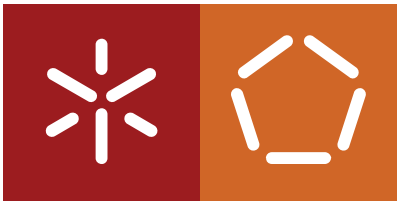
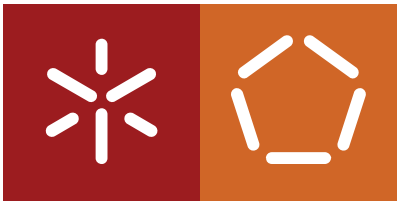


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Catarina Araújo Machado

Geração Automática de Interfaces de Utilizador para Aplicações Web

December 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Catarina Araújo Machado

Geração Automática de Interfaces de Utilizador para Aplicações Web

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
José Creissac Campos

December 2021

AGRADECIMENTOS

Dado por terminado a presente dissertação e, consequentemente, o meu Mestrado Integrado em Engenharia Informática, sinto o dever de endereçar algumas palavras de apreço a todos aqueles que contribuíram de alguma forma para esta bonita caminhada.

Começo por agradecer ao responsável por hoje estar a escrever os agradecimentos da minha dissertação, o meu orientador, Professor Doutor José Creissac Campos, por ter sido um apoio incansável durante o processo de desenvolvimento da presente dissertação. É de louvar toda a sua persistência, conhecimento e disponibilidade ao longo deste último ano. Tenho a certeza que seria impossível ter tido um melhor orientador. Um obrigada especial também ao INESC TEC pela oportunidade do desenvolvimento de uma bolsa de investigação acerca deste tema.

Em segundo lugar, quero agradecer à minha mãe - o amor da minha vida, ao meu pai, ao meu irmão - o meu ídolo, e à minha cunhada, por serem o meu maior pilar. Sem vocês, não teria sempre a perspetiva de que tudo vai correr bem e que tudo vai dar sempre certo. Obrigada por nunca me fazerem esquecer disso e por estarem disponíveis para me ouvirem, para me motivarem e para me desafiarem em literalmente todos os momentos da minha vida.

Em terceiro lugar, quero agradecer a toda a minha família - aos meus avós, aos meus tios e tias, primos e primas por serem muito mais do que família. Todos vocês são os meus melhores amigos. Obrigada por me motivarem a ser uma pessoa sempre melhor.

Por fim, quero agradecer aos amigos que fui fazendo nos diversos projetos associativos em que estive envolvida, tanto dentro como fora do curso, por terem tornado o meu percurso académico mais enriquecedor e bem mais desafiante. Estarei eternamente grata por tudo o que construímos juntos.

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.

ABSTRACT

The main objective of this dissertation is to make a contribution in the automation of web applications' development, starting from prototypes of their graphical user interfaces.

The integration of model-based user interface development concepts with the more traditional user-centred development approach allows for a rethinking of GUI design development, independent of implementation details, and redefining models to realize these graphical interfaces. In the end, the intent is to increase the level of abstraction of the development process, promote better adaptation of applications to different devices and execution environments, and decrease the effort required to develop the graphical interfaces.

Due to the exponential increase in the use of internet-based services and applications, there is an also increasing demand for Web designers and developers. At the same time, the proliferation of languages, frameworks and libraries illustrates the current state of immaturity of web development technologies. This state of affairs creates difficulties in the development and maintenance of Web applications.

An approach is presented that allows designers to use prototyping tools, in this case Adobe XD, to design graphical interfaces, and then automatically converts them to Vue.js + Bootstrap code, thus creating a first version of the implementation. This is done through the interpretation of the SVG file that Adobe XD exports.

The goal is not to produce the final version of the UI. Instead, we aim to produce a first version of the code, which can then be refined by the developer. This enables us to place less requirements on the prototype, regarding the amount of information that it must contain. In the end, we get a skeleton of Vue.js code that is easy to maintain and reuse to further improve the project.

KEYWORDS Model-Based User Interface Development, Web Development, Web Frameworks, Prototyping Tools, Automatic Code Generation.

RESUMO

O principal objetivo da presente dissertação é contribuir para a automatização do desenvolvimento de aplicações *web*, a partir de protótipos das interfaces gráficas de utilizador.

A integração de conceitos de desenvolvimento de interfaces de utilizador baseadas em modelos com a abordagem mais tradicional do desenvolvimento centrado no utilizador permite repensar o desenvolvimento do *design* da interface gráfica, independente dos detalhes de implementação, e redefinir modelos para concretizar essas interfaces gráficas. No fundo, o intuito é aumentar o nível de abstração do processo de desenvolvimento, promover uma melhor adaptação das aplicações aos diferentes dispositivos e ambientes de execução e diminuir o esforço necessário para desenvolver as interfaces gráficas.

Devido ao aumento exponencial da utilização de serviços e aplicações baseadas na *internet*, há uma procura crescente de *web designers* e programadores. Ao mesmo tempo, a proliferação de linguagens de programação, *frameworks* e bibliotecas ilustra o atual estado de imaturidade das tecnologias de desenvolvimento *web*. Este estado cria dificuldades no desenvolvimento e principalmente na manutenção de aplicações *web*.

É apresentada uma abordagem que permite aos *designers* utilizar ferramentas de prototipagem, neste caso foi escolhido Adobe XD, para desenhar interfaces gráficas e depois convertê-las automaticamente para código Vue.js + Bootstrap, criando assim uma primeira versão da implementação. Esta geração é feita através da interpretação do ficheiro SVG que o Adobe XD exporta.

O objetivo não é produzir a versão final da UI, mas sim produzir uma primeira versão do código que pode depois ser aperfeiçoada pelo programador. Este fator permite colocar menos requisitos no protótipo, relativamente à quantidade de informação que este deve conter. No final, obtém-se um esqueleto do código Vue.js, fácil de manter e de reaproveitar para melhorar cada vez mais o projeto.

PALAVRAS-CHAVE Desenvolvimento de Interfaces de Utilizador Baseado em Modelos, Desenvolvimento *Web*, Ferramentas de Prototipagem, *Frameworks Web*, Geração Automática de Código.

CONTEÚDO

Contents iii

1 INTRODUÇÃO 3
1.1 Contextualização . . . 3
1.2 Motivação . . . 4
1.3 Objetivos . . . 5
1.4 Estrutura do Documento . . . 5

2 DESENHO DE INTERFACES DE UTILIZADOR 7
2.1 Desenvolvimento de Interfaces de Utilizador baseado em modelos . . . 7
2.2 Desenvolvimento de Interfaces de Utilizador baseado em modelos vs. User-Centred Design . . . 9
2.3 Ferramentas de Mockups . . . 9
2.3.1 Figma . . . 11
2.3.2 Sketch . . . 12
2.3.3 Adobe XD . . . 13
2.4 Discussão das Ferramentas de Mockups . . . 13

3 DESENVOLVIMENTO DE APLICAÇÕES INTERATIVAS PARA A WEB 15
3.1 Contexto geral da área de Desenvolvimento Web . . . 15
3.1.1 Definição de Aplicação *Web* . . . 16
3.1.2 Definição de *Framework Web* . . . 17
3.1.3 Vantagens e desvantagens das *frameworks* . . . 17
3.1.4 *Client-side* e *Server-side* . . . 18
3.1.5 Arquiteturas de *software* . . . 18
3.2 Frameworks de Desenvolvimento Web . . . 21
3.2.1 Características das *Frameworks* de Desenvolvimento *Web* . . . 21
3.2.2 Análise das Características das *Frameworks* de Desenvolvimento *Web* . . . 24
3.2.3 Rankings de *Frameworks Web* . . . 27
3.3 Análise de Métricas e Frameworks Web . . . 29
3.3.1 Seleção de *Frameworks Web* . . . 30
3.3.2 Comparação de *Frameworks Web* . . . 32
3.4 Discussão . . . 34

4 ARQUITETURAS DE UMA APLICAÇÃO 35
4.1 Mockup Interface Gráfica . . . 35
4.2 Estrutura de um protótipo . . . 36

4.3 Metamodelo da Arquitetura de Componentes Vue.js . . . 37
4.4 Estrutura de pastas Vue.js . . . 39
4.4.1 Estrutura de pastas Vue CLI . . . 39
4.4.2 Estrutura de pastas padrão . . . 40
4.5 Código Vue.js . . . 42
4.6 Exemplificação . . . 44
4.7 Conclusão . . . 46
5 DESCRIÇÃO GENÉRICA DA ABORDAGEM DE IMPLEMENTAÇÃO DA UI 47
5.1 Conversão do Protótipo da Interface Gráfica . . . 47
5.2 Mapeamento entre os protótipos e o metamodelo . . . 48
5.2.1 Mapeamento de Componentes . . . 48
5.2.2 Mapeamento de Páginas . . . 50
5.2.3 Mapeamento do layout . . . 50
5.2.4 Resumo da Abordagem . . . 51
5.3 Requisitos de uma Ferramenta que suporte a Abordagem . . . 51
5.4 Sumário . . . 53
6 DETALHES DA IMPLEMENTAÇÃO DA UI 54
6.1 Tags SVG . . . 54
6.1.1 Texto . . . 55
6.1.2 Botão . . . 56
6.1.3 *Navbar* . . . 56
6.2 Script SVG - Configuração do projeto Vue.js . . . 57
6.2.1 Navegação entre páginas . . . 59
6.3 Parsing SVG - Geração de Páginas . . . 60
6.3.1 Bibliotecas utilizadas . . . 60
6.3.2 Estrutura de Dados . . . 61
6.3.3 Sistema de *grid* . . . 62
6.3.4 Esqueleto Vue.js . . . 63
6.3.5 Gerar Componentes . . . 67
6.3.6 Adicionar novo componente . . . 69
6.4 Sumário . . . 69
7 AVALIAÇÃO 70
7.1 O Protótipo . . . 70
7.2 Configuração do Protótipo . . . 72
7.3 Geração da Interface . . . 72
7.4 Discussão dos Resultados . . . 73
8 CONCLUSÃO E TRABALHO FUTURO 79

Anexos

A EXEMPLO ILUSTRATIVO ESTRUTURA DE DADOS "COMPONENTS" 87
B INICIO.SVG 89
C RESERVAS.SVG 92
D NAVBAR.VUE 95
E FOOTER.VUE 97
F INICIO.VUE 98
G RESERVAS.VUE 101
H CSS.CSS 104
I ESTRUTURA DE DADOS INICIAL - INICIO 107
J ESTRUTURA DE DADOS INICIAL - RESERVAS 112
K ESTRUTURA DE DADOS COMPLETA - INICIO 118
L ESTRUTURA DE DADOS COMPLETA - RESERVAS 126

LISTA DE ACRÓNIMOS

AJAX *Asynchronous JavaScript and XML*. 23
Amazon EC2 *Amazon Elastic Compute Cloud*. 32
API *Application Programming Interface*. 26, 40, 42
AUI *Abstract User Interface*. 8

CSS *Cascading Style Sheets*. 11–14, 34, 38, 40, 42, 43, 47, 67, 68, 72, 75, 80, 81
CUI *Concrete User Interface*. 8

DOM *Document Object Model*. 31

EPS *Encapsulated PostScript*. 12, 47
ES6 *ECMAScript 6*. 31

FUI *Final User Interface*. 8

GUI *Graphical User Interface*. c, 47, 48

H5BP HTML5 Boilerplate. 32
HTML *Hyper Text Markup Language*. 3, 11–16, 22, 24, 25, 32, 39, 42, 47, 77, 80, 81

ICGI *International Conference on Graphics and Interaction*. 5
IDE *Integrated Development Environment*. 31

Java EE *Java Platform, Enterprise Edition*. 30
JPG *Joint Photographic Experts Group*. 11–13, 47, 48
JSON *JavaScript Object Notation*. 31, 39

MBUID *Model-Based User Interface Development*. 4, 7, 9
MDE *Model Driven Engineering*. 9
MVC *Model-View-Controller*. viii, 19, 20, 24, 31, 32, 79
MVP *Model-View-Presenter*. viii, 19, 20, 79
MVVM *Model-View-ViewModel*. viii, 19, 20, 31, 79

ORM *Object Relational Mapper*. 22, 24, 26, 32

PDF *Portable Document Format*. 11–13, 39, 47
PNG *Portable Graphics Format*. 11–13, 47, 48

SVG *Scalable Vector Graphics*. c, d, 6, 11–13, 39, 47–50, 53–57, 59–62, 64, 68–70, 72, 73, 75, 76, 78–81

TIFF *Tag Image File Format*. 12, 47, 48

UCD *User Centered Design*. 9
UI *User Interface*. c, d, 4, 6, 9, 13, 34
UIDL *User Interface Design Language*. 7
URL *Uniform Resource Locator*. 39, 40
UX *User Experience*. 3, 13

XML *Extensible Markup Language*. 31, 48, 50, 51, 54, 55, 59, 60, 72
XSS *Cross-site Scripting*. 24

LISTA DE FIGURAS

Figura 1 Estrutura da *Cameleon Reference Framework*. Retirado de G. Calvary (2002). 8
Figura 2 Ambiente de trabalho Figma. 12
Figura 3 Ambiente de trabalho Sketch. 12
Figura 4 Ambiente de trabalho Adobe XD. 13
Figura 5 Princípio Cliente/Servidor. Retirado de Stevens (2019). 16
Figura 6 *Client-side* vs. *Server-side*. Retirado de Stevens (2019). 16
Figura 7 Estrutura da arquitetura de três camadas. Retirado de Kouraklis (2016). 19
Figura 8 Estrutura da arquitetura MVC. Retirado de Kouraklis (2016). 20
Figura 9 Estrutura da arquitetura MVP. Retirado de Kouraklis (2016). 20
Figura 10 Estrutura da arquitetura MVVM. Retirado de Kouraklis (2016). 20
Figura 11 *Ranking* das *frameworks* (formato gráfico). Retirado de HotFrameworks. 28
Figura 12 *Ranking* das *frameworks* (formato tabela). Retirado de HotFrameworks. 28
Figura 13 *Ranking* das *frameworks* segundo Stack Overflow. Retirado de Stack Overflow (2020). 30
Figura 14 Protótipo da *landing page* do projeto Serve.Me[1] 36
Figura 15 Estrutura de um protótipo. 37
Figura 16 Metamodelo da Arquitetura de Componentes Vue.js. 38
Figura 17 Estrutura de pastas Vue CLI. 39
Figura 18 Estrutura de pastas padrão. 41
Figura 19 Vuemmerce Github Octotree. 45
Figura 20 CoPilot Github Octotree. 46
Figura 21 Campo de texto onde se altera o ID do componente no Adobe XD. . . 49
Figura 22 Exemplo de protótipo em Adobe XD. Os retângulos azuis não fazem parte do *design*, sendo apenas utilizados para delinear *containers*. . . . 52
Figura 23 Duas imagens lado a lado, sem espaço entre elas. 65
Figura 24 Duas imagens lado a lado, com espaço entre elas. 66
Figura 25 Protótipo da Página Inicial do Clube de Ténis desenvolvida em Adobe XD. 71
Figura 26 Protótipo da Página de Reservas do Clube de Ténis desenvolvida em Adobe XD. 71
Figura 27 Resultado *Web* da Página Inicial do Clube de Ténis. 74
Figura 28 Resultado *Web* da Página de Reservas do Clube de Ténis. 74

Figura 29 Protótipo da Página de Informações do Clube de Ténis desenvolvida em Adobe XD. 76
Figura 30 Resultado *Web* da Página de Informações. 77
Figura 31 Resultado *Web* da Página de Informações corrigida. 78

LISTA DE TABELAS

Tabela 1 Comparação de ferramentas de *design* de *mockups*. 14
Tabela 2 Número de artigos em que cada característica de *framework* é mencionada. 24
Tabela 3 Quadro Comparativo de *Frameworks*. 33

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Desde a criação da *world wide web*, nos anos 90, a *internet* tem crescido de forma exponencial (Vossen and Hagemann, 2007). Em 2002 atingiu os quinhentos milhões de utilizadores e, quatro anos depois, chegou a mil milhões de utilizadores[1]. Atualmente, são mais de quatro milhares de milhões de pessoas que acedem quase diariamente à *internet*, cerca de 59% da população mundial[2]. A *internet* desencadeou assim a Quarta Revolução Industrial, levando o mundo a estabelecer-se na Era da Informação.

Este crescimento veio acompanhado de várias mudanças no paradigma das aplicações *web*. No início, com o surgimento do *browser*, as páginas *web* eram estáticas, desenvolvidas apenas através de HTML (*Hyper Text Markup Language*). Com a evolução, o princípio cliente/servidor foi sendo cada vez mais aproveitado, quer do ponto de vista do cliente, com o aumento da qualidade e interatividade das interfaces gráficas, quer do ponto de vista do servidor, com o aumento da lógica associada às aplicações. Ao longo do tempo, a *web* foi sendo cada vez mais explorada passando a ser bastante comercializável, seja através de plataformas de comércio *online*, *blogs*, redes sociais, comunidades, plataformas de *streaming*, serviços no geral e muitas outras opções que estão atualmente à disposição (Vossen and Hagemann, 2007).

A experiência de utilização, isto é, a usabilidade das plataformas, também tem sido um aspeto central de preocupação no desenvolvimento *web*. O conceito de *User Experience* (UX) *design*, presente de forma inconsciente já há milhares de anos, tem como principal objetivo proporcionar uma experiência o mais intuitiva e *user-friendly* possível aos utilizadores (Stevens, 2019).

A complexidade crescente e a necessidade de robustez cada vez mais presentes na *web* levaram à constante necessidade de simplificar e melhorar o suporte ao processo de desenvolvimento de aplicações *web*. As *frameworks web* nasceram com esse objetivo, oferecendo uma alternativa ao extremo de desenvolver aplicações do zero (*"from scratch"*). No entanto, o número de *frameworks web* existentes tem aumentado substancialmente nos últimos anos (Mehrab et al., 2018), o que comprova o atual estado de imaturidade das tecnologias de desenvolvimento *web* (Ignacio Fernández-Villamor et al., 2008), levantando vários problemas uma vez que é necessário escolher uma *framework* e mantê-la atualizada. Note-se que grande percentagem do tempo de desenvolvimento das aplicações é utilizado na respetiva manutenção (Meixner et al., 2011).

1 https://phys.org/news/2009-08-key-milestones-internet.html. Consultado a 27 de março de 2021.
2 https://www.statista.com/statistics/617136/digital-population-worldwide/. Consultado a 27 de março de 2021.

Numa abordagem baseada em modelos é possível aumentar o nível de abstração do processo de desenvolvimento (Meixner et al., 2011), o que pode ajudar a resolver a questão acima por permitir trabalhar a solução de uma forma mais independente da tecnologia a utilizar. Para além disso, é possível gerar código de forma automática. O modelo desenvolvido pode ser executado para fins de simulação ou teste em qualquer etapa do processo de desenvolvimento, o que significa que o comportamento do sistema pode ser avaliado desde a fase de requisitos até à fase de produção, sem a necessidade de alteração da descrição do sistema (Bergmann, 2014). Desta forma, a *web*, e em particular a camada de interface, podem beneficiar de uma abordagem baseada em modelos, uma vez que uma forma de lidar com tantas *frameworks* é procurar aumentar o nível de abstração do desenvolvimento.

1.2 MOTIVAÇÃO

O desenvolvimento de interfaces de utilizador para sistemas interativos tornou-se uma tarefa que requer muito tempo e, portanto, dispendiosa com a difusão das interfaces gráficas. Através da análise de diferentes aplicações de *software*, verificou-se que aproximadamente 48% do código-fonte, 45% do tempo de desenvolvimento, 50% do tempo de implementação e 37% do tempo de manutenção são necessários para aspetos relacionados com as *User Interface*s (UIs) (Meixner et al., 2011). Segundo Meixner et al. (2011), estes valores parecem continuar válidos desde que esses estudos foram feitos, uma vez que a difusão dos sistemas interativos, bem como as suas necessidades, continuaram a aumentar drasticamente nos últimos anos.

Atualmente, os programadores de UI enfrentam essencialmente os seguintes quatro problemas: heterogeneidade dos utilizadores finais, heterogeneidade das plataformas computacionais e das formas de *input* (teclado, rato, *touch*, entre outros), heterogeneidade das linguagens de programação/linguagens de *markup* e ainda a heterogeneidade dos ambientes de trabalho (Meixner et al., 2011).

O desenvolvimento de aplicações *web*, muitas vezes baseado em metodologias de desenvolvimento ágeis[3], representa um desafio para as metodologias baseadas em modelos. Por outro lado, o atual estado de imaturidade das tecnologias de desenvolvimento *web*, visível na proliferação de linguagens, *frameworks* e bibliotecas, cria dificuldades no desenvolvimento e manutenção das aplicações (Ignacio Fernández-Villamor et al., 2008), que uma abordagem baseada em modelos poderia ajudar a solucionar.

O *Model-Based User Interface Development* (MBUID) é uma abordagem que tem como objetivo enfrentar os desafios anteriormente mencionados e diminuir o esforço necessário para desenvolver as interfaces gráficas (Meixner et al., 2011). A ideia do desenvolvimento baseado em modelos é que o desenvolvimento seja feito a partir da definição de modelos da solução que se pretende, sendo depois o código gerado automaticamente. No fundo, o intuito é aumentar o nível de abstração do processo de desenvolvimento.

As metodologias de desenvolvimento baseadas em modelos são uma mais-valia na medida em que oferecem uma maior consistência na documentação e implementação, eliminam os erros de código, contribuem para uma

3 Princípios Do Manifesto Ágil disponíveis em https://agilemanifesto.org/iso/ptpt/principles.html. Consultado a 15 de abril de 2021.

verificação e validação contínua e permitem um maior número de mudanças em estados avançados dos projetos. No entanto, estas metodologias podem limitar os programas a desenvolver (Bergmann, 2014).

Este tema surge, assim, com o objetivo geral de estudar como é que a área *web* poderá tirar partido deste tipo de abordagem, com um foco particular na camada de interface, e dar um contributo para a automatização do desenvolvimento de aplicações *web*, a partir de protótipos das suas interfaces gráficas.

1.3 OBJETIVOS

Como referido, o principal objetivo da presente dissertação é dar um contributo para a automatização do desenvolvimento de aplicações *web*, a partir de protótipos das suas interfaces gráficas.

Para tal, o primeiro objetivo passa por compreender os princípios base do Desenvolvimento de Interfaces de Utilizador baseado em modelos e como se pode tirar partido deste tipo de abordagem. Posteriormente, é necessário escolher uma ferramenta de prototipagem existente atualmente no mercado para servir como ferramenta de trabalho dos *designers* e/ou *developers* que querem utilizar a solução de automatização proposta.

Em seguida, o objetivo é analisar o estado de arte atual da área de desenvolvimento *web* e fazer um estudo das *frameworks web* existentes. Para tal, é necessário fazer um estudo das principais características das *frameworks web* e perceber quais são as *frameworks* mais utilizadas. Por fim, é fundamental escolher uma *framework web* que será a base da aplicação a que o protótipo dará origem.

Numa fase seguinte, é essencial esquematizar a estrutura de um protótipo para servir como base para a posterior interpretação de um *mockup*. E ainda, é fundamental desenvolver um metamodelo da arquitetura de componentes da *framework* escolhida para se perceber melhor como se desenvolve uma aplicação na mesma e quais os passos necessários.

Depois de escolhida a ferramenta de *mockups*, a *framework web* e tendo uma melhor visão da estrutura de um protótipo e da arquitetura de componentes desta última, resta implementar a solução proposta. Previamente à implementação em si, é fundamental decidir a estratégia de desenvolvimento do projeto e qual o formato do ficheiro mais indicado a exportar na ferramenta de *mockups*. Por fim, é necessário trabalhar no algoritmo da abordagem e desenvolver os *scripts* da solução.

Para finalizar, é primordial analisar os resultados obtidos e refletir relativamente ao que se pode retirar da solução implementada.

Uma versão preliminar desta abordagem foi apresentada na Conferência *International Conference on Graphics and Interaction* (ICGI) 2021 (Machado and Campos, 2021).

1.4 ESTRUTURA DO DOCUMENTO

A presente dissertação encontra-se dividida em oito diferentes capítulos.

No Capítulo 1 é feita uma introdução ao tema da dissertação, começando por uma breve contextualização, seguida da motivação e, por fim, os objetivos gerais.

No Capítulo 2, Desenho de Interfaces de Utilizador, é feita uma introdução ao Desenvolvimento de Interfaces de Utilizador baseado em modelos e uma comparação entre esta abordagem e *Design* Centrado no Utilizador. O capítulo termina com uma descrição de várias ferramentas de desenho de interfaces gráficas e com a opção por uma delas para utilizar na presente dissertação.

O Capítulo 3, Desenvolvimento de Aplicações Interativas para a *Web*, debruça-se sobre o estado da arte da área de desenvolvimento *web*, focando aspetos como o surgimento das *frameworks web* e as vantagens e desvantagens da sua utilização. Neste capítulo, encontra-se também um estudo, tanto das principais características das *frameworks web*, como das principais *frameworks web* da atualidade. O capítulo culmina num quadro comparativo destas e com a opção pela *framework* a utilizar na presente dissertação.

No Capítulo 4, Arquiteturas de uma Aplicação, o objetivo prende-se em esquematizar a estrutura de um protótipo e desenvolver um metamodelo da arquitetura de componentes de Vue.js. É feita uma análise de arquiteturas de pastas/código-fonte de aplicações *web* em Vue.js e uma pequena análise da tradução de componentes para código.

O Capítulo 5, Descrição Genérica da Abordagem de Implementação da UI, aborda a estratégia do desenvolvimento do projeto e descreve genericamente a abordagem proposta. Começa por apresentar os motivos pelos quais será utilizado SVG (*Scalable Vector Graphics*) como base para a conversão do protótipo da interface gráfica, quais os componentes Vue.js que serão focados na solução de *software* e o método que será utilizado para mapear os protótipos e o metamodelo de Vue.js e o respetivo algoritmo genérico. Termina com a explicação dos requisitos aos quais a abordagem deve responder.

No Capítulo 6, Detalhes da Implementação da UI, começa por ser analisado como se encontram organizados os ficheiros SVG gerados pelo Adobe XD. Em seguida, é explicada em detalhe a solução de *software* desenvolvida, nomeadamente os dois *scripts* em Python.

O Capítulo 7, Avaliação, apresenta os passos da solução proposta, desde o momento em que recebe um *mockup* desenhado em Adobe XD (através de ficheiros SVG), até se obter a respetiva implementação *web*. Para tal, apresenta um exemplo prático de uma possível aplicação *web*. É ainda feita uma Discussão dos Resultados, com uma análise de exemplos de resultados *web* obtidos através dos *scripts* desenvolvidos.

Por fim, o Capítulo 8 é dedicado à conclusão e ao trabalho futuro.

2

DESENHO DE INTERFACES DE UTILIZADOR

O objetivo deste capítulo é abordar a conceção de interfaces de utilizador. Na Secção 2.1 é feita uma introdução ao tema de Desenvolvimento de Interfaces de Utilizador baseado em modelos, utilizando como ponto de partida a *Cameleon Reference Framework*. Em seguida, na Secção 2.2, é feita uma comparação entre Desenvolvimento de Interfaces de Utilizador baseado em modelos e *Design* Centrado no Utilizador.

Na Secção 2.3 são abordadas as ferramentas de desenho de interfaces mais utilizadas, sendo feita na Secção 2.4 uma breve comparação entre estas e selecionado Adobe XD como a ferramenta a utilizar para desenvolver *mockups*.

2.1 DESENVOLVIMENTO DE INTERFACES DE UTILIZADOR BASEADO EM MODELOS

A atual diversidade de dispositivos disponíveis e dos respetivos tamanhos, formas e estilos ("*form factors*"), bem como a existência de múltiplos ambientes de execução em que as aplicações podem ser disponibilizadas, coloca desafios ao desenvolvimento de aplicações interativas, dada a necessidade de suportar a adaptação das aplicações aos diferentes dispositivos e ambientes de execução. Isto é particularmente relevante na área do desenvolvimento *web*, uma vez que existe um elevado leque de *browsers* e de tipos de dispositivos (Silva and Campos, 2012).

O MBUID (Pinheiro da Silva, 2001) é uma base para a resolução deste problema. As abordagens MBUID baseiam-se no desenvolvimento de modelos, tanto de requisitos, como de soluções previstas, e no seu refinamento gradual até se atingir um sistema em funcionamento (Calvary et al., 2003). Os modelos abstratos da interface de utilizador são progressivamente transformados, incorporando detalhes sobre as modalidades de interação (primeiro) e as plataformas tecnológicas (segundo), para gerar modelos concretos da interface de utilizador (primeiro) e modelos da interface do utilizador final (segundo), de modo a que se possa obter um sistema em execução. A automatização deste processo de refinamento significa que o desenvolvimento pode progredir de forma eficiente, uma vez conhecidos os detalhes das modalidades de interação e das plataformas alvo. Contudo, é de notar que a automatização total do processo é tipicamente considerada inviável, sendo favorecida a automatização parcial (Meixner et al., 2011).

Os modelos e as transformações dos modelos permitem repensar o desenvolvimento do *design* da interface gráfica, independente dos detalhes de implementação, e redefinir modelos para concretizar essas interfaces gráficas. As *User Interface Design Language*s (UIDLs) suportam a descrição de uma interface a vários níveis

de abstração e a realização de transformações entre esses níveis (Silva and Campos, 2012). A *Cameleon Reference Framework* (G. Calvary, 2002) para o desenvolvimento baseado em modelos de interfaces de utilizador *multi-target* identifica quatro níveis de abstração (ver Figura 1):

- *Concepts and Task model* - descreve as tarefas a executar e as entidades que os utilizadores manipulam no seu cumprimento;
- *Abstract User Interface* (AUI) - descreve a interface do utilizador independentemente de qualquer modalidade concreta de interação e da plataforma informática;
- *Concrete User Interface* (CUI) - descreve uma instanciação da AUI para um conjunto concreto de modalidades de interação;
- *Final User Interface* (FUI) - corresponde à interface do utilizador que está a correr numa plataforma, seja por ser executada ou interpretada.

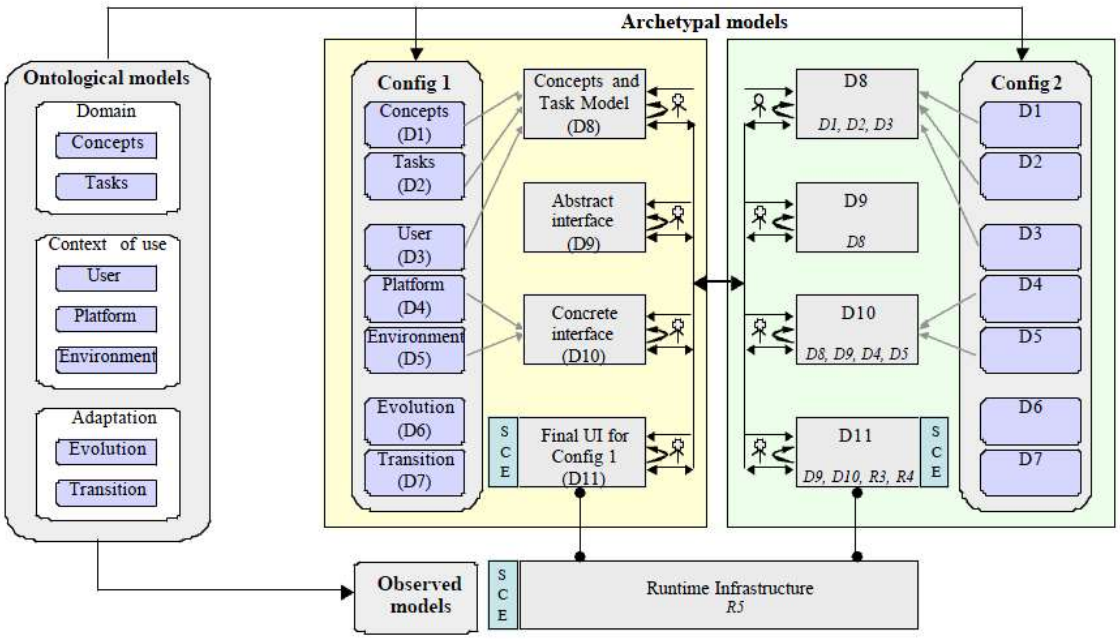


Figura 1: Estrutura da *Cameleon Reference Framework*. Retirado de G. Calvary (2002).

Resumidamente, a *Cameleon Reference Framework*[1] decompõe a conceção da interface do utilizador numa série de diferentes níveis de abstração que procuram apoiar o desenvolvimento de sistemas informáticos interativos de qualidade e ao mesmo tempo reduzir custos através da automatização do processo de desenvolvimento. Este tipo de abordagem é particularmente adequada para o desenvolvimento eficiente de aplicações *multi-target*[2].

1 https://www.w3.org/community/uad/wiki/Cameleon_Reference_Framework. Consultado a 30 de março de 2021.

2 Uma aplicação *multi-target* é composta por várias bibliotecas, módulos de aplicação e recursos criados utilizando diferentes tecnologias e implementadas em diferentes tempos de execução, mas com um ciclo de vida comum.

2.2 DESENVOLVIMENTO DE INTERFACES DE UTILIZADOR BASEADO EM MODELOS VS. USER-CENTRED DESIGN

A MBUID pode ser vista como o equivalente ao *Model Driven Engineering* (MDE) (Kent, 2002) para interfaces gráficas. No entanto, embora a MDE tenha registado um sucesso considerável, a utilização da MBUID na indústria tem sido lenta. De acordo com Meixner et al. (2011), uma maior adoção pode ser obtida por abordagens MBUID que se integrem melhor com abordagens *User Centered Design* (UCD) (Monk, 2000; ISO, 2019).

Ao contrário das abordagens que se esforçam por integrar modelos na UCD (Paternò et al., 2009; Limbourg and Vanderdonckt, 2009), mudando assim a natureza do processo de UCD, neste trabalho adota-se a opinião de que a noção de modelo é que deve ser revista. Na verdade, já no artigo de Puerta and Szkeley (1994), os modelos são vistos como parte do problema, no sentido de que o seu desenvolvimento é dispendioso.

Enquanto o MBUID pode ser visto como um processo algo determinístico desde o modelo abstrato até ao código final, a UCD tem que ver com a avaliação e refinamento iterativo de soluções e ideias de *design*. Este aspeto iterativo do processo é apoiado por protótipos. Estes protótipos consistem em *mockups* da interface de utilizador, mais alguma descrição do controlo do diálogo, e representam, tal como os modelos, o desenho pretendido. Assim, em vez de introduzir notações de modelação no processo UCD, será interessante explorar os resultados do UCD como ponto de partida para uma abordagem baseada em modelos e considerar os protótipos como base para o processo de MBUID.

Considerando que os protótipos já expressam uma noção das modalidades de interface de utilizador a serem utilizadas, pode considerar-se que podem ser vistos como uma expressão de um modelo concreto de interface de utilizador em termos de MBUID. Assim, utilizando técnicas de geração de código para automatizar (parcialmente) o processo de geração de uma interface de utilizador final, a partir de um *mockup* de interface de utilizador, é possível obter-se o melhor de ambas as abordagens. Isto é, aproveita-se a flexibilidade da UCD para explorar e avaliar ideias de *design*, usando protótipos de interface de utilizador, até que se consiga um bom *design* e aproveita-se a automatização da MBUID para gerar a interface de utilizador correspondente, a um custo inferior.

É de notar que o objetivo não é produzir a versão final da UI, mas sim produzir uma primeira versão do código que pode depois ser aperfeiçoada pelo programador. Este fator permite colocar menos requisitos no protótipo, relativamente à quantidade de informação que este deve conter.

2.3 FERRAMENTAS DE MOCKUPS

Embora a prototipagem em papel seja considerada a abordagem mais flexível e de menor custo para o desenho de *mockups* da interface gráfica, as ferramentas de *mockups* da interface gráfica têm-se tornado cada vez mais populares, particularmente para uma prototipagem de maior fidelidade.

Relativamente às *frameworks,* a ideia de *multi-target* é que a aplicação irá correr em diferentes ambientes de execução.

De uma forma simples, o objetivo da presente dissertação consiste na automatização do desenvolvimento de uma aplicação *web* através de um protótipo da interface gráfica e da respetiva navegação entre os *mockups*. Desta forma, é necessário selecionar uma entre as milhares de ferramentas que existem para o *design* de uma aplicação *web*, para assim se poder utilizar essa ferramenta na automatização pretendida.

Para esta seleção, foram analisados *rankings* de alguns artigos, nomeadamente de Baker (2021), Myre (2021) e May and Cahill (2021).

Baker (2021) apresenta o seguinte *ranking* de ferramentas de *design web*:

1. Adobe XD - https://www.adobe.com/pt/products/xd.html ;
2. InVision Studio - https://www.invisionapp.com/studio/ ;
3. Sketch - https://www.sketch.com/ ;
4. Figma - https://www.figma.com/ ;
5. Canva - https://www.canva.com/pt_pt/ ;
6. Proto.io - https://proto.io/ ;
7. Moqups - https://moqups.com/ ;
8. Balsamiq - https://balsamiq.com/ ;
9. UXPin - https://uxpin.com/ ;
10. Cacoo - https://cacoo.com/ ;
11. Fluid UI - https://www.fluidui.com/.

Para a definição desta ordem, a autora utiliza critérios como o preço e a relevância das principais funcionalidades de cada ferramenta. Já Myre (2021) destaca as seguintes ferramentas:

- Sketch, para *design* com detalhe e baseado em vetor;
- Adobe XD, para iniciantes;
- Figma, como uma opção gratuita;
- UXPin, para entregar a documentação do *design* aos programadores;
- InVision Freehand, para colaboração num estilo quadro branco;
- Adobe Photoshop - https://www.adobe.com/pt/products/photoshop.html, para *wireframes* detalhados baseados em píxeis;
- Justinmind - https://www.justinmind.com/, para *wireframes* interativos;

- Mockplus - https://www.mockplus.com/, para organizar projetos complexos.

Por fim, May and Cahill (2021) apresentam 53 ferramentas de prototipagem, sendo estas as primeiras cinco mencionadas:

1. InVision Studio;
2. Sketch;
3. Adobe XD;
4. Marvel - https://marvelapp.com/ ;
5. Figma.

Comparando as três listas, as únicas ferramentas presentes em todos esses *rankings* são o Figma, Sketch e Adobe XD. Desta forma, pode concluir-se que, de acordo com estas classificações, estas são atualmente as ferramentas de *design web* melhor conceituadas e com mais perspetivas de evolução no mercado.

Em seguida, encontra-se uma breve descrição de cada uma destas ferramentas. Para tal, foram consultados artigos de Pimenta (2020), Ivanovs (2020) e Bogawat (2019).

2.3.1 *Figma*

O Figma caracteriza-se por apresentar uma suíte de desenvolvimento de interfaces completa, com soluções integradas que englobam *wireframing*, *design* gráfico, prototipagem de interação e apresentação da navegação por meio de hiperligações. Na Figura 2 encontra-se ilustrado o respetivo ambiente de trabalho.

Uma das maiores vantagens desta ferramenta é o suporte a histórico de versões automatizado. É bastante poderosa no que toca a trabalho colaborativo entre equipas, sejam *designers*, sejam *developers* (Pimenta, 2020). É também possível integrar os testes de utilizador no fluxo de trabalho de conceção através da criação de testes diretamente no *artboard*.

Através do Figma é possível exportar os protótipos para JPG (*Joint Photographic Experts Group*), PNG (*Portable Graphics Format*), SVG e PDF (*Portable Document Format*). O Figma pode também exportar para HTML e CSS (*Cascading Style Sheets*) através de *plugins* e suporta tanto a prototipagem de baixa fidelidade, como a de alta fidelidade (Figma, 2021). Por ser *browser-based*, pode ser utilizado através da *web*, mas também é possível fazer o *download* da versão *desktop* para macOS e Windows.

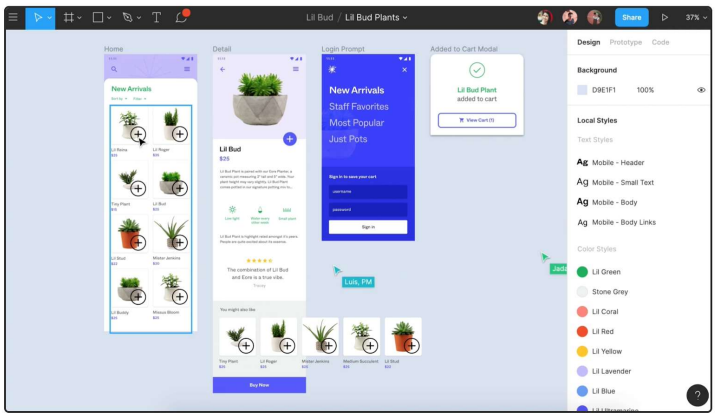


Figura 2: Ambiente de trabalho Figma.

2.3.2 *Sketch*

Sketch apresenta uma suíte completa de *design*, que permite planear, desenvolver, criar e prototipar todo o tipo de interface gráfica. A maior vantagem é ser um *software* utilizado há vários anos por muitos profissionais, garantindo assim que a ferramenta seja melhorada ano após ano.

Trata-se de uma ferramenta simples de utilizar, com poderosas funcionalidades de testes e recolha de *feedback* por parte dos clientes. Apresenta também as funcionalidades necessárias para as equipas trabalharem colaborativamente (Pimenta, 2020).

Sketch apresenta o histórico completo das alterações de um documento e gestão de versões e suporta protótipos de baixa e de alta fidelidade. Esta ferramenta pode exportar protótipos tanto para imagens *bitmap* (JPG, PNG, TIFF (*Tag Image File Format*) e WebP), como vetoriais (SVG, PDF e EPS (*Encapsulated PostScript*)) e é possível exportar para HTML e CSS através de *plugins* (Sketch, 2021). Na Figura 3 encontra-se ilustrado o respetivo ambiente de trabalho.

Uma das maiores limitações desta ferramenta é o facto de apenas estar disponível para o sistema macOS.

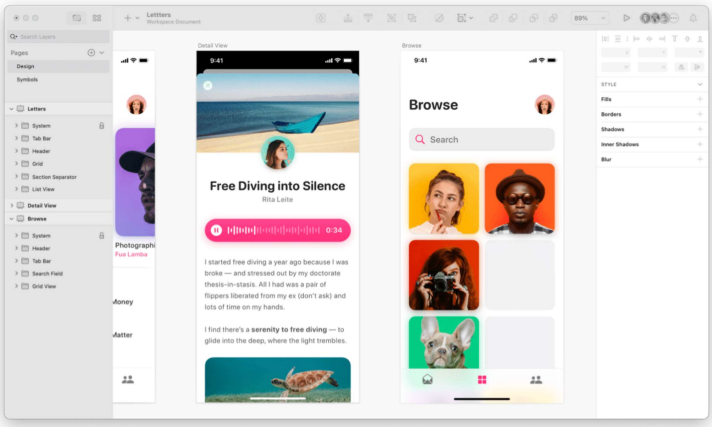


Figura 3: Ambiente de trabalho Sketch.

2.3.3 *Adobe XD*

O Adobe XD é uma ferramenta utilizada por *designers* de UI e de UX para criar *mockups* de alta fidelidade, seja para *websites*, *wireframes*, protótipos ou *artboards*. Tem a opção de trabalho colaborativo, tanto para as equipas, como também para apresentação aos clientes (Pimenta, 2020). Na Figura 4 encontra-se ilustrado o respetivo ambiente de trabalho.

Adobe XD tem o histórico das alterações para documentos na *cloud*, o que permite que seja possível rever as versões anteriores de documentos guardados, bem como colocar etiquetas e preservar essas mesmas versões. Suporta a exportação dos protótipos para JPG, PNG, SVG e PDF (Adobe, 2021). A ferramenta também permite aos utilizadores exportar os protótipos para código HTML e CSS através de *plugins* e é também possível partilhar os protótipos com os clientes.

Esta ferramenta pertence a uma empresa já muito conhecida e bem estabelecida no mercado, a Adobe, e é suportada pelo sistema Windows e macOS (Ivanovs, 2020).

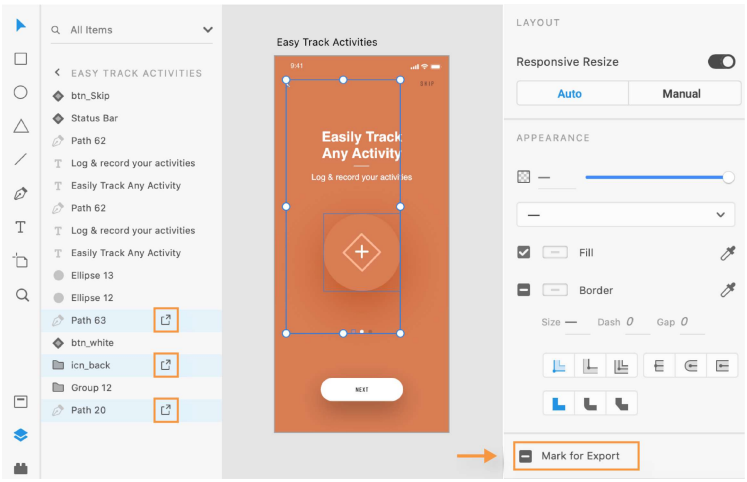


Figura 4: Ambiente de trabalho Adobe XD.

2.4 DISCUSSÃO DAS FERRAMENTAS DE MOCKUPS

A Tabela 1 apresenta um resumo da análise apresentada anteriormente sobre cada uma das ferramentas de *design* exposta. Tal como se pode constatar, as ferramentas têm características semelhantes e qualquer uma seria uma boa escolha. Adobe XD foi escolhida como a ferramenta de protótipos a adoptar, por ser suportada nativamente por ambos os sistemas operativos (o que traduz um alcance mais amplo), apresentar constantemente novas atualizações de bastante qualidade e por ser simples de utilizar.

	Figma	**Sketch**	**Adobe XD**
Testes	Sim	Sim	Sim
Desenvolvimento colaborativo	Sim	Sim	Sim
Controlo de Versões	Sim	Sim	Sim
Sistemas	Web/Windows/macOS	macOS	Windows/macOS
Nível de fidelidade	Baixa/alta	Baixa/alta	Baixa/alta
Formatos de exportação	B V W	B V W	B V W

Formatos de exportação: **B**itmap; **V**etorial; **W**eb (HTML+CSS)

Tabela 1: Comparação de ferramentas de *design* de *mockups*.

3

DESENVOLVIMENTO DE APLICAÇÕES INTERATIVAS PARA A WEB

Este capítulo começa com uma revisão da literatura em relação ao contexto geral da área de desenvolvimento *web* (Secção 3.1), com um foco particular nas *frameworks* de desenvolvimento *web*.

Na Secção 3.2 são apresentadas as características mais importantes que as *frameworks web* devem ter, segundo diferentes autores. É apresentada uma análise das características mais mencionadas pelos diversos autores e é feita uma exposição mais detalhada das mesmas. Posteriormente, na Secção 3.2.3, é feito um levantamento das *frameworks* mais relevantes na atualidade, segundo *rankings* que têm por base diferentes critérios de classificação.

Na Secção 3.3 é apresentado um quadro comparativo das *frameworks* melhor classificadas da Secção 3.2.3, bem como uma apresentação mais detalhada dessas *frameworks*, segundo as métricas mais relevantes encontradas na Secção 3.2.1.

3.1 CONTEXTO GERAL DA ÁREA DE DESENVOLVIMENTO WEB

O Desenvolvimento *Web* é atualmente uma das principais áreas do Desenvolvimento de *Software* (Ignacio Fernández-Villamor et al., 2008), abrangendo tanto o desenvolvimento de *websites*, como de *web services* e de aplicações *web* (Mehrab et al., 2018). No início dos anos 90, a maioria das páginas *web* eram documentos HTML estáticos (Vuksanovic and Sudarevic, 2011). Atualmente, os *websites* são aplicações *web* complexas que realizam transações, apresentam dados em tempo real e proporcionam experiências interativas aos utilizadores (Plekhanova, 2009).

Durante os primórdios da evolução da *web*, os *websites* eram programados manualmente, o que conduzia muitas vezes a muitos erros de código e a muito trabalho humano (Curie et al., 2019). Para além disso, verificava-se que uma grande quantidade de código era duplicada entre projetos, o que levou ao surgimento das *frameworks web* (Walker and Orooji, 2011). Estas *frameworks* vieram colmatar estes problemas e trazer inúmeras vantagens, tal como apresentado na Secção 3.1.3.

A primeira *framework* foi desenvolvida no final dos anos 90 e desde então mais de cinco mil foram lançadas (Mehrab et al., 2018). Uma das possíveis justificações para a existência de tantas *frameworks* reside no facto de quando um programador não está satisfeito com nenhuma *framework* já existente, desenvolver a sua própria (Walker and Orooji, 2011). Esta multiplicação de *frameworks*, no entanto, se por um lado significa maior capaci-

dade de escolha e maior flexibilidade, por outro, torna-se um problema, uma vez que dificulta a transferência de conhecimento entre projetos e até a escolha da melhor tecnologia a adoptar em cada momento.

3.1.1 *Definição de Aplicação Web*

Uma aplicação *web* adopta uma arquitetura cliente/servidor (ver Figura 5). O servidor fornece uma função ou serviço a um ou mais clientes, que por sua vez iniciam os pedidos de serviço através de uma rede.

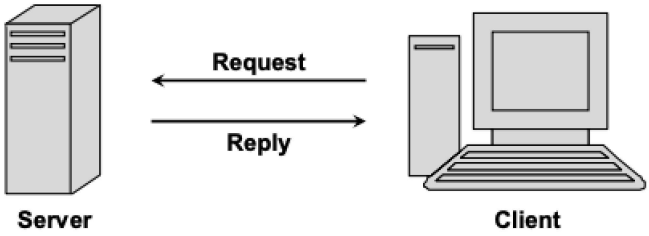


Figura 5: Princípio Cliente/Servidor. Retirado de Stevens (2019).

Como tal, no lado do servidor requer-se um servidor das aplicações *web* para executar as tarefas pedidas e, por vezes, uma base de dados para armazenar a informação (ver Figura 6). Do lado do cliente, as aplicações podem ser acedidas através de um navegador *web* e são desenvolvidas utilizando linguagens suportadas pelos navegadores (por exemplo, HTML e JavaScript) (Al-Fedaghi, 2011).

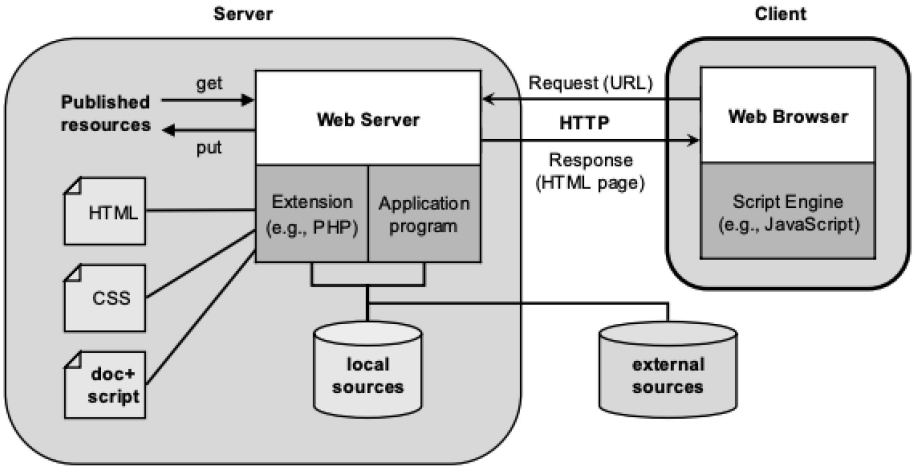


Figura 6: *Client-side* vs. *Server-side*. Retirado de Stevens (2019).

Para desenvolver aplicações *web*, os programadores utilizam tanto tecnologias de *frontend* (*client-side*), como de *backend* (*server-side*) (Pop and Altar Samuel, 2014). As tecnologias de *frontend* têm como principal objetivo apresentar os dados aos utilizadores e permitir a interação destes com o serviço. Por outro lado, as tecnologias de *backend* referem-se a tecnologias de armazenamento e de processamento de dados.

Existem várias abordagens possíveis ao desenvolvimento de uma aplicação *web*. Desde programar tudo "de raiz", tendo o programador que programar todos os detalhes recorrendo a alguma linguagem de programação (abordagem irrealista na maior parte dos casos, dado o esforço), até recorrer a alguma plataforma *low code*, ou até mesmo *no code*, e desenvolver uma aplicação sem efetivamente programar (Marvin, 2018) (abordagem com pouca flexibilidade, em que se troca liberdade por facilidade de desenvolvimento). Assim, num dos casos o desenvolvimento é feito de forma demasiado "manual" e no outro o código é gerado/configurado automaticamente. No entanto, há um meio termo entre estes dois extremos, nomeadamente utilizar uma *framework web* (Liawatimena et al., 2018). Neste caso, apenas é fixado um estilo arquitetural para a aplicação, sendo a liberdade de desenvolvimento (mas também a necessidade de conhecimentos técnicos) maior.

3.1.2 *Definição de Framework Web*

Em ambientes de sistemas de informação, uma *framework* é uma estrutura de apoio na qual outras aplicações de *software* podem ser organizadas e desenvolvidas. Uma *framework* pode incluir programas de apoio, bibliotecas de código, uma linguagem de *scripting*, serviços comuns, interfaces ou outros pacotes de *software*/*utilities* para ajudar a desenvolver e juntar os diferentes componentes de uma aplicação de *software* (Shan and Hua, 2006). Uma *framework web* é basicamente uma ferramenta que ajuda a construir um *website*, seja ele estático ou dinâmico (Curie et al., 2019).

3.1.3 *Vantagens e desvantagens das frameworks*

As *frameworks web* ajudam os programadores ao definirem uma forma padrão de desenvolver e implementar aplicações *web* (Mehrab et al., 2018). Os efeitos positivos das *frameworks* nos projetos estão relacionados com a diminuição de erros de código (Curie et al., 2019), redução do tempo de desenvolvimento, complexidade reduzida, aumento da produtividade, fiabilidade (Mehrab et al., 2018), extensibilidade (Okanovic, 2014; Mehrab et al., 2018), modularidade (Okanovic, 2014), qualidade (Shan and Hua, 2006)(Salas Zarate et al., 2015) e desempenho (Salas Zarate et al., 2015). Outras vantagens são o facto de proporcionarem um ambiente completo para o desenvolvimento, interoperabilidade, consistência, código exaustivamente testado nas bibliotecas, classes e funções, bem como código bem estruturado, utilizando padrões arquiteturais. Para além destas, outras vantagens são a segurança e manutenção, para que os programadores não tenham de construir sistemas personalizados a partir do zero cada vez que lançam um novo *website* e ainda o facto de possuírem componentes de *software* ou blocos de construção de código, de modo a que os programadores possam partilhar e reutilizar o código (Vuksanovic and Sudarevic, 2011).

Estes benefícios proporcionam o que todos os programadores desejam, nomeadamente o rápido desenvolvimento de aplicações, reutilização tanto do código como da conceção, custos de manutenção reduzidos e uma maior facilidade de personalização (Okanovic, 2014).

No entanto, existem também algumas desvantagens no uso de *frameworks web*. Em algumas situações, o desempenho da aplicação pode ser bastante prejudicado devido à complexidade e sobrecarga do código da

framework, que acaba por criar uma maior carga para o *hardware* subjacente (Vuksanovic and Sudarevic, 2011). Para além disso, existem *frameworks* que possuem uma curva de aprendizagem elevada e convenções rígidas que dificultam a flexibilidade da aplicação e a criatividade do programador. Essa curva de aprendizagem é o reflexo da *framework* definir uma série de regras que é necessário cumprir e o facto de encapsular uma série de conceitos que é necessário compreender.

3.1.4 *Client-side e Server-side*

As *frameworks* podem ser classificadas em duas categorias: *client-side* e *server-side*. As *frameworks client-side* suportam o desenvolvimento de componentes que estão no *browser*, tipicamente a interface, permitindo melhor interatividade. Alguns exemplos dessas *frameworks* são as seguintes: Vue.js[1], Angular.js[2] e Ember.js[3] (Curie et al., 2019).

Dado existirem tantos dispositivos com acesso à *web*, é importante que quando se está a desenvolver uma interface gráfica se tenha em consideração a sua responsividade. Esta é uma característica que permite que a aplicação *web* se adapte automaticamente a qualquer tipo de dispositivo, desde um telemóvel com um pequeno ecrã até um computador com um grande monitor (Voutilainen et al., 2015). As *frameworks* são uma mais-valia neste aspeto, uma vez que incorporam bibliotecas e *guidelines* que facilitam esse trabalho.

As *frameworks server-side* suportam o desenvolvimento do código que corre no servidor, gerando as páginas para enviar para o cliente (*browser*). Estas *frameworks* têm também regras e arquiteturas bem definidas e permitem criar diferentes tipos de páginas, podendo ainda fornecer fatores de segurança às mesmas. Exemplos deste tipo de *frameworks* incluem Django[4], Zend[5] e Ruby on Rails[6] (Curie et al., 2019).

Em conclusão, as *frameworks client-side* permitem uma experiência de utilização mais próxima de uma aplicação nativa, enquanto que as *server-side* estão mais presas ao modelo de navegação na *web*.

3.1.5 *Arquiteturas de software*

Como forma de facilitar o desenvolvimento de aplicações *web*, as *frameworks* podem seguir diferentes padrões arquiteturais (*architectural patterns*). Estes padrões são soluções gerais, reutilizáveis e bem estabelecidas para problemas comuns na conceção de *software* e ajudam a documentar as decisões de conceção arquiteturais, bem como facilitam a comunicação entre os *stakeholders* através de um vocabulário comum (Avgeriou and Zdun, 2005).

1 https://vuejs.org/. Consultado a 15 de novembro de 2021.
2 https://angularjs.org/. Consultado a 15 de novembro de 2021.
3 https://emberjs.com/. Consultado a 15 de novembro de 2021.
4 https://www.djangoproject.com/. Consultado a 15 de novembro de 2021.
5 https://www.zend.com/. Consultado a 15 de novembro de 2021.
6 https://rubyonrails.org/. Consultado a 15 de novembro de 2021.

A *three-tier arquitecture* (ver Figura 7) é uma abordagem que defende a organização da aplicação em três camadas, nomeadamente a camada de apresentação, a camada de negócio e a camada de acesso a dados. Essas camadas são uma forma de encapsular a lógica que cada parte executa (Kouraklis, 2016).

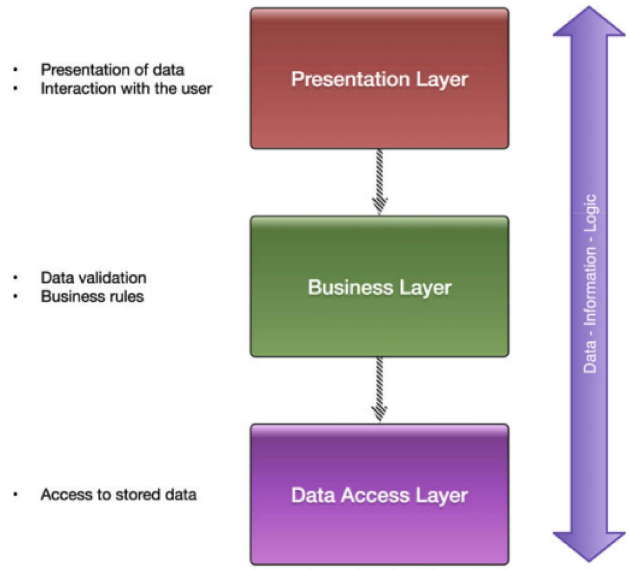


Figura 7: Estrutura da arquitetura de três camadas. Retirado de Kouraklis (2016).

A arquitetura *Model-View-Controller* (MVC) (ver Figura 8) combina várias tecnologias normalmente divididas num conjunto de camadas (Pop and Altar Samuel, 2014) e define uma arquitetura para a camada de interface (Kouraklis, 2016). Desta forma, este padrão ajuda a organizar melhor o código do programa, promovendo a separação da lógica de controlo da interface (*controller*), da definição do conteúdo da interface (*view*) e do acesso à lógica de negócio (*model*). O *Model* suporta basicamente o acesso ao *backend* que contém as camadas da lógica de negócio e de dados (Curie et al., 2019), sendo os ficheiros que o compõe responsáveis pela busca, modificação, inserção e remoção de dados da base de dados. A *View* é responsável pelo aspeto visual da página (*frontend*), mostrando os dados aos utilizadores da aplicação (Vuksanovic and Sudarevic, 2011). Por fim, o *Controller* é responsável pela invocação e gestão do conteúdo das páginas, de acordo com o pedido do utilizador (Nassourou, 2010). Desta forma, o *Controller* aciona e recolhe dados do *Model*, carrega os dados e encaminha-os para as *Views*, que apresentam os resultados ao utilizador.

A arquitetura *Model-View-Presenter* (MVP) (ver Figura 9) surgiu numa tentativa de tentar combater algumas insuficiências da arquitetura MVC. Neste tipo de abordagem, o *Controller* é substituído pelo *Presenter* e as funções, responsabilidades e capacidades de cada parte são alteradas. Desta forma, passa a existir uma evidente separação entre a *View* e o *Model* e a sincronização é realizada pelo *Presenter* (Kouraklis, 2016). Nesta versão do padrão MVP não há comunicação entre a *View* e o *Model*, o que normalmente é intitulado como *Passive View*.

A arquitetura *Model-View-ViewModel* (MVVM) (ver Figura 10) surgiu com uma alternativa aos padrões MVC e MVP. Nesta arquitetura, o *ViewModel* substitui o *Presenter* e o *Controller* (Kouraklis, 2016).

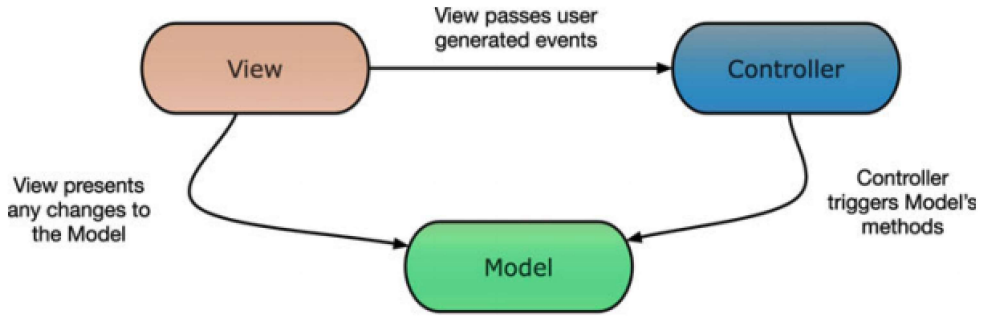


Figura 8: Estrutura da arquitetura MVC. Retirado de Kouraklis (2016).

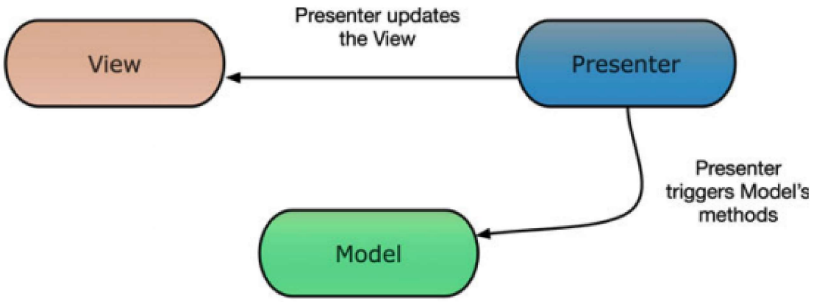


Figura 9: Estrutura da arquitetura MVP. Retirado de Kouraklis (2016).

O *Model* mantém-se com uma lógica semelhante à dos padrões MVP e MVC, sendo responsável pelo acesso a diferentes fontes de dados (por exemplo, bases de dados, ficheiros ou servidores) (*business layer*). A *View* representa os dados no formato apropriado (gráfico ou não gráfico), refletindo o estado dos dados e recolhe a interação e eventos do utilizador. Na arquitetura MVVM, a maior parte do código encontra-se no *ViewModel*. Este componente define o conteúdo que a *View* poderá mostrar e a forma como vai reagir às interações do utilizador. É no *ViewModel* que são descritos vários princípios e estruturas que apresentam dados específicos recolhidos através do *Model* (Kouraklis, 2016).

O *ViewModel* trata também da comunicação entre a *View* e o *Model*, encaminhando todos os dados necessários da *View* para o *Model* de uma forma que o *Model* possa facilmente interpretar. A *View*, por sua vez, através do *ViewModel* observa os dados que estão no *Model* para se manter atualizada.

Figura 10: Estrutura da arquitetura MVVM. Retirado de Kouraklis (2016).

3.2 FRAMEWORKS DE DESENVOLVIMENTO WEB

A escolha da *framework* mais adequada para um determinado projeto é uma etapa-chave no desenvolvimento de aplicações *web* (Ignacio Fernández-Villamor et al., 2008), podendo até ser considerada a fase mais importante. Como atualmente existem muitas *frameworks* baseadas em diferentes linguagens, a escolha da *framework* a utilizar pode não ser uma tarefa fácil (Curie et al., 2019; Salas Zarate et al., 2015) e não existe uma *framework* que seja a melhor para todo e qualquer projeto (Plekhanova, 2009). Escolher uma *framework* inadequada pode prejudicar todo o desenvolvimento da aplicação *web*, quer seja pela falta de documentação existente, por causarem dificuldades aos programadores ao fazerem quaisquer alterações ao comportamento do núcleo da *framework*, quer por afetarem o desempenho e a velocidade dos *websites* (Curie et al., 2019). Para além disso, uma má escolha pode levar a vários problemas e atrasos no desenvolvimento da aplicação *web*, desde perda de tempo a estudar os detalhes de uma nova linguagem que não é a melhor para o projeto, passando por falhas no cumprimento dos prazos, uma vez que os programadores não estão habituados à *framework*, até ao desperdício de tempo a tomar medidas corretivas para escolher uma *framework* diferente. Para evitar todos estes problemas, é muito importante conhecer e identificar as melhores práticas de desenvolvimento *web* (Salas Zarate et al., 2015) e perceber previamente quais são os critérios que se pretende que o projeto reúna e escolher uma *framework* que vá ao encontro desses aspetos (Ignacio Fernández-Villamor et al., 2008).

3.2.1 *Características das Frameworks de Desenvolvimento Web*

Existem vários artigos que tentam enumerar as características mais importantes das *frameworks* e que tentam definir modelos de comparação de *frameworks web*. Ignacio Fernández-Villamor et al. (2008) mencionam que as principais características para se categorizar uma *framework* são as seguintes:

- Descrição de Domínio e Persistência;
- Apresentação;
- Segurança;
- Usabilidade;
- Testes;
- Orientação ao Serviço;
- Orientação aos Componentes;
- Adoção.

Para cada um desses aspetos, é também apresentado um conjunto de perguntas que devem ser respondidas do ponto de vista de cada uma das *frameworks* que se pretende analisar. Respondendo a todas essas

questões, é possível retirar importantes conclusões em relação à interoperabilidade, maturidade, usabilidade e à capacidade de realizar testes de cada uma das *frameworks*.

Curie et al. (2019) apresentam sete diferentes características que as *frameworks* podem ter ou suportar, sendo elas:

- Computação em Nuvem;
- HTML;
- *Template Framework*;
- Mapeamento Relacionado a Objetos;
- Segurança;
- Suporte de Plataforma;
- *Debugging*.

Já Walker and Orooji (2011) focam os seguintes aspetos:

- Instalação;
- Curva de Aprendizagem;
- *Core Library*;
- *Object Relational Mapper* (ORM);
- *Unit Testing*;
- JavaScript incluído;
- Documentação;
- Comunidade;
- Atualizações;
- Reutilização de partes.

No entanto, os autores acabam por concluir que conhecer bem a linguagem é a parte mais importante ao usar-se uma *framework*.

Os aspetos realçados por Vosloo and Kourie (2008) são os seguintes:

- Apresentação;
- Validação de Formulários;

- Validação;
- Fluxo das Páginas;
- Estado da Sessão;
- Autenticação;
- Integração do *Backend*;
- Tratamento de Eventos;
- Concorrência;
- Uso de Recursos;
- *Miscellaneous*.

Para além disso, no artigo de Salas Zarate et al. (2015) são apresentadas as melhores práticas no desenvolvimento *web*. Os autores discutem 23 critérios, desde suporte a AJAX (*Asynchronous JavaScript and XML*) até à utilização de *pattern matching*, mas realçam três que consideram fundamentais, sendo elas:

- Fiabilidade;
- Usabilidade;
- Segurança.

Já no artigo de Plekhanova (2009) é mencionado que as métricas tradicionais de avaliação de uma *framework*, tais como *benchmark performance* e a qualidade técnica da linguagem de programação, são cada vez mais menosprezadas, uma vez que atualmente as diferenças entre estes aspetos são cada vez menores e cada vez menos relevantes para o desenvolvimento *web*. Desta forma, neste artigo são mencionadas sete características e cada uma delas deve ser avaliada numa escala de 1.00 (fraco) a 5.00 (excelente). Posteriormente, é apresentada uma fórmula que atribui diferentes pesos a cada uma destas métricas, culminando assim numa fórmula para a avaliação de *frameworks*. As métricas apresentadas são as seguintes:

- Desenvolvimento da interface do utilizador (peso de 0.2);
- *Maintainability* (peso de 0.15);
- Gestão e migração de dados (peso de 0.2);
- Testabilidade (peso de 0.15);
- Popularidade (peso de 0.1);
- Comunidade e Maturidade (peso de 0.1);
- *Marketability* (peso de 0.1).

Característica	Número de Artigos
Apresentação	2
Segurança	3
Computação em Nuvem	2
Suporte HTML	2
Template Framework	2
Testes	4
Suporte de Plataforma	2
Debugging	2
ORM	2
Suporte de *frameworks* baseadas em JavaScript	2
Documentação	2
Comunidade	2
Validação	3
Usabilidade/Curva de Aprendizagem	2

Tabela 2: Número de artigos em que cada característica de *framework* é mencionada.

3.2.2 *Análise das Características das Frameworks de Desenvolvimento Web*

Para se avaliarem e compararem as diferentes *frameworks*, podem seguir-se diferentes critérios, mas existem algumas características que acabam por ser mais comuns nos diversos autores que fazem esta análise, como se pode constatar a partir dos trabalhos apresentados anteriormente.

A Tabela 2 apresenta o número de artigos em que cada característica é mencionada, podendo-se assim concluir que estas são as características mais relevantes numa *framework web* (apenas se apresenta as características que são mencionadas em mais do que um artigo). Em seguida, encontra-se uma breve descrição do significado de cada uma delas. Cada uma das *frameworks web* terá um nível de suporte a cada uma das características.

Apresentação

A apresentação inclui tudo o que uma aplicação *web* precisa para entregar a interface gráfica no *browser* do utilizador, utilizando para isso HTML ou uma linguagem de *markup* semelhante (Vosloo and Kourie, 2008).

A apresentação é provavelmente o requisito funcional mais simples e mais bem compreendido que uma *framework web* deve proporcionar. É o primeiro requisito reconhecido, uma vez que as primeiras *frameworks web* concentraram-se exclusivamente na apresentação (é de salientar que a apresentação está intimamente relacionada com as preocupações do padrão de *design* MVC).

Segurança

As aplicações na *web* estão sob constante ataque e é necessário o uso de mecanismos de proteção contra os problemas mais comuns, tais como o *Cross-site Scripting* (XSS), as falhas de injeção ou a execução de ficheiros

maliciosos. No entanto, não há uma forma clara de resolver estes problemas e normalmente as *frameworks* recomendam boas práticas ou fornecem ferramentas úteis para a sua gestão, como mecanismos para lidar com a autenticação, autorização e sessões de utilizadores (Ignacio Fernández-Villamor et al., 2008).

Computação em Nuvem

A computação em nuvem é a prestação de serviços tais como armazenamento, servidores, análises, entre outros, através da *internet*. Existem vários benefícios associados ao uso de computação em nuvem, como desempenho, produtividade, flexibilidade e custo. Uma vez que existem vários tipos de *cloud deployments*, como nuvens públicas, privadas e híbridas, o utilizador pode optar pela opção que melhor satisfaz as necessidades da sua aplicação (Curie et al., 2019).

Suporte HTML5

O HTML define as propriedades e os comportamentos da página *web*. Com HTML é possível incorporar imagens, animações, áudios, entre muitos outros elementos, sem utilizar programas de terceiros (Curie et al., 2019).

O HTML5 é a quinta versão do HTML, tendo adicionado muitos novos elementos para responder à evolução tecnológica dos *browsers* e das aplicações.

Template Framework

Os *templates* podem ser utilizados para melhorar a uniformidade, a disposição e a navegação de qualquer *website*. Os *templates* são basicamente páginas *web* pré-desenhadas, constituídos por conjuntos de peças de código HTML, que simplificam o processo de desenvolvimento da aplicação *web*. (Curie et al., 2019).

Testes

Os programadores frequentemente dedicam mais tempo e esforço a encontrar e corrigir erros de código do que efetivamente a escrever novo código. Os testes de *software* são uma atividade destinada a avaliar a qualidade de um programa e a tentar melhorá-lo através da identificação de defeitos e problemas (Salas Zarate et al., 2015).

Suporte de Plataforma

É importante que antes dos programadores se dedicarem ao desenvolvimento de novo código, avaliem as exigências que serão colocadas ao *hardware* e ao sistema operativo. Em suma, seria pouco produtivo investir muito tempo e dinheiro na configuração e codificação para descobrir que o desempenho do servidor é fraco devido ao facto da plataforma não ser adequada. As *frameworks web* foram desenvolvidas para funcionar numa variedade de plataformas, seja Windows, Linux ou macOS (Salas Zarate et al., 2015).

Debugging

O *debugging* é o processo de identificar, localizar e eliminar erros de código (Curie et al., 2019).

ORM

ORM permite que os programadores definam declarativamente o mapeamento entre uma arquitetura orientada a objetos e o modelo relacional da base de dados e permite também que as operações de acesso à base de dados sejam expressas em termos de objetos (Salas Zarate et al., 2015). A principal função da ORM é reduzir a complexidade do código (Curie et al., 2019).

Suporte de frameworks baseadas em JavaScript

A incorporação de JavaScript numa página *web* permite melhorar a experiência dos utilizadores através da conversão de uma página estática numa página interativa. Adicionalmente, o código JavaScript corre do lado do cliente, ou seja, corre no processador do utilizador e não no servidor *web*. Desta forma, diminui-se a largura de banda e a tensão do lado do servidor *web* (Salas Zarate et al., 2015).

Documentação

A documentação de uma *framework* deve cobrir todos os detalhes da mesma. Uma vez que as *frameworks* se baseiam em alguma linguagem de programação, a documentação da *framework* também se baseia na documentação da própria linguagem de programação (Walker and Orooji, 2011).

A documentação pode incluir a *Application Programming Interface* (API), isto é, a lista de todos os métodos de cada biblioteca da *framework*, tutoriais com exemplos de tarefas mais comuns e ainda tutoriais de terceiros (Walker and Orooji, 2011).

Comunidade

A comunidade de apoiantes de uma *framework* é constituída por pessoas que interagem e comunicam sobre a *framework* em *websites* e *blogs*. A comunidade incentiva o crescimento da *framework* e simultaneamente ajuda a esclarecer dúvidas de outros programadores (Walker and Orooji, 2011).

Validação

A validação está relacionada com o *input* do utilizador e frequentemente é vista como parte do tratamento de formulários. Assim, a validação refere-se à definição de critérios para a entrada de dados por parte do utilizador. A validação pode também depender de conhecimentos de domínio mais complexos que podem ter de ser verificados num sistema de *backend* (Vosloo and Kourie, 2008).

O programador deve especificar as validações que devem ser efetuadas, como é que os erros de validação devem ser transmitidos ao utilizador final (para que este os possa corrigir) e que influência devem ter esses erros no comportamento dinâmico da interface gráfica apresentada (Vosloo and Kourie, 2008).

Usabilidade/Curva de Aprendizagem

A usabilidade mede a facilidade de interação de um utilizador com um sistema. Para tal, são consideradas todas as funcionalidades do sistema, bem como a sua interface gráfica. A usabilidade depende de aspetos como o tipo de utilizadores que vão trabalhar com o sistema, quais são os seus objetivos e qual é o contexto de utilização. No âmbito das *frameworks web*, a usabilidade mede a experiência de desenvolvimento dos programadores ao utilizarem a *framework* (Ignacio Fernández-Villamor et al., 2008). Essa experiência acaba por estar intimamente relacionada com a curva de aprendizagem da *framework*.

3.2.3 *Rankings de Frameworks Web*

Tal como mencionado na Secção 3.1, desde o início da era digital foram já desenvolvidas mais de cinco mil *frameworks* de desenvolvimento *web* (Mehrab et al., 2018). Todas essas *frameworks* possuem diferentes características e são reconhecidas por diferentes particularidades. Avaliar o nível de suporte de cada uma delas às características anteriormente mencionadas seria impossível.

No entanto, existem vários estudos que tentam fazer *rankings* das *frameworks web*, tendo por base diferentes fatores como o tamanho da comunidade da *framework*, a popularidade ou até mesmo a quantidade de utilizações bem-sucedidas. Em seguida, serão apresentados alguns desses *rankings*.

No *website* HotFrameworks[7] é apresentado um *ranking* de *frameworks* tendo por base duas diferentes métricas de popularidade, nomeadamente o GitHub *score* e o Stack Overflow *score*. O GitHub *score* é baseado no número de estrelas que o repositório tem no GitHub. Estas estrelas representam o número de utilizadores que colocaram esse determinado repositório como favorito, sendo que os repositórios favoritos de um utilizador aparecem no seu próprio perfil público. O Stack Overflow *score* é baseado no número de perguntas realizadas no Stack Overflow que contêm a *tag* da *framework*.

Uma vez que estas duas medidas de popularidade estão em escalas diferentes, as pontuações finais são normalizadas a uma escala de 0-100 e é feita uma média entre os dois resultados obtidos, culminando assim numa pontuação final. Caso a *framework* não possua um repositório no GitHub ou não tenha uma *tag* inequívoca no Stack Overflow, a métrica em falta é desconsiderada, ficando assim a pontuação final igual ao valor da métrica existente. Segundo o HotFrameworks, neste *ranking* são consideradas todas as *frameworks* capazes de terminar a frase "Acabei de construir esta aplicação *web* em [inserir nome da *framework* aqui]".

Como se pode ver nas Figuras 11 e 12, atualmente React é a *framework* melhor posicionada tendo em conta estas duas métricas. Em segundo lugar está ASP.NET MVC, que apesar de não ter repositório no GitHub, consegue o segundo lugar por ser bastante popular no Stack Overflow. Em seguida, apresentam-se Angular, Ruby on Rails, AngularJS, Vue.js e Django, que surgem empatadas. É de realçar que a *framework* com melhor GitHub *Score* é Vue.js e a *framework* com melhor Stack Overflow *score* é ASP.NET.

7 https://hotframeworks.com/. Consultado a 25 de janeiro de 2021.

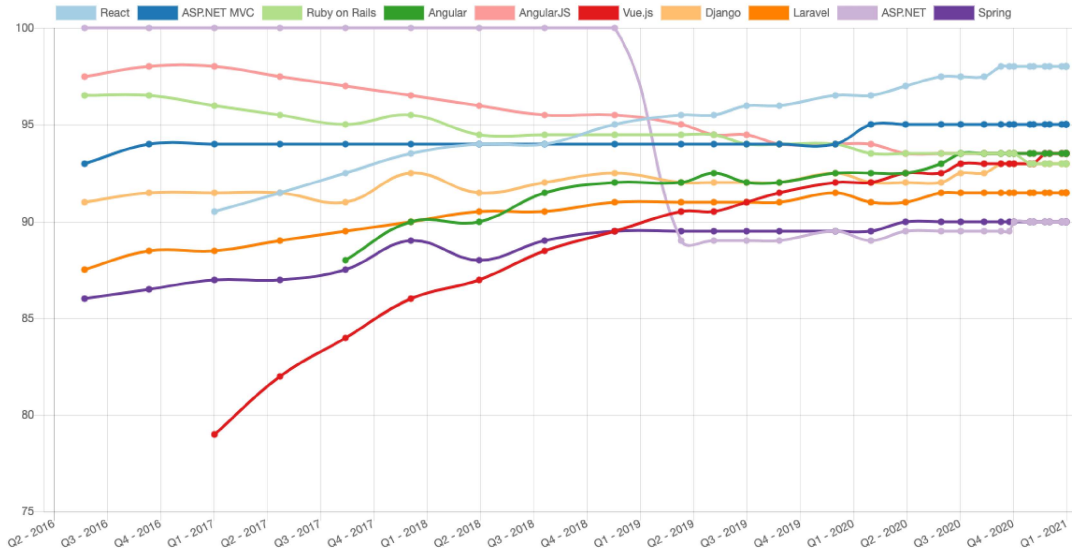


Figura 11: *Ranking* das *frameworks* (formato gráfico). Retirado de HotFrameworks.

Rankings

Framework	Github Score	Stack Overflow Score	Overall Score
React	99	97	98
ASP.NET MVC		95	95
Angular	91	96	93
Ruby on Rails	87	99	93
AngularJS	90	97	93
Vue.js	100	87	93
Django	89	97	93
Laravel	90	93	91
ASP.NET	80	100	90
Spring	86	94	90
Express	88	87	87
Flask	89	83	86
Meteor	86	80	83
Symfony	81	86	83
CodeIgniter	79	86	82
JSF		81	81
Ember.js	80	78	79
.NET Core	77	80	78

Figura 12: *Ranking* das *frameworks* (formato tabela). Retirado de HotFrameworks.

Já Salas Zarate et al. (2015) selecionam as seguintes *frameworks*: Struts, JSF, CakePHP, Grails, Ruby on Rails, Catalyst e Django. Os autores referem que esta seleção foi feita tendo em consideração o nível de maturidade e a utilização bem-sucedida destas *frameworks* em vários projetos (incluindo *websites* muito conhecidos). Mencionam também que essa seleção de *frameworks* foi deduzida com base num *ranking* que tem em conta a experiência de programadores que estão habituados a trabalhar com estas *frameworks* e que afirmam tê-las

selecionado por terem uma grande e ativa comunidade de programadores, bem como, por terem uma curva de aprendizagem muito baixa. Apesar disso, o critério de escolha não é muito claro na globalidade e não fica explícito como é que as *frameworks* foram efetivamente escolhidas.

Desde 2011, o Stack Overflow tem vindo a fazer anualmente um estudo onde questiona os programadores em relação a quais são as suas tecnologias preferidas, os seus hábitos ao programar e as suas preferências de trabalho (Stack Overflow, 2017). No estudo de 2020, foram inquiridos cerca de 65.000 programadores, e uma das questões consistia em selecionar as *frameworks web* que utilizam no seu dia a dia. Desta forma, segundo os resultados desse estudo, que podem ser consultados em Stack Overflow (2020) e na Figura 13, as dez *frameworks web* mais utilizadas em 2020 foram as seguintes:

1. jQuery [8] - https://jquery.com/;
2. React.js - https://reactjs.org/;
3. Angular - https://angular.io/;
4. ASP.NET - https://dotnet.microsoft.com/apps/aspnet/;
5. Express - https://expressjs.com/;
6. ASP.NET Core - https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0/;
7. Vue.js - https://vuejs.org/;
8. Spring - https://spring.io/;
9. Angular.js - https://angularjs.org/;
10. Django - https://www.djangoproject.com/.

Segundo o estudo, jQuery ainda está a liderar a tabela, mas tem vindo a perder utilizadores para React.js e Angular, ano após ano. Em alguns casos, jQuery pode estar a ser utilizado em conjunto com outras *frameworks*.

3.3 ANÁLISE DE MÉTRICAS E FRAMEWORKS WEB

Após a análise à forma como as *frameworks* têm vindo a ser classificadas, nesta secção pretende-se comparar *frameworks web* tendo por base diferentes métricas. Para tal, numa primeira fase selecionam-se sete *frameworks web*, considerando os três *rankings* apresentados na Secção 3.2.1, e é feita uma breve apresentação de cada uma destas. Em seguida, são selecionadas algumas métricas, tendo em consideração as características recolhidas na Secção 3.1. Por fim, são apresentados quadros comparativos dessas métricas nas *frameworks* anteriormente selecionadas.

8 jQuery é considerada uma biblioteca de JavaScript e não tanto uma *framework web*, mas foi incluída por fazer parte do estudo.

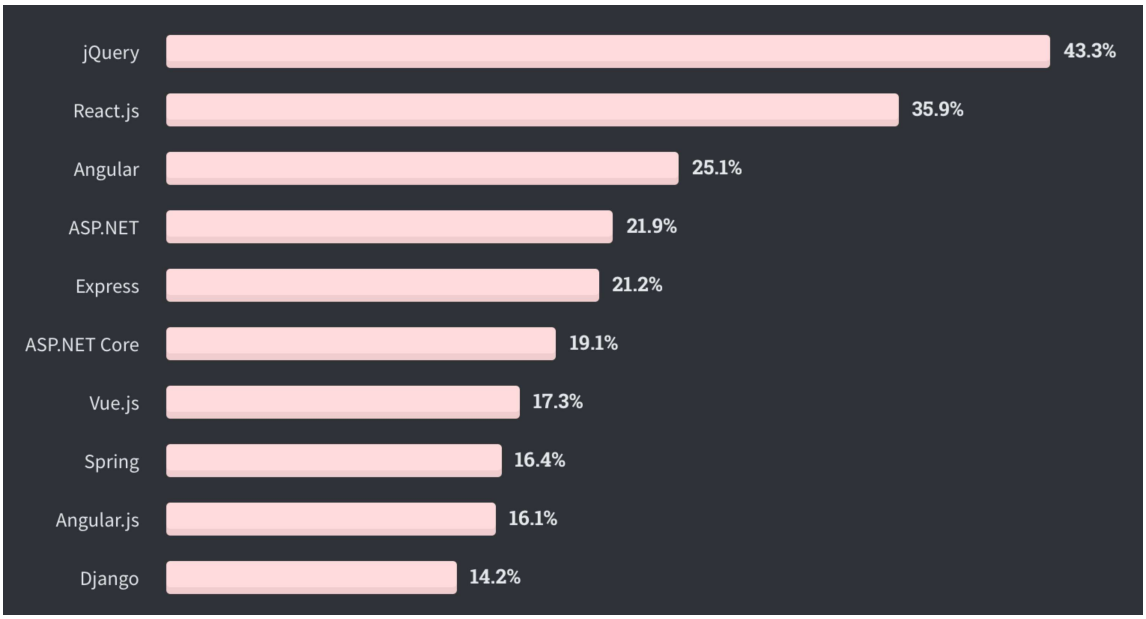


Figura 13: *Ranking* das *frameworks* segundo Stack Overflow. Retirado de Stack Overflow (2020).

3.3.1 *Seleção de Frameworks Web*

A necessidade de se fazer uma pré-seleção das *frameworks* a analisar surge por existirem muitas *frameworks web* e comparar todas elas seria demasiado trabalhoso e pouco produtivo. Desta forma, analisando os *rankings* apresentados na Secção 3.2.3, foram selecionadas as seguintes *frameworks*:

- Spring;
- ASP.NET MVC;
- Angular;
- Django;
- Vue.js;
- React;
- Ruby on Rails.

De seguida, faz-se uma apresentação breve de cada uma das *frameworks*, realçando alguns aspetos que (de forma não exclusiva) as caracterizam.

Spring

Spring é uma *framework* em camadas de *Java Platform, Enterprise Edition* (Java EE), que inclui um "*lightweight container*" para configuração automática de objetos de aplicação através da inversão de controlo (injeção de

dependência), uma camada de abstração para gestão de transações e uma camada de abstração de dados. A *framework* Spring pode também utilizar o Spring IDE (*Integrated Development Environment*), uma interface gráfica do utilizador para os ficheiros de configuração (Shan and Hua, 2006).

ASP.NET MVC

ASP.NET MVC permite desenvolver aplicações *web* dinâmicas com uma separação clara das preocupações e um controlo total sobre a *markup* para um desenvolvimento agradável e ágil. Esta *framework* utiliza o padrão MVC e é considerada uma *framework* leve ("*lightweight*") e altamente testável, que está integrada com características ASP.NET existentes (Islam et al., 2011).

Angular

Angular é uma *framework web* focada no desenvolvimento de interfaces gráficas, baseada em TypeScript e que segue o padrão MVVM (Smith, 2009). A *framework* baseia-se numa hierarquia de componentes. Desta forma, os componentes são os principais blocos de construção e podem exibir informação, renderizar modelos e realizar ações nos dados (Wohlgethan, 2018).

Django

Django é uma *framework web* baseada em Python (Plekhanova, 2009) e baseia-se no padrão MVC (Liawatimena et al., 2018). Foca-se em ser o mais eficiente possível, dando ao programador a possibilidade de realizar quase todas as tarefas com o menor esforço possível de programação (código). Django é também escalável, maduro e rápido, com uma numerosa comunidade de programadores e um enorme conjunto de componentes incorporados. Django pode aceder ou criar instâncias de dados JSON (*JavaScript Object Notation*) ou XML (*Extensible Markup Language*) e lidar com sistemas de gestão de bases de dados relacionais, tais como Oracle, MySQL, SQLite, e PostgreSQL (Liawatimena et al., 2018).

Vue.js

Vue.js é uma *framework web* de JavaScript, focada no desenvolvimento de interfaces gráficas. A arquitetura de Vue.js é normalmente descrita como MVVM. Vue.js destaca-se pelo seu elevado grau de escalabilidade e pela sua facilidade de aprendizagem e é caracterizada por ser baseada em componentes e usar diretivas nos *templates* (Wohlgethan, 2018).

React

React é uma *framework web* em JavaScript que tem como objetivo desenvolver interfaces gráficas. React é caracterizado por ser orientado por componentes. O objetivo geral é transformar o estado atual da aplicação numa vista que possa ser apresentada ao utilizador (através da DOM (*Document Object Model*)). É possível escrever componentes através de duas diferentes abordagens: componentes como funções (mais recomendada) e componentes como classes *ECMAScript 6* (ES6) (Wohlgethan, 2018).

Ruby on Rails

Ruby on Rails é uma *framework* caracterizada por seguir os princípios de "convenção sobre configuração" e de "não se repetir a si mesmo". As aplicações desenvolvidas em Rails têm que seguir o padrão MVC. Além disso, a arquitetura Rails oferece ao programador *web* um conjunto de serviços disponíveis, tais como a persistência de bases de dados independentes do fornecedor, geração automática de código para criação, leitura, atualização e eliminação de recursos ou testes integrados (Ignacio Fernández-Villamor et al., 2008).

3.3.2 *Comparação de Frameworks Web*

Por fim, apresenta-se um quadro comparativo com as *frameworks web* selecionadas na Secção 3.3.1 e com todas as características salientadas na Secção 3.2.2.

Tal como se pode ver na Tabela 3, Spring possui mecanismos de Segurança sendo uma das possíveis ferramentas o *Spring Security*. No caso da Computação em Nuvem é possível recorrer a *Spring Cloud*. Em ASP.NET MVC, um exemplo para utilização de Computação em Nuvem é Azure Cloud Services.

Em relação a Angular, para *debug* é normalmente utilizado Augury e em Wohlgethan (2018) é feita uma referência à existência de Documentação, de uma Comunidade e da possibilidade de integração de JavaScript.

A existência de mecanismos de Segurança em Django é mencionada por Curie et al. (2019) e alguns exemplos para Computação em Nuvem são referidos por Salas Zarate et al. (2015), sendo eles dotCloud, Google App Engine e *Amazon Elastic Compute Cloud* (Amazon EC2). Pelos mesmos autores, para Suporte HTML5 é referido HTML5 Boilerplate (H5BP), para *Debugging* a *Django Debug Toolbar*, para ORM a Django ORM e para JavaScript é mencionado jQuery, ExtJS e Dojo. Por fim, ainda pelos mesmos autores, é referida a Documentação de Django, nomeadamente Sphinx. O *Template Framework* e as Plataformas Suportadas (Windows, Linux, macOS) são referidas por Curie et al. (2019).

No caso de Ruby on Rails, a possibilidade do desenvolvimento da camada de apresentação é mencionada por Ignacio Fernández-Villamor et al. (2008) e de Segurança por Curie et al. (2019). Salas Zarate et al. (2015) menciona para Computação em Nuvem Amazon EC2, Linode, Rackspace e Heroku, para *Debugging to_yaml* e *inspect* e para ORM é ActiveRecord. Por fim, pelos mesmos autores, para JavaScript é referido jQuery, *script.aculo.us* e Dojo e o nome da Documentação é Rdoc. A inexistência de suporte de HTML5 é mencionada por Curie et al. (2019) e por Salas Zarate et al. (2015) e as Plataformas Suportadas (Windows, Linux, macOS) por Curie et al. (2019).

Tanto Vue.js como React apresentam curvas de aprendizagem bastante positivas e são suportadas pelas plataformas Windows, Linux e macOS. Ambas suportam todas as características definidas, de acordo com os vários autores já mencionados.

	Spring	**ASP.NET MVC**	**Angular**	**Django**	**Vue.js**	**React**	**Ruby on Rails**
Apresentação	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Segurança	Sim (Spring Security)	Sim	Sim	Sim	Sim	Sim	Sim
Computação em Nuvem	Sim (Spring Cloud)	Sim (Azure Cloud Services)	Sim	Sim (dotCloud, Google App Engine e Amazon EC2)	Sim	Sim	Sim (Amazon EC2, Linode, Rackspace e Heroku)
Suporte HTML 5	Sim	Sim	Sim	Sim (HTML 5 Boilerplate, H5BP)	Sim	Sim	Não
Template Framework	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Testes	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Suporte de Plataforma	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS
Debugging	Sim	Sim	Sim (Augury)	Sim (Django Debug Toolbar)	Sim	Sim	Sim (debug, to_yaml e inspect)
ORM	Sim	Sim	Sim	Sim (Django ORM)	Sim	Sim	Sim (ActiveRecord)
JavaScript	Sim	Sim	Sim	Sim (jQuery, ExtJS, Dojo)	Sim	Sim	Sim (jQuery, script.aculo.us Dojo)
Documentação	Sim	Sim	Sim	Sim (Sphinx)	Sim	Sim	Sim (Rdoc)
Comunidade	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Validação	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Usabilidade/ Curva de Aprendizagem	Boa	Boa	Muito boa	Boa	Muito boa	Muito boa	Boa

Tabela 3: Quadro Comparativo de *Frameworks*.

3.4 DISCUSSÃO

Um dos grandes objetivos com esta análise de *frameworks web* é escolher a *framework* a utilizar para a geração da aplicação a partir dos protótipos. Como a finalidade é focar na interface gráfica, de todas as *frameworks* mencionadas, as mais relevantes a ter em consideração são as *frameworks* com foco na camada de interface. Desta forma, Angular, Vue.js e React sobressaem em detrimento das restantes.

A partir das descrições apresentadas na Secção 3.3.1 e na Tabela 3, constata-se que essas três *frameworks* têm características semelhantes, e, desta forma, todas são adequadas para o desenvolvimento da camada UI e qualquer uma seria uma boa escolha. Neste caso, optou-se por escolher Vue.js como a *framework* a utilizar no processo.

Para complementar a utilização do Vue.js e para facilitar o processo de *layout* e a responsividade, podem ser utilizadas bibliotecas CSS. Neste caso, foi escolhido Bootstrap[9].

9 https://getbootstrap.com/. Consultado a 15 de julho de 2021.

4

ARQUITETURAS DE UMA APLICAÇÃO

Uma aplicação *web* tipicamente divide-se em três camadas, tal como já referido no capítulo anterior. O foco da presente dissertação está na camada de interface. Para abordar esta camada, após a análise das sete *frameworks* já apresentadas foi escolhida, como já mencionado, a *framework* Vue.js para ser a *framework* a utilizar para a geração do código do protótipo e, consequentemente, para um estudo mais aprofundado do funcionamento de uma *framework web*.

Na interface de uma aplicação *web* existem arquiteturas a vários níveis, nomeadamente arquitetura de informação, também conhecida como arquitetura de interface, que corresponde ao que é apresentado na interface, a arquitetura de componentes, que corresponde aos grandes blocos que compõe a interface em tempo de execução (criando a apresentação pretendida) e a arquitetura de pastas e código-fonte. Desta forma, no desenvolvimento de um novo projeto em Vue.js é importante utilizar uma estrutura de pastas, arquitetura de componentes e, de uma forma geral, seguir as convenções de *naming* (*coding style*) adequadas (Adittane, 2018).

Este capítulo define metamodelos para os níveis da arquitetura de informação (expressa num protótipo) e da arquitetura de componentes do Vue.js e é feita uma análise da estrutura de pastas. É utilizado um *mockup* como exemplo. No caso do metamodelo da estrutura de um protótipo o objetivo é perceber melhor no que consiste um protótipo e como este se encontra estruturado de modo a servir como base para a posterior interpretação de um *mockup*. No caso do metamodelo da arquitetura de componentes do Vue.js, o objetivo é perceber melhor como se desenvolve uma aplicação em Vue.js e respetivas peças necessárias.

4.1 MOCKUP INTERFACE GRÁFICA

Antes de se entrar em detalhe nos objetivos deste capítulo, começa-se por um pequeno exemplo de *mockup* de interface gráfica, onde se podem desde já identificar algumas características de páginas *web*. Na Figura 14 encontra-se um *mockup* de baixa fidelidade de uma das páginas do projeto Serve.Me[1] (plataforma de procura e contratação de serviços). Tal como se pode ver, numa página *web* é possível identificar diferentes componentes e navegação. É comum uma página ter uma *navbar*, que normalmente tem um logótipo (imagem) e botões. O corpo da página pode ter outras imagens, texto e botões ou hiperligações, que por sua vez podem redirecionar

1 https://github.com/catarinamachado/serve.me/. Consultado a 2 de junho de 2021.

o utilizador para outras páginas, como a página de *Login* ou Registo. Podem ainda existir outros componentes como um *footer*.

Figura 14: Protótipo da *landing page* do projeto Serve.Me[1]

Para desenhar o protótipo de uma interface gráfica, pode recorrer-se a diferentes ferramentas de *mockups*, tal como mencionado na Secção 2.3, e na presente dissertação será utilizado como exemplo o Adobe XD, pelos motivos também já mencionados.

4.2 ESTRUTURA DE UM PROTÓTIPO

Utilizando o protótipo da Figura 14 como exemplo ilustrativo, consegue-se identificar desde já algumas peças constituintes na estrutura de um protótipo. É possível perceber que a estrutura de um protótipo terá que ter botões, texto e imagens. Estes são componentes simples, que por si só já podem ser vistos como peças independentes. A estrutura também terá obrigatoriamente que ter uma *navbar* e um *footer*, que podem ser vistos como componentes compostos, uma vez que agregam outros componentes. No protótipo ilustrativo os componentes não têm cor (trata-se de um protótipo de baixa fidelidade), mas nas páginas *web* é comum estes componentes terem cores e tamanhos diferentes, logo, estas terão que ser características associadas aos componentes.

Também é possível analisar que apesar de só estar a ser apresentada uma página, o protótipo tem mais do que uma página, nomeadamente as páginas que se podem inferir através dos botões (Páginas de *Login* e de Registo).

A Figura 15 apresenta a estrutura geral de um protótipo. Em suma, um protótipo de uma aplicação *web* consiste em vários *artboards*, ou seja, várias páginas de protótipos. Cada uma destas páginas contém componentes, que podem ser componentes simples, tais como uma imagem, uma caixa de texto, um botão ou texto, ou podem ser um conjunto de componentes, normalmente dentro de um *container*, tais como uma *navbar*, um *footer*, um formulário ou uma tabela. Cada um destes componentes tem propriedades tais como o tamanho, a cor e um identificador (ID).

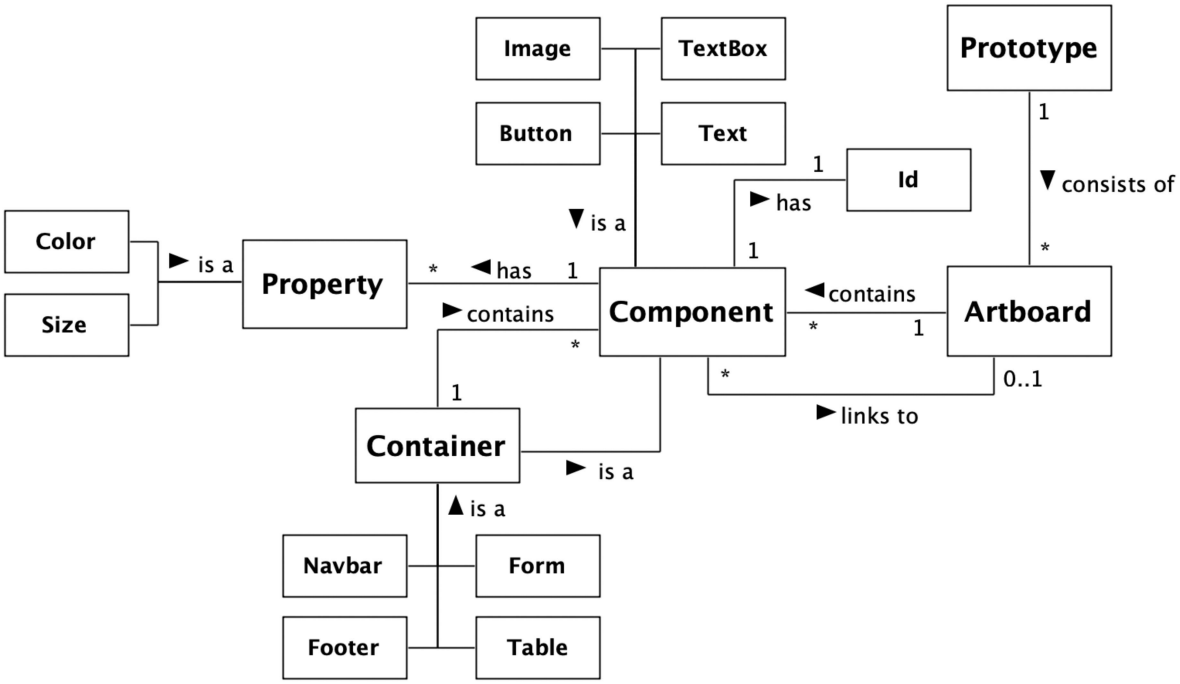


Figura 15: Estrutura de um protótipo.

4.3 METAMODELO DA ARQUITETURA DE COMPONENTES VUE.JS

O objetivo é definir um metamodelo que caracteriza a arquitetura de uma aplicação desenvolvida em Vue.js. Em Vue.js, uma arquitetura de componentes típica para identificar a interface tem vários componentes, *routing*, entre outros.

Assim, o próximo passo consiste em esquematizar, através de um diagrama, como é que se desenvolve uma aplicação em Vue.js e as respetivas peças necessárias. Desta forma, através da leitura da documentação de Vue.js (Vue.js, 2021), da análise de projetos bem conceituados em Vue.js[2], o projeto VUEMMERCE[3] e o

2 https://www.bacancytechnology.com/blog/top-21-amazing-vuejs-projects/. Consultado a 25 de março de 2021.
3 https://github.com/ivanlori/Vuemmerce/. Consultado a 4 de abril de 2021.

projeto COPILOT[4] (explicados em detalhe na Secção 4.6), apresenta-se de seguida o diagrama que representa a estrutura de uma aplicação em Vue.js (Figura 16).

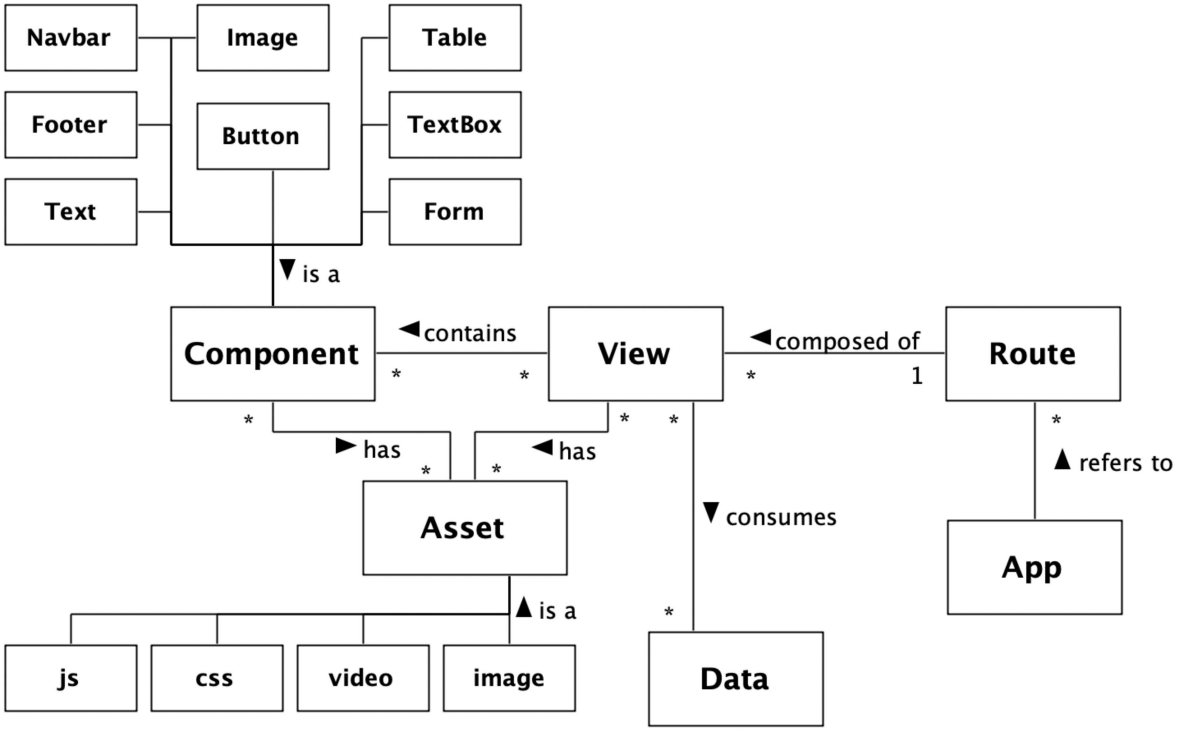


Figura 16: Metamodelo da Arquitetura de Componentes Vue.js.

Aproveitando novamente o protótipo da Figura 14, pode-se tentar perceber como é que este seria implementado em Vue.js. Para começar, este protótipo tornar-se-á numa aplicação *web*, que terá que ter *routing* entre os botões e as páginas a que estes hiperligam. Terá que ter componentes, identificados anteriormente aquando da caracterização do protótipo (*navbar*, botões, texto, entre outros), que serão incorporados em conjunto numa página e darão origem a páginas *web* recheadas de informação. Estes componentes terão personalização (cor, entre outros), a que na linguagem *web* se dá o nome de *assets* de um projeto.

Em suma, analisando a Figura 16 interpreta-se que quando se cria um projeto em Vue.js, começa-se por criar **componentes**, nomeadamente *navbars*, *footers*, texto, imagens, botões, tabelas, caixas de texto, formulários, menus, *modals*, barras de pesquisa, listas, gráficos, entre outros itens que, posteriormente, são agregados nas ***views***, as páginas da interface gráfica que serão disponibilizadas aos utilizadores. Assim, uma ***view*** é composta por um ou mais **componentes** e poderá ter também outro código de apresentação. Para além disso, também consome dados para assim poder apresentar o resultado final ao utilizador, caso a página não seja estática.

Como forma de embelezar e enriquecer a aplicação, existem os *assets*, nomeadamente imagens, vídeos, CSS, JavaScript, entre outros elementos, que são utilizados pelos **componentes** e pelas ***views***.

4 https://github.com/misterGF/CoPilot/. Consultado a 7 de maio de 2021.

O ***routing*** da aplicação corresponde ao *workflow* das páginas e respetivos URLs (*Uniform Resource Locators*), ou seja, é através do ***route*** que fica especificado qual é o URL que permitirá visualizar uma respetiva ***view***.

4.4 ESTRUTURA DE PASTAS VUE.JS

4.4.1 *Estrutura de pastas Vue CLI*

Em seguida, apresenta-se a estrutura de pastas base de um projeto em Vue.js (Figura 17), que corresponde à estrutura que é gerada quando se procede à configuração de um novo projeto através do Vue CLI, utilizando o comando **`vue create nome-do-projeto`**[5].

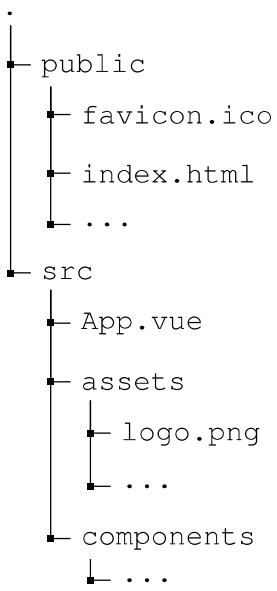


Figura 17: Estrutura de pastas Vue CLI.

Ao ser criado um projeto através do Vue CLI[6], são criadas duas pastas, ***public*** e ***src***. O objetivo da ***public*** é englobar ficheiros como o *favicon*, isto é, o *icon* da website e a base HTML (INDEX.HTML) que será usada para gerar o resto da aplicação. Todos os ficheiros e pastas presentes na pasta ***public*** podem ser acedidas diretamente via URL.

No caso da ***src***, o objetivo passa por agregar o ficheiro APP.VUE, o ficheiro principal, os *assets* do projeto, sejam imagens, SVGs, PDFs ou JSONs e componentes do projeto, como *navbars*, *footers* e outras peças de *frontend* necessárias para a apresentação da interface gráfica ao utilizador final.

5 https://cli.vuejs.org/guide/creating-a-project.html. Consultado a 27 de abril de 2021.
6 https://openclassrooms.com/en/courses/5664336-create-a-web-application-with-vue-js/6535063-create-a-new-project-with-vue-cli/. Consultado a 27 de abril de 2021.

Em suma, este é o esqueleto base de um projeto Vue.js quando utilizado o comando mencionado. Para uma melhor organização do projeto, e tendo ainda em conta a sua dimensão arquitetural, posteriormente são acrescentadas novas pastas.

4.4.2 *Estrutura de pastas padrão*

Criar um projeto através do Vue CLI fornece a estrutura de pastas base, mas para projetos maiores pode não ser suficiente. Desta forma, para aplicações de maior dimensão existem bibliotecas, por exemplo, o Vuex[7] que ajudam a dar mais estrutura ao projeto.

Como forma de aprofundar e melhorar a estrutura anterior, foi feita uma pesquisa com o objetivo de encontrar a melhor e mais completa estrutura de pastas para projetos de maior dimensão. Segundo Adittane (2018), que por sua vez analisou a documentação de Vuex[7] e projetos *open source* utilizados como exemplos de como estruturar aplicações Vue.js, como o caso dos projetos vue-hackernews-2.0[8] e RealWorld[9], a estrutura de pastas mais completa é a que se descreve em seguida (Figura 18).

Mais uma vez, cada um dos ficheiros e pastas tem uma diferente finalidade. De seguida, apresentam-se alguns detalhes sobre as pastas (Adittane, 2018):

- ***assets*** - pasta onde se colocam quaisquer itens que serão importados para os componentes (código CSS, por exemplo);
- ***components*** - pasta onde são colocados todos os componentes do projeto que, agregados, preenchem as páginas (as *views*);
- ***mixins*** - as *mixins* são as partes do código JavaScript que são reutilizadas em diferentes componentes. Numa *mixin* podem colocar-se os métodos de qualquer componente de Vue.js que serão fundidos com os do componente que o utiliza;
- ***router*** - todas as rotas do projeto (vulgarmente descritas no ficheiro INDEX.JS). As rotas permitem mapear URL em páginas da aplicação;
- ***store*** **(opcional)** - definições relacionadas com o *storage*;
- ***translations*** **(opcional)** - definições relacionadas com a tradução para diferentes línguas;
- ***utils*** **(opcional)** - pasta onde são colocadas funções auxiliares, tais como *regex*, constantes ou filtros;
- ***views*** - os componentes agregados dão origem a páginas, que por sua vez têm rotas definidas e podem ser acedidas através de algum URL.

Eventualmente podem ser adicionadas outras pastas dependendo da necessidade, tais como ***filters***, ***constants*** ou **API**.

7 https://vuex.vuejs.org/guide/structure.html. Consultado a 2 de maio de 2021.
8 https://github.com/vuejs/vue-hackernews-2.0/tree/master/src/. Consultado a 2 de maio de 2021.
9 https://github.com/mchandleraz/realworld-vue/tree/master/src/. Consultado a 2 de maio de 2021.

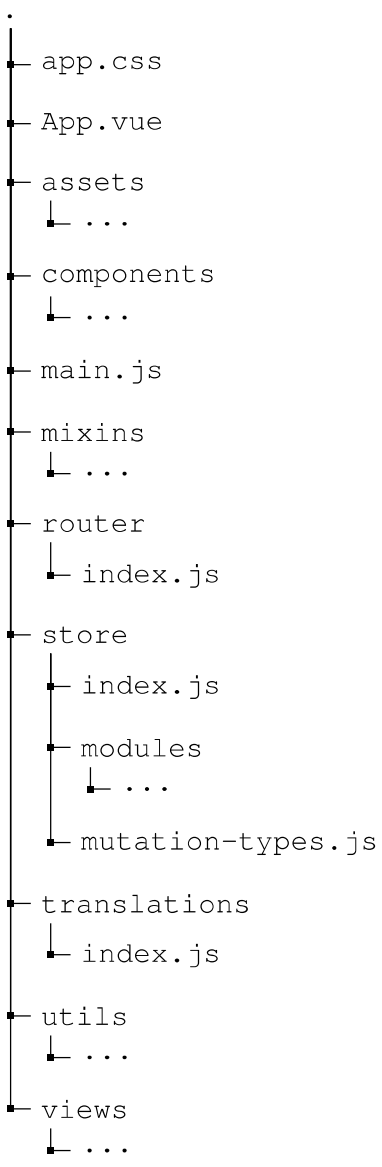


Figura 18: Estrutura de pastas padrão.

4.5 CÓDIGO VUE.JS

Analisando a Figura 16, encontram-se identificadas seis diferentes entidades, nomeadamente o Componente, a *View*, os Dados, os *Assets*, as Rotas e a *App*. O objetivo desta secção é explicar como se expressa cada uma delas em código Vue.js.

Partindo da estrutura de pastas padrão de um projeto Vue.js (ver Figura 18), os Componentes encontram-se definidos em ficheiros na pasta "components", as *Views* encontram-se definidas em ficheiros na pasta "views", os Dados, se aplicável, bem como os *Assets* estarão em ficheiros dentro da pasta "assets". Por fim, as *Routes* estarão descritas no ficheiro "index.js" da pasta "router" e a *App* no seu global no ficheiro "App.vue".

Em relação ao **Componente**, sempre que surge a necessidade da criação de um novo componente é necessário criar um novo ficheiro **.vue**, onde se deve colocar o código HTML dentro da *tag template*. Adicionalmente, pode ser utilizado também CSS dentro de uma *tag style*, tal como se pode ver em seguida.

```
<template>
    // codigo HTML
</template>

<style scoped>
    // codigo CSS
</style>
```

No caso da **View**, é desenvolvido o código final a apresentar ao utilizador. Neste ficheiro **.vue**, podem ser utilizados outros componentes através da respetiva *tag* e pode também ser acrescentada informação de estilo (código CSS) e *scripts* (código JavaScript) adicionais. Nos ficheiros JavaScript dos *assets* do projeto podem definir-se várias funções e métodos onde se pode recorrer a, por exemplo, APIs desenvolvidas pelos programadores para apresentar os dados na interface gráfica.

```
<template>
    // codigo HTML
</template>

<style scoped>
    // codigo CSS
</style>

<script>
    // codigo JavaScript
</script>
```

No caso dos **Dados**, para o consumo dos mesmos podem ser feitas, por exemplo, chamadas a uma API através da biblioteca de JavaScript *axios*.

Os ***assets*** são simplesmente ficheiros auxiliares que podem ser utilizados pelas *views* e pelos componentes. Exemplos típicos de *assets* são imagens ou ficheiros CSS, entre vários outros possíveis exemplos.

No que diz respeito à ***Route***, uma forma de representar as rotas necessárias é através de um ficheiro INDEX.JS na pasta ***router***. Neste ficheiro pode importar-se o Router ("vue-router") e definir o nome das rotas e a *view* que lhe corresponde. No exemplo que se segue são definidas rotas para a Página Inicial (*home*), para a Página de *Login* e para todas as outras páginas que possam ser pesquisadas será apresentada a Página de "Não encontrado".

```
import Vue from "vue";
import Router from "vue-router";

Vue.use(Router);

export default new Router({
  routes: [
    {
      name: "home",
      path: "/",
      component: () => import("@/views/Home")
    }
    ,
    {
      name: "login",
      path: "/login",
      component: () => import("@/views/Login")
    }
    ,
    {
      path: "*",
      component: () => import("@/views/NotFound")
      }
  ]
});
```

No caso da ***App***, o ficheiro começa com a *tag template*, onde é estabelecido de forma transversal o conteúdo de cada uma das páginas da interface gráfica. Por exemplo, pode ficar estabelecido que todas as páginas irão conter um determinado componente e que todas as páginas vão ter um determinado nome ou variável. São também feitos *imports* de CSS, JavaScript e componentes e podem também ser definidas algumas variáveis e funções.

```
<template>
  <div id="app">
    <vue-headful title="Titulo da Pagina"/>
    <TheNavbar/>
```

```
    <router-view></router-view>
    <TheFooter/>
  </div>
</template>

<script>
import './assets/styles/serve-me.css';
import TheNavbar from './components/TheNavbar.vue'
import TheFooter from './components/TheFooter.vue'

export default {
  name: 'App',
  props: {
    typeOf: String,
    nome: String
  },
  components: {
    TheNavbar,
    TheFooter
  }
}
</script>
```

Através do estabelecimento desta relação entre o código Vue.js necessário e as entidades essenciais para o desenvolvimento de uma aplicação, é percetível que há uma relação entre o modelo genérico de Vue.js e a geração da interface da gráfica.

4.6 EXEMPLIFICAÇÃO

Para exemplificar que uma aplicação típica em Vue.js segue as estruturas de pastas e de componentes anteriormente mencionadas, segue-se um exemplo de uma aplicação, retirada de uma lista de projetos bem conceituados em Vue.js[2], com o nome VUEMMERCE [3], um *template* de *e-commerce*.

Fazendo a relação deste projeto com o metamodelo presente na Figura 16 e com as estruturas de pastas previamente apresentadas, no VUEMMERCE, tal como se pode ver através da Figura 19, no que diz respeito aos ***assets***, existem dois ficheiros JavaScript, correspondentes a filtros e validações, que por sua vez são utilizados pelos ***components***, o que vai ao encontro do esperado. Na caso dos ***components***, existem vários componentes de visualização necessários para o *frontend* da aplicação, nomeadamente do tipo *footer*, *header*, *hero*, menu, *products_list*, *search* e *products*. Nesses ficheiros, para além do código de visualização, são também utilizados vários métodos que têm como objetivo povoar a interface gráfica, tal como esperado.

Por fim, no caso das ***views*** e do ***routing*** é nas pastas *layouts* e *pages*, juntamente com alguns dos ficheiros em ***components***, que se encontram especificadas as rotas a seguir quando se clica em cada uma das hiperliga-

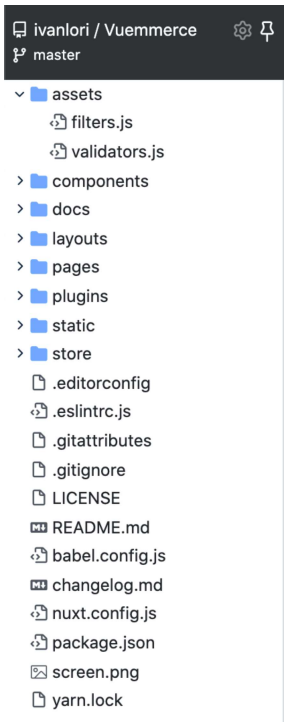


Figura 19: Vuemmerce Github Octotree.

ções presentes na página principal e é em ***layouts/default.vue*** que se encontra em pormenor o aspeto padrão das páginas.

Para além deste e de outros projetos, o projeto CoPilot[8] também pode ser utilizado como exemplo da estrutura de componentes apresentada na Figura 16 e da estrutura de pastas previamente apresentada.

Neste projeto, tal como se pode comprovar através da Figura 20, é dentro da pasta *src* que se encontram as pastas mais relacionadas com o *frontend* da aplicação. Nas outras pastas, encontram-se ficheiros mais relacionados com a instalação e *testing* do projeto.

Desta vez, o projeto não tem ***assets***. No caso dos ***components***, existe o *login*, o componente para o erro 404, alertas, entre vários outros relativos a notificações, mensagens e menus. Dentro da pasta *components*, encontram-se as ***views***, que por sua vez recorrem aos componentes e a código de apresentação para a sua constituição. No caso do ***routing***, é no ficheiro *routes.js* que se encontram especificadas todas as diferentes rotas do projeto.

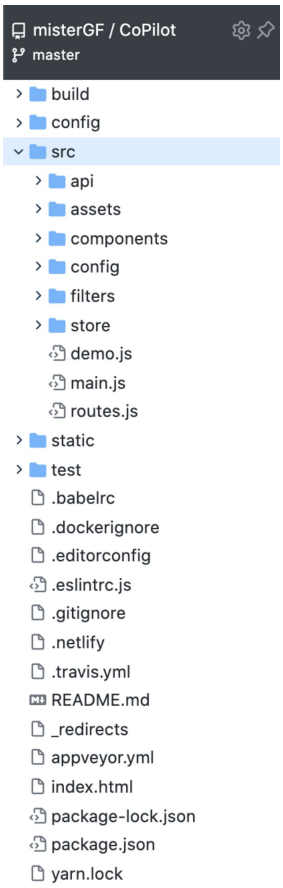


Figura 20: CoPilot Github Octotree.

4.7 CONCLUSÃO

Neste capítulo definiram-se metamodelos, quer de um protótipo de interface de utilizador, quer de uma aplicação *web* desenvolvida em Vue.js. O objetivo é suportar o processo de geração de uma interface em Vue.js, a partir do seu protótipo, que será tema do próximo capítulo.

O exemplo de um protótipo foi útil na medida em que serviu como um instrumento para ilustrar os metamodelos, tanto da estrutura geral de um protótipo, como da arquitetura de componentes de Vue.js apresentados ao longo do capítulo. Estes metamodelos são úteis para se perceber melhor o que se pode esperar de qualquer protótipo de interface gráfica desenvolvido por um *designer* e como se desenvolve uma aplicação em Vue.js, respetivamente. Logo, para perceber como ir de um protótipo a uma aplicação.

A estrutura de pastas de um projeto é um aspeto fundamental, principalmente na futura manutenção da aplicação, daí a importância do estudo de qual a melhor forma de se fazer a estruturação a nível das pastas de um projeto. Os dois exemplos apresentados no final do capítulo acabam por ser meramente ilustrativos em relação à estruturação de uma aplicação Vue.js.

5

DESCRIÇÃO GENÉRICA DA ABORDAGEM DE IMPLEMENTAÇÃO DA UI

Pode olhar-se para as *frameworks* como tendo um metamodelo, que permite decompor a *framework* em diferentes peças e esquematizar a forma como uma página *web* é desenvolvida através da *framework* (Figura 16, página 38). Para desenhar e idealizar interfaces gráficas recorre-se a protótipos (Figura 15, página 37). Analisando estes protótipos, pode-se deduzir que eles podem ser mapeados para o metamodelo da *framework web*.

Este mapeamento é possível, uma vez que, tanto na forma como a *framework web* é utilizada, como na forma como as interfaces gráficas são concebidas, existem componentes. Os componentes são pequenas peças que, quando adicionadas em conjunto, dão origem às páginas *web* que a população está habituada a navegar. Os componentes podem ser botões, imagens, formulários, tabelas, entre outros.

Após o *designer* desenhar o protótipo de uma *Graphical User Interface* (GUI), este terá de ser exportado para algum formato, neste caso via Adobe XD. Essa informação será então utilizada para criar o esqueleto do projeto (Vue.js + Bootstrap).

Este capítulo aborda, assim, a estratégia do desenvolvimento do projeto e faz uma descrição genérica da abordagem. O objetivo é apresentar os motivos pelos quais será utilizado SVG como estratégia de conversão do protótipo da interface gráfica, quais os componentes Vue.js que serão focados na solução de *software* e o método que será utilizado para mapear os protótipos no metamodelo de Vue.js e respetivo algoritmo genérico. O capítulo termina com a explicação dos requisitos aos quais a abordagem responde.

5.1 CONVERSÃO DO PROTÓTIPO DA INTERFACE GRÁFICA

Tal como já mencionado, o objetivo é, através de um protótipo de uma interface gráfica, gerar o código para a visualização das correspondentes páginas *web*. Com o protótipo da interface gráfica desenvolvido, é necessário escolher um formato para a exportação do mesmo. As opções possíveis são os formatos *bitmap* (PNG, JPG, TIFF e WebP), os formatos vetoriais (SVG, PDF e EPS) e os formatos *web* (HTML e CSS).

A solução mais simples seria exportar diretamente o *mockup* para o formato *web* (HTML e CSS) e, instantaneamente, ter-se-ia o código da interface gráfica, passível de ser integrado numa aplicação *web*. No entanto, esta funcionalidade apresenta várias insuficiências, uma vez que o código gerado está orientado para a criação de uma visualização para os utilizadores e não para a implementação da interface gráfica. Isto significa que o código não se encontra bem estruturado, sendo consequentemente difícil de aproveitar e ainda mais complicado

de manter. Em resumo, da perspetiva do desenvolvimento da interface gráfica, utilizar esta funcionalidade tem como consequência um fraco suporte à geração de código da interface gráfica.

Para colmatar esta insuficiência, tendo em consideração as opções de exportação de um protótipo numa ferramenta de *design*, uma alternativa é exportar o *mockup* para SVG, um formato gráfico vetorial para a *web*. O SVG é criado em XML, e não num formato binário fechado (W3C, 2011). Assim, converter um *mockup* para SVG permite-nos ter acesso ao código XML do *design*. Com este ficheiro XML, o objetivo é identificar os componentes e as suas respetivas posições, com o intuito final de gerar automaticamente o código para a página *web*.

Contudo, a exportação para SVG também não é uma solução perfeita, uma vez que não foi desenvolvida para a representação de protótipos. Desta forma, a principal contrapartida é perder-se informação durante o processo de exportação. Por exemplo, perde-se a informação relativa à navegação entre páginas e é necessário inferir do desenho que elementos compõe um protótipo (um botão não é diretamente identificado como um botão, sendo sim um desenho de um retângulo).

Os formatos *bitmap* (PNG, JPG, TIFF e WebP) são utilizados para visualizar imagens. É possível ler e interpretar os píxeis dos ficheiros das imagens, contudo, como o píxel é uma peça de informação ainda mais pequena do que o providenciado pelo SVG, o esforço de interpretação seria consideravelmente superior à solução apresentada anteriormente. Assim sendo, a melhor opção é utilizar SVG como formato de exportação.

Na Figura 16 (página 38) encontra-se um metamodelo da arquitetura de componentes de Vue.js, onde se encontram identificados diferentes componentes que uma aplicação *web* em Vue.js pode englobar. Face ao elevado número de componentes, foi decidido, numa primeira fase, tratar aqueles que são o núcleo base de um protótipo, nomeadamente **Texto**, **Imagem** e **Botão**, bem como três exemplos de componentes compostos, nomeadamente ***Navbar***, ***Footer*** e **Formulário**. Podem facilmente ser adicionados novos componentes em posteriores atualizações da solução de *software* proposta (na Secção 6.3.6 é explicado como isso pode ser feito).

5.2 MAPEAMENTO ENTRE OS PROTÓTIPOS E O METAMODELO

Após o *designer* desenhar o protótipo de uma GUI, este terá de ser exportado para SVG, neste caso via Adobe XD. O formato SVG pode ser lido como um ficheiro XML. A informação no SVG é então utilizada para criar o esqueleto do projeto (Vue.js + Bootstrap).

5.2.1 *Mapeamento de Componentes*

Para resolver o problema de perda de informação, devido ao facto dos *mockups* serem exportados para SVG, como abordado anteriormente, algumas suposições sobre o *input* têm que ser feitas. Em particular, os componentes utilizados no protótipo (que terão de ser mapeados para a implementação) devem ser identificáveis pelos seus IDs. Assim, o *designer* tem que utilizar a notação apresentada abaixo no campo do ID dos componentes no Adobe XD (Figura 21). Desta forma, a estratégia passa por utilizar o ID do componente para estabelecer

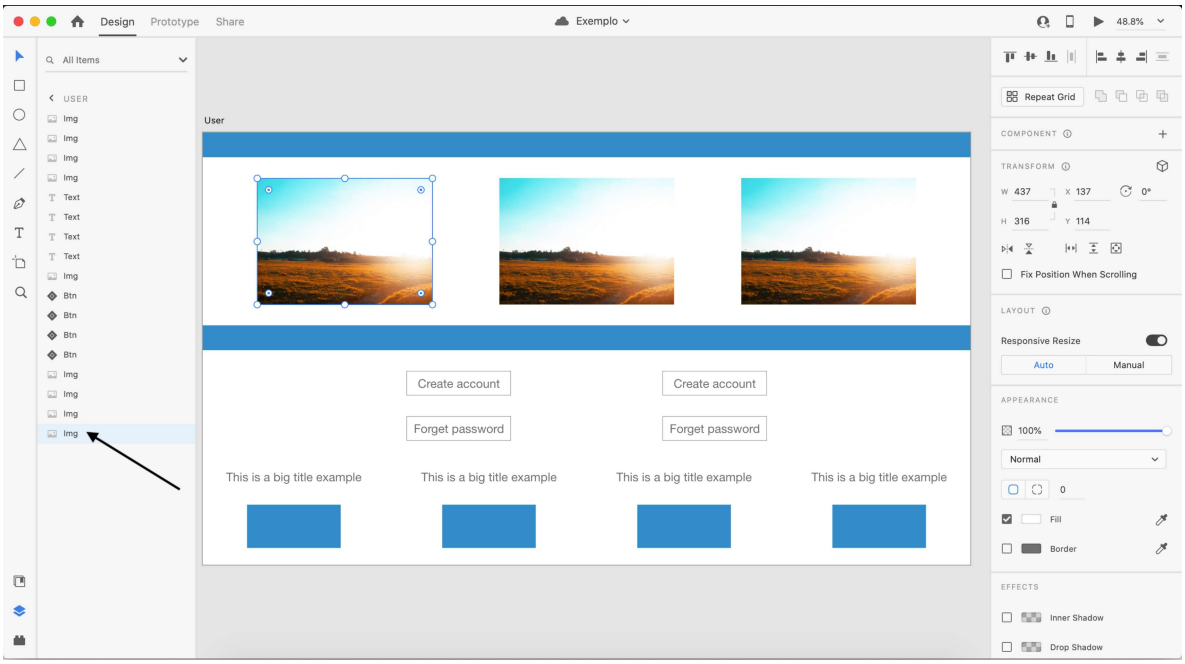


Figura 21: Campo de texto onde se altera o ID do componente no Adobe XD.

uma correspondência entre os elementos SVG (por exemplo, um retângulo) e os componentes do protótipo (por exemplo, um botão ou uma caixa de texto). Caso contrário, perdia-se a informação sobre se, por exemplo, um retângulo representa um botão ou uma caixa de texto. De entre os três componentes simples mencionados, no caso do botão, como se trata de texto com um retângulo no seu redor é necessário um passo adicional no Adobe XD, nomeadamente selecionar o texto e o retângulo e clicar em "Make Component". Assim, estes componentes ficam interligados e no SVG gerado ficam dentro da mesma *tag*.

Os *containers* (*Navbar*, *Footer* e Formulário) são utilizados para se poder perceber se um conjunto de componentes está agrupado. Por exemplo, se várias caixas de texto e títulos são afinal um formulário ou simplesmente elementos soltos na interface gráfica. Neste caso, por uma questão de simplificação da interpretação do SVG gerado, interligar os componentes através do "Make Component" não seria a melhor opção uma vez que o SVG gerado ficaria demasiado complexo, com a possibilidade de existirem *tags* dentro de *tags*, e assim sucessivamente, pois não se sabe à priori quantos componentes poderão estar contidos no *container* (enquanto que no caso do botão se sabe exatamente como este estará definido no SVG). Desta forma, o método utilizado, para identificar o conteúdo de um *container*, é saber qual é a posição do mesmo no *design* e, ao analisar as posições dos componentes da interface, perceber se estes se sobrepõe completamente, ou não. Em caso afirmativo, significa que o componente está contido no *container*. Assim, é possível inferir diretamente o componente (*Navbar*, *Footer* e Formulário).

Em suma, através dos IDs no protótipo, determina-se que componente deve ser usado no Vue.js. Para componentes simples como os botões, este mapeamento é direto; para componentes mais complexos (que agregam vários componentes num só), pode deduzir-se qual utilizar através do ID do *container* que os agrega no protótipo (exemplo na Figura 21).

Um protótipo pode ter muitos componentes, nomeadamente todos os componentes fornecidos pelo Bootstrap, mas nesta fase foram selecionados apenas alguns para simplificar os diagramas e a explicação. Em seguida, encontra-se novamente a lista dos componentes explorados na presente solução e a anotação (ID) que o *designer* deve utilizar para cada um deles no Adobe XD:

- Texto - "Text" (componente simples);
- Imagem - "Img" (componente simples);
- Botão - "Btn" (componente simples);
- Caixa de Texto - "TextBox" (componente simples);
- Navbar - "Navbar" (*container* de componentes);
- Footer - "Footer" (*container* de componentes);
- Formulário - "Form" (*container* de componentes).

5.2.2 *Mapeamento de Páginas*

De forma a mapear um protótipo numa implementação Vue.js, previamente tem que se mapear a estrutura do protótipo (Figura 15) no metamodelo do Vue.js (Figura 16). Uma vez analisado o mapeamento dos Componentes, analisa-se, agora, o das Páginas. Normalmente, uma aplicação *web* tem mais do que uma página e navegação entre elas. Usando Adobe XD, podem-se prototipar várias páginas e a respetiva navegação entre elas. No entanto, quando exportada para SVG, esta informação perde-se e só se fica com o conhecimento de quais páginas existem, mas não com a informação sobre a navegação entre cada uma delas. Para resolver este problema, será necessário expressar esta navegação num ficheiro à parte, neste caso, descrito em XML. Este ficheiro terá que ter a informação do nome da página onde se encontra o componente que se pretende hiperligar, o ID do respetivo componente e, associado a este, o nome da página para onde o utilizador deverá ser redirecionado após o clique (solução apresentada em detalhe na Secção 6.2.1).

5.2.3 *Mapeamento do layout*

A informação sobre o posicionamento dos elementos está escrita em termos de posições absolutas, no SVG. Definir o *layout* de uma página *web* desse modo, no entanto, é considerado má prática. Assim, outro grande desafio é não utilizar as posições absolutas na descrição da disposição dos componentes da interface gráfica. Para tal, pode ser utilizado o sistema de *grid* do Bootstrap. Através das posições absolutas descritas no ficheiro XML é possível deduzir em quantas colunas (12 no máximo) e em quantas secções é que a página está dividida e depois em qual dessas secções/colunas o componente está localizado.

5.2.4 *Resumo da Abordagem*

Em resumo, o algoritmo que a abordagem deve seguir está dividido em duas partes: posição e componentes.

Começando pelo posicionamento, os passos são os seguintes:

- Dividir o protótipo em secções (linhas);
- Dividir o protótipo em colunas;
- Fazer o esqueleto das secções e colunas em Vue.js.

Depois de ter o esqueleto completo, resta colocar os diferentes componentes nas suas respetivas posições. Assim, no que diz respeito aos componentes, o algoritmo é o seguinte:

- Corresponder o ID do componente e o código do componente;
- Fazer as alterações necessárias à cor e possível navegação do componente.

É de salientar que para considerar mais componentes, simplesmente deve indicar-se ao *designer* a sua anotação (ID) e adicionar este mapeamento entre o ID e o código a ser gerado ao algoritmo (que será explicado em pormenor na Secção 6.3.6).

5.3 REQUISITOS DE UMA FERRAMENTA QUE SUPORTE A ABORDAGEM

Uma vez explicado o mapeamento entre os protótipos e o metamodelo, os aspetos relevantes a ter em consideração nesta abordagem e respetivas soluções e, por fim, o algoritmo genérico da implementação, o objetivo da presente secção é resumir todos os requisitos que a solução de *software* assume.

Em primeiro lugar, o protótipo da interface gráfica deve ser interpretado como um projeto, isto é, um só projeto com os vários *mockups* que correspondem às diferentes páginas desse mesmo projeto, compostos por uma *navbar* e um *footer* comum a todas elas (não sendo obrigatório ter ambos) e com a respetiva navegação entre as páginas (tal como apresentado no diagrama da Figura 15, página 37). Para isso, é necessário recolher do *designer* qual o nome de todas as páginas que pertencem ao projeto, qual das diferentes páginas corresponde à página inicial (*homepage*) e, devido ao problema já identificado da perda de informação na navegação entre as páginas, fornecer um ficheiro de texto XML com a informação da navegação. Desta forma, quando a solução de *software* proposta for invocada, todos estes argumentos devem ser fornecidos.

No que diz respeito aos componentes, é necessário perceber se o elemento que se está a interpretar se trata de um componente simples ou de parte de um componente composto. Tal como já explicado, um componente composto é constituído por vários subcomponentes, por exemplo, uma *navbar*, assim como um *footer*, podem conter imagens, botões e texto e um formulário à partida será composto por texto, caixas de texto e botões (Figura 15, página 37). Nestes casos, no Adobe XD, o *designer* terá que delimitar, através de um desenho de um retângulo, os componentes que compõem o componente composto e utilizar o campo do ID para informar

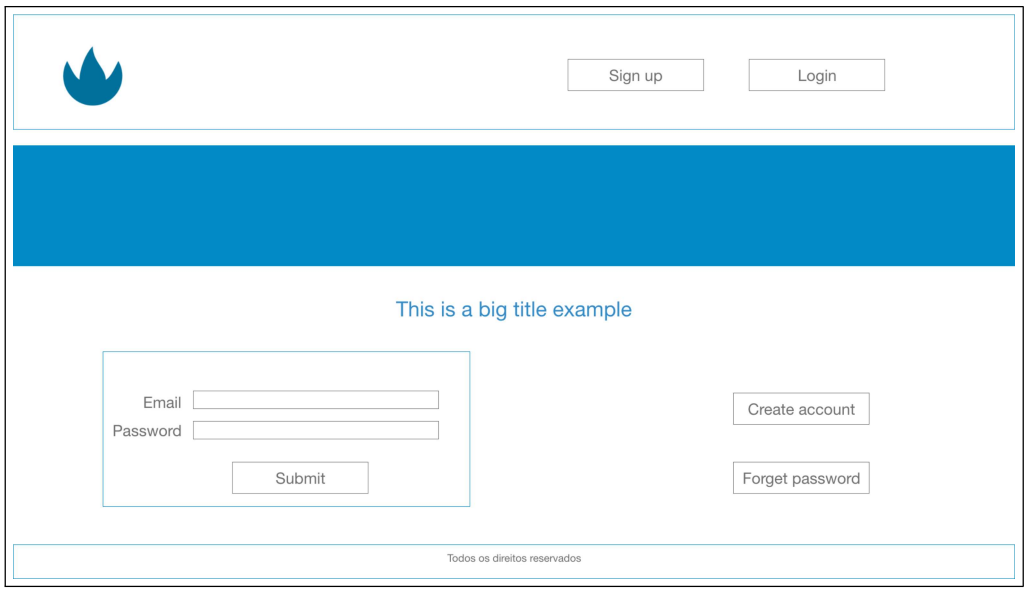


Figura 22: Exemplo de protótipo em Adobe XD. Os retângulos azuis não fazem parte do *design*, sendo apenas utilizados para delinear *containers*.

de que componente se trata. Na Figura 22 encontra-se um exemplo de como fica o aspeto da interface gráfica com esta delimitação.

Assim, a solução de *software* proposta deve ser capaz de interpretar os IDs dos componentes e determinar em cada caso se se trata de parte de um componente composto, ou de um componente simples e independente.

Para além disso, no Adobe XD os componentes podem ser personalizados através de cores (a cor é uma das propriedades do componente, tal como apresentado no diagrama da Figura 15, página 37). Nos componentes escolhidos para a presente solução de *software*, existe o exemplo dos botões, que podem ter um fundo e um contorno de determinadas cores, a *navbar* e o *footer* que por sua vez também podem ter um fundo de alguma cor em particular, bem como o texto em geral, seja de um formulário, de um botão, da *navbar*, do *footer* ou simplesmente texto. Estas informações de personalização devem ser geradas para um ficheiro *.css* comum como *asset* do projeto (no diagrama da Figura 15, página 37, corresponde às propriedades de um componente, já no diagrama do metamodelo, Figura 16, página 38, isso traduz-se nos *assets* de um projeto).

Em relação à navegação entre páginas, a solução de *software* tem que ser capaz de interpretar esse ficheiro fornecido pelo *designer*, que por sua vez deve obedecer a um formato já predefinido. Uma vez essa informação interpretada, os componentes terão que ser gerados com a informação de *routing* no respetivo código Vue.js. Ou seja, este é mais um ponto a ter em consideração ao gerar-se um componente para além dos pontos já mencionados de se determinar de que tipo de componente se trata, se é um componente simples, se é um subcomponente ou se é um componente composto e se contém personalização.

Existem componentes que podem ser comuns a várias páginas e, nesse caso, como boa prática o ideal seria não repetir esse código diversas vezes ao longo do projeto, mas sim reaproveitá-lo. Nesta primeira iteração da solução de *software* proposta, deduz-se à priori que a *navbar* e o *footer* serão componentes que serão utilizados mais do que uma vez no mesmo projeto. Como tal, estes componentes devem ser gerados com este intuito e

serem reaproveitados várias vezes ao longo do projeto Vue.js (caso o projeto tenha mais do que apenas uma página).

Por último, no que diz respeito às posições dos componentes, tal como mencionado, o objetivo é não gerar componentes a utilizar posições absolutas, apenas posições relativas, e aproveitar o sistema de *grid* do Bootstrap para tal. Assim, o objetivo é gerar componentes que utilizem as *tags* de *row* e *col* do Bootstrap e, para tal, determinar em quantas colunas se divide cada linha, com base nas posições absolutas dos componentes, quantos componentes estão em cada coluna e em cada linha, entre outros.

Para além disso, como a *tag* "Text" no SVG não tem a informação da largura do componente (como abordado na secção seguinte), o *designer* deve poder indicar qual é a largura que deseja que o texto ocupe. Para tal, pode utilizar o campo do ID para indicar essa informação, ficando assim com a estrutura "Text-w100", para o caso de querer que o texto ocupe 100px.

5.4 SUMÁRIO

Neste capítulo foi descrita de uma forma geral a estratégia de desenvolvimento do projeto. Apresentam-se os motivos pelos quais se optou pelo SVG como formato de exportação de protótipos e mostra-se como é possível mapear os componentes dos protótipos para Vue.js, mapear a navegação entre as páginas e mapear o *layout* (sistema de *grid* do Bootstrap). No próximo capítulo será explicado em maior detalhe a respetiva implementação.

6

DETALHES DA IMPLEMENTAÇÃO DA UI

No presente capítulo começa por ser explicado como é que o ficheiro SVG é composto e onde é que se encontra a informação que se pretende retirar do mesmo.

Em seguida, é explicada em detalhe a solução de *software* desenvolvida. Para o desenvolvimento da solução foram implementados dois *scripts* em Python. O primeiro, intitulado "script", tem como objetivo criar o projeto Vue.js, fazer o "parse" do ficheiro com as informações de *routing* entre as páginas, correr o *script* "parsing" para todos os ficheiros SVG e criar todos os ficheiros de configuração necessários para o projeto Vue.js, bem como, por fim, executá-lo (processo explicado na Secção 6.2). O segundo, intitulado "parsing", tem como objetivo traduzir um e um só ficheiro SVG (exportado através do Adobe XD) para um ficheiro Vue.js (processo explicado na Secção 6.3).

6.1 TAGS SVG

Para o desenvolvimento da solução de *software* proposta é necessário numa fase inicial analisar que informação é apresentada no ficheiro SVG e de que modo ela é apresentada. Em Adobe XD, tal como se pode ver no menu da esquerda da Figura 21, são várias as opções de figuras que podem ser desenhadas, nomeadamente um retângulo, um círculo, um triângulo, uma linha reta, linhas variáveis, texto ou imagens. Posteriormente, quando se converte um *mockup* para SVG, as opções de desenho mencionadas corresponderão às seguintes *tags*:

- Retângulo - corresponde à *tag* **rect** em XML;
- Círculo - corresponde à *tag* **ellipse** em XML;
- Triângulo - corresponde à *tag* **path** em XML;
- Linha reta - corresponde à *tag* **line** em XML;
- Linha variável - corresponde à *tag* **path** em XML;
- Texto - corresponde à *tag* **text** em XML;
- Imagem - corresponde à *tag* **image** em XML.

Como forma de simplificar a solução, nesta fase o foco, relativamente às formas geométricas, será colocado nos retângulos, uma vez que são a forma de utilização mais comum. Assim, outras figuras serão, para já, tratadas como retângulos. No que diz respeito às restantes opções, o objetivo prender-se-á em decifrar a qual dos componentes Vue.js é que cada uma das *tags* XML pode dar origem com a ajuda do ID introduzido pelo *designer*.

Como forma de efetuar uma melhor análise da informação disponibilizada pelo SVG em relação a cada componente, de seguida apresentam-se excertos dos componentes Texto, Botão e *Navbar*.

6.1.1 *Texto*

Para começar, apresenta-se um excerto SVG do componente Texto com o conteúdo "Nome":

```
<text id="Text-3"
      data-name="Text"
      transform="translate(286 753)"
      fill="#338bc7"
      font-size="30"
      font-family="HelveticaNeue, Helvetica Neue">
      <tspan x="-36.675" y="0">Nome</tspan>
</text>
```

Analisando o excerto apresentado, nota-se que o texto é representado pela *tag* “text”. A informação que se pode retirar dos respetivos atributos é a seguinte:

- id: ID introduzido pelo utilizador no Adobe XD; no entanto, como o ID tem que ser único, aquando da exportação para SVG, o Adobe XD adiciona um traço (“-”) e um número (começa em 1 e vai incrementando) ao ID inserido pelo utilizador;
- data-name: gerado automaticamente aquando da exportação. Na verdade este atributo guarda o ID introduzido pelo utilizador no Adobe XD, sem quaisquer alterações. É de notar que o primeiro componente de cada tipo não tem o atributo “data-name”, uma vez que o ID exportado não vai ser alterado;
- transform: informação da posição inicial do componente representado por coordenadas cartesianas;
- fill: neste caso, a cor do texto;
- font-size: tamanho da letra do texto;
- font-family: tipo da letra do texto;
- *tag* tspan: contém o texto efetivamente escrito pelo utilizador.

Para a conversão para código Vue.js é essencial observar os atributos *id* e *data-name* para identificar o componente, o atributo *transform* para deduzir a sua posição relativa, o *fill* para determinar a cor e, por fim, o texto presente no componente.

É de notar que o componente texto não tem um atributo "largura", pelo que será necessário deduzir a mesma de alguma forma. Uma dessas formas é através da estratégia apresentada na Secção 5.3, isto é, através da introdução dessa informação no ID do componente (exemplo, "Text-w100").

6.1.2 *Botão*

Em seguida, apresenta-se o exemplo de um botão no SVG:

```
<g id="Btn-3" data-name="Btn" transform="translate(420 856)">
    <g id="Btn-4" data-name="Btn">
        <g id="Rectangle-2" data-name="Rectangle" fill="#f7eded"
           stroke="#707070" stroke-width="1">
          <rect width="261" height="61" stroke="none"/>
          <rect x="0.5" y="0.5" width="260" height="60" fill="none"/>
        </g>
    </g>
    <text id="Text-5" data-name="Text" transform="translate(83 42)"
    fill="#707070" font-size="30" font-family="HelveticaNeue,
    Helvetica Neue"><tspan x="0" y="0">Submeter</tspan></text>
</g>
```

Neste caso, observa-se que existe uma *tag* "g" com um ID "Btn-3" que corresponde a um botão. Desta forma, sabendo à priori que se trata de um botão, consegue-se aplicar diretamente o componente botão presente na biblioteca do Bootstrap e efetuar as alterações necessárias. Assim, é importante observar a posição absoluta do componente (*transform*) e na *tag* "g" aninhada observar o *fill* (neste caso, é a cor do fundo do botão) e o atributo *stroke* (corresponde à cor do contorno do botão). Em seguida, é necessário observar a *tag text* aninhada, de modo a identificar a cor do texto e qual é efetivamente o texto. O mesmo acontece para os restantes componentes, tendo por base as suas características.

6.1.3 *Navbar*

Por fim, apresenta-se um excerto do código SVG de uma *Navbar*. Neste caso em específico, trata-se de uma *Navbar*, mas podia ser outro tipo de *container*, como um *Footer* ou um *Form* (a estrutura SVG é a mesma porque se trata simplesmente de um retângulo em torno dos componentes constituintes, tal como se pode ver na Figura 22).

```
<g id="Navbar"
    fill="#d9d8d8"
```

```
    stroke="#008bc7"
    stroke-width="1"
    style="mix-blend-mode: darken;isolation: isolate">
    <rect width="1920" height="220" stroke="none"/>
    <rect x="0.5" y="0.5" width="1919" height="219" fill="none"/>
</g>
```

Neste caso, o importante é identificar qual é o componente de que se trata, a cor do *background*, quais são as coordenadas de início do componente, a altura (atributo *height*) e a largura (atributo *width*) do componente.

É de notar que se um componente iniciar na posição (0,0) o campo *transform* não é apresentado no SVG.

6.2 SCRIPT SVG - CONFIGURAÇÃO DO PROJETO VUE.JS

Existem algumas informações e ficheiros que têm obrigatoriamente de ser fornecidos pelo utilizador, nomeadamente os seguintes:

- Nome do projeto;
- Nome do ficheiro SVG que deverá ser a página inicial da aplicação *web* resultante;
- Lista dos nomes dos ficheiros SVG que irão constituir o projeto;
- Nome do ficheiro com as informações de *routing*.

Desta forma, para um projeto intitulado “Exemplo”, com o ficheiro “Home.svg” como página principal, os ficheiros “Login.svg” e “Register.svg” como outras páginas do projeto e um ficheiro ”infos.xml“ com a informação do *routing*, a invocação do *script* deverá ser feita da seguinte forma:

```
./script Exemplo Home.svg "['Login.svg','Register.svg']" infos.xml
```

A primeira missão deste *script* é tratar os parâmetros recebidos e colocá-los em variáveis de instância. Em segundo lugar, através do comando “vue create” é criado o projeto Vue.js com o nome escolhido pelo utilizador. A terceira tarefa é tratar o ficheiro com as informações de *routing*, que se encontra explicado em maior detalhe na subsecção seguinte (Secção 6.2.1). Em seguida, é aplicado o *script* “parsing” a todos os ficheiros SVG fornecidos como *input* e, desta forma, resta apenas tratar dos ficheiros de configuração, nomeadamente o ficheiro INDEX.JS do *router*, o ficheiro MAIN.JS e o ficheiro APP.VUE.

Para tratar o *routing* é utilizada a biblioteca “vue-router” e, com as informações presentes no ficheiro de *routing* obtido como parâmetro, é feito o mapeamento de todas as *routes* do projeto no ficheiro INDEX.JS presente na pasta *router* criada para este efeito.

Em seguida, apresenta-se o *template* usado em Python que corresponde ao desenvolvimento desse ficheiro.

```
import Vue from "vue";
import Router from "vue-router";

Vue.use(Router);

export default new Router({
    routes: [
        {
        name: "#Nome da Pagina Principal#",
        path: "/",
        component: () => import("@/#Nome do Ficheiro da
                                    Pagina Principal#")
    }
```

Tal como se pode ver, após as configurações iniciais, é criada a informação do *routing* da *homepage* do projeto. Caso existam outras páginas, vai sendo acrescentado o respetivo código ao ficheiro INDEX.JS, tal como se pode ver no excerto seguinte:

```
, {
  name: "#Nome da Pagina#",
  path: "/#Caminho pretendido para a Pagina#",
  component: () => import("@/#Nome do Ficheiro da Pagina#")
}
```

No ficheiro MAIN.JS são importadas as bibliotecas a utilizar, nomeadamente a do *router* e a do *bootstrap-vue*, entre outras definições necessárias para o funcionamento do projeto. Por fim, é gerado o ficheiro APP.VUE com o *template* geral do projeto, as variáveis, importações de ficheiros e definições necessárias para o funcionamento do projeto:

```
<template>
  <div id="app">
    <TheNavbar/>
    <router-view></router-view>
    <TheFooter/>
  </div>
</template>

<script>
import './assets/css.css'
import TheNavbar from './components/Navbar.vue'
import TheFooter from './components/Footer.vue'

```

```
export default {
  name: 'App',
  components: {
    TheNavbar,
    TheFooter
  }
}
</script>
```

O último passo do *script* é, através do gestor de pacotes "npm", instalar as bibliotecas necessárias e correr o projeto. Desta forma, o utilizador não tem que se preocupar com estas instalações, apenas em correr o *script* desenvolvido.

```
    npm install vue bootstrap bootstrap-vue; npm install vue-router; npm run
    serve
```

6.2.1 *Navegação entre páginas*

Como já referido anteriormente, quando se exporta um *mockup* Adobe XD para SVG as informações das hiperligações entre as páginas não ficam descritas no ficheiro resultante. Como tal, para colmatar este problema, é necessário elaborar um novo ficheiro que discrimine estas informações.

Para isso, recorreu-se às mais-valias do XML para a definição deste ficheiro. O ficheiro XML deverá apresentar a seguinte estrutura e respetivos elementos/atributos:

- Elemento *file*

  Esta *tag* deverá ter como atributo o nome ("name") do ficheiro SVG onde se encontra o componente que se pretende hiperligar a outro;

- Elemento *component*

  Dentro deste elemento, que por sua vez já pertence ao elemento *file* deverá estar indicado qual o ID do componente que irá originar a hiperligação e qual a página Vue.js que deverá ser hiperligada (para onde se deverá navegar).

De seguida encontra-se um exemplo do formato a utilizar no ficheiro XML:

```
<?xml version="1.0"?>

<data>
    <file name="User.svg">
```

```
        <component id="Btn-3-id-User" hyperlink="Login"/>
    </file>
    <file name="Login.svg">
        <component id="Btn-3-id-Login" hyperlink="Profile"/>
        <component id="Btn-5-id-Login" hyperlink="Profile"/>
    </file>
</data>
```

Desta forma, o utilizador deverá criar este ficheiro com as informações pretendidas e segundo as normas anteriormente descritas e, tal como mencionado na secção anterior, indicar o nome deste ficheiro como argumento ao correr o *script*. Por sua vez, o *script* irá converter as informações do ficheiro XML para uma *hashtable* onde a chave é o nome do ficheiro SVG que terá os componentes com as hiperligações e o valor será o ID do componente e a página que deverá ser hiperligada ao componente.

Em suma, para o utilizador tirar proveito desta solução de *software*, deverá invocar este *script* no terminal. É de salientar que para a execução deste *script* é necessário ter previamente instalado o "vue-cli" para assim ser possível criar e executar o projeto Vue.js. Este *script* deverá estar na mesma pasta dos ficheiros SVG a traduzir e do ficheiro XML com as informações do *routing*. A invocação deve ser feita da seguinte forma (como já exemplificado no início do capítulo):

```
./script nome_projeto file_homepage array_files file_routing
```

6.3 PARSING SVG - GERAÇÃO DE PÁGINAS

Tendo em conta o algoritmo genérico apresentado na Secção 5.2.4, foi desenvolvido um *script* em Python cujo principal objetivo é traduzir um ficheiro SVG (exportado através do Adobe XD) para um ficheiro Vue.js. O ficheiro Vue.js resultante terá o mesmo nome do ficheiro SVG.

6.3.1 *Bibliotecas utilizadas*

Para o desenvolvimento deste *script* recorreu-se a algumas bibliotecas do Python, nomeadamente:

- XML.ETREE.ELEMENTTREE - para ler o ficheiro SVG;
- RE - para se poder utilizar expressões regulares;
- JSON - para converter uma *string* dicionário para dicionário e para *debug*;
- OS - para utilizar o módulo do sistema operativo;
- SYS - para ler os parâmetros passados como argumento na chamada do *script*.

6.3.2 *Estrutura de Dados*

Na estratégia utilizada para o desenvolvimento deste *script* o primeiro passo foi criar uma estrutura de dados para os componentes e povoá-la com as informações presentes no ficheiro SVG. Assim, foi criado um *array* associativo chamado "components" que tem como chave todos os tipos de elementos passíveis de serem identificados no ficheiro SVG, nomeadamente ***pattern*** (pode conter informações de imagens), ***clip*** (contém a altura e a largura da página), ***image*** (pode conter informações de imagens) e, em seguida, os componentes que podem ser encontrados, isto é, ***Text*** (texto), ***Img*** (imagens), ***Btn*** (botões), ***Navbar***, ***Footer***, ***Form*** (formulários) e ***TextBox*** (caixas para introduzir texto).

Para identificar e recolher as informações dos diferentes componentes no SVG foram desenvolvidas várias funções de *parse*, nomeadamente uma para cada componente. Algumas dessas informações conseguem ser diretamente mapeadas (por exemplo, as cores dos componentes), enquanto que para outras é necessário efetuar algum tratamento dos dados. Em seguida, apresentam-se essas exceções:

- Largura/Altura em percentagem

  Quando a largura ou a altura estão em percentagem é necessário eliminar esse campo para posteriormente poder deduzir-se as respetivas dimensões;

- Texto com largura definida pelo *designer*

  No caso do *designer* ter definido qual é a largura pretendida para o texto (indicando-a no ID) é necessário colocar essa largura no campo respetivo. Caso contrário, deverá ser colocado 0 nesse campo para posteriormente serem deduzidas essas dimensões;

- Informações das imagens

  Por vezes, as informações das imagens, nomeadamente o nome e/ou o caminho para o ficheiro podem estar noutras *tags* presentes no ficheiro SVG. Desta forma, é necessário verificar as *tags defs*, atributo *image* ou *pattern/image* e recolher esses dados;

- Coordenadas iniciais iguais a (0,0)

  Quando o componente começa na posição (0,0) o SVG omite a posição ("translate") do componente. É frequente este processo ocorrer nas *Navbars*. Neste caso, é necessário colocar manualmente as coordenadas iniciais do componente como sendo (0,0);

- Texto e cor do texto nos botões

  Tal como apresentado na Secção 6.1, correspondente às *Tags* SVG, o botão tem aninhada uma *tag* "Text" correspondente às informações do componente Texto dentro do botão. Para simplificar o desenvolvimento da solução, o componente botão criado vai buscar as informações do conteúdo a nível textual e respetiva cor a essa *tag* aninhada e coloca-as na sua estrutura de dados.

No Anexo A encontra-se um exemplo do componente Texto, de uma Imagem e de um Botão nesta estrutura de dados.

Ao longo do processo de *parsing* do ficheiro SVG serão adicionados mais dados a cada elemento da estrutura de dados "components", indispensáveis para a tradução dos mesmos para Vue.js, como é o caso do "x_final" e do "y_final" de cada componente.

6.3.3 *Sistema de grid*

Tal como já mencionado, um dos principais desafios prendia-se em não utilizar as posições absolutas dos componentes. Para tal, recorreu-se ao sistema de *grid* do Bootstrap para posicionar os diferentes componentes. Assim, sabe-se que os componentes terão que estar posicionados em linhas ("*rows*") e, dentro de cada linha, numa das 12 colunas (valor máximo).

Antes de se começar a definir qual a linha e coluna de cada componente, é necessário calcular quais as coordenadas finais de cada um deles (neste momento, apenas se tem o conhecimento das coordenadas iniciais, da altura e da largura). Desta forma, foi adicionado um novo atributo a cada um dos componentes com este cálculo, nomeadamente somar a abcissa com a largura do componente para se obter a abcissa final e somar a ordenada com a altura do componente para se obter a ordenada final.

Para identificar a **linha** de cada componente, o primeiro passo centrou-se em dividir todos os componentes pelas linhas. Para tal, foi criada uma nova estrutura de dados intitulada "components_coordinates" com os componentes ordenados tendo em conta a sua ordenada inicial ("y") do sistema de coordenadas cartesiano. Uma vez tendo os componentes ordenados pela sua respetiva ordenada, a missão prende-se em perceber se os componentes se encontram situados na mesma linha e, para posteriormente facilitar o processo, atribuir um número (correspondente à linha) a cada um dos componentes. Numa nova variável intitulada "sections" foi criada uma *hashtable* onde a chave é o número da linha e, observando os valores das ordenadas iniciais e finais de cada um dos componentes, torna-se possível perceber em que linha se situarão cada um deles.

Em seguida, encontra-se um possível exemplo do valor desta variável:

```
{
  "1": [                    %linha 1
    [
      "Navbar",
      [
        {"x": 0},
        {"y": 0},
        {"width": 1920},
        {"height": 220},
        {"x_final": 1920},
        {"y_final": 220}
      ]
    ],
    ...],
  "2": [                    %linha 2
```

```
    [...]
  ],
  "3": [                    %linha 3
    [...]
  ],
  ...
}
```

Após ter os componentes devidamente organizados em linhas, o objetivo passa por, dentro de cada linha, organizar os componentes em **colunas**. Para alcançar este objetivo, o primeiro passo foi, dentro de cada linha, ordenar os componentes pelo valor da sua abcissa ("x") inicial (sistema de coordenadas cartesiano). Depois disso, através da abcissa inicial e final de cada componente foi possível perceber se estes se encontram na mesma coluna ou não e atribuir um número de coluna a cada um deles. Desta forma, obtém-se uma estrutura de dados organizada pelo número da linha como chave, e, dentro de cada linha, com o número da coluna também como chave. Por fim, encontram-se as informações de cada um dos componentes. Em seguida, apresenta-se um exemplo desta estrutura:

```
{
  "1": {                    %linha 1
    "1": [                  %linha 1, coluna 1
      [...],
    ],
    "2": [                  %linha 1, coluna 2
      [...],
    ]
  },
  "2": {                    %linha 2
   ...
  }
}
```

6.3.4 *Esqueleto Vue.js*

Uma vez obtidas as informações relativas à linha e à coluna de cada um dos componentes, é necessário aproximar ainda mais a posição de cada componente à realidade do sistema de *grid* do Bootstrap. Para isso, é necessário saber o número de colunas que cada um dos componentes ocupa. Neste momento, a atribuição do número da coluna (presente como chave na estrutura de dados) a cada componente está somente relacionada com o facto do componente estar ou não na mesma coluna de outros componentes.

Como o Bootstrap é organizado através de um sistema de *grid* que divide o ecrã em 12 colunas, é possível calcular o tamanho em píxeis de cada uma das colunas dividindo a largura total do *mockup* por 12 (a largura da página encontra-se na *tag* "clip", armazenada numa variável no início do processo de leitura do SVG).

Desta forma, como é conhecida a largura de cada um dos componentes, é possível calcular também o número de colunas que ele ocupará. Em cada um dos componentes foi criado um novo atributo chamado "col" onde o valor é o número de colunas que ele ocupa.

Com estas informações, obtém-se uma nova estrutura de dados onde se encontram as linhas, colunas e o número de colunas que cada um dos componentes ocupa. Como exemplo ilustrativo, em seguida apresenta-se um excerto da estrutura de dados completa, onde se consegue perceber que na linha 2, coluna 1, está uma imagem com as informações que se encontram descritas e que ocupa 12 colunas (último atributo).

```
{
  ...,
  "2": {                    %linha 2
    "1": [                  %linha 2, coluna 1
      [
        "Img-2",
        [
          {"x": 0},
          {"y": 251},
          {"width": 1920},
          {"height": 231},
          {"x_final": 1920},
          {"y_final": 482},
          {"fill": "url(#pattern)"},
          {"stroke": null},
          {"col": 12}
        ]
      ]
    ]
  },
  ...
}
```

No entanto, tal como já referido, há um componente que tem que ser tratado de forma diferente - o componente Texto, uma vez que, caso o *designer* não indique a respetiva largura através do ID do componente, por omissão a largura será 0 (no ficheiro SVG não é indicado esse atributo), pelo que através do cálculo referido o valor do número de colunas obtido será igual a 0.

Para contornar este pormenor, foi feito o cálculo do número de componentes presente na linha e seguiu-se o seguinte algoritmo:

```
Sendo n o n.º de componentes na linha:
    Se n <= 6 -> 12/n
    Senão -> 1
```

O raciocínio foi fazer com que o componente texto tenha um "espaço" equivalente/proporcional ao número de componentes e tamanho da página.

Em suma, neste ponto tem-se uma nova estrutura de dados onde se pode encontrar as linhas, colunas e o número de colunas que cada um dos componentes ocupa. No entanto, não se sabe qual o espaço entre os componentes em cada uma das linhas. Por exemplo, no caso de uma interface gráfica que contenha uma imagem no início do lado direito da página a ocupar 2 colunas e uma imagem no final da página (no lado esquerdo) a ocupar 2 colunas, com esta implementação iria obter-se as duas imagens, uma seguida à outra, do lado direito da página (ver Figura 23).

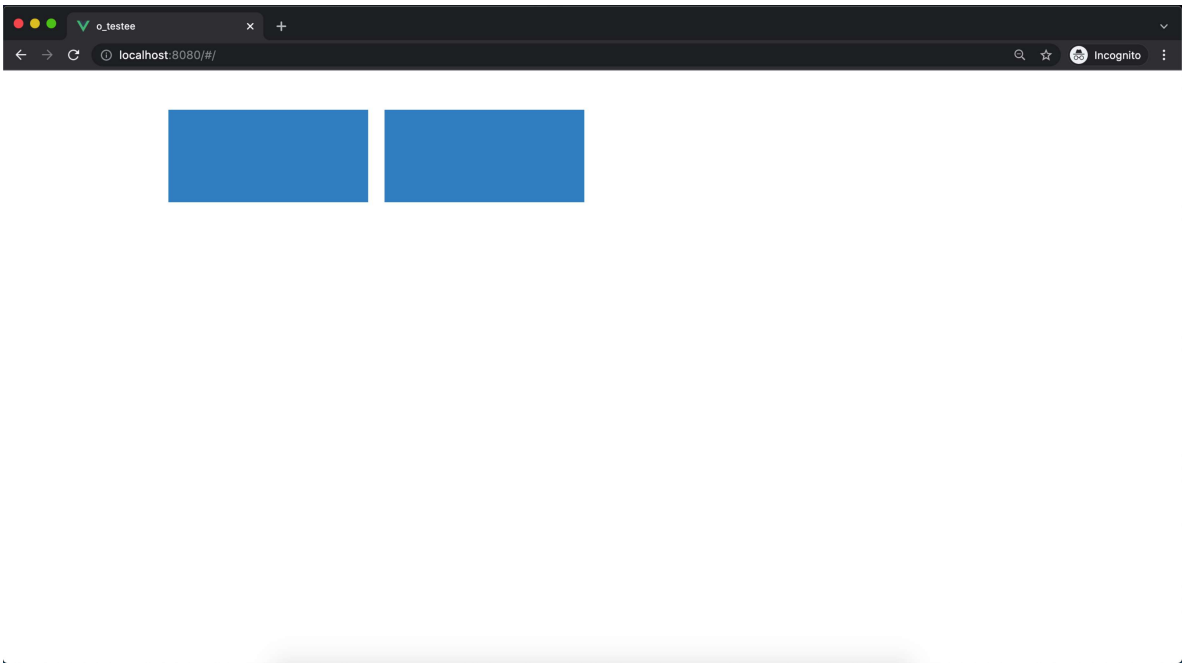


Figura 23: Duas imagens lado a lado, sem espaço entre elas.

A solução foi criar um novo componente intitulado "Empty" que através da informação do "x_final" do componente anterior e do "x" do componente atual calcula qual a largura do espaço em branco entre os componentes (através da subtração destes valores).

Em seguida, calcula-se o número de colunas a que esta largura corresponde (dividindo esse valor por 12). No caso de ser maior do que 0, é adicionado um novo componente à estrutura de dados entre os dois componentes com a seguinte descrição (onde a anotação ### indica valores a preencher pelo algoritmo):

```
[
    "Empty",
    [
     {"x": ###},
     {"y": ###},
     {"width": ###},
     {"x_final": ###},
     {"y_final": ###},
     {"fill": "null"},
     {"stroke": "null"},
     {"col": ###}
    ]
]
```

Desta forma, obteria-se o resultado pretendido (ver Figura 24). É de notar que o mesmo acontece, quer seja entre componentes, quer seja antes de algum componente que não se encontre posicionado logo no início da linha da página.

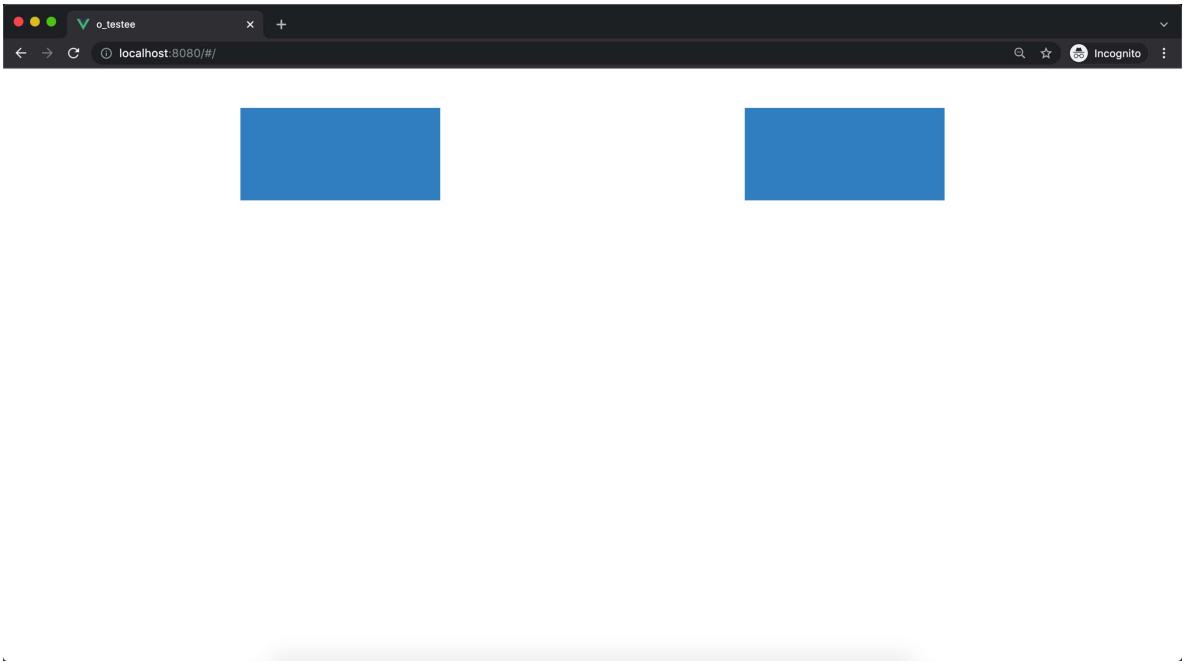


Figura 24: Duas imagens lado a lado, com espaço entre elas.

Em suma, tem-se assim todas as informações necessárias para definir as posições relativas e posicionar os componentes.

6.3.5 *Gerar Componentes*

O último passo consiste em percorrer a estrutura de dados anteriormente criada, e devidamente povoada, e gerar para o ficheiro Vue.js essas informações. Por omissão, assume-se que cada linha terá um espaçamento horizontal de 40 píxeis para a próxima linha (como já referido). Assim, a cada nova entrada na estrutura de dados é gerada a seguinte linha de código:

```
1    <div class="row" style="padding-bottom: 40px;">
```

Para cada coluna, é gerada a seguinte linha de código (onde a *string* "col" corresponde ao número de colunas que o componente ocupa):

```
1    <div class="col-" + str(col) + "">
```

Por fim, são geradas as informações do respetivo componente. Se o componente tiver *routing* associado (isto é, uma hiperligação para outra página) é também gerada essa informação através da *tag* "router-link" do Vue.js (antes de ser gerado o componente):

```
1    <router-link :to="{name: '" +  routing[new_id] + "'}">
```

O *"routing"* é um *hashmap* onde a chave é o ID do componente a hiperligar e o valor é o nome da página do projeto que deve ser aberto ao clicar no componente.

Como forma de apresentar de uma forma mais elucidativa como é gerado cada componente, em seguida apresenta-se como é gerado o componente botão:

```
1    <button class="btn btn-outline-secondary " + new_id +
     "type="button"> + text + "</button>"
```

Tal como se pode ver no excerto de código, existem 2 variáveis que são introduzidas no meio do código, nomeadamente "text" e "new_id". A variável "text" representa o texto que deve constar no conteúdo do botão. A variável "new_id" foi adicionada no atributo *class* de modo a que o componente Vue.js tenha um identificador. O objetivo deste identificador é permitir a alteração das propriedades do componente através de código CSS.

No que diz respeito ao CSS dos componentes, quando cada um dos respetivos componentes é gerado para o ficheiro Vue.js, em simultâneo vai sendo adicionado o CSS de cada um desses componentes a uma variável intitulada "css_css". Após terem sido percorridos todos os componentes, é criado um ficheiro "css.css" dentro da pasta "assets" e o conteúdo de "css_css" é colocado neste ficheiro. Em seguida, encontra-se o exemplo do código que é adicionado à variável "css_css" para o caso do componente botão:

```
.#id componente# {
    color: ### !important;
    background-color: ### !important;
    border-color: ### !important;
}
```

Tal como se pode ver, trata-se de código CSS onde é editada a cor do texto do componente botão, a cor do fundo e a cor da borda. É de salientar que como num projeto podem existir vários ficheiros SVG, podem existir componentes com o mesmo ID (uma vez que quando o Adobe XD exporta os ficheiros reinicia a contagem dos IDs a cada nova página do protótipo). Assim, se fosse utilizado o ID diretamente fornecido pelo SVG podia acontecer de serem alteradas as propriedades CSS de componentes que não são os pretendidos. Para contornar este facto, é atualizado o ID de cada componente e adicionado ao ID uma *string* com o nome da página SVG.

Já no que diz respeito a gerar o componente "Form", como são vários os seus subcomponentes e o objetivo é aplicar o componente Bootstrap correspondente, a abordagem tem que ser diferente. A partir do momento em que aparece um componente com a *tag* "Form", são tratados de forma diferente todos os componentes que aparecerem daí em diante, até aparecer um componente cujo "y_final" ultrapasse o limite do "y" estipulado pelo retângulo do "Form". Não é necessário ter em consideração o valor da abcissa, uma vez que se está a tratar a colunas de uma linha, logo, o limite vertical já está a ser tido em consideração. Quando aparece o primeiro componente fora do formulário, é gerada a *tag* de fim de "Form" ("</form>") e, tanto esse, como os próximos componentes presentes na estrutura de dados, já não serão tratados como estando presentes num formulário.

É de notar que, na versão atual, quando se começa a gerar os componentes dentro do formulário são feitas algumas assunções. Por exemplo, quando aparece um componente "Texto" deduz-se que corresponde a uma *label* e que terá uma "Text Box" correspondente à sua frente. O componente "Text Box" per si presente na estrutura de dados é ignorado (porque já ter sido deduzido e colocado em frente ao texto).

Como já explicado, a *Navbar* e o *Footer* são componentes tratados de forma diferente, isto é, estão preparados para serem reutilizados e, desta forma, são criados em ficheiros independentes.

Quando aparece um componente identificado como sendo uma *Navbar* ou *Footer* coloca-se o valor "True" na variável "print_navbar" ou "print_footer", respetivamente, e a partir daí os componentes são gerados para o Ficheiro "Navbar.vue"/"Footer.vue", criado dentro da pasta "components". Antes de gerar os componentes dentro da *Navbar* são geradas as *tags* iniciais de uma *Navbar*. O mesmo acontecerá para o caso do *Footer*. Tal como no caso do "Form", assim que for atingido o limite do retângulo em torno da *Navbar*/*Footer* este processo é terminado, com as respetivas *tags* geradas no ficheiro e a variável "print_navbar"/"print_footer" volta a ter o valor "False". Nestes dois componentes optou-se por gerar o respetivo CSS no próprio ficheiro Vue.js, devido ao facto de terem os seus próprios ficheiros de forma exclusiva e, portanto, é também feita esta geração antes de passar à análise do próximo componente na estrutura de dados (se existir).

Todas as imagens encontradas no protótipo foram sendo adicionadas a um *array* “images” e, no final, são copiadas para a respetiva diretoria do projeto.

6.3.6 *Adicionar novo componente*

Tal como já explicado, a solução de *software* proposta suporta apenas alguns componentes (Texto, Imagem, Botão, *Navbar*, *Footer*, Formulário e Caixa de Texto). Como mencionado, do ponto de vista do *designer* para a ferramenta suportar novos componentes basta ser-lhe passada a informação do ID a utilizar no Adobe XD para incorporar o novo componente.

Do ponto de vista de implementação técnica, é necessário cumprir os seguintes passos:

1. Na estrutura de dados “components” criar uma nova chave com o nome do novo componente;
2. Criar uma função de *parse* dos parâmetros do SVG do novo componente para os adicionar à estrutura de dados;
3. Adicionar a deteção do novo componente, na secção responsável por percorrer todo o ficheiro SVG, para poder ser invocada a correspondente função de *parse*, quando aparecer um componente com o ID igual ao ID do novo componente;
4. Adicionar uma nova condição, na secção responsável por gerar o código Vue.js do componente para o ficheiro de texto do projeto, com o respetivo código Vue.js e lógicas associadas.

6.4 SUMÁRIO

Neste capítulo foi descrita a implementação da ferramenta desenvolvida nesta dissertação. Para isso, apresenta-se como se encontram organizados os ficheiros SVG gerados pelo Adobe XD e apresentam-se os dois *scripts* desenvolvidos em Python, responsáveis pela configuração do projeto Vue.js e pela geração de páginas. No próximo capítulo será ilustrada a utilização da solução de *software* desenvolvida.

7

AVALIAÇÃO

O objetivo do presente capítulo é apresentar os passos da solução proposta, desde o momento em que se recebe um *mockup* desenhado em Adobe XD (através de ficheiros SVG), até se obter a respetiva implementação *web*. Para tal, apresenta-se um exemplo prático de uma possível aplicação *web*.

7.1 O PROTÓTIPO

O exemplo que será utilizado para ilustrar a abordagem consiste no *website* de um Clube de Ténis. Foram desenhados dois *mockups* de duas das páginas do protótipo do *website*. O *mockup*, desenhado em Adobe XD, da Página Inicial encontra-se na Figura 25 e o *mockup* da Página de Reservas (acessível através do último botão da *navbar* da Página Inicial) na Figura 26.

A Página Inicial é composta por uma *navbar*, que contém uma imagem e três botões, com hiperligações para, respetivamente, a Página de Notícias, a Página sobre as Instalações e a Página das Reservas. Em seguida, contém uma imagem que ocupa toda a largura da página, seguida de uma mensagem de boas-vindas. Posteriormente, contém três imagens e três botões, cada um com uma hiperligação para a Página da Competição, das Aulas Individuais e das Crianças e Jovens, respetivamente. Por fim, tem um *footer* simples, com fundo azul e a mensagem relativa aos direitos.

A Página das Reservas contém a mesma *navbar* da Página Inicial e, de seguida, um título (texto). Posteriormente, tem uma secção com um formulário à esquerda e uma imagem à direita. O formulário é composto por quatro campos, "Nome Completo", "Telemóvel", "Email" e "Data", cada um deles com uma caixa para inserir texto. No final, tem um botão para submissão do formulário. Em seguida, tem quatro colunas com texto, nomeadamente os dados do Clube de Ténis (morada, e-mail, número de telemóvel e nome da rede social). Termina com o *footer*, tal como o da Página Inicial.

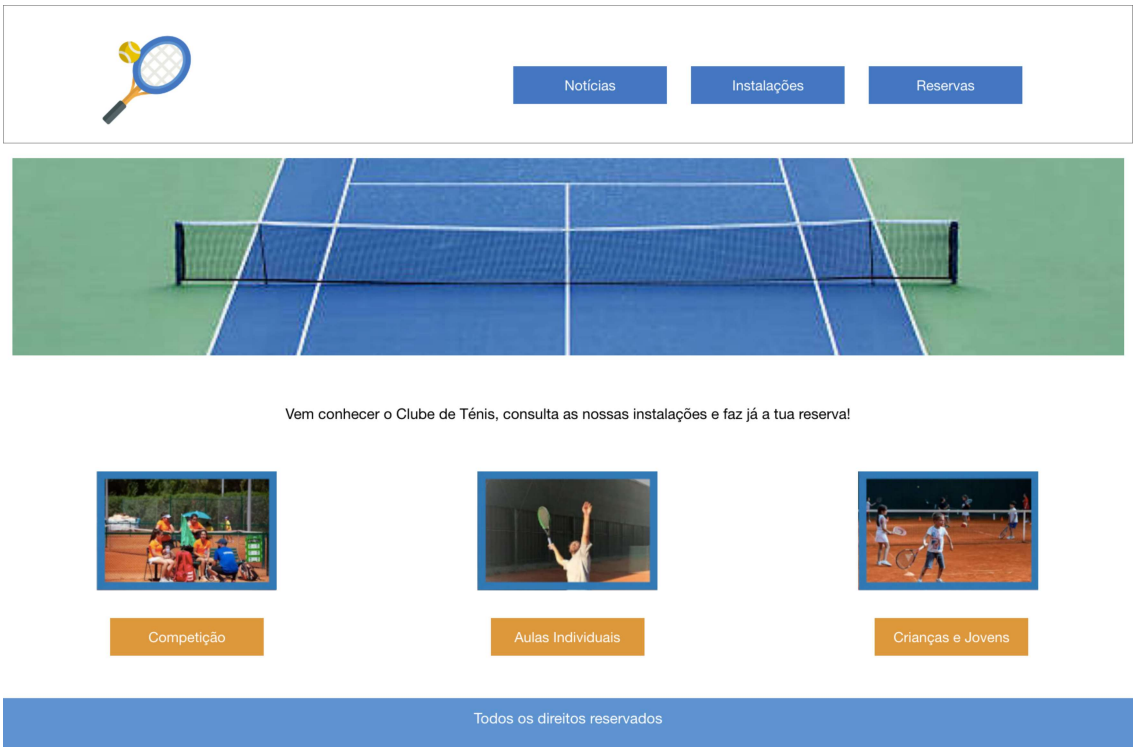


Figura 25: Protótipo da Página Inicial do Clube de Ténis desenvolvida em Adobe XD.

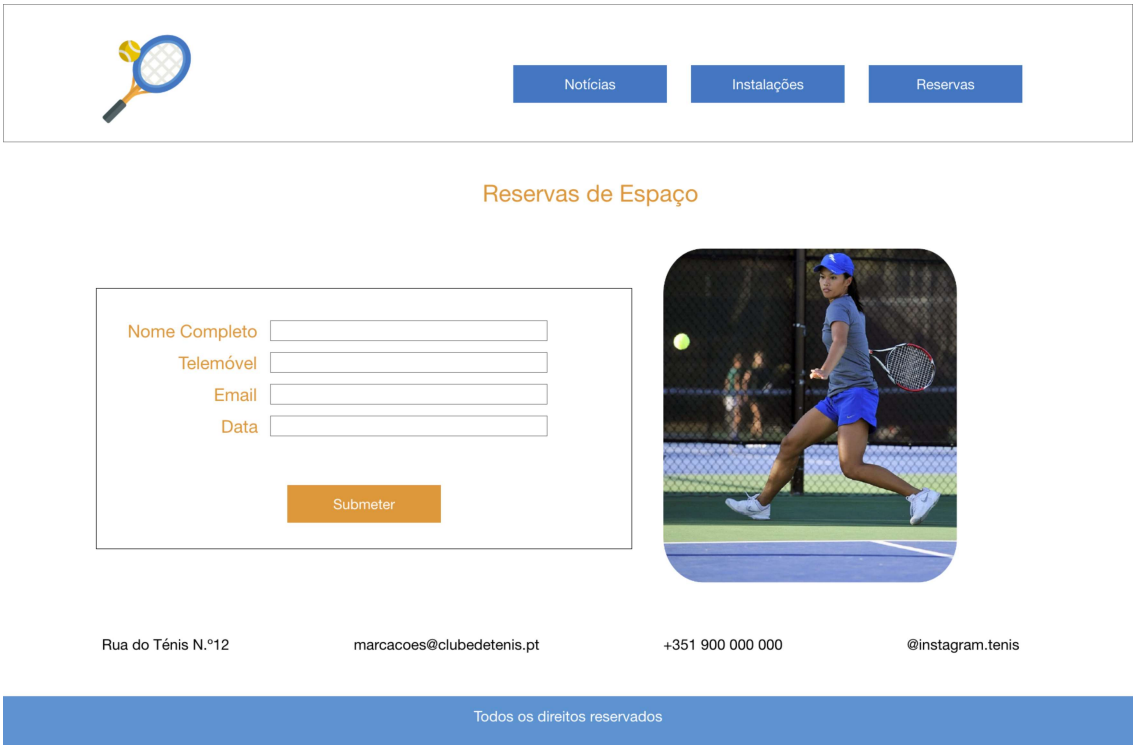


Figura 26: Protótipo da Página de Reservas do Clube de Ténis desenvolvida em Adobe XD.

7.2 CONFIGURAÇÃO DO PROTÓTIPO

Como mencionado na Secção 5.3 e ilustrado na Figura 22, o *designer* tem que desenhar um retângulo em redor da *navbar*, formulário e do *footer*, sendo que isso serve apenas para delimitar os *containers* (não faz parte do *design*). É de notar que, no caso do *footer*, como o *container* tem uma cor de fundo, acaba por fornecer também essa informação adicional acerca do *design*.

É também de salientar que o *designer* tem que colocar no campo do ID do componente, no Adobe XD, o respetivo ID definido na Secção 5.2.1.

Nos Anexos B e C encontra-se o código SVG gerado pelo Adobe XD.

Em relação à navegação entre páginas, neste caso tem que se hiperligar cada um dos botões com a respetiva página. Este processo será ilustrado com a ligação do botão de "Reservas" da *navbar* com a Página de Reservas. O ficheiro XML, onde é definida a navegação e que irá ser passado como argumento ao *script*, deverá conter essa informação. Uma vez que a *navbar* será comum a todas as páginas do projeto e o respetivo componente Vue.js será partilhado entre elas, basta definir a informação de navegação para uma das páginas. Tendo em conta a forma como a solução foi desenvolvida, basta definir a navegação para a última página passada como argumento à *script*. No entanto, se o utilizador colocar a informação em todas as páginas, o programa funcionará corretamente (vai ignorar todas, exceto a última).

O ID do botão das Reservas da Página Inicial pode ser encontrado no Anexo B, linha 63, "Btn-11" e o da Página de Reservas no Anexo C, linha 32, "Btn-5". No ficheiro com as informações de *routing* deve ser acrescentado ao ID do componente "-id-NomeDaPagina" (este será o ID do componente no ficheiro de CSS, de forma a evitar repetições de IDs entre componentes de diferentes páginas), tal como referido na Secção 6.3. O ficheiro XML desenvolvido encontra-se em seguida:

```
<?xml version="1.0"?>

<data>
    <file name="Reservas.svg">
        <component id="Btn-5-id-Reservas" hyperlink="Reservas"/>
    </file>
</data>
```

Desta forma, têm-se os três ficheiros necessários para a geração do código do projeto, nomeadamente o ficheiro SVG da Página Inicial e da Página de Reservas e o ficheiro XML com as informações de *routing*.

7.3 GERAÇÃO DA INTERFACE

Uma vez tendo os protótipos desenhados exportados para SVG e o ficheiro XML de navegação, o seguinte *script* deve ser executado através do terminal.

```
./script projeto Inicio.svg "['Reservas.svg']" navegacao.xml
```

Tal como já explicado, o *script* começa por colocar todos os componentes identificados no SVG num *hashmap*. Para os exemplos apresentados, os *hashmaps* gerados pelo programa encontram-se no Anexo I (Página Inicial) e no Anexo J (Página de Reservas).

Em seguida, tal como explicado na Secção 6.3, começa por dividir as páginas em linhas e, posteriormente, em colunas dentro de cada uma das linhas. Estas informações ficam expostas num *hashmap*, presente no Anexo K para o caso da Página Inicial e presente no Anexo L para o caso da Página das Reservas.

Para o caso da Página Inicial, tal como se pode deduzir pela Figura 25, a interface gráfica pode ser dividida em seis diferentes secções, nomeadamente a *navbar*, a imagem do campo de ténis, o texto, as imagens, os botões e, por fim, o *footer*. Como seria de esperar, na respetiva *hashmap* (Anexo K), com as secções e com as colunas numeradas, pode ver-se que o número de secções vai de 1 até 6. O mesmo acontece com as colunas. A *navbar* é um caso especial em que é considerado que todos os componentes estão na mesma coluna. Isto é o suficiente para se conseguir aplicar o componente da *navbar*, até porque, mesmo estando todos na coluna 1, tem-se a informação de que a *navbar* é constituída por uma imagem ("Img-4", linha 19) que ocupa 1 coluna (linha 27), em seguida, por um espaço em branco ("Empty", linha 33) que ocupa 3 colunas e, por fim, por 3 botões, cada um deles a ocupar 2 colunas.

A secção 2 (linha 89) é composta apenas por 1 coluna com a "Img-5" que ocupa as 12 colunas. A secção 3 (linha 107) também é composta por 1 coluna, que começa por um espaço em branco ("Empty", linha 110) que ocupa 3 colunas e, por fim, o "Text-2" (linha 123). A secção 4 (linha 138) é composta por 3 colunas, cada uma delas com um espaço em branco e uma imagem e a secção 5 (linha 237) também é composta por 3 colunas, cada uma delas também com um espaço em branco e neste caso um botão. Termina na secção 6 com o *footer* que, a par da *navbar*, tem também uma única coluna com o próprio componente do *footer* e com o texto.

Consegue-se deduzir estas informações através da análise das imagens dos protótipos e é essa mesma informação que se encontra nas estruturas de dados.

Com os componentes divididos por secções e colunas, resta gerar para os ficheiros Vue.js o código de cada componente. O resultado *web* obtido para a Página Inicial e para a Página de Reservas encontram-se na Figura 27 e Figura 28, respetivamente.

7.4 Discussão dos Resultados

Como se pode ver através das secções anteriores, o *mockup* da Figura 25 originou a página *web* da Figura 27 e o *mockup* presente na Figura 26 gerou a página *web* da Figura 28, o resultado obtido é bastante semelhante ao pretendido e os ficheiros Vue.js correspondentes, Anexo F e Anexo G, respetivamente, são bastante simples. Nestes, facilmente seria possível alterar o número de colunas que cada componente ocupa, alterar o texto, as cores (por se tratarem de propriedades, neste caso, no Anexo H), as secções, as linhas, adicionar novos componentes, entre outros. Estes dois aspetos eram os mais importantes de obter e foram conseguidos. Neste momento, obtêm-se ficheiros Vue.js próximos da realidade pretendida e preparados para servir como base para o desenvolvimento de uma aplicação *web* com qualidade para os utilizadores finais.

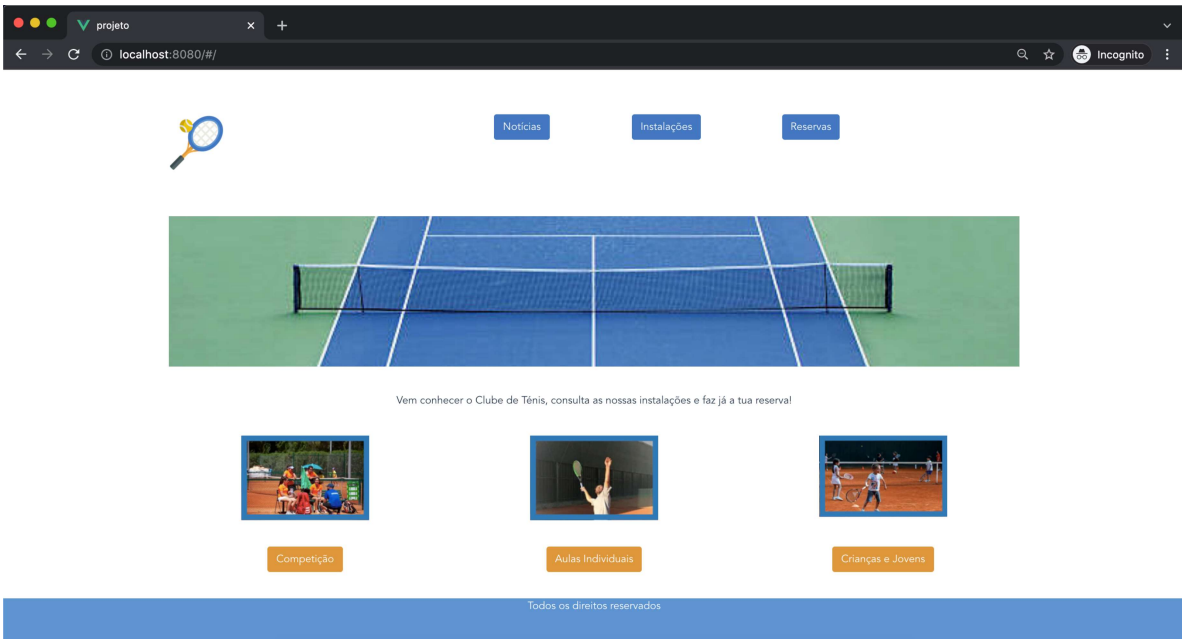


Figura 27: Resultado *Web* da Página Inicial do Clube de Ténis.

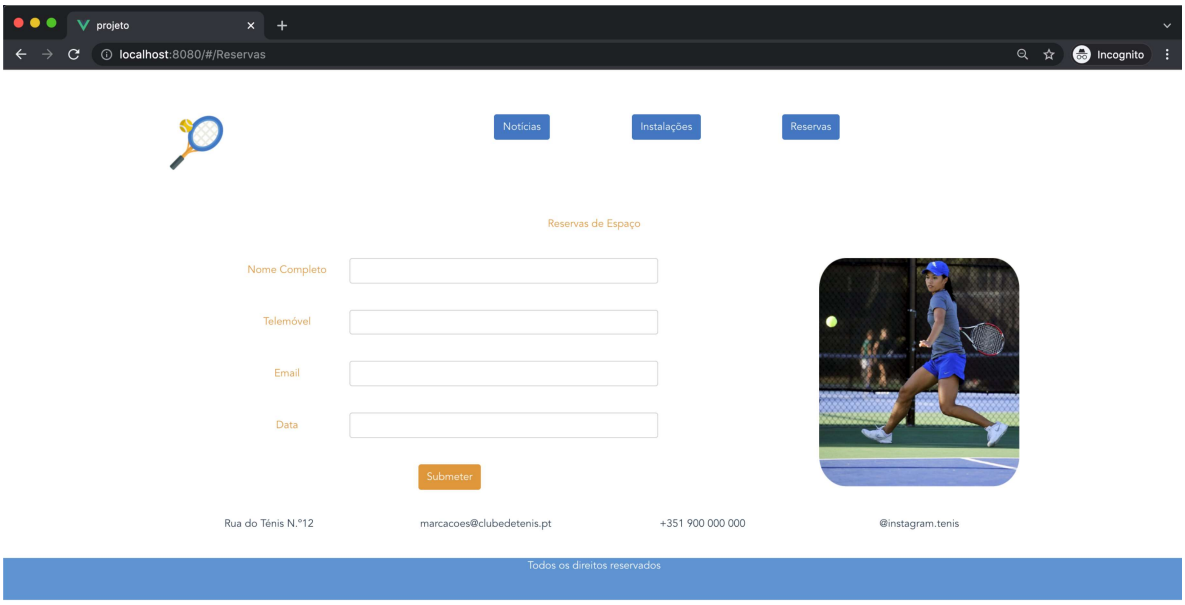


Figura 28: Resultado *Web* da Página de Reservas do Clube de Ténis.

Na Secção 5.3 encontram-se os requisitos esperados de uma ferramenta que suporte a abordagem e nesta fase observa-se que todos eles foram cumpridos. Ou seja, obtém-se um projeto Vue.js com todas as configurações preparadas e o utilizador tem a oportunidade de escolher qual será a página inicial do mesmo e qual é o fluxo de navegação entre todas as páginas. O programa consegue interpretar que a *navbar* e o *footer* são componentes compostos e também os gera para ficheiros à parte (Anexos D e E, respetivamente). No caso do formulário, também componente composto, aplica o componente *form* do Bootstrap/Vue.js como esperado. O

programa adiciona e incorpora também um ficheiro CSS (Anexo H) com todas as informações das cores dos componentes. No resultado final, tal como se pode ver nos anexos, não são utilizadas coordenadas absolutas, apenas posições relativas recorrendo ao sistema de *grid* do Bootstrap.

No entanto, foi necessário fazer algumas simplificações. Para começar, a flexibilidade do *design* dos componentes pode ser melhorada, uma vez que o programa não distingue se um determinado componente está, por exemplo, num formato triangular ou circular, assumindo sempre que se trata de um componente *default*, retirado diretamente do Vue.js/Bootstrap. O mesmo acontece ao nível das próprias imagens que são incorporadas no *design*. O ficheiro das imagens deve ter as dimensões corretas, uma vez que se estiverem demasiado grandes ou demasiado pequenas não ficarão com a proporção desejada no resultado final. Trata-se de um aspeto que tem que ser melhorado posteriormente através de alterações no ficheiro de CSS.

Foram assumidos valores por omissão como o espaçamento entre linhas ser sempre de 40 píxeis e em relação ao formulário, assume-se que o "Texto" vai ocupar sempre 3 colunas e a *"Text box"* vai ocupar 9 colunas, o que pode não corresponder à realidade do pretendido. Aqui a flexibilidade poderia também ser melhorada, visto o programa estar à espera que seja um formulário simples, apenas com o texto, a caixa para inserir texto e botões, não permitindo que existam outros tipos de componentes, como imagens, no formulário (seriam ignoradas).

O tamanho da letra do texto é também um valor por omissão. Como se pode ver na protótipo da Figura 26, o título "Reservas de Espaço" tem um tamanho de letra superior ao texto do formulário. E como se pode ver no respetivo código SVG, presente no Anexo J, tem-se também acesso a essa informação e, de facto, por exemplo, o texto "Nome Completo" tem um tamanho da letra igual a 30 (linha 78) e o título "Reservas" tem um tamanho da letra igual a 40 (linha 103). Para esta alteração ser refletida no código Vue.js, basta editar o atributo "font-size" presente no CSS do componente. Aqui o desafio seria fazer a conversão destas unidades de medida, tamanho 30 e tamanho 40, para a unidade de medida do CSS, que para o resultado ser pretendido deveria ser cerca de 18px e 25px, respetivamente. Uma vez desenvolvida a fórmula a aplicar para obter estas conversões, bastava adicionar este campo ao CSS, tal como se coloca para alterar a cor do texto.

Para melhorar os valores por omissão, o que se pode fazer é o mesmo que foi feito para o caso da largura do texto. Como o ficheiro SVG não indica a largura do texto, o *designer* pode fornecer a largura pretendida através do próprio ID do componente, inserido no Adobe XD (tal como mencionado na Secção 5.3). Caso não forneça, tem que existir sempre um valor por omissão, que neste caso é obtido através do cálculo do número de componentes na linha e dividir equitativamente a largura (explicado na Secção 6.3.4). Outra forma de melhorar os valores por omissão também poderia ser através de um ficheiro de configuração com estes valores. No entanto, tudo pode ser sempre atualizado no final, diretamente no CSS e Vue.js, de uma forma bastante simples. O resultado pretendido é cumprido - obter uma primeira versão da implementação, sendo estes valores por omissão detalhes de refinação do *design*. O objetivo é ajudar o programador/*designer*, não substituí-lo por completo.

Uma outra possível melhoria diz respeito à recursividade dentro de uma coluna, uma vez que uma coluna também se pode dividir em linhas e em colunas, e assim sucessivamente. Para melhor ilustrar esta questão, apresenta-se, na Figura 29, um protótipo de uma Página de Informações que, depois da *navbar* e do texto "O

nosso Clube de Ténis", contém duas colunas, sendo que uma delas se subdivide em duas linhas, cada uma com uma imagem.

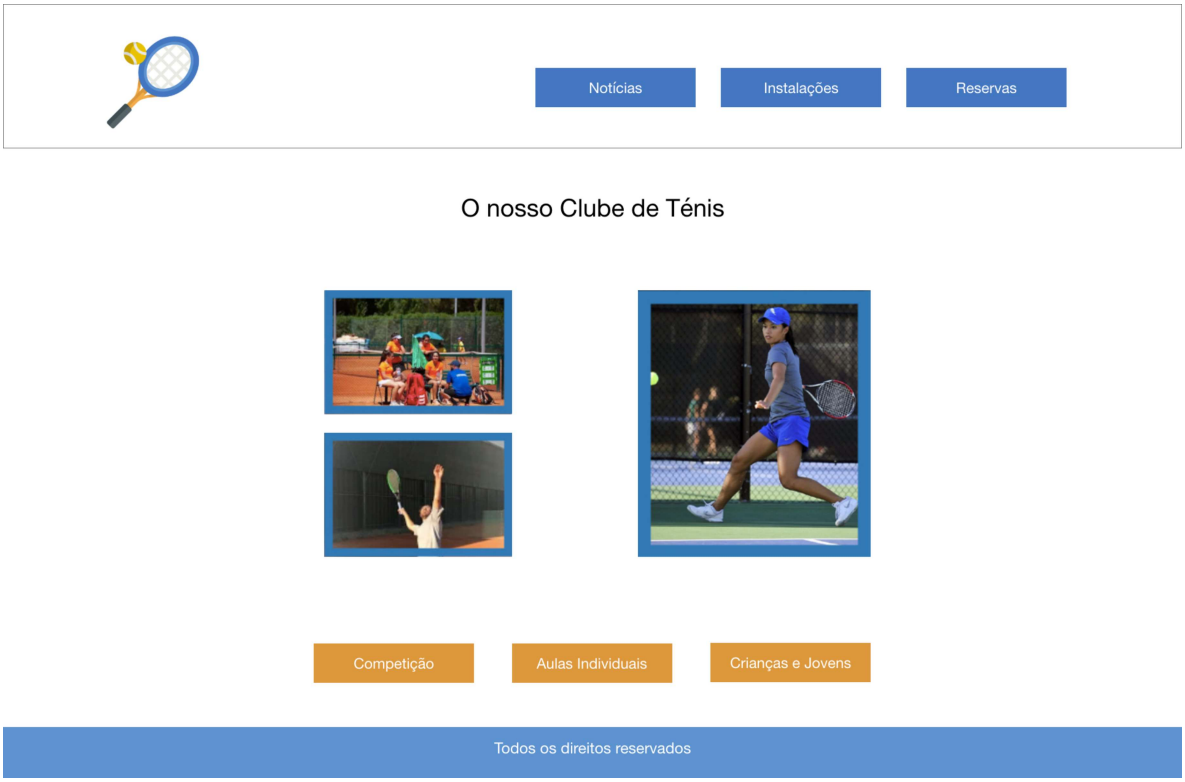


Figura 29: Protótipo da Página de Informações do Clube de Ténis desenvolvida em Adobe XD.

Depois de processado o ficheiro SVG, a página *web* obtida encontra-se na Figura 30. Tal como se pode ver, o resultado obtido não é o esperado. Analisando em pormenor o código Vue.js que é gerado (apresentado em seguida), vê-se que dentro dessa secção o programa interpreta que existe um espaço em branco que ocupa 3 colunas (o que está correto), mas depois percebe que existe uma imagem a ocupar 2 colunas e outra imagem em seguida também a ocupar 2 colunas. Está correto que cada uma dessas imagens deve ocupar 2 colunas, mas o programa acaba por colocar as imagens lado a lado e não uma em cima da outra.

```
<div class="row" style="padding-bottom: 40px;">
    <div class="col-3">
    </div>
    <div class="col-2">
        <img src="./assets/img1.jpeg">
    </div>
    <div class="col-2">
        <img src="./assets/img2.jpeg">
    </div>
    <div class="col-1">
    </div>
    <div class="col-3">
        <img src="./assets/jogar.jpeg">
```

```
    </div>
</div>
```

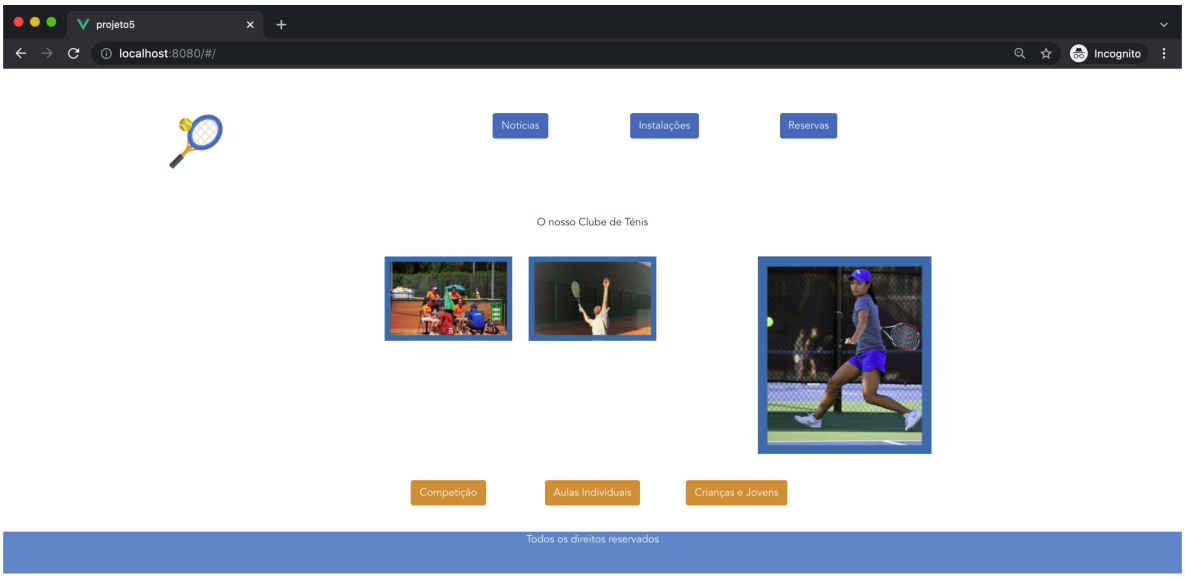


Figura 30: Resultado *Web* da Página de Informações.

Neste momento, para o resultado visual ser o esperado, é suficiente mudar manualmente o elemento de sítio no HTML. No futuro, para evitar este problema, o que teria que acontecer é que dentro de cada célula da grelha deveria ser novamente aplicado o algoritmo de cálculo de *layout*. Isso permitiria concluir que as dimensões do eixo do "x" das duas imagens são as mesmas e os valores dos "y" não se sobrepõe, logo, que dentro da coluna existem duas secções, que devem ser dispostas verticalmente. O resultado obtido seria o que se apresenta na Figura 31, bastante semelhante ao pretendido (Figura 29) e o código gerado seria o seguinte:

```
<div class="row" style="padding-bottom: 40px;">
    <div class="col-3">
    </div>
    <div class="col-2">
        <div class="row" style="padding-bottom: 40px;">
            <div class="col-12">
                <img src="./assets/img1.jpeg">
            </div>
        </div>
        <div class="row">
            <div class="col-12">
                <img src="./assets/img2.jpeg">
            </div>
        </div>
    </div>
```

```
    <div class="col-1">
    </div>
    <div class="col-3">
        <img src="./assets/jogar.jpeg">
    </div>
</div>
```

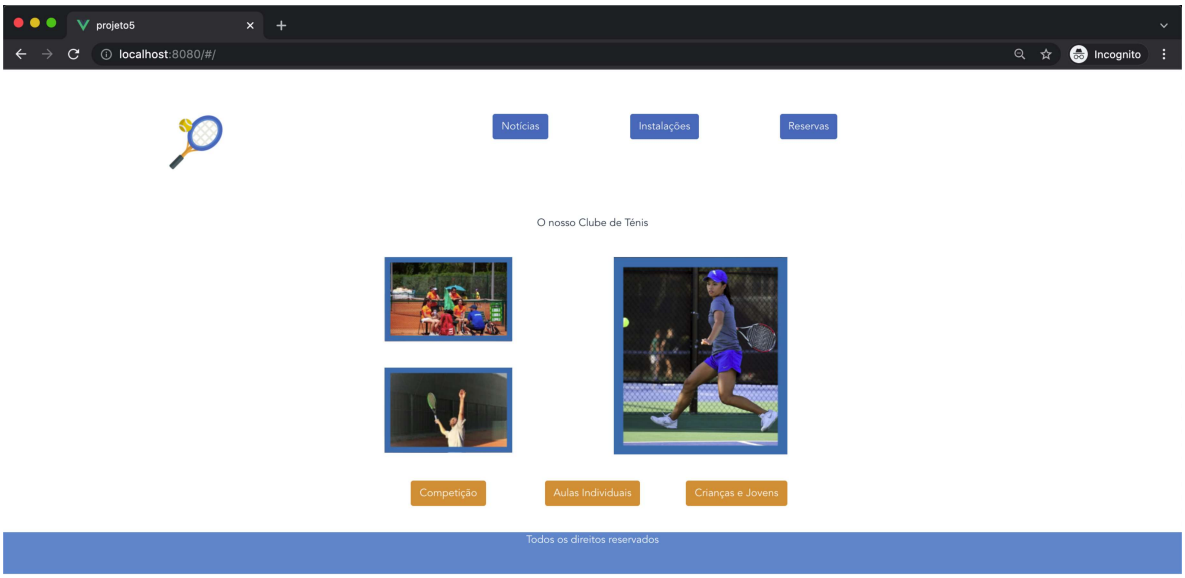


Figura 31: Resultado *Web* da Página de Informações corrigida.

Esta solução não está ainda implementada, tendo ficado para trabalho futuro.

Existe ainda uma grande limitação que está intrínseca ao formato, ao próprio SVG. Sendo o Adobe XD uma *framework* de desenho, falta a estruturação do protótipo, isto é, para a mesma interface gráfica existem várias formas de representação do SVG e da respetiva estrutura de dados (o SVG tem como objetivo representar um desenho, não nasceu propriamente para ser interpretado e processado). Ao testar vários exemplos diferentes de SVG na solução implementada, muitas vezes foi necessário adaptar o código porque a ordem das *tags* mudava no SVG.

Assim, o tratamento do SVG será sempre um processo em curso porque ao testar cada vez mais exemplos, vai se obtendo outras formas de representação e pode sempre melhorar-se os *scripts* desenvolvidos.

8

CONCLUSÃO E TRABALHO FUTURO

O principal objetivo da presente dissertação é dar um contributo para a automatização do desenvolvimento de aplicações *web*, a partir de protótipos das suas interfaces gráficas. Para isso, desenvolveu-se uma abordagem que permite aos *designers* utilizar ferramentas de prototipagem, neste caso foi escolhido Adobe XD, para desenhar interfaces gráficas, e depois convertê-las automaticamente para código Vue.js + Bootstrap, criando assim uma primeira versão da implementação. Esta geração é feita através da interpretação do ficheiro SVG que o Adobe XD exporta.

Na presente dissertação começa-se por abordar o desenho de interfaces de utilizador (Capítulo 2), abordando numa fase inicial o tema do desenvolvimento de interfaces de utilizador baseado em modelos e fazendo uma comparação entre este e o *design* centrado no utilizador. O principal ponto a favor destas abordagens é o facto de permitirem conceber soluções de uma forma mais independente da tecnologia que se utiliza. Em seguida, são abordadas as ferramentas de desenho de interfaces mais utilizadas, acabando por ser selecionado o Adobe XD como a ferramenta a utilizar para desenvolver *mockups*.

Em seguida, no Capítulo 3, abordou-se o desenvolvimento de aplicações interativas para o *web*. Começou-se por estudar o estado atual da área de desenvolvimento *web* e, para isso, numa fase inicial procedeu-se à definição de alguns conceitos, como o de aplicação *web*, *framework web*, *frameworks client-side* e *server-side* e analisou-se as vantagens e desvantagens associadas à utilização de *frameworks*. Para além disso, foram apresentados vários padrões de arquiteturas de *software* para o desenvolvimento de aplicações *web*, nomeadamente a arquitetura de três camadas, MVC, MVP e MVVM. Desta forma, foi possível obter uma visão mais alargada do estado atual da área de desenvolvimento *web*.

Consequentemente, as características mais relevantes das *frameworks web* foram estudadas e foi feito um resumo das características mais mencionadas por diversos autores. Desta forma, foi possível perceber melhor o que caracteriza uma *framework* e quais são as características que se deve ter em consideração aquando da escolha da *framework* a utilizar no próximo projeto *web*. Em seguida, foram estudadas várias *frameworks web*, através da análise de diferentes *rankings* que tinham por base diferentes premissas. As *frameworks* melhor classificadas nesses *rankings* foram selecionadas, culminando assim numa comparação dessas *frameworks* com base nas características previamente recolhidas. Foi decidido que o protótipo será convertido para código Vue.js.

Uma vez que o objetivo da presente dissertação é criar uma primeira versão da implementação de *mockups*, a este ponto tinha-se que o *designer* deve utilizar Adobe XD para desenhar interfaces gráficas, sendo que estas serão convertidas para código Vue.js.

Em seguida, no Capítulo 4, o objetivo prendeu-se em esquematizar a estrutura de um protótipo e desenvolver um metamodelo da arquitetura de componentes de Vue.js. Para isso, começa-se pela apresentação de um exemplo de um protótipo de interface gráfica e, posteriormente, é feita a esquematização da estrutura de um protótipo. Posteriormente, é desenvolvido um metamodelo da arquitetura de componentes de Vue.js e é feita a análise de arquiteturas de pastas/código-fonte de aplicações *web* em Vue.js. Em seguida, é feita uma pequena análise de como se traduz cada um dos componentes para código Vue.js e o capítulo termina com a análise de um exemplo de uma arquitetura típica em Vue.js e respetiva estrutura de pastas e componentes.

No Capítulo 5 começa-se por decidir a estratégia para conversão do protótipo da interface gráfica, nomeadamente através do formato SVG. Assim, após o *designer* desenvolver o *mockup* deve exportar o *design* para SVG (no próprio Adobe XD). Em seguida, apresentou-se quais os componentes Vue.js que serão focados na solução de *software* e o método que será utilizado para mapear os protótipos no metamodelo de Vue.js e o respetivo algoritmo genérico. Terminou-se com a explicação dos requisitos aos quais a abordagem deve responder.

No Capítulo 6 é explicada em detalhe a solução de *software* desenvolvida e os dois *scripts* elaborados em Python. No Capítulo 7 apresenta-se um exemplo prático da aplicação dos *scripts* desenvolvidos e é feita uma análise desse resultado. É, ainda, feita uma discussão dos resultados obtidos com a solução de *software* implementada.

Em suma, o grande objetivo proposto foi concluído com sucesso, isto é, foi definida uma solução para a geração automática de código da interface de utilizador, a partir de protótipos. Ao longo deste processo, os principais desafios estiveram presentes, numa fase inicial, na escolha da ferramenta de desenho de protótipos (Adobe XD) e na escolha da *framework web* (Vue.js), uma vez que existiam várias alternativas igualmente viáveis. Posteriormente, o desafio centrou-se em decidir qual seria a melhor estratégia para abordar o problema. Aqui optou-se por decidir que seria através do formato SVG, facilmente gerado através da ferramenta de protótipos escolhida (Adobe XD) e que seriam desenvolvidos *scripts* em Python para interpretar estes ficheiros SVG. O principal desafio prendeu-se então com a efetiva implementação do algoritmo genérico apresentado. Primeiro, na interpretação e organização de todas as *tags* presentes no ficheiro SVG (que no final se traduziram em componentes Vue.js), em seguida, em decifrar qual a posição relativa de cada um dos componentes (no sistema de *grid* do Bootstrap) e, por fim, em gerar o projeto Vue.js e os componentes com todas as características associadas.

Como já mencionado, existem várias ferramentas de prototipagem de interfaces gráficas que incluem já a funcionalidade de converter *mockups* para código HTML e CSS. Em muitos casos, isto é conseguido através da incorporação de uma imagem gráfica *bitmap* (com descrição da cor de cada píxel) ou vetorial (baseia-se em vetores matemáticos) do desenho no código HTML. Embora normalmente se obtenha um resultado que é quase idêntico ao que foi concebido, esta abordagem tem várias falhas, e é particularmente pobre do ponto de vista da geração do código.

Existem várias soluções atualmente no mercado que permitem desenhar uma interface gráfica e convertê-la para código *web*, como é o caso, por exemplo, do *Webflow*. O *Webflow*[1] permite a geração de código HTML, CSS e JavaScript (limpo e organizado, segundo os mesmos) por meio de uma interface completamente visual. Para além deste, poderiam ser apresentados muitos outros exemplos. No entanto, uma grande vantagem da solução proposta reside no facto da interface gráfica ser desenhada em Adobe XD. Adobe XD é uma ferramenta muito consolidada no mercado e muito utilizada. Desta forma, não é necessário que os *designers* aprendam a trabalhar com novas ferramentas, otimizando o seu tempo.

O facto de ser convertido para uma *framework web* consolidada e muito utilizada como é o caso de Vue.js, permite que o código gerado esteja já organizado numa ótica de componentes. Estes componentes organizados são muito facilmente modificados. Outra mais-valia é o facto dos componentes estarem organizados através de um sistema de *grid*, o que significa que não são utilizadas as posições absolutas dos componentes e, desta forma, a probabilidade de resultarem bem em dispositivos de diferentes dimensões é superior.

Com a abordagem proposta, não é possível obter um resultado tão idêntico, por exemplo, ao obtido com o HTML+CSS diretamente gerado pelas ferramentas de *mockups*, mas o resultado gráfico obtido é ainda bastante satisfatório, especialmente tendo em conta outras contrapartidas, especialmente a possibilidade de evolução do próprio código. A maior vantagem é a flexibilidade do próprio código gerado. O grau de capacidade de resposta é claramente superior e obtém-se um esqueleto de código Vue.js fácil de manter e de reaproveitar para melhorar cada vez mais o projeto.

Existem vários aspetos que podiam ser melhorados em trabalho futuro para que a solução obtida fosse cada vez mais satisfatória. Na implementação propriamente dita, como abordado no Capítulo 7.4, não foram implementadas todas as funcionalidades para que a geração da interface gráfica fosse muito flexível a nível de *design* para o *designer*. Como mencionado, foram feitas algumas assunções acerca do *input*, como os componentes serem visualmente semelhantes aos fornecidos pelo Bootstrap (ou seja, não existir, por exemplo botões redondos e triangulares), valores por omissão em relação ao espaçamento entre linhas, ao tamanho do texto, entre outros. Interpretar o ficheiro com o formato SVG é um enorme desafio e continua a ser um trabalho em curso, uma vez que é sempre possível melhorar o *script* de *parsing* porque se vai descobrindo outras formas de representação das *tags* no SVG.

Como trabalho futuro seria também importante testar a presente solução com *designers* e com *developers* e tentar perceber até que ponto a ideia é útil. No caso dos *designers*, perceber, por exemplo, se é aceitável pedir para colocarem os IDs em cada componente, e no caso dos *developers* se o código gerado é realmente satisfatório ou se prefeririam desenvolver tudo de raiz.

1 https://webflow.com/. Consultado a 17 de dezembro de 2021.

BIBLIOGRAFIA

Sandoche Adittane. How to structure a vue.js project, 2018. URL https://itnext.io/how-to-structure-a-vue-js-project-29e4ddc1aeeb. (Consultado a 30 de abril de 2021).

Adobe. Adobe XD documentation, 2021. URL https://helpx.adobe.com/xd/user-guide.html. (Consultado a 20 de julho de 2021).

Sabah Al-Fedaghi. Developing web applications. *International Journal of Software Engineering and Its Applications*, 5, 05 2011. doi: 10.1007/978-1-4302-3531-6_12.

Paris Avgeriou and Uwe Zdun. Architectural patterns revisited - a pattern language. In *EuroPLoP*, 2005.

Kristen Baker. 15 best wire-frame tools for your website design [2021 guide], 2021. URL https://blog.hubspot.com/website/wireframe-tools/. (Consultado a 23 de maio de 2021).

Arno Bergmann. Benefits and drawbacks of model-based design. *KMUTNB International Journal of Applied Science and Technology*, 7:15–19, 09 2014. doi: 10.14416/j.ijast.2014.04.004.

Ashish Bogawat. Sketch vs figma, adobexd, and other ui design applications, 2019. URL https://www.smashingmagazine.com/2019/04/sketch-figma-adobe-xd-ui-design-applications/. (Consultado a 23 de maio de 2021).

G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

Dasari Curie, Joyce Jaison, Jyoti Yadav, and J Fiona. Analysis on web frameworks. *Journal of Physics: Conference Series*, 1362:012114, 11 2019. doi: 10.1088/1742-6596/1362/1/012114.

Figma. Figma documentation, 2021. URL https://help.figma.com/. (Consultado a 20 de julho de 2021).

et al. G. Calvary. The cameleon reference framework, 2002. D1.1 of the CAMELEON Project R&D Project IST-2000-30104.

José Ignacio Fernández-Villamor, Laura Díaz-Casillas, and Carlos Á. Iglesias. A comparison model for agile web frameworks. In *Proceedings of the 2008 Euro American Conference on Telematics and Information Systems*, EATIS '08, New York, NY, USA, 2008. Association for Computing Machinery. doi: 10.1145/1621087.1621101.

Md Islam, Md Islam, and Tasneem Halim. A study of code cloning in server pages of web applications developed using classic asp.net and asp.net mvc framework. pages 497–502, 12 2011. ISBN 978-1-61284-907-2. doi: 10.1109/ICCITechn.2011.6164840.

ISO. ISO 9241-210:2019 Ergonomics of human-system interaction – part 210: Human-centred design for interactive systems. International Organization for Standardization, 2019.

Alex Ivanovs. Figma vs sketch vs adobe xd: Whichis the better design tool?, 2020. URL https://www.codeinwp.com/blog/figma-vs-sketch-vs-adobe-xd/. (Consultado a 23 de maio de 2021).

S. Kent. Model Driven Engineering. In *Integrated Formal Methods. IFM 2002*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002. doi: 10.1007/3-540-47884-1_16.

John Kouraklis. *MVVM as Design Pattern*. 10 2016. ISBN 978-1-4842-2213-3. doi: 10.1007/978-1-4842-2214-0_1.

Suryadiputra Liawatimena, Harco Leslie Hendric Spits Warnars, Agung Trisetyarso, Edi Abdurahman, Benfano Soewito, Antoni Wibowo, Ford Gaol, and Bahtiar Abbas. Django web framework software metrics measurement using radon and pylint. pages 218–222, 09 2018. doi: 10.1109/INAPR.2018.8627009.

Quentin Limbourg and Jean Vanderdonckt. Multipath transformational development of user interfaces with graph transformations. In *Human-Centered Software Engineering: Software Engineering Models, Patterns and Architectures for HCI*, pages 107–138. Springer, 2009. ISBN 978-1-84800-907-3. doi: 10.1007/978-1-84800-907-3_6.

C. Machado and J.C. Campos. Towards the integration of user interface prototyping and model-based development. In *2021 International Conference on Graphics and Interaction (ICGI) Proceedings*. IEEE, 2021. in press.

Rob Marvin. The best low-code development platforms, Aug 2018. URL https://www.pcmag.com/roundup/353252/the-best-low-code-development-platforms. (Consultado a 25 de janeiro de 2021).

Tom May and Carl Cahill. 53 web design tools to help you work smarter in 2021, 2021. URL https://www.creativebloq.com/features/best-web-design-tools/. (Consultado a 23 de maio de 2021).

Zakaria Mehrab, Raquib Bin Yousuf, Ibrahim Asadullah Tahmid, and Rifat Shahriyar. Mining developer questions about major web frameworks. pages 191–198, 01 2018. doi: 10.5220/0006929501910198.

Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, present, and future of model-based user interface development. *i-com*, 10(3):2–11, 2011.

Andrew Monk. User-centred design. In *Home Informatics and Telematics: Information, Technology and Society*, pages 181–190. Springer, 2000. ISBN 978-0-387-35511-5. doi: 10.1007/978-0-387-35511-5_14.

Maria Myre. The 8 best wireframe tools in 2021, 2021. URL https://zapier.com/blog/best-wireframe-tools/. (Consultado a 23 de maio de 2021).

Mohamadou Nassourou. Java web frameworks which one to choose? 01 2010.

Vensada Okanovic. Web application development with component frameworks. pages 889–892, 05 2014. ISBN 978-953-233-077-9. doi: 10.1109/MIPRO.2014.6859693.

Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction*, 16(4):19:1–19:30, November 2009.

Guilherme Pimenta. Tire suas dúvidas: Adobe xd, figma ou sketch, qual ferramenta de design escolher e por quê?, 2020. URL https://comunidade.rockcontent.com/adobe-xd-figma-ou-sketch/. (Consultado a 23 de maio de 2021).

Paulo Pinheiro da Silva. User interface declarative models and development environments: A survey. In *Interactive Systems Design, Specification, and Verification*, pages 207–226. Springer, 2001. ISBN 978-3-540-44675-0.

J. Plekhanova. Evaluating web development frameworks: Django, ruby on rails and cakephp. 2009.

Dragos-Paul Pop and Adam Altar Samuel. Designing an mvc model for rapid web application development. *Procedia Engineering*, 69, 12 2014. doi: 10.1016/j.proeng.2014.03.106.

Angel R. Puerta and Pedro Szkeley. Model-based interface development. In *Conference Companion on Human Factors in Computing Systems (CHI '94)*, pages 389–390. ACM, 1994.

María Salas Zarate, Giner Alor-Hernández, Rafael Valencia-García, Lisbeth Rodríguez, Alejandro González, and José Cuadrado. Analyzing best practices on web development frameworks: The lift approach. *Science of Computer Programming*, 102, 05 2015. doi: 10.1016/j.scico.2014.12.004.

Tony Shan and Winnie Hua. Taxonomy of java web application frameworks. pages 378–385, 10 2006. doi: 10.1109/ICEBE.2006.98.

Carlos Eduardo Silva and José Creissac Campos. Can gui implementation markup languages be used for modelling? In Marco Winckler, Peter Forbrig, and Regina Bernhaupt, editors, *Human-Centered Software Engineering*, pages 112–129, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-34347-6.

Sketch. Sketch documentation, 2021. URL https://www.sketch.com/docs/. (Consultado a 20 de julho de 2021).

Josh Smith. Patterns - WPF apps with the Model-View-ViewModel design pattern. *MSDN Magazine*, 24(2), February 2009.

Stack Overflow. Developer survey results 2017, 2017. URL https://insights.stackoverflow.com/survey/2017. (Consultado a 12 de janeiro de 2021).

Stack Overflow. 2020 developer survey, 2020. URL https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages. (Consultado a 12 de janeiro de 2021).

Emily Stevens. The fascinating history of ux design: A definitive timeline, July 2019. URL https://careerfoundry.com/en/blog/ux-design/the-fascinating-history-of-ux-design-a-definitive-timeline/. (Consultado a 24 de janeiro de 2021).

Iwan Vosloo and Derrick Kourie. Server-centric web frameworks: An overview. *ACM Comput. Surv.*, 40, 04 2008. doi: 10.1145/1348246.1348247.

Gottfried Vossen and Stephan Hagemann. From version 1.0 to version 2.0: A brief history of the web. 01 2007.

Jari-Pekka Voutilainen, Jaakko Salonen, and Tommi Mikkonen. On the design of a responsive user interface for a multi-device web service. pages 60–63, 05 2015. doi: 10.1109/MobileSoft.2015.16.

Vue.js. Documentação vue.js, 2021. URL https://vuejs.org/v2/guide/index.html/. (Consultado a 5 de abril de 2021).

Irena Vuksanovic and Bojan Sudarevic. Use of web application frameworks in the development of small applications. pages 458–462, 01 2011.

W3C. Scalable vector graphics (svg) 1.1 (second edition), 2011. URL https://www.w3.org/TR/SVG11/. (Consultado a 5 de junho de 2021).

D. Walker and A. Orooji. Metrics for web programming frameworks. In *International Conference on Semantic Web and Web Services*, 2011.

Eric Wohlgethan. Supporting web development decisions by comparing three major javascript frameworks: Angular, react and vue.js, 2018.

ANEXOS

A

EXEMPLO ILUSTRATIVO ESTRUTURA DE DADOS “COMPONENTS”

```
"Text": [
  {
    "id": "Text-5",
    "data-name": "Text",
    "transform": "translate(228 872)",
    "fill": "#707070",
    "font-size": "30",
    "font-family": "HelveticaNeue, Helvetica Neue",
    "width": "10",
    "height": "30",
    "text": "This is a big title example"
  }
]

"Img": [
  {
    "id": "Img-2",
    "transform": "translate(137 114)",
    "{http://www.w3.org/1999/xlink}href": "#image",
    "image": "User-image.png",
    "width": "437",
    "height": "316"
  }
]

"Btn": [
  {
    "id": "Btn",
    "transform": "translate(1150 596)",
    "fill": "#fff",
    "stroke": "#fff",
    "text": "Create account",
    "width": "261",
    "height": "61",
    "text_color": "#38487e"
```

```
    }
]
```

B

INICIO.SVG

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" width="1920" height="1260" viewBox="0 0 1920 1260">
<defs>
  <pattern id="pattern" preserveAspectRatio="none" width="100%" height="100%" viewBox="0 0 259 165">
    <image width="259" height="165" xlink:href="img3.jpeg"/>
  </pattern>
  <pattern id="pattern-2" preserveAspectRatio="none" width="100%" height="100%" viewBox="0 0 4480 794">
    <image width="4480" height="794" xlink:href="campo-tenis.jpg"/>
  </pattern>
  <clipPath id="clip-Inicio">
    <rect width="1920" height="1260"/>
  </clipPath>
</defs>
<g id="Inicio" clip-path="url(#clip-Inicio)">
  <rect width="1920" height="1260" fill="#fff"/>
  <g id="Footer" transform="translate(0 1177)" fill="#5e93d1" stroke="#5e93d1" stroke-width="1">
    <rect width="1920" height="83" stroke="none"/>
    <rect x="0.5" y="0.5" width="1919" height="82" fill="none"/>
  </g>
  <g id="Navbar" fill="#fff" stroke="#707070" stroke-width="1">
    <rect width="1920" height="235" stroke="none"/>
    <rect x="0.5" y="0.5" width="1919" height="234" fill="none"/>
  </g>
  <text id="Text" transform="translate(800 1220)" fill="#fff" font-size="25" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Todos os direitos reservados</tspan><tspan x="0" y="30"></tspan></text>
  <text id="Text-2" data-name="Text" transform="translate(480 703)" font-size="25" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Vem conhecer o Clube de Tenis, consulta as nossas instalacoes e faz ja a tua reserva!</tspan></text>
  <g id="Btn" transform="translate(182 1041)">
```

```
    <g id="Btn-2" data-name="Btn" fill="#dd983c" stroke="#dd983c" stroke-width="1">
      <rect width="261" height="64" stroke="none"/>
      <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
    </g>
    <text id="Text-3" data-name="Text" transform="translate(65 41)" fill="#fff" font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Competicao</tspan></text>
  </g>
  <g id="Btn-3" data-name="Btn" transform="translate(829 1041)">
    <g id="Btn-4" data-name="Btn" fill="#dd983c" stroke="#dd983c" stroke-width="1">
      <rect width="261" height="64" stroke="none"/>
      <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
    </g>
    <text id="Text-4" data-name="Text" transform="translate(40 41)" fill="#fff" font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Aulas Individuais</tspan></text>
  </g>
  <g id="Btn-5" data-name="Btn" transform="translate(1481 1041)">
    <g id="Btn-6" data-name="Btn" fill="#dd983c" stroke="#dd983c" stroke-width="1">
      <rect width="261" height="64" stroke="none"/>
      <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
    </g>
    <text id="Text-5" data-name="Text" transform="translate(32 41)" fill="#fff" font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Criancas e Jovens</tspan></text>
  </g>
  <image id="Img" width="306" height="202" transform="translate(159 792)" xlink:href="img1.jpeg"/>
  <image id="Img-2" data-name="Img" width="306" height="202" transform="translate(806 792)" xlink:href="img2.jpeg"/>
  <rect id="Img-3" data-name="Img" width="306" height="202" transform="translate(1453 792)" fill="url(#pattern)"/>
  <g id="Btn-7" data-name="Btn" transform="translate(867 104)">
    <g id="Btn-8" data-name="Btn" fill="#4477c1" stroke="#4477c1" stroke-width="1">
      <rect width="261" height="64" stroke="none"/>
      <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
    </g>
    <text id="Text-6" data-name="Text" transform="translate(87 41)" fill="#fff" font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Noticias</tspan></text>
  </g>
  <g id="Btn-9" data-name="Btn" transform="translate(1169 104)">
```

```
      <g id="Btn-10" data-name="Btn" fill="#4477c1" stroke="#4477c1" stroke-width="1">
        <rect width="261" height="64" stroke="none"/>
        <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
      </g>
      <text id="Text-7" data-name="Text" transform="translate(70 41)" fill="#fff" font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Instalacoes</tspan></text>
    </g>
    <g id="Btn-11" data-name="Btn" transform="translate(1471 104)">
      <g id="Btn-12" data-name="Btn" fill="#4477c1" stroke="#4477c1" stroke-width="1">
        <rect width="261" height="64" stroke="none"/>
        <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
      </g>
      <text id="Text-8" data-name="Text" transform="translate(81 41)" fill="#fff" font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Reservas</tspan></text>
    </g>
    <image id="Img-4" data-name="Img" width="158" height="158" transform="translate(166 48)" xlink:href="raquete.png"/>
    <rect id="Img-5" data-name="Img" width="1889" height="335" transform="translate(16 260)" fill="url(#pattern-2)"/>
  </g>
</svg>
```

C

RESERVAS.SVG

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" width="1920" height="1260" viewBox="0 0 1920 1260">
  <defs>
    <clipPath id="clip-Reservas">
      <rect width="1920" height="1260"/>
    </clipPath>
  </defs>
  <g id="Reservas" clip-path="url(#clip-Reservas)">
    <rect width="1920" height="1260" fill="#fff"/>
    <g id="Navbar" fill="#fff" stroke="#707070" stroke-width="1">
      <rect width="1920" height="235" stroke="none"/>
      <rect x="0.5" y="0.5" width="1919" height="234" fill="none"/>
    </g>
    <text id="Text" transform="translate(596 1098)" font-size="25" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">marcacoes@clubedetenis.pt</tspan></text>
    <text id="Text-2" data-name="Text" transform="translate(1122 1098)" font-size="25" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">+351 900 000 000</tspan></text>
    <text id="Text-3" data-name="Text" transform="translate(168 1098)" font-size="25" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">Rua do Tenis N.12</tspan></text>
    <text id="Text-4" data-name="Text" transform="translate(1536 1098)" font-size="25" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0">@instagram.tenis</tspan></text>
    <image id="Img" width="499" height="568" transform="translate(1122 416)" xlink:href="play.png"/>
    <g id="Btn" transform="translate(867 104)">
      <g id="Btn-2" data-name="Btn" fill="#4477c1" stroke="#4477c1" stroke-width="1">
        <rect width="261" height="64" stroke="none"/>
        <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
      </g>
```

```
23        <text id="Text-5" data-name="Text" transform="translate(87 41)" fill="#fff"
 font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0"
>Noticias</tspan></text>
      </g>
25    <g id="Btn-3" data-name="Btn" transform="translate(1169 104)">
        <g id="Btn-4" data-name="Btn" fill="#4477c1" stroke="#4477c1" stroke-width=
"1">
27        <rect width="261" height="64" stroke="none"/>
          <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
29      </g>
        <text id="Text-6" data-name="Text" transform="translate(70 41)" fill="#fff"
 font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0"
>Instalacoes</tspan></text>
31    </g>
      <g id="Btn-5" data-name="Btn" transform="translate(1471 104)">
33      <g id="Btn-6" data-name="Btn" fill="#4477c1" stroke="#4477c1" stroke-width=
"1">
          <rect width="261" height="64" stroke="none"/>
35        <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
        </g>
37      <text id="Text-7" data-name="Text" transform="translate(81 41)" fill="#fff"
 font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0"
>Reservas</tspan></text>
      </g>
39    <image id="Img-2" data-name="Img" width="158" height="158" transform="
translate(166 48)" xlink:href="raquete.png"/>
      <g id="Footer" transform="translate(0 1177)" fill="#5e93d1" stroke="#5e93d1"
stroke-width="1">
41      <rect width="1920" height="83" stroke="none"/>
        <rect x="0.5" y="0.5" width="1919" height="82" fill="none"/>
43    </g>
      <text id="Text-8" data-name="Text" transform="translate(800 1220)" fill="#fff
" font-size="25" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0
">Todos os direitos reservados</tspan><tspan x="0" y="30"></tspan></text>
45    <g id="Form" transform="translate(158 483)" fill="rgba(255,255,255,0)" stroke
="#000" stroke-width="1">
        <rect width="911" height="444" stroke="none"/>
47      <rect x="0.5" y="0.5" width="910" height="443" fill="none"/>
      </g>
49    <text id="Text-9" data-name="Text" transform="translate(365 621)" fill="#
dd983c" font-size="30" font-family="HelveticaNeue, Helvetica Neue"><tspan x="
-66.675" y="0">Telemovel</tspan></text>
      <text id="Text-10" data-name="Text" transform="translate(322 567)" fill="#
dd983c" font-size="30" font-family="HelveticaNeue, Helvetica Neue"><tspan x="
-110.31" y="0">Nome Completo</tspan></text>
51    <g id="TextBox" transform="translate(454 592)" fill="#fff" stroke="#707070"
stroke-width="1">
```

```
      <rect width="471" height="35" stroke="none"/>
      <rect x="0.5" y="0.5" width="470" height="34" fill="none"/>
    </g>
    <g id="TextBox-2" data-name="TextBox" transform="translate(454 538)" fill="#
    fff" stroke="#707070" stroke-width="1">
      <rect width="471" height="35" stroke="none"/>
      <rect x="0.5" y="0.5" width="470" height="34" fill="none"/>
    </g>
    <text id="Text-11" data-name="Text" transform="translate(402 729)" fill="#
    dd983c" font-size="30" font-family="HelveticaNeue, Helvetica Neue"><tspan x="
    -31.395" y="0">Data</tspan></text>
    <g id="TextBox-3" data-name="TextBox" transform="translate(454 646)" fill="#
    fff" stroke="#707070" stroke-width="1">
      <rect width="471" height="35" stroke="none"/>
      <rect x="0.5" y="0.5" width="470" height="34" fill="none"/>
    </g>
    <text id="Text-12" data-name="Text" transform="translate(998 336)" fill="#
    dd983c" font-size="40" font-family="HelveticaNeue, Helvetica Neue"><tspan x="
    -183.34" y="0">Reservas de Espaco</tspan></text>
    <g id="Btn-7" data-name="Btn" transform="translate(483 818)">
      <g id="Btn-8" data-name="Btn" fill="#dd983c" stroke="#dd983c" stroke-width=
    "1">
        <rect width="261" height="64" stroke="none"/>
        <rect x="0.5" y="0.5" width="260" height="63" fill="none"/>
      </g>
      <text id="Text-13" data-name="Text" transform="translate(78 41)" fill="#fff
    " font-size="24" font-family="HelveticaNeue, Helvetica Neue"><tspan x="0" y="0
    ">Submeter</tspan></text>
    </g>
    <text id="Text-14" data-name="Text" transform="translate(395 675)" fill="#
    dd983c" font-size="30" font-family="HelveticaNeue, Helvetica Neue"><tspan x="
    -36.675" y="0">Email</tspan></text>
    <g id="TextBox-4" data-name="TextBox" transform="translate(454 700)" fill="#
    fff" stroke="#707070" stroke-width="1">
      <rect width="471" height="35" stroke="none"/>
      <rect x="0.5" y="0.5" width="470" height="34" fill="none"/>
    </g>
  </g>
</svg>
```

D

NAVBAR.VUE

```
<template>
  <nav class="navbar page-navbar">
    <div class="container">
      <div class="row">
        <div class="col-1">
          <img src="../assets/raquete.png">
        </div>
        <div class="col-3">
        </div>
        <div class="col-2">
          <button class="btn btn-outline-secondary Btn-id-Reservas" type="button"
    >
            Noticias
          </button>
        </div>
        <div class="col-2">
          <button class="btn btn-outline-secondary Btn-3-id-Reservas" type="
    button">
            Instalacoes
          </button>
        </div>
        <div class="col-2">
          <router-link :to="{name: 'Reservas'}">
          <button class="btn btn-outline-secondary Btn-5-id-Reservas" type="
    button">
            Reservas
          </button>
          </router-link>
        </div>
      </div>
    </div>
  </nav>
</template>

<style scoped>
```

```
.page-navbar {
34          background-color: #fff !important;
}
36 </style>
```

E

FOOTER.VUE

```
<template>
  <footer class="page-footer">
    <div class="row" style="padding-bottom: 40px;">
        <div class="col-12">
          <div class="Text-8-id-Reservas">
          Todos os direitos reservados
          </div>
        </div>
    </div>
  </footer>
</template>

<style scoped>
.page-footer {
    background-color: #5e93d1 !important;
}
</style>
```

F

INICIO.VUE

```
<template>
  <div id="app">
    <div class="container">

      <div class="row" style="padding-bottom: 40px;">
        <div class="col-12">
          <img src="./assets/campo-tenis.jpg">
        </div>
      </div>

      <div class="row" style="padding-bottom: 40px;">
        <div class="col-3">
        </div>
        <div class="col-12">
          <div class="Text-2-id-Inicio">
          Vem conhecer o Clube de Tenis, consulta as nossas instalacoes e faz ja
    a tua reserva!
          </div>
        </div>
      </div>

      <div class="row" style="padding-bottom: 40px;">
        <div class="col-1">
        </div>
        <div class="col-2">
          <img src="./assets/img1.jpeg">
        </div>
        <div class="col-2">
        </div>
        <div class="col-2">
          <img src="./assets/img2.jpeg">
        </div>
        <div class="col-2">
        </div>
        <div class="col-2">
```

```
          <img src="./assets/img3.jpeg">
        </div>
      </div>

      <div class="row" style="padding-bottom: 40px;">
        <div class="col-1">
        </div>
        <div class="col-2">
          <button class="btn btn-outline-secondary Btn-id-Inicio" type="button">
            Competicao
          </button>
        </div>
        <div class="col-2">
        </div>
        <div class="col-2">
          <button class="btn btn-outline-secondary Btn-3-id-Inicio" type="button"
    >
            Aulas Individuais
          </button>
        </div>
        <div class="col-2">
        </div>
        <div class="col-2">
          <button class="btn btn-outline-secondary Btn-5-id-Inicio" type="button"
    >
            Criancas e Jovens
          </button>
        </div>
      </div>
    </div>
  </div>
</template>


<script>
export default {
  name: 'Inicio',
  components: {
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
```

```
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}

img{
  max-width: 100%;
  height :auto;
}
</style>
```

G

RESERVAS.VUE

```
<template>
  <div id="app">
    <div class="container">

      <div class="row" style="padding-bottom: 40px;">
        <div class="col-6">
        </div>
        <div class="col-12">
          <div class="Text-12-id-Reservas">
          Reservas de Espaco
          </div>
        </div>
      </div>

      <div class="row" style="padding-bottom: 40px;">
        <div class="col-1">
        </div>
        <div class="col-6">
          <form>
            <div class="form-group row" style="padding-bottom: 40px;">
              <label for="Text-10-id-Reservas" class="col-sm-3 col-form-label
    Text-10-id-Reservas">Nome Completo</label>
              <div class="col-sm-9">
                <input class="form-control" id="Text-10-id-Reservas">
              </div>
            </div>
            <div class="form-group row" style="padding-bottom: 40px;">
              <label for="Text-9-id-Reservas" class="col-sm-3 col-form-label Text
    -9-id-Reservas">Telemovel</label>
              <div class="col-sm-9">
                <input class="form-control" id="Text-9-id-Reservas">
              </div>
            </div>
            <div class="form-group row" style="padding-bottom: 40px;">
```

```
33              <label for="Text-14-id-Reservas" class="col-sm-3 col-form-label
    Text-14-id-Reservas">Email</label>
                <div class="col-sm-9">
35                <input class="form-control" id="Text-14-id-Reservas">
                </div>
37            </div>
              <div class="form-group row" style="padding-bottom: 40px;">
39              <label for="Text-11-id-Reservas" class="col-sm-3 col-form-label
    Text-11-id-Reservas">Data</label>
                <div class="col-sm-9">
41                <input class="form-control" id="Text-11-id-Reservas">
                </div>
43            </div>
              <button class="btn btn-outline-secondary Btn-7-id-Reservas" type="
    button">
45              Submeter
              </button>
47          </form>
        </div>
49      <div class="col-2">
        </div>
51      <div class="col-3">
          <img src="./assets/play.png">
53      </div>
    </div>
55
    <div class="row" style="padding-bottom: 40px;">
57    <div class="col-3">
        <div class="Text-3-id-Reservas">
59      Rua do Tenis N.12
        </div>
61    </div>
      <div class="col-3">
63      <div class="Text-id-Reservas">
        marcacoes@clubedetenis.pt
65      </div>
      </div>
67    <div class="col-3">
        <div class="Text-2-id-Reservas">
69      +351 900 000 000
        </div>
71    </div>
      <div class="col-3">
73      <div class="Text-4-id-Reservas">
        @instagram.tenis
75      </div>
      </div>
```

```
      </div>
    </div>
  </div>
</template>


<script>
export default {
  name: 'Reservas',
  components: {
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}

img{
  max-width: 100%;
  height :auto;
}
</style>
```

H

CSS.CSS

```
.Btn-7-id-Inicio {
    color: #fff !important;
    background-color: #4477c1 !important;
    border-color: #4477c1 !important;
}
.Btn-9-id-Inicio {
    color: #fff !important;
    background-color: #4477c1 !important;
    border-color: #4477c1 !important;
}
.Btn-11-id-Inicio {
    color: #fff !important;
    background-color: #4477c1 !important;
    border-color: #4477c1 !important;
}
.Text-2-id-Inicio {
    color: None !important;
}
.Btn-id-Inicio {
    color: #fff !important;
    background-color: #dd983c !important;
    border-color: #dd983c !important;
}
.Btn-3-id-Inicio {
    color: #fff !important;
    background-color: #dd983c !important;
    border-color: #dd983c !important;
}
.Btn-5-id-Inicio {
    color: #fff !important;
    background-color: #dd983c !important;
    border-color: #dd983c !important;
}
.Text-id-Inicio {
    color: #fff !important;
```

```
}
.Btn-id-Reservas {
    color: #fff !important;
    background-color: #4477c1 !important;
    border-color: #4477c1 !important;
}
.Btn-3-id-Reservas {
    color: #fff !important;
    background-color: #4477c1 !important;
    border-color: #4477c1 !important;
}
.Btn-5-id-Reservas {
    color: #fff !important;
    background-color: #4477c1 !important;
    border-color: #4477c1 !important;
}
.Text-12-id-Reservas {
    color: #dd983c !important;
}
.Text-10-id-Reservas {
    color: #dd983c !important;
}
.Text-9-id-Reservas {
    color: #dd983c !important;
}
.Text-14-id-Reservas {
    color: #dd983c !important;
}
.Text-11-id-Reservas {
    color: #dd983c !important;
}
.Btn-7-id-Reservas {
    color: #fff !important;
    background-color: #dd983c !important;
    border-color: #dd983c !important;
}
.Text-3-id-Reservas {
    color: None !important;
}
.Text-id-Reservas {
    color: None !important;
}
.Text-2-id-Reservas {
    color: None !important;
}
.Text-4-id-Reservas {
    color: None !important;
```

```
}
.Text-8-id-Reservas {
    color: #fff !important;
}
```

I

ESTRUTURA DE DADOS INICIAL - INICIO


```
{
  "pattern": [
    {
      "id": "pattern",
      "preserveAspectRatio": "none",
      "viewBox": "0 0 259 165",
      "image": "img3.jpeg"
    },
    {
      "id": "pattern-2",
      "preserveAspectRatio": "none",
      "viewBox": "0 0 4480 794",
      "image": "campo-tenis.jpg"
    }
  ],
  "clip": [
    {
      "id": "clip-Inicio",
      "width": "1920",
      "height": "1260"
    }
  ],
  "image": [],
  "Text": [
    {
      "id": "Text",
      "transform": "translate(800 1220)",
      "fill": "#fff",
      "font-size": "25",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "25",
      "text": "Todos os direitos reservados"
    },
    {
```

```
      "id": "Text-2",
      "data-name": "Text",
      "transform": "translate(480 703)",
      "font-size": "25",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "25",
      "text": "Vem conhecer o Clube de T\u00e9nis, consulta as nossas instala\
    u00e7\u00f5es e faz j\u00e1 a tua reserva!"
    }
  ],
  "Img": [
    {
      "id": "Img",
      "width": "306",
      "height": "202",
      "transform": "translate(159 792)",
      "{http://www.w3.org/1999/xlink}href": "img1.jpeg",
      "image": "img1.jpeg"
    },
    {
      "id": "Img-2",
      "data-name": "Img",
      "width": "306",
      "height": "202",
      "transform": "translate(806 792)",
      "{http://www.w3.org/1999/xlink}href": "img2.jpeg",
      "image": "img2.jpeg"
    },
    {
      "id": "Img-3",
      "data-name": "Img",
      "width": "306",
      "height": "202",
      "transform": "translate(1453 792)",
      "fill": "url(#pattern)",
      "image": "img3.jpeg"
    },
    {
      "id": "Img-4",
      "data-name": "Img",
      "width": "158",
      "height": "158",
      "transform": "translate(166 48)",
      "{http://www.w3.org/1999/xlink}href": "raquete.png",
      "image": "raquete.png"
    },
```

```
    {
      "id": "Img-5",
      "data-name": "Img",
      "width": "1889",
      "height": "335",
      "transform": "translate(16 260)",
      "fill": "url(#pattern-2)",
      "image": "campo-tenis.jpg"
    }
  ],
  "Btn": [
    {
      "id": "Btn",
      "transform": "translate(182 1041)",
      "fill": "#dd983c",
      "stroke": "#dd983c",
      "text": "Competi\u00e7\u00e3o",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-3",
      "data-name": "Btn",
      "transform": "translate(829 1041)",
      "fill": "#dd983c",
      "stroke": "#dd983c",
      "text": "Aulas Individuais",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-5",
      "data-name": "Btn",
      "transform": "translate(1481 1041)",
      "fill": "#dd983c",
      "stroke": "#dd983c",
      "text": "Crian\u00e7as e Jovens",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-7",
      "data-name": "Btn",
      "transform": "translate(867 104)",
```

```
      "fill": "#4477c1",
      "stroke": "#4477c1",
      "text": "Not\u00edcias",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-9",
      "data-name": "Btn",
      "transform": "translate(1169 104)",
      "fill": "#4477c1",
      "stroke": "#4477c1",
      "text": "Instala\u00e7\u00f5es",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-11",
      "data-name": "Btn",
      "transform": "translate(1471 104)",
      "fill": "#4477c1",
      "stroke": "#4477c1",
      "text": "Reservas",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    }
  ],
  "Navbar": [
    {
      "id": "Navbar",
      "fill": "#fff",
      "stroke": "#707070",
      "stroke-width": "1",
      "width": "1920",
      "height": "235",
      "transform": "translate(0 0)"
    }
  ],
  "Footer": [
    {
      "id": "Footer",
      "transform": "translate(0 1177)",
      "fill": "#5e93d1",
      "stroke": "#5e93d1",
```

```
      "stroke-width": "1",
      "width": "1920",
      "height": "83"
    }
  ],
  "Form": [],
  "TextBox": []
}
```

J

ESTRUTURA DE DADOS INICIAL - RESERVAS


```
{
  "pattern": [],
  "clip": [
    {
      "id": "clip-Reservas",
      "width": "1920",
      "height": "1260"
    }
  ],
  "image": [],
  "Text": [
    {
      "id": "Text",
      "transform": "translate(596 1098)",
      "font-size": "25",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "25",
      "text": "marcacoes@clubedetenis.pt"
    },
    {
      "id": "Text-2",
      "data-name": "Text",
      "transform": "translate(1122 1098)",
      "font-size": "25",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "25",
      "text": "+351 900 000 000"
    },
    {
      "id": "Text-3",
      "data-name": "Text",
      "transform": "translate(168 1098)",
      "font-size": "25",
```

```
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "25",
      "text": "Rua do T\u00e9nis N.\u00ba12"
    },
    {
      "id": "Text-4",
      "data-name": "Text",
      "transform": "translate(1536 1098)",
      "font-size": "25",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "25",
      "text": "@instagram.tenis"
    },
    {
      "id": "Text-8",
      "data-name": "Text",
      "transform": "translate(800 1220)",
      "fill": "#fff",
      "font-size": "25",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "25",
      "text": "Todos os direitos reservados"
    },
    {
      "id": "Text-9",
      "data-name": "Text",
      "transform": "translate(365 621)",
      "fill": "#dd983c",
      "font-size": "30",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "30",
      "text": "Telem\u00f3vel"
    },
    {
      "id": "Text-10",
      "data-name": "Text",
      "transform": "translate(322 567)",
      "fill": "#dd983c",
      "font-size": "30",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "30",
      "text": "Nome Completo"
```

```
    },
    {
      "id": "Text-11",
      "data-name": "Text",
      "transform": "translate(402 729)",
      "fill": "#dd983c",
      "font-size": "30",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "30",
      "text": "Data"
    },
    {
      "id": "Text-12",
      "data-name": "Text",
      "transform": "translate(998 336)",
      "fill": "#dd983c",
      "font-size": "40",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "40",
      "text": "Reservas de Espa\u00e7o"
    },
    {
      "id": "Text-14",
      "data-name": "Text",
      "transform": "translate(395 675)",
      "fill": "#dd983c",
      "font-size": "30",
      "font-family": "HelveticaNeue, Helvetica Neue",
      "width": 0,
      "height": "30",
      "text": "Email"
    }
  ],
  "Img": [
    {
      "id": "Img",
      "width": "499",
      "height": "568",
      "transform": "translate(1122 416)",
      "{http://www.w3.org/1999/xlink}href": "play.png",
      "image": "play.png"
    },
    {
      "id": "Img-2",
      "data-name": "Img",
```

```
      "width": "158",
      "height": "158",
      "transform": "translate(166 48)",
      "{http://www.w3.org/1999/xlink}href": "raquete.png",
      "image": "raquete.png"
    }
  ],
  "Btn": [
    {
      "id": "Btn",
      "transform": "translate(867 104)",
      "fill": "#4477c1",
      "stroke": "#4477c1",
      "text": "Not\u00edcias",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-3",
      "data-name": "Btn",
      "transform": "translate(1169 104)",
      "fill": "#4477c1",
      "stroke": "#4477c1",
      "text": "Instala\u00e7\u00f5es",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-5",
      "data-name": "Btn",
      "transform": "translate(1471 104)",
      "fill": "#4477c1",
      "stroke": "#4477c1",
      "text": "Reservas",
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    },
    {
      "id": "Btn-7",
      "data-name": "Btn",
      "transform": "translate(483 818)",
      "fill": "#dd983c",
      "stroke": "#dd983c",
      "text": "Submeter",
```

```
      "width": "261",
      "height": "64",
      "text_color": "#fff"
    }
  ],
  "Navbar": [
    {
      "id": "Navbar",
      "fill": "#fff",
      "stroke": "#707070",
      "stroke-width": "1",
      "width": "1920",
      "height": "235",
      "transform": "translate(0 0)"
    }
  ],
  "Footer": [
    {
      "id": "Footer",
      "transform": "translate(0 1177)",
      "fill": "#5e93d1",
      "stroke": "#5e93d1",
      "stroke-width": "1",
      "width": "1920",
      "height": "83"
    }
  ],
  "Form": [
    {
      "id": "Form",
      "transform": "translate(158 483)",
      "fill": "rgba(255,255,255,0)",
      "stroke": "#000",
      "stroke-width": "1",
      "width": "911",
      "height": "444"
    }
  ],
  "TextBox": [
    {
      "id": "TextBox",
      "transform": "translate(454 592)",
      "fill": "#fff",
      "stroke": "#707070",
      "stroke-width": "1",
      "width": "471",
      "height": "35"
```

```
    },
    {
      "id": "TextBox-2",
      "data-name": "TextBox",
      "transform": "translate(454 538)",
      "fill": "#fff",
      "stroke": "#707070",
      "stroke-width": "1",
      "width": "471",
      "height": "35"
    },
    {
      "id": "TextBox-3",
      "data-name": "TextBox",
      "transform": "translate(454 646)",
      "fill": "#fff",
      "stroke": "#707070",
      "stroke-width": "1",
      "width": "471",
      "height": "35"
    },
    {
      "id": "TextBox-4",
      "data-name": "TextBox",
      "transform": "translate(454 700)",
      "fill": "#fff",
      "stroke": "#707070",
      "stroke-width": "1",
      "width": "471",
      "height": "35"
    }
  ]
}
```

K

ESTRUTURA DE DADOS COMPLETA - INICIO


```
{
  "1": {
    "1": [
      [
        "Navbar",
        [
          {"x": 0},
          {"y": 0},
          {"width": 1920},
          {"height": 235},
          {"x_final": 1920},
          {"y_final": 235},
          {"fill": "#fff"},
          {"stroke": "#707070"},
          {"col": 12}
        ]
      ],
      [
        "Img-4",
        [
          {"x": 166},
          {"y": 48},
          {"width": 158},
          {"height": 158},
          {"x_final": 324},
          {"y_final": 206},
          {"fill": null},
          {"stroke": null},
          {"col": 1}
        ]
      ],
      [
        "Empty",
        [
          {"x": 324},
```

```
          {"y": 104},
          {"width": 543},
          {"x_final": 867},
          {"y_final": 104},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 3}
        ]
      ],
      [
        "Btn-7",
        [
          {"x": 867},
          {"y": 104},
          {"width": 261},
          {"height": 64},
          {"x_final": 1128},
          {"y_final": 168},
          {"fill": "#4477c1"},
          {"stroke": "#4477c1",
          {"col": 2}
        ]
      ],
      [
        "Btn-9",
        [
          {"x": 1169},
          {"y": 104},
          {"width": 261},
          {"height": 64},
          {"x_final": 1430},
          {"y_final": 168},
          {"fill": "#4477c1"},
          {"stroke": "#4477c1"},
          {"col": 2}
        ]
      ],
      [
        "Btn-11",
        [
          {"x": 1471},
          {"y": 104},
          {"width": 261},
          {"height": 64},
          {"x_final": 1732},
          {"y_final": 168},
          {"fill": "#4477c1"},
```

```
          {"stroke": "#4477c1"},
          {"col": 2}
        ]
      ]
    ]
  },
  "2": {
    "1": [
      [
        "Img-5",
        [
          {"x": 16},
          {"y": 260},
          {"width": 1889},
          {"height": 335},
          {"x_final": 1905},
          {"y_final": 595},
          {"fill": "url(#pattern-2)"},
          {"stroke": null},
          {"col": 12}
        ]
      ]
    ]
  },
  "3": {
    "1": [
      [
        "Empty",
        [
          {"x": -1},
          {"y": 703},
          {"width": 481},
          {"x_final": 480},
          {"y_final": 703},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 3}
        ]
      ],
      [
        "Text-2",
        [
          {"x": 480},
          {"y": 703},
          {"width": 0},
          {"height": 25},
          {"x_final": 480},
```

```
          {"y_final": 728},
          {"fill": null},
          {"stroke": null},
          {"col": 12}
        ]
      ]
    ]
  },
  "4": {
    "1": [
      [
        "Empty",
        [
          {"x": -1},
          {"y": 792},
          {"width": 160},
          {"x_final": 159},
          {"y_final": 792},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 1}
        ]
      ],
      [
        "Img",
        [
          {"x": 159},
          {"y": 792},
          {"width": 306},
          {"height": 202},
          {"x_final": 465},
          {"y_final": 994},
          {"fill": null},
          {"stroke": null},
          {"col": 2}
        ]
      ]
    ],
    "2": [
      [
        "Empty",
        [
          {"x": 465},
          {"y": 792},
          {"width": 341},
          {"x_final": 806},
          {"y_final": 792},
```

```
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 2}
        ]
      ],
      [
        "Img-2",
        [
          {"x": 806},
          {"y": 792},
          {"width": 306},
          {"height": 202},
          {"x_final": 1112},
          {"y_final": 994},
          {"fill": null},
          {"stroke": null},
          {"col": 2}
        ]
      ]
    ],
    "3": [
      [
        "Empty",
        [
          {"x": 1112},
          {"y": 792},
          {"width": 341},
          {"x_final": 1453},
          {"y_final": 792},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 2}
        ]
      ],
      [
        "Img-3",
        [
          {"x": 1453},
          {"y": 792},
          {"width": 306},
          {"height": 202},
          {"x_final": 1759},
          {"y_final": 994},
          {"fill": "url(#pattern)"},
          {"stroke": null},
          {"col": 2}
        ]
```

```
      ]
    ]
  },
  "5": {
    "1": [
      [
        "Empty",
        [
          {"x": -1},
          {"y": 1041},
          {"width": 183},
          {"x_final": 182},
          {"y_final": 1041},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 1}
        ]
      ],
      [
        "Btn",
        [
          {"x": 182},
          {"y": 1041},
          {"width": 261},
          {"height": 64},
          {"x_final": 443},
          {"y_final": 1105},
          {"fill": "#dd983c"},
          {"stroke": "#dd983c"},
          {"col": 2}
        ]
      ]
    ],
    "2": [
      [
        "Empty",
        [
          {"x": 443},
          {"y": 1041},
          {"width": 386},
          {"x_final": 829},
          {"y_final": 1041},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 2}
        ]
      ],
```

```
      [
        "Btn-3",
        [
          {"x": 829},
          {"y": 1041},
          {"width": 261},
          {"height": 64},
          {"x_final": 1090},
          {"y_final": 1105},
          {"fill": "#dd983c"},
          {"stroke": "#dd983c"},
          {"col": 2}
        ]
      ]
    ],
    "3": [
      [
        "Empty",
        [
          {"x": 1090},
          {"y": 1041},
          {"width": 391},
          {"x_final": 1481},
          {"y_final": 1041},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 2}
        ]
      ],
      [
        "Btn-5",
        [
          {"x": 1481},
          {"y": 1041},
          {"width": 261},
          {"height": 64},
          {"x_final": 1742},
          {"y_final": 1105},
          {"fill": "#dd983c"},
          {"stroke": "#dd983c"},
          {"col": 2
          }
        ]
      ]
    ]
  },
  "6": {
```

```
    "1": [
      [
        "Footer",
        [
          {"x": 0},
          {"y": 1177},
          {"width": 1920},
          {"height": 83},
          {"x_final": 1920},
          {"y_final": 1260},
          {"fill": "#5e93d1"},
          {"stroke": "#5e93d1"},
          {"col": 12}
        ]
      ],
      [
        "Text",
        [
          {"x": 800},
          {"y": 1220},
          {"width": 0},
          {"height": 25},
          {"x_final": 800},
          {"y_final": 1245},
          {"fill": "#fff"},
          {"stroke": null},
          {"col": 12}
        ]
      ]
    ]
  }
}
```

L

ESTRUTURA DE DADOS COMPLETA - RESERVAS


```
{
  "1": {
    "1": [
      [
        "Navbar",
        [
          {"x": 0},
          {"y": 0},
          {"width": 1920},
          {"height": 235},
          {"x_final": 1920},
          {"y_final": 235},
          {"fill": "#fff"},
          {"stroke": "#707070"},
          {"col": 12}
        ]
      ],
      [
        "Img-2",
        [
          {"x": 166},
          {"y": 48},
          {"width": 158},
          {"height": 158},
          {"x_final": 324},
          {"y_final": 206},
          {"fill": null},
          {"stroke": null},
          {"col": 1}
        ]
      ],
      [
        "Empty",
        [
          {"x": 324},
```

```
          {"y": 104},
          {"width": 543},
          {"x_final": 867},
          {"y_final": 104},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 3}
        ]
      ],
      [
        "Btn",
        [
          {"x": 867},
          {"y": 104},
          {"width": 261},
          {"height": 64},
          {"x_final": 1128},
          {"y_final": 168},
          {"fill": "#4477c1"},
          {"stroke": "#4477c1"},
          {"col": 2}
        ]
      ],
      [
        "Btn-3",
        [
          {"x": 1169},
          {"y": 104},
          {"width": 261},
          {"height": 64},
          {"x_final": 1430},
          {"y_final": 168},
          {"fill": "#4477c1"},
          {"stroke": "#4477c1"},
          {"col": 2}
        ]
      ],
      [
        "Btn-5",
        [
          {"x": 1471},
          {"y": 104},
          {"width": 261},
          {"height": 64},
          {"x_final": 1732},
          {"y_final": 168},
          {"fill": "#4477c1"},
```

```
          {"stroke": "#4477c1"},
          {"col": 2}
        ]
      ]
    ]
  },
  "2": {
    "1": [
      [
        "Empty",
        [
          {"x": -1},
          {"y": 336},
          {"width": 999},
          {"x_final": 998},
          {"y_final": 336},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 6}
        ]
      ],
      [
        "Text-12",
        [
          {"x": 998},
          {"y": 336},
          {"width": 0},
          {"height": 40},
          {"x_final": 998},
          {"y_final": 376},
          {"fill": "#dd983c"},
          {"stroke": null},
          {"col": 12}
        ]
      ]
    ]
  },
  "3": {
    "1": [
      [
        "Empty",
        [
          {"x": -1},
          {"y": 483},
          {"width": 159},
          {"x_final": 158},
          {"y_final": 483},
```

```
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 1}
        ]
      ],
      [
        "Form",
        [
          {"x": 158},
          {"y": 483},
          {"width": 911},
          {"height": 44},
          {"x_final": 1069},
          {"y_final": 927},
          {"fill": "rgba(255,255,255,0)"},
          {"stroke": "#000"},
          {"col": 6}
        ]
      ],
      [
        "Text-10",
        [
          {"x": 322},
          {"y": 567},
          {"width": 0},
          {"height": 30},
          {"x_final": 322},
          {"y_final": 597},
          {"fill": "#dd983c"},
          {"stroke": null},
          {"col": 6}
        ]
      ],
      [
        "Text-9",
        [
          {"x": 365},
          {"y": 621},
          {"width": 0},
          {"height": 30},
          {"x_final": 365},
          {"y_final": 651},
          {"fill": "#dd983c"},
          {"stroke": null},
          {"col": 6}
        ]
      ],
```

```
      [
        "Text-14",
        [
          {"x": 395},
          {"y": 675},
          {"width": 0},
          {"height": 30},
          {"x_final": 395},
          {"y_final": 705},
          {"fill": "#dd983c"},
          {"stroke": null},
          {"col": 6}
        ]
      ],
      [
        "Text-11",
        [
          {"x": 402},
          {"y": 729},
          {"width": 0},
          {"height": 30},
          {"x_final": 402},
          {"y_final": 759},
          {"fill": "#dd983c"},
          {"stroke": null},
          {"col": 6}
        ]
      ],
      [
        "TextBox-2",
        [
          {"x": 454},
          {"y": 538},
          {"width": 471},
          {"height": 35},
          {"x_final": 925},
          {"y_final": 573},
          {"fill": "#fff"},
          {"stroke": "#707070"},
          {"col": 3}
        ]
      ],
      [
        "TextBox",
        [
          {"x": 454},
          {"y": 592},
```

```
          {"width": 471},
          {"height": 35},
          {"x_final": 925},
          {"y_final": 627},
          {"fill": "#fff"},
          {"stroke": "#707070"},
          {"col": 3}
        ]
      ],
      [
        "TextBox-3",
        [
          {"x": 454},
          {"y": 646},
          {"width": 471},
          {"height": 35},
          {"x_final": 925},
          {"y_final": 681},
          {"fill": "#fff"},
          {"stroke": "#707070"},
          {"col": 3}
        ]
      ],
      [
        "TextBox-4",
        [
          {"x": 454},
          {"y": 700},
          {"width": 471},
          {"height": 35},
          {"x_final": 925},
          {"y_final": 735},
          {"fill": "#fff"},
          {"stroke": "#707070"},
          {"col": 3}
        ]
      ],
      [
        "Btn-7",
        [
          {"x": 483},
          {"y": 818},
          {"width": 261},
          {"height": 64},
          {"x_final": 744},
          {"y_final": 882},
          {"fill": "#dd983c"},
```

```
          {"stroke": "#dd983c"},
          {"col": 2}
        ]
      ]
    ],
    "2": [
      [
        "Empty",
        [
          {"x": 744},
          {"y": 416},
          {"width": 378},
          {"x_final": 1122},
          {"y_final": 416},
          {"fill": "null"},
          {"stroke": "null"},
          {"col": 2}
        ]
      ],
      [
        "Img",
        [
          {"x": 1122},
          {"y": 416},
          {"width": 499},
          {"height": 568},
          {"x_final": 1621},
          {"y_final": 984},
          {"fill": null},
          {"stroke": null},
          {"col": 3}
        ]
      ]
    ]
  },
  "4": {
    "1": [
      [
        "Text-3",
        [
          {"x": 168},
          {"y": 1098},
          {"width": 0},
          {"height": 25},
          {"x_final": 168},
          {"y_final": 1123},
          {"fill": null},
```

```
          {"stroke": null},
          {"col": 3}
        ]
      ]
    ],
    "2": [
      [
        "Text",
        [
          {"x": 596},
          {"y": 1098},
          {"width": 0},
          {"height": 25},
          {"x_final": 596},
          {"y_final": 1123},
          {"fill": null},
          {"stroke": null},
          {"col": 3}
        ]
      ]
    ],
    "3": [
      [
        "Text-2",
        [
          {"x": 1122},
          {"y": 1098},
          {"width": 0},
          {"height": 25},
          {"x_final": 1122},
          {"y_final": 1123},
          {"fill": null},
          {"stroke": null},
          {"col": 3}
        ]
      ]
    ],
    "4": [
      [
        "Text-4",
        [
          {"x": 1536},
          {"y": 1098},
          {"width": 0},
          {"height": 25},
          {"x_final": 1536},
          {"y_final": 1123},
```

```
          {"fill": null},
          {"stroke": null},
          {"col": 3}
        ]
      ]
    ]
  },
  "5": {
    "1": [
      [
        "Footer",
        [
          {"x": 0},
          {"y": 1177},
          {"width": 1920},
          {"height": 83},
          {"x_final": 1920},
          {"y_final": 1260},
          {"fill": "#5e93d1"},
          {"stroke": "#5e93d1"},
          {"col": 12}
        ]
      ],
      [
        "Text-8",
        [
          {"x": 800},
          {"y": 1220},
          {"width": 0},
          {"height": 25},
          {"x_final": 800},
          {"y_final": 1245},
          {"fill": "#fff"},
          {"stroke": null},
          {"col": 12}
        ]
      ]
    ]
  }
}
```