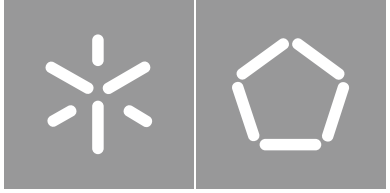
Universidade do Minho
Escola de Engenharia

Nelson Tiago Silva Sousa

Sistematização do desenvolvimento de interfaces web

Dezembro de 2021



Universidade do Minho
Escola de Engenharia

Nelson Tiago Silva Sousa

Sistematização do desenvolvimento de interfaces web

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sob a orientação de:
José Creissac Campos

Dezembro de 2021

Agradecimentos

A realização desta dissertação contou com importantes apoios e incentivos, sem os quais não se teria tornado realidade e aos quais estarei eternamente grato.

Ao Professor Doutor José Francisco Creissac Freitas Campos pelo incansável apoio e disponibilidade total para resolver dúvidas, pelos contributos, opiniões e correções de grande relevância e pelo saber que transmitiu.

À academia da Universidade do Minho, e à direção de curso de Engenharia Informática, pelas boas condições de trabalho e de formação académica, que permitiram, ao longo destes cinco anos, adquirir os conhecimentos e valores necessários para a minha formação enquanto profissional e pessoa.

Aos meus amigos, e cara-metade, pelo apoio incondicional, amizade, e incentivo, sem os quais não teria sido possível ultrapassar os diversos obstáculos desta jornada.

Por último, agradecer à minha família, em especial aos meus pais, por todo o apoio e carinho prestados, pelo incentivo na perseguição da formação académica, e por todos os sacrifícios, por eles feitos, para que eu cumprisse os meus objetivos.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Vila Nova de Famalicão, 28 de Dezembro de 2021
(Local) (Data)

(Nelson Tiago Silva Sousa)

"Sometimes it is the people no one can imagine anything of who do the things no one can imagine." (Alan Turing)

Resumo

Sistematização do desenvolvimento de interfaces web

Esta dissertação aborda o processo de implementação de interfaces web, mais concretamente, utilizando a *framework React*. As interfaces de utilizador são peças fundamentais de qualquer produto computacional interativo. Uma boa interface consegue conquistar o utilizador e fazer com que este utilize o produto, enquanto uma interface de menor qualidade pode ser a causa para a pouca utilização de um *software*. Por este motivo, existem abordagens e metodologias focadas na criação de interfaces, para proporcionarem uma boa experiência ao utilizador e fazer com que este utilize o *software* desenvolvido.

Após a conceção da interface, é necessário proceder à sua implementação. Para isso existem diversas tecnologias e abordagens. Entre as diferentes tecnologias há ainda múltiplas *frameworks* de desenvolvimento, cada uma com as suas características específicas, o que dificulta, por exemplo, a transição de uma tecnologia para outra. O ideal seria tornar o processo de desenvolvimento de uma interface o mais independente possível da tecnologia a ser utilizada.

Tendo em vista a resolução deste problema a dissertação apresenta duas contribuições principais.

Um processo de interpretação do *design* e da sua divisão em componentes, com o objetivo de maximizar a reutilização de código e consequentemente a eficácia no processo de implementação. A divisão do *design* é feita através de uma abordagem atómica, onde componentes mais atómicos se juntam e formam componentes mais complexos.

A criação de uma arquitetura genérica capaz de representar uma aplicação *React*, com o objetivo de fornecer uma visão de mais alto nível, mostrando todas as diferentes entidades que existem na arquitetura de uma aplicação, e também a forma como estas entidades se relacionam. Isto permite uma separação de responsabilidades, separando a definição da interface, da sua lógica de negócio, e da interação com serviços externos.

Além disso, a arquitetura genérica serviu de ponto de partida para a criação de uma estrutura de organização do código capaz de suportar o crescimento dos projetos ao longo do tempo. Estrutura que facilita, e sistematiza, o trabalho dos programadores, dado que estes ficam a saber exatamente onde têm de inserir determinados novos ficheiros, ou onde está um qualquer ficheiro que precisa de ser alterado quando é necessário atualizar um componente da interface.

Por último, para provar que os conceitos descritos anteriormente são aplicáveis, para ajudar os programadores a aplicá-los, e para sistematizar o processo de implementação, criou-se uma ferramenta de

geração de código. A ferramenta permite criar diferentes partes da arquitetura genérica automaticamente. É também possível gerar um componente *React* partindo de um protótipo de uma interface.

Palavras-chave: interface, implementação, *design*, componente, *React*, *framework*, arquitetura de *software*, metodologia

Abstract

Systematization of web interface development

This dissertation addresses the process of implementing web interfaces, more specifically, utilizing the React framework. The user interfaces are fundamental parts of any interactive computer product. A good interface is able to conquer its users and make them use the product, while a lesser quality interface may be the cause for the little usage of a software. For this reason, there are approaches and methodologies focused on the designing of interfaces, in order to provide a good experience to the user and make them use the developed software.

After designing the interface, it is necessary to proceed with its implementation. For this, there are several technologies and approaches. Among the different technologies, there are also multiple development frameworks, each with its characteristics, which makes it difficult, for example, to transition from one technology to another. The ideal would be to make the process of interface development as independent as possible of the specific technology used.

Intending to solve this problem, the dissertation presents two main contributions.

A process of interpretation of the design and division into components, to maximize the reuse of code and consequently the efficiency of the implementation process. The division of the design is done through an atomic approach, where more atomic components come together and form more complex components.

The creation of a generic architecture capable of representing a React application, with the goal of providing a higher-level view, showing all the different entities that exist in an application's architecture, and also how these entities relate to each other. This allows a separation of responsibilities, separating the definition of the interface, its business logic, and the interaction with external services.

Furthermore, the generic architecture served as a starting point for the creation of a structure for code organization, capable of supporting the growth of projects over time. A structure that makes the programmer's work easier and more systematic, since they know exactly where to insert new files, or where to find a file that needs to be changed, when it is necessary to update a component of the interface.

Finally, a code generation tool was created, to prove that the concepts described above are applicable, to help programmers apply them, and to systematise the implementation process. The tool allows programmers to create different parts of the generic architecture automatically. It is also possible to generate a React component starting from a prototype of an interface.

Keywords: interface, implementation, design, component, *React*, framework, software architecture, methodology

Índice

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Siglas

CRF Cameleon Reference Framework 5

MBUID Model-Based User Interface Development 1, 3, 5, 6
MDD Model Driven Development 1, 4, 5

PIM Plataform Independent Model 5
PSM Plataform Specific Model 5

UCD User-Centered Design 1, 2, 3, 4
UI User Interface 1, 2, 3, 6, 8, 28, 33, 41
UX User experience 1, 2

1 Introdução

1.1 Contexto

Qualquer produto computacional interativo tem como uma das suas principais componentes a interface de utilizador (UI, do inglês *User Interface*), componente que contem as vistas a que o utilizador final tem acesso e que permite que este interaja com o produto. O desenvolvimento desta componente é uma área já muito investigada no âmbito do desenvolvimento de software. Conceitos como *User experience (UX)*, e abordagens como o *User-Centered Design (UCD)*, descrito no standard ISO13407 (1999), surgem neste sentido e têm como principal propósito que a interface final seja de boa qualidade, isto é, percetível e de fácil interação para os utilizadores finais.

A qualidade de uma interface pode ser descrita em diferentes dimensões. Do ponto de vista do utilizador, as interfaces são de boa qualidade quando a experiência de utilização é simples e agradável. Quando a interface permite aos utilizadores utilizar com facilidade as funcionalidades que um produto de software fornece, então esta interface é de boa qualidade. No ponto de vista do programador, as interfaces possuem qualidade quando estão bem implementadas, quando o código é bem estruturado, quando é fácil reutilizar partes da interface, e quando é fácil alterar ou adicionar elementos de uma interface sem causar erros nas restantes partes da mesma. Esta diferença reflete-se, até, em distintas abordagens ao desenvolvimento de software, mais focadas nos utilizadores, ou mais focadas na arquitetura do software, tal como constatado por John et al. (2003).

Do mesmo modo, tal como o desenvolvimento de software possui a abordagem de *Model Driven Development* (Hailpern & Tarr, 2006), também o desenvolvimento de interfaces possui a sua própria abordagem de desenvolvimento orientados a modelos, a *Model-Based User Interface Development* (Meixner & Calvary, 2014). Esta abordagem incentiva a criação e utilização de modelos abstratos para representar as interfaces do nosso sistema. O objetivo é refinar os modelos até ao código final que representa a interface.

O desenvolvimento de interfaces pode, então ser dividido em duas grandes etapas: o *design* da interface, responsabilidade dos *designers*, e a sua implementação, responsabilidade dos engenheiros de software. O *design* consiste em criar o *layout* e todos os elementos que compõem a vista do produto.

No caso da web o *design* refere-se à criação de todas as páginas que compõem a aplicação. Esta parte é realizada por pessoas mais especializadas nas áreas de *design*, UI/UX, e UCD. Habitualmente são construídos protótipos, e/ou imagens que procuram projetar o que se pretende obter como produto final.

A implementação tem como objetivo materializar o *design*, seguindo os protótipos elaborados pelos *designers* na fase anterior. Esta etapa é tão boa quanto mais idênticos forem o resultado final e o *design* criado inicialmente. Nesta fase os principais responsáveis são os programadores, que usando uma ou mais tecnologias do vasto leque disponível atualmente, convertem os protótipos no código que representa o produto final. Tudo isto pode levar algum tempo em função da complexidade da interface, e da experiência dos programadores.

Ambas as fases deste processo podem possuir varias iterações. O processo proposto pelo UCD é por natureza iterativo sendo realizadas diversas alterações no protótipo ao longo do tempo. A implementação de um *design* também pode ser um processo iterativo. Além disso, a implementação pode incluir ainda repetições não desejáveis, porque a interface implementada não respeitou o *design* e foi recusada, ou se existirem alterações ao *design* após este estar implementado.

Posto isto, um processo de implementação ineficiente e lento faz com que todo o desenvolvimento do produto seja atrasado ao longo destas iterações.

1.2 Problema

Atualmente, os programadores possuem uma grande diversidade de soluções metodológicas e tecnológicas com as quais é possível implementar uma interface web. Desde a utilização de linguagens como HTML, Javascript ou CSS, à utilização de *frameworks* de desenvolvimento de interfaces como *Angular* ou *React*, ou até a utilização de plataformas *low-code* como a *Outsystems* (Martins et al., 2020) que permitem a criação de produtos de software completos com pouca utilização de programação.

Esta diversidade faz com que a implementação não possua um método sistemático, praticado pela generalidade dos programadores, com o propósito de os guiar durante todo o processo, dos mais experientes aos menos experientes, de forma a que este seja realizado o mais rápido e eficazmente possível.

Existem alguns padrões arquiteturais como o *Model View Controller* (MVC) (Krasner, Pope et al., 1988), o *Model View Presenter* (MVP) (Potel, 1996), e o *Model View View Model* (MVVM) (Britch, 2017), que especificam a arquitetura das interfaces, e a forma como comunicam com as suas diferentes partes. Contudo, estes modelos são de alto nível e não especificam com detalhe as diferentes parte de uma interface.

A falta de método, faz com que algumas das interfaces sejam desenvolvidas de maneira acoplada e ineficiente, não existindo uma separação de responsabilidades. Isto leva a que o processo seja mais lento, e a manutenção também mais difícil, já que não existe reutilização de código e as paginas são definidas como peças de software atómicas, mesmo que existam partes iguais em páginas diferentes.

Tendo em conta esta realidade, os programadores podem levar mais tempo até conseguir entregar

uma implementação que corresponda a 100% ao *design* pedido. A razão é não trabalharem com recurso a um método sistemático, ou com arquitetura em mente. Tudo isto se torna ainda pior quando o processo sofre várias iterações e são elaboradas consecutivas versões do *design* e da implementação, havendo um constante "passar do testemunho" entre os *designers* e os programadores até que tudo esteja concluído.

A "passagem de testemunho", aquando da entrega do *design* aos programadores, pode trazer problemas. O *design* pode não estar totalmente claro, levando o programador a voltar ao *designer* para esclarecer possíveis dúvidas. O programador pode também ser menos experiente e demorar mais do que o previsto a implementar a interface pedida, pode até não conseguir um produto final totalmente igual ao estipulado inicialmente.

Para combater esse problema, tal como o *design* da interface possui sistematizações capazes de obter uma interface abstrata a partir do modelo de domínio, e de tarefas, também era de grande interesse sistematizar o processo de implementação da mesma, desde o ponto em que o *design* está feito até ao código correspondente. Criando um modelo/*guideline* a seguir, boas práticas e padrões de desenho.

Na parte da implementação, após o *design* estar decidido, são menos os artigos que tentam abordar e generalizar este processo. A falta de normas, e padrões de desenho na programação de interfaces é evidente. Deixando os programadores sem ponto de referência.

Gamma et al. (1994) compilaram uma série de padrões de desenho, mundialmente conhecidos hoje em dia, que podem ser reutilizados na programação por objetos para responder a determinado problema. Estes padrões arquiteturais acrescentam processo e método ao desenvolvimento orientado a objetos. Um programador sabe que ao ter de resolver determinado problema recorrente, contemplado num destes padrões, tem por onde se guiar de modo a cometer o menor número de erros inesperados.

Também na parte da implementação da interface existe a necessidade e o espaço para o aparecimento de padrões de desenho de determinado componente ou funcionalidade, com o propósito de sistematizar o processo de desenvolvimento da UI, para que este se torne mais consistente, diminuindo assim a sua duração e possíveis erros resultantes.

1.3 Enquadramento

Tal como foi dito anteriormente, *User-Centered Design* e *Model-Based User Interface Development* são duas abordagens de *design* e desenvolvimento de interfaces. No presente capítulo irão ser descritas estas duas abordagens bem como o enquadramento da dissertação nas mesmas.

1.3.1 User-Centered Design

Uma área que visa dar resposta à dificuldade de desenhar interfaces de qualidade é o desenvolvimento centrado no utilizador (UCD, do inglês *User-Centered Design*), que segundo Ritter et al. (2014) significa:

> "considering human characteristics and capabilities during system design. It means explicitly asking: who is going to use the system/technology and why(...). Being user-centered means knowing why, as well as how, users do what they do when they do it".

Trata-se de desenvolver software pensando segundo a perspetiva do utilizador, deixando de lado falsas assumpções sobre o que este sabe, ou perceciona. Os *designers* mesmo que se revejam como utilizadores do produto onde estão a trabalhar, não devem restringir o grupo de utilizadores a si mesmos. Maioritariamente porque possuem uma visão global do software e das funcionalidades às quais ele dá resposta, coisa que os utilizadores finais não têm.

O processo de desenvolvimento centrado no utilizador é composto por várias etapas. Em primeiro lugar temos a análise, onde são revistos os requisitos do produto, estudados os tipos de utilizadores, as tarefas que estes podem fazer na interface, e os fluxos da mesma. Em seguida temos a fase de implementação, onde são elaborados modelos de tarefas que especificam as ações que um utilizador pode realizar na interface, modelos conceptuais do *design* que representam a interface, e modelos de navegação que representam o fluxo entre janelas ou páginas. Nesta fase são também criados protótipos do *design* final. Por último temos a fase de testes e avaliação onde se verifica a validade do *design* recorrendo a testes aos protótipos, heurísticas como as de Nielsen e Molich (1990), e *design walkthoughs* como, por exemplo, o *Cognitive Walkthough* (Blackmon et al., 2002). Caso existam alterações a fazer, volta-se a iterar o processo.

Após os protótipos estarem testados, e validados, estes representam a interface final a ser implementada. Começa a partir deste momento o processo de implementação, que é o foco desta dissertação.

Esta abordagem proporciona inúmeras vantagens aos utilizadores. Seguindo o exemplo de Galitz (2007), num sistema que necessite da análise de 4.8 milhões de ecrãs por ano, uma pequena falha na interface que leve a um atraso de 1 segundo em cada ecrã refletia num aumento de quase um ano de trabalho por pessoa para processar todos os ecrãs. O UCD permite colmatar estas falhas na interface de modo que os utilizadores possuam a melhor experiência possível com o software. Também Vredenburg et al. (2002) concluíram, após um estudo a uma centena de trabalhadores, que os métodos de UCD melhoram a usabilidade do produto.

No entanto, após estabelecido um *design*, é necessário passar à sua implementação.

1.3.2 Model-Based User Interface Development

A utilização de modelos no desenvolvimento de *software* é já uma abordagem comum. O *Model Driven Development (MDD)* (Hailpern & Tarr, 2006) é uma metodologia de desenvolvimento que tem como ponto de partida o desenvolvimento de modelos para representar um sistema. Os modelos dão uma visão geral do produto mesmo antes de estar pronto. Esta abordagem traz como vantagens a capacidade de clarificar conceitos-chave perante toda a equipa, bem como a arquitetura base do projeto, o que ajuda a prever futuros problemas e arranjar soluções para os mesmos, antes de começar a programar. Os

programadores abstraem-se do código e pensam primeiro no produto de software de forma abstrata, o que acelera o processo de implementação.

Este processo possui três fases, a elaboração de um *Plataform Independent Model (PIM)*, de seguida a elaboração de um *Plataform Specific Model (PSM)*, e por último a implementação do código. O PIM, é o modelo independente da tecnologia, representa o produto, mas sem ter em consideração quais as tecnologias ou linguagens em que este vai ser implementado. O modelo de conceptual da base de dados é um bom exemplo de um PIM, visto que representa a base de dados sem qualquer tipo de dependência sobre qual o motor de base de dados que iremos utilizar.

O PSM, é a segunda fase, e representa os modelos dependentes da tecnologia, nestes modelos já estão especificados detalhes sobre a tecnologia a ser usada. Por exemplo, um modelo de classes ao nivel do PSM, representa as classes e a forma como elas se relacionam, varia consoante a tecnologia usada. Se usarmos J2EE iremos obter classes e bibliotecas diferentes daquelas que teríamos caso usássemos .NET. Este ultimo modelo, permite-nos ter uma visão geral do produto de software tendo em conta a tecnologia que irá ser utilizada, o que facilita o desenvolvimento do código, que é a última fase do processo.

No âmbito do desenvolvimento de sistemas interativos, existe o *Model-Based User Interface Development (MBUID)* que segundo Meixner e Calvary (2014) visa diminuir o esforço necessário para desenhar e implementar interfaces. Este é o equivalente ao MDD, mas na área das interfaces de utilizador. A MBUID propõe a utilização de modelos de alto nível que representem a interface do sistema, e permitam aos *designers* analisar o mesmo antes de partir para a sua especificação concreta.

Diversos autores abordam a *Model-Based User Interface Development*. Calvary et al. (2001) propõem uma *framework* para lidar com a transformação de interfaces para diferentes dispositivos usando também a abordagem por modelos. Esta viria a ser conhecida e amplamente aceite por *Cameleon Reference Framework (CRF)*.

A CRF divide o seu processo em 4 fases. A *Task-oriented specification* consiste em elaborar os modelos de tarefas que caracterizam a interação entre o utilizador e o sistema. Após a sua elaboração, passamos à definição da interface propriamente dita.

Primeiro temos a *Abstract Interface*, um modelo que representa a interface com todos os seus componentes e relações entre os mesmos. Nesta fase todos os componentes são independentes da tecnologia e da implementação, por isso é-lhe atribuída a designação de abstrata. Segue se a *Concrete Interface*, desenvolvida de forma independente da tecnologia a ser utilizada na implementação, mas já refletindo o tipo de UI a construir, se é para um telemóvel, por exemplo, ou para um computador. Esta, sofrendo as devidas transformações, transforma-se no código final que dá origem à interface. Por último, temos então a *Final Interface* que corresponde ao código-fonte que compõe a interface em HTML, Java, ou outra qualquer tecnologia.

A presente dissertação enquadra-se no processo transformação de uma *Concrete Interface* numa *Final Interface*. Assumindo como *Concrete Interface* um *mockup* elaborado pelos *designers* que representa a interface final. Uma questão que se pode então colocar é qual o nível de automatização na geração de código que devemos considerar.

A geração de interfaces a partir de modelos foi desde cedo estudada. Ver, por exemplo, as análises realizadas por Szekely (1996) ou por Pinheiro (2001). Szekely (1996) aborda várias ferramentas de geração de interfaces com base em modelos e propõe uma arquitetura genérica para estas ferramentas. Pinheiro (2001) comparou quatorze ambientes diferentes de desenvolvimento em MBUID. Estes ambientes fornecem ferramentas de geração do *design* e do código da interface com base em modelos. O autor concluiu que embora os ambientes de desenvolvimento estivessem estáveis e prontos a ser comercializados, ainda existiam pontos a ser estudados e melhorados para estes terem uma maior aceitação. Pinheiro refere como uma das lacunas, a necessidade de ajustamentos manuais que as interfaces geradas por estas ferramentas necessitam.

Na verdade, este é um problema que se mantém atual e a conclusão é que, a não ser para domínios muito específicos, apesar do potencial menor esforço necessário para desenvolver a interface, o processo de geração não deve ser totalmente automatizado. A UI tem um papel demasiado importante num produto de software, visto que é a peça que está em contacto com os utilizadores, capaz de os atrair ou repelir do produto. Uma abordagem totalmente automática na sua criação traria falhas ao nível da experiência do utilizador, em termos visuais e de interação com o sistema. Isto porque cada interface deve adaptar-se ao produto que representa e aos utilizadores, até em pequenos pormenores. Logo ao automatizar a sua criação o foco nestes pequenos pormenores e na interface em geral seria menor, abrindo espaço para falhas, já que os modelos são uma visão abstraída da interface.

Uma solução para este problema seria incluir detalhe suficiente nos modelos, para que estes gerassem interfaces específicas o suficiente para cada produto de software. Contudo, este nível de detalhe, iria aumentar a dificuldade e o tempo de elaboração dos modelos, diminuindo a sua utilidade enquanto ferramentas de *design*.

1.4 Objetivos

Esta dissertação é então motivada pela atual lacuna de um método sistemático a seguir no processo de implementação de interfaces, e da falta de padrões de desenho/*guidelines* que auxiliem os programadores, tornando o processo de desenvolvimento mais ágil e eficaz. O objetivo é compilar um conjunto de boas práticas, técnicas, padrões, ou arquiteturas que auxiliem o programador a implementar um *design* recebido, criando assim um método que, quando seguido passo a passo, transforme um prototipo de interface no seu respetivo código. Este propósito apresenta-se importante na medida em que melhoraria um processo realizado, imensas vezes, por todas as empresas de desenvolvimento de software web existentes.

Como objeto de estudo utilizar-se-á a *framework React*, visando aplicar na prática os conceitos abordados. Apesar da escolha desta *framework*, as práticas e conceitos desta dissertação são genéricos a todo o desenvolvimento de interfaces web.

Por último, e para mostrar a validade destes conceitos e do método criado, irá ser implementado um

protótipo de uma ferramenta. Esta ferramenta possuirá funcionalidades capazes de automatizar partes do processo, tornando-o ainda mais sistemático. Uma das funcionalidades terá como objetivo partir de um protótipo da interface e gerar o código correspondente.

1.5 Estrutura do documento

Após este primeiro capítulo com o enquadramento do problema, no Capítulo 2 é levantado o estado de arte do problema, para perceber o que já é feito de bom, e o que ainda necessita de investigação e melhorias no âmbito do desenvolvimento de interfaces. Na Secção 2.1, abordam-se as várias alternativas de implementação de interfaces. Na Secção 2.2 é feita uma comparação entre as *frameworks* mais usadas na atualidade. A seguir, na Secção 2.3, é abordada a técnica de desenvolvimento orientado ao componente. A abordagem *Atomic design*, que defende a divisão dos diversos elementos de um *design* é discutida na Secção 2.4. Por último, na Secção 2.5 são discutidas as melhores formas de implementação de estilos em componentes.

No Capítulo 3 estudam-se as aplicações *React*. Na Secção 3.1 são analisadas diversas aplicações *open-source*, e criada uma arquitetura genérica para uma aplicação *React*. Com base nas mesmas aplicações *open-source*, e na arquitetura genérica desenhada, na Secção 3.2 é especificada a estrutura de pastas ideal para uma aplicação *React*.

No Capítulo 4 discute-se a implementação de uma interface em *React*. Inicialmente, na Secção 4.1, explica-se o funcionamento geral da *framework*. A seguir, na Secção 4.2 discute-se a divisão do *design* em diferentes componentes *React*, tendo por base o *Atomic design* referido no estado de arte. Na Secção 4.3 comparam-se componentes de classe e componentes funcionais em *React*. A interação com APIs e criação de serviços é abordada na Secção 4.4.

No Capítulo 5 descreve-se a implementação das ferramentas de geração de código criadas no âmbito da dissertação. Começa-se por descrever o objetivo das ferramentas na Secção 5.1. A seguir, na Secção 5.2, descreve-se a ferramenta de geração de código através de comandos. E por último, na Secção 5.3, descreve-se a ferramenta de geração de código a partir de protótipos.

No Capítulo 6 cria-se o caso de estudo para pôr em prática as ferramentas elaboradas, o objetivo é implementar uma pequena aplicação. Na Secção 6.1 é descrito o caso de estudo, são especificados os requisitos da aplicação, tais como o *design* das páginas e navegação entre elas. A seguir, na Secção 6.2 são descritos todos os passos necessários para proceder à implementação da aplicação, utilizando a ferramenta e os conceitos desenvolvidos na dissertação. Por último, na Secção 6.3, reflete-se sobre a utilidade e a eficácia da ferramenta com base nos resultados do processo de implementação.

No Capítulo 7, escrevem-se as conclusões finais desta dissertação. Resume-se e avalia-se o trabalho feito, tanto a nível conceptual como também a nível prático através das ferramentas criadas. Por último, mencionam-se os tópicos que não foi possível de abordar neste documento, mas que, no futuro, faria sentido explorar.

2

Implementação de interfaces web

Neste capítulo discute-se o estado de arte da implementação de interfaces web. Começando pelas diferentes abordagens de implementação, e em particular pelo uso de *frameworks*. Abordam-se temas como o desenvolvimento orientado ao componente, muito utilizado atualmente, e também o *Atomic Design*, método de interpretar o *design* e dividi-lo em partes consecutivamente mais pequenas. Por último, discutem-se as vantagens e desvantagens das diferentes formas de aplicar estilo às interfaces web.

2.1 Abordagens na implementação de interfaces

Um dos principais entraves à normalização da implementação de interfaces é a grande de quantidade de tecnologias existentes para o fazer. Na área da web, uma página pode ser implementada em HTML+CSS puros, processo cada vez menos usado em projetos de tamanho considerável porque leva muito mais tempo a desenvolver. O mais habitual hoje em dia é recorrer a *frameworks* que oferecem funcionalidades e bibliotecas aos programadores, encurtando o processo de desenvolvimento. Neste âmbito a oferta é vasta, existindo de momento uma quantidade abundante de *frameworks* dedicadas ao desenvolvimento da UI, Wohlgethan (2018) faz uma comparação entre três das mais populares, *React*, *Angular* e *Vue*.

Além disso, a importância da rapidez e da exatidão na implementação das interfaces tem levado ao surgimento de ferramentas cujo objetivo é a geração de código. Existem algumas plataformas, como a Anima App e a OutSystems que geram código a partir de protótipos ou outros modelos da aplicação. A Anima é um *plugin* que utiliza os protótipos feitos em programas como *AdobeXD*, *Sketch*, e *Figma*, permitindo criar *designs* responsivos e gerar código HTML, CSS e mais recentemente *React Components*. A OutSystems é uma empresa portuguesa cujo o produto é uma plataforma *low-code* (Service Studio), que permite desenhar e aprovisionar aplicações web sem quase se necessitar de escrever código. A plataforma trata da geração da aplicação. Tudo o que o utilizador tem de fazer é desenhar a estrutura do software e os *designs* das páginas.

Estas ferramentas apesar de acelerarem o processo, possuem algumas limitações. No caso da

Anima, pode haver partes da interface gerada que não ficaram exatamente iguais ao *design* definido, levando à necessidade de alterações manuais no fim do processo. A OutSystems Service Studio é uma plataforma *low-code* que apesar de possuir muitas funcionalidades, não fornece a mesma liberdade quando comparando com um desenvolvimento mais tradicional.

2.2 Frameworks

Atualmente, o desenvolvimento de interfaces web passa em grande parte pela utilização de *frameworks* como *Angular*, *React*, ou *Vue*, as que apresentam, na atualidade, um maior domínio de mercado. Para além destas existem ainda uma quantidade considerável de *frameworks* menos conhecidas. No site simform[1] é possível encontrar uma lista das melhores *frameworks* neste momento, segundo a opinião do autor.

A Figura 1 mostra uma comparação entre as três principais *frameworks* referidas acima, usando o *Google Trends*, para saber quais as mais procuradas, no motor de busca da *Google*, no último ano. É possível notar que *Vue* (linha vermelha) atingiu um pico superior às restantes, porém, *React* (linha azul) foi mais consistente e é atualmente aquela que possui mais procuras.

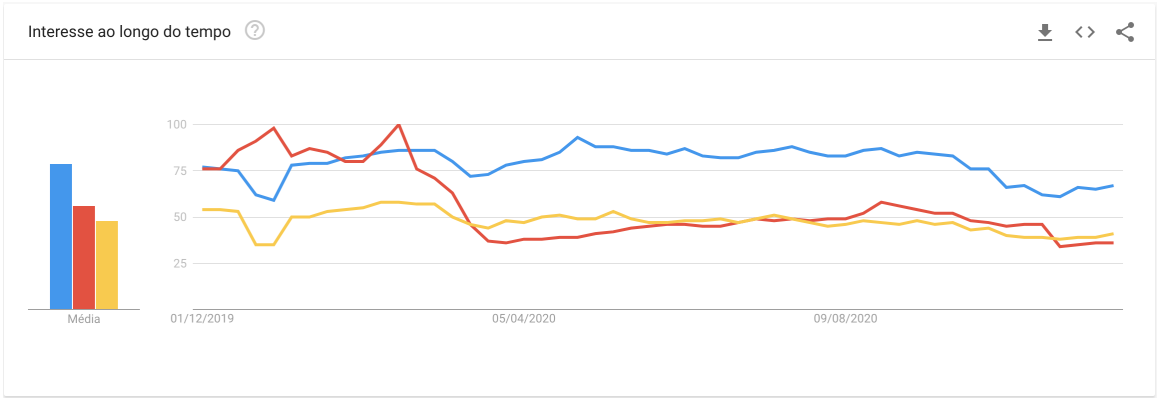


Figura 1: React vs Vue vs Angular by: Google Trends

Estas três *frameworks* possuem uma coisa em comum, todas seguem um modelo de desenvolvimento orientado ao componente. Trata-se de uma técnica poderosa, cada vez mais utilizada no desenvolvimento de interfaces. Iremos explicar melhor este conceito na Secção 2.3. Apesar deste aspeto em comum são também várias as diferenças entre estas 3 tecnologias, em termos de fluxo de dados, linguagens suportadas, arquitetura, utilização por parte de grandes empresas, e suporte a desenvolvimento *mobile*.

Angular[2], a mais antiga das três, é por esse motivo a mais robusta e com mais funcionalidades básicas cobertas à partida. Está, portanto, menos dependente de pacotes externos, tornando a aplicação mais segura em teoria. Contudo, esta maturidade torna a plataforma mais complexa, e com uma curva

[1] https://www.simform.com/best-frontend-frameworks/ - visitado em 20/12/2021
[2] https://angular.io/ - visitado em 23/12/2021

de aprendizagem mais acentuada. Nesta tecnologia os componentes chamam-se *directives* e conseguem passar informação em dois sentidos, de pai para filho e vice-versa, o que pode tornar mais confuso o fluxo de informação. Em termos de linguagens é apenas possível programar em TypeScript, fator que aliado à complexidade da *framework* poderá estar na origem da queda de popularidade e uso da mesma. Ao suportar apenas TypeScript, a *framework* limita as opções e obriga os programadores, que não dominem a linguagem, a gastar tempo extra a aprendê-la.

React[3] distingue-se dos rivais logo ao nível do seu *design*, utilizando um *Virtual-DOM* capaz de ser modificado muito mais rapidamente que o *DOM* real, as alterações são renderizadas para este *Virtual-DOM* e depois compiladas para o *DOM* do *browser*. A informação nesta tecnologia é passada de componente em componente apenas num sentido, de pai para filho, simplificando o fluxo de dados. Outro dos pontos fortes é o uso de *JSX*, técnica que permite escrever HTML de forma mais fácil, interligando o HTML com a própria lógica da vista. Em termos de aprendizagem, esta *framework* é considerada relativamente fácil de aprender, e em poucos minutos é possível ter todo um ecossistema *React* pronto a ser utilizado. Para além, disso através de *React Native* é possível desenvolver Aplicações Android e IOs com muito poucas diferenças relativamente à versão web.

Por último *Vue*[4] é conhecido pela facilidade com que é possível começar a trabalhar, sendo a simplicidade um conceito base da tecnologia. Em termos de conceitos, esta *framework*, reutiliza pontos fortes das suas concorrentes. Utiliza *Web componentes* como o *React* com um fluxo de pai para filho, mas também dá a possibilidade de utilizar os mesmos com comunicação em duplo sentido ao estilo do *Angular*. Todavia a falta de grandes projetos implementados em *Vue* trás uma carência de boas práticas e bibliotecas conceituadas para a comunidade. No âmbito mobile, o *Vue* possui a sua própria *framework* de desenvolvimento, o *Vue Native*.

Em termos de procura em ambiente profissional, à data, a *framework React* é a que mais vagas possui no site *Linkedin*.

Em suma, é possível compilar as diferenças e semelhanças na tabela 1.

	Angular	React	Vue
Fluxo de informação	Bi-directional	Sentido unico	Ambas
Virtual Dom	Não	Sim	Não
Supporte mobile	-	React Native	Vue Native
Linguagens	TypeScript	JavaScript/TypeScript	JavaScript/TypeScript
Vagas no Linkedin	38791	107902	28165

Tabela 1: Tabela de comparação entre frameworks

[3] https://reactjs.org/ - visitado em 23/12/2021
[4] https://vuejs.org/ - visitado em 23/12/2021

2.3 Desenvolvimento orientado ao componente

Desenvolvimento orientado ao componente é um paradigma de programação que visa estruturar o código e as funcionalidades de determinado software por componentes. Os componentes são peças independentes, que juntas formam o produto. Cada componente pode receber o seu respetivo *input* e a partir disso produzir diferentes resultados, consoante a sua lógica interna (Nierstrasz et al., 1993).

No caso das janelas da interface, os componentes podem ser elementos da vista como botões, barras de navegação, formulários entre muitos outros. É possível, e boa prática, utilizar componentes dentro de componentes, numa organização hierárquica. Por exemplo, um componente que defina uma barra de navegação pode incluir um componente que defina um botão.

Esta abordagem permite isolar as várias peças que constituem uma interface, tornando-as independentes e reutilizáveis em contextos diferentes. Analisemos a página principal do *YouTube* para clarificar o conceito. É de fácil perceção que na Figura 2 existem vários elementos que se repetem na página. Con-

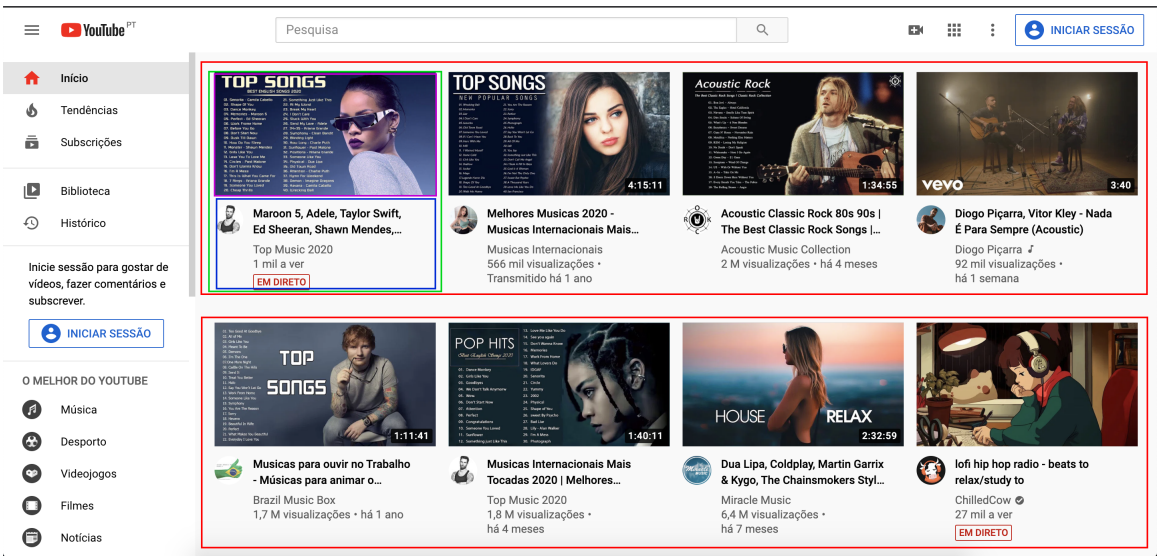


Figura 2: Youtube página principal

seguimos ver duas listas horizontais, onde os elementos são todos cartões que representam vídeos. Cada cartão possui uma imagem e um texto. Utilizando o desenvolvimento orientado a componentes só teríamos que programar dois componentes para fazer esta parte da janela. Um componente que recebesse a imagem e o texto e produzisse o cartão de apresentação do vídeo, e um componente lista que contivesse quatro destes cartões. Usando como exemplo a *framework React* o resultado prático seria o apresentado na Listagem 2.1 (onde VideoCard e VideoList definem os dois componentes referidos anteriormente).

```
1 class VideoCard extends Component {
2   render() {
3     return (
4       <div>
5         <img src={this.props.video.videoImgUrl} />
```

```
6           <div className="video-infos">
7             <span>{this.props.video.videoName}</span>
8             <span>{this.props.video.author}</span>
9             <span>{this.props.video.views}</span>
10            <span>{this.props.video.date}</span>
11          </div>
12        </div>
13      );
14    }
15  }
16
17  class VideoList extends Component {
18    render() {
19      return (
20        <div>
21          this.props.videos.map(video =>{
22            <VideoCard
23              video={video}
24            />
25          })
26        </div>
27      );
28    }
29  }
```

Listagem 2.1: Componentes VideoCard e VideoList em *React*

Com estes dois componentes, e algum estilo adicionado, conseguiríamos então representar as duas listas de vídeos que vemos na Figura 2.

2.4 Atomic design

Um *design* tipicamente pode ser dividido em várias partes, e cada parte por sua vez pode ser sub-dividida em partes mais pequenas. Frost (2016) escreve sobre esta abordagem, e afirma que existe uma analogia entre a química e o *design* de aplicações. O autor sugere que tal como no universo existem os átomos que se agrupam para formar elementos mais complexos como moléculas ou organismos, também num *design* existem componentes de natureza mais simples como botões, *labels*, e *input fields*, que podem ser considerados átomos. Estes átomos, quando agrupados, podem formar moléculas, componentes que reutilizam vários componentes mais simples, como *forms*, e *cards* construidos a partir de botões, *labels*, e *inputs*. Podemos ainda agrupar um conjunto de moléculas para construir um organismo, o que no mundo do *design* significaria compor uma secção à custa de vários componentes mais simples. Um possível exemplo seria uma *navbar* constituída por botões, formulários de pesquisa e imagens. O autor descreve ainda a forma como as várias moléculas são agrupadas numa página como *templates*. Estes

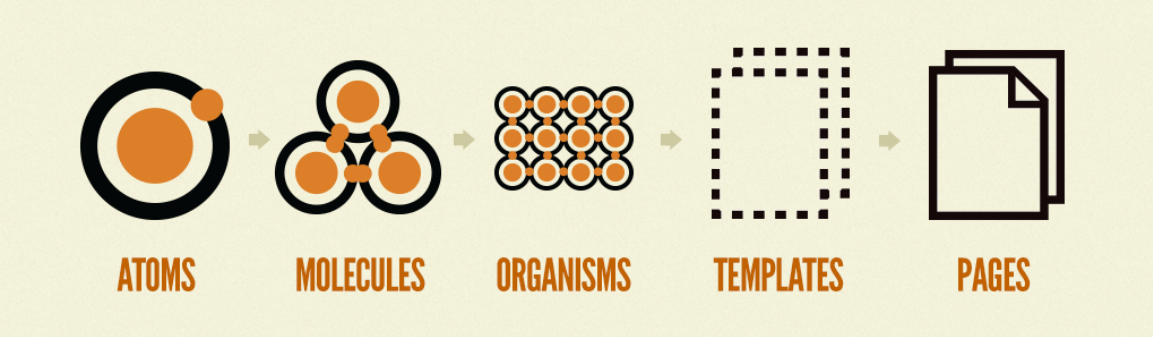


Figura 3: Componentes do atomic *design*, Fonte: https://bradfrost.com/blog/post/atomic-web-design/

templates são utilizados pelas páginas, paginas estas, que são o nível mais alto de encapsulamento, envolvendo todos as outras partes mais simples.

Podemos considerar que estes átomos, moléculas, organismos, *templates*, e paginas são nada mais do que componentes. Uns mais simples e outros mais complexos. Sendo os componentes mais complexos uma composição de componentes mais simples. A Figura 3 resume esta hierarquia de componentes.

Se ao nível do *design* este já é um conceito presente, talvez fizesse sentido que os programadores também pensassem da mesma forma ao receber o *mockup* que irão ter de implementar. Caso consigam olhar para este *mockup* e identificar os componentes e subcomponentes presentes no mesmo, conseguirão tirar melhor partido das *frameworks*, que já são orientadas ao componente. Será, assim, mais fácil perceber o que vão ter de implementar e quais as dependências entre componentes.

A utilização desta abordagem poderá significar um sistematizar do processo e um aumento da reutilização de código face à abordagem comum de começar a programar uma interface como uma só entidade, de forma não modular e sem divisão de responsabilidades. Isto porque ver o *design* deste modo dá-nos uma perspetiva geral da página e da forma como os elementos interagem uns com os outros. O que nos permite dividir responsabilidades, antecipar problemas e tomar decisões as decisões certas logo no início do processo.

Como programadores, quando nos é entregue uma tarefa de implementação de um *design*, é-nos entregue também o *mockup* construído pelo *designer*. É então da nossa responsabilidade olhar para o *mockup* e implementar uma réplica fiel do mesmo na tecnologia escolhida para o projeto. É também neste momento que uma reflexão e um olhar critico sobre a melhor forma de implementar o *mockup* recebido, ao invés de passar de imediato à implementação, nos pode poupar algum tempo e fazer com que surjam menos erros pelo caminho. Posto isto, a capacidade de receber um *design* e isolar os seus diferentes componentes desde os mais simples (átomos) até aos mais complexos (organismos), utilizando esta abordagem, ajuda o programador a ter uma visão mais desacoplada, e a maximizar a reutilização dos componentes. Minimizando os erros e o tempo de desenvolvimento.

Vejamos o exemplo da *homepage* da rede social Facebook, representada na Figura 4. Analisando o *design* é possível notar a presença de 4 organismos independentes. São estes, a barra de navegação no topo, a coluna com o título “Home” à esquerda, a coluna com o título “Suggested” à direita, e ao centro

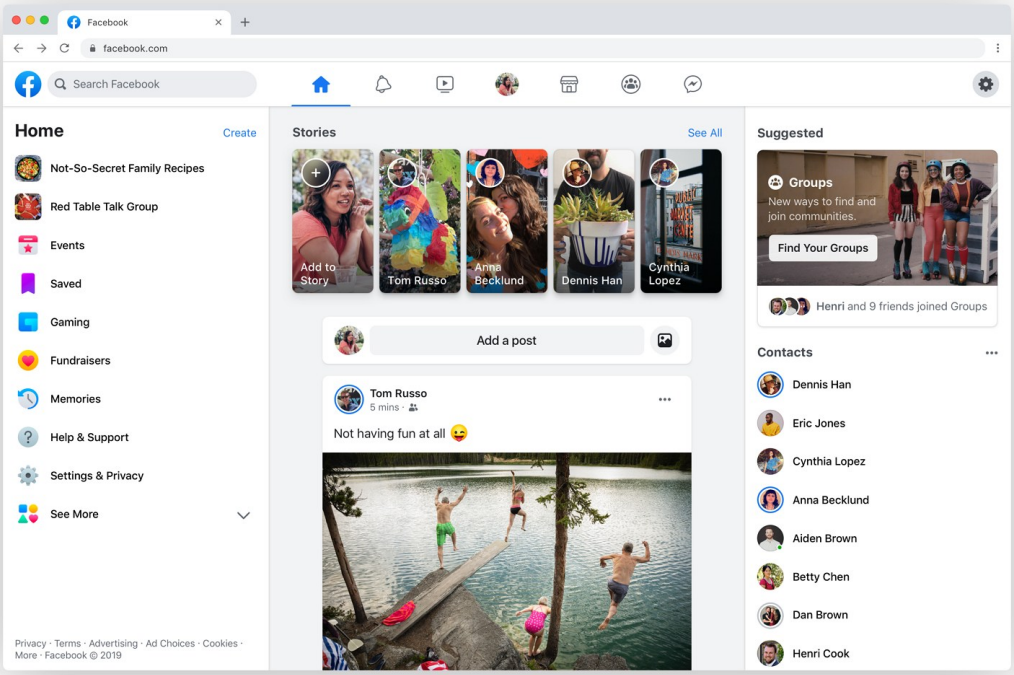


Figura 4: Facebook homepage design

possuímos os "Stories" e a secção do *feed* com publicações. Dentro de cada um destes organismos existem diferentes moléculas menos complexas, como, por exemplo, o cartão de apresentação de cada *storie*, o formulário de pesquisa da barra de navegação, o formulário para adicionar uma nova publicação, entre outros. A um nível ainda mais granular possuímos os nossos átomos, que nesta página são os ícones, os botões, e os textos, estes compõem todos os componentes mais complexos.

Uma análise atenta permite não só identificar os vários tipos de componentes, mas também perceber que existem componentes utilizados em mais do que um organismo. Na coluna da esquerda, possuímos várias entradas representadas com um ícone à esquerda e texto à direita, e nos contactos o mesmo acontece. Podemos então deduzir que todos estes elementos podem reutilizar o mesmo componente, variando apenas a imagem que usam e o seu texto.

A árvore da figura 5 representa os principais componentes da *homepage* do Facebook. É possível ver a azul a página, a vermelho os organismos, a amarelo as moléculas, e a verde os átomos.

2.5 Melhores práticas na aplicação de estilo a componentes

Uma das partes mais cruciais de qualquer interface é o seu estilo. Estilo refere-se a um conjunto de regras sobre como os elementos que estão presentes na página devem ser apresentados. Sem este, ficamos apenas com um conjunto de elementos HTML, todos com a mesma aparência. Como podemos ver pelas

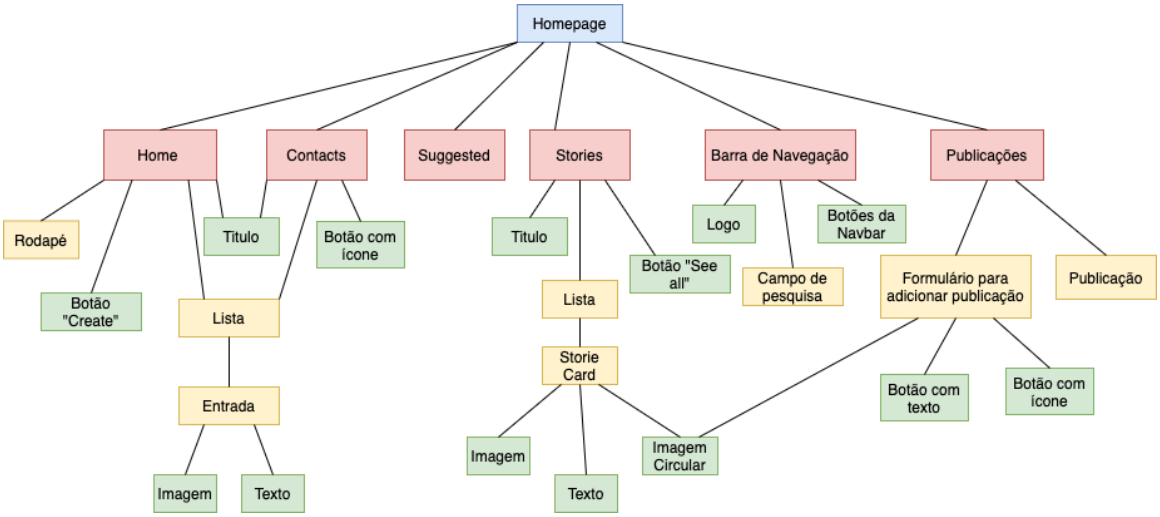


Figura 5: Árvore de componentes da homepage do Facebook

Figuras 6 e 7, a utilização de estilo muda drasticamente o aspeto da interface. É possível configurar tipos de letra, tamanhos, espaçamentos, alinhamentos, cores e muitas outras propriedades que dão uma roupagem diferente aos elementos tornando as interfaces mais atrativas, e interativas.

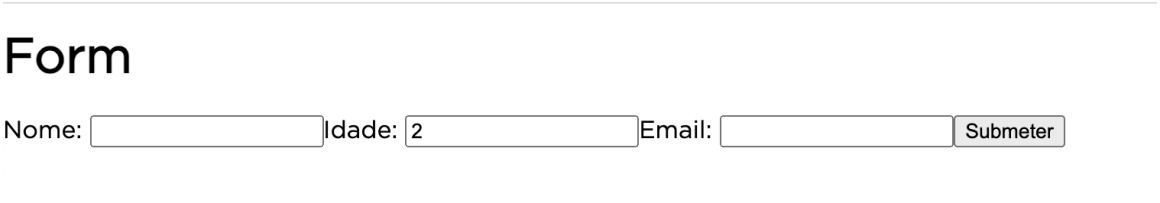


Figura 6: Formulário sem utilizar estilo

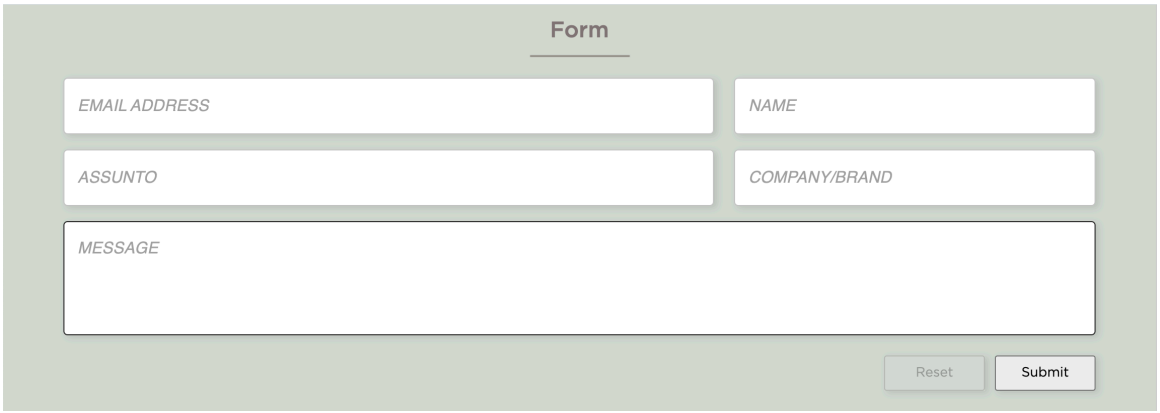


Figura 7: Formulário com o uso de estilo

Existem algumas formas diferentes de aplicar estilo a componentes. Nesta secção iremos abordá-las explicando o que são *CSS*, *Inline-CSS*, *CSS in JS*, *CSS modules*, e *CSS preprocessors*.

2.5.1 CSS

CSS ou *Cascading Style Sheets* é uma linguagem de estilo lançada em 17 de dezembro de 1996, e a sua especificação é atualmente mantida pela W3C[5]. O objetivo desta linguagem é separar o conteúdo da página das suas propriedades de estilo, como cores, tipos de letra, layouts, etc.

Para escrever regras em *CSS* são usados *selectors* que identificam um ou mais elementos. São então definidas propriedades que irão ser aplicadas a todos os elementos que façam *match* com o *selector* descrito. Considere-se, por exemplo, a Listagem 2.2.

```
1 h1 {
2     background-color: white;
3     font-size: 16px;
4     color: red;
5 }
```

Listagem 2.2: CSS regra exemplo

Com o bloco de *CSS* estamos a dizer que todos os elementos identificados pelo *selector* "h1", que se refere a todos os elementos com a etiqueta "h1", irão possuir uma cor de fundo branca, um tamanho de letra de 16px, e a cor vermelha.

O nome *Cascading* advém da regra de prioridade, "em cascata", usada no caso de existir mais do que uma regra aplicável ao mesmo elemento. Considere-se o cenário apresentado na Listagem 2.3.

```
1  <style>
2  h1 {
3    font-size: 16px;
4  }
5
6  h1 {
7    font-size: 22px;
8  }
9  </style>
10
11 <div>
12   <h1>Hello World</h1>
13 </div>
```

Listagem 2.3: CSS exemplo de colisão entre regras

O tamanho de letra que irá ser usado no elemento **h1** é 22px porque esta é a última regra a ser definida. Para além deste critério existem ainda outros como a origem da regra, isto é, caso a regra seja escrita *inline* tem prioridade, sobre regras escritas em ficheiros separados, que por sua vez tem prioridade sobre regras importadas de outras bibliotecas. Também a especificidade do seletor tem influência na prioridade das regras. Seletores mais específicos têm prioridade sobre outros, por exemplo, o seletor `.title > h1`[6]

[5] https://www.w3.org/TR/css-2020/ - visitado em 20/12/2021

[6] O caractere "`>`" faz referência aos descendentes diretos de um elemento

(A) é mais específico e mais prioritário do que o `h1` (B), logo para um elemento `h1` da classe "title" vai ser aplicada a regra A com prioridade sobre a regra B. No site da W3C[7] encontra-se especificado em maior pormenor este algoritmo.

2.5.2 Inline-CSS

Habitualmente as regras de *CSS* são escritas em ficheiro separados, carregados para a página HTML. Ou nas etiquetas <style></style> no próprio ficheiro HTML, sendo esta segunda forma menos comum. Também é possível definir regras de *CSS* na declaração do próprio elemento, através de *Inline-CSS*.

Usando a propriedade *style* de um elemento HTML podemos especificar o estilo desse mesmo elemento.

```
1  <h1 style="color:blue;">Hello World</h1>
```

Listagem 2.4: Inline CSS

No exemplo da Listagem 2.4 está a ser especificada a cor do texto "Hello World" através de *Inline-CSS*. Estas regras, como foi referido anteriormente, assumem uma prioridade alta no algoritmo do *CSS* sobrepondo-se a qualquer regra especificada num ficheiro à parte, a menos que essa regra leve a etiqueta "!important!".

Esta abordagem por mais simples e prática que seja não apresenta vantagens a um nível mais alto. Ao usar *Inline-CSS* não só repetimos escusadamente enumeras regras de *CSS*, mas também dificultamos imenso o trabalho futuro de alterar transversalmente uma propriedade, já que ela estará espalhada por diferentes elementos, ao invés de estar centralizada numa só classe. Por exemplo, usar *Inline-CSS* para por um título com a cor vermelha em todos os elementos que representassem títulos na nossa página, ao invés de criar uma classe título com a propriedade cor vermelha e fazer com que todos os títulos da página usassem essa classe, irá levar a uma repetição desnecessária de regras de CSS. E se quiséssemos, mais tarde, alterar a cor dos títulos para verde, teríamos de fazer a alteração em todos os elementos, quando seria muito mais fácil modificar uma única regra de *CSS* referente a uma classe que todos usassem.

2.5.3 CSS preprocessors

Os *CSS preprocessors* são ferramentas que geram *CSS* a partir de uma sintaxe própria, mais simples. Estes pré-processadores existem porque adicionam funcionalidades ao *CSS* original, como variáveis, seletores aninhados, entre outras. Atualmente Sass[8], Less[9], e Stylus[10] são alguns dos mais populares e mais usados.

[7] https://www.w3.org/TR/WD-CSS2-971104/cascade.html - visitado em 20/12/2021
[8] https://sass-lang.com/ - visitado em 20/12/2021
[9] https://lesscss.org/ - visitado em 20/12/2021
[10] https://stylus-lang.com/ - visitado em 20/12/2021

Para além de introduzirem novas funcionalidades, por possuírem a sua própria sintaxe e terem a possibilidade de utilizar seletores aninhados, os pré-processadores conseguem remover algum açúcar sintático da linguagem *CSS* original tornando o código mais legível e a sua manutenção mais fácil.

```
//CSS
.header .header-nav {
  display: flex;
  flex-direction: column;
}
.header .header-nav .col li {
  background-color: red;
  font-size: 16px;
}

.title {
  color: red;
}

//Sass

.header
  .header-nav
    display: flex
    flex-direction: column

    .col
      li
        background-color: $my-imported-red
        font-size: 16px

.title
  color: $my-imported-red
```

Listagem 2.5: CSS vs Sass

A Listagem 2.5 apresenta a diferença entre a mesma regra em *CSS* e usando o pré-processador Sass. Como podemos observar, com o uso do Sass o estilo ficou bem mais legível, e sem açúcar sintático desnecessário. Além disso, conseguimos usar variáveis para coisas como cores e garantir ser usado exatamente o mesmo vermelho em todos os sítios supostos.

Apesar destes benefícios, o uso de pré-processadores não fornece uma solução para o problema dos conflitos de regras. Isto é, quando um projeto cresce, cresce também a probabilidade de criarmos uma regra que sem querer entra em conflito com outra já existente. Se a regra que criamos tiver prioridade sobre a que já existe, podemos estar a alterar o aspeto da interface sem saber. Para combater este problema existem metodologias de nomenclatura de classes para evitar conflitos. Algumas das mais

conhecidas são o BEM[11], OOCSS[12], e o SMACSS[13]. Estas metodologias permitem-nos de uma forma consistente escolher os nomes das nossas classes *CSS* bem como organizar da melhor forma todas as nossas regras de estilo. Contudo, não deixa de continuar a ser um processo manual, pelo que o aparecimento de erros e conflitos nunca poderá ser totalmente erradicado.

2.5.4 CSS modules

Os *CSS modules* são uma abordagem que surgiu especificamente para mitigar o problema do "scope" global que as regras de *CSS* possuem. Projetos de grande escala usam muitos ficheiros de estilo com variadíssimas classes, e importam também muitos estilos externos. Neste cenário a probabilidade de haver colisões nas regras de estilo é alta, e pode levar a erros difíceis de detetar e de corrigir. Por exemplo, se decidimos importar uma biblioteca externa com o seu próprio estilo, esse estilo pode possuir regras para classes que nós já usamos no código e causar conflitos que alteram o aspeto da restante aplicação sem o nosso consentimento. Com os *CSS modules* o "scope"de cada ficheiro de estilo está limitado ao próprio elemento. Isto é feito através da geração de um nome de classe único quando o código é compilado.

```
1 import React, { Component } from 'react';
2 import styles from './Button.module.css'; // Css module (Compiliado pelo
     react e carregado para o browser)
3 import './another-stylesheet.css'; // Regular CSS (Carregado globalmente
     para o browser)
4 class Button extends Component {
5   render() {
6     // reference as a js object
7     return <button className={styles.error}>Error Button</button>;
8   }
9 }
```

Listagem 2.6: CSS Modules - Exemplo

Em *React* é muito fácil de usar esta metodologia, vejamos o exemplo na Listagem 2.6. Quando o site for renderizado o resultado será:

```
<button class="Button_error_ax7yz">Error Button</button>
```

A classe do elemento *button* é única agora e não há hipótese de colisão com outras regras de estilo que estejam a ser carregadas na aplicação. Isto porque o *React* compilou o CSS como um módulo, e gerou um nome de classe único para o elemento HTML, que apenas corresponde com as regras de CSS do módulo importado.

[11] http://getbem.com/introduction/ - visitado em 20/12/2021
[12] https://www.keycdn.com/blog/oocss - visitado em 20/12/2021
[13] http://smacss.com/ - visitado em 20/12/2021

2.5.5 CSS in JS

CSS in JS como o próprio nome indica é um método de escrever *CSS* usando a linguagem Javascript. Isto é possível recorrendo a bibliotecas, como Styled-Components[14], JSS[15], ou Emotion[16]. Estas bibliotecas compilam o Javascript em regras de *CSS* para os *browsers* interpretarem.

```
1
2  import React from 'react';
3  import injectSheet from 'react-jss';
4
5  const styles = {
6    wrapper: {
7      display: 'flex',
8      flexDirection: 'column',
9      alignItems: 'center',
10     justifyContent: 'center',
11     width: '100%',
12     padding: '50px',
13     color: '#444',
14     border: '1px solid #1890ff',
15   },
16   title: {
17     color: '#0d1a26',
18     fontWeight: 400,
19   }
20   }
21 };
22
23
24 const ExampleJss = ({ classes }) => (
25   <div className={classes.wrapper}>
26     <h1 className={classes.title}>Example JSS.</h1>
27   </div>
28 );
29
30 export default injectSheet(styles)(ExampleJss)
```

Listagem 2.7: CSS in JS - Exemplo

Vejamos o exemplo da Listagem 2.7 que utiliza a biblioteca JSS. Neste exemplo vemos que os estilos estão definidos pela variável `styles`, e que estes estilos são injetados no componente `ExampleJss` pela função `injectSheet` da biblioteca `react-jss`. Os vários elementos podem importar apenas

[14] https://styled-components.com/ - visitado em 20/12/2021
[15] https://cssinjs.org - visitado em 20/12/2021
[16] https://emotion.sh/docs/introduction - visitado em 20/12/2021

sub-partes dos estilos como é feito nas linhas 25 e 26, onde são importados os estilos do "wrapper" e do "title" respetivamente.

O *CSS in JS* é uma abordagem que surge para combater algumas das fraquezas do *CSS*. A primeira é o "scope" global do algoritmo *cascade*. Quando definimos uma regra em *CSS* esta não tem um conceito modular, tenta sempre fazer *match* com todos os elementos da página. Como já foi explicado, isto pode levar a conflitos quando possuímos muitos elementos e muitas regras. Com a abordagem *CSS in JS*, tal como com os *CSS modules*, cada estilo é apenas aplicado ao componente ao qual está associado. Isto é conseguido porque as bibliotecas geram nomes únicos para as classes de cada componentes, assim é impossível haver duas regras diferentes sobre a mesma classe, erradicando os conflitos. No entanto, este método apresenta maior *build-time* devido ao tempo de compilação dos estilos, e pode tornar-se ligeiramente mais difícil fazer *debugging* devido aos nomes das classes serem aleatórios.

Outra das vantagens em relação às outras abordagens tem de ver com o facto de estarmos a usar Javascript para definir o estilo dos nossos componentes, isto permite-nos usar o poder desta linguagem e fazer depender os nossos estilos de variáveis globais importadas, do estado interno do componente, ou de qualquer outro fator programável. Isto facilita muito o conceito de aplicações com vários temas, como, por exemplo, o agora famoso modo noite em diversas aplicações conhecidas.

O facto de ser uma biblioteca a gerar o estilo através do código Javascript faz com que as bibliotecas tratem da compatibilidade das regras para *browsers* diferentes e para versões diferentes. Assim, só nos temos de preocupar em escrever as regras na forma suportada pela biblioteca que depois é gerado o *CSS* para todos os casos.

Em termos de manutenção esta abordagem também parece ser mais eficaz, pois cada estilo está associado apenas a um componente, no mesmo ficheiro ou em ficheiros "vizinhos", ao invés de estar num ficheiro com várias regras diferentes para elementos diferentes, o que pode levar a erros quando queremos, por exemplo, alterar a cor de um componente e, sem querer, alteramos a cor de vários componentes do site.

2.5.6 O que devemos usar?

Mediante tantas opções é mais fácil começar por excluir aquelas que não devemos utilizar. O *Inline CSS* por estar espalhado por todo o projeto, e ser dificílimo de manter não deve ser uma opção para fornecer estilo aos nossos componentes.

O *CSS*, por agora, também perde para os pré-processadores. Apesar de já ter adotado algumas das funcionalidades que estes fornecem, como as variáveis globais, ainda não possui tantas ferramentas como estes. Contudo, no futuro este problema pode ser mitigado e poderemos deixar de precisar de pré-processadores. Por exemplo, neste momento existe já uma especificação para uma implementação de seletores aninhados em *CSS*[17]. Para além desta, outras funcionalidades podem começar a ser adotadas pelo *CSS* original e a necessidade de utilizar pré-processadores deixa de existir.

[17] https://drafts.csswg.org/css-nesting-1/ - visitado em 20/12/2021

Caso o projeto em questão seja de grande dimensão, possua vários componentes, ou dependa de vários componentes de bibliotecas externas, devemos tentar evitar as colisões de *CSS*. Para isso temos duas hipóteses. Podemos usar *CSS in JS*, ou usar pré-processadores combinados com *CSS modules* ou com metodologias de nomenclatura como o BEM, sendo que esta ultima não é 100% eficaz porque continua a ser um processo manual de evitar colisões.

Entre *CSS in JS* ou pré-processadores + *CSS modules* a decisão já é muito mais difícil de tomar. Por um lado, com pré-processadores + *CSS modules* podemos reciclar algumas das nossas regras de estilo, utilizando a mesma classe em componentes diferentes. Por outro lado, se queremos orientar verdadeiramente a nossa aplicação ao componente e reutilizá-los, em vez de reutilizar regras de estilo, *CSS in JS* é a melhor opção, visto que agregamos o estilo ao componente e não utilizamos o mesmo estilo em componentes totalmente diferentes. Além disso, *CSS in JS* torna-se muito mais dinâmico em função do estado do componente, ou da aplicação, devido ao uso do Javascript. No entanto, a curva de aprendizagem para esta abordagem também deve ser tida em conta, visto que não é trivial.

Visto que toda a nossa arquitetura geral, divisão do *design*, e estrutura do projeto se baseia em componentes, o mais apropriado parece ser usar *CSS in JS* para acoplar o estilo ao componente e fazer com que este seja reutilizável da mesma forma em qualquer parte da aplicação ou entre diferentes aplicações.

2.6 Sumário

Neste capítulo foram abordados diversos tópicos referentes à implementação de interfaces web. Na Secção 2.1 foram expostas diferentes abordagens de implementação.

Na Secção 2.2, aprofundou-se a temática das *frameworks* por serem, atualmente, o recurso mais utilizado na produção de interfaces. Compararam-se as três *frameworks* mais utilizadas, *React*, *Angular*, e *Vue*.

A seguir, na Secção 2.3, discutiu-se o desenvolvimento orientado ao componente, e a forma como este é aplicado na implementação de interfaces.

Na Secção 2.4 foi apresentado um processo de interpretação do *design* que visa a divisão do mesmo em componentes, uns mais simples, e outros mais complexos, os últimos construidos a partir da composição dos primeiros.

Por último em na Secção 2.5, abordaram-se diferentes formas de adicionar estilo a uma interface web, mais concretamente, CSS, *Inline-CSS*, *CSS preprocessors* e *CSS in JS*. Foram mencionados os pontos fortes e as lacunas existentes de cada abordagem.

3

Análise de uma Aplicação em React

Neste capítulo, iremos propor uma arquitetura genérica para as aplicações *React*, com base no conhecimento teórico sobre a *framework* e na análise de vários projetos de software que a utilizam. Para além desta arquitetura, será descrita uma estrutura para serem organizados todos os ficheiros que compõem uma interface em *React*, para ser possível manter o projeto, mesmo com o aumento do número de componentes e ficheiros.

3.1 Arquitetura Genérica de um aplicação em React

3.1.1 Projetos existentes

Existem neste momento imensas aplicações escritas em *React*. Muitas delas possuem até o seu código público, em repositórios online, onde qualquer pessoa pode consultar o que constitui realmente determinada aplicação web. É possível identificar, ao nível arquitetural, pontos em comum entre as mais variadíssimas aplicações por mais distintas que sejam as áreas onde estão inseridas.

Posto isto, visando chegar a uma possível arquitetura genérica de uma aplicação em *React*, foram analisados diversos projetos de código aberto, com tamanho e reputação consideráveis. Para esta análise utilizamos o GitHub, plataforma que serve de repositório para uma enorme quantidade de produtos de software. Esta plataforma possui um sistema de classificações, chamado GitHub *stars*, onde cada utilizador pode dar uma GitHub *star* a repositórios que considere interessantes. Assim, escolhemos projetos que usassem *React* e fossem populares, ou seja, que tivessem uma quantidade de GitHub *stars* aceitável, acima de 5000 mil.

Exemplos usados

1. **DevHub** - https://github.com/devhubapp/devhub - Aplicação desktop e mobile de gestão de notificações e atividade no GitHub. 7300 GitHub stars

2. **Mattermost** - https://github.com/mattermost/mattermost-server - Aplicação desktop e mobile de chat, alternativa ao Slack. 19900 GitHub stars

3. **Real World** - https://github.com/gothinkster/realworld - Aplicações exemplo utilizando diferentes *stacks* de desenvolvimento. 54900 GitHub stars

4. **UpTerm** - https://github.com/railsware/upterm - Aplicação para um terminal linux. 19500 GitHub stars

5. **Nylas Mail** - https://github.com/nylas/nylas-mail - Aplicação de mail desktop. 24600 GitHub stars

6. **CodeSandbox** - https://github.com/codesandbox/codesandbox-client - Aplicação web para suportar desenvolvimento isolado em cloud - 10300 GitHub stars

Após analisar os projetos selecionados conseguimos identificar práticas que se repetem, e entidades que estão presentes na maioria dos repositórios. Todos os projetos analisados possuem componentes *React*, que possuem o seu próprio estilo, herdado de ficheiros separados, ou descrito no próprio componente.

Para além da presença de componentes isolados e com o seu próprio estilo, outras práticas foram encontradas em alguns projetos, como o uso do *React router* ou de serviços. Estes são apenas usados em alguns dos projetos, como a *codesandbox-client* ou o *nylas-mail*, porque dependem do propósito da aplicação.

Tendo em conta os aspetos em comum retirados dos projetos analisados e o conhecimento geral da *framework*, elaborou-se uma possível arquitetura genérica de uma aplicação *React* visível na Figura 8. Alguns dos projetos podem não estar mapeados a 100% nesta arquitetura, mas seria possível convertê-los.

3.1.2 Arquitetura genérica

Nesta esta secção iremos explicar a arquitetura proposta, e os seus diferentes componentes, com base nas aplicações analisadas. A representação da arquitetura pode ser consultada na Figura 8. Como ponto de partida temos a entidade **App**, que encapsula todos os elementos da aplicação. Esta entidade dá normalmente origem a um ficheiro, de nome *App.js*. Aqui são importadas bibliotecas genéricas que dão suporte ao projeto. É muito comum o uso do **Router**, uma biblioteca *React* que permite declarar páginas, associá-las a componentes, e mapear estas páginas com rotas *web*.

Na aplicação *web Mattermost* vemos no ficheiro *App.jsx*[1] (Listagem 3.1) a estrutura descrita no anterior parágrafo. Neste ficheiro, é declarado o componente de entrada para a aplicação e dentro deste é usado o *Router* para definir as diferentes páginas do produto, associadas a rotas como "/login", "/register", "/settings", etc (linhas 20 a 29).

[1] https://github.com/mattermost/mattermost-webapp/blob/master/components/app.jsx - visitado em 20/12/2021

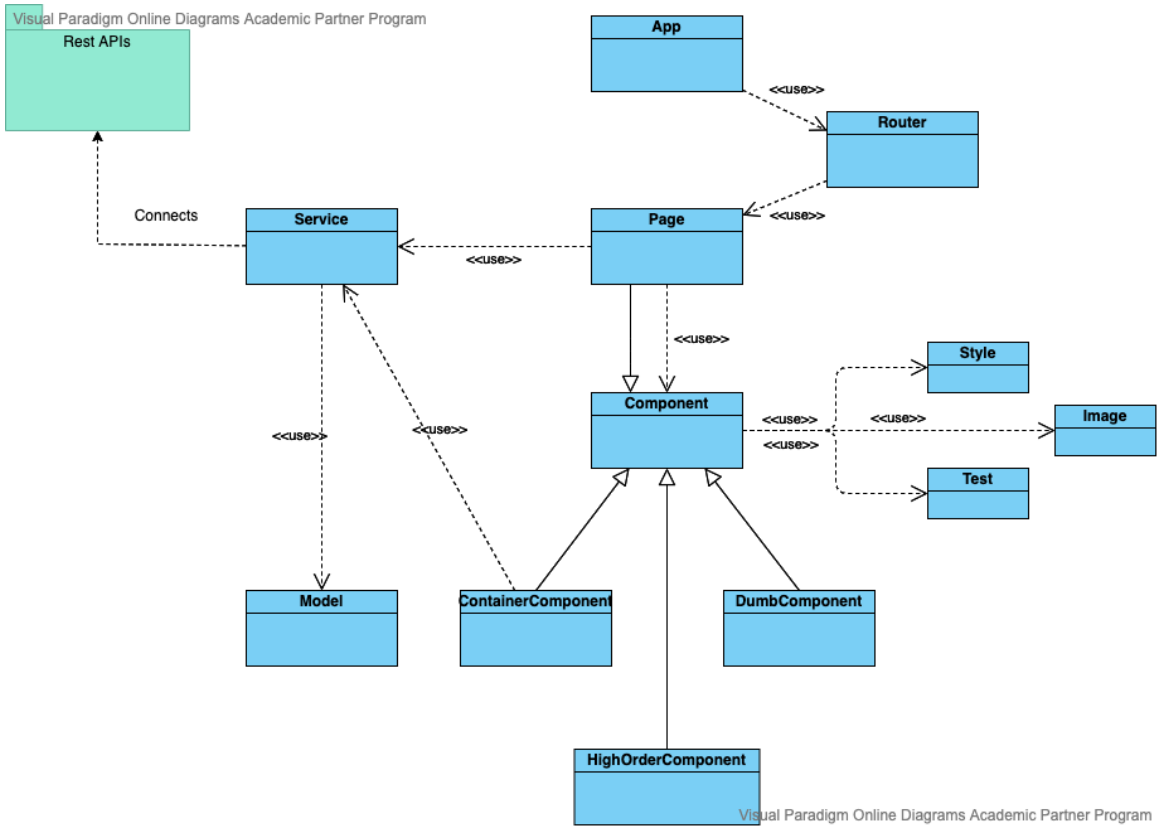


Figura 8: Arquitetura genérica de uma aplicação em *React*

```
import Header from "./components/Header";
import React from "react";
import { Switch, Route, withRouter } from "react-router-dom";
import { inject, observer } from "mobx-react";
import PrivateRoute from "components/PrivateRoute";

import Login from "pages/Login";
import Home from "pages/Home";
import Register from "pages/Register";
(...)

export default class App extends React.Component {
  (...)

  render() {
    if (this.props.commonStore.appLoaded) {
      return (
        <div>
          <Header />
          <Switch>
            <Route path="/login" component={Login} />
```

```
            <Route path="/register" component={Register} />
            <Route path="/editor/:slug?" component={Editor} />
            <Route path="/article/:id" component={Article} />
            <PrivateRoute path="/settings" component={Settings} />
            <Route path="/@:username" component={Profile} />
            <Route path="/@:username/favorites" component={Profile} />
            <Route path="/" component={Home} />
          </Switch>
        </div>
      );
    }
    return <Header />;
  }
}
```

Listagem 3.1: Ficheiro App.js da aplicação *Mattermost*

Cada página é encapsulada num componente, o que dá origem a outro elemento na nossa arquitetura, as **Pages**. Estas são componentes que definem uma determinada página do produto. Por exemplo, para uma página de *login* temos o componente *Login*, no ficheiro *Login.js*, onde irão ser importados todos os componentes necessários para a implementar.

Como mencionado anteriormente uma página web pode ser constituída por diversos componentes, dando origem a outro elemento na nossa arquitetura, os **Components**. Estes definem cada parte do produto, desde peças mais granulares como botões, campos de texto ou formulários, até às partes mais complexas, como um cartão de apresentação, ou uma lista de resultados. É possível algumas especializações dentro desta categoria, sendo o conceito de **ContainerComponent** uma delas. Este tipo de componentes diferencia-se por possuírem lógica de negócio no seu comportamento. Habitualmente é nestes componentes que se carregam dados de *APIs*, e caso necessário faz-se algum tipo de tratamento destes dados para, a seguir, transmiti-los ao componente responsável por renderizar a vista dos mesmos.

Consideremos o exemplo na Listagem 3.2.

```
import { EventColumn } from '../components/columns/EventColumn'
import { IssueOrPullRequestColumn } from '../components/columns/
    IssueOrPullRequestColumn'
import { NotificationColumn } from '../components/columns/
    NotificationColumn'
(...)

export const ColumnContainer = React.memo((props: ColumnContainerProps) =>
    {
  (...)

  switch (column.type) {
    case 'activity': {
```

```
11        return (
12          <EventColumn
13            (...)
14          />
15        )
16      }
17      case 'issue_or_pr': {
18        return (
19          <IssueOrPullRequestColumn
20            (...)
21          />
22        )
23      }
24      case 'notifications': {
25        return (
26          <NotificationColumn
27            (...)
28          />
29        )
30      }
31  (...)
```

Listagem 3.2: Componente ColumnContainer da aplicação *DevHub*

O componente *ColumnContainer*[2] da aplicação *DevHub* é responsável por, mediante o tipo de coluna recebida, decidir qual o componente responsável por renderizar a informação. Este componente apenas trata da lógica de saber qual o tipo de coluna que irá ser apresentada e passa a apresentação da mesma ao componente responsável.

Vemos então uma clara diferenciação entre componentes que tratam de carregar dados ou de fazer operações no âmbito da lógica de negócio da aplicação, e aqueles que apenas mostram dados no ecrã. Os últimos podemos categorizar de **DumbComponents**, cujo papel é renderizar determinada vista com base no *input* recebido. Exemplos destes são os botões, os *input fields*, e quaisquer outros que não possuam lógica de negócio. Se quisermos especificar ainda mais esta categoria, podemos subdividir estes **DumbComponents** em *AtomComponents*, *MolecularComponents*, e *OrganismComponents*, derivados do *Atomic design* especificado na Secção 2.4. Contudo, do ponto de vista de estrutura estes acabam por ser todos iguais, sendo os componentes mais complexos apenas composições dos componentes mais simples.

Todos os componentes podem importar ficheiros de estilo para definir regras de apresentação, surgindo assim a categoria **Style**, ficheiros com regras sobre o estilo de um ou mais componentes. É também comum as *web apps* possuirem as suas próprias **Images** utilizadas pelos componentes que constituem a aplicação.

[2] https://github.com/devhubapp/devhub/blob/master/packages/components/src/containers/ColumnContainer.tsx - visitado em 20/12/2021

Infelizmente menos comum é a existência de testes unitários aos componentes. Tal como as diferentes rotas das *APIs* são testadas para descobrir eventuais erros, também os componentes do *frontend* devem possuir os seus próprios ficheiros de teste (**Test**). Isto permite aumentar a confiança na aplicação e descobrir possíveis erros muito mais facilmente. Dos projetos que analisámos apenas 2 possuíam testes aos componentes, um bom exemplo é o *Mattermost* que possui testes para a maioria dos seus componentes, por exemplo, no ficheiro *admin_user_card.test.jsx*[3] é possivel ver um conjunto de testes ao componente *AdminUserCard*.

Existe ainda um último tipo de componentes, os **HighOrderComponents**, a função destes é encapsular funcionalidades comuns a vários componentes de modo a serem reutilizadas. Apresenta bastantes semelhanças com o padrão *Decorator* (Gamma et al., 1994) na medida em que acrescenta comportamento a componentes já existentes. Um exemplo é o componente *Route* da biblioteca *react-router*, este componente é utilizado para atribuir o comportamento de uma rota web a outro determinado componente.

```
1 (...)
2 <Route path="/dashboard" component={Dashboard} />
3 <Route path="/new-dashboard" component={Dashboard} />
4 <Route path="/curator" component={Curator} />
5 (...)
```

Listagem 3.3: Ficheiro index.tsx da aplicação *CodeSandbox*

No ficheiro *index.tsx*[4] (Listagem 3.3) do projeto *CodeSandbox*, podemos ver que o mesmo **HighOrderComponent**, o componente *Route*, é utilizado para criar as diferentes páginas associadas a determinadas rotas. As páginas são posteriormente definidas pelo seu próprio componente, por exemplo, o componente *Dashboard* ou *Curator*.

Por último, na maioria das aplicações web não estáticas é necessário interagir com APIs ou serviços externos à aplicação. É boa prática separar as lógicas e interações com outros sistemas da implementação da UI. Surgem assim os **Services**, módulos com lógica externa à interface, como, por exemplo, métodos para consumir uma API, ou métodos que interagem com um serviço de geolocalização. Para além destes, qualquer lógica, relativamente complexa, externa à UI deve estar encapsulada e isolada num serviço. Serviço este que irá ser consumido pelos **ContainerComponents** ou pelas **Pages**. Por exemplo, em *UpdatesService.ts*[5], no projeto *UpTerm*, é descrito um serviço que interage com a API do GitHub através de HTTP.

Quando existe a necessidade de representar os dados obtidos pelos serviços devemos usar os **Models**. Estes são apenas classes que representam entidades no nosso projeto. Por exemplo, se obtivermos uma rota que nos fornece os dados de um aluno via API, podemos mapear a informação do aluno num **Model** e adicionar comportamento extra a esse modelo de dados caso seja necessário.

[3] https://github.com/mattermost/mattermost-webapp/blob/master/components/admin_console/admin_user_card/admin_user_card.test.jsx - visitado em 20/12/2021

[4] https://github.com/codesandbox/codesandbox-client/blob/c8d6fc8a8739a1662bce963ac0746b64621ae820/packages/app/src/app/pages/index.tsx - visitado em 20/12/2021

[5] https://github.com/railsware/upterm/blob/master/src/services/UpdatesService.ts - visitado em 20/12/2021

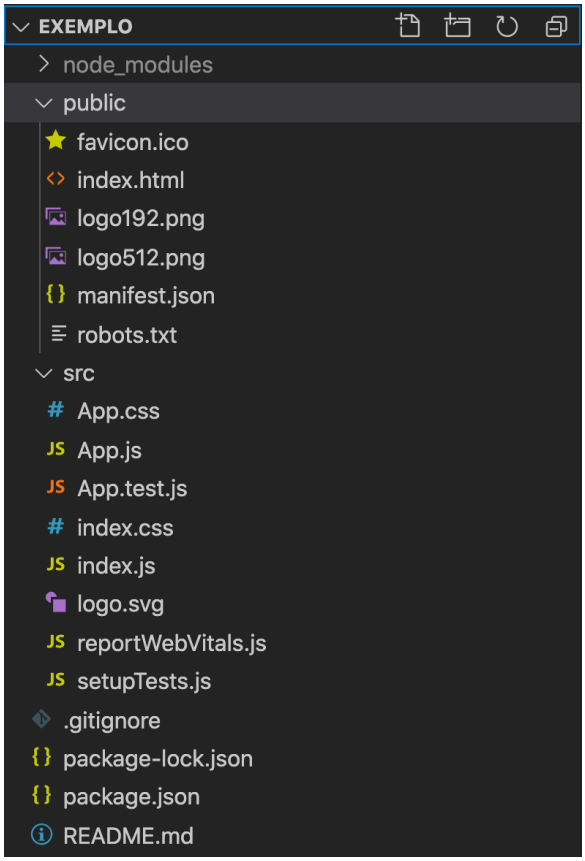


Figura 9: Estrutura de ficheiros do create react app

3.2 Estrutura dos ficheiros

Após chegar a uma arquitetura genérica é importante também definir uma estrutura viável para organizar os ficheiros que compõem a aplicação. Com o passar do tempo os repositórios tendem a aumentar, e se não possuirmos um projeto organizado, sabendo exatamente onde está a implementação de determinado componente, torna-se muito difícil manter o produto e desenvolver novas funcionalidades ou alterações.

Quando criamos um novo projeto *React*, usando a *create react app*[6], é gerada a estrutura de pastas e ficheiros apresentada na Figura 9. Possuímos então 2 pastas, a pasta *public* e a pasta *src*. Na pasta *public* estão as imagens que vão servir de ícones para a aplicação, e o ficheiro de HTML inicial com a informação pública da página, como titulo ou descrição. É nesta página que a *framework React* irá injetar o seu HTML após cada renderização. Dentro da pasta *src* está o código que compõe o produto. Neste caso temos o ficheiro *index.js* responsável por injetar o código *React* na página web definida no ficheiro *index.html*. Temos apenas um componente, descrito no ficheiro *App.js*, que representa toda a aplicação neste momento. Por último temos ainda ficheiros de estilo e de testes, que são utilizados na aplicação.

Há medida que o tamanho da aplicação aumenta fica impraticável ter todos os componentes e os seus ficheiros de estilo na raiz da pasta *src*. Dab Abramov programador na empresa *Facebook* afirma

[6] https://github.com/facebook/create-react-app - visitado em 20/12/2021

que a rede social possui mais de 30 000 componentes *React*[7]. Para conseguir gerir uma aplicação desta escala é necessário organizar os projetos melhorando esta estrutura inicial.

A mudança mais evidente, e que está mais presente nos projetos analisados, é a criação de uma pasta *components*, onde se colocam todos os componentes da aplicação. Ficando na raiz da pasta *src* apenas os ficheiros *index* e *App* que servem como *facade* para os restantes.

Como vimos na secção anterior existem vários tipos de componentes, *Pages*, *ContainerComponents*, *HighOrderComponents*, e *DumbComponents*. Habitualmente vemos serem criadas subpastas para as páginas e para os *ContainerComponents* da aplicação, surgem então as pastas `/pages`, e `/containers` na raiz da pasta `/src`. Os restantes componentes ficam pendurados na raiz da pasta *components*. Serviços e modelos, também devem ficar isolados nas suas próprias pastas, `/services` e `/models` respetivamente.

As imagens utilizadas pelos componentes ficam guardadas na pasta `/assets/images`. Dentro desta pasta podem ainda ser agrupadas, por tipo, as imagens usadas um pouco por toda a aplicação. Podemos, por exemplo, agrupar os ícones todos na pasta `/assets/images/icons`, e os logótipos na pasta `/assets/images/logos`. As restantes imagens podem ser agrupadas pela página onde são utilizadas, assim a pasta `/assets/images/homepage` teria todas as imagens utilizadas na *homepage*. Outros recursos estáticos como, por exemplo, as fontes de texto também devem ser agrupadas e pendurados nesta pasta `/assets`.

Os ficheiros de estilo são, após os ficheiros de código, aqueles que existem em maior quantidade no projeto. Para os organizar de forma a facilitar a constante manutenção de uma aplicação costuma-se seguir uma de duas abordagens. Uma abordagem consiste em separar os ficheiros pelo seu tipo, organizando-os em pastas. A outra é uma abordagem mais desacoplada, agrupando todos os ficheiros que definem determinado componente no mesmo local.

Na primeira abordagem, opta-se por criar uma pasta com o nome de `stylesheets` na raiz da pasta `/assets`, para guardar todos os ficheiros de estilo. São criadas sub-pastas, tal como é feito para as imagens, onde cada sub-pasta representa uma página e guarda todos os ficheiros de estilo para os componentes utilizados naquela página. Mesmo dividindo por página continuam a existir uma quantidade considerável de ficheiros de estilo por isso, é necessário muitas vezes fazer uma separação ainda mais granular ao nível do componente. Podemos então acabar com uma estrutura do género de `/assets/stylesheets/homepage/navbar/navbar_burguer_button.css`, o que pode não ser benéfico visto que ficamos com uma estrutura bastante aninhada de ficheiros, dificultando a manutenção dos mesmos. Também poderíamos ter apenas um único ficheiro *homepage.css* que definissem as propriedades de todos os componentes da *homepage*, mas nesse caso estaríamos a acoplar demasiado as regras de estilo o que prejudicaria a reutilização de componentes.

Em contrapartida, na segunda abordagem surge o conceito de *self-contained components*, componentes independentes e desacoplados, onde todos os ficheiros que definem determinado componente

[7] https://github.com/facebook/react/issues/9463#issuecomment-295643228 - visitado em 20/12/2021

estão na mesma pasta. Posto isto, tudo o que diz respeito a um componente está concentrado num só local o que facilita a realização de alterações, e faz com que exista também uma muito menor probabilidade de alterar indesejadamente outra parte da aplicação. Aplicando este raciocínio aos ficheiros de estilo e de código, estes passariam a coabitar na pasta do seu respetivo componente. Por exemplo, na pasta `/components/Navbar`, teríamos o ficheiro `Navbar.js` e `Navbar.css`.

Para evitar importações extensas, por exemplo, `import './components/Navbar/Navbar.js'` podemos criar um ficheiro *package.json* na pasta do componente, com o seguinte conteúdo:

```
{
  main: 'Navbar.js'
}
```

Este ficheiro está a especificar qual o modulo principal da pasta onde está inserido. Agora podemos importar o componente `Navbar` apenas com o seguinte comando `import './components/Navbar'`.

Por último temos os ficheiros de teste que para seguirem esta lógica de *self-contained components* também devem residir na pasta do componente que pretendem testar.

Ainda assim, é possível recorrer a ficheiros mais genéricos, como ficheiros de estilo com definições de cores, fontes, ou espaçamentos transversais a toda a aplicação. Esses devem ficar na pasta `/stylesheets` pendurada na pasta `/assets`.

Em suma o resultado da estrutura do projeto corresponde ao da Listagem 3.4.

```
1  /public
2  index.html (página principal)
3  ... (informações publicas da página)
4  /src
5    /assets
6      /stylesheets
7        colors.css (ficheiro de estilo com a definição das cores de todo o
             projeto)
8        fonts.css (ficheiro de estilo com a definição das fontes de todo o
             projeto)
9      /images
10       /homepage
11         homepageImg1.png (imagem usada na homepage)
12       /logos
13         logo1.png (logo usado ao longo da aplicação)
14       /icons
15         icon1.png (icon usado ao longo da aplicação)
16   /components
17     /Navbar (Componente que representa uma Navbar)
18       Navbar.js (Ficheiro de código do componente Navbar)
19       navbar.css (Ficheiro de estilo do componente Navbar)
20       Navbar.test.js (Ficheiro de testes do componente Navbar)
```

```
      package.json (Ficheiro que especifica o modulo principal da pasta)
    (...)
  /pages
    /Homepage (Componente que representa a homepage da aplicação)
      Homepage.js (Ficheiro de código do componente Homepage)
      homepage.css (Ficheiro de estilo do componente Homepage)
      Homepage.test.js (Ficheiro de testes do componente Navbar)
    (...)
  /containers
    /NavbarContainer
        NavbarContainer.js (Ficheiro de código do componente container da
           Navbar)
        NavbarContainer.test.js (Ficheiro de testes do componente container
            da Navbar)
    (...)
  /services
    /FetchUserInfoService (Serviço que trata de ir buscar informação do
       utilizador)
      FetchUserInfoSercice.js ((Ficheiro de código do serviço para ir
         buscar informação do utilizador)
  (...)
App.js (Ponto de entrada da aplicação React)
App.test.js (Ficheiro de testes ao ponto de entrada da aplicação React)
index.js (iniciliza a aplicação React na página publica)
(...)
```

Listagem 3.4: Estrutura de ficheiros geral de um projeto

3.3 Sumário

Neste capítulo, na Secção 3.1, apresentamos o resultado da análise de vários projetos *React*, com repositório público no *github*. O objetivo era identificar as práticas em comum a nível arquitetural e estrutural. Posto isto, foi definida uma arquitetura genérica de uma aplicação *React* (Figura 8). Foi descrita a função de cada entidade pertencente à arquitetura definida, e foram também apresentados exemplos do uso destas entidades nos projetos analisados.

A nível estrutural, na Secção 3.2, discutiram-se diferentes formas de organização de pastas e ficheiros num projeto *React*. A abordagem dos *self-contained components* foi a escolhida por ser a mais modular e permitir uma maior reutilização dos componentes. A estrutura final idealizada pode ser consultada na Listagem 3.4

4

Implementação em React

Neste capítulo iremos abordar mais concretamente a *framework React*, ilustrando como se deve proceder à implementação de interfaces utilizando as melhores práticas e melhores ferramentas da framework. Abordaremos o funcionamento da framework, a implementação de componentes com e sem estado, e o acesso a APIs.

4.1 Como funciona o React

O *React* funciona com recurso a uma virtual DOM para onde são renderizados os elementos de UI com um custo relativamente baixo. Depois esta virtual DOM tem a responsabilidade de encontrar a forma mais eficiente de modificar a DOM original do *browser* para que a página fique como esperado.

React é uma *framework* orientada a componentes, sendo que cada componente em *React* possui o seu ciclo de vida. É importante conhecermos este ciclo para percebermos o fluxo da framework e conseguirmos implementar corretamente uma interface. O ciclo de vida de um componente é constituído por 3 fases, *Mounting*, *Updating*, e *Unmounting*.

Mounting

Esta fase é a primeira de todas, e consequentemente responsável por inicializar o componente, o seu estado e os seus elementos. Aqui, ainda não possuímos acesso à DOM, porque o componente ainda não foi renderizado no ecrã. Depois da sua inicialização o componente é renderizado, e esta fase termina com o componente no ecrã e na DOM do browser.

Updating

Sempre que é passada uma nova propriedade ao componente, ou sempre que este, internamente, altera o seu próprio estado, é feita uma atualização para refletir essas alterações na interface. Tal como na fase de *Mounting*, esta alteração desencadeia um conjunto de funções, o que pode levar o componente a ser novamente renderizado no ecrã com as mais recentes atualizações.

Unmounting

Esta é a última fase do ciclo de vida de um componente, que acontece quando este é removido da DOM. Normalmente aqui são feitas operações para libertar memória que ficará desnecessariamente ocupada.

Estado dos componentes

Para além do seu ciclo de vida outra das propriedades importantes de um componente é o seu estado. Componentes podem ter estado e fazer variar o seu aspeto em função deste.

Um componente pode inicializar o seu próprio estado de forma independente, ou em função de propriedades que lhe são passadas na inicialização. Por exemplo, se temos um componente que representa um quadrado, podemos passar como propriedade o tamanho dos lados desse quadrado no momento em que chamamos o componente (`<Quadrado lado={2} />`). Assim dentro do componente quadrado o seu estado vai depender do valor recebido na propriedade "lado", estado este que vai fazer variar o tamanho do quadrado no ecrã.

4.2 Análise do design e divisão em componentes

Se analisarmos a árvore da Figura 5 (ver página 15), que representa a divisão de componentes da página do *Facebook* presente na Figura 4 (ver página 14), vemos que vários componentes utilizam os mesmos átomos. Por exemplo, tanto o **Formulário para adicionar publicação** como o **Storie Card** possuem uma imagem circular. Analisando esta árvore conseguimos perceber todas estas dependências, e começar a inferir os componentes que teremos de programar para implementar aquele *design*. A Listagem 4.1 representa os componentes necessários para implementar a página da Figura 4.

```
/pages
  /homepage
    Homepage.js (Componente que caracteriza a página)
/components
  /Home
    Home.js (Componente que caracteriza a área com o titulo "Home")
  /Contacts
    Contacts.js (Componente que caracteriza a área dos contactos)
  /Suggested
    Suggested.js (Componente que caracteriza a área com o titulo "Suggested
       ")
  /StoryCard
    StoryCard.js (Componente que caracteriza o cartão que mostra um story)
  /Navbar
    Navbar.js (Componente que caracteriza a barra de navegação do topo)
  /NewPublicationForm
```

```
16      NewPublicationForm.js (Componente que define o formulário para
            adicionar uma publicação)
17    /Publication
18      Publication.js (Componente que caracteriza uma publicação)
19    /CircularImage
20      CircularImage.js (Componente que caracteriza uma imagem circular)
21    /LabelWithImage
22      LabelWithImage.js (Componente que caracteriza um texto com uma imagem à
            esquerda)
```

Listagem 4.1: Lista de componentes a implementar para a *homepage* do *Facebook*

A partir desta análise já conseguimos listar os diferentes *DumbComponents* que vamos ter de implementar. Fazendo a correspondência para o *Atomic design* da Secção 2.4 já vemos nesta lista componentes dos vários tipos: átomos, moléculas e organismos. Por exemplo, possuímos o organismo **Contacts**, representado pelo componente `Contacts.js`, que irá utilizar a molécula **LabelWithImage**, representada pelo componente `LabelWithImage.js`, que por sua vez utiliza o átomo **CircularImage**, representado pelo componente `CircularImage.js`.

Agora que já sabemos quais os componentes necessários para caracterizar a página, devemos analisar o que é conteúdo estático e o que não é. Conteúdo estático é aquele que não vária com o tempo nem com a utilização. Por outro lado, temos o conteúdo dinâmico que vai a variar com o tempo e com o utilizador da aplicação, este tipo de conteúdo normalmente requer interações com uma API para carregar os dados necessários para mostrar ao utilizador. Seguindo a arquitetura genérica da Secção 3.1, estas chamadas à API vão ser feitas em *ContainersComponents* recorrendo a serviços que contêm toda a lógica necessária para esta interação. Os *ContainersComponents* irão utilizar os serviços para obter os dados de APIs e passar estes aos componentes cuja função é apresentar esses dados no ecrã.

Neste exemplo da Figura 4 possuímos os contactos, os *stories*, e as publicações como conteúdo dinâmico, dado que estes variam consoante o utilizador e o tempo. Iremos então precisar dos serviços e *ContainersComponents* presentes na Listagem 4.2.

```
1  /containers
2    /StoriesContainer
3      StoriesContainer.js
4    /PublicationsContainer
5      PublicationsContainer.js
6    /ContactsContainer
7      ContactsContainer.js
8  /services
9    ContactsService.js
10   StoriesService.js
11   PublicationsService.js
```

Listagem 4.2: Lista de serviços e *ContainerComponents* a implementar para a *homepage* do *Facebook*

Estes *ContainerCompontents* irão utilizar os serviços para obter de toda a informação, aplicar qualquer tipo de lógica aos dados ou ao próprio fluxo da aplicação e chamar os *DumbComponents* responsáveis por renderizar os diferentes elementos.

Em suma, fazendo uma análise mais cuidada e pormenorizada do *design* que nos é entregue, é possível dividir este em componentes de diversos tipos, alguns reutilizáveis dentro de outros. Passamos então a saber, mesmo antes de começar a programar, quantos componentes temos de implementar e onde irá ficar cada parte do código. Além disso, conseguimos inferir possíveis chamadas a APIs externas que dão origem a *ContainersComponents*, e serviços.

4.3 Implementação de componentes

4.3.1 Class Componentes

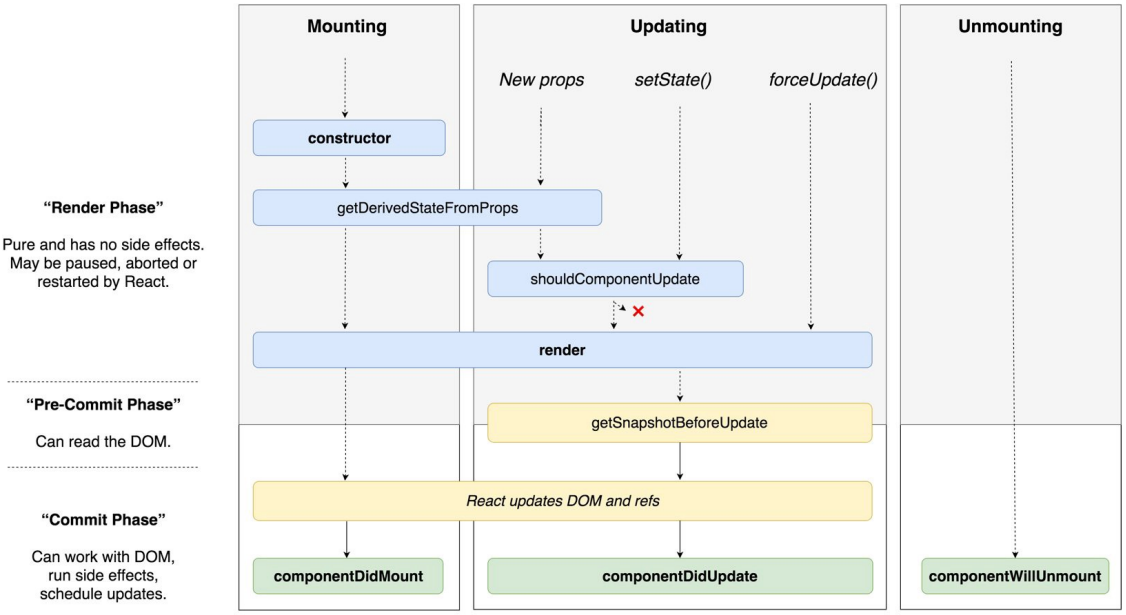


Figura 10: Ciclo de vida de um componente de classe *React*, Fonte: https://dev.to/jaylcaetano/react-component-lifecycle-2npl

A primeira abordagem que surgiu para implementar componentes em *React* foi a utilização da classe `React.Component` implementada pela framework.

Ao estenderem esta classe, os componentes herdam o ciclo de vida de um componente *React* e os seus *lifecycle methods*, presentes na Figura 10.

1. `componenteDidMount()` - Ocorre no fim da fase de "mounting", aqui podemos reagir ao facto de o componente já estar "mounted" e fazer operações, como ir buscar dados a uma API.

2. `componenteDidUpdate()` - Ocorre no fim da fase de "update", o que nos permite reagir à mudança de estado do componente.

3. `componenteWillUnmount()` - Ocorre antes da fase de "unmount", o que nos permite limpar recursos que irão deixar de ser precisos quando o componente for eliminado.

Todos estes métodos são síncronos e executam depois do método `render()` e antes de o componente ser apresentado/removido do ecrã.

Vejamos o componente `LabelWithImage` da homepage do Facebook presente na Figura 11.

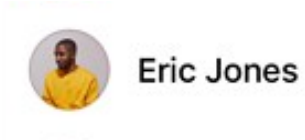


Figura 11: Componente LabelWithImage

Analisando o componente percebemos que tanto o nome como a imagem variam consoante a sua utilização, e não consoante o tempo, por isso estas partes têm de ser dinamicamente derivadas através das propriedades passadas a este componente. Para isso, usamos o método `constructor()` que vai ser responsável por construir o estado do componente. Neste caso, através das propriedades recebidas pelo componente que invoca o `LabelWithImage`. O resultado seria o apresentado na Listagem 4.3 (linhas 1 a 10).

```
1  import React, { Component } from 'react'
2
3  class LabelWithImage extends Component {
4    constructor(props) {
5      super(props);
6      this.state = {
7        labelText: props.labelText,
8        imageUrl:  props.imageUrl
9      };
10   }
11
12   render() {
13     return (
14       <div>
15         <img src={this.props.imageUrl} />
16         <span>{this.props.labelText}</span>
17       </div>
18     )
19   }
20 }
```

Listagem 4.3: *React* - Componente de classe

Agora é preciso renderizar o componente, para isso implementamos a função `render()` retornando os elementos que queremos ver apresentados na DOM. O componente ficaria então finalizado com as linhas 12 a 20 da Listagem 4.3.

Para efeitos de estudo, imaginemos que agora queremos aumentar o tamanho da imagem sempre que alguém clica no nome. Ou seja, o componente agora passa a ser dinâmico ao longo do tempo e precisa de se atualizar a ele próprio. Para isso precisamos de:

1. Atualizar o elemento *span* para que reagir ao evento de click.
2. Atualizar o estado para armazenar o tamanho atual da imagem
3. Criar um método para atualizar o tamanho da imagem no estado

O resultado final seria então o apresentado na Listagem 4.4.

```
1  import React, { Component } from 'react'
2
3  class LabelWithImage extends Component {
4    constructor(props) {
5      super(props);
6      this.state = {
7        labelText: props.labelText,
8        imageUrl:  props.imageUrl,
9        imageSize: 20
10     };
11     this.handleOnClick = this.handleOnClick.bind(this)
12   }
13
14   handleOnClick(){
15     this.setState({imageSize: this.state.imageSize + 1})
16   }
17
18   render() {
19     return (
20       <div>
21         <img width={this.state.imageSize} height={this.state.imageSize} src
               ={this.props.imageUrl} />
22         <span onClick={this.handleOnClick}>{this.props.labelText}</span>
23       </div>
24     )
25   }
26 }
```

Listagem 4.4: *React* - Componente de classe completo com o atualização de estado)

No construtor (linhas 4 a 10) adicionamos ao estado um novo atributo para guardar o tamanho da imagem. De seguida, nas linhas 14 a 16, criamos a função que irá lidar com o clicar do utilizador na imagem. Esta

função, por cada clique, irá atualizar o estado do componente, aumentando em 1 pixel a variável que guarda o tamanho da imagem. Por último no elemento HTML da imagem (linha 21) indicamos que a altura e a largura da imagem dependem do valor do atributo `imageSize` que está guardado no estado do componente.

4.3.2 Functional Componentes

Mais recentemente surgiu uma nova abordagem para a construção de componentes em *React*, os componentes funcionais. Atualmente as aplicações mais modernas como o *Facebook* utilizam esta técnica para compor os seus componentes.

Os componentes funcionais, como o próprio nome indica, são representados através de funções. Cada componente é uma função em Javascript e o valor retornado pela função é o elemento, ou conjunto de elementos, que irão ser renderizados no ecrã.

Por exemplo, para renderizar um simples título com o texto "Hello Joseph", onde o nome é passado como propriedade, fazemos:

```
const HelloWorldTitle = (props) =>
  <h1>Hello {props.name}<h1>
```

Para utilizar o componente fazemos `<HelloWorldTitle name={Joseph} />`. Neste caso as propriedades são recebidas como argumento da função, e podemos utilizá-las na construção dos elementos que vão ser retornados.

Ao contrário dos *Class Components*, os *Functional Componentes* não possuem *lifecycle methods*, para colmatar esta funcionalidade a framework criou os React Hooks.

React Hooks

Os React Hooks são funções criadas pela *framework* que permitem interagir com, e reagir ao, estado de um componente funcional. Podemos dizer que são o equivalente aos *lifecycle methods* dos componentes de classe, embora operem de forma diferente.

Na Figura 12 conseguimos visualizar o ciclo de vida de um componente funcional, os React Hooks disponíveis, e altura em que estes executam. Existem vários hooks com utilidades distintas que podem ser consultados na documentação do *React*[1], porem os mais essenciais são o `useState()` e o `useEffect()`.

O `useState()` permite-nos guardar e alterar o estado do componente. O valor que queremos guardar no estado é passado como argumento à função, sendo retornado um par com um *getter* e um *setter* para interagir com esse mesmo valor. Quando utilizamos o *setter* do `useState()` estamos a inicializar a fase de **Updating** do componente.

[1] https://reactjs.org/docs/hooks-reference.html - visitado em 20/12/2021

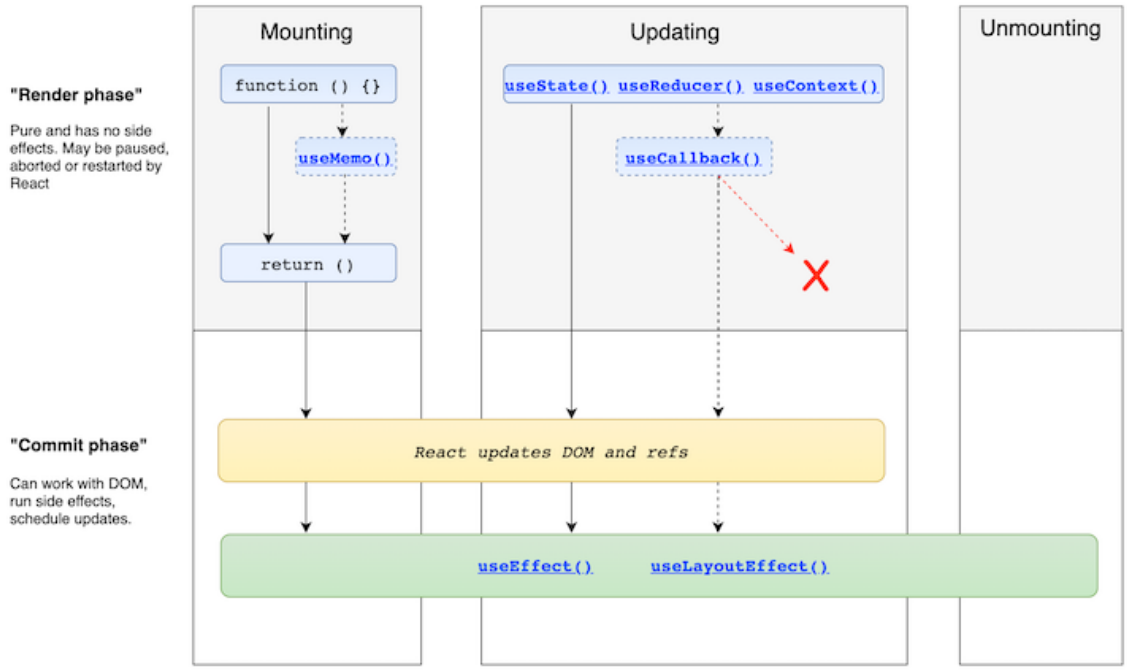


Figura 12: Ciclo de vida de um componente funcional em *React*, Fonte: https://blog.logrocket.com/guide-to-react-useeffect-hook/

Já o `useEffect()` permite reagir a qualquer alteração que ocorra no componente. Aceita como argumento uma função, e uma lista com as dependências às quais queremos reagir. A função incluída como argumento irá ser chamada quando o valor de uma ou mais dependências for alterado.

Ao contrário das funções `componentDidMount()`, e `componentDidUpdate()`, o `useEffect()` é assíncrono e não bloqueante. O que quer dizer que a função responsável por reagir a um determinado efeito pode só executar depois de o componente ser apresentado no ecrã.

Voltando ao exemplo da Figura 11, com a alteração do tamanho da imagem ao clicar no nome. Mas agora utilizando um componente funcional, e React Hooks, teríamos o código da Listagem 4.5.

```
1  import React, { useState } from 'react'
2  const LabelWithImage (props) => {
3    const [imageSize, setImageSize] = useState(props.imageSize)
4    const handleOnClick = () =>
5      setImageSize(imageSize + 1)
6
7    return(
8      <div>
9        <img width={imageSize} height={imageSize} src={props.imageUrl} />
10       <span onClick={() => handleOnClick()}>{props.labelText}</span>
11     </div>
12   )
13 }
```

Listagem 4.5: *React* - Exemplo de um componente funcional com atualização de estado

Na linha 3, estamos usar o `useState()` para conseguir ler e alterar o tamanho da imagem dentro do componente. Depois criamos a função `handleOnClick()` para reagir aos cliques do utilizador, atualizando a variável que guarda o tamanho da imagem através do *React hook*. Por último, tal como no componente de classe, é necessário que o a altura e largura do elemento HTML que representa a imagem dependam da variável que guarda o tamanho da mesma (linha 9).

Podemos concluir que esta abordagem é mais legível e menos verbosa. Estas características motivaram as empresas a adotar este método de criar componentes em detrimento do uso de componentes de classe.

4.4 Interação com APIs

Sendo o *React* uma framework apenas de UI, a menos que o conteúdo do website seja estático, é sempre necessário interagir com APIs para obter a informação necessária à renderização das páginas, ou até para efetuar ações no *backend* da aplicação. Tal como foi dito na Secção 3.1, o código destas interações deve estar isolado em serviços para conseguirmos manter a aplicação desacoplada e organizada.

Imaginemos a título de exemplo que queremos renderizar as publicações da *homepage* do Facebook, presentes na Figura 4. Primeiro temos de construir o serviço (entidade Service na Arquitura genérica da Figura 8) responsável por ir buscar as últimas publicações através de alguma API. Imaginemos que o existe um *endpoint* da *Facebook* que retorna uma lista com as últimas publicações no URL `https://api.facebook.com/posts`. O serviço ficaria representado no ficheiro `PostsService.js` presente na Listagem 4.6.

```
1 const BASE_URL = 'https://api.facebook.com/posts'
2
3 const getPosts = () => {
4   return await fetch(BASE_URL)
5     .then(response => response.json())
6 }
7
8 export { getPosts }
```

Listagem 4.6: PostsService.js

Agora que já existe no serviço o método `getPosts()`, responsável por retornar os posts mais recentes, temos apenas de criar o *ContainerComponent* que irá utilizar este método.

Assim surge o `PostsContainer.js` responsável por utilizar o serviço para ir buscar as publicações e invocar o componente responsável por desenhar cada publicação, e assim desenhar a lista de todas as publicações. Este componente seria representado como na Listagem 4.7.

```
1 import React, { useEffect, useState } from 'react'
2 import { getPosts } from './services/PostsService'
3
```

```
4  const PostsContainer = () => {
5    const [posts, setPosts] = useState([])
6
7    useEffect(() => {
8      setPosts(getPosts()) // utiliza o método getPosts do serviço para obter
           os posts mais recentes e depois atualiza o estado interno do
         componente com a função setPosts
9    }, [])
10
11   return (
12     <div className='posts-container'>
13       {
14         posts.length > 0
15         ?  posts.map((post) => <Post post={post} />)
16         :  <LoadingPosts />
17       }
18     </div>
19   )
20 }
21
22 export default PostsContainer
```

Listagem 4.7: PostsContainer.js

São usados 2 React Hooks, o `useState` para guardar as publicações no estado do componente, e `useEffect` para ir consultar a API através do serviço previamente criado. É neste `useEffect` que se preenche o estado com as publicações. Isto acontece apenas uma vez, depois do componente ser desenhado, uma vez que a lista de dependências está vazia.

Como o componente já foi desenhado quando o `useEffect` é chamado, é necessário termos um componente de *loading* (linha 16) para mostrar ao utilizador enquanto não atualizámos o estado do componente com as publicações vindas da API. Depois é só verificar se já possuímos as publicações no estado. Caso possuamos chamamos o componente responsável por desenhar cada publicação, caso contrário apresentamos o componente de *loading*.

A maquina de estados do componente representada na Figura 13 ajuda-nos a compreender os vários estados existentes e as suas transições.

4.5 Sumário

Neste capítulo começamos por explicar, na Secção 4.1, o funcionamento do *React*, mencionando as diferentes etapas do ciclo de vida de um componente e o funcionamento do seu estado.

A seguir, na Secção 4.2, analisamos o *design* e, fazendo a ponte com a divisão de um *design* em componentes sugerida pela Secção 2.4, vimos como é fácil, a partir dessa divisão, identificar quais os

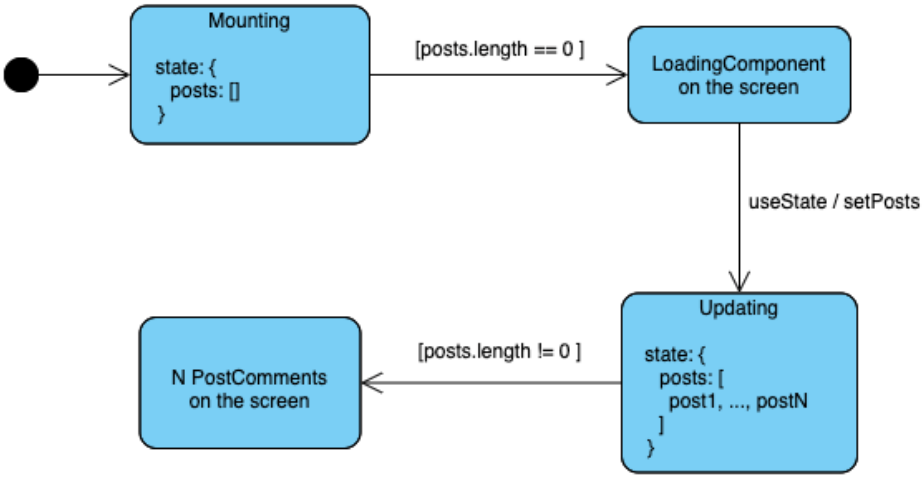


Figura 13: Máquina de estados do PostsContainer

componentes que necessitamos de implementar para construir uma determinada página.

Na Secção 4.3, foram analisadas as duas formas de construir componentes, utilizando componentes de classe, ou componentes funcionais. Conclui-se que os componentes funcionais são menos verbosos e mais legíveis, razões que levam as empresas mais modernas a usá-los ao invés de componentes de classe.

Por último, na Secção 4.4, analisamos como devemos interagir com APIs externas de modo a obter as informações necessárias para preencher o estado dos componentes.

5

Ferramenta de geração de código

Neste capítulo irá ser descrita a ferramenta de geração de código implementada durante a dissertação. Foram implementadas 2 funcionalidades principais, a geração de código a através de comandos, e a partir de protótipos. Irão ser descritas todas as abordagens, e os algoritmos implementados para elaborar as duas funcionalidades.

5.1 Objetivo

O objetivo geral desta ferramenta é mostrar ser possível aplicar os conceitos anteriormente referidos, e que estes dão origem a um processo de desenvolvimento mais sistemático e eficaz. Trata-se de uma prova de conceito, para demonstrar, não só que possível sistematizar o processo de desenvolvimento de interfaces, como também que, no limite, é possível automatizar parte do mesmo com a geração de código a partir de protótipos de interface.

Posto isto, a ferramenta terá duas principais funcionalidades:

1. A primeira, baseada em comandos que permitam ao utilizador gerar partes genéricas da arquitetura de uma interface, tais como, páginas, componentes, ficheiros de testes, etc. (Ver Secção 5.2)

2. A segunda, e mais complexa, terá como objetivo, a partir de um qualquer protótipo de interface, gerar o respetivo código e estilo. (Ver Secção 5.3)

Para implementar estas funcionalidades, iremos usar a linguagem Javascript, com a framework NodeJS, isto por ser uma linguagem de *scripting*[1], o que nos fornece com facilidade a habilidade de executar código de forma simples para automatizar determinadas tarefas, sem a necessidade de criar classes e objetos.

[1] https://en.wikipedia.org/wiki/Scripting_language - visitado em 20/12/2021

5.2 Geração de código baseada em comandos

Esta funcionalidade tem o particular objetivo de auxiliar os programadores no desenvolvimento de interfaces em *React*, utilizando a arquitetura genérica e a estrutura de pastas propostas nas Secções 3.1 e 3.2.

O ponto de partida é então o projeto base, criado pela ferramenta *create react app*[2]. Que nos oferece uma estrutura de pastas ainda longínqua do que foi idealizado na Secção 3.2. Assim o primeiro comando que devemos implementar, após ser criado o projeto base, é:

- create-folder-structure <path_to_repository>

Este comando irá permitir converter automaticamente a estrutura de pastas criada pela aplicação *create react app* na estrutura de pastas idealizada nesta dissertação.

O próximo passo prende-se com a geração de um componente genérico. Como a programação é orientada ao componente, o passo mais repetido pelos programadores é a criação dos mesmos. Para cada componente um programador precisa do criar o seu ficheiro em *React*, o seu ficheiro de estilo, e o seu ficheiro de testes.

Tendo isto em conta, seria de grande utilidade um comando que criasse todos estes ficheiros, mesmo que fossem apenas "boilerplates", para que os programadores os pudessem editar à posteriori.

Surge então o comando:

- create-component <path_to_repository> <component_name> <html_elements>?

Com este comando conseguimos automatizar o processo inicial de criação de um componente *React*. Todos os ficheiros são automaticamente criados. O componente é guardado numa pasta com o nome do mesmo, e inserido no local certo dentro do projeto, na raiz da pasta `/components`, conforme a estrutura idealizada.

Agora que já possuímos componentes, resta-nos gerar as páginas e rotas que os vão utilizar, surge então o último comando desta funcionalidade:

- create-route <path_to_repository> <route_name> <components_names>?

Este comando semelhante ao comando `create-component`, cria, automaticamente, novas rotas para a aplicação. O processo passa por editar o ficheiro com definição das rotas (`App.js`), e criar os ficheiros necessários para representar a página. Isto inclui ficheiros de código, de estilo, e de testes, tal como é feito em `create-component`.

Nas próximas secções iremos explicar em detalhe todos os comandos, mencionando a sua utilidade, e explicando como foram implementados.

[2] https://github.com/facebook/create-react-app - visitado em 20/12/2021

5.2.1 Comando create-folder-structure

Ao executar este comando a estrutura de pastas do projeto deve ser automaticamente reorganizada para que fique igual ao idealizado nesta dissertação. Assim é necessário que este comando consiga mover e criar, ficheiros e pastas. O código do mesmo encontra se num repositório no GitHub[3].

A diferença entre a estrutura do projeto antes e após executar o comando está presente na Figura 14. Como podemos observar foram criadas as pastas para dividir os diferentes tipos de ficheiros do projeto, e moveram-se os ficheiros de estilo para a pasta `src/assets/stylesheets`. Está então cumprido o objetivo deste comando.

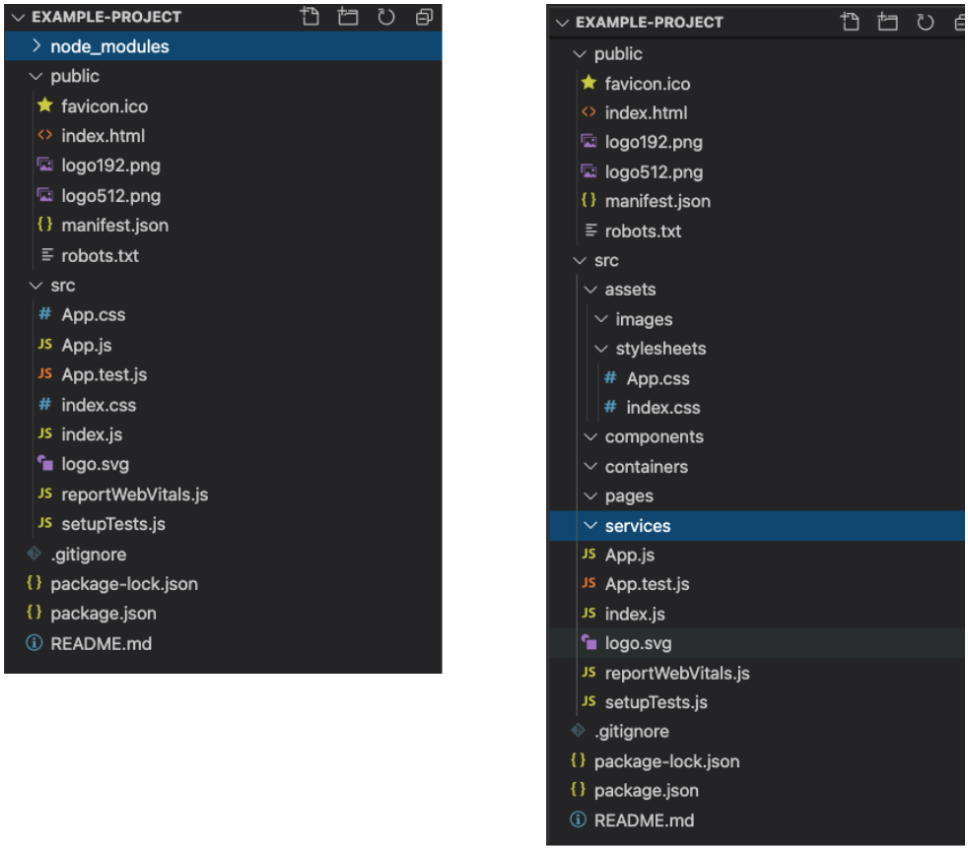


Figura 14: Antes e depois do comando createFolderStructure

5.2.2 Comando create-component

O objetivo deste comando é criar componentes seguindo a noção de *self-contained components*. Ou seja, no fim da sua execução, é criado um ficheiro com o código do componente (<component_name>.js), um ficheiro com o estilo do componente (<component_name>.sass), um ficheiro com testes ao componente (<component_name>.test.js), e um ficheiro a indicar o ponto de entrada do componente (package.json).

[3] https://github.com/nelsontss/reactDeveloperHelpers/blob/main/command_generator/createFolderStruct.js - visitado em 20/12/2021

A abordagem escolhida foi partir de um ficheiro "boilerplate" que representasse a estrutura de um qualquer componente. Elaborou-se então o ficheiro ComponentBoilerplate.js presente na Listagem 5.1.

```
1  import React from 'react'
2  import './ComponentBoilerplate.sass'
3
4  const ComponentBoilerplate = (props) => {
5    return (
6      <div>
7        Hello Im ComponentBoilerplate
8      </div>
9    )
10 }
11
12 export default ComponentBoilerplate
```

Listagem 5.1: ComponentBoilerplate.js

Dado o nome de um componente, é apenas necessário substituir as ocorrências de "ComponentBoilerplate" pelo nome em questão. Por exemplo, se executarmos `create-component example-project VideoCard` é criado o ficheiro apresentado na Listagem 5.2.

```
1  import React from 'react'
2  import './VideoCard.sass'
3
4  const VideoCard = (props) => {
5    return (
6      <div>
7        Hello Im VideoCard
8      </div>
9    )
10 }
11
12 export default VideoCard
```

Listagem 5.2: Ficheiro VideoCard.js

Conseguimos observar que foi criado um ficheiro para representar o componente "VideoCard", mas que este, claramente, não representará a versão final pretendida. É de relembrar que a finalidade deste comando é agilizar a criação de componentes, e não automatizar a sua geração, pelo que, nesta fase, é da responsabilidade do programador editar os mesmos para implementar a restante parte. Mais à frente, na Secção 5.3 é descrita uma funcionalidade de geração de componentes onde o código gerado estará muito mais próximo da versão final do componente pretendido.

Existe ainda a possibilidade de gerar o componente com elementos HTML pré-definidos através do argumento opcional <html_elements>. Ao passarmos uma lista de elementos HTML, o comando irá

inseri-los no código do componente pela mesma ordem que são passados. Por exemplo, se for executado, `create-component example-project VideoCard span,div,img` é criado o ficheiro, `VideoCard.js`, apresentado na Listagem 5.3.

```
1  import React from 'react'
2  import './VideoCard.sass'
3
4  const VideoCard = (props) => {
5    return (
6      <div>
7        <span></span>
8        <div></div>
9        <img></img>
10     </div>
11   )
12 }
13
14 export default VideoCard
```

Listagem 5.3: Ficheiro VideoCard.js, com os elementos HTML passados como argumento

Os elementos passados como argumento foram envolvidos numa , e representam agora o componente VideoCard, tal como pretendido. Não lhes é passada nenhuma propriedade, valor, ou classe. Isto porque iria tornar o comando cada vez mais complexo e o objetivo destes comandos é serem de utilização rápida e fácil.

Na parte do estilo a única automatização feita pela execução deste comando é a criação de um ficheiro de estilo vazio, com o nome do componente. Este ficheiro é automaticamente importado, pelo que é apenas necessário começar a especificar as regras de estilo para estas serem aplicadas.

Por último é gerado um ficheiro de testes para incentivar os programadores a testarem os seus componentes, este ficheiro é também gerado através de um "boilerplate", especificado na Listagem 5.4.

```
1  import { render, screen }   from '@testing-library/react';
2  import ComponentBoilerplate from './ComponentBoilerplate';
3
4  test('renders hello text', () => {
5    render(<ComponentBoilerplate />);
6    const linkElement = screen.getByText(/Hello Im ComponentBoilerplate/i);
7    expect(linkElement).toBeInTheDocument();
8  });
```

Listagem 5.4: Ficheiro de testes genérico

Tal como na geração do código *React*, também neste caso apenas é substituído o nome "ComponentBoilerplate" pelo nome do componente que queremos implementar. Por exemplo, utilizando mais uma vez o comando `create-component example-project VideoCard`, teríamos o ficheiro de testes presente na Listagem 5.5.

```
1 import { render, screen }   from '@testing-library/react';
2 import VideoCard from './VideoCard';
3
4 test('renders hello text', () => {
5   render(<VideoCard />);
6   const linkElement = screen.getByText(/Hello Im VideoCard/i);
7   expect(linkElement).toBeInTheDocument();
8 });
```

Listagem 5.5: Ficheiro de testes para o componente VideoCard

Este ficheiro possui apenas um teste que verifica se o componente contém o texto "Hello Im VideoCard", o teste passa caso o comando tenha sido invocado sem <html_elements>, como podemos observar na Listagem 5.2, onde o componente contem o texto que procuramos. Para executar estes testes podemos usar a biblioteca JEST[4].

Em termos de estrutura de pastas o componente gerado e todos os seus ficheiros dependentes permanecem numa pasta com o nome do componente criada na pasta `src/components`. Como podemos ver na Figura 15.

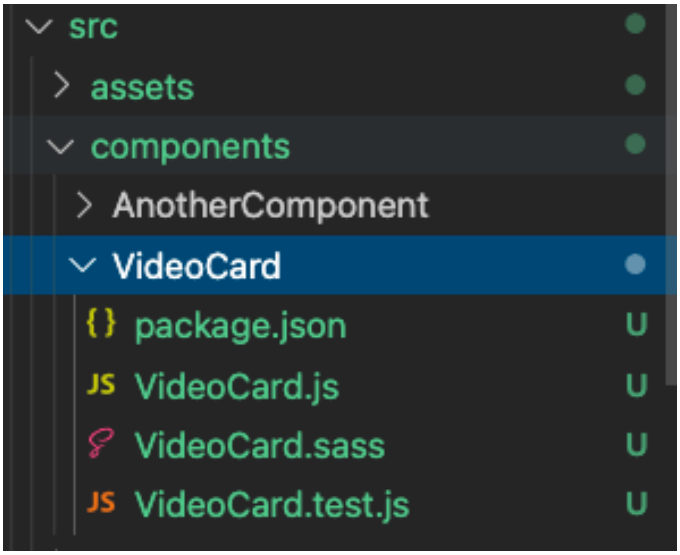


Figura 15: Estrutura da pasta do Componente

Com este comando também é possível criar **ContainerComponents**, podemos fazê-lo se passarmos o sufixo "Container" no nome do componente. Por exemplo, se executarmos create-component example-project VideoCardsContainer, irá ser criado um componente com esse nome, mas na pasta `src/containers` como podemos ver na Figura 16.

[4] https://jestjs.io/docs/tutorial-react - visitado em 20/12/2021

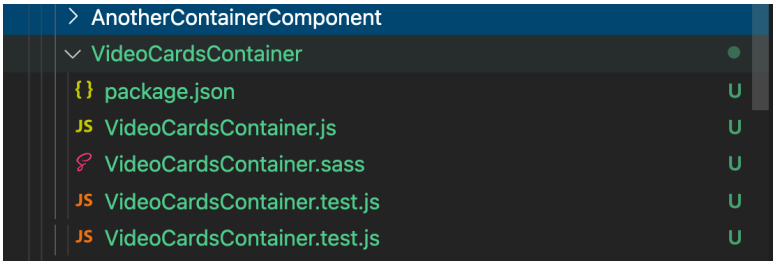


Figura 16: Criação de um ContainerComponent

5.2.3 Comando create-route

A finalidade desta última funcionalidade é gerar um componente *React* que represente uma página web, numa determinada rota.

Como mencionado na Secção 3.1, para implementar uma página em *React* recorremos a um **Router** implementado pela biblioteca *react-router*[5]. Para exemplificar esta funcionalidade imaginemos um *website*, com várias páginas, onde queremos adicionar a rota de `/home` para representar a *homepage* do *website*.

Em primeiro lugar, no ficheiro de entrada do projeto (App.js) é necessário importar os componentes que irão definir as rotas (Listagem 5.6). Nas linhas 1 a 5 importamos os componentes do **Router** que vão ser essenciais para definir as rotas.

```
1 import {
2   BrowserRouter as Router,
3   Switch,
4   Route
5 } from "react-router-dom";
6
7 import Home from './pages/Home'
```

Listagem 5.6: Importação das rotas e dos componentes do *react-router* no ficheiro App.js

Na linha 7 importamos o componente "Home" que define a página que queremos para a rota `/home`.

Precisamos, em seguida, de gerar o componente "Home", para isso podemos reutilizar muito do comportamento implementado na Secção 5.2.2, visto que a página é apenas mais um componente composto por outros mais simples. A única diferença é que para efeitos de manutenção este componente do tipo **Page** é separado e posto na pasta `src/pages`. Foi então generalizado o comportamento comum de criação de um componente no ficheiro sharedCreateComponent.js[6], sendo depois utilizado pelos comandos *create-component* e *create-route*.

Por último precisamos de declarar as rotas no corpo do componente App.js, como especifica a Listagem 5.7.

[5] https://reactrouter.com/ - visitado em 20/12/2021

[6] https://github.com/nelsontss/reactDeveloperHelpers/blob/main/command_generator/sharedCreateComponent.js - visitado em 20/12/2021

```
<Router>
  {/* A <Switch> looks through its children <Route>s and
      renders the first one that matches the current URL. */}
  <Switch>
    <Route path="/home">
      <Home />
    </Route>
  </Switch>
</Router>
```

Listagem 5.7: Declaração das rotas no ficheiro App.js

Declaramos a rota `/home` através do componente implementado pela biblioteca, que irá mostrar a página implementada no componente "Home".

Para ser possível adicionar sempre novas rotas com a execução deste comando, mantendo as que já foram criadas, é criado um ficheiro (settings.json) com a seguinte estrutura:

```
{
  "routes": ["Home"]
}
```

Neste ficheiro são guardadas todas as rotas adicionadas pelo comando, para ser possível reconstruir o ficheiro com todas as rotas a cada nova execução.

Em suma, ao executar os seguintes comandos:

- create-route example-project Home
- create-route example-project Profile

Obtemos o seguinte código no ficheiro `App.js`, presente na Listagem 5.8.

```
import './assets/stylesheets/App.css';
import {
  BrowserRouter as Router,
  Switch,
  Route
} from "react-router-dom";

import Home from './pages/Home'
import Profile from './pages/Profile'

function App() {
  return (
    <Router>
      {/* A <Switch> looks through its children <Route>s and
          renders the first one that matches the current URL. */}
```

```
16        <Switch>
17          <Route path="/home">
18            <Home />
19          </Route>
20          <Route path="/profile">
21            <Profile />
22          </Route>
23        </Switch>
24      </Router>
25    );
26  }
27
28  export default App;
```

Listagem 5.8: Ficheiro App.js com as rotas "/home" e "/profile"

Onde podemos ver declaradas as rotas `/home` e `/profile`, importadas dos componentes "Home" e "Profile".

Na pasta `/src/pages` são geradas as páginas "Home" e "Profile". Como vemos na Figura 17.

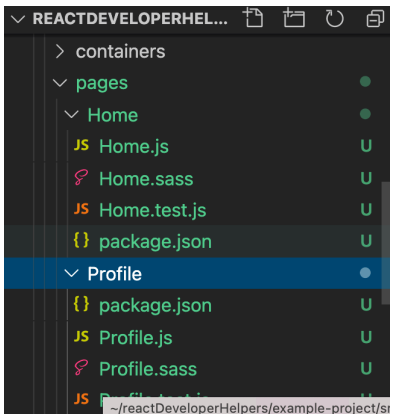


Figura 17: Componentes "Home" e "Profile"

No ficheiro settings.json ficam guardadas as rotas adicionadas, da seguinte forma:

```
{
  "routes": ["Home","Profile"]
}
```

Podemos ainda importar automaticamente componentes para as páginas definidas. Se passarmos os nomes dos componentes no argumento opcional `<components_names>?` a ferramenta irá importar todos eles para o ficheiro que define a página a ser criada. Assim, se usarmos o comando:

- create-route example-project Home 'HomeHeader,HomeContent,HomeFooter'

O ficheiro "Home.js", presente na Listagem 5.9, irá importar e mostrar todos os componentes passados como argumento.

```
1  import React from 'react'
2  import './Home.sass'
3  import HomeHeader from './components/HomeHeader'
4  import HomeContent from './components/HomeContent'
5  import HomeFooter from './components/HomeFooter'
6
7  const Home = (props) => {
8    return (
9      <div>
10       <HomeHeader></HomeHeader>
11       <HomeContent></HomeContent>
12       <HomeFooter></HomeFooter>
13     </div>
14   )
15 }
16
17 export default Home
```

Listagem 5.9: Ficheiro Home.js com a importação de componentes

5.3 Geração de código baseada em protótipos

Tal como o nome indica, o objetivo desta funcionalidade é partir dos protótipos de interface, concebidos pelos *designers*, e gerar os respetivos componentes. Uma funcionalidade deste género automatiza muito do trabalho que um programador tem. Mesmo que o código gerado não seja ideal, o número de alterações a fazer não supera o tempo de implementar tudo de raiz.

5.3.1 Ponto de partida

No processo de *design* são usadas diversas ferramentas, dependendo do *designer*. Os protótipos podem ser feitos em programas como *AdobeXD*, *Figma*, *Sketch*, entre outros.

Cada um destes programas possui as suas particularidades, a sua forma de implementar um protótipo, e o seu resultado final. Para a implementação desta funcionalidade iremos utilizar o *Figma*[7], e tentar automatizar a geração de código, partindo de protótipos elaborados por este programa.

No *Figma*, como podemos ver na Figura 18, cada elemento corresponde a uma camada. Por exemplo, o texto "MESSAGE" é representado pela camada com o mesmo nome à esquerda. Cada elemento pode ser aninhado num *auto-layout*. Um *auto-layout* é mais uma camada que dita a forma como os elementos aninhados sobre ela se organizam. Por exemplo, o *auto-layout* com o nome "company display-flex align-

[7] https://www.figma.com - visitado em 20/12/2021

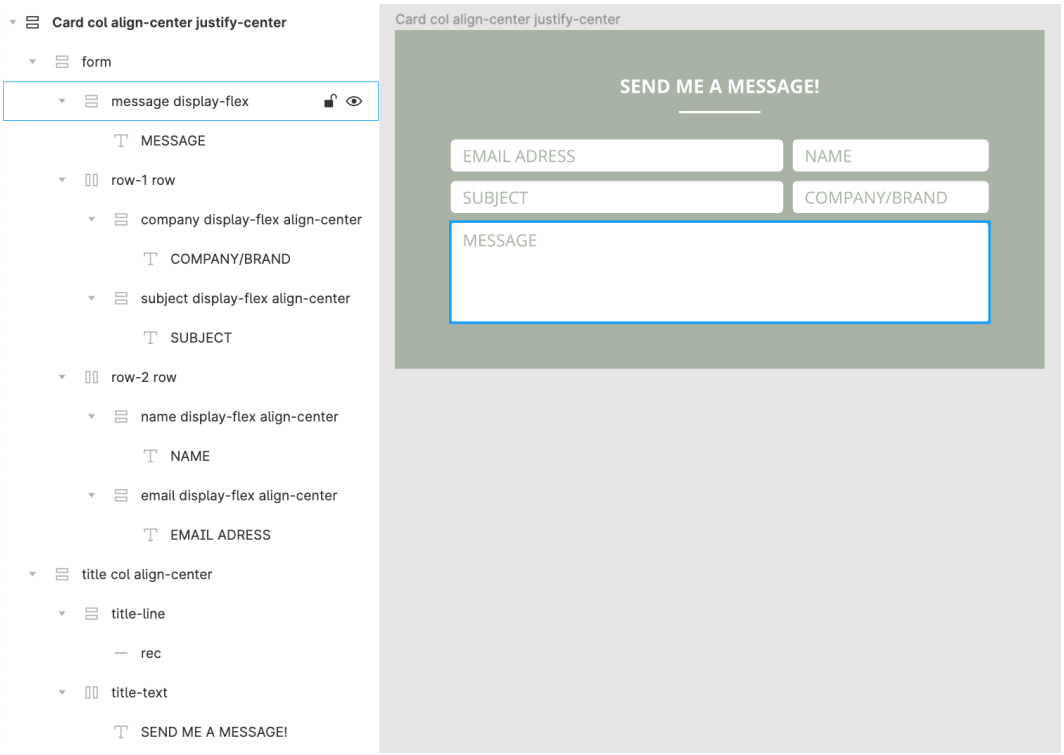


Figura 18: Figma - Organização por camadas

center", possui controlo sobre o posicionamento do seu elemento aninhado que neste caso é o texto "COMPANY/BRAND". Na Figura 19 podemos ver que esta camada especifica as margens, o *padding*, e alinhamento do texto "COMPANY/BRAND", para além de especificar o seu próprio tamanho e cor de fundo.

É possível ter *auto-layouts* dentro de *auto-layouts*, sendo que os de níveis mais altos ditam a organização daqueles que estão mais aninhados. O *auto-layout* com o nome "row-1 row" alinha horizontalmente os dois *auto-layouts* aninhados, e define o espaçamento entre eles. Num nível ainda mais alto o *auto-layout* com o nome "form" alinha os seus *auto-layouts* aninhados verticalmente, e define os seus espaçamentos.

5.3.2 Exportação, e análise do ficheiro SVG

Agora que já sabemos como é construído um protótipo no *Figma*, temos que saber como vamos passar este protótipo meramente visual para código *React*. Primeiro precisamos de exportar o componente para algum tipo de ficheiro, para que possamos inferir os elementos e as propriedades de cada *mockup*. No caso do *Figma*, a ferramenta permite-nos exportar qualquer camada e todas as camadas aninhadas nessa mesma para um ficheiro SVG. Conseguimos então exportar o componente "Card" da Figura 18 para um ficheiro SVG, o resultado está presente no Anexo I.

Após analisar o ficheiro SVG produzido pelo *Figma* é possível perceber que:

- Cada camada de "auto-layout" dá origem a um nodo no SVG com a etiqueta <g>

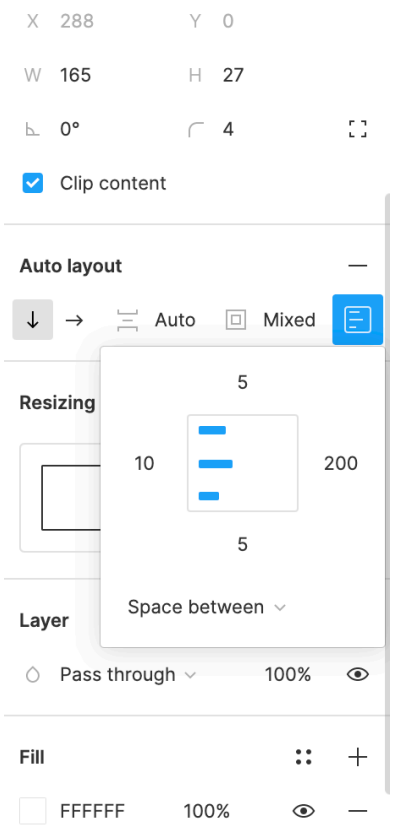


Figura 19: Figma - Auto-layout

- A propriedade id de cada elemento contem o nome da camada no *Figma*.
- Caso a camada possua cor de fundo, a etiqueta <g> irá conter uma etiqueta <rect> como primeiro filho. Este <rect> contem as propriedades de posicionamento, tamanho e cor da camada sobre a qual está aninhado.
- Os textos são representados no SVG por nodos com a etiqueta <text>, esses nodos possuem a cor, tamanho, e fonte do texto. Bem como o seu posicionamento e o texto em questão.

Limitações

A melhor opção disponível para ler o conteúdo dos protótipos construídos no *Figma* são os ficheiros SVG, porque as outras opções seriam imagens ou ficheiros PDF. Uma possível parceria com a empresa, que nos desse acesso à ferramenta e a mais informações sobre os protótipos também seria uma opção válida, e a explorar no futuro.

Apesar de ser a melhor opção disponível, a funcionalidade de exportação de protótipos para ficheiros SVG não visa a geração de código. Por isso, partir deste ficheiro para gerar código *React* tem algumas limitações, essencialmente porque a informação presente no ficheiro é insuficiente, em vários aspetos, para caracterizar com precisão uma interface.

Não existe nenhuma referência sobre as margens e orientações dos elementos do SVG, apenas possuímos o seu posicionamento absoluto, a sua altura, e a sua largura. Não temos nenhuma informação se dois elementos vizinhos estão na mesma linha ou se estão na mesma coluna. Também não existe nada que nos diga, diretamente, qual o espaço entre 2 elementos vizinhos.

Os elementos de "auto-layout" (com etiqueta <g>), não possuem nenhum tipo de informação sobre a seu posicionamento ou sobre as suas dimensões.

Nos campos de texto a situação também não é a ideal, visto que não possuímos as suas dimensões, apenas o tamanho da fonte, a fonte, e o seu posicionamento absoluto.

Estas limitações irão afetar a geração dos estilos, por isso na Secção 5.3.6 serão descritas as abordagens implementadas para as contornar.

5.3.3 Abordagem

Para atacar o problema, foi definida uma abordagem de três passos. O ponto de partida é o ficheiro SVG exportado a partir do protótipo elaborado no *Figma*, e o objetivo é, no final, obter um componente *React* que represente o protótipo inicial.

Os três passos desta abordagem são:

1. Transformar o conteúdo do ficheiro SVG num objeto que represente os vários elementos HTML daquele protótipo.
2. Gerar o código HTML dos vários elementos presentes no objeto obtido no primeiro passo.
3. Criar um componente *React* e injetar nesse mesmo componente o código HTML, gerado no segundo passo.

Primeiro, para criar um objeto que represente os vários elementos HTML do protótipo do *Figma*, é necessária uma classe que os represente, e guarde todas as suas propriedades, daqui em diante chamaremos esta classe de `HtmlElement`. A primeira etapa consiste em criar um algoritmo para ler a informação do ficheiro SVG e a transformar num objeto da classe `HtmlElement`. Este algoritmo é explicado na Secção 5.3.4.

A classe `HtmlElement` consiste no seguinte:

```
class HtmlElement {
  constructor(type, classname, styles, value, children, position){
    this.type      = type
    this.classname = classname
    this.value     = value
    this.children  = children
    this.styles    = styles
}
```

Esta classe representa um elemento HTML e contém as seguintes propriedades:

- **type -** *String* com o tipo de elemento HTML, (ex. 'div').
- **classname -** *String* com o nome da classe do elemento HTML (ex. 'login-btn').
- **value -** *String* com o conteúdo do elemento HTML (ex. 'Login').
- **children -** Uma coleção de *HtmlElements* que representam os elementos HTML aninhados ao elemento pai.
- **styles -** Uma coleçao de propriedades de estilo de um elemento HTML (ex. [{font-size: 15px}, {width: 200px}]).
- **position -** Um par de *Floats* que representa a posição de um elemento HTML no ecrã (ex. [200, 50]).

Em seguida é necessário gerar o código HTML correspondente a um objeto da classe `HtmlElement`. Para isto precisamos de implementar mais um algoritmo que consiga partir de um objeto `HtmlElement` e produzir o seu respetivo código HTML. Este algoritmo está especificado na Secção 5.3.5.

Por último, é necessário criar um componente *React*, que irá representar o protótipo do componente elaborado no *Figma*. Neste componente é injetado o código HTML obtido no passo anterior. A estrutura genérica do componente *React* é apresentada na Listagem 5.10.

```
1  import React from 'react'
2  import './GeneratedComponent.sass'
3
4  const GeneratedComponent = (props) => {
5    return (
6      // Os elementos html do SVG serão inseridos aqui.
7    )
8  }
9
10 export default GeneratedComponent
```

Listagem 5.10: *React* - Estrutura genérica de um componente

O código HTML é injetado na linha 6, e a partir deste momento já possuímos um componente *React* com todos os elementos HTML derivados do protótipo inicial.

Para o último passo, podemos reutilizar a funcionalidade de criação de componentes da secção 5.2.2. Apenas teremos que adaptar a função para receber uma *string* com os elementos HTML gerados e introduzi-los no código *React* que já é criado pela funcionalidade.

Também será necessário que a classe consiga gerar um ficheiro de estilo com as propriedades de estilo dos elementos HTML, para que a aparência do componente seja igual ao *design*. A abordagem desta geração será explicada na Secção 5.3.5

No diagrama da Figura 20 podemos ver todo o fluxo da abordagem utilizada.

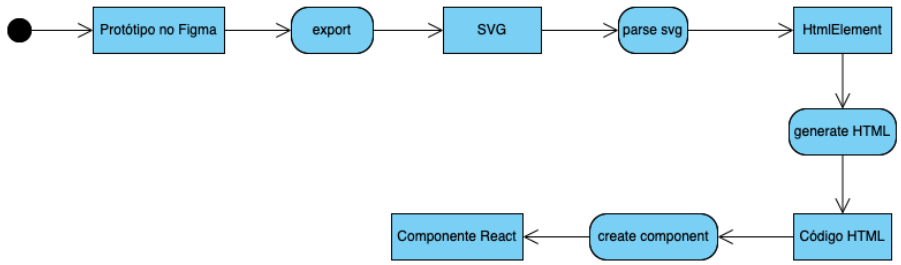


Figura 20: Fluxo da abordagem

5.3.4 Transformação do SVG num HtmlElement

Para transformar o SVG num objeto `HtmlElement`, é preciso mapear cada nodo no seu respetivo elemento HTML. Foi utilizada uma abordagem recursiva, onde é feito *parsing* do elemento principal e depois dos elementos aninhados de igual forma. Os `HtmlElements` que resultam do *parsing* dos elementos aninhados são colocados na propriedade **children** do `HtmlElement` pai. O *parsing* de um elemento em particular depende do tipo de nodo, da seguinte forma:

- **<g> -** Caso seja do tipo g, é criada um `HtmlElement` do tipo "div". Depois é necessário perceber se existe um *background* associado a este nodo (caso o primeiro filho seja do tipo <rect>), e associar as propriedades do *background* a este `HtmlElement`.
- **<rect> -** Caso seja do tipo rect, é preciso verificar através da propriedade **id** que não é um *background* (se não possuir id é um *background*). Caso não seja um *background* é criada um `HtmlElement` do tipo 'div' com as propriedades do nodo.
- **<text> -** Caso seja do tipo text, é criado um `HtmlElement` do tipo "div" com as propriedades do texto, e com o valor do mesmo que está contido no nodo <tspan> dentro do nodo <text>
- **classnames -** A propriedade classnames da classe `HtmlElement` será preenchida com o valor da propriedade id de cada elemento do svg.
- **styles -** A propriedade styles da classe `HtmlElement` será preenchida com os estilos presentes no elemento, caso existam. É necessário mapear alguns destes estilos, em regras de CSS. Outros já são idênticos. Por exemplo, a propriedade "width=400" dá origem a uma regra de CSS "width: 400px", já a propriedade "rx=4" dá origem à regra "border-radius: 4px".

O pseudocódigo da Listagem 5.11 explica com mais clareza o algoritmo. Este pseudocódigo está em inglês e com uma sintaxe parecida com linguagem Javascript, isto para termos uma perceção geral do algoritmo, sem os seus pormenores de implementação.

```
1 function parseSvg(svgInitialNode) {
2   htmlElement = parseNode(svgInitialNode)
```

```
  children    = []
  foreach (childrenNode in svgInitialNode.children) {
    children.push(parseSvg(childrenNode))
  }
  htmlElement.children = children
  return htmlElement
}

function parseNode(node) {
  switch(node) {
    case g
      htmlElement = parseTagG(node)
    case rect
      htmlElement = parseTagRect(node)
    case text
      htmlElement = parseTagText(node)
  }
  return htmlElement
}
```

Listagem 5.11: Algoritmo de transformação de um SVG num HtmlElement

5.3.5 Geração do HTML

Para gerar o HTML correspondente ao conteúdo do ficheiro SVG, partimos então de uma instância da classe `HtmlElement` que foi construída através da transformação do SVG. Foi definido um método da classe `HtmlElement` que será o responsável por gerar a *string* HTML de uma dada instância. Este método terá de olhar para o tipo de elemento em questão, para os seus elementos aninhados, e também para a sua classe. A Listagem 5.12 ilustra o método.

```
generateChildrenOrValue(identation) {
  if (this.value){
    return `${' '.repeat(identation)}${this.value}`
  } else {
    return this.children.map(entry => entry.generateHtml(identation)).join(
        '\n')
  }
}

divString = (identation) => {
  return (
    `${' '.repeat(identation)}<${this.type}${this.generateClassname()}>\n`
       +
    `${this.generateChildrenOrValue(identation + 2)}\n` +
    `${' '.repeat(identation)}</${this.type}>`
```

```
14    )
15  }
16
17  generateHtml(identation = 4) {
18    switch (this.type) {
19      case 'div':
20        return this.divString(identation)
21    }
22  }
```

Listagem 5.12: Métodos responsáveis por gerar o HTML de um HtmlElement

Começamos por ver o tipo de elemento, para já só é dado suporte a "divs", mas no futuro poderão ser adicionados novos elementos. É então gerada a *string* com as etiquetas do elemento ( ), e no meio dessas etiquetas irão ser inseridos os elementos aninhados ou então o valor do elemento em questão (comportamento da função `generateChildrenOrValue`). Para a inserção dos elementos aninhados é usada de forma recursiva a função `generateHtml` para cada um dos elementos da propriedade `children` (comportamento da linha 5).

5.3.6 Geração do estilos

Na geração dos estilos iremos utilizar *Sass*, principalmente por ser menos verboso, e ter menos açúcar sintático, o que faz com que seja mais fácil de gerar instruções *Sass* do que instruções *CSS*.

A estrutura do ficheiro de estilo gerado será a apresentada na Listagem 5.13.

```
1   .htmlElement.classname
2     // estilos do elemento em questão
3
4     .htmlElement.children[0].classname
5       // estilos do primeiro elemento aninhado ao elemento em questão
6       .htmlElement.children[0].children[0].classname
7         // estilos de elementos ainda mais aninhados
8         ...
9           .htmlElement.children[0](n+1 vezes).children[0].classname
10            // estilos do elemento mais aninhado
11    .htmlElement.children[1].classname
12      // estilos do segundo elemento aninhado ao elemento em questão
```

Listagem 5.13: Estrutura do ficheiro de estilos gerado

O ficheiro começa com uma regra sobre a classe do elemento da raiz, que representa o componente, vinda da variável de classe `classname`. A seguir, dentro dessa regra, apresentam-se todas as propriedades de estilo do elemento, que se encontram na variável de classe `styles`.

Depois, para cada elemento aninhado e de forma recursiva até ao elemento mais aninhado, surgem novas regras com as classes dos respetivos elementos e com as suas respetivas propriedades de estilo.

As funções apresentadas na Listagem 5.14 implementam este algoritmo recursivo.

```
1  generateChildrenStyles(identation) {
2    if(this.children)  return `${this.children.map(entry => entry.
       generateStylesSass(identation)).join('\n')}`
3  }
4
5  generateStylesSass(identation = 0) {
6    let string = ''
7    if(this.classname){
8      string = string.concat(this.classnameSassString(identation))
9      identation += 2
10   }
11   if (this.styles) {
12     string = string.concat(`${this.stylesSassString(identation)}\n`)
13   }
14   return string.concat(`${this.generateChildrenStyles(identation)}`)
15 }
```

Listagem 5.14: Algoritmo de geração de estilos

A função `generateStylesSass` é responsável por gerar o estilo de um elemento, colocando a regra para a seu nome de classe (linhas 7 a 10), e as suas propriedades de estilo (linhas 11 a 13). A seguir, concatena o que foi gerado com a chamada recursiva sobre cada um dos seus elementos aninhados (linha 14).

Posicionamento, tamanhos e margens

Tal como no processo de *design*, também na programação dos estilos queremos definir o posicionamento dos nossos elementos com base em alinhamentos. Por exemplo, alinhar os elementos no meio do ecrã, ou à esquerda, ou no canto superior direito. Esta é a abordagem mais utilizada. Por outro lado, atribuir uma posição absoluta a um elemento HTML não é uma prática comum, isto porque consoante o tamanho do ecrã a posição onde queremos o elemento pode variar. Como a informação sobre os alinhamentos que utilizamos no "auto-layout" do *Figma* não passa para o ficheiro SVG, é necessário encontrar outra forma de perceber que tipo de alinhamentos queremos usar em determinados elementos.

Para o efeito iremos utilizar o campo id de cada elemento do ficheiro SVG. Foi definido um conjunto de *keywords* a utilizar neste campo, bem como os estilos correspondentes a cada *keyword*. A utilização destas *keywords* é da responsabilidade dos *designers* que implementam o protótipo, para que o processo seja automático do lado dos programadores. A Tabela 2 mostra a correspondência entre *keywords* e o seu respetivo estilo.

Utilizando esta abordagem quando for necessário especificar que uma camada contém os seus elementos em coluna e alinhados ao centro, usamos as *keywords* "col" e "align-center" no id da mesma,

Keywords	CSS rules
row	display: flex flex-direction: row
col	display: flex flex-direction: col
align-center	align-items: center
justify-center	justify-content: center
display-flex	display: flex

Tabela 2: Tabela com as *keywords* de alinhamento

tal como está na camada "title col align-center" da Figura 18. Isto vai fazer com que na fase da transformação do SVG num `HtmlElement` sejam adicionados aos seus estilos as propriedades que estão nas *keywords* utilizadas, que neste caso seriam:

```
'display': 'flex',
'flex-direction': 'column',
'align-items': 'center'
```

Deste modo, todos os elementos dentro do elemento com id "title col align-center" ficam posicionados em coluna, e alinhados ao centro.

Visto que não iremos usar posicionamentos absolutos, para que o componente gerado fique igual ao protótipo é preciso conseguir deduzir as margens do mesmo para os seus componentes vizinhos, caso contrário acabaríamos com os elementos todos encostados uns aos outros. Conseguimos calcular estas margens através das propriedades que estão no SVG. Vejamos o exemplo da Listagem 5.15.

```
1 <g id="row">//elemento A
2   <rect x="47" y="127" width="280" height="27" rx="4" fill="white"/> //
        elemento A.1
3   <rect x="335" y="127" width="165" height="27" rx="4" fill="white"/>//
        elemento A.2
4 </g>
```

Listagem 5.15: Elemento com 2 sub-elementos dispostos numa linha

Aqui temos uma camada com dois subelementos dispostos numa linha (através da *keyword* "row"). Como estão dispostos em linha, apenas existe margem à direita do elemento A.1, para o elemento A.2, como podemos observar na Figura 21. A margem à direita pode então ser calculada da seguinte forma:

1. margin-right = A.2.x - (A.1.x + A.1.width)

Este calculo funciona quer estejam 2, 3 ou mais elementos consecutivos, em linha. A margem à direita de um elemento X é sempre calculada com base no posicionamento e dimensões do elemento Y imediatamente à direita.

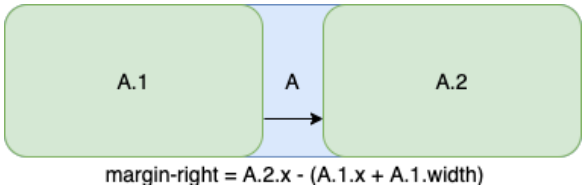


Figura 21: Calculo da margem à direita

Caso os elementos estejam dispostos em coluna (através da *keyword* "col"), como na Listagem 5.16, iremos obter uma margem abaixo do elemento A.1, para o elemento A.2, como podemos observar na Figura 22. Essa margem pode ser calculada da seguinte forma:

1. margin-down = A.2.y - (A.1.y + A.1.height)

```
1 <g id="col">//elemento A
2   <rect x="47" y="92" width="280" height="27" rx="4" fill="white"/> //
       elemento A.1
3   <rect x="47" y="127" width="280" height="27" rx="4" fill="white"/>//
       elemento A.2
4 </g>
```

Listagem 5.16: Elemento com 2 sub-elementos dispostos numa coluna

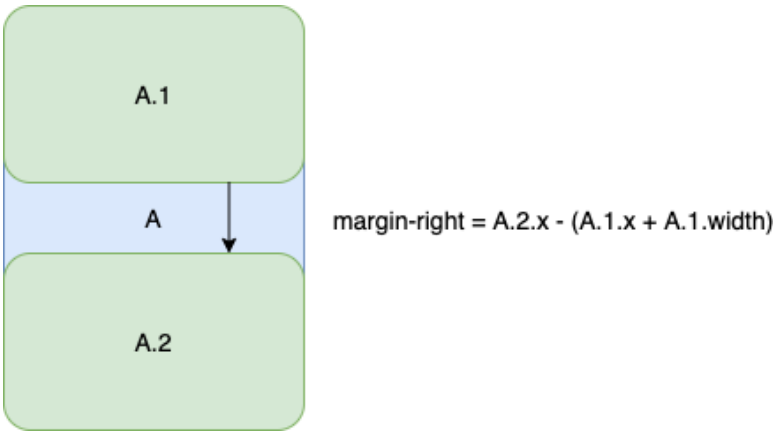


Figura 22: Calculo da margem para baixo

Tal como o cálculo anterior, também este se aplica caso sejam 2, 3 ou mais elementos consecutivos, em coluna. A margem para baixo de um elemento X é sempre calculada com base no posicionamento e dimensões do elemento Y imediatamente abaixo.

Como iremos ter elementos aninhados em outros elementos, irá ser necessário lidar com casos mais complexos, como, por exemplo, o caso presente na Listagem 5.17.

```
1 <g id="col">//elemento A
2   <g id="row">//elemento B
```

```
3     <rect x="47" y="92" width="280" height="27" rx="4" fill="white"/> //
          elemento B.1
4     <rect x="335" y="92" width="165" height="27" rx="4" fill="white"/>//
          elemento B.2
5   </g>
6   <g id="row">//elemento C
7     <rect x="47" y="127" width="280" height="27" rx="4" fill="white"/> //
          elemento C.1
8     <rect x="335" y="127" width="165" height="27" rx="4" fill="white"/>//
          elemento C.2
9   </g>
10 </g>
```

Listagem 5.17: Elemento complexo com vários elementos aninhados

Neste caso, possuímos um elemento A com os elementos B e C dispostos em coluna. Os elementos B e C possuem os seus elementos aninhados dispostos em linha. Para calcular a margem entre os elementos, B.1/B,2, e C.1/C.2, podemos utilizar as fórmulas vistas em cima. Contudo, não temos forma de calcular as margens entre os elementos B e C, isto porque as etiquetas <g> que os representam não possuem o seu posicionamento nem as suas dimensões. Para resolver isto temos de calcular as dimensões e o posicionamento de cada etiqueta <g>. Fazemos isto a partir dos seus elementos aninhados. Este calculo difere caso o elemento "pai" possua os seus elementos "filho" em linha ou em coluna. A Figura 23, mostra como seria o cálculo para um elemento com os filhos dispostos em linha. A altura do elemento <g> é dada pela altura máxima dos seus filhos. A largura é a diferença entre, a soma do posicionamento horizontal do último filho com a largura do mesmo, e o posicionamento horizontal do primeiro filho. Tanto o posicionamento horizontal como vertical são iguais aos do primeiro filho.

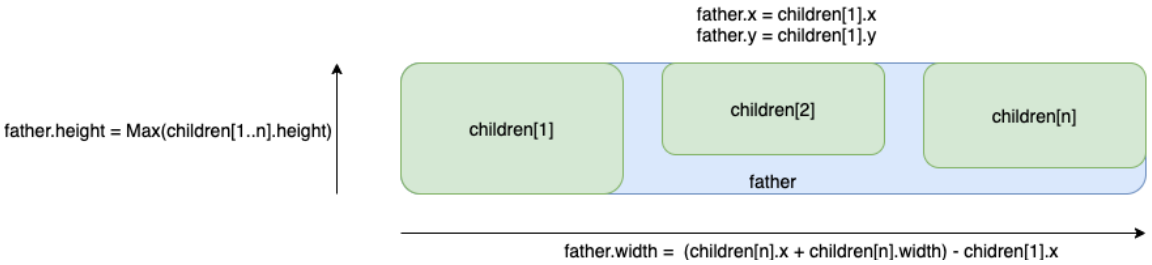


Figura 23: Calculo das dimensões com disposição em linha

Já a Figura 24, mostra como seria o cálculo para um elemento com os filhos dispostos em coluna. A largura do elemento <g> é dada pela largura máxima dos seus filhos. A altura é a diferença entre, a soma do posicionamento vertical do último filho com a altura do mesmo, e o posicionamento do vertical do primeiro filho. Tanto o posicionamento horizontal como vertical são iguais aos do primeiro filho.

Com a abordagem das *keywords*, é necessário apenas deduzir as margens de qualquer elemento para que o estilo gerado para o componente o faça ficar igual ao protótipo do *Figma*. Apenas o elemento <text>

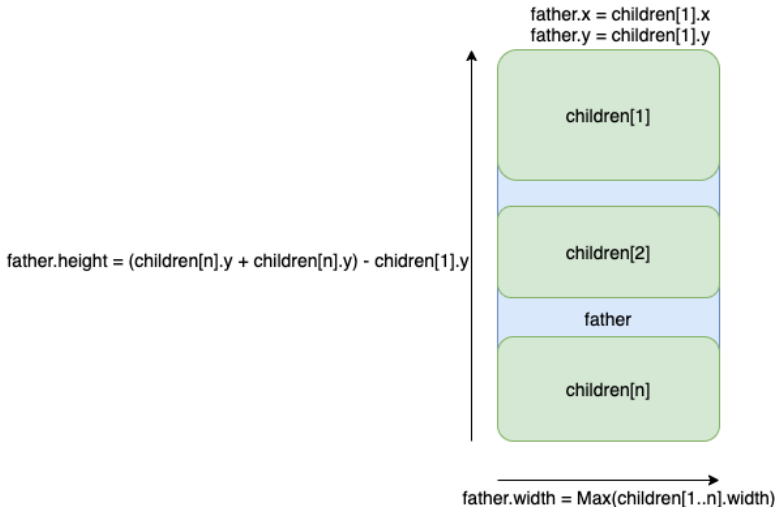


Figura 24: Calculo das dimensões com disposição em coluna

não possibilita a dedução das suas margens. Além disso, também não é possível calcular as dimensões de um elemento <g> que contenha um elemento <text>. Isto porque os elementos <text> não possuem a sua altura nem a sua largura, como podemos ver na Listagem 5.18. Como o elemento C não possui nem altura, nem largura, é impossível calcular a margem entre os elementos B e C pela fórmula que vimos anteriormente. Também não é possível calcular a dimensões do elemento A, pelas mesmas razões.

```
1 <g id="email row display-flex align-center"> // elemento A
2   <rect x="47" y="92" width="280" height="27" rx="4" fill="white"/> //
      elemento B
3   <text id="EMAIL ADRESS" fill="#A9B5A8" xml:space="preserve" style="white-
      space: pre" font-family="Open Sans" font-size="14" letter-spacing="0em
      "><tspan x="57" y="110.931">EMAIL ADRESS</tspan></text> // elemento C
4 </g>
```

Listagem 5.18: Elemento com um campo de texto

É necessário implementar uma abordagem capaz de deduzir a altura e a largura de um elemento <text>, para que os cálculos definidos anteriormente resultem com este tipo de elementos. A altura de um elemento <text> pode ser dada pelo seu *font-size*. O *font-size* de um texto representa a altura do mesmo desde o ponto mais baixo de um caractere descendente, até ao ponto mais alto de um caractere ascendente. Caracteres descendentes são aqueles cuja altura cresce no sentido descendente, por exemplo, as letras “g”, “q”, ou “y”. Os caracteres ascendentes são aqueles cuja altura cresce no sentido ascendente, por exemplo, as letras “N”, “h”, ou “b”. Assim a altura de uma fonte é dado pela distância entre o fundo de um caractere como a letra “g” e o topo de um caractere como a letra “N”[8]. Posto isto iremos utilizar a propriedade *font-size* como a altura de um elemento <text>.

[8]É possível encontrar mais informação sobre este tópico no seguinte fórum de *design* gráfico - https://graphicdesign.stackexchange.com/questions/4035/what-does-the-size-of-the-font-translate-to-exactly/8964#8964 - visitado em 20/12/2021

A largura de um campo de texto irá depender dos caracteres presentes no texto, e do seu tamanho. Quanto mais caracteres, e maior for o tamanho da fonte, maior a largura de um texto. Para além disso, a largura também difere de caractere para caractere, por exemplo, o caractere "i" é menos largo do que o caractere "A". E por último diferentes tipos de fontes também vão possuir diferentes larguras de caracteres. Ou seja, conseguimos calcular a largura de um campo de texto se soubermos o seu tipo de fonte, o valor do campo de texto, e a largura de cada caractere daquele tipo de fonte. Esta abordagem já é utilizada pela biblioteca *string-pixel-width*[9], que possui suporte para um conjunto limitado de fontes. Esta biblioteca calcula a largura de um campo de texto com base em três parâmetros, o tamanho do texto (font-size), a fonte (font-family), e o valor do mesmo. Esta biblioteca irá ser utilizada para definir a largura de um elemento <text> com base nas propriedades *font-size*, *font-family*, e o valor do elemento <tspan> contido em <text>.

Assim, estão ultrapassados os principais entraves que não iriam permitir que o nosso componente *React* ficasse igual ao protótipo criado no *Figma*.

5.4 Sumário

Neste capítulo é descrita a ferramenta elaborada visando aplicar os conceitos discutidos nos capítulos anteriores e sistematizar o processo de implementação de interfaces. Na Secção 5.1 são descritas as funcionalidades e os seus objetivos.

A seguir, na Secção 5.2, descreve-se em particular a funcionalidade de geração de código com base em comandos. É descrito o funcionamento, a implementação, e a utilidade de cada comando disponível.

Por último, na Secção 5.3, aborda-se a implementação da funcionalidade de geração de código com base em protótipos. São descritos todos os momentos da geração, desde a exportação do protótipo para um ficheiro SVG, passando pela geração do código HTML e dos estilos, até à criação de um componente *React*. Ao longo desta Secção são ainda discutidas as limitações encontradas neste processo de geração e a soluções utilizadas para as mitigar. Também são explicados todos os algoritmos criados para conseguir o objetivo de gerar um componente *React* com base num protótipo de interface.

[9] https://github.com/adambisek/string-pixel-width

6

Exemplo prático

Neste capítulo irá ser apresentado um caso de estudo prático para que seja possível demonstrar a utilização da ferramenta de geração de código desenvolvida durante a dissertação. O objetivo será implementar uma interface, especificada durante este capítulo, com recurso às funcionalidades da ferramenta. No final reflete-se sobre a utilidade e a eficácia da ferramenta com base nos resultados obtidos e no esforço de implementação.

6.1 Especificação do exemplo

A interface que iremos implementar terá o objetivo de funcionar como um portal para enviar *emails*. Este projeto terá como *homepage* um formulário simples com as informações do *email* a enviar. Podemos consultar o *design* da *homepage* na Figura 18 (ver página 54). Neste *design* possuímos diversas camadas, presentes também na Figura 18. A primeira, especifica o nome do componente, e também o alinhamento em coluna e ao centro dos seus elementos aninhados. A seguir, possuímos um título e um formulário. O título são duas camadas em coluna, uma com o texto e outra com uma linha horizontal. O formulário está dividido em três linhas, as primeiras duas com duas colunas, e a última apenas com uma coluna. Cada coluna possui um campo de texto com fundo branco.

Para além da *homepage*, a nossa interface possuirá uma página de entrada onde os utilizadores irão introduzir um nome e uma senha de maneira a conseguirem aceder à página principal. O *design* desta página pode ser consultado na Figura 25. Na Figura 26 é possível observar todas as camadas e *keywords* utilizadas neste componente. Possuímos uma primeira camada de *auto-layout* com o nome do componente e as *keywords*, "col", "align-center", e "justify-center", que vão dispor o conteúdo do componente em coluna, e alinhado ao centro, verticalmente e horizontalmente. De seguida, temos duas camadas de *auto-layout* uma para o botão e outra para a do formulário de *login*. A camada do formulário possui ainda duas camadas aninhadas, uma para cada campo de texto. Os campos de texto e o botão são os elementos mais atómicos, representados, cada um, por uma camada de *auto-layout* e uma camada de texto.

De realçar que os *designs*, para além de possuírem o alinhamento apropriado, fazem também uso

Figura 25: Componente de login

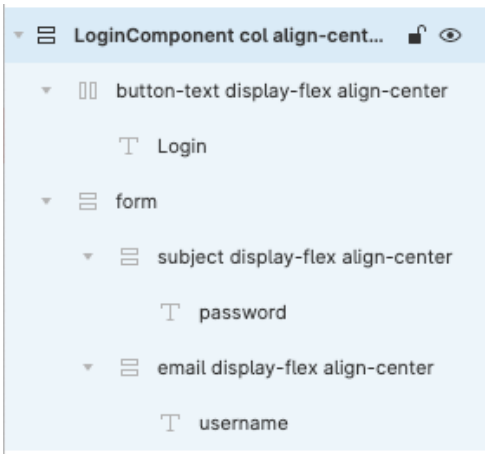


Figura 26: Auto-layouts LoginComponent

das *keywords* no nome das camadas para descrever esse alinhamento e fazer passar à ferramenta de geração de código essa informação.

Para terminar, em termos de navegação a nossa aplicação terá um fluxo simples. Os utilizadores quando clicarem no botão "Login", da Figura 25, irão ser redirecionados para a *homepage* presente na Figura 18 (ver página 54).

6.2 Implementação

O objetivo desta implementação é então criar um novo projeto para a aplicação e implementar as diferentes rotas, componentes, e páginas.

Primeiro vamos criar um projeto com o comando:

- npx create-react-app exemplo-pratico

A seguir geramos a estrutura de pastas idealizada na Secção 3.2, recorrendo ao comando:

- npm run create-folder-structure exemplo-pratico

Agora que o projeto possui a estrutura e arquitetura idealizada, vamos começar a implementar a rota `/home`, correspondente à *homepage*. Para isso usamos o comando:

- npm run create-route exemplo-pratico Home Card

Onde "Card" irá ser o componente criado para definir o formulário de envio de *email* (este componente será criado, de seguida, com recurso ao protótipo da Figura 18, ver página 54). Como resultado deste comando é gerado um componente na pasta `/pages` com o nome `Home`, presente na Listagem 6.1.

```
1  import React from 'react'
2  import './Home.sass'
3  import Card from 'components/Card'
4
5  const Home = (props) => {
6    return (
7      <Card />
8    )
9  }
10
11 export default Home
```

Listagem 6.1: Ficheiro Home.js

Repetimos o mesmo processo para criar a página de entrada (Figura 25), com o comando:

- npm run create-route exemplo-pratico Login LoginComponent

Neste momento, falta apenas criar os componentes `Card` e `LoginComponent`. Para isso é necessário exportar os respetivos protótipos do *Figma* para ficheiros SVG. Após isso, obtemos os ficheiros presentes nos Anexos I e III. A geração destes dois componentes é análoga, pelo que, iremos apenas descrever a geração do componente `Card`.

Com o ficheiro SVG já é possível gerar o componente, através do comando:

- npm run generate-component exemplo_pratico <path_to_svg>

Como resultado, iremos obter a pasta `/Card`, presente na Figura 27, dentro de `/src/components`. São criados os mesmos ficheiros da Secção 5.2.2, mas neste caso, no conteúdo dos ficheiros `Card.js`, e `Card.sass` já permanecem, o código *React*, e as regras de estilo, respetivamente.

No ficheiro `Card.js`, foi gerado o código para o componente *React*, o resultado pode ser consultado na Listagem 6.2.

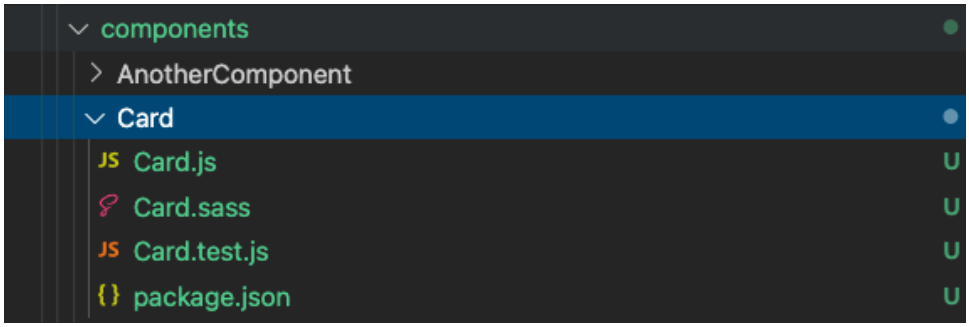


Figura 27: Pasta criada após a geração do componente

```
import React from 'react'
import './Card.sass'

const Card = (props) => {
  return (
    <div className='Card col align-center justify-center'>
      <div className='title col align-center'>
        <div className='title-text'>
          <div className='text-field'>
            SEND ME A MESSAGE!
          </div>
        </div>
        <div className='title-line'>
          <div className='rec'>

          </div>
        </div>
      </div>
      <div className='form'>
        <div className='row-2 row'>
          <div className='email display-flex align-center'>
            <div className='text-field'>
              EMAIL ADRESS
            </div>
          </div>
          <div className='name display-flex align-center'>
            <div className='text-field'>
              NAME
            </div>
          </div>
        </div>
        <div className='row-1 row'>
          <div className='subject display-flex align-center'>
            <div className='text-field'>
```

```
35              SUBJECT
36            </div>
37          </div>
38          <div className='company display-flex align-center'>
39            <div className='text-field'>
40              COMPANY/BRAND
41            </div>
42          </div>
43        </div>
44        <div className='message display-flex'>
45          <div className='text-field'>
46            MESSAGE
47          </div>
48        </div>
49      </div>
50    </div>
51  )
52 }
53
54 export default Card
```

Listagem 6.2: Ficheiro Card.js

Conseguimos observar que todos os elementos presentes no SVG foram representados neste componente, e que foi possível gerar o seu respetivo código HTML. Também conseguimos perceber que a passagem do nome das camadas no *Figma* para o nome das classes em HTML foi um sucesso e agora conseguimos definir as regras de estilo sobre estas classes.

O ficheiro `Card.sass`, presente no Anexo II, contém todas as regras de estilo para o componente. O algoritmo de geração dos estilos foi um sucesso e conseguimos definir para cada elemento HTML os seus espaçamentos, posicionamentos, cores e outras propriedades presentes no protótipo. Com estes ficheiros o componente fica totalmente representado e pronto para ser utilizado em qualquer parte da aplicação. Inclusive na rota `/home`, onde foi incluído anteriormente.

Neste momento se inicializarmos o servidor *React*, e formos ao endereço `localhost:3000/home`, obtemos na página o componente gerado a partir do protótipo como podemos ver na Figura 28.

Para além da execução dos comandos, é necessário instalar algumas dependências para que tudo funcione em harmonia. Temos de instalar a biblioteca *react-router-dom*, para que as rotas funcionem. Necessitamos da biblioteca *node-sass* para conseguir utilizar *Sass*, um pré-compilador de CSS. Por último, precisamos de instalar a fonte "Open Sans", da *GoogleFonts*, usada no protótipo criado. Estes passos teriam de ser feitos independentemente da utilização ou não utilização das ferramentas implementadas.

Conceitos como a navegação entre rotas são facilmente implementados a partir do código gerado. Após gerarmos o componente `LoginComponent`, correspondente ao protótipo da Figura 25, e criarmos a rota `/login` com esse componente, podemos introduzir a navegação pretendida para a nossa aplicação

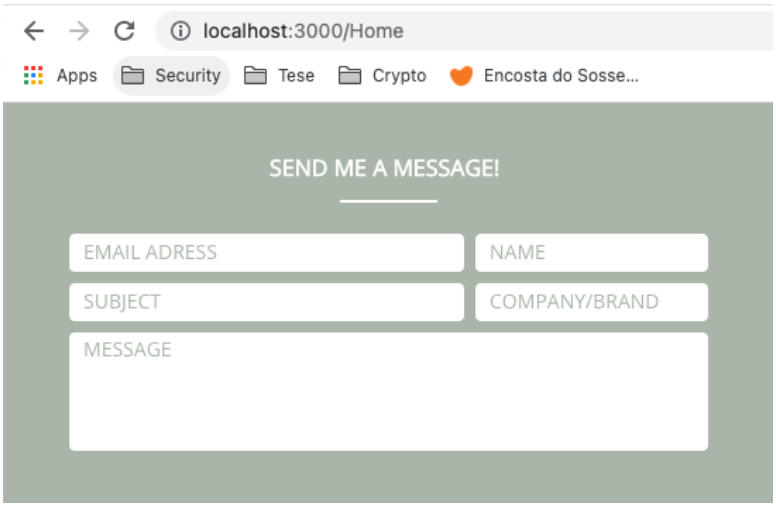


Figura 28: Resultado da geração do componente

com muito pouco esforço de implementação. Para isso é apenas necessário utilizar uma função da biblioteca *react-router-dom*. O código da Listagem 6.3 mostra o resultado do ficheiro `LoginComponent.js` após ter sido implementada esta navegação.

```
1  import React from 'react'
2  import './LoginComponent.sass'
3  import { useHistory } from "react-router-dom";
4
5  const LoginComponent = (props) => {
6    const history = useHistory();
7
8    return (
9      <div className='LoginComponent col align-center justify-center'>
10       <div className='form'>
11         <div className='email display-flex align-center'>
12           <div className='text-field'>
13             username
14           </div>
15         </div>
16         <div className='subject display-flex align-center'>
17           <div className='text-field'>
18             password
19           </div>
20         </div>
21       </div>
22       <div className='button-text display-flex align-center' onClick={() =>
             history.push("/home")}>
23         <div className='text-field'>
24           Login
25         </div>
```

```
26          </div>
27        </div>
28      )
29  }
30
31  export default LoginComponent
```

Listagem 6.3: Ficheiro LoginComponent.js com navegação para a rota home

Na linha 6 é definido um *hook* da biblioteca *react-router-dom* que nos permite, entre outras coisas, navegar para diferentes rotas da nossa aplicação. Na linha 22 definimos, através do evento de clique (função `onClick`), que o utilizador será redirecionado para a rota `/home`. Para redirecionar é utilizado o *hook* definido na linha 6. Tudo o resto é código gerado pela ferramenta.

6.3 Refleção

Em suma, neste exemplo, com apenas seis comandos foi criado um projeto *React* de raiz, adaptada a sua estrutura de pastas, inseridas duas rotas web, e gerados dois componentes através de protótipos elaborados no *Figma*. O tempo de execução destes seis comandos esta na ordem dos segundos. Ou seja, em muito pouco tempo foram feitas várias ações, que caso tivessem sido implementadas manualmente levariam consideravelmente mais tempo ao programador. Apenas para criar o componente *React* o programador teria de olhar para o protótipo e escrever cada linha de código, e cada elemento HTML desse componente. No total foram geradas 54 linhas de código, para produzir o ficheiro do Anexo I.

Para além do tempo, esta abordagem automatiza a criação de ficheiros, e de rotas, respeitando a arquitetura e estrutura pensadas inicialmente. Desta forma garantimos que o projeto evolui da forma projetada, e que todos os nossos ficheiros permanecem organizados no seu respetivo local. O crescimento do nosso produto de software desta forma é muito mais controlado e garantimos que existe uma sistematização na adição de novos componente e de novas rotas.

É verdade que os componentes gerados são meramente estáticos, mas a sua criação pode servir de ponto de partida para os programadores. Ajudando na sistematização do processo de implementação de uma interface. Vimos que também que com muito pouco esforço foi possível introduzir navegação entre diferentes rotas geradas pela ferramenta, reforçando a utilidade do código gerado, e a semelhança com uma possível solução final.

No âmbito da funcionalidade de geração através de protótipos, é possível afirmar que o protótipo do *Figma* da Figura 18, e o componente gerado em *React* na Figura 28 estão praticamente iguais. Por isso, podemos afirmar que se cumpriu o propósito inicial de gerar um componente *React* com apenas um protótipo e um comando, sem qualquer necessidade de escrever código e numa questão de segundos. Isto aumenta a velocidade do processo e a sistematização do mesmo quando comparado com uma implementação manual.

6.4 Sumário

Neste capítulo, foram utilizadas na prática as funcionalidades da ferramenta implementada durante a dissertação, com o objetivo de implementar uma interface.

Na Secção 6.1 foi especificada a interface que iria ser implementada, incluindo o design das suas páginas, as suas rotas, e a navegação entre elas.

A seguir, na Secção 6.2, é descrito todos os passos da implementação, os comandos usados e os efeitos dos mesmos no projeto.

Por último, na Secção 6.3 refletiu-se sobre o resultado produzido pela ferramenta, e a sua eficácia. Conseguiu-se perceber que alguns dos comandos nos ajudam a cumprir a arquitetura e estrutura pensadas no Capítulo 3. Além disso, verificou-se que os componentes gerados são idênticos aos protótipos que estão na sua origem, poupando assim linhas de código e tempo aos programadores.

7

Conclusão

Nesta dissertação foi abordado o problema da falta de sistematização no processo de desenvolvimento de interfaces por parte dos programadores. No presente capítulo iremos concluir sobre todo o trabalho realizado, e expor possíveis tópicos de trabalho futuro.

7.1 Trabalho realizado

Foi proposta a criação e análise de alguns mecanismos para combater este problema e tornar mais sistemático, e consequentemente mais eficaz, o processo de implementação de uma interface. Para isso, foram analisadas diversas partes do processo de implementação, tais como, a análise do *design*, a estruturação e arquitetura de um produto de software, a criação de componentes, e a criação de estilos. Em cada uma destas partes foi especificado um método de execução capaz de guiar o programador do início ao fim de cada tópico.

Na análise do *design* conseguiu-se descrever um processo que visa permitir que os programadores consigam analisar um protótipo de uma interface e extrair dele os componentes que terão de implementar, maximizando a reutilização de componentes presentes em mais do que uma parte do protótipo.

Foi criada uma arquitetura genérica de uma aplicação *React* que visa permitir escalar o nosso produto de software de uma forma organizada e manutenível. Foram descritas várias entidades dentro desta arquitetura, todas elas com funções distintas. Esta arquitetura permitiu dividir responsabilidades, fazendo com que seja muito mais fácil saber o que determinado elemento, ou ficheiro de código, faz dentro de uma interface. Além disso, a criação de uma estrutura de pastas garante o suporte e a divisão de todos os diferentes tipos de elementos presentes na arquitetura de uma interface web, o que permite ao projeto escalar em número de ficheiros de código de forma organizada.

Na criação de componentes foi descrito o processo de implementação de componentes em *React*, seguindo as mais recentes recomendações da *framework*. Foi utilizado o conceito de *self-contained component*, que visa a criação de componentes independentes e desacoplados, capazes de serem reutilizados em qualquer parte do software. Este conceito maximiza a reutilização de código e aumenta a produtividade, isto porque reduzimos a quantidade de trabalho repetido presente na implementação.

Foram analisadas várias formas de implementação de estilo nas interfaces web. Concluiu-se que usar o estilo acoplado ao componente é uma boa prática, para não haver conflitos de estilo, o que permite fazer alterações no aspeto de um componente com a segurança de que não alteramos mais nada na restante interface. Em termos de tecnologias, *CSS in JS* foi das opções analisadas aquela com mais vantagens. A sua flexibilidade, devido ao facto de ser programável, fornece às nossas aplicações um estilo mais dinâmico e dependente do estado dos componentes.

Foram criadas duas ferramentas, no âmbito da dissertação, que permitem automatizar algumas partes do processo de implementação de interfaces. Estas automatizações tornam a implementação mais sistemática e menos suscetível a erros ou desvios daquilo que é a arquitetura pensada inicialmente.

Com os comandos implementados conseguimos montar um projeto *React* com a arquitetura idealizada. Os comandos das ferramentas conseguiram gerar componentes, ficheiros de estilo, e páginas *web*. Com isto, podemos poupar, aos programadores, dezenas de linhas por ficheiro gerado.

7.2 Trabalho Futuro

Com o crescimento dos projetos de *software*, o estado dos componentes tende a aumentar, em tamanho e em complexidade. Para organizar, manter e controlar o estado dos componentes, em *React*, são utilizadas bibliotecas como o *Redux*[1]. Existem varias abordagens e padrões na utilização destas bibliotecas, estas abordagens poderiam ser analisadas e sistematizadas como continuação do trabalho iniciado nesta dissertação.

As ferramentas elaboradas, no futuro, poderiam ser melhoradas. Conceitos como a geração, através de comandos, de componentes com estado, ou a geração de outras entidades presentes na arquitetura genérica, como serviços, e testes poderiam também ser automatizados.

A geração de componentes com base em protótipos também poderia ser melhorada. Pode ser introduzido, com facilidade, suporte a novos tipos de elementos HTML, como imagens, ícones, etc. A noção de estado de um componente, por exemplo, valores de campos de texto, também pode ser introduzida. Se quisermos gerar um componente com estado podemos tentar representar esse estado nos protótipos e consequentemente exportar o mesmo para o ficheiro SVG, a partir daí conseguiríamos refletir o estado no respetivo componente gerado.

A procura por uma parceria com um *software* de desenvolvimento destes protótipos, por exemplo, o *Figma*, seria interessante, isto porque poderíamos ter um acesso privilegiado à informação das várias camadas do protótipo. O que permitia ultrapassar todas as limitações da geração de componentes através de ficheiros SVG, e gerar componentes ainda mais próximos do seu estado final.

Seria também interessante avaliar as ferramentas com base na opinião de diversos programadores, isto é, fazer com que programadores utilizassem as ferramentas e as avaliassem em termos de utilidade, eficácia, usabilidade, etc.

[1] https://redux.js.org/ - visitado em 20/12/2021

Bibliografia

Blackmon, M. H., Polson, P. G., Kitajima, M. & Lewis, C. (2002). Cognitive Walkthrough for the Web. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 463–470. https://doi.org/10.1145/503376.503459 (ver p. 4)

Britch, D. (2017). *Enterprise Application Patterns using Xamarin.Forms*. DevDiv, .NET; Visual Studio product teams. (Ver p. 2).

Calvary, G., Coutaz, J. & Thevenin, D. (2001). A Unifying Reference Framework for the Development of Plastic User Interfaces, 173–192. https://doi.org/10.1007/3-540-45348-2_17 (ver p. 5)

Frost, B. (2016). *Atomic design*. Brad Frost Pittsburgh. (Ver p. 12).

Galitz, W. O. (2007). *The Essential Guide to User Interface Design* (Third). Wiley. (Ver p. 4).

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. (Ver pp. 3, 28).

Hailpern, B. & Tarr, P. (2006). Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, *45*(3), 451–461. https://doi.org/10.1147/sj.453.0451 (ver pp. 1, 4)

ISO13407. (1999). *Human-centred design processes for interactive system teams* (ISO 13407:1999). International Organization for Standardization. Geneva, Switzerland. (Ver p. 1).

John, B. E., Bass, L. & Adams, R. J. (2003). Communication across the HCI/SE divide: ISO 13407 and the Rational Unified Process. *Proceedings of HCI International* (ver p. 1).

Krasner, G. E., Pope, S. T. et al. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, *1*(3), 26–49 (ver p. 2).

Martins, R., Caldeira, F., Sá, F., Abbasi, M. & Martins, P. (2020). An overview on how to develop a low-code application using OutSystems. *2020 International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*, 395–401. https://doi.org/10.1109/ICSTCEE49637.2020.9277404 (ver p. 2)

Meixner, G. & Calvary, G. (2014). *Introduction to Model-Based User Interfaces* (W3C Note). W3C. https://www.w3.org/TR/2014/NOTE-mbui-intro-20140107/. (Ver pp. 1, 5)

Nielsen, J. & Molich, R. (1990). Heuristic Evaluation of User Interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 249–256. https://doi.org/10.1145/97243.97281 (ver p. 4)

Nierstrasz, O., Gibbs, S. & Tsichritzis, D. (1993). Component-Oriented Software Development. *Communications of the ACM*, *35*. https://doi.org/10.1145/130994.131005 (ver p. 11)

Pinheiro, P. (2001). User Interface Declarative Models and Development Environments: A Survey, 207–226. https://doi.org/10.1007/3-540-44675-3_13 (ver p. 6)

Potel, M. (1996). MVP: Model-View-Presenter the Taligent programming model for C++ and Java. *Taligent Inc*, 20 (ver p. 2).

Ritter, F. E., Baxter, G. D. & Churchill, E. F. (2014). *Foundations for Designing User-Centered Systems* (First). Springer. https://doi.org/10.1007/978-1-4471-5134-0. (Ver p. 3)

Szekely, P. (1996). *Retrospective and Challenges for Model-Based Interface Development*. Springer, Vienna. https://doi.org/10.1007/978-3-7091-7491-3_1. (Ver p. 6)

Vredenburg, K., Mao, J.-Y., Smith, P. W. & Carey, T. (2002). A survey of user-centered design practice. *Proceedings of the SIGCHI conference on Human factors in computing systems*, 471–478 (ver p. 4).

Wohlgethan, E. (2018). *Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js* (tese de bacharelato). Hochschule für Angewandte Wissenschaften Hamburg. (Ver p. 8).

I

Ficheiro SVG do componente da Figura 18

```
<svg width="547" height="285" viewBox="0 0 547 285" fill="none" xmlns="http://www.w3.org/2000/svg">
<rect width="547" height="285" fill="#E5E5E5"/>
<g id="Card col align-center justify-center">
  <rect width="547" height="285" fill="#A9B5A8"/>
  <g id="title col align-center">
    <g id="title-text">
      <text id="SEND ME A MESSAGE!" fill="white" xml:space="preserve" style="white-space: pre" font-family="Open Sans" font-size="16" font-weight="bold" letter-spacing="0em"><tspan x="189.453" y="53.207">SEND ME A MESSAGE!</tspan></text>
    </g>
    <g id="title-line">
      <rect id="rec" x="239" y="68" width="69" height="2" rx="1" fill="white"/>
    </g>
  </g>
  <g id="form">
    <g id="row-2 row">
      <g id="email display-flex align-center">
        <rect x="47" y="92" width="280" height="27" rx="4" fill="white"/>
        <text id="EMAIL ADRESS" fill="#A9B5A8" xml:space="preserve" style="white-space: pre" font-family="Open Sans" font-size="14" letter-spacing="0em"><tspan x="57" y="110.931">EMAIL ADRESS</tspan></text>
      </g>
      <g id="name display-flex align-center">
        <rect x="335" y="92" width="165" height="27" rx="4" fill="white"/>
        <text id="NAME" fill="#A9B5A8" xml:space="preserve" style="white-space: pre" font-family="Open Sans" font-size="14" letter-spacing="0em"><tspan x="345" y="110.931">NAME</tspan></text>
      </g>
```

```
    </g>
    <g id="row-1 row">
      <g id="subject display-flex align-center">
        <rect x="47" y="127" width="280" height="27" rx="4" fill="white"/>
        <text id="SUBJECT" fill="#A9B5A8" xml:space="preserve" style="white
            -space: pre" font-family="Open Sans" font-size="14" letter-
            spacing="0em"><tspan x="57" y="145.931">SUBJECT</tspan></text>
      </g>
      <g id="company display-flex align-center">
        <rect x="335" y="127" width="165" height="27" rx="4" fill="white"/>
        <text id="COMPANY/BRAND" fill="#A9B5A8" xml:space="preserve" style=
            "white-space: pre" font-family="Open Sans" font-size="14" letter
            -spacing="0em"><tspan x="345" y="145.931">COMPANY/BRAND</tspan
            ></text>
      </g>
    </g>
    <g id="message display-flex">
      <rect x="47" y="162" width="453" height="84" rx="4" fill="white"/>
      <text id="MESSAGE" fill="#A9B5A8" xml:space="preserve" style="white-
          space: pre" font-family="Open Sans" font-size="14" letter-spacing=
          "0em"><tspan x="57" y="181.931">MESSAGE</tspan></text>
    </g>
  </g>
</g>
</svg>
```

II

Ficheiro Card.sass

```
.Card.col.align-center.justify-center
  width: 547px
  height: 285px
  background-color: #A9B5A8
  display: flex
  flex-direction: column
  align-items: center
  justify-content: center
  .title.col.align-center
    display: flex
    flex-direction: column
    align-items: center
    height: 32.793px
    width: 168.84px
    margin-bottom: 22px
    .title-text
      width: 168.84px
      height: 16px
      margin-bottom: 14.793px
      .text-field
        color: white
        font-family: 'Open Sans', sans-serif
        font-size: 16px
        font-weight: bold
        width: 168.84px
        height: 16px
        line-height: 16px
        margin-top: -0.20700000000000074px

    .title-line
      width: 69px
      height: 2px
```

```
33      border-radius: 1px
34      background-color: white
35      .rec
36        width: 69px
37        height: 2px
38        border-radius: 1px
39        background-color: white
40
41  .form
42    height: 154px
43    width: 453px
44    .row-2.row
45      display: flex
46      flex-direction: row
47      height: 27px
48      width: 453px
49      margin-bottom: 8px
50      .email.display-flex.align-center
51        width: 280px
52        height: 27px
53        border-radius: 4px
54        background-color: white
55        align-items: center
56        display: flex
57        margin-right: 8px
58        .text-field
59          color: #A9B5A8
60          font-family: 'Open Sans', sans-serif
61          font-size: 14px
62          width: 96.06px
63          height: 14px
64          line-height: 14px
65          margin-left: 10px
66
67      .name.display-flex.align-center
68        width: 165px
69        height: 27px
70        border-radius: 4px
71        background-color: white
72        align-items: center
73        display: flex
74        .text-field
75          color: #A9B5A8
76          font-family: 'Open Sans', sans-serif
77          font-size: 14px
```

```
78          width: 40.760000000000005px
79          height: 14px
80          line-height: 14px
81          margin-left: 10px
82
83    .row-1.row
84      display: flex
85      flex-direction: row
86      height: 27px
87      width: 453px
88      margin-bottom: 8px
89      .subject.display-flex.align-center
90        width: 280px
91        height: 27px
92        border-radius: 4px
93        background-color: white
94        align-items: center
95        display: flex
96        margin-right: 8px
97        .text-field
98          color: #A9B5A8
99          font-family: 'Open Sans', sans-serif
100         font-size: 14px
101         width: 56.160000000000004px
102         height: 14px
103         line-height: 14px
104         margin-left: 10px
105
106     .company.display-flex.align-center
107       width: 165px
108       height: 27px
109       border-radius: 4px
110       background-color: white
111       align-items: center
112       display: flex
113       .text-field
114         color: #A9B5A8
115         font-family: 'Open Sans', sans-serif
116         font-size: 14px
117         width: 121.4px
118         height: 14px
119         line-height: 14px
120         margin-left: 10px
121
122   .message.display-flex
```

```
123       width: 453px
124       height: 84px
125       border-radius: 4px
126       background-color: white
127       display: flex
128       .text-field
129         color: #A9B5A8
130         font-family: 'Open Sans', sans-serif
131         font-size: 14px
132         width: 63.720000000000006px
133         height: 14px
134         line-height: 14px
135         margin-left: 10px
136         margin-top: 5px
```

III

Ficheiro SVG do componente da Figura 25

```
<svg width="374" height="285" viewBox="0 0 374 285" fill="none" xmlns="http://www.w3.org/2000/svg">
  <rect width="374" height="285" fill="#E5E5E5"/>
  <g id="LoginComponent col align-center justify-center">
    <rect width="374" height="285" fill="#A9B5A8"/>
    <g id="form">
      <g id="email display-flex align-center">
        <rect x="47" y="84.5" width="280" height="27" rx="4" fill="white"/>
        <text id="username" fill="#A9B5A8" xml:space="preserve" style="white-space: pre" font-family="Open Sans, sans-serif" font-size="14" letter-spacing="0em"><tspan x="57" y="103.431">username</tspan></text>
      </g>
      <g id="subject display-flex align-center">
        <rect x="47" y="119.5" width="280" height="27" rx="4" fill="white" />
        <text id="password" fill="#A9B5A8" xml:space="preserve" style="white-space: pre" font-family="Open Sans, sans-serif" font-size="14" letter-spacing="0em"><tspan x="57" y="138.431">password</tspan></text>
      </g>
    </g>
    <g id="button-text display-flex align-center">
      <rect x="47" y="176.5" width="280" height="24" rx="4" fill="white"/>
      <text id="Login" fill="#A9B5A8" xml:space="preserve" style="white-space: pre" font-family="Open Sans, sans-serif" font-size="16" font-weight="bold" letter-spacing="0em"><tspan x="165.297" y="194.707">Login</tspan></text>
    </g>
  </g>
</svg>
```