

ISPGAYA

instituto superior politécnico

Licenciatura em Engenharia Informática
Projeto de Engenharia Informática em Contexto Empresarial

Miguel Mendes Rodrigues

ispg2020102211@ispgaya.pt

DESENVOLVIMENTO DE APLICAÇÃO PARA GESTÃO DE CLÁUSULAS FINANCEIRAS

**Relatório de estágio orientado pelo Professor Doutor Mário Jorge Dias
Lousã e apresentada à Escola Superior de Ciência e Tecnologia**

Julho de 2023

Agradecimentos

Primeiramente, quero agradecer à ITSector e à sua equipa de recrutamento pela oportunidade de integrar uma empresa com esta importância no mercado empresarial e por poder desenvolver-me num ambiente familiar e desafiante.

Agradeço ao Professor Doutor Mário Dias Lousã (orientador de estágio pelo ISPGAYA) por todo o acompanhamento e disponibilidade ao longo do estágio, ao Engenheiro Fernando Pires (orientador de estágio pela ITSector) pelo acompanhamento próximo, disponibilidade e, acima de tudo, pela ajuda contínua.

Por último, agradeço à minha família e à minha namorada pelo apoio constante e por me motivarem sempre a ser e fazer melhor todos os dias.

Resumo

No contexto do trabalho a que se refere este estudo, foram analisadas as questões mais importantes referentes à experiência de estágio curricular numa software house (ITSector) como Frontend Developer, como a experiência vivida, abordagem da empresa ao estágio, tecnologias e metodologias utilizadas e todo o trabalho desenvolvido. O presente documento relata as duas grandes fases vivenciadas no estágio: a fase da academia e a fase do projeto.

O objetivo da primeira fase foi proporcionar uma formação em JavaScript e ReactJS, tal como a introdução às metodologias de trabalho Scrum. Durante a academia, cuja duração foi de quatro semanas, foram fornecidos cursos online e exercícios práticos, nos quais os estagiários tiveram a oportunidade de participar em daily meetings para discutir as tarefas atribuídas, bem como analisar e debater as dificuldades à medida que estas iam surgindo.

Após a incorporação numa equipa de trabalho, na fase correspondente ao desenvolvimento da aplicação para gestão de cláusulas financeiras, foi necessário desenvolver e adquirir conhecimentos e competências de forma a cumprir as tarefas solicitadas, tendo em consideração as necessidades do cliente. Neste sentido, foram utilizadas diversas ferramentas, como, por exemplo, ReactJS (e, implicitamente, JavaScript), ReduxJS, Styled Components e ainda outras tecnologias desenvolvidas pelo cliente final.

O estágio curricular permitiu a aquisição e o desenvolvimento de conhecimentos, competências e habilidades fulcrais para a integração e colaboração numa equipa multidisciplinar, nomeadamente com colegas de *Backend*, *Frontend* e testes de software.

Abstract

In the context of the work to which this study refers, the most important issues related to the curricular internship experience in a software house (ITSector) as Frontend Developer were analyzed, such as the experience lived, the company's approach to the internship, technologies and methodologies used and all the work developed. This document reports the two major phases experienced in the internship: the academy phase and the project phase.

The objective of the first phase was to provide training in JavaScript and ReactJS, as well as an introduction to Scrum work methodologies. During the academy, which lasted four weeks, online courses and practical exercises were provided, in which the trainees had the opportunity to participate in daily meetings to discuss the assigned tasks, as well as analyze and discuss difficulties as they arose.

After joining a work team, in the phase corresponding to the development of the application for the management of financial clauses, it was necessary to develop and acquire knowledge and skills to fulfill the requested tasks, considering the client's needs. In this sense, several tools were used, such as ReactJS (and, implicitly, JavaScript), ReduxJS, Styled Components and other technologies developed by the end customer.

The curricular internship allowed the acquisition and development of knowledge, skills, and abilities central to the integration and collaboration in a multidisciplinary team, namely with Backend, Frontend, and software testing colleagues.

Índice

Agradecimentos	ii
Resumo	iii
Abstract	iv
Lista de abreviaturas e siglas	Erro! Indicador não definido.
Introdução	11
1. Tecnologias Utilizadas	12
1.1. ReactJS	12
1.2. ReduxJS	12
1.3. Styled Components	12
1.4. Tecnologias alternativas	13
2. Metodologias de Trabalho	18
2.1. Metodologia Agile	18
2.2. SCRUM <i>Framework</i>	19
2.3. Metodologia alternativa - Waterfall	21
3. Projetos Similares e Associados	22
4. Bases Teóricas	23
5. Descrição do Trabalho	23
5.1. Academia	23
5.1.1. <i>Módulo JavaScript</i>	23
5.1.2. <i>Módulo ReactJS</i>	25
5.2. Projeto – Aplicação para gestão de cláusulas financeiras	67
5.2.1. <i>Apresentação da equipa</i>	67
5.2.2. <i>Metodologia adotada</i>	68
5.2.3. <i>Duração do projeto</i>	68
5.2.4. <i>Configuração inicial do projeto</i>	69
5.2.5. <i>Sprint n.º 1</i>	71
5.2.6. <i>Sprint n.º 2</i>	72
5.2.7. <i>Sprint n.º 3</i>	72

5.2.8.	<i>Sprint n.º 4</i>	73
5.2.9.	<i>Sprint n.º 5</i>	74
5.2.10.	<i>Sprint n.º 6</i>	74
5.2.11.	<i>Sprint n.º 7</i>	75
6.	Cronograma.....	77
7.	Meios Previstos e Meios Necessários	78
8.	Problemas e Decisões	78
9.	Considerações Finais	79
	Glossário	80

Índice de figuras

Figura 1 - Exercícios javascript básicos.....	24
Figura 2 - Exercícios JavaScript mais complexos	25
Figura 3 - Aspeto inicial do projeto fornecido (página Exemplo)	26
Figura 4 - Componente de Cabeçalho desenvolvido	27
Figura 5 - Componente do exercício 1	27
Figura 6 - Cabeçalho com subtítulo	28
Figura 7 - Código do cabeçalho sem subtítulo.....	28
Figura 8 - Cabeçalho sem subtítulo	28
Figura 9 - Componente geral do exercício 2.....	29
Figura 10 - Componente de título	30
Figura 11 - Componente do botão	30
Figura 12 - Estado inicial da página com título renderizado	30
Figura 13 - Estado da página após o clique sobre o botão.....	31
Figura 14 - Componente geral do exercício 2TS	32
Figura 15 - Componente de título em TypeScript	32
Figura 16 - Componente <Button /> em TypeScript	33
Figura 17 - Função que recebe todos os posts	34
Figura 18 - Função que recebe todos os comentários	34
Figura 19 - useEffect usado para armazenar todos os posts	35
Figura 20 - Mapeamento de "posts" para o ecrã	35
Figura 21 - Função que faz mostrar os comentários de um "post"	36
Figura 22 - Variáveis de estado (comentários e "post" aberto)	36
Figura 23 - Mapeamento de comentários de um "post"	37
Figura 24 - Apresentação dos "posts" na página	37
Figura 25 - Apresentação dos comentários de um "post"	38
Figura 26 - Forma de apresentação das checkboxes para filtragem	38
Figura 27 - Armazenamento e gestão de "checkboxs" seleccionadas	39
Figura 28 - Filtragem de "posts"	39
Figura 29 - Alteração no array mapeado.....	39
Figura 30 - Aspeto final da solução do exercício	40
Figura 31 - Função de extração de informação dos utilizadores	41
Figura 32 - Armazenamento da informação na localStorage.....	41

Figura 33 - Tabela populada com dados dos utilizadores.....	42
Figura 34 - Aspeto da tabela populada	43
Figura 35 – Código do modal de criação de utilizador	44
Figura 36 - Modal de criação de utilizador	45
Figura 37 - Função que armazena o novo utilizador à localStorage.....	45
Figura 38 - Função de remoção de utilizador	46
Figura 39 - Alteração de ícon do botão e transformação da célula em	46
Figura 40 - Comportamento da tabela quando o botão de editar é pressionado	47
Figura 41 - Funções responsáveis pela edição de utilizador.....	47
Figura 42 - Aspeto inicial do exercício 4.....	48
Figura 43 - Criação da store.....	48
Figura 44 - Definição das actions	49
Figura 45 - Criação dos reducers	49
Figura 46 - Código do componente <Title />	50
Figura 47 - Código do componente <ToDoList />	50
Figura 48 - Criação de tarefas.....	51
Figura 49 - Código do componente <ToDoItem />	52
Figura 50 - Duas tarefas criadas e uma selecionada	52
Figura 51 - Código do componente <NewToDo />.....	53
Figura 52 - Código do toastReducer	54
Figura 53 - Código do reducer de utilizadores.....	55
Figura 54 - Código das actions	55
Figura 55 - Aspeto inicial	56
Figura 56 - Aspeto do modal de criação de utilizador.....	56
Figura 57 - Função que trata o formulário	57
Figura 58 - Hook customizado para mostrar Toasts	57
Figura 59 - Notificação de erro de idade	58
Figura 60 - Utilizador criado e mostrado no ecrã	58
Figura 61 – Código do componente <UserList />	59
Figura 62 - Código do componente de <UserCard />.....	60
Figura 63 - Código do componente <Toaster />.....	61
Figura 64 - Testes de renderização de componentes	62
Figura 65 - Teste de simulação de preenchimento do formulário de criação de utilizador	62

Figura 66 - Teste de modal fechar	63
Figura 67- Teste de consumo de dados de serviço externo	64
Figura 68 - Testes de erro de API	64
Figura 69 - Testes de renderização	65
Figura 70 - Teste de renderização de textarea's após clique no botão de editar	65
Figura 71 - Teste de remoção de utilizador	66
Figura 72 - Teste de edição de utilizador.....	66
Figura 73 - Cobertura de testes de todos os exercícios	67
Figura 74 - Esboço da estrutura da aplicação	70
Figura 75 - Cronograma do projeto	77

Índice de tabela

Tabela 1 - Comparação entre ReactJS, VueJS e AngularJS	14
Tabela 2 Comparação entre ReduxJS e Zustand.....	16
Tabela 3 - Comparação entre Styled Components e Tailwind	18
Tabela 4 - Comparação entre as metodologias Waterfall e Agile	22
Tabela 5 – Planeamento do projeto por sprints.....	69

Introdução

O presente documento foi desenvolvido no âmbito da unidade curricular “Projeto de Engenharia Informática em Contexto Empresarial” integrada na Licenciatura em Engenharia Informática, lecionada no Instituto Politécnico Superior Gaia (ISPGAYA) e visa relatar o estágio curricular realizado na empresa ITSector.

A ITSector, líder em tecnologia na área da banca, desenvolve soluções inovadoras e personalizadas para as principais instituições financeiras em todo o mundo. A empresa atua na área da banca digital, seguros e gestão de investimentos, oferecendo aos seus clientes soluções tecnológicas de alta qualidade, tal como uma ampla gama de serviços. Neste sentido, o principal objetivo do estágio na referida empresa correspondeu ao desenvolvimento de *interfaces* de uma aplicação web.

O estágio curricular foi dividido em duas principais fases: academia e o desenvolvimento do projeto. A academia consistiu numa formação, na qual foram facultados aos estagiários todas as bases técnicas necessárias para o desenvolvimento do projeto, bem como os métodos e metodologias de trabalho da empresa. A fase do projeto foi onde o autor foi integrado numa equipa de desenvolvimento multidisciplinar que iria desenvolver uma aplicação web para um dos principais clientes da empresa.

Este documento inicia-se com uma análise sobre as tecnologias utilizadas, tanto na academia, como no projeto, e comparam-se com outras alternativas (secção 1). Relativamente às metodologias implementadas, é igualmente apresentada uma análise, tal como ocorre com as tecnologias utilizadas. De seguida, relata-se em pormenor todo o trabalho desenvolvido na academia (secção 5.1), bem como as etapas do desenvolvimento do projeto, os constrangimentos e como os mesmos foram ultrapassados (secção 5.2). O presente documento finaliza com as considerações finais do autor relativamente a todo o estágio curricular.

1. Tecnologias Utilizadas

1.1. ReactJS

O ReactJS, uma biblioteca JavaScript, foi a principal tecnologia utilizada no desenvolvimento da aplicação. Amplamente reconhecido pela sua eficiência e desempenho, o ReactJS é conhecido por facilitar a criação de *interfaces* de utilizador interativas e reativas. Segundo o sítio web JavaTPoint, a sua abordagem baseada em componentes permite a construção modular e reutilizável de elementos de *interface*, proporcionando uma melhor experiência de utilizador. Para além disso, a fonte revela também que o ReactJS é altamente escalável, permitindo que a aplicação seja facilmente expandida no futuro.

1.2. ReduxJS

O ReduxJS foi adotado como uma biblioteca de gestão de estado na aplicação desenvolvida. Esta biblioteca oferece uma solução robusta para a gestão do estado global da aplicação, sendo particularmente útil em projetos de médio a grande porte. O ReduxJS utiliza um padrão unidirecional de fluxo de dados, proporcionando previsibilidade e consistência no controlo do estado da aplicação. Com este é possível armazenar e partilhar informações cruciais entre diferentes componentes, facilitando o desenvolvimento de funcionalidades complexas e evitando, simultaneamente, más práticas como *prop drilling*, que corresponde ao processo de transportar *props* através de vários componentes intermediários para alcançar um componente específico. Com o ReduxJS, é possível aceder ao estado global de qualquer componente sem a necessidade de transportar as *props* explicitamente.

1.3. Styled Components

Para a estilização dos componentes da aplicação, recorreu-se à biblioteca Styled Components. A sua abordagem inovadora permite escrever estilos CSS diretamente nos componentes JavaScript, garantindo, assim, uma maior modularidade e reutilização de estilos. Com esta biblioteca é possível criar estilos de forma mais declarativa e intuitiva, tornando o código mais legível e de fácil manutenção. Além disso, esta oferece recursos

avançados como o suporte a *props* dinâmicas, permitindo a criação de componentes estilizados de forma flexível e adaptável.

1.4. Tecnologias alternativas

1.4.1. VueJS e AngularJS

O VueJS e o AngularJS são duas alternativas ao ReactJS no desenvolvimento de aplicações web. O VueJS é um *framework* progressivo de JavaScript, enquanto o AngularJS é um *framework* completo de desenvolvimento de aplicações web.

O VueJS permite que os desenvolvedores adotem, gradualmente, as suas funcionalidades em projetos existentes, tornando-o uma opção adequada para equipas que desejam uma curva de aprendizagem mais suave. Com a sua sintaxe intuitiva e documentação clara, o VueJS facilita a criação de componentes reutilizáveis e a construção de *interfaces* interativas.

Por outro lado, o AngularJS é um *framework* abrangente que oferece uma solução completa para o desenvolvimento de aplicações web. Este fornece uma estrutura para a criação de aplicações complexas, com recursos que suportam a realização de testes automáticos, a injeção de dependência e roteamento. Como é referido em vários artigos da plataforma “SPOTDRAFT”, o AngularJS segue uma abordagem “*batteries included*”, proporcionando todas as ferramentas necessárias para a construção de aplicações web escaláveis.

Ao comparar estas tecnologias com o ReactJS, é importante considerar os seguintes aspetos:

- **Curva de aprendizagem:** O VueJS possui uma curva de aprendizagem mais suave comparativamente ao AngularJS e o ReactJS, segundo a sua própria documentação. A sua sintaxe permite que os desenvolvedores adotem, gradualmente, as suas funcionalidades. Por outro lado, o AngularJS possui uma curva de aprendizagem mais íngreme devido à sua abordagem completa e à necessidade de aprender conceitos específicos do *framework*. O ReactJS tem igualmente uma curva de aprendizagem moderada, porém a sua ampla adoção e vasta comunidade permitem que seja mais fácil encontrar recursos e suporte.
- **Flexibilidade:** O VueJS e o ReactJS permitem que os desenvolvedores criem componentes reutilizáveis e adotem uma abordagem modular para o

desenvolvimento de *interfaces* de utilizador. O AngularJS, embora seja flexível em diversos aspetos, pode ser mais rígido em algumas áreas devido às suas convenções e estrutura mais prescritiva.

- **Ecosistema e comunidade:** O ReactJS possui um ecossistema maduro e uma comunidade ativa, o que proporciona uma ampla gama de bibliotecas, ferramentas e recursos disponíveis. Tal como o anterior, o VueJS possui um ecossistema crescente, com uma comunidade envolvida e uma variedade de extensões e plugins. O AngularJS, embora ainda seja usado em diversos projetos antigos, a sua popularidade e suporte têm vindo a diminuir quando comparados com o ReactJS e o VueJS.

De acordo com vários artigos na plataforma “Relevant Software”, no contexto de um projeto de grande escala, o ReactJS, habitualmente, é considerado como a melhor opção devido a diversas razões. Este oferece um equilíbrio entre flexibilidade e estrutura, permitindo o desenvolvimento de aplicações complexas e escaláveis. A sua vasta comunidade e o vasto ecossistema garantem uma ampla disponibilidade de recursos e suporte. Para além destes fatores, o ReactJS é amplamente adotado pela indústria, o que pode facilitar a colaboração e a contratação de desenvolvedores com experiência nessa tecnologia.

A Tabela 1 apresenta uma comparação entre o ReactJS, VueJS e AngularJS nos aspetos anteriormente mencionados.

Tabela 1 - Comparação entre ReactJS, VueJS e AngularJS

Fonte: Autor

Aspeto	ReactJS	VueJS	AngularJS
Curva de aprendizagem	Moderada	Suave	Moderada
Flexibilidade	Alta	Alta	Alta
Ecosistema e comunidade	Amplo	Crescente	Decrescente
Popularidade	Alta	Crescente	Decrescente

Esta comparação destaca algumas diferenças chave entre as tecnologias, ajudando na escolha mais adequada para um projeto específico. No caso de um projeto de grande escala, considerando a flexibilidade, a ampla adoção e o suporte da comunidade, o ReactJS destaca-se como a opção mais recomendada.

1.4.2. Zustand

O Zustand é uma alternativa ao ReduxJS para a gestão de estado em aplicações JavaScript. Este é um gestor de estado leve e escalável, projetado para simplificar o controlo do estado global em aplicações React. O Zustand utiliza um paradigma baseado em *hooks*, permitindo que este seja facilmente adotado e integrado em projetos existentes. Uma das principais características mais apelativas do Zustand é a sua simplicidade. Ele oferece, pois apresenta uma API minimalista, um conjunto reduzido de conceitos e funções, diz o *blog* “LogRocket”.

Comparando o Zustand com o ReduxJS, é importante considerar os seguintes aspetos:

- **Complexidade:** O Zustand é projetado com o objetivo de ser simples e direto. Este oferece uma API menos complexa e requer uma menor configuração quando em comparação com o ReduxJS. Esta simplicidade pode ser vantajosa em projetos menores e mais simples, pois reduz a curva de aprendizagem e a quantidade de código necessário para gerir o estado.
- **Escalabilidade:** O ReduxJS é uma biblioteca altamente escalável e amplamente adotada que fornece recursos robustos para gerir o estado global em projetos de médio a grande porte. O Zustand, embora seja escalável, é mais adequado para projetos menores e situações em que a complexidade da gestão de estado é menor.
- **Ecossistema e suporte:** O ReduxJS possui um ecossistema maduro, com uma grande gama de ferramentas, bibliotecas e documentação disponíveis. Além disso, devido à sua popularidade, é mais provável encontrar recursos e suporte relacionados ao ReduxJS. O Zustand, por ser uma biblioteca mais recente, tem um ecossistema menos desenvolvido e pode ter menos recursos e suporte disponíveis.

A Tabela 2 apresenta uma comparação entre o Zustand e o ReduxJS nos aspetos mencionados anteriormente.

Tabela 2 Comparação entre ReduxJS e Zustand

Fonte: Autor

Aspeto	ReduxJS	Zustand
Complexidade	Complexo	Simples
Escalabilidade	Adequado para projetos de média e grande escala	Adequado para projetos de menor escala
Ecossistema e suporte	Ecossistema maduro e ampla comunidade	Menos desenvolvido

No contexto do projeto desenvolvido, o ReduxJS foi o selecionado tendo em conta dois principais motivos. Primeiramente, o ReduxJS já era amplamente utilizado noutros projetos da empresa para o mesmo cliente, garantindo consistência e facilitando a colaboração entre as equipas. Em segundo, o ReduxJS possui um ecossistema mais maduro e uma comunidade maior, o que oferece mais recursos e suporte para o desenvolvimento.

1.4.3. Tailwind

O Tailwind é uma biblioteca de classes utilitárias de CSS que oferece uma abordagem diferente para o desenvolvimento de *interfaces* de utilizador. Em vez de se utilizarem estilos CSS tradicionais, o Tailwind permite a construção de componentes e *layouts* através da aplicação direta de classes HTML pré-definidas.

Com o Tailwind, em vez de escreverem estilos CSS personalizados, os desenvolvedores podem aproveitar as classes utilitárias fornecidas pela biblioteca para estilizar os elementos. Estas classes incluem estilos para o posicionamento, o espaçamento, as cores, a tipografia, entre outros aspetos visuais e funcionais.

Ao comparar o Tailwind com os Styled Components, uma abordagem popular para estilização em aplicações React, é importante considerar os seguintes aspetos:

- **Flexibilidade:** O Tailwind oferece uma ampla variedade de classes utilitárias que podem ser aplicadas diretamente aos elementos HTML, que permite um

alto grau de flexibilidade, pois os estilos podem ser combinados e personalizados de forma granular para atender às especificidades necessárias.

- **Curva de aprendizagem:** O *Tailwind* é uma tecnologia de fácil aprendizagem, uma vez que se baseia no uso direto de classes CSS. Os desenvolvedores podem aproveitar os nomes intuitivos das classes utilitárias para estilizar os elementos sem a necessidade de escrever CSS personalizado. Os Styled Components, por sua vez, requerem o conhecimento de uma sintaxe específica e a compreensão de conceitos como componentes, propriedades e herança de estilos.
- **Manutenção e escalabilidade:** Os Styled Components apresentam uma vantagem na manutenção e escalabilidade devido à sua abordagem de estilos encapsulados. Com esta abordagem, cada componente React possui os seus próprios estilos, definidos no ficheiro correspondente, facilitando a localização e modificação de estilos específicos. No caso do *Tailwind*, como os estilos são aplicados diretamente através de classes utilitárias, a manutenção pode tornar-se mais desafiadora à medida que o número de classes utilizadas aumenta.

Considerando estes aspetos, os Styled Components são frequentemente considerados a melhor opção para o desenvolvimento de *interfaces* de utilizador em aplicações React devido às seguintes razões:

- **Modularidade e reutilização:** Os Styled Components permitem a criação de componentes React personalizados com estilos encapsulados. Isso promove a reutilização e a modularidade, facilitando a manutenção e o desenvolvimento de *interfaces* consistentes.
- **Flexibilidade e personalização:** Com os Styled Components, é possível estilizar os componentes de forma granular, definindo estilos específicos para cada elemento individualmente. Essa abordagem oferece maior flexibilidade e controle sobre os estilos aplicados aos componentes.
- **Manutenção e escalabilidade:** Os Styled Components facilitam a manutenção, pois os estilos estão diretamente associados aos componentes nos respetivos ficheiros. Isso torna mais fácil localizar, modificar e atualizar os estilos quando necessário. (este também está igual ao de cima).

A Tabela 3 apresenta uma comparação entre o Tailwind e os Styled Components nos aspetos mencionados.

Tabela 3 - Comparação entre Styled Components e Tailwind

Fonte: Autor

Aspeto	Styled Components	Tailwind
Flexibilidade	Alta	Alta
Curva de aprendizagem	Baixa	Moderada
Manutenção e Escalabilidade	Desafiadora	Facilitada

Esta comparação destaca as diferenças entre estas duas tecnologias, sendo possível concluir que cada um tem as suas vantagens e adequações para diferentes cenários. No entanto, considerando a modularidade, a flexibilidade, a facilidade de manutenção e, principalmente, a escalabilidade, os Styled Components são, segundo as estatísticas presentes no portal “BuiltWith”, frequentemente considerados a melhor opção para o desenvolvimento de *interfaces* de utilizador em aplicações React.

2. Metodologias de Trabalho

2.1. Metodologia Agile

Segundo um artigo da empresa portuguesa “XPAND IT”, a metodologia Agile é uma abordagem iterativa e incremental para o desenvolvimento de software que valoriza a colaboração, a flexibilidade e a entrega de valor contínuo. Baseia-se em princípios e valores estabelecidos no Manifesto Agile, que enfatiza a interação humana, a adaptação a mudanças e a entrega de software funcional.

Na empresa ITSector, a metodologia Agile é amplamente utilizada no desenvolvimento de projetos, incluindo aquele em que o estagiário se integrou. Conforme mencionado num artigo recente do *website* “indeed.com”, esta metodologia proporciona uma série de benefícios, tais como:

- **Colaboração:** Esta metodologia promove uma colaboração estreita e contínua entre os vários membros da equipa, incluindo desenvolvedores, *designers*, gestores de projeto e *stakeholders*. Através de reuniões frequentes, como as

reuniões diárias de acompanhamento (*daily stand-ups*), a equipa compartilha informações, discute desafios e toma decisões em conjunto.

- **Flexibilidade:** A abordagem permite uma maior flexibilidade relativamente a mudanças e ajustes durante o desenvolvimento do projeto. As prioridades e requisitos podem ser reavaliados e ajustados em cada iteração, permitindo que a equipa responda de forma rápida e eficiente às necessidades emergentes do cliente ou do mercado.
- **Entrega de valor contínua:** Com a metodologia Agile, o principal foco corresponde à entrega de incrementos de software funcional em curtos períodos de tempo. Tal significa que o cliente recebe valor desde as fases iniciais do projeto, em vez de esperar pela entrega final. Esta abordagem permite validar e obter um *feedback* constante, garantindo que o desenvolvimento está alinhado com as expectativas e necessidades do cliente.

2.2. SCRUM *Framework*

No projeto em que o estagiário participou, a metodologia Agile foi aplicada através da utilização do SCRUM, um *framework* ágil muito adotado na gestão de projetos. O SCRUM baseia-se em princípios colaborativos e iterativos, permitindo a entrega contínua de incrementos de software funcionais ao longo do tempo.

O SCRUM é composto por uma série de papéis, eventos e artefactos essenciais para o seu funcionamento efetivo. De acordo com os documentos oficiais da *framework* os principais elementos do SCRUM incluem:

1. Papéis:

- **Scrum Master:** É o responsável por facilitar a implementação do SCRUM, remover obstáculos e garantir que a equipa está a seguir os princípios e práticas do *framework*.
- **Product Owner:** Representa o cliente ou o utilizador final, definindo os requisitos e priorizando o *backlog* do produto. O *Product Owner* tem como responsabilidade a garantia de que o produto está alinhado com as necessidades do cliente.
- **Equipa de Desenvolvimento:** Composta pelos desenvolvedores, esta é responsável por criar o produto, seguindo as diretrizes estabelecidas

pelo *Product Owner*. A equipa de desenvolvimento é autogerida e colaborativa, trabalhando de modo coletivo para a entrega dos segmentos do produto.

2. Eventos:

- ***Sprint***: O *sprint* é um período fixo, geralmente entre uma a quatro semanas, durante o qual a equipa de desenvolvimento trabalha para entregar um segmento funcional do produto. No final de cada *sprint*, ocorre uma revisão e retrospectiva para avaliar o trabalho realizado e planear o próximo *sprint*.
- ***Daily Scrum***: É uma reunião diária de curta duração em que a equipa partilha o progresso, discute desafios e coordena as atividades do dia. O objetivo desta é manter todos os membros da equipa atualizados e alinhados.
- ***Sprint Review***: No final de cada *sprint*, a equipa de desenvolvimento apresenta o incremento do produto ao *Product Owner* e a outras partes interessadas. Esta é uma oportunidade para se obter *feedback* e para se validar se o trabalho realizado está de acordo com as expectativas.
- ***Sprint Retrospective***: Também realizada no final de cada *sprint*, a retrospectiva permite que a equipa reflita sobre o processo de trabalho, identifique melhorias e estabeleça ações para o próximo *sprint*.

3. Artefactos:

- ***Product Backlog***: É uma lista de todos os requisitos, funcionalidades e melhorias desejadas para o produto. O *Product Owner* é responsável por priorizar e atualizar o *backlog* tendo em conta as necessidades do cliente.
- ***Sprint Backlog***: É uma lista das tarefas que a equipa de desenvolvimento selecionou para realizar durante o *sprint*. O *Sprint Backlog* é derivado do *Product Backlog* e é utilizado para monitorizar o progresso e o cumprimento dos objetivos do *sprint*.

No projeto em que o estagiário participou, a utilização do SCRUM proporcionou uma estrutura clara para o planeamento, execução e acompanhamento do trabalho. Os eventos regulares, como o *Daily Scrum*, *Sprint Review* e *Sprint Retrospective*, permitiram uma comunicação eficiente entre os diversos membros da equipa e permitiu ainda uma

abordagem iterativa de desenvolvimento. Através da divisão do trabalho em *sprints* e da priorização do *backlog* do produto, a equipa pôde responder de forma rápida e flexível às mudanças de requisitos e às necessidades emergentes.

2.3. Metodologia alternativa - Waterfall

A metodologia Waterfall é um modelo sequencial de desenvolvimento de software, no qual as fases do projeto são executadas numa ordem linear, seguindo uma abordagem mais orientada a documentos, avança a “Adobe” através de um artigo no seu blog “Adobe Experience Cloud Blog”.

Na metodologia Waterfall, as fases do projeto, como a análise de requisitos, o *design*, a implementação, os testes e a implantação, são executadas numa sequência rígida. Cada fase é concluída antes de se avançar para a seguinte, na qual se dá uma ênfase na documentação detalhada de cada etapa.

Embora a metodologia Waterfall possa ser adequada para projetos com requisitos estáveis e bem definidos, esta pode apresentar algumas limitações relativamente à flexibilidade e capacidade de resposta a mudanças. Ao contrário da abordagem Agile, em que as alterações de requisitos e ajustes podem ser incorporadas ao longo do desenvolvimento, na metodologia Waterfall, mudanças significativas, após o início de uma fase, podem exigir uma reformulação extensa.

A Tabela 4 apresenta uma comparação entre as metodologias Waterfall e Agile apresentadas anteriormente.

Tabela 4 - Comparação entre as metodologias Waterfall e Agile

Fonte: Autor

	Metodologia Waterfall	Metodologia Agile
Abordagem	Sequencial e linear	Iterativa e incremental
Flexibilidade	Menos flexível, mudanças exigem retrabalho extenso	Altamente flexível, mudanças são incorporadas ao longo do desenvolvimento
Planeamento	Planeamento detalhado no início do projeto	Planeamento adaptativo a cada iteração
Documentação	Documentação detalhada em cada fase do projeto	Documentação menos extensa, focada e essencial
Comunicação	Menos ênfase na comunicação contínua com o cliente	Colaboração intensa com o cliente e a equipa de desenvolvimento
Entrega	Entrega final no término do projeto	Entrega contínua de segmentos funcionais
Requisitos	Requisitos estáveis e bem definidos	Requisitos voláteis e passíveis de mudanças
Riscos	Riscos identificados e mitigados no início do projeto	Riscos identificados e gerenciados ao longo do desenvolvimento
Experiência	Mais adequada para projetos com requisitos bem definidos e previsíveis	Mais adequada para projetos com requisitos voláteis e necessidade de rápida adaptação

3. Projetos Similares e Associados

Uma vez que este género de projetos, inclusivamente o desenvolvido no estágio, destinam-se a um uso interno do cliente, não existe conhecimento de outros projetos semelhantes. A aplicação resultante deste projeto comunica com outras aplicações do cliente através de serviços de API's, para a obtenção de dados e condições financeiras relativos aos clientes do mesmo.

4. Bases Teóricas

A maioria das bases teóricas para o desenvolvimento do projeto no âmbito do estágio, foram fornecidas e trabalhadas na academia da ITSector. As outras bases necessárias ao longo do projeto foram obtidas através da documentação das linguagens utilizadas, nomeadamente ReactJS, ReduxJS e Styled Components.

5. Descrição do Trabalho

5.1. Academia

O primeiro dia resumiu-se a uma introdução de como iria funcionar a academia, o que iria ser feito e como se organizaria o trabalho. Neste dia, foi definido que se iria trabalhar segundo as metodologias AGILE, utilizando o *framework* SCRUM para que se criassem rotinas, com vista a uma futura integração no projeto. Diariamente, pelas 16 horas, decorriam as *Daily Scrum Meetings*, nas quais todos os estagiários apresentavam o que estavam a desenvolver, os seus progressos, o que iriam realizar de seguida e as dificuldades que apresentavam. No que se refere à metodologia adotada ao nível da formação, a academia baseou-se na autoaprendizagem dos estagiários, sendo que os formadores se apresentaram disponíveis para auxiliar, quer seja com a explicação de conceitos, quer seja com a resolução de exercícios.

Com o iniciar da academia, os conceitos teóricos exigidos pela mesma foram adquiridos através da plataforma *Pluralsight*, que fornece inúmeros cursos, em formato vídeo, sobre um diverso leque de temas/tecnologias.

5.1.1. Módulo JavaScript

De modo a compreender e a fundamentar os conceitos básicos de JavaScript, foi recomendada a realização de dois cursos da *Pluralsight*. O primeiro assentou em conceitos mais elementares e fundamentais, tais como:

- Tipos de variáveis.
- Scope de variáveis.
- Condições.
- Ciclos.
- O que são *Arrays*.

- O que são Objetos.
- Manipulação de *Arrays* e Objetos.
- Funções.
- Classes.
- Objetos *Date*, *DateTime* e *TimeStamp*.

Terminado o referido curso, que teve a duração de cerca de 6 horas, foi iniciado o momento de colocar os conceitos adquiridos à prova. Para esse efeito, foi lançado o desafio de se criar algo como um “caderno de exercícios” que apresenta-se os diversos exercícios resolvidos, que requeriam o envolvimento de HTML e de CSS. Alguns dos referidos exercícios resolvidos consistiam na:

- ✓ implementação de condições;
- ✓ implementação de ciclos;
- ✓ manipulação de objetos do tipo *Date* e *DateTime*;
- ✓ realização de cálculos a partir de números escritos em *inputs*;
- ✓ realização de cálculos de dias entre datas;
- ✓ manipulação de *Arrays*;
- ✓ implementação de funções;
- ✓ implementação e manipulação de classes.

Na Figura 1 estão apresentadas as resoluções dos diversos exercícios de JavaScript referidos anteriormente.

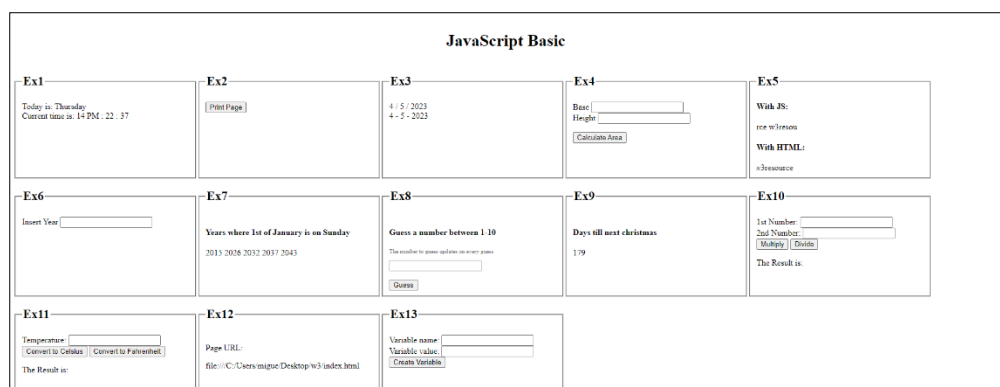


Figura 1 - Exercícios javascript básicos

Os exercícios elaborados são de carácter público e estão presentes no *website w3resource* na secção de JavaScript. Depois de terminada a primeira fase de exercícios,

iniciou-se a realização de um conjunto de exercícios mais complexos, que abordavam conceitos, igualmente mais complexos. Alguns dos conceitos abordados são:

- as promessas;
- o assincronismo;
- o controlo de formulários;
- a manipulação da DOM;
- o acesso a dados, usando HTTP.

Todos os conceitos foram desenvolvidos e resolvidos seguindo o mesmo padrão de consumir a documentação da linguagem, cursos e realização de exercícios, como é apresentado na figura seguinte.



Figura 2 - Exercícios JavaScript mais complexos

O referido módulo teve a duração de cerca de duas semanas, tendo sido concluído com sucesso.

5.1.2. Módulo ReactJS

Para a realização do módulo ReactJS, recorreu-se aos cursos da plataforma referida anteriormente para se compreender e desenvolver os principais conceitos, porém também os mais avançados. Desta vez, os exercícios foram fornecidos pelos formadores (cf. Figura 3), bem como um repositório da plataforma *Azure DevOps*, na qual seriam resolvidos os exercícios. Para além da proposta de solução, em termos funcionais, era pedido que fosse

criado um *design* para cada página, na qual cada página corresponde a um exercício. Para tal, poderia ser usado CSS puro, ou ainda o *framework* de CSS “Bootstrap”.

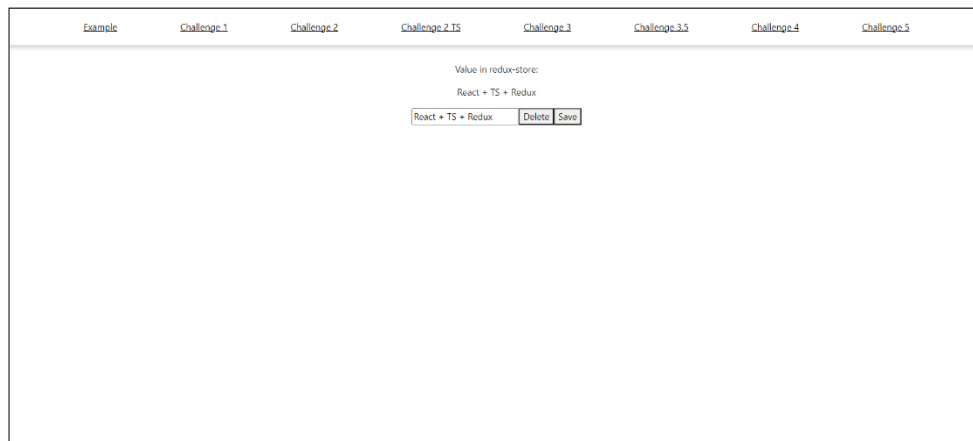


Figura 3 - Aspeto inicial do projeto fornecido (página Exemplo)

O plano prático continha seis exercícios, cuja dificuldade aumentava gradualmente após a resolução de cada exercício. Antes de se iniciar o desenvolvimento dos dois primeiros exercícios, foi recomendada a leitura da documentação oficial de ReactJS para que fossem adquiridos os conceitos básicos e fundamentais da biblioteca.

- **Exercício 1**

O primeiro exercício tinha como principais objetivos compreender a base da criação de um componente funcional, com a utilização de “*props*” entre componentes, e a renderização condicional.

Ainda no âmbito do primeiro exercício, foi solicitada a criação de um componente que renderizasse um cabeçalho, sendo o conteúdo construído por via de “*props*”. Este cabeçalho deveria mostrar um título (<h2>) e, opcionalmente, um subtítulo (<h4>).

Como se pode observar na Figura 4, o componente funcional encontra-se à espera de receber, via “*props*”, um “*title*” e um “*subtitle*”. Este renderiza sempre um título <h2> (o que torna a “*prop*” obrigatória) e valida se existe um subtítulo a ser passado por “*prop*”, antes de renderizar o subtítulo <h4> (renderização condicional).



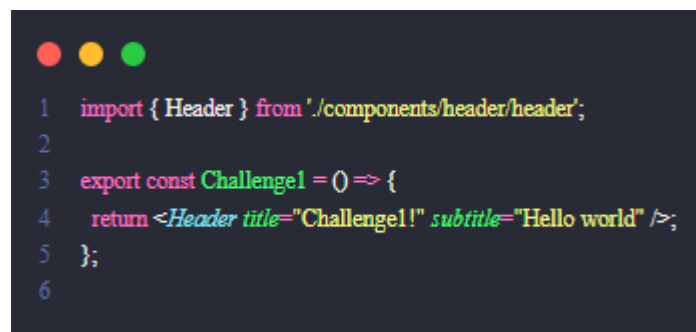
```

1  import './header.css';
2  export const Header = ({ title, subtitle }) => {
3    return (
4      <header>
5        <h2>{title}</h2>
6        {subtitle && <h4>{subtitle}</h4>}
7      </header>
8    );
9  };
10

```

Figura 4 - Componente de Cabeçalho desenvolvido

Após a construção do componente, foi necessário chamá-lo no componente geral do exercício. Uma vez que está a ser passado um subtítulo, o cabeçalho apresenta-o (cf. Figura 5).



```

1  import { Header } from './components/header/header';
2
3  export const Challenge1 = () => {
4    return <Header title="Challenge1!" subtitle="Hello world" />;
5  };
6

```

Figura 5 - Componente do exercício 1

Quando este não existe, o componente, simplesmente, não renderiza nada no seu lugar (cf. Figuras 6, 7 e 8). Se não existisse a validação antes, ou seja, se a “*prop subtitle*” tem um conteúdo definido, a aplicação não iria funcionar, e seria mostrado um erro de campo indefinido.

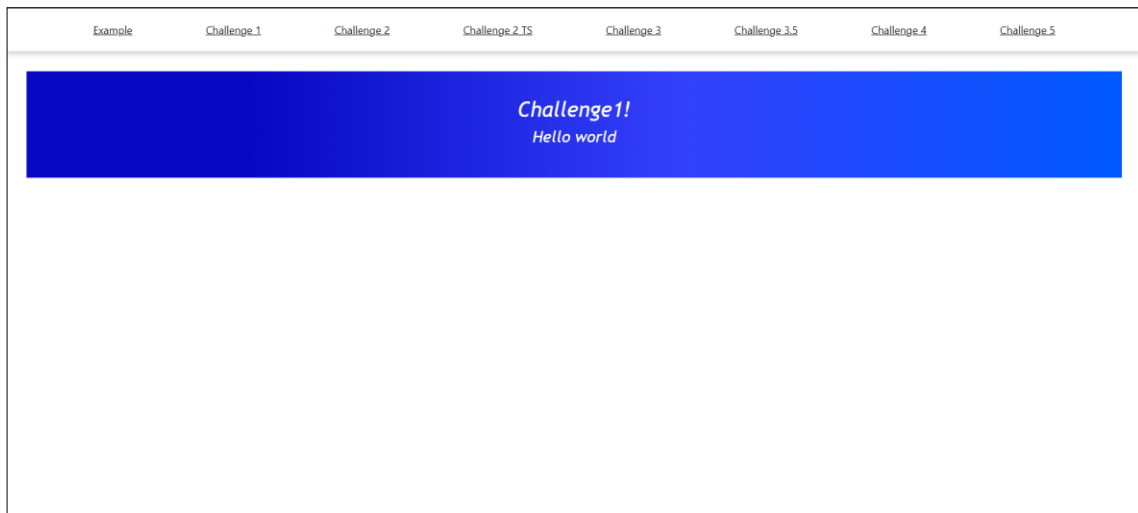


Figura 6 - Cabeçalho com subtítulo

```
1 import { Header } from './components/header/header';
2
3 export const Challenge1 = () => {
4   return <Header title="Challenge1!" />;
5 };
6
```

Figura 7 - Código do cabeçalho sem subtítulo

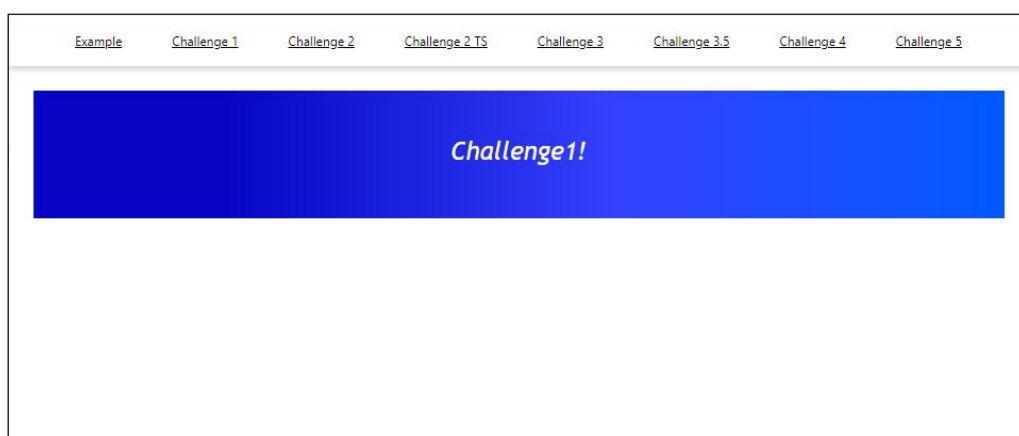


Figura 8 - Cabeçalho sem subtítulo

- **Exercício 2**

No segundo exercício continuou-se a trabalhar a renderização condicional, utilizando funções. Para o efeito, foi solicitado o desenvolvimento de um componente que renderizasse um título (elemento de texto) e um botão. Pretendia-se que o botão fosse capaz de esconder e mostrar o elemento de texto, quando o utilizador carregasse nele.

Para a resolução deste desafio optou-se pela criação de dois componentes: `<Title />` (cf. Figura 10) e `<Button />` (cf. Figura 11). Para controlar se o título devia ser apresentado ou não, fez-se uso de uma das funcionalidades mais populares de ReactJS: o *hook* `useState`. Este `useState` permite controlar o estado de uma variável que neste caso é booleana (verdadeiro ou falso). De modo mais detalhado, renderiza-se o componente `<Title />` com um título a ser enviado por “*prop*”, mas condicionado pela variável “*show*” controlada pelo `useState`. Criou-se a função “*handleClick*” que permitiu alternar o valor da variável “*show*” para o seu oposto, ou seja, se “*show*” é verdadeiro, quando o botão é clicado, este transforma-se em falso. Alternando o estado desta variável, o componente de título é ou não renderizado e o texto dentro do componente do botão é alterado (cf. Figura 9).

```
1  import { useState } from 'react';
2  import Button from './components/button/button';
3  import Title from './components/title/title';
4
5  export const Challenge2 = () => {
6    const [show, setShow] = useState(true);
7    const handleClick = () => {
8      setShow(!show);
9    };
10   return (
11     <div>
12       {show && <Title title="Challenge2!" />}
13       <Button handleClick={handleClick} show={show} />
14     </div>
15   );
16 };
17
```

Figura 9 - Componente geral do exercício 2

```

1  import './title.css';
2  const Title = ({ title }) => {
3    return <h1 className="title">{title}</h1>;
4  };
5
6  export default Title;
7

```

Figura 10 - Componente de título

```

1  import './button.css';
2
3  const Button = ({ handleClick, show }) => {
4    return (
5      <button onClick={() => handleClick} className="submit">
6        {show ? 'Hide Text' : 'Show Text'}
7      </button>
8    );
9  };
10

```

Figura 11 - Componente do botão

Visto que a variável “*show*” tem um estado inicial de verdadeiro, inicialmente o título é renderizado e a página fica com o seguinte aspeto:

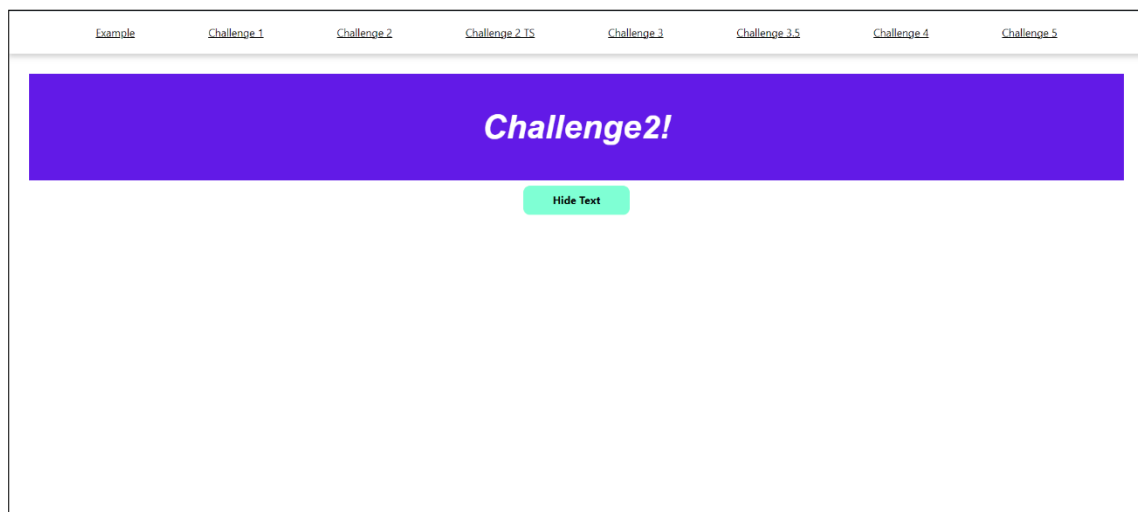


Figura 12 - Estado inicial da página com título renderizado

Após o clique sobre o botão, o título é escondido e o texto do botão muda, tal como é ilustrado na figura seguinte.

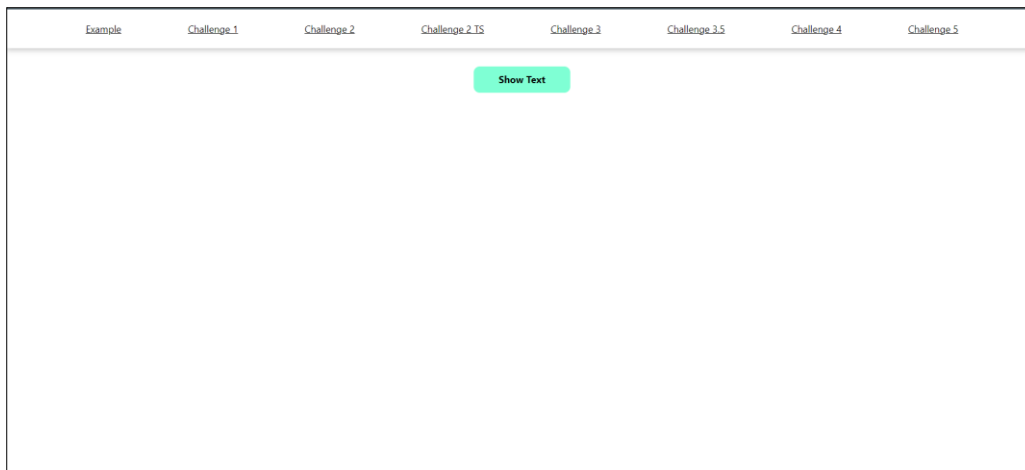


Figura 13 - Estado da página após o clique sobre o botão

- **Exercício 2TS**

No exercício 2TS foram solicitadas as mesmas funcionalidades do exercício anterior, porém com a diferença de se usar o TypeScript, em vez de JavaScript, uma vez que esta é a tecnologia utilizada maioritariamente nos projetos da ITSector.

Antes de se iniciar o desenvolvimento deste exercício foi recomendado o curso “TypeScript Basics”, da *Pluralsight*.

Quando se utiliza TypeScript, a lógica não se altera, apenas se criaram “tipos” para as variáveis e “*props*”, para se obter uma consistência de dados em toda a aplicação. Deste modo, os componentes apresentam um aspeto diferente no código, apesar de fazerem o mesmo (cf. Figura 14).

```

1 import Button from './components/button/button';
2 import Title from './components/title/title';
3
4 export const Challenge2TS: React.FC = () => {
5   const [show, setShow] = useState<boolean>(true);
6   const handleClick = (): void => {
7     setShow(!show);
8   };
9   return (
10    <div>
11      {show && <Title title="Challenge2TS!" />}
12      <Button handle={handleClick} show={show} />
13    </div>
14  );
15 };
16

```

Figura 14 - Componente geral do exercício 2TS

No caso do componente `<Title />` deu-se tipo às “*props*” que conteriam apenas um único elemento de “*title*”, tal como se pode observar na figura seguinte.

```

1 type Props = {
2   title: string;
3 };
4
5 const Title = ({ title }: Props): JSX.Element => {
6   return <h1 className="title">{title}</h1>;
7 };
8
9 export default Title;
10

```

Figura 15 - Componente de título em TypeScript

No que se refere ao componente do botão, procedeu-se ao mesmo processo, no entanto, neste caso, as “*props*” possuíam dois elementos, um deles sendo uma função e o outro um booleano (cf. Figura 16).



```

1  type Props = {
2    handle: React.MouseEventHandler<HTMLButtonElement>;
3    show: boolean;
4  };
5
6  const Button = ({ handle, show }: Props): JSX.Element => {
7    return <button onClick={handle} className="submit">`${show ? 'Hide' : 'Show'} Text`</button>;
8  };
9

```

Figura 16 - Componente <Button /> em TypeScript

Após este exercício, todos os restantes exercícios foram desenvolvidos usando TypeScript, em vez de JavaScript.

Terminada esta fase, assumiu-se que os conceitos básicos de ReactJS já estavam compreendidos e dominados, aumentando assim o nível de complexidade dos exercícios seguintes.

- **Exercício 3**

O terceiro exercício pediu a funcionalidade de visualização de “*posts*” e comentários, e que toda a informação fosse extraída do serviço <https://jsonplaceholder.typicode.com>, partindo da biblioteca *Axios*. Esperava-se ainda que fosse possível filtrar os “*posts*” pelo *ID* (identificador único) do utilizador que tivesse publicado o respetivo “*post*”, e que, ao carregar num “*post*”, fosse possível ver os respetivos comentários. Tendo em consideração esta problemática, foi idealizada uma possível solução, que consistia em mostrar os “*posts*” com um aspeto de cartão, e, quando se clicava sobre o mesmo, os comentários seriam apresentados por baixo do respetivo “*post*”.

Assim, iniciou-se este procedimento através do desenvolvimento das funções que iriam carregar as informações servidas pelos serviços do site <https://jsonplaceholder.typicode.com> (cf. Figuras 17 e 18).

```

1  import axios from 'axios';
2
3  export interface Post {
4    userId: number;
5    id: number;
6    title: string;
7    body: string;
8  }
9
10 export async function fetchPosts(): Promise<Post[]> {
11   const response = await axios.get('https://jsonplaceholder.typicode.com/posts');
12   return response.data;
13 }
14

```

Figura 17 - Função que recebe todos os posts

```

1  import axios from 'axios';
2
3  export interface Comment {
4    postId: number;
5    id: number;
6    name: string;
7    email: string;
8    body: string;
9  }
10
11 export async function fetchComments(postId: number): Promise<Comment[]> {
12   const response = await axios.get('https://jsonplaceholder.typicode.com/posts/${postId}/comments');
13   return response.data;
14 }
15

```

Figura 18 - Função que recebe todos os comentários

Tendo acesso a toda a informação necessária, foi desenvolvido o componente de “*posts*”, que recebe, via “*props*”, os *ID*’s descritos no filtro para que se pudesse filtrar. Executou-se, primeiramente, a função “*fetchData*” através de um *useEffect* para que se pudessem renderizar os “*posts*” (Figura 19). Um *useEffect* corresponde a outro *hook* de React que permite executar algo no primeiro “render” da página, e cada vez que exista uma alteração em variáveis presentes no seu “*array*” de dependências.



```

1  useEffect(() => {
2    const fetchData = async (): Promise<void> => {
3      const data = await fetchPosts();
4      setPosts(data);
5    };
6    fetchData();
7  }, []);

```

Figura 19 - useEffect usado para armazenar todos os posts

Assim, com a informação armazenada, foi possível mapeá-la para se renderizarem todos os “*posts*” para o ecrã (Figura 20).



```

1  return (
2    <ul className="post-list">
3      {posts.map(post => (
4        <li key={post.id} className="post">
5          <div className="post-content">
6            <h2 className="post-title">{post.title}</h2>
7            <p>User {post.userId}</p>
8            <p className="post-body">{post.body}</p>
9          </div>
10         </li>
11       )})
12    </ul>
13  );
14 }

```

Figura 20 - Mapeamento de “posts” para o ecrã

Terminado o mapeamento dos “*posts*”, foi necessário mostrar os respetivos comentários quando se clica sobre o “*post*”. De forma a não sobrecarregar a aplicação com demasiados pedidos ao serviço que devolve os comentários, optou-se por apenas fazer esse pedido de comentários para um “*post*” quando o mesmo é clicado (Figura 21). Para este efeito, usaram-se mais duas variáveis de estado, uma para armazenar comentários e outra

para gerir se o “*post*” estava a mostrar os comentários ou não (Figura 22). Entende-se que os comentários são mostrados quando o “*post*” está aberto.



```
1  const handleClick = async (postId: number): Promise<void> => {
2    if (isPostOpen.state) {
3      setComments([]);
4      setIsPostOpen({ postId: postId, state: false });
5    } else {
6      const data = await fetchComments(postId);
7      setComments(data);
8      setIsPostOpen({ postId: postId, state: true });
9    }
10  };
```

Figura 21 - Função que faz mostrar os comentários de um "post"



```
1  const [comments, setComments] = useState<Comment[]>([]);
2  const [isPostOpen, setIsPostOpen] = useState({ postId: -1, state: false });
```

Figura 22 - Variáveis de estado (comentários e "post" aberto)

Estando os comentários de um determinado “*post*” a armazenar, restava mapeá-los para o ecrã. Tal foi possível usando o mesmo método que se tinha utilizado para os “*posts*” (cf. Figura 23).

```

1  return (
2    <ul className="post-list">
3      {posts.map(post => (
4        <li key={post.id} className="post" onClick={() : Promise<void> => handleClick(post.id)}>
5          <div className="post-content">
6            <h2 className="post-title">{post.title}</h2>
7            <p>User {post.userId}</p>
8            <p className="post-body">{post.body}</p>
9          </div>
10         {comments.map(comment => {
11           if (comment.postId === post.id) {
12             return (
13               <div className="comments" key={comment.id}>
14                 <ul className="comment">
15                   <li className="com-user">
16                     <span className="com-name">{comment.name}</span>
17                     <span className="com-email">{comment.email}</span>
18                   </li>
19                   <li className="com-body">{comment.body}</li>
20                 </ul>
21               </div>
22             );
23           }
24           return null;
25         })}
26       </li>
27     )}
28   </ul>
29 );

```

Figura 23 - Mapeamento de comentários de um "post"

Conforme ilustra na Figura 24, nesta fase os “posts” já se encontravam apresentados.

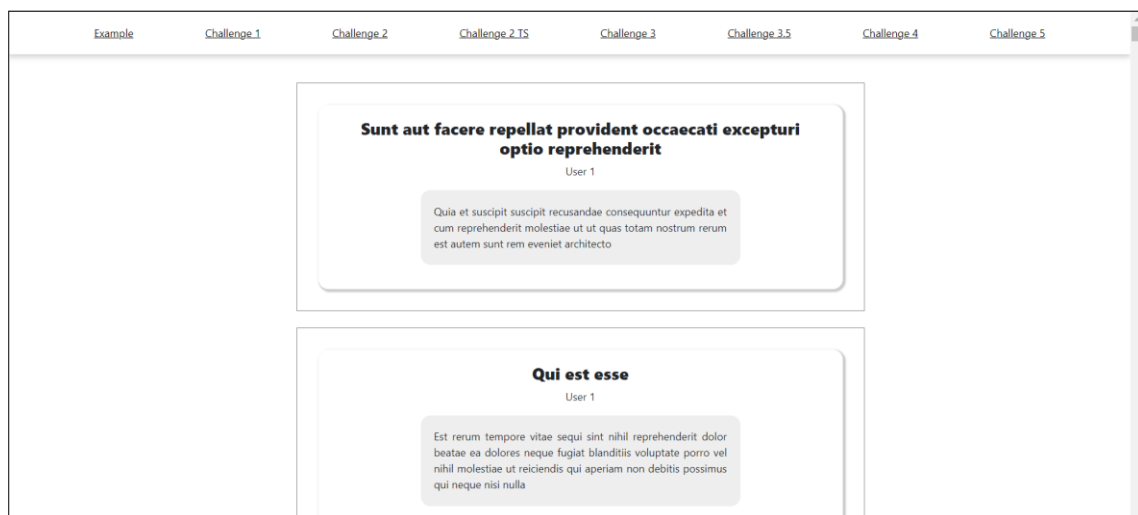


Figura 24 - Apresentação dos "posts" na página

À semelhança da presença dos “posts” e como se observa na Figura 25, era igualmente visível os comentários depois de se clicar sobre o “post”.

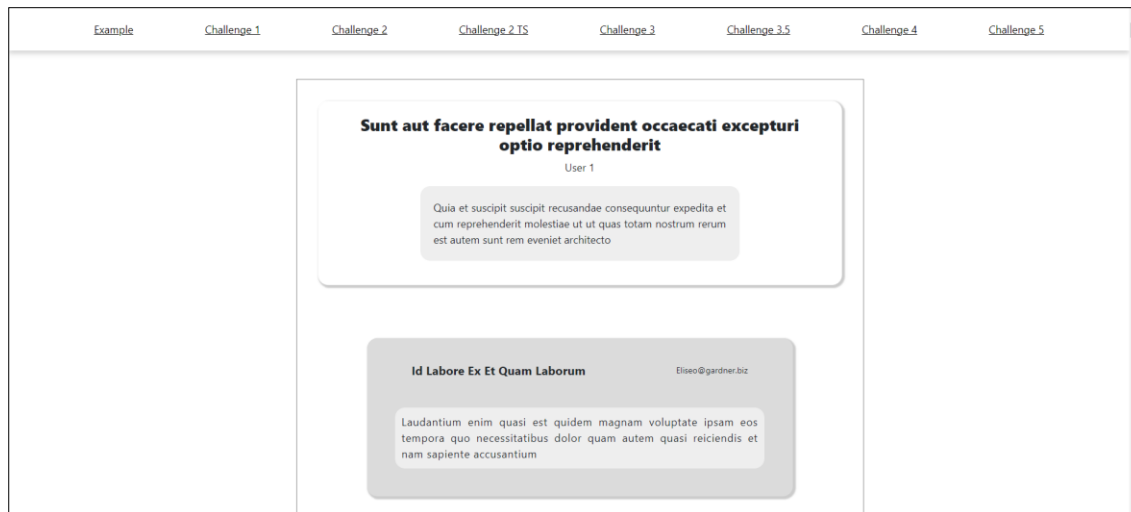


Figura 25 - Apresentação dos comentários de um "post"

Para os filtros, optou-se por um implementar uma “checkbox” para cada *ID*, como apresentado na seguinte figura.



Figura 26 - Forma de apresentação das checkboxes para filtragem

No componente dos filtros optou-se por utilizar uma variável de estado que armazena os *ID*'s dos utilizadores correspondentes às “checkboxs” seleccionadas, que era atualizada a cada mudança (nas “checkboxs”) (cf. Figura 27).

```

1  const [ids, setIds] = useState<number[]>(<[]>);
2
3  const handleChange = (e: ChangeEvent<HTMLInputElement>): void => {
4    const value = parseInt(e.target.value);
5    if (ids.includes(value)) {
6      setIds(ids.filter(id => id !== value));
7    } else {
8      setIds([...ids, value]);
9    }
10 };
11
12 useEffect(() => {
13   onChange(ids);
14 }, [ids, onChange]);

```

Figura 27 - Armazenamento e gestão de "checkboxs" seleccionadas

Os referidos *ID*'s eram recebidos pelo componente `<Posts />`, através de “*props*”, como mencionado anteriormente, e filtrava os “*posts*” de acordo com uma expressão utilizando métodos nativos do JavaScript (*filter*¹ e *includes*²). É de notar que, caso não existissem *ID*'s seleccionados para a filtragem, todos os “*posts*” eram renderizados para a página (cf. Figura 28).

```

1  const filteredPosts = ids.length === 0 ? posts : posts.filter(post => ids.includes(post.userId));

```

Figura 28 - Filtragem de "posts"

Para finalizar o exercício era necessário atualizar o *array* que é mapeado para renderizar os “*posts*”, para que, assim, o filtro tivesse verdadeiramente efeito (cf. Figuras 29 e 30).

```

1  filteredPosts.map(post => (

```

Figura 29 - Alteração no array mapeado

¹ permite filtrar um *array*.

² verifica se existe um determinado valor no *array*.



Figura 30 - Aspetto final da solução do exercício

• Exercício 3.5

Como se pode compreender pela numeração apresentada para este exercício, o mesmo abordava essencialmente os mesmos conceitos que o exercício anterior, porém com apenas uma alteração, que correspondia ao requisito da utilização da “*localStorage*”. A *localStorage* opera como uma memória do *browser* do utilizador da aplicação que pode ser manipulada pela aplicação *web* que o utilizador visita. Para a conclusão deste exercício, era esperado que esta memória fosse utilizada para armazenar informação. Os requisitos do exercício eram os seguintes:

- desenvolver funcionalidades que permitissem o CRUD (Criar, Ler, Atualizar e Remover) de utilizadores;
- a informação relativa a esses mesmos utilizadores deveria ser obtida através do serviço correspondente que o *website* <https://jsonplaceholder.typicode.com> fornece (utilizando a biblioteca *Axios*) e armazenada apenas na *localStorage*.

Para solucionar o exercício foi elaborada uma tabela com as colunas “nome”, “cidade” e “rua” para apresentar as informações sobre os utilizadores e, ainda, duas colunas extras de edição e remoção que iriam apresentar os botões que serviriam como ligação a essas mesmas funcionalidades.

Tal como no exercício anterior, começou-se por extrair a informação dos utilizadores (cf. Figura 31).


```

1  export async function fetchUsers(): Promise<User[]> {
2  try {
3    const response: Axios.Response = await axios.get('https://jsonplaceholder.typicode.com/users');
4
5    if (response.status !== 200) {
6      const error = new Error('Failed to fetch users: ${response.status} ${response.statusText}');
7      throw error;
8    }
9
10   const users: any[] = response.data;
11
12   return users.map((user: User) => ({
13     id: user.id,
14     name: user.name,
15     address: {
16       street: user.address.street,
17       city: user.address.city,
18     },
19   }));
20 } catch (error: any) {
21   throw new Error('Failed to fetch users: ${error.status} ${error.statusText}');
22 }
23 }
24

```

Figura 31 - Função de extração de informação dos utilizadores

Como a solução consistiu numa única tabela, foi criado apenas um componente que correspondia a essa tabela. Neste componente começou-se por guardar a informação na *localStorage* e, de seguida, montar a tabela já com a informação dos utilizadores, tal como se observa na figura seguinte.

```

1  useEffect(() => {
2    fetchUsers().then(data => {
3      localStorage.setItem('users', JSON.stringify(data));
4    });
5  }, []);

```

Figura 32 - Armazenamento da informação na localStorage

Depois da informação ser automaticamente guardada na *localStorage*, foi necessário guardar os dados aqui existentes numa variável, que, posteriormente, foi mapeada para popular a tabela (Figuras 33 e 34).

```

1  const storedUsers = JSON.parse(localStorage.getItem('users') || '[]');
2  return (
3    <main>
4      <table id="users">
5        <thead>
6          <tr>
7            <th>#</th>
8            <th>Name</th>
9            <th>City</th>
10           <th>Street</th>
11           <th>Update</th>
12           <th>Delete</th>
13         </tr>
14       </thead>
15       <tbody>
16         {users.map((user, index, number) => {
17           return (
18             <tr key={user.id}>
19               <th>{index + 1}</th>
20               <td key={`name-${user.id}`}>{user.name}</td>
21               <td key={`city-${user.id}`}>{user.address.city}</td>
22               <td key={`street-${user.id}`}>{user.address.street}</td>
23               <td key={`update-${user.id}`}>
24                 <button onClick={(): void => handleUpdate(user)} className="update" id="btnUpdate">
25                   <FontAwesomeIcon icon={faPencil} />
26                 </button>
27               </td>
28               <td key={`delete-${user.id}`}>
29                 <button
30                   onClick={(): void => handleDelete(user.id)}
31                   className="delete"
32                   id="btnDelete"
33                 >
34                   <FontAwesomeIcon icon={faTrashCan} />
35                 </button>
36               </td>
37             </tr>
38           );
39         })}
40       </tbody>
41     </table>
42   </main>
43 );

```

Figura 33 - Tabela populada com dados dos utilizadores





















Example	Challenge 1	Challenge 2	Challenge 2 TS	Challenge 3	Challenge 3.5	Challenge 4	Challenge 5
#	Name	City	Street	Update	Delete		
1	Leanne Graham	Gwenborough	Kulas Light				
2	Ervin Howell	Wisokyburgh	Victor Plains				
3	Clementine Bauch	McKenziehaven	Douglas Extension				
4	Patricia Lebsack	South Elvis	Hoeger Mall				
5	Chelsey Dietrich	Roscoeview	Skiles Walks				
6	Mrs. Dennis Schulist	South Christy	Norberto Crossing				
7	Kurtis Weisnat	Howemouth	Rex Trail				
8	Nicholas Runolfsdottir V	Allyaview	Ellsworth Summit				
9	Glenna Reichert	Bartholomebury	Dayna Park				
10	Clementina DuBuque	Lebsackbury	Kattie Turnpike				

Figura 34 - Aspeto da tabela populada

No que se refere ao CRUD, neste momento apenas se possuía a funcionalidade de visualização (“Read”) pronta. Para a implementação do criar (“Create”), optou-se por desenvolver um modal, no qual o utilizador da aplicação poderia preencher os campos necessários para a criação de um utilizador.

Para um modal apresentar o comportamento espectável, foi necessário implementar desde o início a lógica que permita que este se apresente visível ou não. Com o objetivo de compor o aspeto visual deste modal foram utilizados os componentes de estilo já existentes no *framework* “*Bootstrap*”, apresentados nas (cf. Figuras 35 e 36).

```

1  const [openModal, setOpenModal] = useState(false);
2
3  const handleShow = (): void => {
4    setOpenModal(true);
5  };
6
7  const handleClose = (): void => {
8    setOpenModal(false);
9  };
10
11  return (
12    <>
13      <Button onClick={handleShow} className="submit">
14        Add User
15      </Button>
16      <Modal show={openModal} onHide={handleClose} id="modal-container">
17        <Modal.Header closeButton>
18          <Modal.Title>Create a new User</Modal.Title>
19        </Modal.Header>
20        <Modal.Body>
21          <Form onSubmit={}>
22            <Form.Group className="mb-3" controlId="formName">
23              <Form.Label>Full Name</Form.Label>
24              <Form.Control type="text" placeholder="Enter full name" />
25            </Form.Group>
26            <Form.Group className="mb-3" controlId="formCity">
27              <Form.Label>City</Form.Label>
28              <Form.Control type="text" placeholder="Enter city" />
29            </Form.Group>
30            <Form.Group className="mb-3" controlId="formStreet">
31              <Form.Label>Street</Form.Label>
32              <Form.Control type="text" placeholder="Enter street" />
33            </Form.Group>
34            <Button variant="primary" type="submit" className="btnSubmit">
35              Submit
36            </Button>
37          </Form>
38        </Modal.Body>
39      </Modal>
40    </>
41  );

```

Figura 35 – Código do modal de criação de utilizador

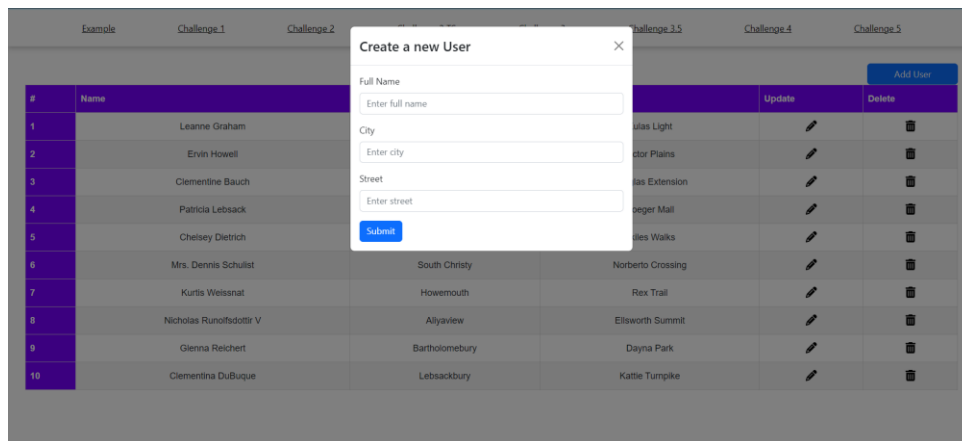


Figura 36 - Modal de criação de utilizador

Após o modal estar construído, foi implementada a lógica de criação do utilizador e de armazenamento dos dados preenchidos na *localStorage* (cf. Figura 37).

```

1  function handleSubmit(event: React.FormEvent<HTMLFormElement>): void {
2    event.preventDefault();
3    const storedUsers = JSON.parse(localStorage.getItem('users') || '[]');
4    const updatedName = document.querySelector<HTMLTextAreaElement>('#formName')?.value;
5    const updatedCity = document.querySelector<HTMLTextAreaElement>('#formCity')?.value;
6    const updatedStreet = document.querySelector<HTMLTextAreaElement>('#formStreet')?.value;
7    const nextId = storedUsers[storedUsers.length - 1].id + 1;
8    localStorage.setItem(
9      'users',
10     JSON.stringify([
11       ...storedUsers,
12       {
13         id: nextId,
14         name: updatedName,
15         address: { city: updatedCity, street: updatedStreet },
16       },
17     ])
18   );
19   handleClose();
20 }

```

Figura 37 - Função que armazena o novo utilizador à localStorage

Para a implementação da opção de remoção, sempre que o botão correspondente é pressionado, é enviado para a função de “*delete*” o *ID* correspondente ao utilizador que será, posteriormente, removido da *localStorage* (cf. Figura 38).

```

1  const handleDelete = (id: number): void => {
2    const newUsers = storedUsers.filter((user: User) => user.id !== id);
3    localStorage.setItem('users', JSON.stringify(newUsers));
4    localStorage.removeItem(`${id}`);
5    setUsers(newUsers);
6  };

```

Figura 38 - Função de remoção de utilizador

Para se proceder à edição dos dados de um utilizador, tornaram-se as células da linha correspondente a tal em elementos que aceitam escrita. Neste caso, o que se pretendia era que, quando se pressionasse sobre o lápis (botão de editar), este se transformasse num visto (✓) e o conteúdo das células dessa linha em “*textarea*’s” pudesse ser editado pelo utilizador da aplicação (não percebi muito bem esta parte a seguir a conteúdo das células). Pressionando no botão com o visto, o que estiver escrito nas “*textarea*’s” será submetido e guardado na *localStorage* novamente (cf. Figura 39 e 40).

```

1  <td key={`street-${user.id}`}>
2    {userToUpdate?.isUpdating && userToUpdate.id === user.id ? (
3      <textarea
4        cols={30}
5        rows={1}
6        style={{ margin: '0.2rem', resize: 'none' }}
7        defaultValue={user.address.street}
8      />
9    ) : (
10     user.address.street
11   )}
12 </td>
13 <td key={`update-${user.id}`}>
14   {userToUpdate?.isUpdating && userToUpdate.id === user.id ? (
15     <button
16       onClick={(): void => handleUpdateSubmit(user)}
17       className="confirm-update"
18     >
19       <FontAwesomeIcon icon={faCheckSquare} />
20     </button>
21   ) : (
22     <button
23       onClick={(): void => handleUpdate(user)}
24       className="update"
25       id="btnUpdate"
26     >
27       <FontAwesomeIcon icon={faPencil} />
28     </button>
29   )}
30 </td>

```

Figura 39 - Alteração de ícon do botão e transformação da célula em

Example	Challenge.1	Challenge.2	Challenge.2.TS	Challenge.3	Challenge.3.5	Challenge.4	Challenge.5
---------	-------------	-------------	----------------	-------------	---------------	-------------	-------------

#	Name	City	Street	Update	Delete
1	Leanne Graham	Gwenborough	Kulas Light		
2	Ervin Howell	Wisokyburgh	Victor Plains		
3	Clementine Bauch	McKenziehaven	Douglas Extension		
4	Chelsey Dietrich	Roscoeview	Skiles Walks		
5	<input type="text" value="Mrs. Dennis Schullist"/>	<input type="text" value="South Christy"/>	<input type="text" value="Norberto Crossing"/>		
6	Kurtis Weissnat	Howemouth	Rex Trail		
7	Nicholas Runolfsdottir V	Aliyaview	Ellsworth Summit		
8	Glenna Reichert	Bartholomebury	Dayna Park		
9	Clementina DuBuque	Lebsackbury	Kattie Turnpike		

Figura 40 - Comportamento da tabela quando o botão de editar é pressionado

Segundo o descrito na Figura 39, o que se observa na prática é que, quando o botão com o lápis é pressionado, é executada a função *handleUpdate()* que recebe o *ID* do utilizador dessa linha da tabela e altera o estado para “está a ser editado”, tornando, assim, as células em “*textarea*’s” e alterando o lápis do botão para um visto. Quando o botão com o visto é pressionado, os valores presentes nas “*textarea*’s” são guardados em variáveis e vão reescrever os dados desse utilizador na *localStorage* (cf. Figura 41).

```

1  const handleUpdateSubmit = (user: User): void => {
2    const updatedName = document.querySelector<HTMLTextAreaElement>(`td:nth-of-type(1) textarea`).value;
3    const updatedCity = document.querySelector<HTMLTextAreaElement>(`td:nth-of-type(2) textarea`).value;
4    const updatedStreet = document.querySelector<HTMLTextAreaElement>(`td:nth-of-type(3) textarea`).value;
5    const index = storedUsers.findIndex((u: User) => u.id === user.id);
6    storedUsers[index] = { id: user.id, name: updatedName, address: { city: updatedCity, street: updatedStreet } };
7    localStorage.setItem('users', JSON.stringify(storedUsers));
8    setUserToUpdate({ id: user.id, isUpdating: false });
9  };
10
11 const handleUpdate = (user: User): void => {
12   setUserToUpdate({ id: user.id, isUpdating: true });
13 };
14

```

Figura 41 - Funções responsáveis pela edição de utilizador

• Exercício 4

No quarto exercício abandonou-se o trabalho com o *localStorage* e iniciou-se a aprendizagem com Redux. O objetivo do exercício era criar uma lista de tarefas, com as

funcionalidades de adicionar tarefas, marcá-las como concluídas e eliminá-las após a conclusão. Todas as tarefas criadas deveriam ser guardadas na “*store*” do Redux e, quando eliminadas, deveriam ser removidas. A conclusão de uma tarefa fazia com que o aspeto da tarefa no ecrã fosse diferente e habilitasse a funcionalidade de remoção.

Na prática, começou-se por desenvolver a *interface*, optando-se por implementar apenas um campo de texto e um botão de submissão para criar tarefas (cf. Figura 42).

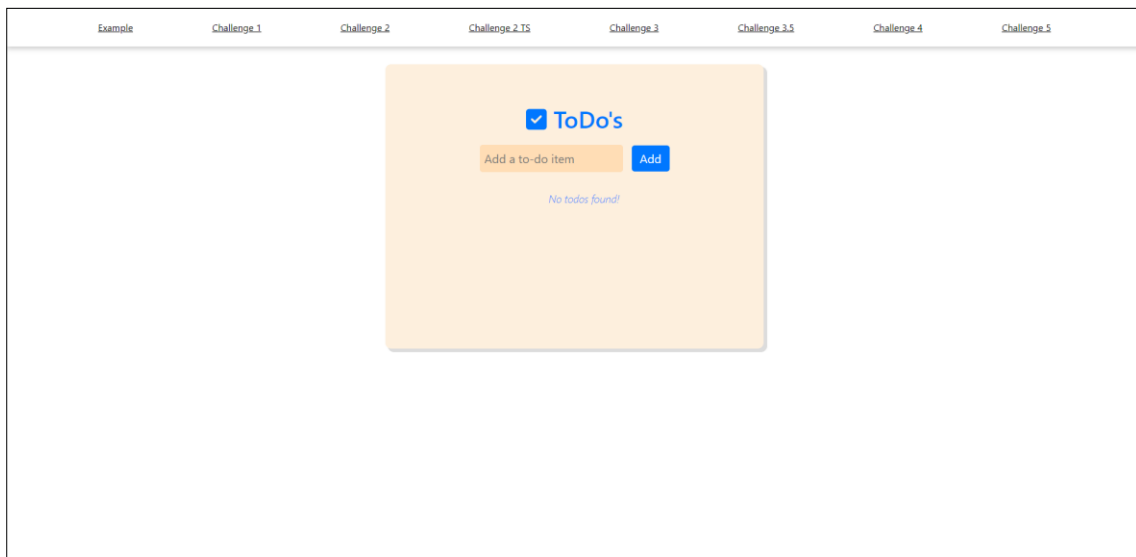
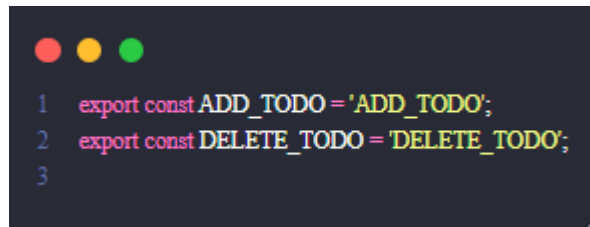


Figura 42 - Aspeto inicial do exercício 4

Após a criação do aspeto inicial, configurou-se a “*store*” de Redux (cf. Figura 43). Para que a “*store*” fosse utilizável, foi necessário definir as funções que manipulam os dados armazenados. Estas funções, denominadas “*reducers*” (cf. Figura 45), são chamadas em qualquer lugar no código da aplicação através das “*actions*” (cf. Figura 44).

```
1  const store = configureStore({  
2    reducer: todosReducer,  
3  });  
4  
5  export default store;  
6
```

Figura 43 - Criação da store



```

1  export const ADD_TODO = 'ADD_TODO';
2  export const DELETE_TODO = 'DELETE_TODO';
3

```

Figura 44 - Definição das actions



```

1  interface ToDoState {
2    todos: string[];
3  }
4
5  const initialState: ToDoState = {
6    todos: [],
7  };
8
9  export const todosSlice = createSlice({
10    name: 'todos',
11    initialState,
12    reducers: {
13      addTodo: (state, action: PayloadAction<string>) => {
14        state.todos.push(action.payload);
15      },
16      deleteTodo: (state, action: PayloadAction<string>) => {
17        const index = state.todos.findIndex(todo => todo === action.payload);
18        if (index !== -1) {
19          state.todos.splice(index, 1);
20        }
21      },
22    },
23  });
24
25  export const { addTodo } = todosSlice.actions;
26  export const { deleteTodo } = todosSlice.actions;
27
28  export default todosSlice.reducer;
29

```

Figura 45 - Criação dos reducers

Deste modo, a *interface* consistia em quatro componentes, sendo um destes o título. Este título era composto por texto estático e um ícone fornecido pela biblioteca *FontAwesome* (cf. Figura 46).

```

1  const Title = () : JSX.Element => {
2    return (
3      <h1 className="title-4">
4        <FontAwesomeIcon icon={faCheckSquare} /> ToDo's
5      </h1>
6    );
7  };
8
9  export default Title;
10

```

Figura 46 - Código do componente <Title />

O segundo componente correspondia ao componente pai da lista, cuja função era ler todas as tarefas presentes na *store* do Redux e, para cada uma, renderizar um componente <ToDoItem /> no ecrã (cf. Figura 47). Caso não existissem tarefas na *store*, era apresentada uma mensagem de que não existiam tarefas criadas. A leitura da *store* era feita através de um *useSelector* (*hook* nativo de React) (cf. Figura 48).

```

1  const ToDoList: React.FC = () => {
2    const todos = useSelector((state: any) => state.todos);
3
4    return (
5      <ul className="todoList">
6        {todos.length > 0 ? (
7          todos.map((todo: string, index: number) => <ToDoItem key={index} item={todo} />)
8        ) : (
9          <p className="notFound">No todos found!</p>
10        )}
11      </ul>
12    );
13  };
14  export default ToDoList;
15

```

Figura 47 - Código do componente <ToDoList />



Figura 48 - Criação de tarefas

O componente que apresenta a tarefa no ecrã, designado por `<ToDoItem />`, renderiza uma linha na lista, gerindo se a linha é pressionada, ou seja, a tarefa foi concluída, e se é eliminada. Quando a linha é pressionada, ocorre uma atualização no estado do próprio componente *“deactive”*, permitindo que o botão de eliminar surja (cf. Figura 49). Esta alteração de estado, faz alterações de estilo à linha da tarefa, para que o utilizador entenda que foi pressionada (cf. Figura 50).

```

1  const ToDoItem: React.FC<ToDoItemProps> = ({ item }) => {
2    const dispatch = useDispatch();
3    const [deactivate, setDeactivate] = useState<boolean>(false);
4
5    const handleToDoClick = (): void => {
6      setDeactivate(!deactivate);
7    };
8
9    const handleDeleteToDo = (): void => {
10     dispatch(deleteToDo(item));
11   };
12   return (
13     <li className={deactivate ? 'todoItem deactivate' : 'todoItem'} onClick={handleToDoClick}>
14       <span style={{ textDecoration: deactivate ? 'line-through' : '' }}>{item}</span>
15       {deactivate && (
16         <button onClick={handleDeleteToDo} className="btnDelete">
17           <FontAwesomeIcon icon={faTrashCan} />
18         </button>
19       )}
20     </li>
21   );
22 };
23
24 export default ToDoItem;
25

```

Figura 49 - Código do componente <ToDoItem />

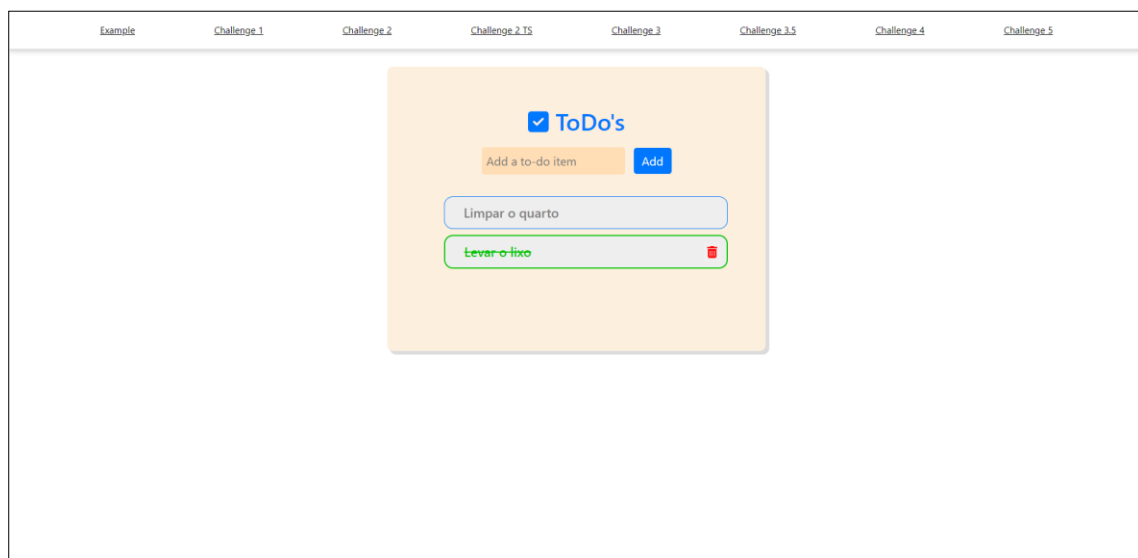


Figura 50 - Duas tarefas criadas e uma selecionada

O último componente consistia no campo de texto e ao seu correspondente botão de submissão para criar tarefas. Este componente tinha a função de adicionar novas tarefas à *store* do Redux (cf. Figura 51).

```

1  const NewToDo: React.FC = () => {
2    const dispatch = useDispatch();
3    const [newToDo, setNewToDo] = React.useState("");
4
5    const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
6      e.preventDefault();
7      if (newToDo.trim() === "") {
8        return;
9      }
10     dispatch(addToDo(newToDo));
11     setNewToDo("");
12   };
13
14   return (
15     <form onSubmit={handleSubmit}>
16       <input
17         type="text"
18         placeholder="Add a to-do item"
19         value={newToDo}
20         className="inptToDo"
21         onChange={e => setNewToDo(e.target.value)}
22       />
23       <button type="submit" className="sbmtToDo">
24         Add
25       </button>
26     </form>
27   );
28 };
29
30 export default NewToDo;
31

```

Figura 51 - Código do componente <NewToDo />

É de notar que todos os componentes foram criados em simultâneo e não pela ordem descrita anteriormente.

• Exercício 5

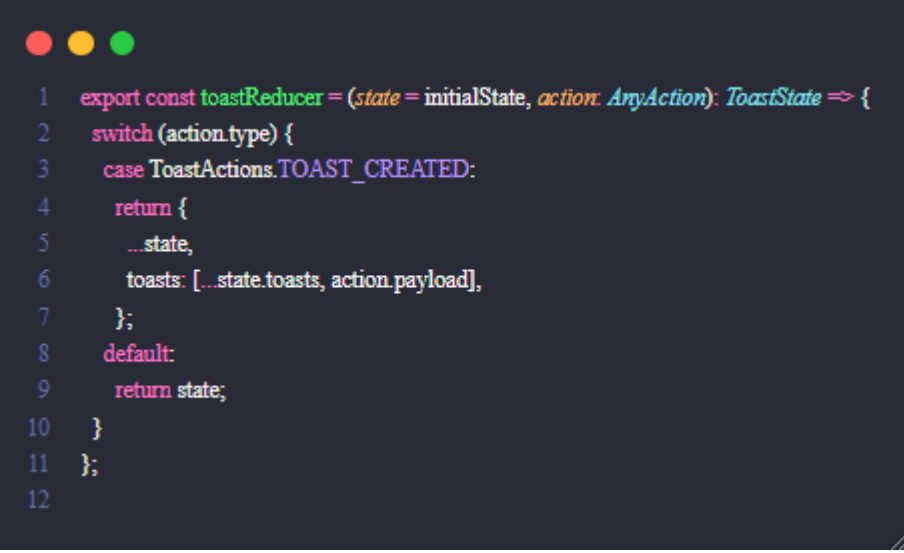
O quinto exercício correspondeu ao último exercício “prático” da academia e o mais extenso, visando englobar todos os conceitos, bibliotecas e ferramentas. O objetivo do

exercício consistia em criar uma aplicação de gestão de utilizadores, que devia respeitar os seguintes requisitos:

- Funcionalidades de CRUD para gestão de utilizadores;
- Um componente que retornasse notificações (erro ou sucesso) para cada ação;
- Fazer uso da *store*, *actions* e *reducers* do Redux para tratamento dos utilizadores e das notificações;
- Um *hook* que permitisse adicionar mais alertas à *store* do Redux;
- A criação de utilizadores deveria seguir uma condição à escolha (neste caso, o utilizador deveria ter idade superior a 18 anos).

Para completar o exercício, utilizou-se a biblioteca *Bootstrap* para renderizar “*Toasts*”, que correspondem a notificações que habitualmente são mostradas num dos cantos do ecrã, e um modal para criar utilizadores.

Desta vez, optou-se por configurar inicialmente a *store*, os *reducers* (tanto para a gestão de utilizadores, como de notificações) e as respetivas *actions*. Para o *reducer* de notificações (ou *Toasts*) apenas existe a funcionalidade de criar (cf. Figura 52).



```
1 export const toastReducer = (state = initialState, action: AnyAction): ToastState => {
2   switch (action.type) {
3     case ToastActions.TOAST_CREATED:
4       return {
5         ...state,
6         toasts: [...state.toasts, action.payload],
7       };
8     default:
9       return state;
10  }
11 };
12
```

Figura 52 - Código do toastReducer

Já no *reducer* dedicado aos utilizadores, existe a funcionalidade adicional de remoção, conforme ilustrado na figura seguinte.

```

1  export const userReducer = (state = initialState, action: AnyAction): UserState => {
2    switch (action.type) {
3      case UserActions.USER_CREATED:
4        return {
5          ...state,
6          users: [...state.users, action.payload],
7        };
8      case UserActions.USER_DELETED:
9        return {
10         ...state,
11         users: state.users.filter(user => user.id !== action.payload.id),
12       };
13      default:
14        return state;
15    }
16  };
17

```

Figura 53 - Código do reducer de utilizadores

Para esta aplicação, as *actions* recebem um valor e executam o respetivo *reducer* passando-lhe esse valor (cf. Figura 54).

```

1  export const createUser = (newUser: User): PayloadAction<string, User> => {
2    return action(UserActions.USER_CREATED, newUser);
3  };
4  export const changeUser = (newValue: string): PayloadAction<string, string> => {
5    return action(UserActions.USER_CHANGED, newValue);
6  };
7  export const deleteUser = (user: User): EmptyAction<string> => {
8    return action(UserActions.USER_DELETED, user);
9  };
10 export const createToast = (toast: TToast): EmptyAction<string> => {
11   return action	ToastActions.TOAST_CREATED, { ...toast });
12 };
13

```

Figura 54 - Código das actions

A *interface* continha um título, um botão de abertura de um modal, com um formulário para adicionar utilizadores, e uma secção que listava os utilizadores guardados em forma

de “cartões”. No caso de não existirem utilizadores guardados na *store*, era apresentada uma mensagem que indicava isso (cf. Figura 55).

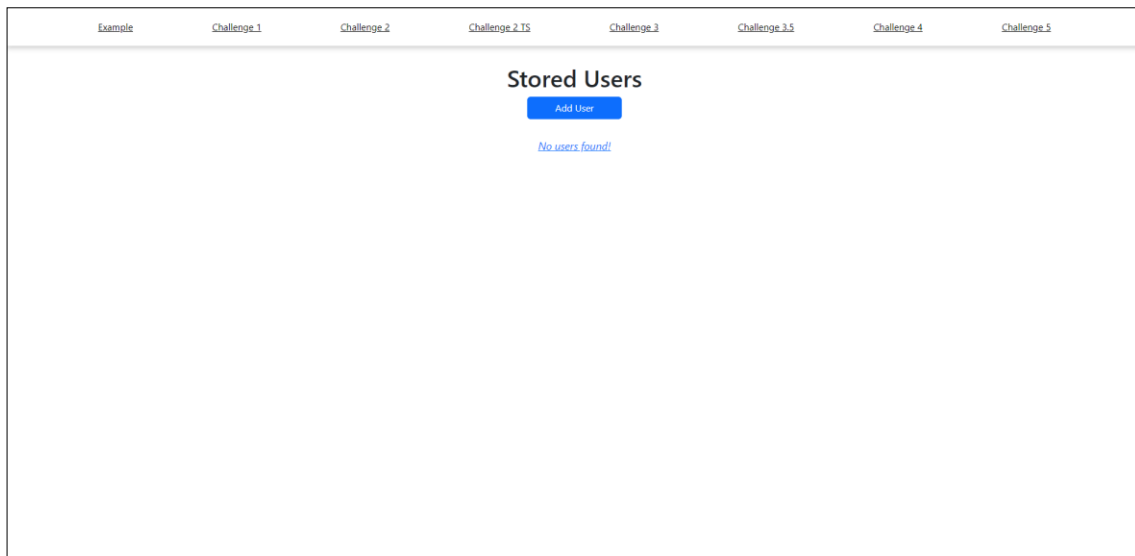


Figura 55 - Aspeto inicial

O modal para a criação de utilizadores consistia num modal com os campos de “Nome Completo”, “Idade” e “Sexo” (cf. Figura 56).

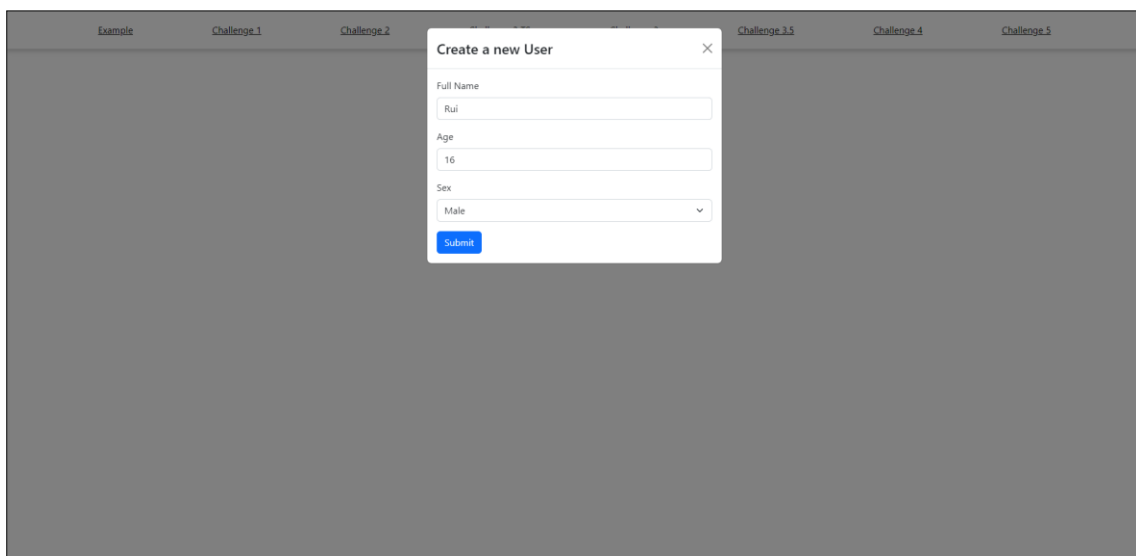


Figura 56 - Aspeto do modal de criação de utilizador

Na função que tratava da submissão do formulário foram realizadas algumas validações, como ilustradas na Figura 57. É igualmente tratado o caso em que o formulário é marcado como completo, estando os campos por preencher. Tal como é, ainda, validado

o formato do campo Nome Completo, de forma a não permitir o envio de números e caracteres especiais utilizando Regex.

```
1  const handleSubmit = (e: React.FormEvent<HTMLFormElement>): void => {
2    e.preventDefault();
3    if (!name.match(/^[a-z ,'-]+$/i)) return showToast({ type: 'warning', text: 'That is not a valid name!' });
4    if (sex === '---Choose an Option---' || sex === 'undef' || !sex)
5      return showToast({ type: 'danger', text: 'Select a valid option!' });
6    if (age < 18) return showToast({ type: 'warning', text: 'User must be over 18 years old!' });
7    const newId = lastId + 1;
8    const newUser: User = { id: newId, name: name, age: +age, sex: sex };
9    dispatch(createUser(newUser));
10   showToast({ type: 'success', text: 'User created successfully!' });
11   setLastId(newId);
12   handleClose();
13  };
```

Figura 57 - Função que trata o formulário

Para apresentar as notificações no ecrã, foi criado um *hook* customizado, que poderia ser executado sempre que era necessário expor uma notificação e atualizava a *store* automaticamente (cf. Figura 58).

```
1  const useToast = () => ((toast: TToast) => void) => {
2    const dispatch = useDispatch();
3    const [toasts, setToasts] = useState<TToast[]>([]);
4
5    useEffect(() => {
6      if (toasts.length > 0) {
7        const latestToast = toasts[toasts.length - 1];
8        dispatch(createToast(latestToast));
9      }
10     }, [toasts, dispatch]);
11
12     const showToast = (toast: TToast): void => {
13       setToasts([...toasts, toast]);
14     };
15
16     return showToast;
17   };
18
19   export default useToast;
20
```

Figura 58 - Hook customizado para mostrar Toasts

No formulário da Figura 56 tentou-se criar um utilizador com a idade de 16 anos. Uma vez que tal não é permitido, a função apresentada na Figura 57 utilizou este *hook* para apresentar uma notificação que alerta para a impossibilidade de adicionar um utilizador menor de idade (cf. Figura 59).

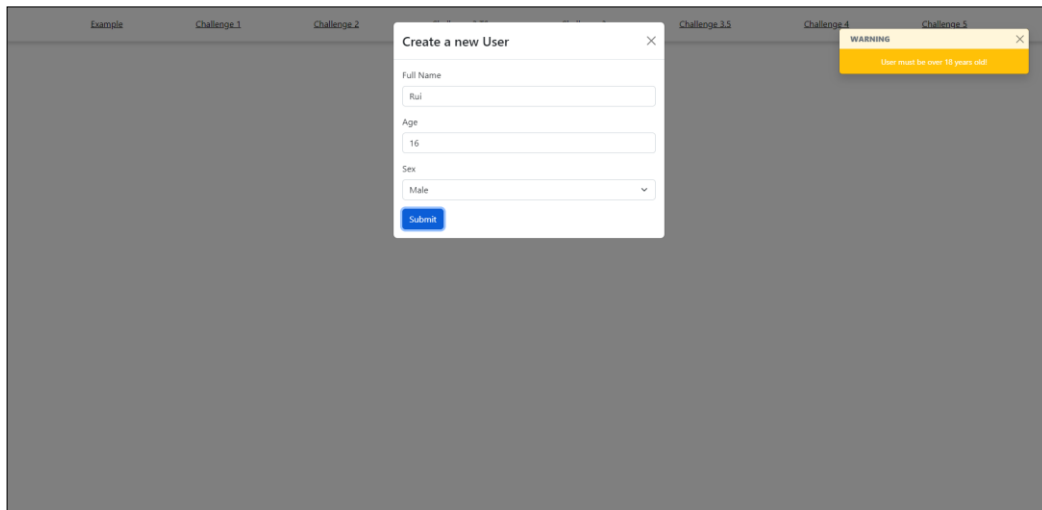


Figura 59 - Notificação de erro de idade

Preenchendo o campo de idade com o número 18, pode-se, assim, criar um utilizador, que é exibido no ecrã com as informações registadas, e é ainda criado um botão que permite eliminar o registo (cf. Figura 60).

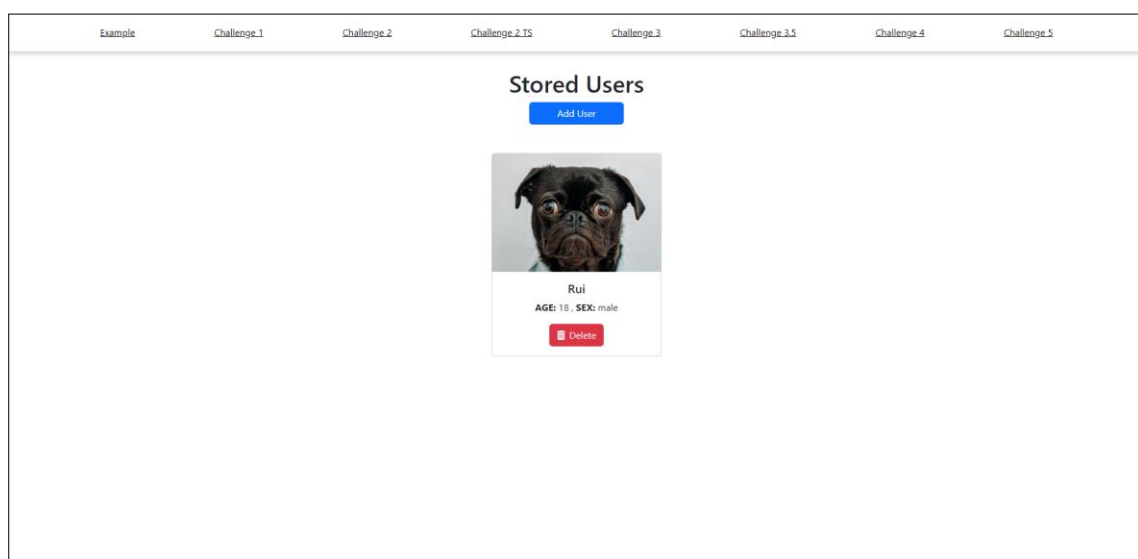


Figura 60 - Utilizador criado e mostrado no ecrã

Para se atingir esta solução, foram criados diversos componentes como `<UserList />`, `<UserCard />` e `<Toaster />` que, trabalhando em conjunto, resultaram no produto final.

- `<UserList />`

Este componente procura na *store* do Redux os utilizadores já criados e mapeia-os para mostrar um cartão para cada um. Caso não existam utilizadores criados, este apresenta uma mensagem que informa isso (cf. Figura 61).

```
1  const UserList = () : JSX.Element => {
2    const [users, setUsers] = useState<User[]>([]);
3    const state = useSelector((state: RootState) => state.user.users);
4
5    useEffect(() => {
6      setUsers(state);
7    }, [state]);
8
9    useEffect(() => {
10     const unsubscribe = store.subscribe(() => {
11       setUsers(state);
12     });
13     return unsubscribe;
14   }, [state]);
15
16   return (
17     <section className="user-list">
18       {users.length > 0 ? (
19         users.map(user => <UserCard key={user.id} user={user} />)
20       ) : (
21         <p className="no-users">No users found!</p>
22       )}
23     </section>
24   );
25 };
26
27 export default UserList;
28
```

Figura 61 – Código do componente `<UserList />`

- `<UserCard />`

Este componente é chamado pelo componente *UserList* que recebe as informações de um único utilizador e utiliza componentes de estilo da biblioteca *Bootstrap* para mostrar as informações corretamente, de modo legível e apelativo (cf. Figura 62).

```
1  const UserCard = ({ user }: Props): JSX.Element => {
2    const showToast = useToast();
3    const dispatch = useDispatch();
4    const handleDelete = (): void => {
5      dispatch(deleteUser(user));
6      return showToast({ type: 'success', text: 'User deleted with success!' });
7    };
8    return (
9      <Card style={{ width: '18rem' }} className="user-card">
10        <Card.Img variant="top" src={pfp} className="pfp" />
11        <Card.Body>
12          <Card.Title className="user-card-title">{user.name}</Card.Title>
13          <Card.Text>
14            <span className="card-sub">Age:</span> {user.age} , <span className="card-sub">Sex:</span>{' '}
15            {user.sex}
16          </Card.Text>
17          <Button variant="danger" className="delete-btn" onClick={handleDelete}>
18            <FontAwesomeIcon icon={faTrashCan} />
19            Delete
20          </Button>
21        </Card.Body>
22      </Card>
23    );
24  };
25
26  export default UserCard;
```

Figura 62 - Código do componente de `<UserCard />`

- `<Toaster />`

O componente `<Toaster />` é responsável por apresentar, no canto superior direito, uma notificação e é sempre executado pelo *hook* customizado (cf. Figura 63).

```

1  const Toaster = (data: TToast): JSX.Element => {
2    const [show, setShow] = useState(true);
3
4    return (
5      <ToastContainer position="top-end" style={{ margin: '40px' }}>
6        <Toast onClose={() => setShow(false)} show={show} delay={3000} autohide bg={data.type}>
7          <Toast.Header>
8            
9            <span className="me-auto" style={{ fontWeight: 900 }}>
10              {data.type.toUpperCase()}
11            </span>
12          </Toast.Header>
13          <Toast.Body className="toast-text">{data.text}</Toast.Body>
14        </Toast>
15      </ToastContainer>
16    );
17  };
18
19  export default Toaster;
20

```

Figura 63 - Código do componente `<Toaster />`

• Exercício 6

O último exercício foi um pouco diferente dos realizados anteriormente, ou seja, não era pedido que se desenvolvessem aplicações ou funcionalidades, mas sim que se fizessem testes automatizados, mais especificamente testes unitários. O objetivo deste exercício passava por testar o exercício 3.5, para ter uma cobertura mínima de testes de 70%. Para o efeito, deviam ser utilizadas as tecnologias Jest e Enzyme. Sendo que se tratava de um novo tema, sentiu-se a necessidade de adquirir novos conhecimentos, recorrendo a dois cursos da plataforma *Pluralsight*, o “Test-driven Development Using React” e o “Testing React Applications with Jest”).

Iniciou-se o exercício por testar o componente `<AddUserModal />` que permitia criar registos. Os primeiros testes corresponderam a testes mais simples e básicos e verificaram apenas se o componente era renderizado para o ecrã, e se o modal aparecia quando o botão de adicionar utilizador era pressionado (cf. Figura 64).

```

1 describe(<AddUserModal />, () => {
2   it('should render the AddUserModal component', () => {
3     const wrapper = shallow(<AddUserModal setUsers={() => {}} />);
4     expect(wrapper.exists()).toBe(true);
5   });
6
7   it('should render the modal when the open-button is clicked', () => {
8     const wrapper = shallow(<AddUserModal setUsers={() => {}} />);
9     const button = wrapper.find('.submit');
10    button.simulate('click');
11    expect(wrapper.find('Modal')).toHaveLength(1);
12  });
13

```

Figura 64 - Testes de renderização de componentes

De seguida, executou-se um teste que simulava o preenchimento do formulário e a consequente submissão (cf. Figura 65).

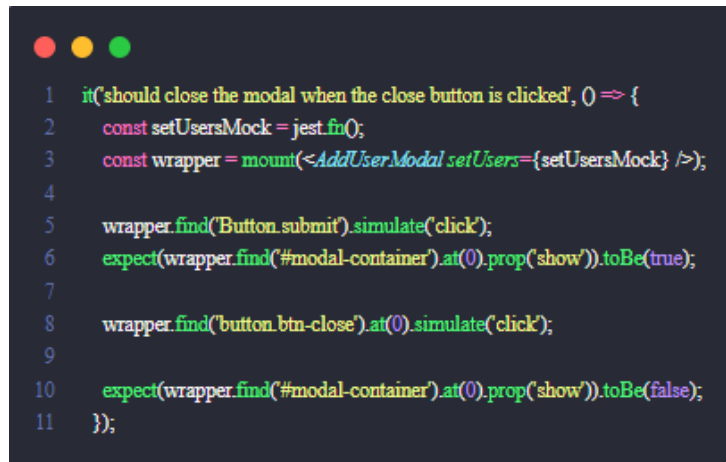
```

1 it('adds a user to local storage when form is submitted', async () => {
2   const setUsers = jest.fn();
3   const wrapper = mount(<AddUserModal setUsers={setUsers} />);
4   const updatedName = 'John Doe';
5   const updatedCity = 'New York';
6   const updatedStreet = 'Broadway';
7   wrapper.find('button.submit').first().simulate('click');
8   wrapper
9     .find('#formName')
10    .at(0)
11    .simulate('change', { target: { value: updatedName } });
12   wrapper
13     .find('#formCity')
14    .at(0)
15    .simulate('change', { target: { value: updatedCity } });
16   wrapper
17     .find('#formStreet')
18    .at(0)
19    .simulate('change', { target: { value: updatedStreet } });
20   wrapper.find('button.btnSubmit').first().simulate('submit');
21   const storedUsers = JSON.parse(localStorage.getItem('users')!);
22   expect(storedUsers).toContainEqual({
23     id: expect.any(Number),
24     name: updatedName,
25     address: { city: updatedCity, street: updatedStreet },
26   });
27 });

```

Figura 65 - Teste de simulação de preenchimento do formulário de criação de utilizador

Por fim, o último teste deste componente verificava se o modal fechava automaticamente depois do formulário ser submetido (cf. Figura 66).

A screenshot of a code editor with a dark background and light-colored text. The code is a Jest test for a modal component. It starts with a line number 1 and an 'it' block. Line 2 defines a mock function 'setUsersMock'. Line 3 mounts the 'AddUserModal' component with the mock. Line 4 is a blank line. Line 5 simulates a click on the submit button. Line 6 expects the 'show' prop to be true. Line 7 is a blank line. Line 8 simulates a click on the close button. Line 9 is a blank line. Line 10 expects the 'show' prop to be false. Line 11 closes the test block. The code is as follows:

```
1 it('should close the modal when the close button is clicked', () => {
2   const setUsersMock = jest.fn();
3   const wrapper = mount(<AddUserModal setUsers={setUsersMock} />);
4
5   wrapper.find('Button.submit').simulate('click');
6   expect(wrapper.find('#modal-container').at(0).prop('show')).toBe(true);
7
8   wrapper.find('button.btn-close').at(0).simulate('click');
9
10  expect(wrapper.find('#modal-container').at(0).prop('show')).toBe(false);
11 });
```

Figura 66 - Teste de modal fechar

Concluídos os testes para este componente, passou-se para a função que recolhia os dados do serviço remoto. Testar consumos de serviços externos é bastante desafiante, pois uma chamada a um serviço demora habitualmente um determinado tempo e, no caso de demorar tempo do que o suposto, tal pode fazer com que o teste falhe mesmo que este tenha sido bem executado e a aplicação funcione corretamente. Um outro aspeto a testar foi o formato de receção dos dados. No teste, foi necessário definir o formato pretendido e validar se, de facto, a aplicação formatava os dados da mesma forma (cf. Figura 67).

```

1  it('should fetch users from the API and return them in the correct format', async () => {
2    const mockUsers: User[] = [
3      {
4        id: 1,
5        name: 'John Doe',
6        address: {
7          street: '123 Main St',
8          city: 'Anytown',
9        },
10     },
11   ];
12   global.fetch = jest.fn().mockResolvedValue({
13     ok: true,
14     json: async () => mockUsers,
15   } as Response);
16   const result = await fetchUsers();
17   expect(fetch).toHaveBeenCalledTimes(1);
18   expect(fetch).toHaveBeenCalledWith('https://jsonplaceholder.typicode.com/users');
19   expect(result).toEqual(
20     mockUsers.map(user => ({
21       id: user.id,
22       name: user.name,
23       address: {
24         street: user.address.street,
25         city: user.address.city,
26       },
27     }))
28   );
29 });

```

Figura 67- Teste de consumo de dados de serviço externo

Apesar de o consumo ser o aspeto mais importante, considerou-se pertinente testar os casos em que o pedido ao serviço falha. Para tal, desenvolveram-se dois testes, um para o caso em que o pedido falha por erro da API e outro para outro tipo de erro (cf. Figura 68).

```

1  test('should throw an error when the API request fails', async () => {
2    global.fetch = jest.fn().mockRejectedValueOnce({ status: 500, statusText: 'Internal Server Error' });
3    try {
4      await fetchUsers();
5    } catch (error) {
6      expect(error).toEqual(new Error('Failed to fetch users: 500 Internal Server Error'));
7    }
8  });
9
10 it('should throw an error when the API response is not ok', async () => {
11   global.fetch = jest.fn().mockResolvedValueOnce({
12     ok: false,
13     status: 404,
14     statusText: 'Not Found',
15   } as Response);
16
17   try {
18     await fetchUsers();
19   } catch (error) {
20     expect(error).toEqual(new Error('Failed to fetch users: 404 Not Found'));
21   }
22 });

```

Figura 68 - Testes de erro de API

O restante componente do exercício 3.5 é o `<Table />`. Como este corresponde ao maior e mais complexo componente, este exigiu igualmente uma maior quantidade de testes. Iniciou-se por definir os testes mais básicos que verificam se os componentes eram renderizados para o ecrã (cf. Figura 69).

```
1 it('should render the Table component', () => {
2   const wrapper = shallow(<Table />);
3   expect(wrapper.exists()).toBe(true);
4 });
5
6 it('should render a HTML table', () => {
7   const wrapper = shallow(<Table />);
8   expect(wrapper.find('table')).toHaveLength(1);
9 });
10
11 it('renders a row in the table for each user in the localStorage', async () => {
12   const wrapper = mount(<Table />);
13   await Promise.resolve();
14   wrapper.update();
15   expect(wrapper.find('tbody tr')).toHaveLength(JSON.parse(localStorage.getItem('users')).length);
16 });
```

Figura 69 - Testes de renderização

Testou-se, ainda, o caso em que o botão de Editar é pressionado e em que é previsível que as células da linha correspondente se tornem *textarea*'s (cf. Figura 70).

```
1 it('should show textareas for each cell of corresponding row when update button is clicked', async () => {
2   const wrapper = mount(<Table />);
3   await Promise.resolve();
4   wrapper.update();
5   wrapper.find('button.update').first().simulate('click');
6   expect(wrapper.find('td textarea')).toHaveLength(3);
7 });
```

Figura 70 - Teste de renderização de *textarea*'s após clique no botão de editar

A funcionalidade de remover um utilizador da *localStorage* após pressionar o botão de remoção foi também testada, tal como apresenta a (cf. Figura 71).

```

1 it('should delete a user from localStorage when delete button is clicked', async () => {
2   const initialAmountOfUsers = JSON.parse(localStorage.getItem('users')).length;
3   const wrapper = mount(<Table />);
4   await Promise.resolve();
5   wrapper.update();
6   wrapper.find('button.delete').first().simulate('click');
7   const updatedAmountOfUsers = JSON.parse(localStorage.getItem('users')).length;
8   expect(updatedAmountOfUsers).toBe(initialAmountOfUsers - 1);
9 });

```

Figura 71 - Teste de remoção de utilizador

Por fim, foi testada a totalidade da funcionalidade de edição de utilizador. Neste teste verifica-se, se após o clique sobre o botão de editar, as *textarea's* aparecem, e, se após o preenchimento das mesmas tal como o clique sobre o botão de confirmação, a *localStorage* é realmente atualizada (cf. Figura 72).

```

1 it('should update a user in localStorage when confirm-update button is clicked', async () => {
2   const updatedUser = {
3     id: 2,
4     name: 'Jane Brown',
5     address: { city: 'Las Vegas', street: '111 Root St' },
6   };
7   const wrapper = mount(<Table />);
8   await Promise.resolve();
9   wrapper.update();
10  wrapper.find('button.update').at(1).simulate('click');
11  const nameTextarea = wrapper.find('td textarea').at(0);
12  const streetTextarea = wrapper.find('td textarea').at(1);
13  const cityTextarea = wrapper.find('td textarea').at(2);
14  nameTextarea.props().value = updatedUser.name;
15  nameTextarea.simulate('change', { target: { value: updatedUser.name } });
16  streetTextarea.props().value = updatedUser.address.street;
17  streetTextarea.simulate('change', { target: { value: updatedUser.address.street } });
18  cityTextarea.props().value = updatedUser.address.city;
19  cityTextarea.simulate('change', { target: { value: updatedUser.address.city } });
20  wrapper.find('button.confirm-update').simulate('click');
21  await Promise.resolve();
22  wrapper.update();
23  const users = JSON.parse(localStorage.getItem('users'));
24  const updatedUserInLocalStorage = users.find((user: User) => user.id === updatedUser.id);
25  expect(updatedUserInLocalStorage).toEqual(updatedUser);
26 });

```

Figura 72 - Teste de edição de utilizador

Com o propósito de consolidar os conhecimentos adquiridos tanto de testes, de modo geral, como também das tecnologias Jest e Enzyme, foram escritos, por iniciativa própria, testes semelhantes para todos os outros exercícios.

Este exercício foi, à semelhança dos anteriores, bem sucedido, obtendo uma cobertura de 80% na maioria dos componentes, conforme é ilustrado na figura seguinte.

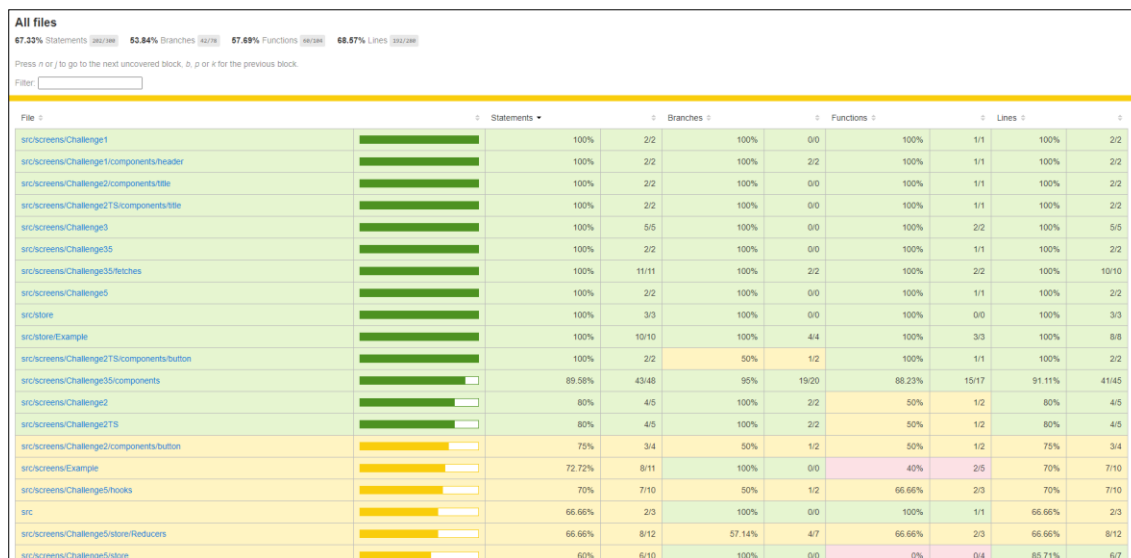


Figura 73 - Cobertura de testes de todos os exercícios

5.2. Projeto – Aplicação para gestão de cláusulas financeiras

Por questões de confidencialidade o nome do cliente e outras informações relativas ao projeto, tais como código, detalhes da aplicação, entre outros, não serão revelados. Para facilitar o entendimento das explicações, atribuir-se-ão nomes fictícios ao longo desta secção, quando tal for necessário. Ao cliente chamaremos simplesmente de “Cliente” e à aplicação de “APP”.

5.2.1. Apresentação da equipa

A equipa dedicada a este projeto era multidisciplinar, isto é, constituída por elementos de diferentes áreas, incluindo *Backend*, *Frontend*, *Quality Assurance* (QA) e Gestão de Projeto. Inicialmente, a equipa de *Backend* era composta por três elementos, porém, ao longo do projeto, sucedeu-se uma redução para apenas dois elementos. A equipa de *Frontend*, por sua vez, manteve-se consistente, com três membros, sendo que o autor deste relatório fez parte desta. A área de QA tinha ao seu dispor um elemento responsável pelos testes e pela validação do software. Além

disso, havia um Gestor de Projeto encarregado de coordenar e supervisionar o desenvolvimento do projeto.

5.2.2. Metodologia adotada

Como referido anteriormente neste documento, a metodologia adotada para o projeto foi a AGILE com auxílio do *framework* SCRUM. No arranque do projeto, foi definido que cada *sprint* iria ter a duração de duas semanas, sendo que no primeiro dia de cada *sprint* iria ser feito o *sprint planning* e, por sua vez, no último dia o *sprint review* e o *sprint retrospective*. Para além destas reuniões com carácter mais formal, diariamente existiam as reuniões mais breves, denominadas *daily scrum meetings* (ou apenas *daily*), nas quais cada elemento da equipa relatava sobre o que tinha realizado até ao momento, o que iria ainda realizar até à próxima *daily*, se tinha encontrado alguma dificuldade que estivesse a atrasar e a comprometer o seu trabalho, e, conseqüentemente, se necessitava de auxílio de algum colega. É de notar que, para além destes momentos proporcionados pelo SCRUM, também existiam canais de comunicação entre todos os elementos da equipa, constantemente disponíveis, para pedidos de ajuda, esclarecimentos, entre outros assuntos.

5.2.3. Duração do projeto

O projeto iniciou-se a 27 de março de 2023, com a previsão de conclusão a 19 de maio de 2023. Contudo, ao longo do desenvolvimento do mesmo, ocorreram inúmeros contratempos que implicaram que o seu término ocorresse a 30 de junho de 2023.

O projeto referido estava dividido em três grandes fases, sendo que este estágio, bem como as estimativas apresentadas, só dizem respeito à primeira fase, que consiste no desenvolvimento da aplicação base.

Uma vez que o cada *sprint* tinha a duração de duas semanas, o planeamento do projeto seguiu a seguinte tabela:

Tabela 5 – Planeamento do projeto por sprints

Fonte: Autor

Nº do sprint	Data de início (dd/mm/aaaa)	Data de fim (dd/mm/aaaa)
1	27/03/2023	07/04/2023
2	10/04/2023	21/04/2023
3	24/04/2023	05/05/2023
4	08/05/2023	19/05/2023
5	22/05/2023	02/06/2023
6	05/06/2023	16/06/2023
7	19/06/2023	30/06/2023

5.2.4. Configuração inicial do projeto

De modo a facilitar o arranque do projeto foi utilizado um projeto anterior, designado Projeto B, desenvolvido pela ITSector para o mesmo cliente como base para o referido projeto. Este projeto foi utilizado como base pelo facto de utilizar as mesmas tecnologias e por seguir uma estrutura semelhante à pretendida.

Todas as questões relativas a ambientes e *pipelines* (DevOps) eram da total responsabilidade do cliente. Como inicialmente, nenhuma destas questões estava resolvida, o *Frontend* e o *Backend* foram desenvolvidas separadamente, sem haver qualquer tipo de ligação entre ambos até ao *sprint* n.º 6.

Para além das tecnologias já mencionadas neste documento (ReactJS, ReduxJS e Styled Components), foram utilizadas duas bibliotecas da autoria do cliente.

A primeira (Biblioteca A) é uma biblioteca de componentes, que tem como objetivo que todas as aplicações do cliente utilizarem os mesmos componentes, de modo a haver consistência no aspeto visual.

A segunda, denominada Biblioteca B, foi utilizada no projeto em questão, com o propósito de estruturar e simplificar a configuração e escalabilidade da aplicação. Esta biblioteca emprega uma abordagem específica, que consiste em utilizar vários ficheiros de configuração para definir as páginas da aplicação, as chamadas dos serviços do *Backend* para componentes específicos e a navegação da APP. O fluxo de funcionamento desta biblioteca segue um padrão pré-definido, em que cada elemento de navegação corresponde

a uma entidade específica. Dentro desse fluxo, cada entidade possui um conjunto de etapas bem definidas. No "Passo 1" do fluxo, ocorre a pesquisa e a listagem dos dados dessa entidade, utilizando filtros para restringir a informação exibida na tabela da página. Nesta etapa, são apresentados um componente de filtro e um componente de tabela. Ao selecionar uma linha da tabela, o utilizador é redirecionado para o "Passo 2".

No "Passo 2", denominado página de detalhes, são exibidas todas as informações relacionadas com a linha selecionada na tabela. Esta biblioteca tem como principal foco a realização das operações básicas de CRUD (Criar, Ler, Atualizar e Remover) para cada entidade. No "Passo 1", é possível criar elementos na entidade em questão, bem como editar e remover elementos existentes. Já no "Passo 2", é possível editar e remover o elemento exibido na página de detalhes.

Através desta explicação formal, fica evidente que a "Biblioteca B" desempenha um papel fundamental na gestão e manipulação das entidades do projeto, proporcionando uma estrutura consistente e simplificada para a implementação do CRUD, facilitando a navegação entre as etapas definidas no fluxo específico, podendo também originar vários problemas caso o *design* da aplicação não siga o padrão da biblioteca.

Como se pode observar na Figura 74, a estrutura da aplicação para a entidade de “Cláusulas Financeiras sob gestão” não segue propriamente o padrão da biblioteca B, ou seja, para a mesma entidade são necessário dois Passos 1, dois CRUD’s e outras operações. Esta estruturação originou alguns percalços no desenvolvimento das páginas desta entidade.

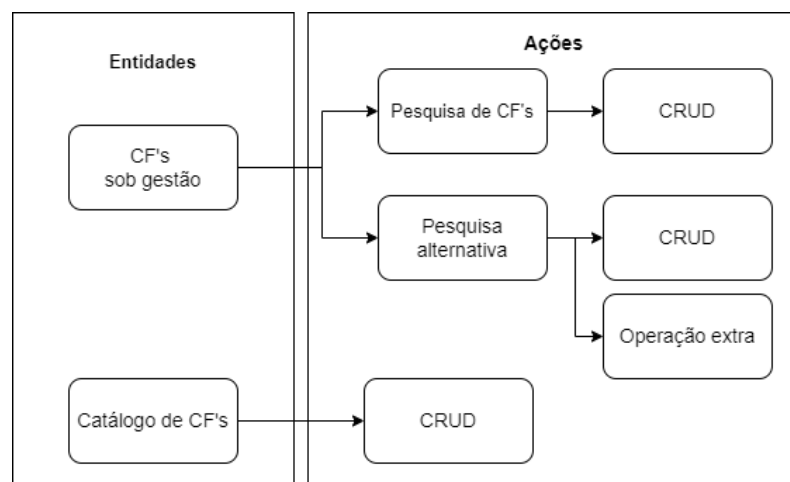


Figura 74 - Esboço da estrutura da aplicação

5.2.5. *Sprint n.º 1*

O primeiro *sprint* teve inicialmente uma semana de atraso, uma vez que dois dos três elementos da equipa de *Frontend* (incluindo o autor) não tinham acesso ao repositório do projeto que pertence ao cliente. Assim, nesta primeira semana foi apenas feito um estudo da estrutura do projeto B e do funcionamento da biblioteca B, através do código disponibilizado pelo elemento que possuía acesso ao projeto. Este foi o primeiro contacto com as tecnologias do cliente o que tornou o processo de entendimento e aprendizagem algo desafiante. Contudo, começou-se rapidamente a alterar localmente o código para acelerar este processo.

Na segunda semana, já com acesso ao repositório, iniciou-se o desenvolvimento. O autor começou por alterar os filtros de ambas as entidades no ficheiro de configuração da biblioteca B para que correspondessem ao *design* esperado. Esta tarefa foi prontamente concluída, tendo-se avançando no mesmo dia para a tarefa seguinte.

A segunda tarefa a ser desenvolvida neste primeiro *sprint* foi mais complexa e trabalhosa do que a anterior, consistindo na criação de Cláusulas Financeiras (CF's) no catálogo. Como esta seria a primeira tarefa complexa e, por haver ainda pouco entendimento sobre o funcionamento das tecnologias e da estrutura do projeto, optou-se por subdividir esta tarefa nas seguintes partes:

- Estruturação base da página;
- Configuração do formulário para diferentes tipos de CF's;
- Criação de uma *feature* extra no formulário;
- Configuração de campos obrigatórios para os diferentes tipos de CF's;
- Submissão dos dados do formulário para o *Backend*.

A estruturação inicial da página foi algo relativamente fácil de realizar, porém interessante, pois permitiu compreender de uma melhor forma como funcionava a biblioteca de componentes do cliente.

Configurar o formulário tendo em conta os diferentes tipos de CF's e a criação da *feature* extra foram tarefas executadas rapidamente, pois envolviam apenas conhecimentos de ReactJS, tais como renderização condicional, condições, entre outros.

As últimas duas tarefas foram mais desafiantes, pois necessitavam de um maior conhecimento do projeto B. No entanto, foram concluídas antes do término previsto para o *sprint*. A submissão de dados, apesar de ter sido feita, não foi corretamente considerada como terminada, porque não era possível testar a recessão dos dados por parte do *Backend*, uma vez que os ambientes responsáveis por permitir esta conexão não estavam configurados.

5.2.6. *Sprint n.º 2*

Para o segundo *sprint* estava previsto reconfigurar todos os filtros da aplicação, uma vez que os requisitos para o seu comportamento foram alterados. Esta tarefa não se revelou complexa, apesar de ter sido necessário mais algum estudo sobre o projeto, para compreender como e onde os componentes dos filtros estavam definidos. Tendo em conta que esta tarefa terminou antes do previsto, foi prestado auxílio a outros elementos da equipa com as páginas em que estavam a trabalhar.

5.2.7. *Sprint n.º 3*

No *sprint* anterior deu-se por terminada a entidade de catálogo e passou-se para a entidade sob gestão. Nesta fase foram revelados as maiores dificuldades, devido ao facto desta entidade não seguir o padrão que a biblioteca B requer. O autor começou por trabalhar em conjunto com outro elemento da equipa de *Frontend* para configurar, de forma dinâmica, a funcionalidade de modo a se obter dois filtros e duas tabelas na mesma página. Foi difícil alcançar esta funcionalidade, uma vez que a aplicação não estava preparada para esta funcionalidade e a tarefa teve a duração de metade do *sprint* (uma semana).

Na segunda metade do *sprint*, a subequipa formada na primeira metade, foi desfeita e ambos os elementos trabalharam nas páginas de detalhes (Passo 2). Coube ao estagiário trabalhar na página de detalhes da pesquisa alternativa e ao outro elemento trabalhar na página de detalhe de CF's. Como ficou definido o detalhe de CF's ser a página que segue o padrão da biblioteca B, a página de detalhe da operação alternativa foi criada de raiz, o que exigiu algum trabalho extra e mais complexo do que previsto, para poder conectá-la com o ficheiro de configuração.

Atendendo que as tarefas da *sprint* n.º 3 terminaram um dia e meio antes do previsto, aproveitou-se esse tempo para realizar testes manuais à aplicação para identificar eventuais problemas que não tinham sido detetados e, inclusivamente, reformular o código que não se encontrava otimizado.

5.2.8. *Sprint* n.º 4

No final do *sprint* n.º 3 encontraram-se *bugs* relativos a utilizar a mesma página com os mesmos componentes em diferentes pontos da aplicação, sendo que esses componentes não estavam a ser atualizados com a nova informação quando os encontrávamos num outro sítio da aplicação. A resolução destes mesmos problemas foi trabalhosa sendo o mais desafiante perceber o que estava a causar o problema. Depois de muitos testes, tentativas de replicar o problema e muita pesquisa na documentação do ReactJS, conseguiu-se compreender que as *props* do componente principal da página não eram atualizadas como seria suposto. Para resolver este problema, tomou-se a decisão de utilizar a *localStorage* do *browser* para armazenar a informação que não chegava ao componente corretamente. A *localStorage*, por vezes, não é a solução ideal para a grande maioria dos projetos, pois pode ser manipulável pelo utilizador, contudo, como neste caso, o produto final seria para utilização interna do cliente, não foram levantados problemas para a implementação desta solução, uma vez que a informação armazenada na *localStorage* não era sensível, sendo apenas relativa à navegação na aplicação.

Após a resolução destes *bugs*, iniciou-se o desenvolvimento da operação extra. Novamente, a biblioteca B não estava preparada para ter esta funcionalidade, sendo necessário desenvolver esta página de raiz. Apesar de ser mais desafiador, desenvolver páginas de raiz torna-se mais fácil devido ao facto de não envolver alterações no ficheiro de configuração da aplicação e tudo pode ser construído manualmente, seguindo o design adequado.

A resolução dos *bugs* preencheram a maior parte do tempo do *sprint* e o desenvolvimento da operação extra preencheu o tempo que restou do mesmo.

5.2.9. *Sprint n.º 5*

No primeiro dia do quinto *sprint* foram disponibilizados os ambientes de desenvolvimento pelo cliente e, para se proceder à integração do *Backend* com o *Frontend*, foi necessário migrar o projeto do repositório atual para o repositório do ambiente de *Frontend*. Esta migração acarretou várias problemáticas, tais como o facto de o ambiente ter deixado de funcionar durante um dia e o facto de existirem várias partes da aplicação que deixaram de funcionar, sendo assim necessário refazer o código e resolver os novos *bugs*.

O passo seguinte consistiu em estudar mais aprofundadamente o projeto e a biblioteca B, para assim possuir um melhor entendimento intrínseco de como são feitos os pedidos à API de *Backend* e como esses pedidos são ligados aos componentes que mostram e enviam a informação. Neste sentido, pretendia-se preparar o próximo *sprint*, totalmente dedicado à integração do *Backend* com o *Frontend*.

5.2.10. *Sprint n.º 6*

Já com o estudo concluído, começou-se a trabalhar na integração do *Backend* com o *Frontend*. Uma vez que a entidade catálogo foi a primeira a ser concluída e visto que era a mais estável e a que se antevia a provocar menos desafios, foi a primeira a ser integrada. O autor começou por integrar o filtro do catálogo e a tabela correspondente. Desde o primeiro momento notaram-se bastantes discrepâncias entre o que o *Backend* considerava necessário para o *Frontend* e o que o *Frontend* realmente necessitava. Durante a fase de integração houve uma colaboração bastante ativa entre a equipa de *Backend* e a de *Frontend* para alinhar a informação que era necessária, o formato em que era esperado a informação chegar, entre outros aspetos.

Estando os filtros do catálogo integrados, aproveitou-se para fazer o mesmo para os filtros e tabelas da entidade sob gestão, uma vez que o trabalho seria semelhante.

Após a conclusão da tarefa, passou-se à integração da página de criação de CF no catálogo, uma vez que já tinha sido desenvolvida pelo autor no primeiro *sprint*. Esta tarefa foi complexa, pois existiam na página várias *dropdowns* cujas opções seriam populadas através de serviços de *Backend*. Foi detetado um problema com os campos obrigatórios, após a integração dos serviços com as *dropdowns*, tendo a submissão da informação para

a API originado, igualmente, vários desafios. Popular as opções das *dropdowns* foi trabalhoso e demorado. A solução apresentava um baixo nível de complexidade, mas exigia a escrita de bastante código e até de alguma repetição do mesmo. Esta tarefa foi concluída em pouco menos de um dia. A problemática com os campos obrigatórios necessitou de mais tempo para ser resolvido, porque não era possível encontrar um padrão nas tentativas de replicar o problema, para que se pudesse perceber o que estava errado. O problema estava no componente em que o utilizador escreve a informação, fornecido pela biblioteca A do cliente, que não conseguia assimilar a informação no caso de o utilizador escrever bastante rápido e com poucos caracteres, e a seguir seleccionar rapidamente outro componente. Este problema foi reportado à equipa do cliente que faz a manutenção desta biblioteca, não tendo sido corrigido até ao momento da redação deste documento. Quanto à submissão dos dados do formulário para a API, a maioria dos problemas encontrados foram, mais uma vez, discordâncias entre o que o *Backend* estava à espera de receber e o que o *Frontend* estava efetivamente a enviar. Após várias conversas dentre as duas equipas, as informações e os formatos dos dados foram alinhados o que resolveu todos os problemas.

Com algum tempo até ao término do *sprint* e já com todas as tarefas concluídas, o autor prestou, novamente, auxílio aos restantes elementos da equipa de *Frontend* para ser possível integrar completamente a entidade catálogo.

5.2.11. *Sprint n.º 7*

O sétimo *sprint* foi o último em que o autor participou e, infelizmente, não pôde contribuir até ao final desta fase do projeto. Atendendo a que a equipa de infraestruturas do cliente ao fazer uma alteração aos ambientes de desenvolvimento, acidentalmente apagou a base de dados que servia a aplicação, impossibilitou que se prosseguisse com as tarefas previstas. Nesse sentido, o estagiário tomou a iniciativa de criar e configurar um componente modal que permitiria alterar o estado de CF's relacionado com outra tarefa, com o objetivo de acelerar o processo de desenvolvimento do trabalho, deixando-o pronto a ser utilizado em cerca de meio dia de trabalho.

Na segunda semana do *sprint* deu-se continuidade à integração, sendo que o autor ficou encarregue de integrar as páginas que tinha desenvolvido anteriormente, sendo essas páginas o detalhe da pesquisa alternativa e a operação extra. Integrar a página de detalhes

foi simples, porém o mesmo não se observou com a operação extra. A página desta operação contava com vários serviços associados: serviços que iriam popular *dropdowns* no formulário existente, dois serviços externos do cliente que seriam chamados em casos específicos e, ainda, um serviço desenvolvido pela equipa de *Backend* para outros casos genéricos.

No último dia do *spint* foi comunicado aos restantes elementos da equipa de *Frontend* todos os pontos que o autor considerou relevantes de melhorar ou alterar na aplicação, tais como as questões de otimização da aplicação, as correções no código existente, questões de responsividade da aplicação, entre outros, bem como o estado do seu trabalho e outros detalhes. O que estava previsto foi concluído e deu-se, assim, por terminado o estágio curricular.

6. Cronograma

No seguinte cronograma, é possível observar quais as tarefas em que o autor participou durante o projeto.

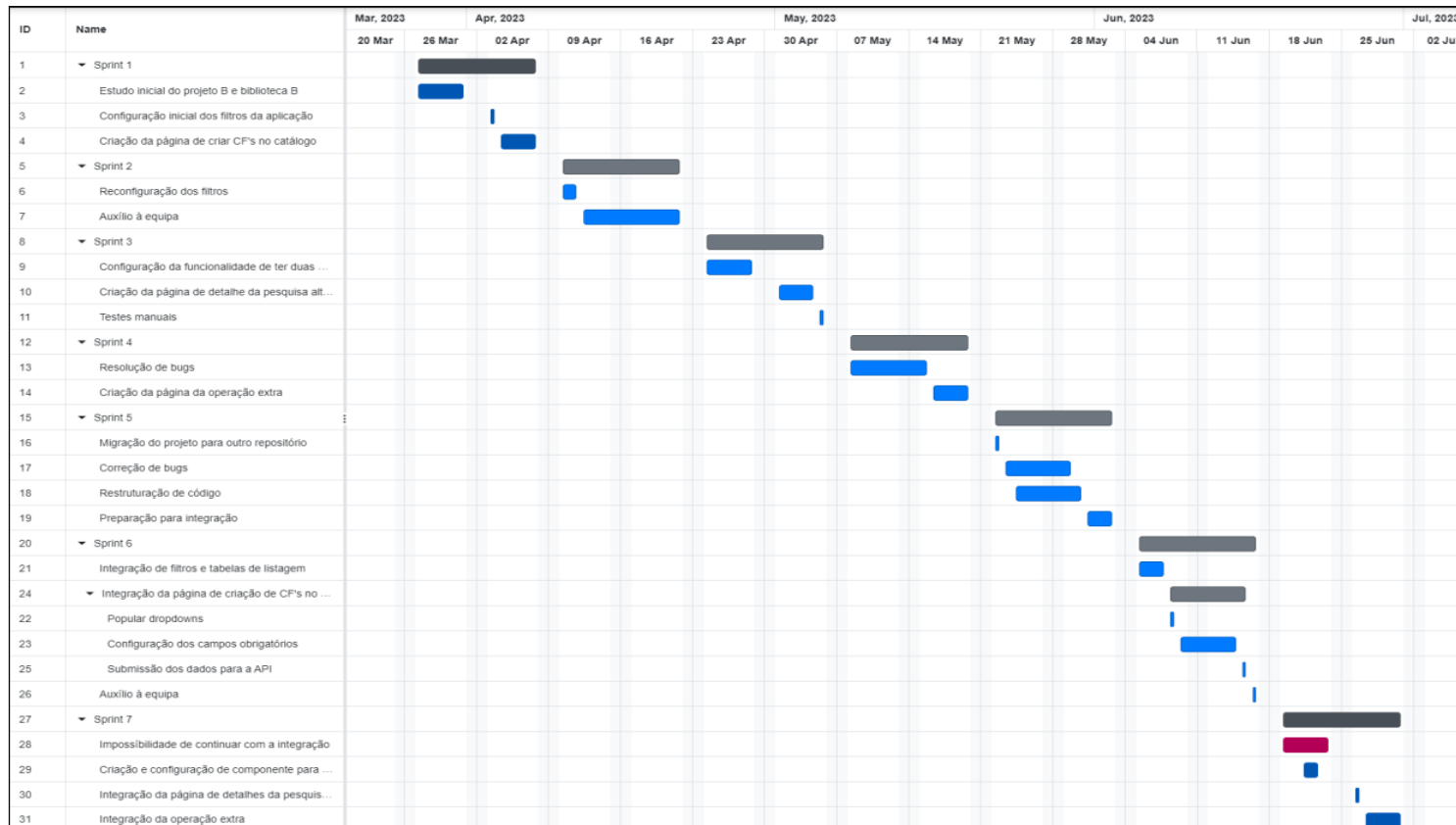


Figura 75 - Cronograma do projeto

7. Meios Previstos e Meios Necessários

No desenvolvimento do projeto de software foram necessários meios específicos para garantir um ambiente adequado. Para tal, foi essencial contar com um computador suficientemente capaz de suportar as atividades de desenvolvimento e uma conexão estável à internet. Para além disso, foi necessário utilizar um software de edição de texto para escrever o código, sendo que a escolha do software ficou ao critério do programador, que optou por utilizar o *Visual Studio Code*, que corresponde a uma ferramenta com a qual este já tinha experiência e se sentia confortável.

Quando se refere um "computador suficientemente capaz", refere-se a um sistema com as especificações técnicas apropriadas para executar as tarefas de desenvolvimento de forma eficiente. Isto inclui ter um processador com velocidade e capacidade de processamento adequadas, uma memória RAM suficiente para executar várias tarefas de modo simultâneo, um armazenamento adequado para os ficheiros e recursos necessários e ainda uma placa gráfica que possa suportar os requisitos visuais. É importante considerar as necessidades específicas do projeto e do ambiente de trabalho ao determinar os requisitos do computador. No caso do autor do relatório, a escolha do *Visual Studio Code* como software de edição de texto foi baseada na familiaridade e conforto do programador com a ferramenta, o que contribuiu para uma experiência de desenvolvimento mais eficiente e personalizada.

8. Problemas e Decisões

Ao longo do projeto apresentaram-se inúmeros desafios, sendo que os mais complexos foram originados pelo cliente e as suas equipas. No início do projeto, a maioria da equipa de desenvolvimento não tinha acesso ao projeto, o que atrasou o arranque do mesmo. Este atraso na disponibilização dos ambientes de desenvolvimento provocou um impacto numa fase mais avançada do projeto, pelo facto de a integração com os serviços de *Backend* não ser feita aquando do desenvolvimento da *interface*. Esta situação originou diversas incoerências entre o *Backend* e o *Frontend*. A situação que ocorreu no *sprint* n.º 7, no qual foi apagada a base de dados da aplicação, provocou, ainda mais, atrasos e impedimentos de trabalho.

No decorrer do desenvolvimento do projeto, várias decisões tiveram de ser tomadas, maioritariamente relacionadas com a resolução de problemas na implementação de

funcionalidades ou no desenvolvimento da *interface*. Todas as decisões foram discutidas entre os elementos da equipa de *Frontend*, sendo que as decisões menos impactantes não necessitavam de discussão e o programador tinha a liberdade de as tomar autonomamente.

9. Considerações Finais

É importante destacar a experiência vivida desde o início do estágio que foi, asseguradamente, excelente. A implementação da academia é uma iniciativa muito positiva que as bases necessárias das tecnologias e enquadra os estagiários nas metodologias de trabalho da empresa. Ter a oportunidade de participar num projeto “real” para um dos principais clientes da empresa foi algo que superou as expectativas iniciais. Poder acompanhar todo o planeamento do projeto e poder participar em reuniões periódicas com o cliente foram experiências que não se esperava serem proporcionadas a um estagiário.

Ao longo do estágio foram desenvolvidas várias competências, tanto técnicas, na qual se aumentaram os conhecimentos de base adquiridos anteriormente e se desenvolveram e adquiriam outros novos conhecimentos, tal como pessoais, tais como ao nível da proatividade e da tomada de decisões, da superação da timidez e da autonomia.

No que se refere ao contributo ao nível do projeto, considera-se que este foi bastante positivo, demonstrando profissionalismo. Considera-se que tudo o que foi desenvolvido foi realizado de um modo eficaz e concluído com qualidade.

Glossário

API - Uma *interface* de programação de aplicações é uma forma de dois ou mais programas de computador comunicarem entre si. É um tipo de *interface* de software que oferece um serviço a outras partes do software.

Backend - O desenvolvimento *Backend* significa trabalhar em software do lado do servidor, que se concentra em tudo o que não se vê num sítio Web. Os programadores de *Backend* asseguram que o sítio Web funciona corretamente, concentrando-se nas bases de dados, na lógica de *Backend*, na *interface* de programação de aplicações, na arquitetura e nos servidores.

Componentes (ou *components*) - Os componentes são partes do código independentes e reutilizáveis. Estes têm a mesma finalidade que as funções JavaScript, mas funcionam isoladamente e retornam HTML. Existem dois tipos de componentes: os componentes de classe e os componentes funcionais.

CRUD - É um acrónimo que surge da área da programação e que se refere às quatro funções consideradas como necessárias para implementar uma aplicação de armazenamento persistente: criar, ler, atualizar e eliminar.

DOM - O *Document Object Model* é uma *interface* multiplataforma e independente da linguagem que trata um documento HTML ou XML como uma estrutura em árvore, em que cada nó é um objeto que representa uma parte do documento.

Frontend - O desenvolvimento Web *Frontend* é o desenvolvimento da *interface* gráfica do utilizador de um sítio Web, através da utilização de HTML, CSS e JavaScript (ou outras tecnologias derivadas), para que os utilizadores possam visualizar e interagir com este sítio Web.

Funções - São módulos de código "autónomos" que realizam uma tarefa específica. As funções habitualmente "recebem" dados, processam-nos e "devolvem" um resultado. Quando uma função é escrita, esta pode ser utilizada vezes sem conta. As funções podem ser "chamadas" a partir do interior de outras funções.

Hooks (ou React Hooks) - São funções JavaScript simples que se pode utilizar para isolar a parte reutilizável de um componente funcional. Funcionam de modo semelhante como as funções, porém permitem o armazenamento de estados.

JavaScript - Frequentemente abreviado como JS, é uma linguagem de programação que é uma das principais tecnologias da *World Wide Web*, juntamente com HTML e CSS.

localStorage - É uma funcionalidade do navegador Web que permite que as aplicações Web armazenem dados localmente no dispositivo de um utilizador.

Modal - Também designado por janela modal ou *lightbox* é um elemento da página Web que é apresentado à frente de todos os elementos da página e desativa todos os outros conteúdos da página. Para regressar ao conteúdo principal, o utilizador tem de interagir com o modal, concluindo uma ação ou fechando-o.

Prop Drilling - Ocorre quando um componente pai transporta dados para os seus filhos e, de seguida, estes transportam os mesmos dados para os seus próprios filhos. Este processo pode continuar indefinidamente. No final, é uma longa cadeia de dependências de componentes que poderá ser difícil de gerir e manter.

Props - É uma palavra-chave no React que significa propriedades que são usadas para transportar dados de um componente para outro.

ReactJS - É uma biblioteca JavaScript *Frontend* gratuita e de código aberto para a construção de *interfaces* de utilizador baseadas em componentes. É mantida pela *Meta* e por uma comunidade de programadores individuais e de empresas.

ReduxJS - É uma biblioteca JavaScript de código aberto para gerir e centralizar o estado de aplicações. É recorrentemente utilizada com bibliotecas como React para construir *interfaces* de utilizador.

Renderização - É o processo do React de descrever uma *interface* de utilizador com base no estado atual da aplicação e nos *props*. A renderização inicial numa aplicação React é a primeira renderização quando a aplicação é iniciada, enquanto que a re-renderização ocorre quando ocorre uma mudança no estado para descobrir quais as partes da UI que precisam de uma atualização.

TypeScript - É uma linguagem de programação de alto nível gratuita e de código aberto, desenvolvida pela *Microsoft* que adiciona tipagem estática com anotações de tipo opcionais ao JavaScript.

useEffect - Permite que os componentes reajam a eventos do ciclo de vida, como montagem no DOM, nova renderização e desmontagem.

useState - *Hook* que é utilizado para armazenar variáveis que fazem parte do estado da sua aplicação e que serão alteradas à medida que o utilizador interage com o seu *website*.