State>(initialState);
const [result, setResult] = useState<any>(null);
const router = useRouter();

useEffect(() => {
  if (result?.success) {
    router.push('/dashboard/appointments');
  } else if (result) {
    setState(prevState => ({
      ...prevState,
      errors: result.errors,
      message: result.message
    }));
  }
}, [result, router]);

const handleSubmit = (event: React.FormEvent) => {
  event.preventDefault();
  const formData = new FormData(event.target as HTMLFormElement);

  const submitForm = async () => {
    const res = await createAppointment(state, formData);
    setResult(res);
  };

  submitForm();
};

const minDate = getMinDate();
```

Figura 88 - Tratamento da submissão do formulário

A função *handleSubmit*, na figura 88, trata da submissão do formulário. Esta função é acionada quando o utilizador clica no botão *create*. Os dados inseridos são passados para um *formData* e enviados posteriormente para a função *createAppointment* na classe *actions.ts* na pasta *"/lib"* onde serão validados. O resultado dessa requisição é atualizado. Se estiver tudo bem e for bem-sucedido, o utilizador é redirecionado para a página onde aparece a tabela

com a lista de reuniões atualizada. Caso contrário serão listados os erros, como por exemplo variáveis por inserir no formulário ou variáveis que não cumprem com os requisitos definidos no zod.

```

'use server';

import { z } from 'zod';
import { transformTimeToDate } from './utils';
import { getServerSession } from 'next-auth';

const AppointmentSchema = z.object({
  _id: z.string().optional(),
  subject: z.string()
    .min(5, { message: 'Subject must be at least 5 characters long.' })
    .max(30, { message: 'Subject must be at most 30 characters long.' }),
  organizer: z.string().optional(),
  room_id: z.string({
    required_error: 'Please select a room.',
  }),
  date: z.coerce.date(),
  start: z.string().regex(/^\d{2}:\d{2}$/, {
    message: 'Please insert a start time.',
  }),
  end: z.string().regex(/^\d{2}:\d{2}$/, {
    message: 'Please insert an end time.',
  }),
  open: z.enum(['True', 'False'], {
    required_error: 'Please select an appointment status.',
  })
}).superRefine((data, ctx) => {
  const [startHours, startMinutes] = data.start.split(':').map(Number);
  const [endHours, endMinutes] = data.end.split(':').map(Number);
  const startTime = new Date(data.date).setHours(startHours, startMinutes);
  const endTime = new Date(data.date).setHours(endHours, endMinutes);
  const currentTime = new Date();

  if (data.date.toDateString() === currentTime.toDateString()) {
    if (startTime <= currentTime.getTime()) {
      ctx.addIssue({
        code: z.ZodIssueCode.custom,
        message: 'Start time must be greater than the current time.',
        path: ['start'],
      });
    }
  }

  if (endTime <= startTime) {
    ctx.addIssue({
      code: z.ZodIssueCode.custom,
      message: 'End time must be greater than start time.',
      path: ['end'],
    });
  }
});

```

Figura 89 - Critérios de validação do formulário

Na figura 89, aparecem os critérios de validação do formulário que caso existam erros aquando do preenchimento do formulário de criação ou de edição serão lançados para que o utilizador os corrija. A classe é do tipo 'use-server' pois vai lidar com requisições à API, neste caso, a inserção de novas reuniões e a edição de reuniões já existentes.

```
export const createAppointment = async (prevState: State, formData: FormData) => {
  const validatedFields = AppointmentSchema.safeParse(
    Object.fromEntries(formData.entries())
  );

  const session = await getServerSession();
  const username = session?.user?.name;

  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
      message: 'Missing Fields. Failed to Create Appointment.',
      success: false,
    };
  }

  const { subject, room_id, date, start, end, open } = validatedFields.data;

  const transformedDate = date.toISOString();
  const transformedStart = transformTimeToDate(start, date);
  const transformedEnd = transformTimeToDate(end, date);

  const appointment = {
    subject,
    organizer: username ?? '',
    date: transformedDate,
    start: transformedStart,
    end: transformedEnd,
    open,
    room_id,
  };
};
```

Figura 90 - Validação e transformação dos dados do formulário de criação

Na figura 90, os dados inseridos no formulário de criação são validados através do método *safeParse()* onde serão verificados os erros desse formulário de acordo com os esquemas de validação do zod que está representado na figura 76. Depois, é verificado se a validação foi bem sucedida. Se sim, continua para o tratamento dos dados e caso contrário, serão mostrados os erros.

O nome do utilizador que está logado é recuperado através da invocação da função *getServerSession()*. Os campos todos são extraídos e transformados no formato necessário para serem inseridos na API e base de dados, no caso da data e horários de começo e de fim.

O organizador é associado ao nome do utilizador logado. Este processo também é igual na edição de reuniões.

```
try {
  const saveAppointment = async (appointmentData: {
    subject: string;
    organizer: string;
    date: string;
    start: string;
    end: string;
    open: "True" | "False";
    room_id: string;
  }) => {
    const response = await fetch(`http://localhost:3000/api/appointments`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(appointmentData),
    });

    if (!response.ok) {
      throw new Error('Failed to save appointment');
    }

    return response.json();
  };

  await saveAppointment(appointment);
  return { success: true, errors: {}, message: null };
} catch (error) {
  console.error('Error creating appointment:', error);
  return { message: "Failed to create appointment.", success: false, errors: {} };
};
```

Figura 91 - Inserção da reunião na API

Na figura 91, a reunião com os dados já tratados, é enviada para o *endpoint* da API através de uma requisição 'POST' e o objeto será convertido para o formato JSON para ser armazenado na API. Se existirem erros, a função é enviada para o bloco catch, onde serão listados os erros.

4.1.7 Editar reunião

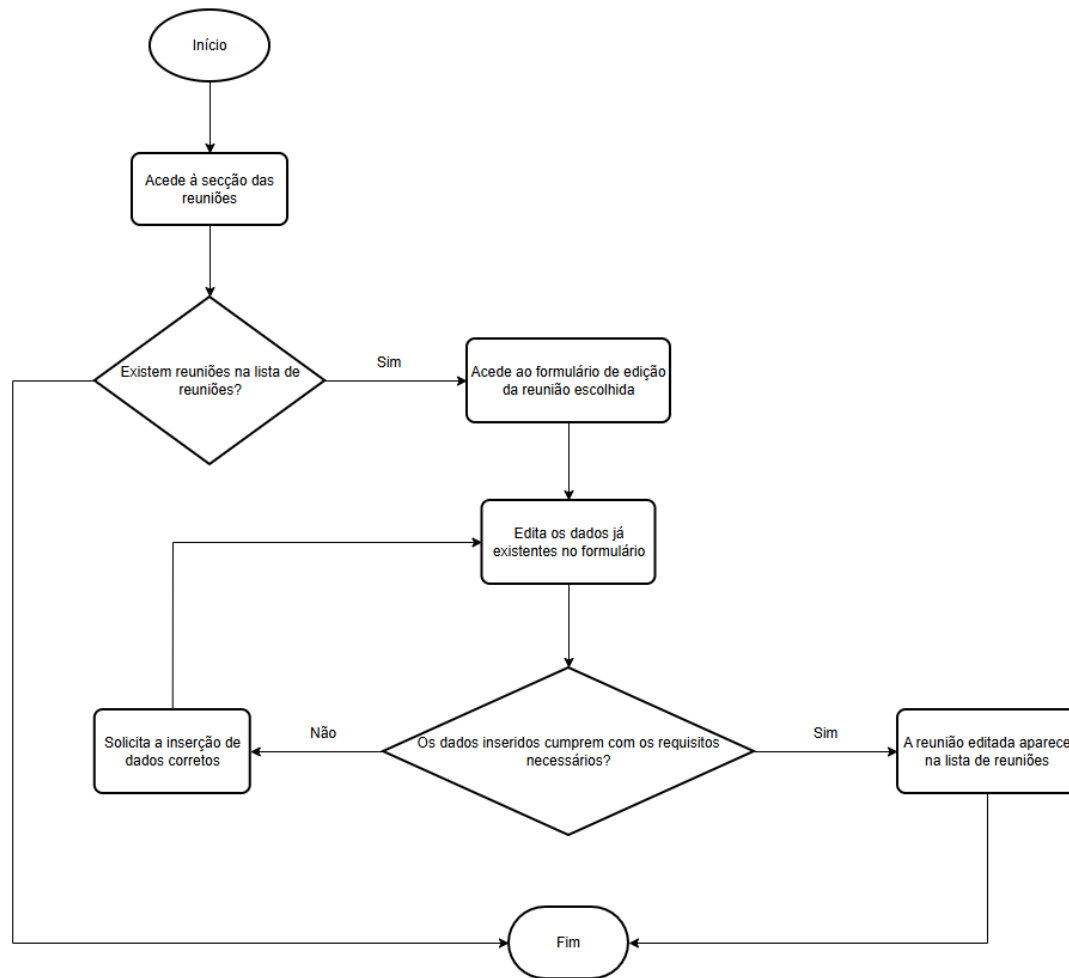


Figura 92 - Fluxograma da edição de uma reunião

Na figura 92, está representado o fluxograma responsável pela edição de uma reunião. O utilizador já logado começa por aceder à secção da aplicação responsável pelas reuniões caso ainda não esteja lá. Se não existirem reuniões na tabela, o processo é terminado pois não vão ser editadas reuniões que não existem. Caso contrário, o utilizador escolhe a reunião a ser editada e acede ao formulário de edição dessa reunião onde estão representados os dados associados a essa reunião. De seguida, o utilizador edita os campos que achar necessário e quando este processo estiver concluído, o sistema verifica se os dados estão no formato correto e cumprem com os requisitos necessários. Caso isso seja verdade, a reunião é atualizada e aparece na tabela das reuniões já atualizada. Caso contrário, o sistema solicita ao utilizador que corrija os dados que estão errados.

A edição da reunião será efetuada em um formulário de criação que é criado e invocado no ficheiro *page.tsx* no caminho `"/dashboard/appointments/[id]/edit"` que aparece na figura

93. Neste caso aparece o id pois é necessário saber o id para sabermos qual a função a ser selecionada para ser editada.

```
import Form from '@app/ui/appointments/update-form';
import Breadcrumbs from '@app/ui/appointments/breadcrumbs';
import { fetchAppointmentById, fetchRooms } from '@app/lib/data';
import { notFound } from 'next/navigation';

export default async function Page({ params }: { params: { id: string } }) {
  const id = params.id;
  const [appointment, rooms] = await Promise.all([
    fetchAppointmentById(id),
    fetchRooms(),
  ]);

  if (!appointment) {
    notFound();
  }

  return (
    <main>
      <Breadcrumbs
        breadcrumbs={[
          { label: 'Appointments', href: '/dashboard/appointments' },
          {
            label: 'Edit Appointment',
            href: `/dashboard/appointments/${id}/edit`,
            active: true,
          },
        ]}
      />
      <Form initialAppointment={appointment} rooms={rooms} />
    </main>
  );
}
```

Figura 93 - Classe page.tsx

Esta classe possui o *'use client'* pois será acionada quando o utilizador clica no botão *Edit* na linha da tabela associada à reunião a ser editada. Depois, as funções *fetchRooms()* e *fetchAppointmentById()* são chamadas, sendo que a última permite ao utilizador escolher a reunião a ser editada e preencher o formulário de edição com os dados da reunião em questão. O *Form* que é importado da classe *update-form.tsx* na pasta *"/ui/appointments"* representa o formulário de criação de uma reunião.

```

export async function fetchAppointmentById(id: string) {
  noStore();

  try {
    const appointment_res = await fetch('http://localhost:3000/api/appointments', { next: { revalidate: revalidationTime } });
    if (!appointment_res.ok) {
      throw new Error(`HTTP error! Status: ${appointment_res.status}`);
    }
    const appointments = await appointment_res.json();

    const filteredAppointment = appointments
      .filter((appointment: Appointment) =>
        appointment._id.includes(id)
      )
      .map((appointment: Appointment): AppointmentForm => {
        return {
          _id: appointment._id,
          subject: appointment.subject,
          date: appointment.date,
          start: appointment.start,
          end: appointment.end,
          room_id: appointment.room_id,
          open: appointment.open,
        };
      });

    return filteredAppointment[0];
  } catch (error) {
    console.error('Database Error:', error);
    throw new Error('Failed to fetch appointment.');
```

Figura 94 - Função *fetchAppointmentById*

A função *fetchAppointmentById()* encontra-se representada na figura 94.

A função começa por adquirir a lista de reuniões presente na API das reuniões e caso exista um erro, é enviada para o bloco *catch*.

De seguida é criada uma instância de *filteredAppointment*, onde a lista de reuniões será filtrada pelo id. Caso o id exista será devolvida a reunião em questão com as variáveis todas em questão à exceção do organizador que está definido com o nome do utilizador que está logado. Estas informações seriam armazenadas em um tipo chamado *AppointmentForm* que seria utilizado para preencher o formulário de edição com as informações da reunião a ser editada. Finalmente seria retornada a reunião a ser editada.

Figura 95 - Formulário de edição

Na figura 95, aparece o formulário de edição da reunião que já tem os dados da reunião a ser editada para que o utilizador saiba quais são os dados e a edição seja mais simples e cómoda. Na função *update-form.tsx* aparecem trechos de código como na figura 93 que permitem que isso aconteça.

```
defaultValue={initialAppointment?.subject || ''}
```

Figura 96 - Exemplo de variável com dados definidos da reunião no formulário de edição

Os dados existentes são obtidos a partir do id da reunião e os dados editados são posteriormente validados.

```
useEffect(() => {
  if (result?.success) {
    router.push('/dashboard/appointments');
  } else if (result) {
    setState(prevState => ({
      ...prevState,
      errors: result.errors,
      message: result.message
    }));
  }
}, [result, router]);

const handleSubmit = (event: React.FormEvent) => {
  event.preventDefault();
  const formData = new FormData(event.target as HTMLFormElement);

  if (initialAppointment && initialAppointment._id) {
    formData.append('_id', initialAppointment._id);
    formData.append('organizer', initialAppointment.organizer);

    const submitForm = async () => {
      const res = await updateAppointment(initialAppointment._id, state, formData);
      setResult(res);
    };

    submitForm();
  } else {
    setState(prevState => ({
      ...prevState,
      message: 'Appointment ID is missing.'
    }));
  }
};
```

Figura 97 - Tratamento dos dados da reunião da reunião a ser editada e da sua submissão

A função *handleSubmit*, na figura 97, trata da aquisição dos dados da reunião e da submissão do formulário. Esta função é acionada quando o utilizador clica no botão *edit* na lista de reuniões. Inicialmente, é verificado se existe uma reunião com aquele id específico. Se sim, os dados atualizados, passados por um *formData* são enviados para a função

updateAppointment na classe *actions.ts* que também recebe o id como parâmetro pois irá necessitar do mesmo para atualizar a reunião no *endpoint* da API correspondente a esse id. Nessa classe, os dados são validados de acordo com os esquemas de validação do zod. O resultado dessa requisição é atualizado. Se estiver tudo bem e for bem-sucedido, o utilizador é redirecionado para a página onde aparece a tabela com a lista de reuniões atualizada. Caso contrário serão listados os erros, como por exemplo variáveis por inserir no formulário ou variáveis que não cumprem com os requisitos definidos no zod.

A validação e a transformação dos dados na função *updateAppointment* será feita da mesma forma que na função *createAppointment*.

Na figura 98, a reunião editada com os dados já tratados, é enviada para o *endpoint* da API correspondente ao id da reunião através de uma requisição 'PUT' e o objeto será convertido para o formato JSON para ser armazenado na API. Se existirem erros, a função é enviada para o bloco *catch*, onde serão listados os erros.

```
try {
  const editAppointment = async (appointmentData: {
    _id: string;
    subject: string;
    organizer: string;
    date: string;
    start: string;
    end: string;
    open: "True" | "False";
    room_id: string;
  }) => {
    const response = await fetch(`http://localhost:3000/api/appointments/${appointmentData._id}`, {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(appointmentData),
    });

    if (!response.ok) {
      throw new Error('Failed to update appointment');
    }

    return response.json();
  };

  await editAppointment(updatedAppointment);
  return { success: true, errors: {}, message: null };
} catch (error) {
  console.error('Error creating appointment:', error);
  return { message: "Failed to create appointment.", success: false, errors: {} };
}
```

Figura 98 - Inserção da reunião editada na API

4.1.8 Eliminar reunião

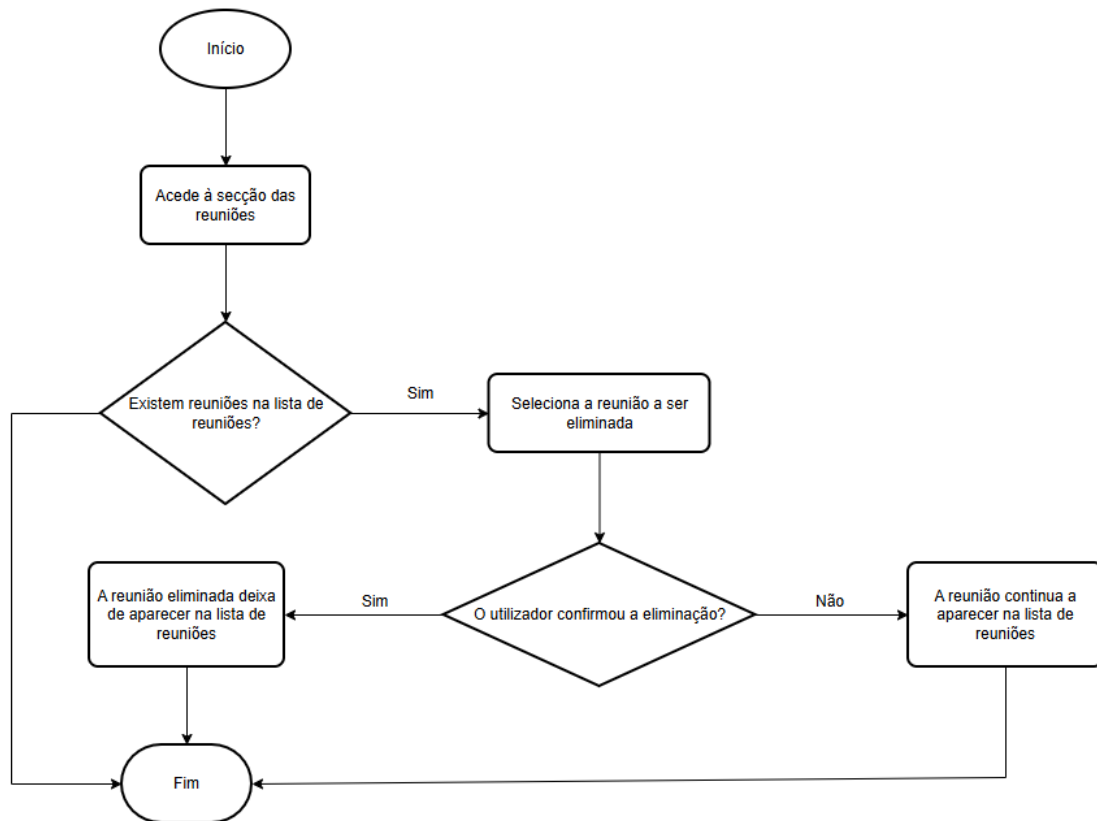


Figura 99 - Fluxograma da eliminação de uma reunião

Na figura 99, está representado o fluxograma responsável pela eliminação de uma reunião. O utilizador já logado começa por aceder à secção da aplicação responsável pelas reuniões caso ainda não esteja lá. Se não existirem reuniões na tabela, o processo é terminado pois não vão ser eliminadas reuniões que não existem. Caso contrário, o utilizador escolhe a reunião a ser eliminada. De seguida, o utilizador é questionado pelo sistema para averiguar se deseja mesmo eliminar a reunião que selecionou. Se o utilizador confirmar a eliminação, a reunião é eliminada e a tabela das reuniões aparece já atualizada sem a reunião eliminada. Caso contrário, não acontece nada e a reunião em questão continua a existir e a ser representada na tabela das reuniões.

Na função *DeleteAppointments*, que se encontra na figura 97, recebe por parâmetro o id da reunião a ser eliminada que é uma variável do tipo string. O *useRouter()* que aparece na linha seguinte é um *hook* que serve para dar *refresh* à página após a eliminação da reunião.


```

export function DeleteAppointment({ id }: { id: string }) {
  const router = useRouter();

  const removeAppointment = async () => {
    const confirmed = confirm("Are you sure?");

    if (confirmed) {
      const res = await fetch(`http://localhost:3000/api/appointments?_id=${id}`, {
        method: "DELETE",
      });

      if (res.ok) {
        router.refresh();
      }
    }
  };

  return (
    <button
      onClick={removeAppointment}
      className="flex h-10 items-center rounded-lg bg-red-600 px-4 text-sm font-medium text-white"
    >
      <span className="hidden md:block">Delete</span>{' '}
      <TrashIcon className="h-5 md:m1-4" />
    </button>
  );
}

```

Figura 100 - Função DeleteAppointments

A função *removeAppointment* é assíncrona e será chamada quando o utilizador clicar no botão de eliminação. Antes de a reunião selecionada ser apagada, a função *confirm* permite que apareça um *pop-up* que irá perguntar ao utilizador se deseja mesmo apagar a reunião, tal como parece na figura 101.



Figura 101 - Pop-up de eliminação

Caso o utilizador carregue no 'OK', será enviada uma requisição 'DELETE' para a API no endereço que incluirá o id da reunião em questão. Se a requisição for bem-sucedida, a página será atualizada sem a reunião eliminada.

No JSX, é criado o botão de eliminação que invocará a função *removeAppointment* sempre que for clicado, é vermelho e possui um pequeno ícone do caixote do lixo para que se possa distinguir dos outros botões, neste caso de criação e edição.

4.2 Testes

Nesta secção vão ser abordados os vários tipos de testes realizados para verificar se o projeto desenvolvido cumpre com os requisitos impostos pelo cliente e pela organização.

4.2.1 Testes de integração

Os testes de integração [13] asseguram que todos os componentes de uma aplicação funcionam em conjunto como é suposto e verificar se a integração desse mesmos componentes satisfaz os requisitos do utilizador e os requisitos técnicos da organização. Para isso, foi utilizado o Postman para verificar se as APIs funcionam corretamente.

No caso das reuniões, foram desenvolvidos testes para criar, obter por ID, editar e eliminar e obter a lista de todas as reuniões. No caso das salas, foram criados testes para obter uma sala por ID e obter a lista de todas as salas. Nestes testes a resposta de cada API foi testada para garantir que todos os requisitos estão completamente funcionais.

Os testes efetuados vão ser analisados abaixo para se possa entender melhor como foram feitos.

4.2.1.1 Criar reunião

```
{
  "subject": "Meeting 3",
  "organizer": "Diogo Ferreira",
  "date": "2024-07-08T00:00:00.000+00:00",
  "start": "2024-07-08T14:00:00.000+00:00",
  "end": "2024-07-08T16:00:00.000+00:00",
  "open": "True",
  "room_id": "661fd8c1e9876c829a1f2741"
}
```

Figura 102 - Body Criar Reunião

Na figura 102, encontra-se um exemplo de uma reunião a ser criada no Postman.

```

1  const responseJson = pm.response.json();
2
3  // Code Passed
4  pm.test("Status code is 201: Appointment created", function () {
5    pm.response.to.have.status(201);
6  });

```

Figura 103 - Teste de Criar Reunião

Em caso de sucesso, é devolvido o código 201 juntamente com a frase de sucesso no Postman que indica que a reunião foi criada com sucesso como aparece na figura 103.

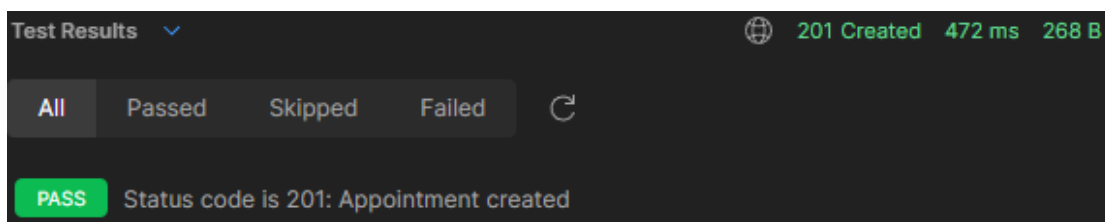


Figura 104 - Código de sucesso da criação da reunião

```

{
  "id": "66852de1f959136bae4e32a",
  "subject": "Meeting 3",
  "organizer": "Diogo Ferreira",
  "date": "2024-07-08T00:00:00.000Z",
  "start": "2024-07-08T14:00:00.000Z",
  "end": "2024-07-08T16:00:00.000Z",
  "open": "True",
  "room_id": "661fd8c1e9876c829a1f2741"
}

```

Figura 105 - Reunião criada

Na figura 105, aparece a reunião criada no *endpoint* da API em caso de sucesso.

4.2.1.2 Listar reuniões

```

const responseJson = pm.response.json();

// Code Passed
pm.test("Status code is 200", function () {
  pm.response.to.have.status(200);
});

```

Figura 106 - Teste de Listar Reuniões

No caso de listar as reuniões, é utilizado o comando GET juntamente com o caminho da aplicação na pasta da API onde se encontra a lista das reuniões. Caso o pedido tenha sucesso, é devolvido o código 200 tal como aparece na figura 107.

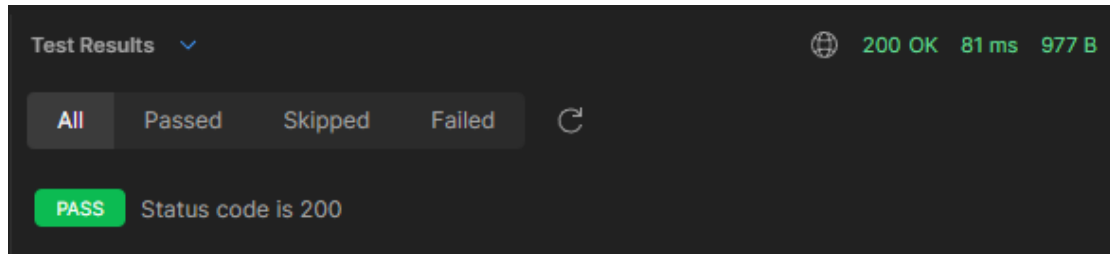


Figura 107 - Código de sucesso da listagem de reuniões

4.2.1.3 Obter reunião por ID

```
1  const responseJson = pm.response.json();
2
3  // Code Passed
4  pm.test("Status code is 200", function () {
5    pm.response.to.have.status(200);
6  });
```

Figura 108 - Teste Obter Reunião por ID

```
{
  "appointment": {
    "_id": "66852de1f959136beae4e32a",
    "subject": "Meeting 3",
    "organizer": "Diogo Ferreira",
    "date": "2024-07-08T00:00:00.000Z",
    "start": "2024-07-08T14:00:00.000Z",
    "end": "2024-07-08T16:00:00.000Z",
    "open": "True",
    "room_id": "661fd8c1e9876c829a1f2741"
  }
}
```

Figura 109 - Reunião Obtida por ID

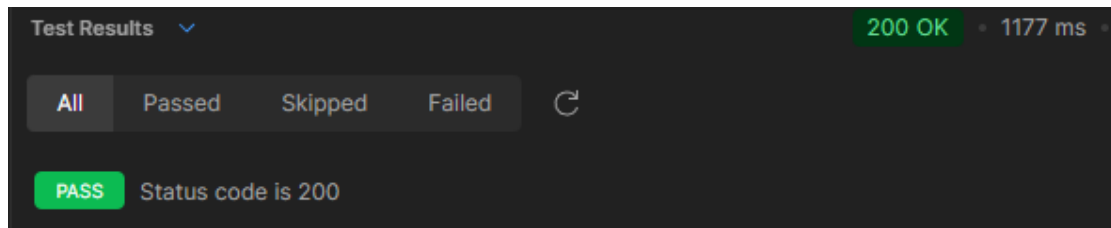


Figura 110 - Código de sucesso da listagem de uma reunião específica

Em caso de sucesso é devolvido o código 200 tal como aparece na figura 110.

4.2.1.4 Editar reunião

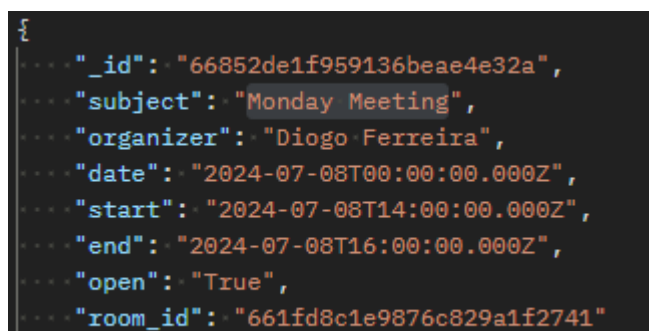


Figura 111 - Body Editar Reunião

Na figura 111 encontra-se um exemplo no Postman de uma reunião já existente em que foi alterado pelo menos um dos campos, à exceção do “_id”.

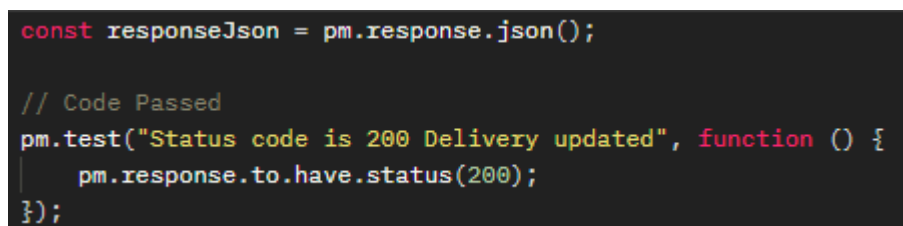


Figura 112 - Teste Editar Reunião

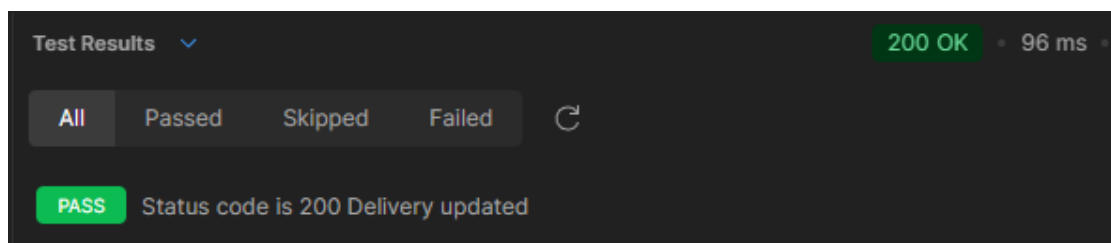


Figura 113 - Código de sucesso da edição de uma reunião

Em caso de sucesso é devolvido o código 200, tal como parece na figura 113, que indica que a reunião foi editada com sucesso.

```
{
  "_id": "66852de1f959136beae4e32a",
  "subject": "Monday Meeting",
  "organizer": "Diogo Ferreira",
  "date": "2024-07-08T00:00:00.000Z",
  "start": "2024-07-08T14:00:00.000Z",
  "end": "2024-07-08T16:00:00.000Z",
  "open": "True",
  "room_id": "661fd8c1e9876c829a1f2741"
}
```

Figura 114 - Reunião Editada

Na figura 114, aparece a reunião editada no *endpoint* da API em caso de sucesso.

4.2.1.5 Eliminar reunião

```
const responseJson = pm.response.json();

// Code Passed
pm.test("Status code is 200: Appointment deleted", function () {
  pm.response.to.have.status(200);
});
```

Figura 115 - Teste Eliminar Reunião

Em caso de sucesso é devolvido o código 200, tal como parece na figura 115, que indica que a reunião foi eliminada com sucesso.

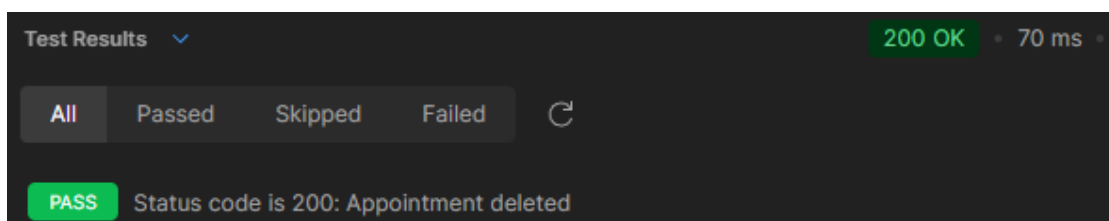


Figura 116 - Código de sucesso da eliminação de uma reunião

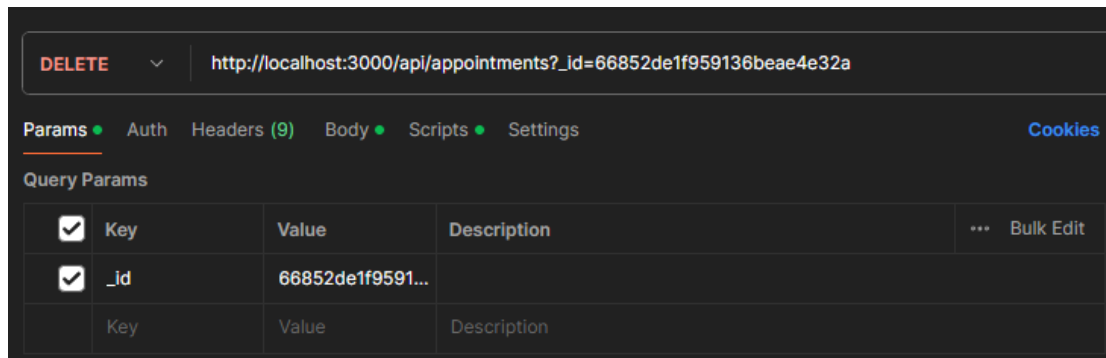


Figura 117 - Processo de eliminação de uma sala

Na figura 117, aparece como este processo é realizado no Postman. No URL, temos o endereço do endpoint da API no servidor onde a aplicação está a ser executada. O que aparece a seguir ao ponto de interrogação representa um parâmetro de *query* que identifica o *_id* da reunião a ser eliminada.

4.2.1.6 Listar salas

```
const responseJson = pm.response.json();

// Code Passed
pm.test("Status code is 200", function () {
  pm.response.to.have.status(200);
});
```

Figura 118 - Teste Listar Salas

No caso de listar as reuniões, é utilizado o comando GET juntamente com o caminho da aplicação na pasta da API onde se encontra a lista das salas. Caso o pedido tenha sucesso, é devolvido o código 200, tal como aparece na figura 119.

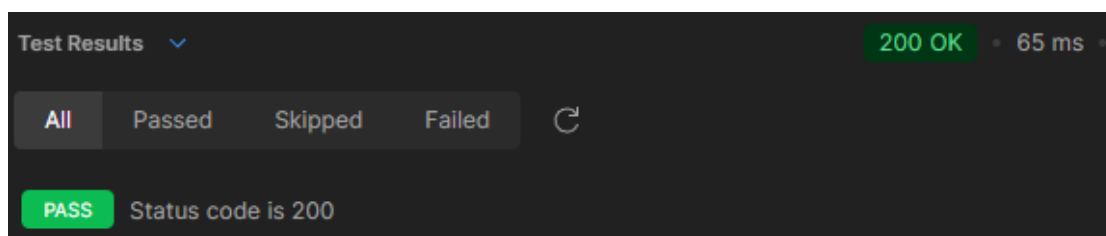


Figura 119 - Código de sucesso da listagem das salas

4.2.1.7 Obter sala por ID

```
1  const responseJson = pm.response.json();
2
3  // Code Passed
4  pm.test("Status code is 200", function () {
5    pm.response.to.have.status(200);
6  });
```

Figura 120 - Teste para Obter Sala por ID

```
{
  "appointment": {
    "_id": "661fd8c1e9876c829a1f2741",
    "name": "Arkanoid Room (1..3)",
    "roomAlias": "arkanoid-room-(1..3)",
    "email": "arkanoid@devscope.net"
  }
}
```

Figura 121 - Sala Obtido por ID

Em caso de sucesso é devolvido o código 200, tal como aparece na figura 122.

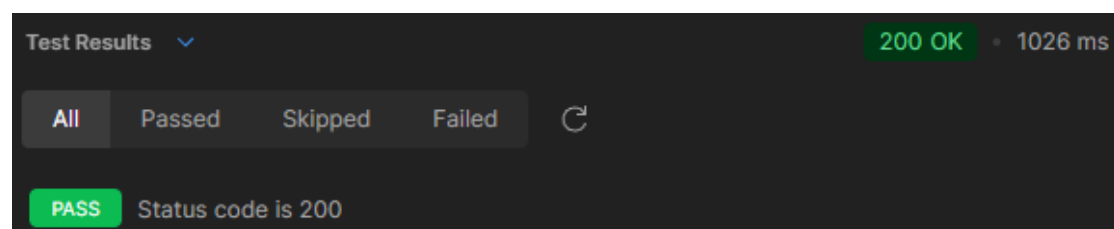


Figura 122 - Código de sucesso da listagem de uma sala específica

Nos testes de integração do Postman, todos os testes passaram com sucesso e os resultados esperados refletiram-se na API e na base de dados juntamente com a tabela das reuniões.

Os testes unitários não foram possíveis de serem desenvolvidos e portanto não se pode tirar conclusões sobre os seus resultados e impacto no desempenho e fiabilidade do projeto e suas funções e componentes.

5 Conclusões

O projeto desenvolvido tinha como objetivo a criação de uma aplicação que recorresse ao ISR juntamente com técnicas de programação assíncrona e de implementação de pesquisa. Os objetivos deste projeto consistiam na criação de tabelas que lista as salas e reuniões existentes que eram obtidas a partir de endpoints das APIs respetivas que comunicavam com uma base de dados online MongoDB, e na criação, edição e eliminação de reuniões. Os objetivos mencionados anteriormente foram concluídos e o grau de concretização dos mesmos juntamente com o que não foi concretizado ou o que pode ser melhorado vão ser analisados em maior detalhe nas secções deste capítulo.

5.1 Objetivos concretizados

Tal como foi mencionado acima, os principais objetivos foram concretizados com sucesso. Para isso vamos efetuar uma análise mais profunda sobre esses objetivos e a implicação no desempenho da aplicação:

- Criação de uma base de dados Mongo DB: a criação de uma base de dados em MongoDB foi feita com sucesso o que indiretamente acabou por permitir que as outras etapas do projeto fossem feitas pois a base de dados é necessária para armazenar os dados da aplicação;
- Criação de APIs para efetuar operações CRUD com as reuniões: as APIs foram criadas com sucesso o que também indiretamente permitiu que a aplicação funcionasse corretamente em termos de casos de uso e objetivos. Este correto funcionamento foi evidenciado nos testes de integração do Postman que passaram todos;
- Implementação do ISR: um dos objetivos principais deste projeto foi implementado o que permitiu ao utilizador ter acesso à informação necessária sem carga excessiva ao servidor, tornando a aplicação mais rápida;
- Implementação de técnicas de pesquisa nas tabelas das salas e reuniões: as técnicas de pesquisa em ambas as tabelas foram implementadas com sucesso o que permite ao utilizador filtrar mais rapidamente as salas ou as reuniões necessárias em ambos os casos;

- Listagem em tabelas das salas e reuniões, recorrendo à paginação no último caso: a listagem foi totalmente implementada o que permite ao utilizador juntamente com a pesquisa mencionada e o ISR mencionados acima encontrar tudo que precisa mais rapidamente e eficazmente. A paginação permite ao utilizador listar apenas algumas instâncias de cada vez diminuindo assim o tempo de recarregamento;
- Implementação do login recorrendo ao *Microsoft 365* que foi aplicado recorrendo ao Azure AD: O login também foi implementado com sucesso o que permite ao utilizador que aceda aos seus dados e às funcionalidades da aplicação mantendo a sua privacidade enquanto utiliza as conta da organização;
- Criação, edição e eliminação de reuniões: Estas operações foram implementadas com sucesso pois após as mesmas as tabelas aparecem atualizadas. Os resultados destas operações também se refletem nas respetivas APIs e na base de dados. A validação do formulário através da função zod permitiu a inserção ou a edição de reuniões com dados coerentes e corretos.

Estes objetivos provavelmente irão permitir que a aplicação possa ser utilizada pelos potenciais utilizadores da organização devido ao seu desempenho e usabilidade

A concretização destes objetivos permite que a aplicação funcione de forma bem-sucedida e que possa vir a desempenhar um papel positivo na organização, embora sejam necessárias algumas alterações e melhorias que melhorem o desempenho da aplicação e que permitam averiguar se a aplicação funciona como é suposto sem quaisquer problemas. Essa questão vai ser analisada na próxima secção.

5.2 Limitações e trabalho futuro

Apesar de os objetivos principais terem sido cumpridos ainda existem algumas limitações que terão de ser corrigidas no futuro:

- Aperfeiçoamento dos testes: embora os testes de integração do Postman tenham sido realizados, os testes unitários não foram realizados como era suposto, o que deve ser corrigido no futuro;

- Validação das variáveis na criação e edição de reuniões: apesar de esses casos de uso funcionarem bem e fazerem o que é esperado, é importante a inclusão de mais requisitos no zod para cobrir novos casos e evitar erros, como por exemplo a verificação de horários ocupados quando se marca uma reunião em uma determinada sala;
- Filtragem de dados com base no utilizador atualmente logado: como a aplicação será utilizada por mais que um utilizador na organização, seria necessário que a aplicação mostrasse apenas os dados do utilizador que estivesse logado para que o mesmo pudesse ter acesso apenas às informações sobre as reuniões que marcou e também pudesse consequentemente criar, editar e eliminar reuniões;

Os pontos mencionados são os pontos que terão de ser alterados ou melhorados para que a aplicação possa ter um melhor desempenho e eficácia e proporcione uma melhor experiência aos respetivos utilizadores e se possam detetar eventuais falhas no caso dos testes e da validação das variáveis para se assegurar que tudo funciona como suposto e não aconteçam erros ou falhas inesperadas.

5.3 Apreciação final

Ao concluir o projeto, é possível afirmar que se tratou de uma experiência enriquecedora e desafiadora. A criação de um projeto que aplica o ISR juntamente com técnicas de programa assíncrona e de implementação de pesquisa permitiu uma melhoria do desempenho e usabilidade na utilização da aplicação.

Durante todo o processo foram adquiridos conhecimentos valiosos sobre as tecnologias utilizadas e sobre projetos de *Next.js* em *frontend*. A solução desenvolvida tem potencial para ter um impacto positivo na organização. Este projeto proporcionou uma oportunidade para aplicar conhecimentos adquiridos ao longo da licenciatura, assim como adquirir novas habilidades e experiências.

Durante o desenvolvimento do projeto, foi possível aprofundar conhecimentos sobre bases de dados, APIs e *frontend* e as conexões entre os mesmos juntamente com a criação de formulários e o tratamento de dados.

O trabalho neste projeto permitiu ganhar conhecimentos sobre o desenvolvimento de projetos e aplicações no contexto da organização, neste caso da Devscope, embora o projeto tenha sido realizado remotamente.

Referências

- [1] “(4) DevScope: sobre nós | LinkedIn.” Accessed: Jul. 03, 2024. [Online]. Available: <https://www.linkedin.com/company/devscope/about/>
- [2] “Analyzing the Impact of Next.JS on Site Performance and SEO,” *International Journal of Computer Applications Technology and Research*, Oct. 2023, doi: 10.7753/IJCATR1210.1004.
- [3] J. Vepsäläinen, A. Hellas, and P. Vuorimaa, “Implications of Edge Computing for Static Site Generation,” 2023, Accessed: Mar. 17, 2024. [Online]. Available: <https://staticsitegenerators.net/>
- [4] “Docs | Next.js.” Accessed: Jun. 14, 2024. [Online]. Available: <https://nextjs.org/docs>
- [5] “Incremental Static Regeneration (ISR).” Accessed: Aug. 27, 2024. [Online]. Available: <https://vercel.com/docs/incremental-static-regeneration>
- [6] “Visual Explanation and Comparison of CSR, SSR, SSG and ISR - DEV Community.” Accessed: Sep. 05, 2024. [Online]. Available: <https://dev.to/pahanperera/visual-explanation-and-comparison-of-csr-ssr-ssg-and-isr-34ea>
- [7] “Rendering: Server-side Rendering (SSR) | Next.js.” Accessed: Jul. 31, 2024. [Online]. Available: <https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering>
- [8] “What is server-side rendering?” Accessed: Jun. 14, 2024. [Online]. Available: <https://www.educative.io/answers/what-is-server-side-rendering>
- [9] “Rendering: Static Site Generation (SSG) | Next.js.” Accessed: Jul. 31, 2024. [Online]. Available: <https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation>
- [10] “Data Fetching: Incremental Static Regeneration (ISR) | Next.js.” Accessed: Jun. 14, 2024. [Online]. Available: <https://nextjs.org/docs/pages/building-your-application/data-fetching/incremental-static-regeneration>
- [11] “(9) Compreendendo as diferenças entre SSR, SSG e ISR em Next.js | LinkedIn.” Accessed: Aug. 27, 2024. [Online]. Available: <https://www.linkedin.com/pulse/compreendendo-diferen%C3%A7as-entre-ssr-ssg-e-isr-em-nextjs-diego-lisb%C3%B4a-/>

- [12] "Learn Next.js | Next.js." Accessed: Aug. 27, 2024. [Online]. Available: <https://nextjs.org/learn/dashboard-app>
- [13] "O que são testes de integração? Tipos, Processo & Implementação," <https://www.zaptest.com/pt-pt>, Accessed: Jul. 03, 2024. [Online]. Available: <https://www.zaptest.com/pt-pt/o-que-sao-testes-de-integracao-mergulho-profundo-nos-tipos-processo-e-implementacao>

